Solving Linear Systems on Vector and Shared Memory Computers

Jack J. Dongarra   Iain S. Duff   Danny C. Sorensen   Henk A. Van der Vorst

January 5, 1998

Contents

About the Authors

Jack Dongarra is a computer scientist specializing in numerical algorithms in linear algebra and high-performance computing at Oak Ridge National Laboratory's Mathematical Sciences Section and at the University of Tennessee's Computer Science Department. His current research involves the development, testing, and documentation of high-quality mathematical software and the design of algorithms and techniques for high-performance computer architectures. He was involved in the design and implementation of the packages EISPACK, LINPACK, the Level 2 and 3 BLAS, LAPACK, ScaLAPACK, PVM, and MPI. Other experience includes work as a computer scientist and senior computer scientist in the Mathematics and Computer Science Division at Argonne National Laboratory from 1973 to 1989, as a visiting scientist with the Center for Supercomputing Research and Development at the University of Illinois at Urbana during 1987, as a visiting scientist at IBM's T. J. Watson Research Center in 1981, as a consultant to Los Alamos Scientific Laboratory in 1978, as a research assistant with the University of New Mexico in 1978, and as a visiting scientist at Los Alamos Scientific Laboratory in 1977. Dongarra received a Ph.D. in applied mathematics from the University of New Mexico in 1980, an M.S. in computer science from the Illinois Institute of Technology in 1973, and a B.S. in mathematics from Chicago State University in 1972. He is an editor of several journals in the area of high-performance computing.

Iain S. Duff is currently Group Leader of Numerical Analysis in the Central Computing Department at the Rutherford Appleton Laboratory. He is also the Project Leader for the Parallel Algorithms Group at CERFACS in Toulouse and is a visiting professor at the University of Strathclyde. Duff obtained a first-class honors degree in mathematics and natural philosophy from the University of Glasgow in 1969 and was awarded a D. Phil. in mathematics from Oxford University in 1972. Before April 1990, he was Group Leader of Numerical Analysis at the Harwell Laboratory. He worked in the Computer Science and Systems Division at Harwell from 1975. Before joining Harwell, he was a Harkness Fellow visiting Stony Brook and Stanford and spent two years as a lecturer in computing science at the University of Newcastle. He has had several extended visits to Argonne National Laboratory, the Australian National University, the University of Colorado at Boulder, Stanford University, and the University of Umeå. He is a fellow of the Institute of

Mathematics and Its Applications, a member of SIAM, an editor and associate editor of several journals, and author of over 90 papers.

Danny C. Sorensen is a professor in the Mathematical Sciences Department of Rice University. His research interests are in numerical analysis and parallel computation, with specialties in numerical linear algebra, use of advanced-computer architectures, programming methodology and tools for parallel computers, and numerical methods for nonlinear optimization. Sorensen was a computer scientist and senior computer scientist in the Mathematics and Computer Science Division at Argonne National Laboratory from 1980 to 1989. He has also been a visiting professor at the Department of Operations Research at Stanford University and the Department of Mathematics, University of California at San Diego, and a visiting scientist with the Center for Supercomputing Research and Development at the University of Illinois at Urbana.

Henk A. van der Vorst is a professor in numerical analysis in the Mathematical Department of Utrecht University in the Netherlands. His current research interests include iterative solvers for linear systems, large sparse eigenproblems, and overdetermined systems and the design of algorithms for parallel and vector computers. He has used and tested supercomputers—including the Japanese supercomputers—for over a decade. Van der Vorst received a Ph.D. in applied mathematics from the University of Utrecht in 1982 for a thesis on the effect of preconditioning. Together with Meijerink, he proposed in the mid-1970s the so-called ICCG method, which is now widely used for solving certain types of discretized partial differential equations. Van der Vorst was a senior consultant at the Academic Computing Centre Utrecht for more than 12 years, until fall 1984. From 1984 to 1990 he was a professor in numerical linear algebra and supercomputing in the Department of Technical Mathematics and Computer Science of Delft University.

Preface

The purpose of this book is to unify and document in one place many of the techniques and much of the current understanding about solving systems of linear equations on vector and shared-memory parallel computers. This book is not a textbook, but it is meant to provide a fast entrance to the world of vector and parallel processing for these linear algebra applications. We intend this book to be used by three groups of readers: graduate students, researchers working in computational science, and numerical analysts. As such, we hope this book can serve both as a reference and as a supplement to a teaching text on aspects of scientific computation.

The book is divided into four sections: (1) introduction to terms and concepts, including an overview of the state of the art for high-performance computers and a discussion of performance evaluation (Chapters 1-4); (2) direct solution of dense matrix problems (Chapter 5); (3) direct solution of sparse matrix problems (Chapter 6); and (4) iterative solution of sparse matrix problems (Chapter 7). Any book that attempts to cover these topics must necessarily be somewhat out of date before it appears, because the area is in a state of flux. We have purposely avoided highly detailed descriptions of popular machines and have tried instead to focus on concepts as much as possible; nevertheless, to make the description more concrete, we do point to specific computers.

Rather than include a floppy disk containing the software described in the book, we have included a pointer to *netlib*. The problem with floppies in books is that they are never around when one needs them, and the software may undergo changes to correct problems or incorporate new ideas. The software included in *netlib* is in the public domain and can be used freely. With *netlib* we hope to have up-to-date software available at all times. A directory in *netlib* called *ddsv* contains the software, and Appendix A of this book discusses what is available and how to make a request from *netlib*.

This book only touches on topics relating to massively parallel SIMD computers and distributed-memory machines, partly because our experience lies in shared-memory architectures and partly because the areas of massively parallel and distributed-memory computing are still rapidly changing. We express appreciation to all those who helped in the preparation of this work, in particular to Gail

Pieper for her tireless efforts in proofreading drafts and improving the quality of the presentation; Ed Anderson, Mary Drake, Jeremy Du Croz, Al Geist, Peter Mayes, Esmond Ng, Antoine Petitet, Giuseppe Radicati, and Charlie Van Loan for their help in proofreading and their many suggestions to improve the readability; and Reed Wade for his assistance in preparing the figures. Much of the dense linear algebra parts would not be possible without the efforts and support of the developers of LAPACK and ScaLAPACK.

Introduction

The recent availability of advanced-architecture computers has had a very significant impact on all spheres of scientific computation including algorithm research and software development in numerical linear algebra. This book discusses some of the major elements of these new computers and indicates some recent developments in sparse and full linear algebra that are designed to exploit these elements.

The two main novel aspects of these advanced computers are the use of vectorization and parallelism, although how these are accommodated varies greatly between architectures. The first commercially available vector machine to have a significant impact on scientific computing was the CRAY-1, the first machine being delivered to Los Alamos in 1976. Thus, the use of vectorization is by now quite mature, and a good understanding of this architectural feature and general guidelines for its exploitation are now well established. However, the first commercially viable parallel machine was the Alliant in 1985, and more highly parallel machines did not appear on the marketplace until 1988. Thus, there remains a relative lack of definition and maturity in this area, although some guidelines and standards on the exploitation of parallelism are beginning to emerge.

We are algebraists rather than computer scientists; as such, one of our intentions in writing this book is to provide the computing infrastructure and necessary definitions to guide the computational scientist and, at the very least, to equip him or her with enough understanding to be able to read computer documentation and appreciate the influence of some of the major aspects of novel computer design. The majority of this basic material is covered in Chapter 1, although we address further aspects related to implementation and performance in Chapters 3 and 4. In such a volatile marketplace it is not sensible to concentrate too heavily on any specific architecture or any particular manufacturer, but we feel it is useful to illustrate our general remarks by reference to some currently existing machines. This we do in Chapter 2 and Appendix C, as well as in Chapter 4 where we present some performance profiles for current machines.

Linear algebra—in particular, the solution of linear systems of equations—lies at the heart of most calculations in scientific computing. We thus concentrate on this area in this book, examining algorithms and software for dense coefficient matrices in Chapter 5 and for sparse systems in

Chapters 6 and 7, where we discuss direct and iterative methods of solution, respectively. Although we have concentrated on this aspect of linear algebra, many of our observations and techniques extend to other areas—for example, the eigenproblem or the solution of least-squares problems, of which brief mention is made in Section 5.5.

Within scientific computation, parallelism can be exploited at several levels. At the highest level a problem may be subdivided even before its discretization into a linear (or nonlinear) system. This technique, typified by domain decomposition, usually results in large parallel tasks ideal for mapping onto a distributed-memory architecture. In keeping with our decision to minimize machine description, we refer only briefly to this form of algorithmic parallelism in the following, concentrating instead on the solution of the discretized subproblems. Even here, more than one level of parallelism can exist—for example, if the discretized problem is sparse. We discuss sparsity exploitation in Chapters 6 and 7.

Our main algorithmic paradigm for exploiting both vectorization and parallelism in the sparse and the full case is the use of block algorithms, particularly in conjunction with highly tuned kernels for effecting matrix-vector and matrix-matrix operations. We discuss the design of these building blocks in Section 5.1 and their use in the solution of dense equations in the rest of Chapter 5. We discuss their use in the solution of sparse systems in Chapter 6, particularly Sections 6.4 and 6.5.

As we said in the Preface, this book is intended to serve as a reference and as a supplementary teaching text for graduate students, researchers working in computational science, and numerical analysts. At the very least, the book should provide background, definitions, and basic techniques so that researchers can understand and exploit the new generation of computers with greater facility and efficiency.

Chapter 1

High Performance Computing

In this chapter we review some of the basic features of traditional and advanced computers. The review is not intended to be a complete discussion of the architecture of any particular machine or a detailed analysis of computer architectures. Rather, our focus is on certain features that are especially relevant to the implementation of linear algebra algorithms.

1.1 Trends in Computer Design

In the past decade, the world has experienced one of the most exciting periods in computer development. Computer performance improvements have been dramatic - a trend that promises to continue for the next several years. One reason for the improved performance is the rapid advance in microprocessor technology. Microprocessors have become smaller, denser, and more powerful. Indeed, if cars had made equal progress, you could buy a car for a few dollars, drive it across the country in a few minutes, and "park" the car in your pocket! The result is that microprocessor-based supercomputing is rapidly becoming the technology of preference in attacking some of the most important problems of science and engineering. To exploit microprocessor technology, vendors have developed *highly parallel computers.*

Highly parallel systems offer the enormous computational power needed for solving some of our most challenging computational problems such as simulating the climate. Unfortunately, software development has not kept pace with hardware advances. New programming paradigms, languages, scheduling and partitioning techniques, and algorithms are needed to fully exploit the power of these highly parallel machines.

A major new trend for scientific problem solving is *distributed computing.* In distributed computing, computers connected by a network are used collectively to solve a single large problem. Many scientists are discovering that their computational requirements are best served not by a single, monolithic computer but by a variety of distributed computing resources, linked by high-speed networks.

There is little to suggest that these trends will not continue to grow. The computer architectures will by necessity have to change from our traditional sequential execution to parallel execution computers.

By parallel computing we mean a set of processes that are able to work together to solve a computational problem. There are are a few things that are worthwhile to point out. First, the use of parallel processing is now everywhere, from the personal computer to the fastest computers available parallel processing techniques are being employed. Second, parallel processing does not necessarily imply high-performance computing.

In this Chapter we explore some of the issues involved in the use of high-performance computing. These issues are at the the heart of effective use of the fastest supercomputers.

1.2 Traditional Computers and Their Limitations

The traditional, or conventional, approach to computer design involves a single instruction stream. Instructions are processed sequentially and result in the movement of data from memory to functional unit and back to memory. Specifically,

- a scalar instruction is fetched and decoded,
- addresses of the data operands to be used are calculated,
- operands are fetched from memory,
- the calculation is performed in the functional unit, and
- the resultant operand is written back to memory.

As demands for faster performance increased, modifications were made to improve the design of computers. It became evident, however, that a number of factors were limiting potential speed: the switching speed of the devices (the time taken for an electronic circuit to react to a signal), packaging and interconnection delays, and compromises in the design to account for realistic tolerances of parameters in the timing of individual components. Even if a dramatic improvement could be made in any of these areas, one factor still limits performance: *the speed of light.* Today's supercomputers have a cycle time on the order of nanoseconds. The CRAY T90, for example, has a cycle time of

2.2 nanosecond (nsec.). One nanosecond translates into the time it takes light to move about a foot (in practice, the speed of pulses through the wiring of a computer ranges from 0.3 to 0.9 foot per nanosecond). Faced by this fundamental limitation, computer designers have begun moving in the direction of parallelism.

1.3 Parallelism within a Single Processor

Parallelism is not a new concept . In fact, Hockney and Jesshope point out that Babbage's analytical engine in the 1840s had aspects of parallel processing [198].

1.3.1 Multiple Functional Units

Early computers had three basic components: the main memory, the central processing unit (CPU), and the I/O subsystem. The CPU consisted of a set of registers, the program counter, and one arithmetic and logical unit (ALU) , where the operations were performed one function at a time. One of the first approaches to exploiting parallelism involved splitting up the functions of the ALU—for example, into a floating-point addition unit and a floating-point multiplication unit—and having the units operate in parallel.

In order to take advantage of the multiple functional units, the software (e.g., the compiler) had to be able to schedule operations across the multiple functional units to keep the hardware busy. Also, the overhead in starting operations on the multiple units had to be small relative to the time spent performing the operations. Once computer designers had added multiple functional units, they turned to investigating better ways to interconnect the functional units, in order to simplify the flow of data and to speed up the processing of data.

1.3.2 Pipelining

Pipelining is the name given to the segmentation of a functional unit into different parts, each of which is responsible for partial decoding/interpretation and execution of an operation.

The concept of pipelining is similar to that of an assembly line process in an industrial plant. Pipelining is achieved by dividing a task into a sequence of smaller tasks, each of which is executed on a piece of hardware that operates concurrently with the other stages of the pipeline (see Figures 1.1, 1.2, and 1.3). Successive tasks are streamed into the pipe and get executed in an overlapped fashion with the other subtasks. Each of the steps is performed during a clock period of the machine. That is, each suboperation is started at the beginning of the cycle and completed at the end of the cycle. The technique is as old as computers, with each generation using ever more sophisticated

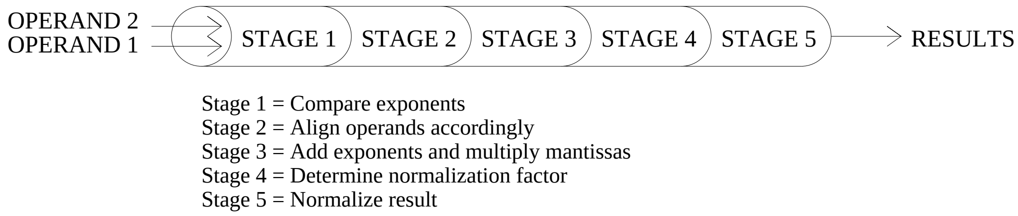

Figure 1.1: A simplistic pipeline for floating-point multiplication

variations. An excellent survey of pipelining techniques and their history can be found in Kogge [220].

Pipelining was used by a number of machines in the 1960s, including the CDC 6600, the CDC 7600, and the IBM System 360/195. Later, Control Data Corporation introduced the STAR 100 (subsequently the CYBER 200 series), which also used pipelining to gain a speedup in instruction execution. In the execution of instructions on machines today, many of the operations are pipelined—including instruction fetch, decode, operand fetch, execution, and store.

The execution of a pipelined instruction incurs an overhead for filling the pipeline. Once the pipeline is filled, a result appears every clock cycle. The overhead or *startup* for such an operation depends on the number of stages or segments in the pipeline.

1.3.3 Overlapping

Some architectures allow for the *overlap* of operations if the two operations can be executed by independent functional units. *Overlap* is similar but not identical to pipelining. Both employ the idea of subfunction partitioning, but in slightly different contexts. Pipelining occurs when *all* of the following are true:

- Each evaluation of the basic function (for example, floating-point addition and multiplication) is independent of the previous one.
- Each evaluation requires the same sequence of stages.
- The stages are closely related.
- The times to compute different stages are approximately equal.

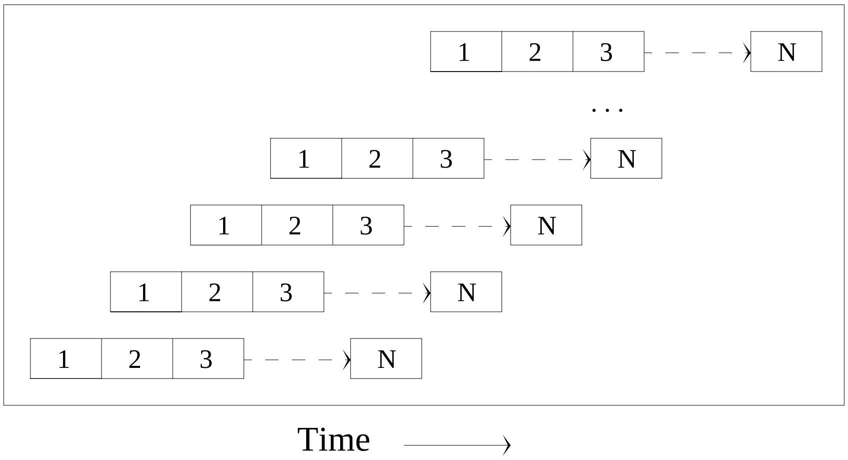

Figure 1.2: Pipelined execution of an N-step process

Overlap, on the other hand, is typically used when *one* of the following occurs:

- There may be some dependencies between evaluations.
- Each evaluation may require a different sequence of stages.
- The stages are relatively distinct in their purpose.
- The time per stage is not necessarily constant but is a function of both the stage and the data passing through it.

1.3.4 RISC

In the late 1970s computer architects turned toward a simpler design in computer systems to gain performance. A RISC (reduced instruction set computer) architecture is one with a very fast clock cycle that can execute instructions at the rate of one per cycle. RISC machines are often associated with pipeline implementations since pipeline techniques are natural for achieving the goal of one instruction executed per machine cycle. The key aspects of a RISC design are as follows:

- single-cycle execution (for most instructions, usually not floating-point instructions),

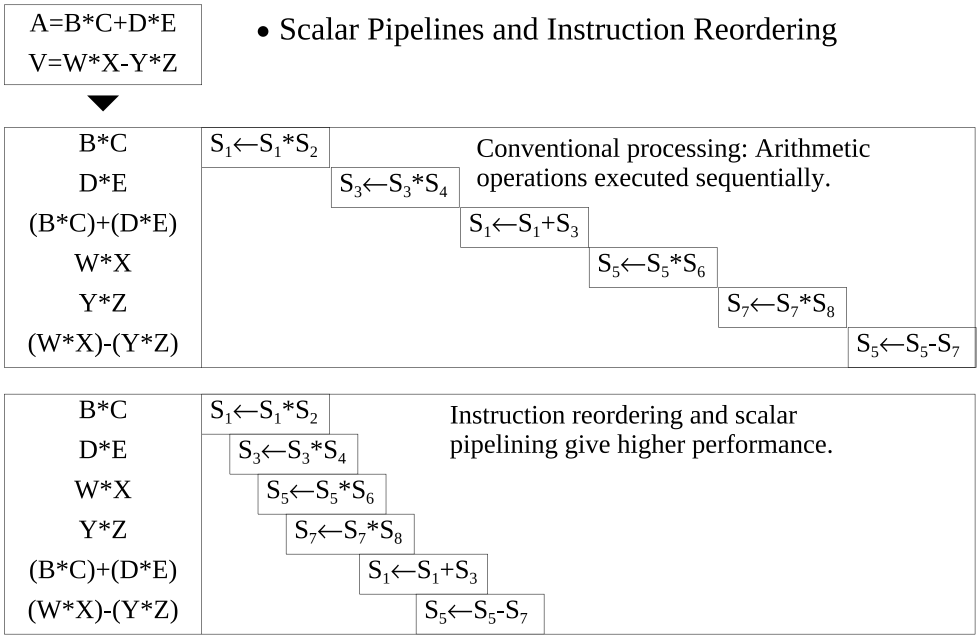

Figure 1.3: Scalar pipelines

- simple load/store interface to memory,
- register-based execution,
- simple fixed-format and fixed-length instructions,
- simple addressing modes,
- large register set or register windows, and
- delayed branch instructions.

The design philosophy for RISC is simplicity and efficiency. That is, RISC makes efficient use of the hardware through a simplification of the processor's instruction set and execution of the instruction set.

It is interesting to note that RISC machines have been around before the 1970's, in fact the CDC 6600 built in 1964 by Seymour Cray was a RISC design. At the time, RISC did not become widely recognized and adopted as the architecture of choice for a number of reasons. There where a number of technical barriers that needed to be overcome in order to permit RISC architecture to become feasible.

- Improvements to cache memory speed for instruction and data fetching
- Primary memory size increases
- Primary memory cost decreases
- Advanced pipelining of instruction execution
- Compiler optimization techniques

It can be argued that as super-scalar processors, the RISC processors are close to matching the performance level of vector processors with matched cycle times. Moreover, they exceed the performance of those vector processors on non-vector problems.

1.3.5 VLIW

Very long instruction word (VLIW) architectures are reduced instruction set computers with a large number of parallel, pipelined functional units but only a single thread of control. That is, parallel (side-by-side) execution of instructions. So that each VLIW instruction is actually two or more RISC instructions. Each clock cycle a VLIW instruction is fetched and executed. VLIWs provide a fine-grained parallelism. In VLIWs, every resource is completely and independently controlled by the compiler. There is a single thread of control, a single instruction stream, that initiates each fine-grained operation; any such operations can be initiated each cycle. All communications are completely choreographed by the compiler and are under explicit control of the compiled program. The source, destination, resources, and time of a data transfer are all known by the compiler. There is no sense of packets containing destination addresses or of hardware scheduling of transfers.

Such fine-grained control of a highly parallel machine requires very large instructions, hence the name "very long instruction word" architecture . These machines offer the promise of an immediate speedup for general-purpose scientific computing. But unlike previous machines, VLIW machines are difficult to program in machine language; only a compiler for a high-level language, like Fortran, makes these machines feasible.

1.3.6 Vector Instructions

One of the most obvious concepts for achieving high performance is the use of *vector instructions.* Vector instructions specify a particular operation that is to be carried out on a selected set of operands (called vectors). In this context a vector is an ordered list of scalar values and is inherently one dimensional. When the control unit issues a vector instruction, the first element(s) of the vector(s) is (are) sent to the appropriate pipe by way of a data path. After some number of clock cycles (usually one), the second element(s) of the vector(s) is (are) sent to the same pipeline using the same data path. This process continues until all the operands have been transmitted.

Vector computers rely on several strategies to speed up their execution. One strategy is the inclusion of vector instructions in the instruction set. The issue of a single vector instruction results in the execution of all the component-wise operations that make up the total vector operation. Thus, in addition to the operation to be performed, a vector instruction specifies the starting addresses of the two operand vectors and the result vector and their common length.

The time to execute a vector instruction is given by

`startup_time + vector_length`

The time to complete a pipelined operation is a function of the startup time and the length of the vector. The time to execute two overlapped vector operations is given by

`startup_time_2 + vector_length`.

The `startup_time_2` is equal to the maximum startup time of the two operations plus one cycle, assuming the independent operation can be initiated immediately after the first has started.

1.3.7 Chaining

Pipelined processes can often be combined to speed up a computation. *Chaining*, or linking, refers to the process of taking the result of one pipelined process and directing it as input into the next pipelined process, without waiting for the entire first operation to complete . The operations that chain together vary from machine to machine; a common implementation is to chain multiplication and addition operations (see Figure 1.4).

If the instructions use separate functional units (such as the addition and multiplication units in Figure 1.4), the hardware will start the second vector operation while the first result from the first operation is just leaving its functional unit. A copy of the result is forwarded directly to the second functional unit, and the first execution of the second vector is started. The net effect is that the execution of both vector operations takes only the second functional unit startup time longer than the first vector operation.

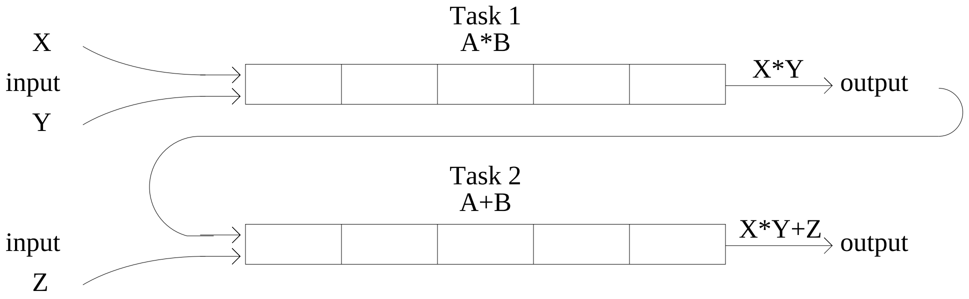

Figure 1.4: Chaining multiplication and addition

1.3.8 Memory-to-Memory and Register-to-Register Organizations

In some computers, the floating-point functional units may communicate directly with main memory to receive and transfer data. In this case, the operands flow from memory into the functional units and the results flow back to memory as one user-issued operation. This architecture is often referred to as a *memory-to-memory* organization.

Memory-to-memory organization allows source operands and intermediate and final results to be retrieved directly between the pipelines and the main memory. Information on the starting point of the vector, the distance between vector elements (increment), and the vector length must be specified in order to transfer streams of data between the main memory and the pipelined functional units. The CYBER 205 and ETA-10 were examples of this organization.

In contrast to memory-to-memory organization, the floating-point units may have a path to a set of *vector registers,* each register holding several entries. The functional unit then interfaces with the vector registers, and the vector registers in turn have a path to the memory system. The majority of vector machines use this organization, which is often referred to as a *register-to-register* organization.

The presence of vector registers may reduce the traffic flow to and from main memory (if data can be reused), and their fast access times assist in reducing the startup time, by reducing the memory latency of data arriving from main memory. In some sense, then, the vector registers are a fast intermediate memory.

1.3.9 Register Set

Scalar registers are a form of very-high-speed memory used to hold the most heavily referenced data at any point in the program's execution. These registers can send data to the functional units in one clock cycle; this process is typically an order of magnitude faster than the memory speed. As one might expect, memory with this speed is expensive; and as a result, the amount of high-speed memory supplied as registers is limited.

In general, the speed of main memory is insufficient to meet the requirements of the functional units. To ensure that data can be delivered fast enough to the functional units and that the functional units can get rid of their output fast enough, most manufacturers have introduced a special form of very-high-speed memory, the so-called vector register. A *vector register* is a fixed set of memory locations in a special memory under user control.

For example, on Cray computers, a vector register consists of 64 elements. The rule is for a functional unit to accept complete vector registers as operands. The hardware makes it possible for the elements of the vector register to be fed one by one to the functional unit at a rate of exactly one per clock cycle per register. Also, the register accepts 64 successive output elements of the functional unit at a rate of one element per clock cycle. The locations within a vector register cannot be accessed individually.

Loading, storing, and manipulating the contents of a vector register are done under control of special vector instructions. These vector instructions are issued automatically by the compiler, when applicable. Since a functional unit typically involves three vector operands, two for input and one for output, there are more than three vector registers. For example, the Cray processors typically have 8 or 16 vector registers. The seemingly extra registers are used by the system in order to keep intermediate results as operands for further instructions. The programmer can help the compiler exploit the contents of these registers by making it possible to combine suitable statements and expressions.

The best source of information about techniques for exploiting the contents of vector and scalar registers is the appropriate computer manual. We also note that the increasing sophistication of compilers is making it less necessary to "help" compilers recognize potentially optimizable constructs in a program.

1.3.10 Stripmining

Vectors too large to fit into vector registers require software fragmentation, or *stripmining* . This is automatically controlled by the compiler. The compiler inserts an outer loop to break the operation into pieces that can be accommodated by the vector register. After the first strip or segment is complete, the next one is started. Because overhead is associated with each vector operation, stripmining incurs a startup overhead for each piece.

1.3.11 Reconfigurable Vector Registers

A special feature of some vector processors is a dynamically reconfigurable vector register set. The length and number of vector registers required usually vary between programs and even within different parts of the same program. To make the best use of the total vector register capacity, the registers may be concatenated into different lengths under software control.

For example, in the Fujitsu VPP-300 computer, for each processor there are 8192 elements in the vector registers. The registers can be configured into any of the following:

```
# registers x length in words

        32 x 256
        64 x 128
       128 x  64
       256 x  32
       512 x  16
      1024 x   8
```

The length of the vector register is specified in a special register and is set by an instruction generated by the compiler. To best utilize the dynamically reconfigurable vector registers, the compiler must know the frequently used vector lengths for each program. If the vector length is set too short, load-store instructions will have to be issued more frequently. If it is set unnecessarily long, the number of vector registers will decrease, resulting in frequent saves and restores of the vector registers. In general, the compiler puts a higher priority on the number of vectors than on the vector length.

1.3.12 Memory Organization

The flow of data from the memory to the computational units is the most critical part of a computer design. The object is to keep the functional units running at their peak capacity. Through the use of a memory hierarchy system (see Table 1.1), high performance can be achieved by using *locality of reference* within a program. (By locality of reference we mean that references to data are contained within a small range of addresses and that the program exhibits reuse of data.) In this section we discuss the various levels of memory; in Section 1.4, we give details about memory management.

At the top of the hierarchy are the *registers* of the central processing unit.

The registers in many computers communicate directly with a small, very fast *cache* memory of perhaps several hundred to a thousand words. Cache is a form of storage that is automatically filled and emptied according to a fixed scheme defined by the hardware system.

Main memory is the memory structure most visible to the programmer. Since random access to memory is relatively slow, requiring the passage of several clock cycles between successive memory references, main memory is usually divided into *banks* (see Figure 1.5). In general, the smaller the memory size, the fewer the number of banks.

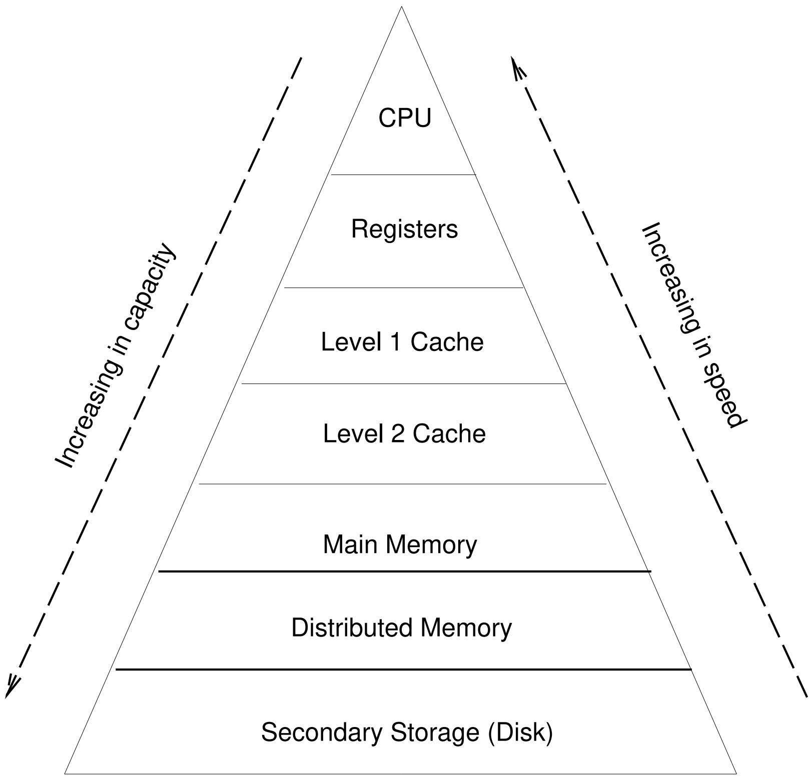

Table 1.1: A typical memory hierarchy

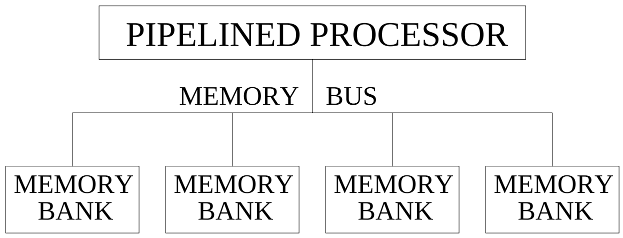

Figure 1.5: Multiple memory banks in a pipelined computer system

Associated with memory banks is the memory *bank cycle time*, the number of clock periods a given bank must wait before the next access can be made to data in the bank. After an access and during the memory bank cycle time, references to data in the bank are suspended until the bank cycle time has elapsed. The CRAY T932 with cm03 memory, for example, has 1024 MWords divided into 1024 banks, with a bank cycle time of 4 processor cycles (each processor cycle is 2.2 nsec). (Note that the CRAY T932 with cm02 memory has 512 banks with a bank cycle time of 7 processor cycles.) This is due to the physical constraint in the number of wires that connect memory to the control portion of the machine. Basically, many locations in memory share common connections.

1.4 Data Organization

Fast memory costs more than slow. Moreover, the larger the memory, the longer it takes to access items. The objective, then, is to match at a reasonable cost the processor speed with the rate of information transfer or the *memory bandwidth* at the highest level.

1.4.1 Main Memory

In most systems the bandwidth from a single memory module is not sufficient to match the processor speed. Increasing the computational power without a corresponding increase in the memory bandwidth of data to and from memory can create a serious bottleneck. One technique used to address this problem is called *interleaving*, or banked memory. With interleaving, several modules can be referenced simultaneously to yield a higher effective rate of access. Specifically, the modules are arranged so that N sequential memory addresses fall in N distinct memory modules. By keeping all N modules busy accessing data, effective bandwidths up to N times that of a single module are possible.

In vector processors, the banks operate with their access cycles out of phase with one another. The reason for such an arrangement is that random access memory is slow relative to the processor, requiring the passage of several clock periods between successive memory references. In order to keep the vector operations streaming at a rate of one word per clock period to feed the pipeline, vectors are stored with consecutive operands in different banks. The phase shift that opens successively referenced banks is equal to one processor clock cycle.

A *bank conflict* occurs if two memory operations are attempted in the same bank within the bank cycle time. For example, if memory has 16 banks and each bank requires 4 cycles before it is ready for another request (a bank cycle time of 4), then for a vector data transfer, a bank conflict will occur if any of 4 consecutive addresses differs by a multiple of 16. In other words, bank conflicts can occur only if the memory address increment is a multiple of 8. If the increment is a multiple of 16, every transferred word involves the same memory bank, so the transfer rate is one-fourth of the maximum transfer rate. If the increment is a multiple of 8 but not a multiple of 16, vector memory transfers occur at one-half the maximum transfer rate, since alternate words use the same bank.

	Banks							
	1	2	3	4	5	6	7	8
L	0	1	2	3	4	5	6	7
o	8	9	10	11	12	13	14	15
c	16	17	18	19	20	21	22	23
a	24	25	26	27	28	29	30	31
t	32	33	34	35	36	37	38	39
i	.	.	.	.	.	.	.	.
o	.	.	.	.	.	.	.	.
n	.	.	.	.	.	.	.	.
s	.	.	.	.	.	.	.	.

Sequential Access with Bank Cycle Time of 4

bank 1	0								8											
bank 2		1								9										
bank 3			2								10									
bank 4				3								11								
bank 5					4								12							
bank 6						5								13						
bank 7							6								14					
bank 8								7								15				
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

clock cycle

Every Fourth Element Access with Bank Cycle Time of 4

bank 1	0				8				16				24							
bank 2																				
bank 3																				
bank 4																				
bank 5		4				12				20				28						
bank 6																				
bank 7																				
bank 8																				
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

clock cycle

Memory bank conflicts cannot occur when processing sequential components of a one-dimensional array or a column of a two-dimensional array in vector mode. However, if equally spaced elements of a one-dimensional array are being processed, bank conflicts are likely to occur if the spacing is a multiple of 8. The effect of such bank conflicts on processing time can be quite pronounced.

A basic attribute used to measure the effectiveness of a memory configuration is the memory bandwidth. This is the maximum number of words that can be accessed per second. The primary factors affecting the bandwidth are the memory module characteristics and the processor architecture.

The processor architecture may also be arranged to ensure a high degree of memory access. Some designs have more than one path to memory. For example, the NEC SX series has multiple paths to fetch one logical vector from memory. In other cases, paths are used for distinct vectors. For example, on the CRAY T90 the central memory bandwidth of one stream from memory is two words per cycle of 2.2 nanoseconds, or 909 million words per second. The T90 allows for three streams from memory (two for loads and one for stores) for each processor or 2.7 billion words per second per processor (there can be a maximum of 32 processors on the T90, giving a total of 87 gigawords per second (698 GB/sec) as the peak theoretical memory bandwidth of the system). The memory system is divided into 1024 banks and can sustain this rate if there are no bank conflicts.

1.4.2 Cache

Cache memories are high-speed buffers inserted between the processors and main memory to capture those portions of the contents of main memory currently in use. Since cache memories are typically five to ten times faster than main memory, they can reduce the effective memory access time if carefully designed and implemented.

The idea of introducing a high-speed buffer memory between the slow main memory and the arithmetic registers goes back to the ATLAS computer in 1956 [198]. The technique was adopted by IBM for both the System 360 and the System 370 computers. In the IBM System 360 Model 85, for example, the cache (32,768 bytes of 162-nsec semiconductor memory) held the most recently used data in blocks of 64 bytes. If the data required by an instruction was not in the cache, the block containing it was obtained from the slower main memory (4 Mbytes of 756-nsec core storage, divided into 16 different banks) and replaced the least frequently used block in the cache.

Since then, cache memories have been present in most computers from minicomputers to large-scale mainframes. Such a feature is particularly advantageous when memory references concentrate in limited regions of the address space. In such cases, most references will be to data in the cache memory, and the overall memory performance will be effectively that of the faster cache memory.

Cache systems resemble paged systems (see Section 1.5) in basic concept. Their implementation, however, gives rise to great differences in performance. In a paged system, data is retrieved and mapped from disk to main memory by the operating system. In a cache system, data is retrieved and mapped from main memory to a high-speed buffer by the hardware.

Specifically, the process works as follows. When a processor makes a memory request, it generates the address of the desired word and searches the cache for the reference. If the item is found in cache, a *hit* occurs, and a copy is sent to the processor, without a request being made of the main memory (thus taking less time). If the item is not found in cache, a *miss* (or *cache fault*) is generated. The request must then be passed on to the main memory system. When the item is returned to the processor, a copy is stored in the cache, where room must be found for the item. Obviously, cache misses can be very costly in terms of speed.

The speed consideration also dictates that a block of cache, often referred to as a *cache line*, be only a few words long. Cache lines are loaded on demand, and writes (stores or write backs) are usually performed in one of two ways. A block is replaced and written back to memory when a block is about to be discarded or overwritten in cache (referred to as *write back*); or every time a write occurs, the information is written in cache as well as written back in memory (referred to as *write through*).

Cache organizations differ primarily in the way main memory is mapped into a line in cache. In one organization scheme, called *direct mapped cache*, each cache line is mapped into a preassigned location in the cache. For example, for a cache with k blocks, each i, $i+k$, $i+2*k$, etc., memory locations will be mapped into block i of the cache. This organization is simple; however, the efficiency of a program may suffer badly from poor placement of data.

In another organization, called *set associative cache*, the cache is partitioned into distinct sets of lines, each set containing a small, fixed number of lines. Each address of main memory is mapped into a particular set. With this scheme the entire cache need not be searched during a reference, only the set to which the address is mapped. When an item that is not in cache is referenced, it is mapped into the first free location in a designated set. If no free location is found, then the least recently used location is overwritten.

1.4.3 Local Memory

On the CRAY-2, there was a "user-managed" cache called *local memory*. Local memory in this case is not a required interface between main memory and registers, as is the cache just described. Rather, it is an auxiliary storage place for vectors and scalars that may be required more than once and cannot be accommodated in the register set. Since the local memory is under user (software) control, it is technically incorrect to call it a cache.

1.5 Memory Management

In many computer systems, large programs often cannot fit into main memory for execution. Even if there is enough main memory for a program, the main memory may be shared between a number of users, causing any one program to occupy only a fraction of the memory, which may not be sufficient for the program to execute. The usual solution is to introduce management schemes that intelligently allocate portions of memory to users as necessary for the efficient running of their programs.

In earlier computers, when the entire program could not all fit into main memory at one time, a technique called *overlays* was used. Portions of the program were brought into memory when needed, overlaying those that were no longer needed. It was the user's responsibility to manage the overlaying of a program.

More common today is the use of *virtual memory.* Virtual memory gives programmers the illusion that there is a very large address space (memory) at their disposal. In the virtual memory concept, the data are stored on *pages* . A page contains a fixed amount of data. For example, in the CYBER 205, data can be stored on "large pages," each of which has a capacity of 65,536 words of 64 bits. The machine instructions refer to virtual addresses of operands, where a virtual address consists of the page number of the page on which the operand is located and the address of the operand on the page. For each program, the operating system keeps a page table which shows where all the pages are physically located (either in main memory or in secondary memory).

When an operand is required that is not available on the pages in main memory, the complete page containing the operand is transported from secondary memory to main memory and thereby overwrites the space of some other page. Usually it overwrites the page that has been least recently used. If data on this page has been changed during its stay in main memory, it is first written back to secondary memory before being overwritten. Transport of a (large) page is called a (large) *page fault.* In organizing an algorithm, one should keep in mind that a page fault takes some I/O time. A good strategy is to carry out as many operations as possible on a block of data, where the block size is chosen such that it fits on the number of pages that one has available during execution.

We illustrate the effect of large page faults with an example given by Winter [327]. This example is for the CYBER 205, but similar effects take place on all virtual memory machines. On the CYBER 205, a large page fault takes about 0.5 second of I/O time. Thus, a large number of large page faults can lead to a large I/O time. Though the virtual memory management is done by the operating system and the user has no information about the pages that are actually in main memory, he can influence the number of large faults.

The example itself concerns the dense matrix-matrix multiplication $C = AB + C$. The matrices are square and of order 1024. Since a large page contains 65,536 elements, it follows that 64 columns of a matrix can be stored on 1 large page and that all three matrices can be stored on 48 large pages in total (about 3 million words). Let us assume that the available main memory space can contain 16 large pages (about 1 million words). This implies that if we access successive columns of a matrix, we have a large page fault after each 64 columns. We now consider three different ways to compute $C = AB + C$.

(a) Inner-product approach - ijk:

```
      DO 30 I=1,N
         DO 20 J=1,N
            DO 10 K=1,N
               C(I,J)=A(I,K)*B(K,J)+C(I,J)
10          CONTINUE
20       CONTINUE
30    CONTINUE
```

This algorithm is far from optimal for the CYBER 205, for any size of the matrices, since it does not vectorize (the CYBER 205 must have unit increments on the vector in order to vectorize); thus, the CPU time is about 7 minutes. This is a minor problem, however, in comparison with I/O considerations. For each requested row of A, we cause 16 page faults. Since we have to compute about 1 million elements of C, this leads to about 16 million page faults of 0.5 second each. Hence the I/O time is at least 93 days, and in reality the turnaround time will be considerably larger since the machine usually has other work.

(b) Column-wise update of C - jki:

```
      DO 30 J=1,N
         DO 20 K=1,N
            DO 10 I=1,N
               C(I,J)=A(I,K)*B(K,J)+C(I,J)
10          CONTINUE
20       CONTINUE
30    CONTINUE
```

The innermost loop is now a multiple of a vector added to another vector or SAXPY operation, and the two inner loops together represent a matrix-vector product. This approach leads to vector code on the CYBER 205, and hence the CPU time will be in the order of tens of seconds. Again, however, we must consider the I/O time. For each value of J we need access to all the columns of A, which leads to 1024*16 large page faults. Moreover, we need 16 large page faults for B as well as C. The total amount of about 16,000 page faults results in a minimal I/O time of about 2.25 hours. Though this is much better than under (a), an imbalance still exists between CPU and I/O time.

(c) Partitioning the matrices into block-columns - blocked jki:

Now we implicitly partition the matrices in blocks of 256 by 256 each, $nb = 256$. Each block can be placed on precisely 1 large page.

```
      DO 40 J = 1, N, NB
         DO 30 K = 1, N, NB
            DO 20 JJ = J, J+NB-1
               DO 10 KK = K, K+NB-1
                  C(:, JJ) = C(:, JJ) + A(:, KK) * B(KK, JJ)
10             CONTINUE
20          CONTINUE
30       CONTINUE
40    CONTINUE
```

The great gain is made with respect to the I/O time: the scheme leads to paging A 4 times and B and C only once, resulting in only 96 large page faults, and hence takes only a modest 70 seconds of I/O time.

The goal of designing a multilevel memory hierarchy is to achieve a performance close to that of the fastest memory, at a cost per bit close to that of the slowest memory.

1.6 Parallelism through Multiple Pipes or Multiple Processors

Machine architects have a number of further options in designing computers with higher computational speeds. These options involve more parallelism in one way or another. The two most common strategies are to increase the number of functional units (sometimes referred to as *functional pipes*) in the processor or to increase the number of processors in the system. A third strategy is a hybrid approach. NEC has a system that features a four-processor machine with 16 arithmetic pipes in each processor, thus presenting an architecture that utilizes both forms of parallelism.

The multiple-pipe strategy was adopted by CDC in the CYBER 205 and more recently by Cray Research and the three major Japanese manufacturers Fujitsu, Hitachi, and NEC. In a multiple-pipe machine, more than one pipe is available for each arithmetic operation. Thus, one can perform an addition operation, say, on several vectors simultaneously. This may be on independent vectors or on separate parts of the same vectors where the separation had been performed through stripmining (Section 1.3.10). In both cases a high degree of sophistication is required in the Fortran compiler, and there is little (other than the normal vectorization tricks) that the Fortran programmer can do. This architectural feature is often further complicated by multi-function pipes, where the same hardware can perform more than one type of arithmetic operation, commonly both an addition and a multiplication operation.

Though this strategy reduces the CPU time for vector instructions by roughly the number of pipelines involved, it should not be confused with parallelism at the instruction level. The multiple pipelines cannot be controlled separately; they all must share the work coming from one single vector instruction. The larger the number of pipelines, the longer the vector length must be in order to get reasonable computing speeds, since the startup time for a single pipeline is more dominant in the multiple situation (see Section 4.4).

Another strategy—the use of multiple processors—was first adopted by Cray Research with the introduction of the X-MP in 1982. Manufacturers have offered products with multiple processors for years; however, until recently these machines have been used principally to increase throughput via multiple job streams. Today, two approaches to multiprocessing are being actively investigated:

1. SIMD (single instruction stream/multiple data stream) . In this class, multiple processing elements and parallel memory modules are under the supervision of one control unit. All the processing elements receive the same instruction broadcast from the control unit, but operate on different data sets from distinct data streams. SIMD machines (often referred to as *array processors*) permit explicit expression of parallelism in a program. Program segments that cannot be converted into parallel executable form are sent to the processing units and are executed synchronously on data fetched from parallel memory modules under the control of the control unit. A number of SIMD architectures have been marketed over the years; AMT DAP-610, Thinking Machines CM-2, and the MasPar MP-2 are examples. While SIMD architectures work well for some applications, they do not provide high-performance across all applications and are viewed by many as specility hardware.

   Another subclass of the SIMD systems are the vector processors. Vector processors act on arrays of similar data rather than on single data items using specially structured CPUs. When data can be manipulated by these vector units, results can be delivered with a rate of one, two and, in special cases, three per clock cycle. So, vector processors execute on their data in an almost parallel way but only when executing in vector mode. In this case they are several times faster than when executing in conventional scalar mode. For practical purposes vector processors are therefore mostly regarded as SIMD machines.

2. MIMD (multiple instruction stream/multiple data stream). In MIMD machines, the processors are connected to the memory modules by a network. Effective partitioning and assignment are essential for efficient multiprocessing. Most multiprocessor systems can be classified in this category.

1.7 Message Passing

The message passing programming model is based on the assumption that there are a number of processes in the system which have their local memories and can communicate with each other by coordinating memory transfers from one process to another. Thus, this model is of primary importance for distributed memory parallel computers.

On a shared-memory machine, communication between tasks executing on different processors is usually through shared data or by means of *semaphores*, variables used to control access to shared data. On a local-memory machine, however, the communication must be more explicit and is normally done through message passing.

In message passing, data is passed between processors by a simple send-and-receive mechanism. The main variants are whether an acknowledgement is returned to the sender, whether the receiver is waiting for the message, and whether either processor can continue with other computations while the message is being sent. An important feature of message passing is that, on nearly all machines, it is much more costly than floating-point arithmetic.

Message passing performance is usually measured in units of time or bandwidth (bytes per second). In this report, we choose time as the measure of performance for sending a small message. The time for a small, or zero length, message is usually bounded by the speed of the signal through the media (latency) and any software overhead in sending/receiving the message. Small message times are important in synchronization and determining optimal granularity of parallelism. For large messages, bandwidth is the bounded metric, usually approaching the maximum bandwidth of the media. Choosing two numbers to represent the performance of a network can be misleading, so the reader is encouraged to plot communication time as function of message length to compare and understand the behavior of message-passing systems.

Message passing time is usually a linear function of message size for two processors that are directly connected. For more complicated networks, a per-hop delay may increase the message-passing time. Message-passing time, t_n, can be modeled as

$$t_n = \alpha + \beta n + (h-1)\gamma$$

with a start-up time, α, a per-byte cost, β, and a per-hop delay, γ, where n is the number of bytes per message and h the number of hops a message must travel. On most current message-passing multiprocessors the per-hop delay is negligible due to "worm-hole" routing techniques and the small diameter of the communication network [139]. The results reported in this report reflect nearest-neighbor communication. A linear least-squares fit can be used to calculate α and β from experimental data of message-passing times versus message length. The start-up time, α, may be slightly different than the zero-length time, and $1/\beta$ should be asymptotic bandwidth. The message length at which half the maximum bandwidth is achieved, $n_{1/2}$, is another metric of interest and is equal to $(\alpha + (h-1)\gamma)/\beta$ [203]. As with any metric that is a ratio, any notion of "goodness" or "optimality" of $n_{1/2}$ should only be considered in the context of the underlying metrics α, β, γ, and h. For a more complete discussion of these parameters see [202, 200].

There are a number of factors that can affect message-passing performance. The number of times the message has to be copied or touched (e.g., checksums) is probably most influential and obviously a function of message size. The vendor may provide hints as to how to reduce message copies, for example, posting the receive before the send. Second order effects of message size may also affect performance. Message lengths that are powers of two or cache-line size may provide better performance than shorter lengths. Buffer alignment on word, cache-line, or page may also affect performance. For small messages, context-switch times may contribute to delays. Touching all the pages of the buffers can reduce virtual memory effects. For shared media, contention may also affect performance.

There are of course other parameters of a message-passing system that may affect performance for given applications. The aggregate bandwidth of the network, the amount of concurrency, reliability, scalability, and congestion management may be issues.

Table 1.2 shows the measured latency, bandwidth, and $n_{1/2}$ for nearest neighbor communication. The table also includes the peak bandwidth as stated by the vendor. For comparison, typical data rates and latencies are reported for several local area network technologies.

Table 1.2: Multiprocessor Latency and Bandwidth. (sm) refers to a one way communication and (PVM) refers to use of PVM as the message passing layer.

Machine	OS	Latency $n = 0$ (μs)	Bandwidth $n = 10^6$ (MB/s)	$n_{1/2}$ bytes	Theoretical Bandwidth (MB/s)
Cray T3D (sm)	MAX 1.2.0.2	3	128	363	300
Cray T3D (PVM)	MAX 1.2.0.2	21	27	1502	300
Intel Paragon	OSF 1.0.4	29	154	7236	175
IBM SP-2	MPI	35	35	3263	40
SGI	IRIX 6.1	10	64	799	1200
TMC CM-5	CMMD 2.0	95	9	962	10
Ethernet	TCP/IP	500	0.9	450	1.2
FDDI	TCP/IP	900	9.7	8730	12
ATM-100	TCP/IP	900	3.5	3150	12

In one way, the message-passing overheads illustrated in Table 1.3 are the Achilles heel of the MIMD local-memory machines; in another, they serve to determine those very applications for which message-passing architectures are competitive.

The main features of message passing programming model are:

- Each process has a distinct address space. Data can be transfered from one address space to another.

- If a variable is declared in each process of a computer with p processes, then there are p different variables with the same name, with perhaps different values.
- If function evaluation on a given process requires data from another process, the data must be sent from one process to the other by message passing statements in the source and destination processes.
- The programmer must split the data structures of the problem domain and distribute their parts among the processes.

The division of labor and sharing of data on a distributed memory message passing computer are things that must program explicitly. The compiler does not provide this decomposition.

1.8 Virtual Shared Memory

From the programming standpoint it would be attractive if a distributed memory parallel computer would allow each process to treat all of the memory in the system as a single large pool of memory rather than separate distributed memories. This would allow the programmer to reference data not locally contained through a conventional assignment or reference rather than calls to the message passing library. This would give the illusion of shared memory in a system that has physically distributed memory. Underneath the programming layer there would be separate memories that the system would manage. This would free the programmer from writing explicate message passing statements.

Virtual shared memory blurs the distinction between traditional shared memory MIMD and traditional distributed memory MIMD computing. This model of computing has compelling attractions since it allows a conventional style of programming. One thing to remember that to ensure performance miminizing memory transfers is still critical.

1.8.1 Routing

In a message-passing architecture a physical wire must exist to transmit the message. In any system with a large number of processors, it is not feasible for a wire to exist between every pair of processors. Not only would the wiring become unbearably complicated, but also the number of connections to a single processor would become too great. Thus, in all machines of which we are aware, there is either a global bus to which all processors are connected or a fixed topology between the processors.

The bus connection is more common in a shared-memory architecture where the memory is attached to the bus in much the same way as each processor. There the routing is simple; each processor merely accesses the shared memory directly through the bus. The main problem one can have is bus saturation, because the bus is commonly much slower than the processors and thus can be easily overloaded with data requests. In bus-connected local-memory machines, bus contention is again a problem, but routing is simple and the message-passing model is essentially that of the previous section.

In local-memory architectures, the fixed topology is usually a two-dimensional mesh (more commonly, the ends are wrapped to form a torus) or a hypercube (see Section 1.4). The hypercube is particularly rich in routing paths and has the important property that any two nodes in a k-dimensional hypercube (that is, a hypercube with 2^k nodes) are at most a distance k apart. If a binary representation is used to label the nodes of the hypercube, then direct connections can exist between two nodes whose representation differs in only one bit. A possible routing scheme is to move from one processor to another by changing one bit at a time as necessary and sending the message via the intermediate nodes so identified.

Since intermediate nodes during the routing of a message do nothing except route the message to the next node, it is important that they can do this rerouting without interrupting any work in progress at that intermediate node. The effect of this design is felt strongly in the three versions of the Intel iPSC, where the first version did not have transparent routing. As the data in Table 1.3 show, the transfer rate for sending messages to non-neighboring nodes in the hypercube is considerably reduced in the iPSC/2 although the startup time is still significant [138].

Table 1.3: **Transfer Rates for Message Passing on Hypercubes (kB/sec)**

Message	Adjacent nodes		5 hops	
Length	64 bytes	4096 bytes	64 bytes	4096 bytes
iPSC/1	56	501	18	322
iPSC/2	160	1888	145	1788
iPSC/860	640	2424	457	2269
NCUBE	116	374	41	141

1.9 Interconnection Topology

How information is communicated or passed between processors and memory is the single most important aspect of hardware design from the algorithm writer's point of view. The interconnection network between processors and memories can have various topologies depending on the investment in hardware and desired transfer requirements.

From the physical standpoint, a network consists of a number of switching elements and interconnection links. The two major switching methodologies are *circuit switching* and *packet switching*. In circuit switching, a physical path is actually established between a source and a destination. In packet switching, data is put in a packet and routed through the interconnection network without establishing a physical connection path. In general, circuit switching is much more suitable for bulk data transmission, while packet switching is more efficient for short data messages. A third option is to use both in the same system, producing a hybrid.

The communication links between processing elements fall into two groups, regular and irregular; most are regular. Regular topologies can be divided into two categories:

1. *Static.* In a static topology, links between two processors are passive, and dedicated buses cannot be reconfigured for direct connections to other processors. Topologies in the static category can be classified according to dimensions required for layout, such as one-dimensional, two-dimensional, three-dimensional, and hypercube.

2. *Dynamic.* In a dynamic topology, links can be reconfigured by setting the network's active switching elements. There are three classes: single-stage, multistage, and crossbar.

   - A *single-stage* network is composed of a stage of switching elements cascaded to a fixed connection pattern.
   - A *multistage* network consists of more than one stage of switching elements and is usually capable of connecting an arbitrary input terminal to an arbitrary output terminal. Multistage networks can be one sided or two sided; a one-sided network has the inputs and outputs on the same side, whereas a two-sided network usually has an input side and an output side and can be divided into three subclasses: blocking, rearrangeable, and nonblocking.
   - In a *crossbar* network , every input port can be connected to a free output port without blocking.

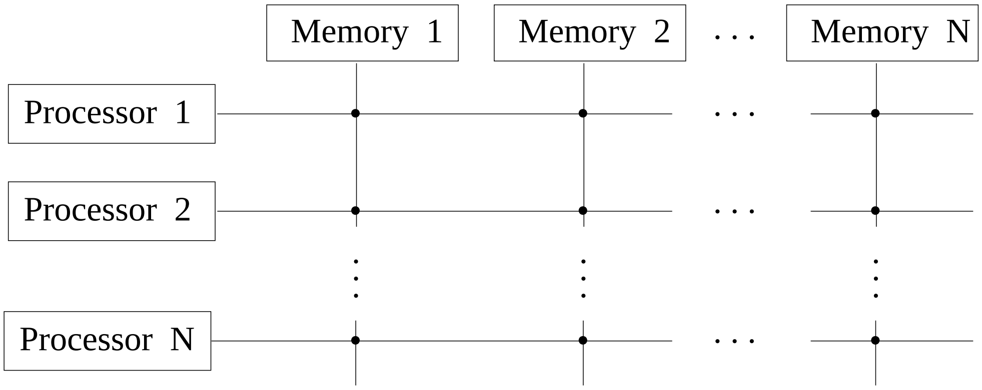

Figure 1.6: Crossbar switch

1.9.1 Crossbar Switch

The crossbar switch (see Figure 1.6) is the most extensive and expensive connection providing direct paths from processors to memories. With p processors and m memory modules, a crossbar connecting them would require $p * m$ switches, where a switch allows a path to be established allowing data to flow from one part of the machine to another. Contention will occur if two or more accesses are made to the same memory, offering minimal possible contention but with high complexity.

1.9.2 Timeshared Bus

Perhaps the simplest way to construct a multiprocessor is to connect the processors to a shared bus (see Figure 1.7). Each processor has access to the common bus, which is connected to a central memory or memories. This configuration allows for easy expansion.

However, one must be cautious about possible degradation in performance with large numbers of processors. Processor synchronization is achieved by reading from and writing to shared-memory locations. As the number of processors increases, there is a tendency for those shared locations to receive an increasing proportion of the memory references. The performance of the system may degrade rapidly if the data transfer rate on the bus, referred to as the *bus bandwidth*, is not able to deliver data to accommodate the processors. Typically, therefore, bus connections are limited to a

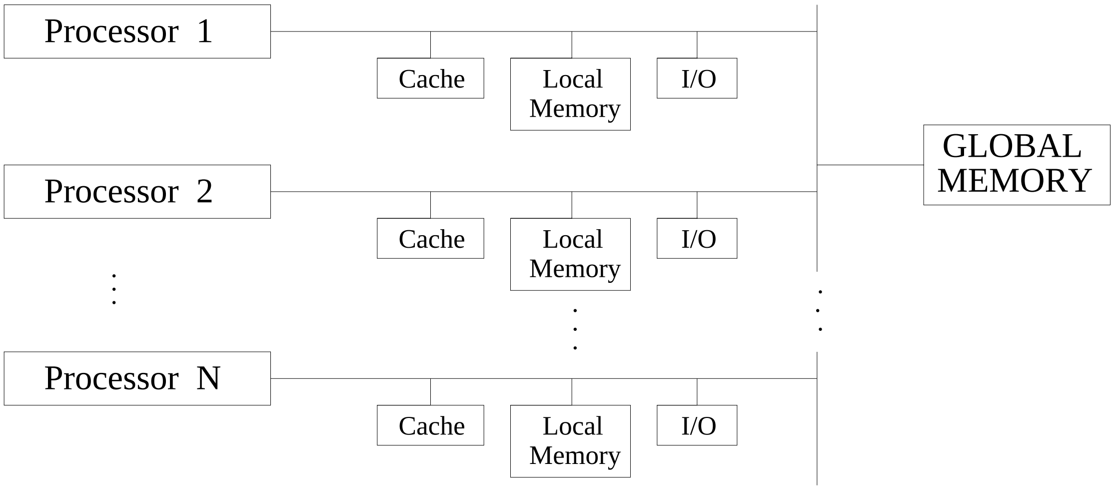

Figure 1.7: **Bus-connected system**

modest number (< 30) of processors. Caches and local memory may also be used to help relieve the bandwidth bottleneck.

1.9.3 Ring Connection

A ring-connected architecture provides point-to-point connections between processors as well as a cyclic interconnection scheme. Processors place on the ring a message containing the destination address as well as the source address. The message goes from processor to processor until it reaches the destination processor. The advantage of a ring is that the connections are point-to-point and not bus connected. To take advantage of the ring, one can treat the architecture as if it were a pipeline. The effective bandwidth can be utilized as long as computations keep the pipeline filled.

A simple variant is a linear array, where a processor is connected to its two nearest neighbors except at the ends where there is only one connection (see Figure 1.8).

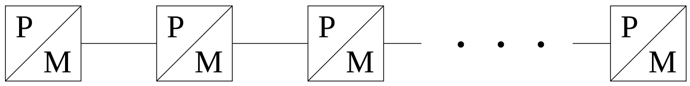

Figure 1.8: Linear array (P = processor, M = memory)

1.9.4 Mesh Connection

A simple extension of a linear array of processors is to connect the processors into a two-dimensional grid, where each processor can be thought of as being connected to a neighbor on the north, south, east, and west. The maximum communication length for p processors connected in a square mesh is $O(\sqrt{p})$.

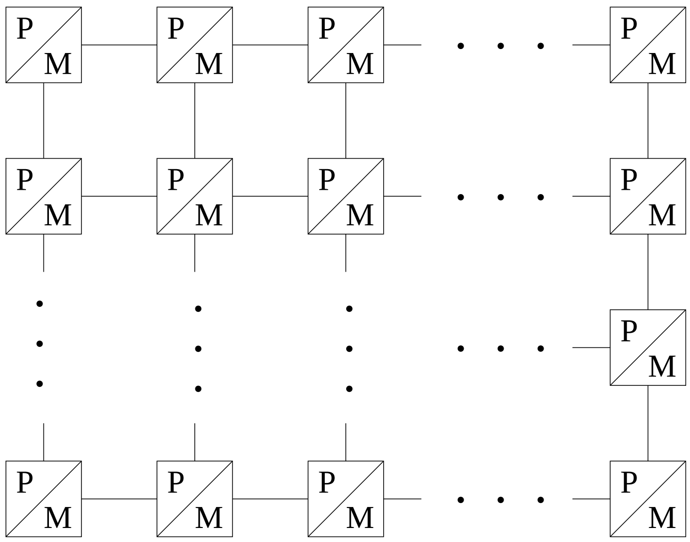

Figure 1.9: **Mesh connection (P = processor, M = memory)**

A variant of this, adopted by machines such as the AMT DAP, is to connect the processors in the first column of Figure 1.9 to those in the last, and those in the first row to those in the last,

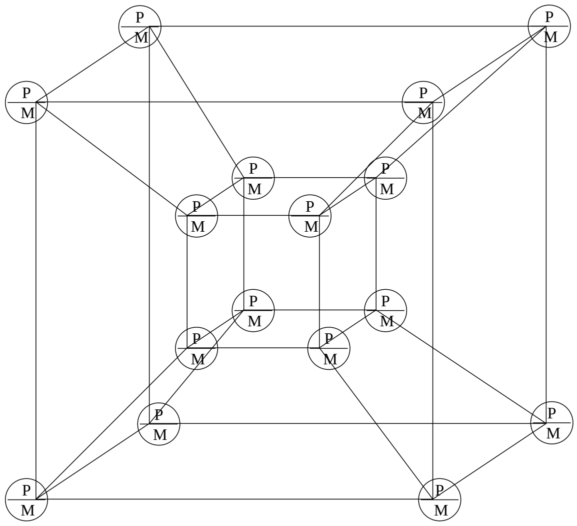

Figure 1.10: Hypercube connection

yielding a toroidal topology.

1.9.5 Hypercube

The hypercube derives its name from the direct connection network used to interconnect its processors or nodes. There are $N = 2^n$ nodes, each of which is connected by fixed communications paths to n other nodes. The value of n is known as the dimension of the hypercube. If the nodes of the hypercube are numbered from 0 to $2^n - 1$, then the connection scheme can be defined by the set of edges that can be drawn between any two nodes whose numberings differ by one bit position in their binary representations. The hypercube design has been used by a number of vendors to form a loosely coupled, distributed-memory, message-passing concurrent MIMD computer. An example of a hypercube topology for $n = 4$ is shown in Figure 1.10.

1.9.6 Multi-staged Network

A multi-staged network is capable of connecting an arbitrary processor to an arbitrary memory. Generally, a multi-staged network consists of n stages where $N = 2^n$ is the number of input and output lines. The interconnection patterns from stage to stage determine the network topology. Each stage is connected to the next stage by N paths (see Figure 1.11).

This strategy approximates the connectivity and throughput of a crossbar switch while reducing its cost scaling factor from N^2 to $N \log N$, at the price of an increase in the network latency of $O(\log N)$.

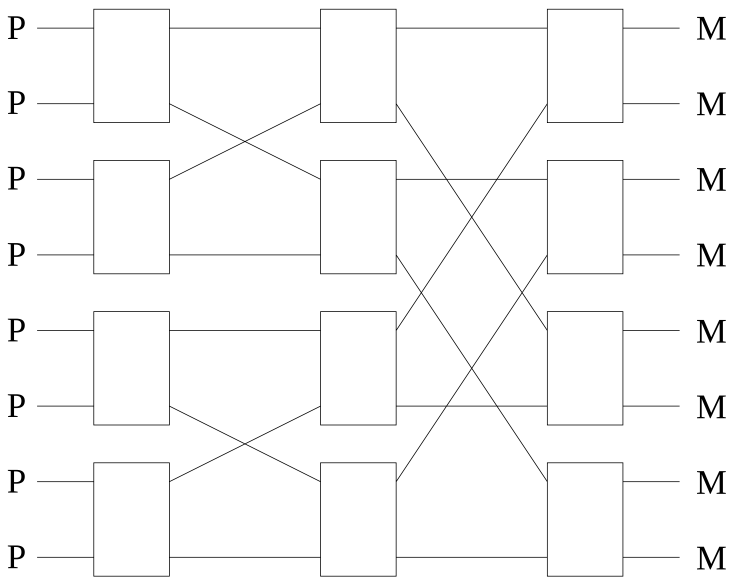

Figure 1.11: **Shuffle-exchange network**

The tradeoffs among some of the different network connections are given in Table 1.4.

Table 1.4: **Tradeoffs among Different Network Connections for p Processors**

Network	Minimum Latency	Maximum Bandwidth per Processor	Wires	Switches	Example
Completely connected	Constant	Constant	$O(p^2)$	-	-
Crossbar	Constant	Constant	$O(p)$	$O(p^2)$	Cray
Bus	Constant	$O(1/p)$	$O(p)$	$O(p)$	SGI Power Challenge
Mesh	$O(\sqrt{p})$	Constant	$O(p)$	-	Intel Paragon
Hypercube	$O(\log p)$	Constant	$O(p \log p)$	-	-
Switched	$O(\log p)$	Constant	$O(p \log p)$	$O(p \log p)$	IBM SP-2

1.10 Programming Techniques

Much of the discussion in Chapters 5 through 7 concerns the efficient programming of vector and parallel computers. Here we concentrate on a few basic ideas.

The first observation is as much algorithmic as programming oriented: the programmer should take advantage of any independence within the calculation by isolating independent calculations, usually in separate subroutines, thus expediting the use of macroprocessing on the target machine(s). Such a modular approach is now generally recognized as good programming practice even in the serial regime.

At the finer level, parallelism (as far as the Fortran programmer is concerned) is usually handled by the compiler (we include any manufacturer-supplied pre- or post-compiler). The normal source of exploitable parallelism is found within the Fortran DO-loop. Thus the primary concern of the programmer should be to remove barriers to vectorization or parallelism from the DO-loops. We now discuss some of these barriers and ways of overcoming them.

Usually the simpler the DO-loop, the easier it is for the compiler to recognize vectorization or parallelism opportunities. In particular, it is important to explicitly remove from DO-loops calculations that need not be present within them (i.e., loop-invariant constructions). Furthermore, since a relatively small proportion of the code may prevent a DO-loop from vectorizing, it may be advantageous to split the loop into two loops—one vectorizable, the other not. IF statements and indirect addressing , in particular, can be vectorized only in simple cases; often, by splitting the loop, the part inhibiting vectorization in the complicated original loop can be vectorized in its simpler form, so that both parts of the split loop vectorize. For a comparison of various vectorizing compilers, see [48].

On some machines, difficult constructs can be replaced by calls to machine-dependent routines. Examples include the replacement of IF statements by conditional vector merges on Crays or the CYBER 205, or the explicit call to gather/scatter routines for indirect addressing on early versions of Cray supercomputers. We caution, however, that while these replacements may be necessary for optimal implementation on particular architectures, they do reduce portability .

Some manufacturers allow programmer assistance to the compiler through directives inserted as special Fortran comments in the code immediately prior to the relevant DO-loop. Each manufacturer provides its own set of compiler directives; but, since the directives appear as comments in the program, they do not limit the portability of the program. For example, to ensure that the compiler will vectorize the following loop, we inserted the following directives to make the compiler ignore potential recursion within the loop. These directives are for the Cray and NEC compiler, respectively.

```
CDIR$ IVDEP
*VDIR NODEP
      DO 10 J = K+1, N
         A(I,J) = A(I,J) - A(I,K)*A(K,J)
   10 CONTINUE
```

Although the necessity for these directives decreases as compilers improve, there are cases when they are essential, and we do not discourage their use in general because portability is not compromised. Nevertheless, a much more satisfactory situation would be for the directives to be standardized. See [215] for a comparison of different parallel Fortran dialects.) We note that the use of the standard and standardized building blocks of the Basic Linear Algebra Subprograms, BLAS (discussed at length in Chapter 5), is satisfactory and should not be considered a barrier to portability.

Many programs have nested loops , commonly only to a depth of two or three. Often, it is mathematically immaterial how the loops are ordered, but the performance can be dramatically affected by this order. Since most vectorizing compilers vectorize only on the innermost loop (the IBM VS Fortran compiler is a notable exception), it is common practice to make the innermost loop that of longest vector length. Similarly, most parallelizing compilers parallelize over the outermost loop so that the size of each task or granularity of separate processes is kept high. On the Alliant the most common way of treating nested loops is termed COVI (concurrent outer – vector inner), although other ways of exploiting loop structure are possible. Again, we acknowledge that as compilers become increasingly sophisticated, such *loop inversion* may be done automatically when beneficial.

Another technique often used with nested loops is *loop unrolling*. As with loop inversion, loop unrolling may be done automatically by the compiler. For loop unrolling, the same vector entry is updated in several successive passes of the innermost loop. For example, in the code

```
      DO 200 J=1,N
         DO 100 I=1,M
            V(I) = V(I) + F(I,J)
100      CONTINUE
200   CONTINUE
```

the entries $V(I)$ might typically be fetched and saved at every iteration of the J loop (loop 200). If, however, the J loop is unrolled (here to a depth of 4), the resulting code

```
      DO 200 J=1,N,4
         DO 100 I=1,M
            V(I) = V(I) + F(I,J)
     *              + F(I,J+1)
     *              + F(I,J+2)
     *              + F(I,J+3)
100      CONTINUE
200   CONTINUE
```

provides four times the number of adds for each vector load/store of V and so should perform better for machines with vector registers or a memory hierarchy. The two problems here are that the code can become complicated (sometimes called pornographic) and can confuse rather than help the compiler. Indeed, as compilers improve and as explicit higher-level BLAS are used in algorithm design, loop unrolling is rapidly becoming an obsolete technique [97, 93].

Finally, there are some constructs that no amount of directives can help—for example, forward recurrences. Here only algorithmic changes can help, which usually increase vectorization or parallelism but at the cost of more arithmetic operations. This topic is discussed further in Section 3.4.

A good reference or guidebook to these techniques can be found in [226].

1.11 Trends Network Based Computing

Stand-alone workstations delivering several tens of millions of operations per second are commonplace, and continuing increases in power are predicted. When these computer systems are interconnected by an appropriate high-speed network, their combined computational power can be applied to solve a variety of computationally intensive applications. Indeed, network computing may even provide supercomputer-level computational power. Further, under the right circumstances, the network-based approach can be effective in coupling several similar multiprocessors, resulting in a configuration that might be economically and technically difficult to achieve with supercomputer hardware.

To be effective, distributed computing requires high communication speeds. In the past fifteen years or so, network speeds have increased by several orders of magnitude.

Among the most notable advances in computer networking technology are the following:

- Ethernet – the name given to the popular local area packet-switched network technology invented by Xerox PARC. The Ethernet is a 10 Mbit/s broadcast bus technology with distributed access control.
- FDDI – the Fiber Distributed Data Interface. FDDI is a 100-Mbit/sec token-passing ring that uses optical fiber for transmission between stations and has dual counter-rotating rings to provide redundant data paths for reliability.
- HiPPI – the high-performance parallel interface. HiPPI is a copper-based data communications standard capable of transferring data at 800 Mbit/sec over 32 parallel lines or 1.6 Gbit/sec over 64 parallel lines. Most commercially available high-performance computers offer a HIPPI interface. It is a point-to-point channel that does not support multi-drop configurations.
- SONET – Synchronous Optical Network. SONET is a series of optical signals that are multiples of a basic signal rate of 51.84 Mbit/sec called OC-1. The OC-3 (155.52 Mbit/sec) and OC-12 (622.08 Mbit/sec) have been designated as the customer access rates in future B-ISDN networks, and signal rates of OC-192 (9.952 Gbit/sec) are defined.
- ATM – Asynchronous Transfer Mode. ATM is the technique for transport, multiplexing, and switching that provides a high degree of flexibility required by B-ISDN. ATM is a connection-oriented protocol employing fixed-size packets with a 5-byte header and 48 bytes of information.

These advances in high-speed networking promise high throughput with low latency and make it possible to utilize distributed computing for years to come. Consequently, increasing numbers of universities, government and industrial laboratories, and financial firms are turning to distributed computing to solve their computational problems.

Chapter 2

Overview of Current High-Performance Computers

A much-referenced taxonomy of computer architectures was given by Flynn [153]. He divided machines into four categories: SISD (single instruction stream/single data stream), SIMD (single instruction stream/multiple data stream), MISD (multiple instruction stream/single data stream), and MIMD (multiple instruction stream/multiple data stream). Although these categories give a helpful coarse division (and we, in fact, use these categories throughout our book), the current situation is more complicated, with some architectures exhibiting aspects of more than one category. Indeed, many of today's machines are really a hybrid design. For example, the CRAY X-MP has up to four processors (MIMD), but each processor uses pipelining (SIMD) for vectorization. Moreover, where there are multiple processors, the memory can be local or global or a combination of these. There may or may not be caches and virtual memory systems, and the interconnections can be by crossbar switches, multiple bus-connected systems, timeshared bus systems, etc.

In this chapter we briefly discuss advanced computers in terms of several different categories, more closely related to size and cost than to design: supercomputers, mini-supercomputers, vector mainframes, and novel parallel processors.

2.1 Supercomputers

Supercomputers are by definition the fastest and most powerful general-purpose scientific computing systems available at any given time. They offer speed and capacity significantly greater than the most widely available machines built primarily for commercial use. The term *supercomputer* became

Table 2.1: Performance Trends in Scientific Supercomputing

Year	Machine	Speed
1964	CDC 6600	1 Mflop/s
1975	CDC 7600	4 Mflop/s
1979	CRAY-1	160 Mflop/s
1983	CYBER 205	400 Mflop/s
1986	CRAY-2	2 Gflop/s
1990	NEC SX-3	22 Gflop/s
1992	TMC TM-5	130 Gflop/s
1994	Fujitsu VPP-500	205 Gflop/s
1997	Intel	1.8 Tflop/s

prevalent in the early 1960s, with the development of the CDC 6600. That machine, first marketed in 1964, boasted a performance of 1 Mflop/s (millions of floating-point operations per second). (Throughout this book we shall refer to a floating-point operation as either a floating-point addition or multiplication in full precision (usually 64-bit arithmetic).)

During the next fifteen years, the peak performance of supercomputers grew at an extremely rapid rate; and since 1980, that trend has accelerated. Machines projected for 1997 are expected to have a maximum speed of close to 2 Tflop/s (trillions of floating-point operations per second), more than 2,000,000 times that of the CDC 6600 (see Table 2.1).

By far the most significant contributor to supercomputing in the United States has been private industry. Companies led by Cray Research, IBM, Intel, and SGI have devoted their resources to producing state-of-the-art machines that enable scientists and engineers to tackle problems never before possible. It is from these commercial ventures that we have seen the development of computers capable of solving complex numerical and non-numerical problems. Table 2.1 lists the principal conventional supercomputers that have been marketed in the past several years. We include the CRAY-1 largely as a benchmark, since it could not now be considered a supercomputer in terms of performance. The next generation, with higher speed and more parallelism, is already under development.

The price of the systems in Table 2.1 depends on the configuration, with most manufacturers offering systems in the \$5 million to \$20 million range. All use ECL logic with LSI (Large Scale Integration), except the CRAY X-MP, the CRAY-1 built with SSI (Small Scale Integration), and the ETA-10 in CMOS ALSI (Advanced Large Scale Integration). All use pipelining and/or multiple functional units to achieve vectorization/parallelization within each processor. Cray and IBM are the only supercomputer manufacturers to offer multiple-processors machines, although some additional vendors have announced multiprocessor machines. The form of synchronization on both the Cray and IBM machines is essentially event handling.

2.2 RISC Based Processors

Significant advances have occurred in RISC microprocessors over the past few years. The very first RISC machines could achieve only one instruction per clock, in the best of circumstances. In 1988 came a machine that could do more than one instruction per clock, but only one floating-point operation. Around 1989, the IBM RS/6000 introduced the concept of doing more than one floating-point operation. And now we have processors that can do not only more than one floating-point operation but one more than one memory reference per clock cycle. This match-up is critical because doing a lot of floating-point operations is a waste of time unless the data can get to part of the machine where the computation is performed.

With their faster clock rates, increased density, and memory hierarchies, microprocessors are rapidly closing the performance gap with supercomputers. In addition, microprocessors have an obvious and substantial cost advantage.

The IBM RISC System/6000 has a good example of a RISC based processor. In the RS/6000 there is a superscalar second-generation RISC architecture [31]. It is the result of advances in compiler and architecture technology that have evolved since the late 1970s and early 1980s.

Like other RISC processors, the RISC System/6000 implements a register-oriented instruction set, the CPU is hardwired rather than microcoded, and it features a pipelined implementation. The floating-point unit is integrated in the CPU, minimizing the overhead associated with separate floating-point coprocessors.

The RISC System/6000 has the ability to dispatch multiple instructions and to overlap the execution of the fixed-point, the floating-point, and the branch functional units.

The design has allowed the implementation of a CPU that executes up to four instruction per cycle: one branch instruction, one condition register instruction, one fixed-point instruction, and one floating-point multiply-add instruction. A second pipelined instruction can begin on the next cycle on an independent set of operands. This means that two independent floating-point operations per cycle can be executed.

Of particular interest is the fact that loads and independent floating-point operations can occur in parallel. The compiler takes advantage of this capability in many cases: with a well-designed algorithm, it is possible to execute two floating-point operations on separate data items and "hide" one memory reference all in the same cycle.

RISC processors are close to matching the performance level of vector processors with matched cycle times. Because of the regularity of vector loops and the ability of the RISC architecture to issue floating-point instruction every cycle and complete two floating-point operations per cycle we expect that the RISC superscalar machines will perform at the same rates as the vector machines for vector operations with similar cycle times. Moreover, the RISC machines will exceed the performance of those vector processors on non-vector problems.

Table 2.3 list some of the current RISC processors in terms of their peak and achieved performance on the LINPACK Benchmark.

Also listed in Table2.3 are some of the supercomputers from Cray Research. Its interesting to see that the current line of RISC processors are about one generation behind the conventional vector supercomputers in terms of their peak performance. The achieved performance lages further behind as a result of the memory bandwidth on the RISC processors does not achieve the same balance as in the conventional vector supercomputers.

2.3 Parallel Processors

While most supercomputers in the past have looked to vector processing to provide performance, more recently computational scientists have turned to parallel-processing. A number of vendors are developing parallel-processing systems. Such systems range from smaller (8- to 32-processor) machines such as the SGI or Fujitsu to highly parallel (hundreds of processors) systems such as the IBM SP-2 or the Cray T3E.

A few years ago there where two approaches to multiprocessing are being actively investigated: SIMD and MIMD. The three principal SIMD machines on the market are listed in Table 2.4.

Over the past few years is became apparent that SIMD machines had limitation on a number of important scientific problems and are no longer considered for general purpose scientific computing.

The main stay of parallel scientific computing falls into MIMD class of systems. MIMD systems fall into two classes: shared memory and distributed memory. Shared-memory architectures are composed of a varying number of processors and memory modules connected by means of a high-speed interconnect, such as a crossbar switch or efficient routing network. All processors share all memory modules and have the ability to execute different instructions on each of the processors by using different data streams. Examples of systems in this class of architecture are shown in Table 2.5.

Distributed-memory architectures are composed of a number of processing nodes, each containing one or more processors, local memory, and communications interfaces to other nodes. Such architectures are scalable, have no shared memory among the processing nodes, exchange data through their network connections, and execute independent (multiple) instruction streams by using different data streams. Examples of systems in this class are listed in Table 2.6.

In the discussion of parallel machines, the implementations of the two memory organizations are often misinterpreted. The term "shared memory" means that a single address space is accessible to every processor in the system. Hence any storage location may be read and written by any processor. Communication of concurrent processes is accomplished through *synchronized* access of shared variables in a shared memory system. The opposite of a shared address space should be multiple private address spaces, implying explicit communication. In this case communication between concurrent processes requires the explicit act of sending data from one processor to be received by another processor. These notions do not imply anything about the physical partitioning of the memory, nor anything about the proximity (in terms of access time) of a processor to a memory module. If physical memory is divided into modules with some placed near each processor (allowing faster access time to that memory), then physical memory is distributed. The real opposite of distributed memory is centralized memory, where access time to a physical memory location is the same for all processors. Hennessy and Patterson present a lively discussion of this topic [195].

From this line of reasoning, we can see that shared address vs. multiple addresses and distributed memory vs. centralized memory are separate, distinct issues. SIMD or MIMD architectures can have shared address and a distributed physical memory (for example, the Convex Exemplar and the SGI Power Challenge Array).

Figure 2.1 shows several machines according to this scheme.

In general, one can argue that it is easier to develop software for machines on the shared side of the addressing axis and that it is easier to build large-scale machines on the distributed end of the vertical axis.

It is our belief that parallel processing today is heading in the direction of a shared address space where the physical memory is distributed. In this model, the hardware deals with issues of data movement. This is not to say that data placement is not important. Indeed, from a standpoint of efficiency, data placement and minimization of data movement will be of paramount importance. No matter which communication scheme is used, the programmer and/or compiler must be conscious of the fact that the software generated will be for a large-scale parallel machine.

Since different memories have different access times, choices for placement and movement of data as the computation proceeds can have a significant effect on the overall performance of applications.

The model of programming from the user's point of view may look like that of Figure 2.2, processors in a *sea of memory*, where the processors have access to all memory through a shared address space, and the time to access the data depends on the distance the data is from the processor.

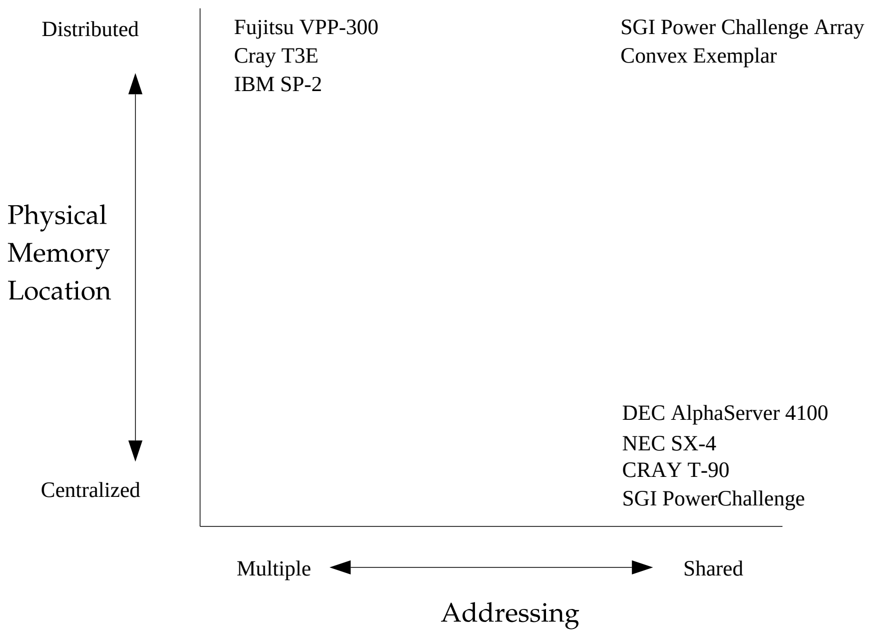

Figure 2.1: Physical memory vs. address space

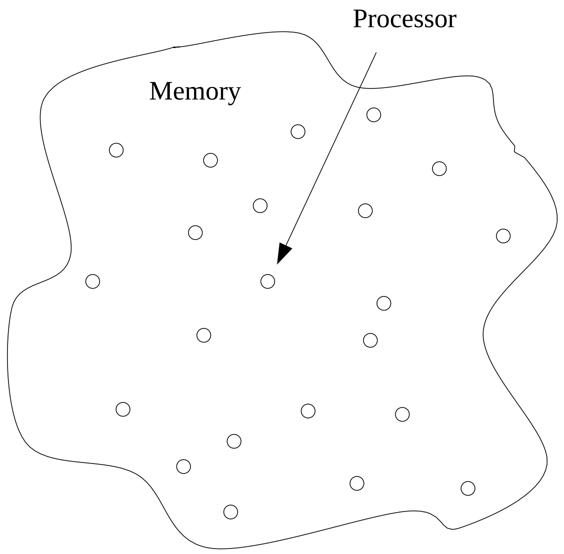

Figure 2.2: Programming model from user's viewpoint

Table 2.2: Supercomputers: Past, Present, and Future

Machine	Maximum Rate, in Mflop/s	Memory, in Mbytes	Number of Processors
CRAY-1 †	160	32	1
CRAY X-MP †	941	512	4
CRAY-2	1951	4096	4
CRAY Y-MP †	2667	256	8
CRAY-3 †	16000	16384	16
CRAY C-90	16000		16
CRAY T-90	58182		32
CYBER 205 †	400(a)	128	1
ETA-10P †	333	2048(b)	2
ETA-10Q †	1680	2048(b)	8
ETA-10E †	3048	2048(b)	8
Fujitsu VP-30E *	133	1024	1
Fujitsu VP-50E *	286	1024	1
Fujitsu VP-100E *	429	1024	1
Fujitsu VP-200E *	857	1024	1
Fujitsu VP-400E *	1714	1024	1
Fujitsu VP-2600/20 *	8000	2048	1
Fujitsu VPP-300 *	35200	32768	16
Hitachi S-810/20	857	512(c)	1
Hitachi S-820/80	3000	512(c)	1
NEC SX/1E	324	1024(d)	1
NEC SX/1A	650	1024(d)	1
NEC SX/2A	1300	1024(d)	1
NEC SX/3 model 44	22000	16384	4
NEC SX-4	64000		32

(a) 800 Mflop/s for 32-bit arithmetic
(b) Also 4 Mwords of local memory with each processor
(c) Also a 12-Gbyte extended memory
(d) Also an 8-Gbyte extended memory
†Computer no longer manufactured
*Marketed in the West by Siemens (the VP-50 to 400 and S series range)

Table 2.3: Performance Numbers Comparing Various RISC Processors Using the Linpack Benchmark

Machine	MHz	Cycle time nsec	Linpack n=100 Mflop/s	% peak	Ax = b n=1000 Mflop/s	% peak	Theor Peak Mflop/s
HP PA-8000	180	5.5	158	22%	510	71%	720
DEC Alpha	350	2.9	164	23%	510	73%	700
HP PA-8000	160	6.2	140	22%	421	66%	640
IBM Power2	71.5	14	140	49%	254	89%	286
SGI Power	90	11.1	126	42%	308	86%	360
HP 735	100	10	61	31%	107	54%	198
DEC Alpha	200	5	43	22%	155	78%	200
DEC Alpha	182	5	39	21%	141	77%	182
IBM RS-580	66	15	38	30%	104	83%	125
DEC Alpha	160	6	36	23%	114	71%	160
DEC Alpha	150	7	30	20%	107	71%	150
IBM RS-550	42	24	26	31%	70	83%	84
HP 730/750	66	15	24	36%	47	71%	66
SGI Indy2/Crim	100	10	15	30%	32	64%	50
IBM RS-530	25	40	15	30%	42	84%	50
KSR (1 proc)	40	25	15	38%	31	78%	40
Intel i860	40	25	10	25%	34	85%	40
CRAY T90	454	2.2	576	32%	1603	89%	1800
CRAY C90	238	4.2	387	41%	902	95%	952
CRAY J90	100	10	106	53%	203	101%	200
CRAY Y-MP	166	6	161	48%	324	97%	333
CRAY X-MP	118	8.5	121	51%	218	93%	235
CRAY 3	481	2.1	327	34%	876	92%	948
CRAY 2	244	4.1	120	25%	384	79%	488
CRAY 1	80	12.5	27	17%	110	69%	160

Table 2.4: SIMD Systems

Company	System	Elements	Topology
MasPar	MP-2	1,024 - 16,384	nearest neighbor

Table 2.5: MIMD Systems with Shared Memory

Company	System	Elements	Connection
Convex	Exemplar SPP	4 - 64	distributed crossbar
CRAY	J-90	1 - 32	crossbar
CRAY	T-90	1 - 32	crossbar
DEC	AlphaServer 4100	1 - 4	bus
NEC	SX-4	1 - 32	crossbar
SGI	Power Challenge	1 - 18	bus
Sequent	Symmetry	2 - 30	bus
Sun Microsystems	Enterprise 6000	1 - 30	bus

Table 2.6: MIMD Systems with Distributed Memory

Company	System	Processors	Topology
Cray Research	T3E	16 - 2048	3-D mesh
Fujitsu	VPP-300	4 - ??	2-D torus
IBM	SP-2	8 - 512	switch network
Intel	Paragon	8 - 9000	mesh
nCube	Model 2	64 - 2048	hypercube

Chapter 3

Implementation Details and Overhead

In this chapter, we are concerned with efficiency on vector and parallel processors. Specifically, we focus on implementation details that can decrease overhead and can improve performance. In a few instances we also discuss how an understanding of these implementation details can be exploited to achieve better efficiency.

3.1 Parallel Decomposition and Data Dependency Graphs

An illuminating way to describe a parallel computation is to use a control flow graph. Such a graph consists of nodes that represent processes and directed edges that represent execution dependencies. These graphs can represent both fine-grain and coarse-grain parallelism. However, as an aid to programming they are perhaps best restricted to the coarse-grain setting. In this case it is convenient to have the nodes represent serial subroutines or procedures that may execute in parallel. Leaves of such a graph represent processes that may execute immediately and in parallel. The graph asserts that there are no data dependencies between leaf processes. That is, there is no contention for write access to a common location between two leaf processes. The control flow representation of parallel computation differs from a data flow representation in the sense that the edges of the control flow graph do not represent data items. Instead, these edges represent assertions that data dependencies do not exist once there are no incoming edges. The programmer must take the responsibility for ensuring that the dependencies represented by the graph are valid.

The term *execution dependency* is self-explanatory. It means that one process is dependent upon the completion of, or output from, another process and is unable to begin execution until this condition is satisfied. Additional processes may in turn be dependent on the execution of this one. The parallel execution may be coordinated through the aggregation of this information into a

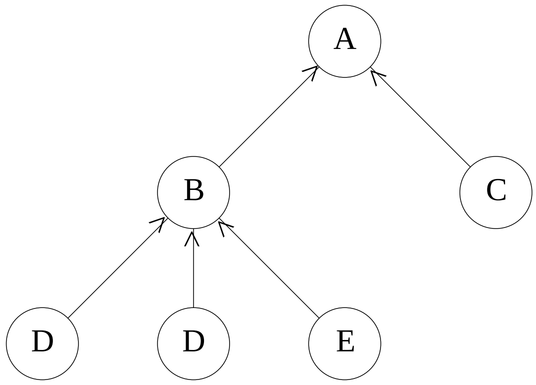

Figure 3.1: Large-grain control flow graph

control flow graph.

A special case of execution dependency is *deadlock*. This phenomenon, encountered only in MIMD machines, is caused by a circular dependency chain so that all activity ceases, with all processors either idle or in a wait state. Although it is sometimes described as the *bête noir* of parallel processing, it is by no means the most difficult bug to detect or fix. In particular, acyclic control flow graphs do not admit deadlock. Thus, describing a parallel decomposition in terms of such a graph avoids this problem from the outset.

We shall try to explain the concept of a control flow graph through a generic example (see Figure 3.1. In this example we show a parallel computation consisting of five processes. The nodes of the graph correspond to five subroutines (or procedures) A, B, C, D, and E (here with two "copies" of subroutine D operating on different data). We intend the execution to start simultaneously on subroutines C, D, D, and E since they appear as leaves in the dependency graph (D will be initiated twice with different data). Once D, D, and E have completed, B may execute. When B and C have completed execution, A may start. The entire computation is finished when A has completed.

This abstract representation of a parallel decomposition can readily be transformed into a parallel program. In fact, the transformation has great potential for automation. These ideas are developed more completely in [100, 101]. A nice feature of the control flow graph approach is that the level of detail required for the representation of a parallel program corresponds to the abstract level at which the programmer has partitioned his parallel algorithm. Let us illustrate this with a simple example—the solution of a triangular linear system partitioned by blocks.

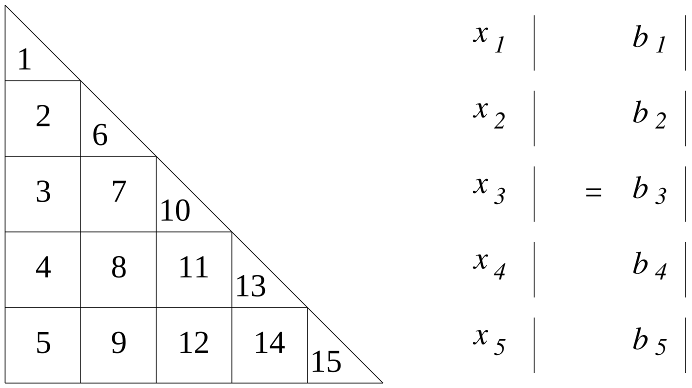

Figure 3.2: Partitioning for the triangular solve

We can consider solving a triangular system of equations $Tx = b$ in parallel by partitioning the matrix T and vectors x and b as shown in Figure 3.1.

The first step is to solve the system $T_1 x_1 = b_1$. This will determine the solution for that part of the vector labeled x_1. After x_1 has been computed, it can be used to update the right-hand side with the computations

$$b_2 = b_2 - T_2 x_1$$

$$b_3 = b_3 - T_3 x_1$$

$$b_4 = b_4 - T_4 x_1$$

$$b_5 = b_5 - T_5 x_1.$$

Notice that these matrix-vector multiplications can occur in parallel, as there are no dependencies. However, there may be several processes attempting to update the value of a vector b_j (for example, 4, 8, and 11 will update b_4), and these will have to be synchronized through the use of locks or the use of temporary arrays for each process. As soon as b_2 has been updated, the computation of x_2 can proceed as $x_2 = T_6^{-1} b_2$. Notice that this computation is independent of the other matrix-vector operations involving b_3, b_4, and b_5. After x_2 has been computed, it can be used to update the right-hand side as follows:

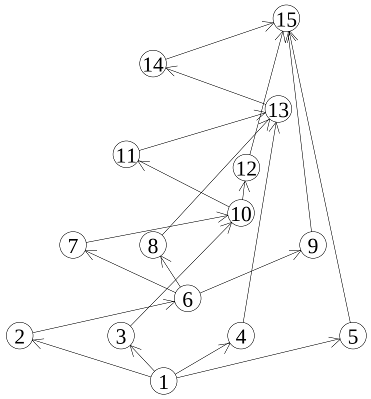

Figure 3.3: Dependency graph for solution of triangular systems

$$b_3 = b_3 - T_7 x_2$$
$$b_4 = b_4 - T_8 x_2$$
$$b_5 = b_5 - T_9 x_2.$$

The process is continued until the full solution is determined. The control dependency graph for this is shown in Figure 3.3.

A complete understanding of the data dependencies among communicating parallel processes is, of course, essential to obtaining a correct parallel program. Once these dependencies have been understood, a control flow graph can be constructed that will assure that the data dependencies are satisfied. A control flow graph may go further by introducing execution dependencies to force a particular order of computation if that is deemed necessary for purposes of load balancing or performance.

3.2 Synchronization

Although the mechanisms for synchronization vary between machines, the necessity and reason for synchronization are the same in all cases. Synchronization is required when two or more concurrently executing processes must communicate, to signal their completion to each other, or to indicate that some data is now available for the use of the other, or simply to provide information to the other of its current progress. The two (or more) processes can then synchronize in the same way that people going on separate missions might synchronize their watches.

The crudest and most common form of synchronization is that of the *barrier*, often used in the join of a fork-and-join. Here a task records its completion and halts. Only when all (or a designated subset) of processes have reported to the barrier can the tasks halted at the barrier continue.

Most shared-memory parallel computers have at least one synchronization primitive. These are typically a test-and-set, or lock, function. A lock function tests the value of a lock variable. If the value is *on,* the process is blocked until the lock variable is *off.* When the lock variable is found to be *off,* it is read and set to *on* in an autonomous operation. Thus, exactly one process may acquire a lock variable and set it to *on.* A companion to a lock function is an unlock function. The purpose of an unlock function is to set the value of a lock variable to *off.*

These low-level synchronization primitives may be used to construct more elaborate primitives such as a barrier. However, explicit use of locks for synchronization should be avoided. Instead, one should try to use higher-level synchronization mechanisms available through the compiler or system, since the likelihood of introducing synchronization errors is high when low-level locks are used to construct intricate synchronization.

Locks are perhaps the easiest form of synchronization to use and understand. Normally, the programmer is allowed to designate locks by number. When a task wishes to have exclusive use of a section of code (and, presumably, exclusive use of the data accessed), it tries to set a lock through a subroutine call. In this situation a programmer has a section of program that is to be executed by at most one process at a time, called a *critical section.* If the lock has already been set by another task, it is unable to set the lock and continue into the critical section until the task that set that particular lock unsets or unlocks it. An example of the use of locks and critical sections follows:

```
CALL LOCK( <lock variable> )
     execute critical section
CALL UNLOCK( <lock variable> )
```

A more flexible and powerful form of synchronization can be achieved through the use of *events.* Events can be posted, awaited, checked for, and unposted. Any processor can unpost an event, and, after checking its existence, the programmer is free to continue if appropriate. Although this form of synchronization is not supported by all manufacturers, it is becoming increasingly common.

3.3 Load Balancing

Load balancing involves allocating tasks to processors so as to ensure the most efficient use of resources. With the advent of parallel processing, the ability to make iterations of loops run in parallel, referred to as microtasking, has become increasingly important. It can be very efficient to split off even small tasks: the determining factor is whether wall clock time has been reduced without having to pay significantly more for the increase in CPU time (resulting from parallel overhead).

When a job is being multiprocessed, a common measure of efficiency is the ratio of the elapsed time on one processor to that on p processors divided by p. This ratio reflects both time lost in communication or synchronization and idle time on any of the p processors. It is this idle time that load-balancing strategies seek to minimize.

To illustrate the effect of load balancing, let us assume that we have four processors and seven pieces of work (tasks) that can be executed simultaneously by any processor. Assume that six of the tasks each require 1 unit of work and the seventh requires 2 units. Clearly, a good load-balancing strategy would allocate the six jobs to three processors and let the longest task reside on the fourth. In that way the job can be completed in two units, the length of the longest task. If, however, two of the short tasks are executed on the same processor as the longest one, then the job will take 4 units and only half the processing power will be used.

The situation can become more complicated when the length of different tasks is not so simply related or if the length of a task is not known until it is executing. In a sense, we have a dynamic bin-filling problem to solve. We now—and not for the first time—encounter a tradeoff situation in parallel processing. Clearly, load balancing is easier if we have smaller tasks. However, smaller tasks have the adverse effect of increasing synchronization costs or startup overhead.

We acknowledge that the importance of load balancing *can* be overstated. For example, in a system that is multiprogrammed, the fact that one or more processors are idle should not be of great concern since the idle time (from one user's point of view) can be taken up by another job. This point is of particular importance when the parallelism, measured in number of simultaneously executable tasks, varies during the course of a job. Thus it might be efficient at one time to use all the processors of a system while at another time to use only one or two processors.

Finally, we observe that on a highly parallel machine, many tasks are required to saturate the machine, so that the problem size has to be large enough to generate enough parallelism. This phenomenon has been observed and documented by the group at Sandia (see, for example, [186, 187]) who have revised Amdahl's law on the basis of constant runtime rather than constant problem size (see Sections 4.1 and 4.3).

3.4 Recurrence

Recurrence refers to the use of a previously computed value of an expression used in a loop. One can have either a forward or a backward recurrence. Backward recurrences—where old values of earlier data are used—create no problems for vectorization or parallelism. Forward recurrences—the dependence of an assignment upon data that has just been reset—do inhibit vectorization. Consider the following simple loop:

```
      DO 10 I = 1,K
         A(I) = A(I+L)
10    CONTINUE
```

If the integer variable L is positive, we have a backward recurrence, and no problems with vectorization need arise. If L is negative, however, the new value for *A(I)* is dependent on a value computed only L steps previously. This means that, without restructuring, the loop cannot be vectorized because the assignment to *A(I+L)* will still be in the arithmetic pipes when the new value is needed for the assignment to *A(I)*. Finally, if the value of L is not known at compile time, then the compiler must assume that it is unsafe to vectorize and should not do so unless the programmer knows that its sign will always be positive and so informs the compiler by means of a directive.

Forward recurrences are often encountered in the solution of sparse linear systems, especially when iterative methods are used. Often, as in discretized two-dimensional and three-dimensional partial differential equations, these recurrences can be avoided by a suitable rearrangement of the computational scheme. For example, for certain classes of problem, a number of similar forward recurrences can be solved simultaneously, a procedure that then leads to possibilities for vectorization or parallelism. We shall see examples of this approach in Chapter 7.

In many situations, however, one really needs to solve only one (large) bi- or tridiagonal system before other computations can proceed. Three basic techniques exist for vectorizing or parallelizing the algorithm for these linear systems: recursive doubling [106, 297], cyclic reduction [194, 223], and divide and conquer [53, 322]. Let us briefly describe these three techniques with respect to their application for bidiagonal systems (the application to tridiagonal systems is similar and is described in the references). For simplicity we assume that the bidiagonal system has a unit diagonal (this can often be arranged in iterative solution methods, and it saves costly division operations; see Chapter 7). Then the solution of

$$\begin{pmatrix} 1 & & & & & & \\ -a_2 & 1 & & & & & \\ & -a_3 & 1 & & & & \\ & & \cdot & \cdot & & & \\ & & & \cdot & \cdot & & \\ & & & & \cdot & \cdot & \\ & & & & & -a_n & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \cdot \\ \cdot \\ \cdot \\ b_n \end{pmatrix}$$

is obtained by the forward recurrence

$$x_1 = b_1$$

$$x_i = b_i + a_i * x_{i-1}, i = 2, 3, ..., n.$$

The basic trick of *recursive doubling* is to insert the recursion for x_{i-1} in the right-hand side:

$$\begin{aligned} x_i &= b_i + a_i * (b_{i-1} + a_{i-1} * x_{i-2}) \\ &= \tilde{\tilde{b}}_i + \tilde{a}_i * x_{i-2}. \end{aligned}$$

Hence we obtain two recurrence relations: one for the even indexed unknowns and one for the odd indexed unknowns. These relations have been obtained at the cost of the computation of the new coefficients $\tilde{a}_i$ and $\tilde{b}_i$, which can be done in vector mode or in parallel. The trick can be repeated and leads each time to twice as many recurrence relations of half the length of the previous step. For some parallel architectures this may be an advantage, but for vector machines it does not really relieve the problem.

The situation is slightly better with *cyclic reduction,* where the repetition of the recurrence relation is carried out only for the even indexed unknowns. This leaves a recurrence of half the original length for the even unknowns. After that has been solved, the odd indexed unknowns can be solved by one vector operation. This trick can also be repeated, and many optimized codes are based on a few steps of cyclic reduction. On vector machines it is often advantageous to combine two successive steps of cyclic reduction.

The third technique can be regarded as a *divide and conquer* approach. The approach is to view the given bidiagonal system as a block bidiagonal system. In the first step the off-diagonal elements in all the diagonal blocks are eliminated, which can be done in parallel or in vector mode. This step leads to fill-in in the off-diagonal blocks. The fill-in is then eliminated in the second step, which can also be done either in parallel or in vector mode. The divide and conquer technique was first proposed in [53], but it is more widely known as Wang's method after Wang [322] who described the method for the parallel solution of tridiagonal systems. In [313] this approach is taken for the vectorization of the algorithm for bidiagonal systems, and its performance is analyzed for some different vector architectures. The divide and conquer technique has been generalized by Meier [235] and Dongarra and Johnsson [98] for more general banded linear systems.

All three techniques lead to a substantial increase in the number of floating-point operations (for bidiagonal systems, a factor of about 2.5), which effectively reduces the gains obtained by vectorization or parallelism. This makes these techniques unattractive in a classical serial environment. Another potential disadvantage is that these techniques require the systems to be of huge order (several thousands of unknowns) in order to outperform the standard forward recurrence with respect to CPU time. Often, scalar optimization by loop unrolling leads to surprisingly effective code. See [276] for an extreme example.

3.5 Indirect Addressing

Simple loops involving operations on full vectors can be implemented efficiently on both vector and parallel machines, particularly if access is to components stored contiguously in memory. For many machines, constant stride vectors can be handled with almost the same efficiency so long as bank conflict or cache problems are avoided. For many calculations, however—for example, computations on sparse matrices—access is not at all regular and may be directed by a separate index vector.

A typical example is a sparse SAXPY operation of the form

```
      DO 10 I = 1,K
         W(I) = W(I) + ALPHA * A(ICN(I))
   10 CONTINUE
```

where the entries of A are chosen by using the index vector ICN. One problem is that the compiler typically will have no knowledge of the values in ICN and hence may not be able to reorganize the loop for good vectorization or parallelism. The original vector supercomputers, notably the CRAY-1, could do nothing but execute scalar code with such loops, but now most machines offer hardware indirect addressing which allows vectorization.

The implementation varies from machine to machine. For a vector supercomputer, the entries of ICN are usually loaded into a vector register and then used in a hardware instruction to access the relevant entries of A. If the machine has several pipes to memory, a high asymptotic rate can still be reached, although the startup time is usually substantially greater than for similar operations with direct addressing because of the extra loads for ICN. We see that typically between a quarter and a half of the asymptotic rate for directly addressed loops can be obtained with a startup time of about twice the directly addressed counterpart.

The moral is that today we need not shy totally away from indirect addressing, although we should be aware of the possibilities of bank conflict and the higher startup time. The last factor can be especially important when dealing with sparse matrices because typical loop lengths are related to the number of entries in a row or column rather than the problem dimension. This situation has led, for example, to novel schemes for holding sparse matrices prior to matrix-vector multiplication (see, for example, Saad [268] or Radicati and Robert [261]) in order to increase the length of the indirectly addressed loop.

A final word of advice: if one is assigning to an indirectly addressed vector (that is, a reference of the form `A(ICN(I))` appears on the left of the equation), an automatic parallelizer cannot know whether repeated reference is being made to the same location, and so automatic parallelizing will not take place. In such instances, the programmer may choose to use compiler directives to ensure parallelization of the loop.

3.6 Message Passing

On a shared-memory machine, communication between tasks executing on different processors is usually through shared data or by means of *semaphores*, variables used to control access to shared data. On a local-memory machine, however, the communication must be more explicit and is normally done through message passing.

In message passing, data is passed between processors by a simple send-and-receive mechanism. The main variants are whether an acknowledgment is returned to the sender, whether the receiver is waiting for the message, and whether either processor can continue with other computations while the message is being sent.

There are a number of factors that can affect message-passing performance. The number of times the message has to be copied or touched (e.g., checksums) is probably most influential and obviously a function of message size. The vendor may provide hints as to how to reduce message copies, for example, posting the receive before the send. Second order effects of message size may also affect performance. Message lengths that are powers of two or cache-line size may provide better performance than shorter lengths. Buffer alignment on word, cache-line, or page may also affect performance. For small messages, context-switch times may contribute to delays. Touching all the pages of the buffers can reduce virtual memory effects. For shared media, contention may also affect performance.

There are of course other parameters of a message-passing system that may affect performance for given applications. The aggregate bandwidth of the network, the amount of concurrency, reliability, scalability, and congestion management may be issues.

An important feature of message passing is that, on nearly all machines, it is much more costly than floating-point arithmetic. Message passing is normally modeled by the formula

$$t = (\frac{k}{B})\alpha + \beta k, \tag{3.1}$$

where t is the time to pass a message of length k between two processors, α the startup cost, B the buffer size, and β the unit transfer cost. In most machines, α, the latency, is very high, as we illustrate in Table 3.1.

Message passing time is usually a linear function of message size for two processors that are directly connected. For more complicated a networks, a per-hop delay may increase the message-passing time. Message-passing time, t_n, can be modeled as

$$t_n = \alpha + \beta n + (h-1)\gamma$$

with a start-up time, α, a per-byte cost, β, and a per-hop delay, γ, where n is the number of bytes per message and h the number of hops a message must travel. On most current message-passing multiprocessors the per-hop delay is negligible due to "worm-hole" routing techniques and the small diameter of the communication network [139]. The results reported in this report reflect nearest-neighbor communication. A linear least-squares fit can be used to calculate α and β from experimental data of message-passing times versus message length. The start-up time, α, may be slightly different than the zero-length time, and $1/\beta$ should be asymptotic bandwidth. The message length at which half the maximum bandwidth is achieved, $n_{1/2}$, is another metric of interest and is equal to $(\alpha + (h-1)\gamma)/\beta$ [203]. As with any metric that is a ratio, any notion of "goodness" or "optimality" of $n_{1/2}$ should only be considered in the context of the underlying metrics α, β, γ, and h. For a more complete discussion of these parameters see [202, 200].

We measured latency and bandwidth on a number of different multiprocessors. Table 3.1 shows the measured latency, bandwidth, and $n_{1/2}$ for nearest neighbor communication. The table also includes the peak bandwidth as stated by the vendor. For comparison, typical data rates and latencies are reported for several local area network technologies.

Figure 3.4 graphically summarizes the communication performance of the various multiprocessors in a two-dimensional message-passing metric space. The upper-left region is the high performance area, lower performance and LAN networks occupy the lower performance region in the lower right.

In one way, the message-passing overheads illustrated in Table 3.6.3 are the Achilles heel of

Table 3.1: Multiprocessor Latency and Bandwidth.

Machine	OS	Latency $n = 0$ (μs)	Bandwidth $n = 10^6$ (MB/s)	$n_{1/2}$ bytes	Theoretical Bandwidth (MB/s)
Convex SPP1000 (PVM)	SPP-UX 3.0.4.1	76	11	1000	250
Convex SPP1000 (sm 1-n)	SPP-UX 3.0.4.1	2.5	82	1000	250
Convex SPP1000 (sm m-n)	SPP-UX 3.0.4.1	12	59	1000	250
Convex SPP1200 (PVM)	SPP-UX 3.0.4.1	63	15	1000	250
Convex SPP1200 (sm 1-n)	SPP-UX 3.0.4.1	2.2	92	1000	250
Convex SPP1200 (sm m-n)	SPP-UX 3.0.4.1	11	71	1000	250
Cray T3D (sm)	MAX 1.2.0.2	3	128	363	300
Cray T3D (PVM)	MAX 1.2.0.2	21	27	1502	300
Intel Paragon	OSF 1.0.4	29	154	7236	175
Intel Paragon	SUNMOS 1.6.2	25	171	5856	175
Intel Delta	NX 3.3.10	77	8	900	22
Intel iPSC/860	NX 3.3.2	65	3	340	3
Intel iPSC/2	NX 3.3.2	370	2.8	1742	3
IBM SP-1	MPL	270	7	1904	40
IBM SP-2	MPI	35	35	3263	40
KSR-1	OSF R1.2.2	73	8	635	32
Meiko CS2 (sm)	Solaris 2.3	11	40	285	50
Meiko CS2	Solaris 2.3	83	43	3559	50
nCUBE 2	Vertex 2.0	154	1.7	333	2.5
nCUBE 1	Vertex 2.3	384	0.4	148	1
NEC Cenju-3	Env. Rel 1.5d	40	13	900	40
NEC Cenju-3 (sm)	Env. Rel 1.5d	34	25	400	40
SGI	IRIX 6.1	10	64	799	1200
TMC CM-5	CMMD 2.0	95	9	962	10
Ethernet	TCP/IP	500	0.9	450	1.2
FDDI	TCP/IP	900	9.7	8730	12
ATM-100	TCP/IP	900	3.5	3150	12

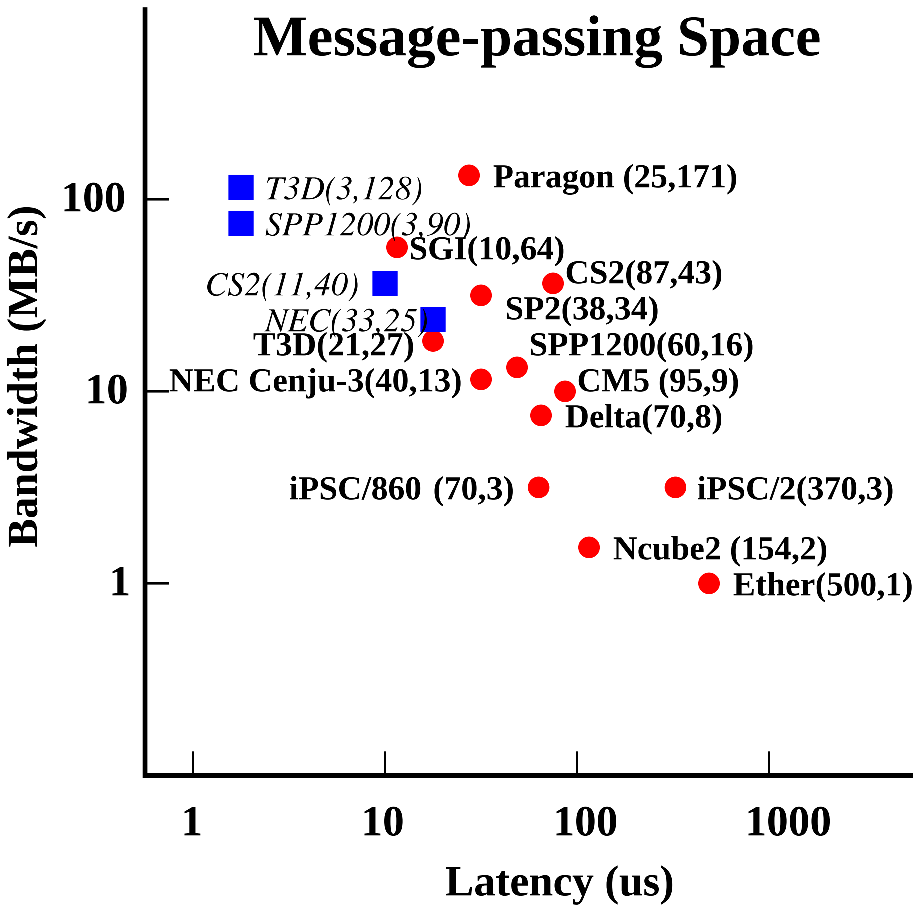

Figure 3.4: Latency/bandwidth space for 0-byte message (latency) and 1 MB message (bandwidth). Block points represent shared-memory copy performance.

the MIMD local-memory machines; in another, they serve to determine those very applications for which message-passing architectures are competitive.

In Table 3.2 we have converted the floating point execution rates observed for each problem to operations per cycle and also calculated the number of cycles consumed, as overhead (latency), during communication. For addition details see the paper [85].

Table 3.2: Message passing latency and computation performance

	Clock cycle		Linpack 100		Linpack 1000		Latency	
Machine / OS	MHz	(nsec)	Mflop/s	(flop/cl)	Mflop/s	(flop/cl)	us	(cl)
Convex SPP1000 (PVM)	100	(10)	48	(.48)	123	(1.23)	76	(7600)
Convex SPP1000 (sm 1-n)							2.6	(260)
Convex SPP1000 (sm m-n)							11	(1080)
Convex SPP1200 (PVM)	100	(8.33)	65	(.54)	123	(1.02)	63	(7560)
Convex SPP1200 (sm 1-n)							2.2	(264)
Convex SPP1200 (sm m-n)							11	(1260)
Cray T3D (sm) / MAX 1.2.0.2	150	(6.67)	38	(.25)	94	(.62)	3	(450)
Cray T3D (PVM)							21	(3150)
Intel Paragon / OSF 1.0.4	50	(20)	10	(.20)	34	(.68)	29	(1450)
Intel Paragon / SUNMOS 1.6.2							25	(1250)
Intel Delta / NX 3.3.10	40	(25)	9.8	(.25)	34	(.85)	77	(3080)
Intel iPSC/860 / NX 3.3.2	40	(25)	9.8	(.25)	34	(.85)	65	(2600)
Intel iPSC/2 / NX 3.3.2	16	(63)	.37	(.01)	–	(–)	370	(5920)
IBM SP-1 / MPL	62.5	(16)	38	(.61)	104	(1.66)	270	(16875)
IBM SP-2 / MPI	66	(15.15)	130	(1.97)	236	(3.58)	35	(2310)
KSR-1 / OSF R1.2.2	40	(25)	15	(.38)	31	(.78)	73	(2920)
Meiko CS2 (MPI) / Solaris 2.3	90	(11.11)	24	(.27)	97	(1.08)	83	(7470)
Meiko CS2 (sm)							11	(990)
nCUBE 2 / Vertex 2.0	20	(50)	.78	(.04)	2	(.10)	154	(3080)
nCUBE 1 / Vertex 2.3	8	(125)	.10	(.01)	–	(–)	384	(3072)
NEC Cenju-3 / Env Rev 1.5d	75	(13.3)	23	(.31)	39	(.52)	40	(3000)
NEC Cenju-3(sm) / Env Rev 1.5d	75	(13.3)	23	(.31)	39	(.52)	34	(2550)
SGI Power Challenge / IRIX 6.1	90	(11.11)	126	(1.4)	308	(3.42)	10	(900)
TMC CM-5 / CMMD 2.0	32	(31.25)	–	(–)	–	(–)	95	(3040)

3.6.1 Performance Prediction

Based on the parameters obtained for a specific computer system is it possible to pridict the performance of a given algorithm given the balance between computation and communication. Figure 3.5 shows the prediction of time spent performing computation and the time spent in communiction for performing LU factorization with partial pivoting on the IBM SP-2 based on the bandwidth and latency numbers. These predictions are within 10% of the actual observed performance for the software.

3.6.2 Message Passing Standards

Over the last few years there has been a number of efforts started to develop and standardize on a interface for parallel computing. Foremost in these efforts have been the HPF, PVM, and MPI activities.

The Message Passing Interface (MPI) is a portable message-passing standard that facilitates the development of parallel applications and libraries. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran 77 or C. MPI also forms a possible target for compilers of languages such as High Performance Fortran [219]. Commercial and free, public-domain implementations of MPI already exist. These run on both tightly-coupled, massively-parallel machines (MPPs), and on networks of workstations (NOWs).

The MPI standard was developed over a year of intensive meetings and involved over 80 people from approximately 40 organizations, mainly from the United States and Europe. Meeting attendance was open to the technical community. The meets where announced on various bulletin boards and mailing lists. MPI operated on a very tight budget (in reality, it had no budget when the first meeting was announced). The Advanced Research Projects Agency (ARPA) through the National Science Foundation (NSF) have provided partial travel support for the U.S. academic participants. Support for several European participants was provided by the European Commission through the ESPRIT project. Formal voting at the meetings was done by a single vote per organization. In order to vote, the organization must have had a representative at two of the last three meetings. In order to give guidance for preparation of formal proposals, informal votes where often taken involving everyone present. Many vendors of concurrent computers were involved, along with researchers from universities, government laboratories, and industry. This effort culminated in the publication of the MPI specification [157]. Other sources of information on MPI are available [253] or are under development.

Researchers incorporated into MPI the most useful features of several systems, rather than choosing one system to adopt as the standard. MPI has roots in PVM [86, 166], Express [256], P4 [47], Zipcode [282], and Parmacs [49], and in systems sold by IBM, Intel, Meiko, Cray Research, and Ncube.

MPI is used to specify the communication among a set of processes forming a concurrent program. The message-passing paradigm is attractive because of its wide portability and scalability. It is easily compatible with both distributed-memory multicomputers and shared-memory multi-

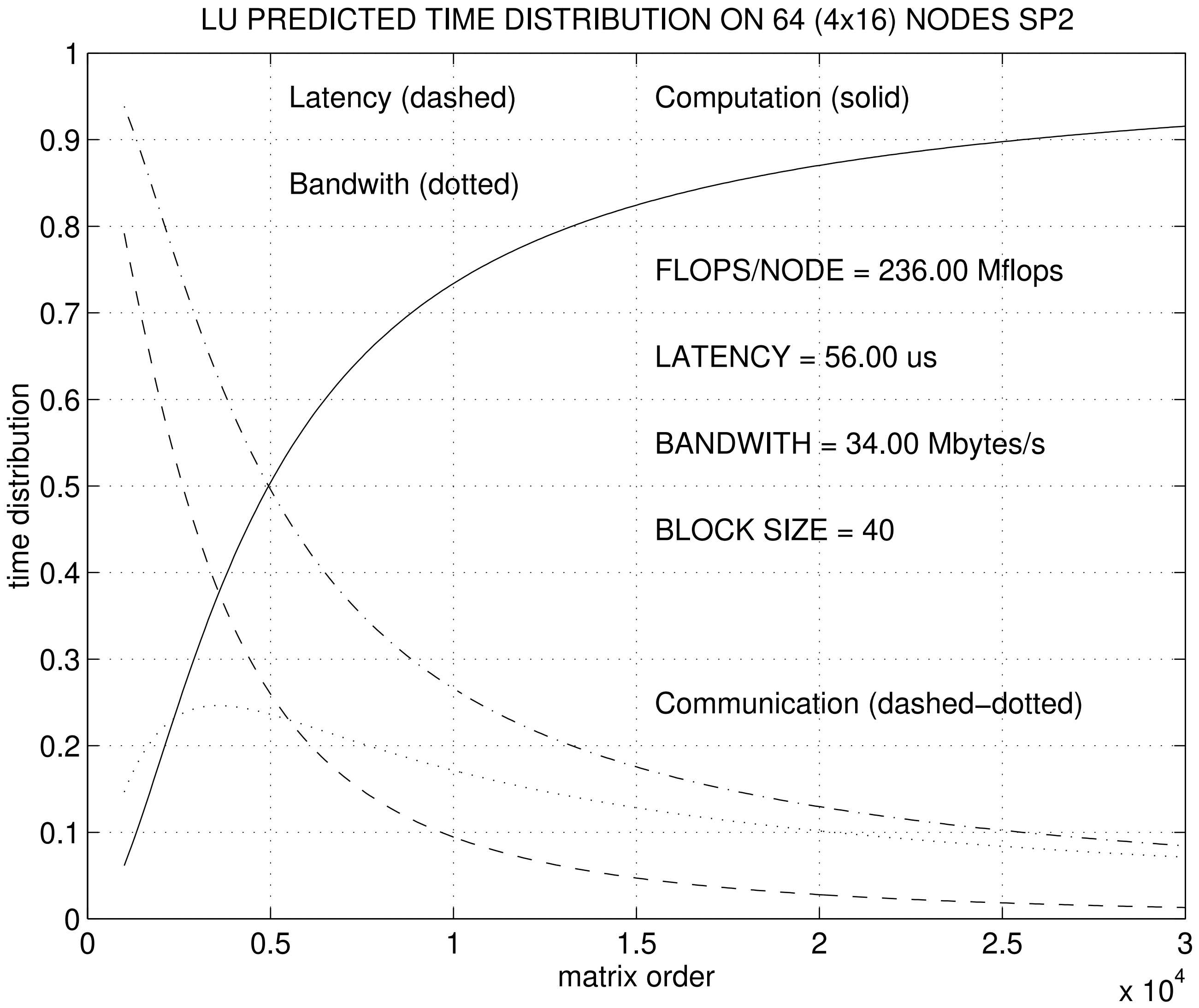

Figure 3.5: The prediction of computation and communication in LU decomposition

processors, NOWs, and combinations of these elements. Message passing will not be made obsolete by increases in network speeds or by architectures combining shared and distributed-memory components.

Though much of MPI serves to standardize the "common practice" of existing systems, MPI has gone further and defined advanced features such as user-defined datatypes, persistent communication ports, powerful collective communication operations, and scoping mechanisms for communication. No previous system incorporated all these features.

The standard specifies the form of the following.

- Point to point communications, that is, messages between pairs of processes.
- Collective communications: communication or synchronization operations that involve entire groups of processes.
- Process groups: how they are used and manipulated.
- Communicators: a mechanism for providing separate communication scopes for modules or libraries. Each communicator specifies a distinct name space for processes, a distinct communication context for messages and may carry additional, scope-specific information.
- Process topologies: functions that allow the convenient manipulation of process labels, when the processes are regarded as forming a particular topology, such as a Cartesian grid.
- Bindings for Fortran 77 and ANSI C: MPI was designed so that versions of it in both C and Fortran had straightforward syntax. In fact, the detailed form of the interface in these two languages is specified and is part of the standard.
- Profiling interface: the interface is designed so that runtime profiling or performance-monitoring tools can be joined to the message-passing system. It is not necessary to have access to the MPI source to do this and hence, portable profiling systems can be easily constructed.
- Environmental management and inquiry functions: these functions give a portable timer, some system-querying capabilities, and the ability to influence error behavior and error-handling functions.

A key feature needed to support the creation of robust, parallel libraries is to guarantee that communication within a library routine does not conflict with communication extraneous to the routine. The concepts encapsulated by an MPI communicator provide this support.

A **communicator** is a data object that specifies the scope of a communication operation, that is, the group of processes involved and the communication context. **Contexts** partition the communication space. A message sent in one context cannot be received in another context.

Communicators are especially important for the design of parallel software libraries. Suppose we have a parallel, matrix multiplication routine as a member of a library. We would like to allow distinct subgroups of processes to perform different matrix multiplications concurrently. A communicator provides a convenient mechanism for passing into the library routine the group of processes involved, and within the routine, process ranks will be interpreted relative to this group. The grouping and labeling mechanisms provided by communicators are useful, and communicators will typically be passed into library routines that perform internal communications.

Such library routines can also create their own, unique communicator for internal use. For example, consider an application in which process 0 posts a wildcarded, non-blocking receive just before entry to a library routine. Such "promiscuous" posting of receives is Such library routines can also create their own, unique communicator a common technique for increasing performance. Here, if an internal communicator is not created, incorrect behavior could result since the receive may be satisfied by a message sent by process 1 from within the library routine, if process 1 invokes the library ahead of process 0. Another example is one where a process sends a message before entry into a library routine, but the destination process does not post the matching receive until after exiting the library routine. In this case, the message may be received, incorrectly, within the library routine.

These problems are avoided by proper design and usage of parallel libraries. One workable design is for the application program to pass communicators into the library routine that specifies the group and ensures a safe context. Another design has the library create a "hidden" and unique communicator that is set up in a library initialization call, again leading to correct partitioning of the message space between application and library.

MPI does not claim to be the definitive answer to all needs. Indeed, MPI's insistence on simplicity and timeliness of the standard precludes that. The MPI interface provides a useful basis for the development of software for message-passing environments. Besides promoting the emergence of parallel software, a message-passing standard provides vendors with a clearly defined, base set of routines that they can implement efficiently. Hardware support for parts of the system is also possible, and this may greatly enhance parallel scalability.

PVM (Parallel Virtual Machine) [166] indexParallel Virtual Machine is a byproduct of an ongoing Heterogeneous Distributed Computing research project at Oak Ridge National Laboratory and the University of Tennessee. The PVM project is viewed be the developers as a research project that evolves over time. As such, the interface is updated to reflect new ideas and approaches to distributed computing. PVM has its origins in distributed heterogeneous computing. The general goals of this project are to investigate issues in, and develop solutions for, heterogeneous concurrent computing. PVM is an integrated set of software tools and libraries that emulates a general-purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architecture. The overall objective of the PVM system is to to enable such a collection of computers to be used cooperatively for concurrent or parallel computation.

The PVM system is composed of two parts. The first part is the system part which is handled through a daemon, called *pvmd3* and sometimes abbreviated *pvmd*, that resides on all the computers making up the virtual machine. (An example of a daemon program is the mail program that runs in the background and handles all the incoming and outgoing electronic mail on a computer.) Pvmd3 is designed so any user with a valid login can install this daemon on a machine. When a user wishes to run a PVM application, he first creates a virtual machine by starting up PVM. The PVM application can then be started from a Unix prompt on any of the hosts. Multiple users can configure overlapping virtual machines, and each user can execute several PVM applications simultaneously.

The second part of the system is a library of PVM interface routines. It contains a functionally complete repertoire of primitives that are needed for cooperation between tasks of an application. This library contains user-callable routines for message passing, spawning processes, coordinating tasks, and modifying the virtual machine. The PVM computing model is based on the notion that an application consists of several tasks. Each task is responsible for a part of the application's computational workload. Sometimes an application is parallelized along its functions; that is, each task performs a different function, for example, input, problem setup, solution, output, and display. This process is often called functional parallelism. A more common method of parallelizing an application is called data parallelism. In this method all the tasks are the same, but each one only knows and solves a small part of the data. This is also referred to as the SPMD (single-program multiple-data) model of computing. PVM supports either or a mixture of these methods. Depending on their functions, tasks may execute in parallel and may need to synchronize or exchange data, although this is not always the case.

The PVM software provides a unified framework within which parallel programs can be developed in an efficient and straightforward manner using existing hardware. PVM enables a collection of heterogeneous computer systems to be viewed as a single parallel virtual machine. PVM transparently handles all message routing, data conversion, and task scheduling across a network of incompatible computer architectures.

The PVM computing model is simple yet very general, and accommodates a wide variety of application program structures. The programming interface is deliberately straightforward, thus permitting simple program structures to be implemented in an intuitive manner. The user writes his application as a collection of cooperating *tasks*. Tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronization between tasks. The PVM message-passing primitives are oriented towards heterogeneous operation, involving strongly typed constructs for buffering and transmission. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast, barrier synchronization, and global sum.

PVM tasks may possess arbitrary control and dependency structures. In other words, at any point in the execution of a concurrent application, any task in existence may start or stop other tasks or add or delete computers from the virtual machine. Any process may communicate and/or synchronize with any other. Any specific control and dependency structure may be implemented under the PVM system by appropriate use of PVM constructs and host language control-flow statements.

Owing to its ubiquitous nature (specifically, the virtual machine concept) and also because of its simple but complete programming interface, the PVM system has gained widespread acceptance in the high-performance scientific computing community.

3.6.3 Routing

In a message-passing architecture a physical wire must exist to transmit the message. In any system with a large number of processors, it is not feasible for a wire to exist between every pair of processors. Not only would the wiring become unbearably complicated, but also the number of connections to a single processor would become too great. Thus, in all machines of which we are aware, there is either a global bus to which all processors are connected or a fixed topology between the processors.

The bus connection is more common in a shared-memory architecture where the memory is attached to the bus in much the same way as each processor. There the routing is simple; each processor merely accesses the shared memory directly through the bus. The main problem one can have is bus saturation, because the bus is commonly much slower than the processors and thus can be easily overloaded with data requests. In bus-connected local-memory machines, bus contention is again a problem, but routing is simple and the message-passing model is essentially that of the previous section.

In local-memory architectures, the fixed topology is usually a two-dimensional mesh (more commonly, the ends are wrapped to form a torus) or a hypercube (see Section 1.4). The hypercube is particularly rich in routing paths and has the important property that any two nodes in a k-dimensional hypercube (that is, a hypercube with 2^k nodes) are at most a distance k apart. If a binary representation is used to label the nodes of the hypercube, then direct connections can exist between two nodes whose representation differs in only one bit. A possible routing scheme is to move from one processor to another by changing one bit at a time as necessary and sending the message via the intermediate nodes so identified.

Since intermediate nodes during the routing of a message do nothing except route the message to the next node, it is important that they can do this rerouting without interrupting any work in

Table 3.3: Transfer Rates for Message Passing on Hypercubes (kB/sec)

Message	Adjacent nodes		5 hops	
Length	64 bytes	4096 bytes	64 bytes	4096 bytes
iPSC/1	56	501	18	322
iPSC/2	160	1888	145	1788
iPSC/860	640	2424	457	2269
NCUBE	116	374	41	141

progress at that intermediate node. The effect of this design is felt strongly in the three versions of the Intel iPSC, where the first version did not have transparent routing. As the data in Table 3.6.3 show, the transfer rate for sending messages to non-neighboring nodes in the hypercube is considerably reduced in the iPSC/2 although the startup time is still significant [138].

Chapter 4

Performance: Analysis, Modeling, and Measurements

The computational speed of modern architectures is usually expressed in terms of Mflops. For the most powerful of today's supercomputers, even Gigaflops are used to describe their high-performance (1 Gflop = 1000 Mflops). From the definition of Mflops it follows that when N floating-point operations (flops) are carried out in t microseconds (10^{-6} seconds), the speed of computation is given by

$$r = \frac{N}{t} \text{ Mflops}, \tag{4.1}$$

and, vice versa, when N flops are executed with an average computing speed of r Mflops, then the CPU time is

$$t = \frac{N}{r} \text{ microsec.} \tag{4.2}$$

Different operations may be executed at quite different speeds, depending, for example, on the advantage taken of pipelining possibilities, references to memory, etc. This fact implies that we cannot simply analyze the performance of an algorithm by counting the number of flops to be executed (as used to be a good indication for classical computers). Instead, we must take into account the fact that specific parts of an algorithm may exploit the opportunities offered by a given architecture. In this chapter, we describe the principles for analyzing and modeling the performance of computational work to help predict, understand, and guide the software/program development.

4.1 Amdahl's Law

In most realistic situations not all operations in a complex algorithm are executable with high computational speeds. When two parts of a job are executed each at a different speed, the total CPU time can be expressed as a function of these speeds. It turns out that the lower of these speeds may have a dramatic influence on the overall performance. This effect was first pointed out by Amdahl [6], and therefore the relation between performance (computational speed) and CPU time is often called *Amdahl's law*.

4.1.1 Simple Case of Amdahl's Law

We first discuss a simple case of Amdahl's law. Let us assume that the execution of a given algorithm involves N flops and that on a certain computer a fraction f of these flops is carried out with a speed of V Mflops, while the rest is executed at a rate of S Mflops. Let us further assume that this fraction f is well suited to the given architecture, while the other part does not take much advantage of speedup possibilities ($V \gg S$). The total CPU time t, according to (4.2), is expressed by

$$t = \frac{fN}{V} + \frac{(1-f)N}{S} = N(\frac{f}{V} + \frac{1-f}{S}) \text{ microsec}, \tag{4.3}$$

and for the overall performance r we have from (4.1) that

$$r = \frac{N}{t} = \frac{1}{f/V + (1-f)/S} \text{ Mflops (Amdahl's law)}. \tag{4.4}$$

From (4.3) we see that

$$t > \frac{(1-f)N}{S} \text{ microsec}. \tag{4.5}$$

If the algorithm had executed completely with the lower speed S, the CPU time would have been $t = N/S$ microseconds. This implies that the relative gain in CPU time obtained from executing the portion fN at the speed V instead of the much lower speed S is bounded by $\frac{1}{1-f}$. Thus, f must be rather close to 1 in order to benefit significantly from high computational speeds.

This effect is nicely illustrated by Figure 4.1, where we display the overall performance r as a function of f for the situation in which $V = 130$ Mflops and $S = 4$ Mflops. These numbers represent a realistic situation for the CRAY-1 (on more recent computers the ratio $\frac{V}{S}$ can be much larger, depending on the type of operation).

We see that f has to be quite large—say, larger than 0.8 (or 80%)—in order to obtain significant improvements in the overall speed. Since we usually encounter a whole range of speeds for the

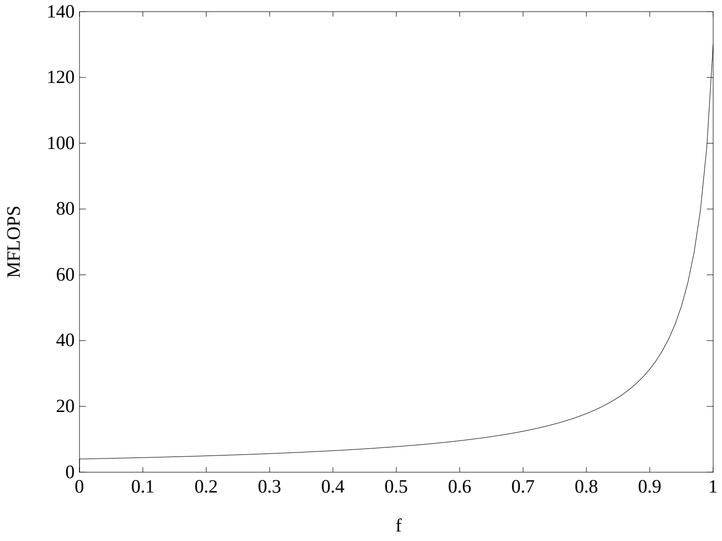

Figure 4.1: **Amdahl's law for the performance** r

different parts of an algorithm, we also give a more general form of Amdahl's law in order to facilitate the analysis of realistic situations.

4.1.2 General Form of Amdahl's Law

Assume that the execution of an algorithm consists of executing the consecutive parts $A_1, A_2, ..., A_n$ such that the N_j flops of part A_j are executed at a speed of r_j Mflops. Then the overall performance r for the $N = N_1 + N_2 + ... + N_n$ flops is given by

$$r = \frac{N}{(N_1/r_1 + N_2/r_2 + ... + N_n/r_n)} \text{ Mflops.} \tag{4.6}$$

This follows directly from the observation that the CPU time t_j for the execution of part A_j is given by $t_j = \frac{N_j}{r_j}$ microsec. Hence the total time for the algorithm is the sum of these times, and (4.1) gives the desired result.

With Amdahl's law we can study the effect of different speeds for separate parts of a computational task. However, this is not the end of the story. Previously we have seen that improvements in the execution of computations usually are obtained only after some startup time. This fact implies that the computational speed depends strongly on the amount (and type) of work. In Section 4.2 we shall discuss this effect for pipelined operations.

4.2 Vector Speed and Vector Length

We have seen in Section 1.2.2 that operations that can be pipelined—the so-called vector operations– require a certain startup time before the first results are delivered. Once the pipeline produces results, the subsequent results are delivered at much shorter intervals. Hence the average computing speed for long vectors may be rather large, but for very short vectors the effect of pipelining may be barely noticeable.

The performance r_N for a given loop of length N is usually expressed, after Hockney and Jesshope [198], in terms of parameters r_∞ and $n_{1/2}$. The values of these parameters are characteristic for a specific loop: they represent the performance in Mflops for very long loop length (r_∞), and the loop length for which a performance of about $1/2r_\infty$ Mflops is achieved ($n_{1/2}$).
For many situations we have in fairly good approximation that

$$r_N = \frac{r_\infty}{n_{1/2}/N + 1} \text{ Mflops.} \tag{4.7}$$

The actually observed Mflops rates may differ slightly from the model, since (4.7) does not account for stripmining effects (see Section 1.2.10) or memory bank conflicts (see Section 1.3.1).

It is straightforward from (4.7) that for loop length $N = qn_{1/2}$ a performance of $r_\infty q/(1+q)$ Mflops will be obtained. Consequently, if in a given application the loop length is N, then it is highly desirable that the $n_{1/2}$-values concerned be small relative to N.

As a rule of thumb we have that $n_{1/2}$ and r_∞ are proportional to the number of pipelines of a given computer. If a model comes with different numbers of pipelines, then of course the r_∞-values for the largest model are highest, but it will also require larger vector lengths to obtain reasonable performances.

The same holds on a p-processor parallel computer when parallelism in a code is realized through splitting the loops over the processors. For many loops this approach will lead to an increase of r_∞ and $n_{1/2}$ by a factor of p relative to single-processor execution. In general, however, on a p-processor system r_∞ grows less than with p, and $n_{1/2}$ grows faster, because of synchronization effects. Especially in sparse matrix codes, parallelism is often obtained through loop splitting; and hence we must have fairly large vector lengths for high-performance execution on a parallel multiple-pipeline processor system (as, e.g., the NEC SX-3).

Our modeling approach for the performance of parallel vector systems has been simplified in that we have not taken into account the effects of memory hierarchy (e.g., cache memory). For computational models that include these effects see, e.g., [162, 201].

4.3 Amdahl's Law—Parallel Processing

Pipelining is only one way to help improve performance. Another way is to execute parts of a job in parallel on different processors. Of course, these processors themselves may be vector processors, a situation that further complicates the analysis.

4.3.1 A Simple Model

In parallel processing the total CPU time is usually larger than the CPU time involved if the computation had taken place on a single processor, and it is much larger than the CPU time spent per processor. Since the goal of parallel processing is to reduce the wall clock time, we shall compare the wall clock times for different numbers of processors in order to study the accelerating effect of parallel processing.

It may be expected that if the computations can be carried out completely in p equal parallel parts, the wall clock time will be nearly $1/p$ of that for execution on only one processor. As is already clear from Amdahl's law, the non-parallel (or serial) parts may have a negative influence on this reduction. We shall make this more explicit.

Let t_j denote the wall clock time to execute a given job on j parallel processors. The speedup, S_p, for a system of p processors is then by definition

$$S_p = \frac{t_1}{t_p}. \tag{4.8}$$

The efficiency, E_p, of the p-processor system is defined by

$$E_p = \frac{S_p}{p}. \tag{4.9}$$

The value of E_p expresses, in a relative sense, how busy the processors are kept. Evidently $0 \leq E_p \leq 1$.

Let us assume for simplicity that a job consists of basic operations all carried out with the same computational speed and that a fraction f of these operations can be carried out in parallel on p processors, while the rest of the work can keep only 1 processor busy. The wall clock time for the parallel part is then given by $\frac{ft_1}{p}$; and, if we ignore all synchronization overhead, the time for the serial part is given by $(1-f)t_1$. Consequently, the total time for the p-processor system is

$$t_p = \frac{ft_1}{p} + (1-f)t_1 = \frac{t_1(f+(1-f)p)}{p} \geq (1-f)t_1. \tag{4.10}$$

From this the speedup S_p directly follows:

$$S_p = \frac{p}{(f+(1-f)p)}. \tag{4.11}$$

Note that $f < 1$ and the speedup S_p is bounded by $S_p \leq \frac{1}{1-f}$. Relation (4.11) is often referred to as *Ware's law* [323].

These relations show that the speedup S_p is reduced by a factor $f + (1-f)p = O(p)$, which leads to considerable reductions for increasing values of p, even for f very close to 1. For example, when 90% of the work can be done entirely in parallel on 10 processors, then the speedup is only about 5 (instead of the desired value 10), and the speedup for very large p is limited by 10.

The model (4.11) represents a highly idealized and simplified situation. In practice, matters are usually complicated by factors such as the following:

- The number of parallel portions of work has to be large enough to match a given number of processors.
- The (negative) influence of synchronization overhead for the parallel execution has been ignored.
- Often one has to modify part of the underlying algorithm in order to obtain a sufficient degree of parallelism, a procedure that in many situations leads to increasing computational complexity.

Nevertheless, the simple model is useful since it helps us to predict upper bounds for the speedup. In Table 4.1 we list the speedup for several values of the parallelizable fraction f and the number of processors p, under the assumption that the parallel fraction can be executed entirely in parallel, without any overhead, on p processors.

For example, suppose that we have observed a speedup $S_4 = 3.77$ on a four-processor system for a certain job. Assume that this job can be seen as being composed of a serial part and a parallel part, suitable for nicely distributed parallel processing on at least four processors. It follows from Table 4.1 (and the equation (4.11)) that the fraction of parallel work is about 98%. This does not look too bad, but if this job could also be executed (without any additional overhead or complications) on 16 processors, then the speedup would have been 12.31 at most.

Table 4.1: Speedups for Values of f and p

f	p=1	p=2	p=3	p=4	p=8	p=16	p=32	p=64	$p=\infty$
1.00	1.00	2.00	3.00	4.00	8.00	16.00	32.00	64.00	∞
.99	1.00	1.98	2.94	3.88	7.48	13.91	24.43	39.26	100.00
.98	1.00	1.96	2.88	3.77	7.02	12.31	19.75	28.32	50.00
.96	1.00	1.92	2.78	3.57	6.25	10.00	14.29	18.18	25.00
.94	1.00	1.89	2.68	3.39	5.63	8.42	11.19	13.39	16.67
.92	1.00	1.85	2.59	3.23	5.13	7.27	9.19	10.60	12.50
.90	1.00	1.82	2.50	3.08	4.71	6.40	7.80	8.77	10.00
.75	1.00	1.60	2.00	2.28	2.91	3.37	3.66	3.82	4.00
.50	1.00	1.33	1.50	1.60	1.78	1.88	1.94	1.97	2.00
.25	1.00	1.14	1.20	1.23	1.28	1.31	1.32	1.33	1.33
.10	1.00	1.05	1.07	1.08	1.09	1.10	1.11	1.11	1.11
.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

In fact, we conclude from Table 4.1 that even for this apparently highly parallel job, the speedup soon falls well behind the number of processors. In most practical situations it is quite unrealistic to assume that a given job consists of only two parts, a strictly serial part and one that can execute in parallel on p processors. Often, some parts can be executed in parallel on p processors while other parts can still be executed in parallel but on fewer than p processors.

Moler [243] properly observes that an increase in the number of processors usually goes along with an increase in the problem size, and he introduces the notion of being "*effectively parallel*" to go along with this. (See also Gustafson's model in Section 4.3.1.) Let the non-parallel part of the computations be denoted by $\alpha(M) = 1 - f$, where M is a measure of the total number of operations. Moler reports that for many algorithms, $\alpha(M)$ decreases when M is increased. In line with this, an "effectively parallel algorithm" is defined as an algorithm for which $\alpha(M)$ goes to zero when M grows.

If the parallel part can be executed in parallel on q processors, then it follows for such an algorithm that

$$S_q = \frac{q}{1+(q-1)a(M)} \text{ and } E_q = \frac{1}{1+(q-1)a(M)} \tag{4.12}$$

and hence S_q grows to q and E_q grows to 1 for growing M.

Now the question arises whether (local) memory is large enough to handle a problem size for which S_q is sufficiently close to q. Moler notes that in many situations memory size is a bottleneck in realizing large speedups, simply because it often limits the problem size.

4.3.2 Gustafson's Model

The basic premise of Amdahl's law is that the proportion of code that can be vectorized or parallelized remains constant as vector length (that is, problem size) increases. As Gustafson [186] pointed out, this is often not appropriate. An alternative model can be based on fixing the computation time rather than the problem size, as f varies. Thus, in Gustafson's model, if we assume that the problem can be solved in one unit of time on our parallel machine and that the time in serial computation is $1 - f$ and in parallel computation is f, then if there are p processors in our parallel system, the time for a uniprocessor to perform the same job would be $1 - f + fp$. Then the speedup, $S_{p,f}$, is given by

$$S_{p,f} = p + (1 - p)(1 - f).$$

We see that, as the number of processors increases, so does the proportion of work in parallel mode, and speedup is not limited in the same way as in Amdahl's law. By using this measure, the Sandia team [187] obtained near-optimal speedups on an NCube machine to win the first Bell Prize in 1988. (Gordon Bell established an award in 1987 to demonstrate his personal commitment to parallel processing. The prize consists of two $1,000 awards to be given annually for ten years for the best scientific or engineering program with the greatest speedup [99].)

4.4 Examples of $(r_\infty, n_{1/2})$-values for Various Computers

As we have seen in the preceding sections, the performance for a given (vector) operation depends on the architecture of the computer and on the ability of its Fortran compiler to generate efficient code. From a modeling point of view the length of the clock cycle is of no importance, since it is only a scaling factor for r_∞. The factors that really make computers behave differently include features such as direct memory access, numbers of paths to memory, the bandwidth, memory hierarchy, and chaining.

To give some idea of the impact of these architectural aspects on the performance, we report on our own measurements of the performance parameters for widely different architectures. In this section, we consider rather simple Fortran DO-loops, for which we measured the CPU times for different vector lengths (see Table 4.4). By a simple linear least-squares approximation, we determine the values of r_∞ and $n_{1/2}$ for a number of supercomputers and mini-supercomputers. In

Table 4.2: Various Simple Fortran DO-Loops

Number of the Loop	Number of Flops per Pass (n)	Operation per Pass of the Loop	Operation
1	2	v1(i)=v1(i)+a*v2(i)	update
2	8	v1(i)=v1(i)+s1*v2(i)+s2*v3(i)+ s3*v4(i)+ s4*v5(i)	4-fold vector update
3	1	v1(i)=v2(i)/v3(i)	divide
4	2	v1(i)=v1(i)+s*v2(ind(i))	update+gather
5	2	v1(i)=v2(i)-v3(i)*v1(i-1)	bidiagonal system
6	2	s=s+v1(i)*v2(i)	innerproduct

Section 4.5, we shall present performance measurements for a more complicated algorithm (taken from LINPACK).

We have deliberately restricted ourselves to this special selection because they play a role in our algorithms:

- Loop 1 is the well-known vector update, which is central in standard Gaussian elimination and in updating of current iteration vectors in iterative processes. Therefore we will see this loop again in Chapters 5 and 7.

- Loop 2 represents the combined effect of 4 vector updates, often referred to as a GAXPY. The performances for this loop, compared to those for Loop 1, show how an architecture (or compiler) can improve performance by, e.g., keeping $v1$ in the registers. This capability is fully exploited in the so-called Level 2 BLAS approach (Chapter 5). Hence, if the performances for Loop 2 are markedly better than those for Loop 1, we may expect big improvements by using Level 2 BLAS kernels instead of Level 1 BLAS kernels (this point will be further explained and highlighted in Chapter 5).

- Loop 3 has been included to demonstrate how slow a full-precision divide operation can be. Divide operations, which occur quite naturally in matrix computations (e.g., scaling), often can be largely replaced by multiply operations with the only once-computed inverse elements.

- Loop 4 is quite representative for direct solvers for sparse matrix systems. It also indicates the speed-reducing effects of indirect addressing, which is also typical of sparse matrix computations. Loops of this type will be encountered in Chapter 6.

Table 4.3: Performance of CRAY J90 and CRAY T90

	CRAY J90 CFT77 V???		CRAY T90 (1 proc.) CFT77 V???	
Loop Number	r_∞	$n_{1/2}$	r_∞	$n_{1/2}$
1	97	19	1428	159
2	163	29	1245	50
3	21	6	219	43
4	72	21	780	83
5	4.9	2	25	9
6	120	202	474	164

- Loop 5 is often seen in the solution of (block) tridiagonal systems, as well as in some preconditioning operations (this will be explained in Chapter 7). It also shows to some extent how slow scalar operations on a vector computer can be.
- Loop 6 is frequently used in iterative processes for the computation of iteration parameters and for stopping criteria (Chapter 7).

The reader who wishes to determine the $(r_\infty, n_{1/2})$-values for other loops, or in other circumstances, may do so by using a code that we have made available in *netlib* (see Appendix A).

4.4.1 CRAY J90 and CRAY T90 (one processor)

Knowing only the clock cycle times, 10 nsec for the J90 and 2.2 nsec for the T90, one would expect the CRAY T90 to be about 4.5 times faster than the CRAY J90. We see in Table 4.3 that this is not the case, except for indirect addressing constructions and inner products. The main reason that the CRAY T90 is fast (in Fortran!) is that the hardware has multiple pipelines allowing 4 floating point operations to complete per cycle instead of 2 floating point operations. It is, of course, possible to write more efficient code in Cray assembly language (CAL) for many of the above operations.

4.4.2 CRAY X-MP (one processor; clock cycle time 8.5 nsec)

To demonstrate the progress in compilers and the dependence of performance measurements on the quality of the Fortran compilers, we give performance parameters for two different compilers on the CRAY X-MP.

Table 4.4: Performance of CRAY X-MP (one processor)

	CFT 1.15 BF2		CFT77 V2.0*207	
Loop Number	r_∞	$n_{1/2}$	r_∞	$n_{1/2}$
1	166	84	166	29
2	202	30	207	13
3	26	24	31	16
4	82	44	94	19
5	5.7	1	15.3	13
6	166	292	206	286
7	178	27	174	16

We see from Table 4.4 that the r_∞ value has improved for several loops, but most impressive is the reduction of the $n_{1/2}$ value for most loops. This means that real supercomputing speeds are often obtained for very modest vector lengths N. Other remarkable points are that the CRAY X-MP architecture leads to a performance close to the peak performance (235 Mflops) in Fortran for some important loops, in contrast with the CRAY-1 or CRAY-2 for which it is not so easy to approach peak performance. Note also that indirect addressing leads to a reduction in speed only by a factor of roughly two (this also in contrast with the CRAY-1 and many other supercomputers).

4.4.3 CYBER 205 (2-pipe) and ETA-10P (single processor)

The CYBER 205 computers were superseded by the ETA-10 computers. It is therefore of interest to see how the designers of the architecture attempted to improve on the older 205 models. In comparing the data (see Table 4.5), one must realize that clock cycles are different for the tested machines (CYBER 205: 20 nsec, ETA-10P: 24 nsec), so their r_∞ values must be compared in a relative sense. Both machines have been tested with the same Fortran compiler: FORTRAN200 - level 678.

Table 4.5: **Performance of CYBER 205 and ETA-10P**

Loop	CYBER 205		ETA-10P	
Number	r_∞	$n_{1/2}$	r_∞	$n_{1/2}$
1	200	170	167	140
2	200	160	167	55
3	16	20	13.6	21
4	30	46	44	86
5	2.9	1	1.6	1
6	100	162	83.3	243
7	100	96	83.3	70

For both machines the peak performance is actually achieved for some loops, because of lack of stripmining effects. Note the more favorable performance of the ETA-10P with respect to indirect addressing.

For very simple loops, such as Loop 6, we see that the scalar loop overhead leads to a rather large $n_{1/2}$. Actually, performance results as represented above might inspire us to rather misleading conclusions, since for more complicated problems, such as Loop 2, we see that the ETA-10P outperforms the CYBER 205 even for smaller vector lengths. The explanation is that for more complicated loops (which comprise more vector operations), the scalar overhead is much less than the sum of the overheads for each of the individual vector operations. Furthermore, there seems to be more opportunity to overlap this scalar overhead with the vector operations.

4.4.4 IBM 3090/VF (1 processor; clock cycle time 18.5 nsec)

Each processor of the IBM 3090 has a relatively small cache memory, 64 kilobytes. The performance parameters were determined for vector operands that were not available in cache at the moment of starting the vector operations. The timing experiments reported in Table 4.6 were carried out with the Fortran compiler FORTVS2 (as of December 10, 1988).

Table 4.6: **Performance of IBM 3090/VF (1 processor)**

Loop Number	r_∞	$n_{1/2}$	$r_{\max}(cache)$
1	21	46	30
2	41	42	61
3	2.7	16	2.9
4	11	34	16
5	6.5	8	8.8
6	29	74	45
7	27	37	38

In many dense matrix algorithms, it is possible to reuse intermediate data stored in cache memory. In that case, the performance may be much better, as is shown in the right half of Table 4.6 in the column labeled $r_{\max}(cache)$. This column gives the maximum performances for operations for which all the operands happen to be in cache. The size of the cache and the type of operation determine the maximum vector length of the operations; for example, for the vector update, the maximum vector length is about 1500, while for the sparse matrix-vector product, the maximum vector length is about 500 (with the size of cache on this machine). For sparse matrix computations it will, in general, be difficult to use the cache efficiently, especially when indirect addressing is involved.

4.4.5 NEC SX/2

The NEC SX/2 has a cache memory for scalar operands. This hardly influences the performance for most vector operations. Also in this case we have assumed that, at the start of the vector loop, scalar data such as addressing information was not available in cache. The Fortran compiler FORTRAN 77/SX, Rev. 036, has been used for the timing experiments.

Table 4.7: **Performance on the NEC SX/2**

Loop Number	r_∞	$n_{1/2}$
1	548	148
2	900	108
3	61	50
4	35	30
5	21	8
6	905	607
7	843	106

As Table 4.7 shows, the performance is relatively poor when indirect addressing is involved; the performance for the update is reduced by a factor of about 16.

Note also the rather modest performance for the vector divide. For the vector-vector multiply the values are $r_\infty = 270$ Mflops and $n_{1/2} = 151$. Clearly, it pays to invert the vectors if they are required more than once in vector-divide operations.

4.4.6 Convex C-1 and Convex C-210

The Convex C-1 and C-210 computers represent two generations of mini-supercomputers and therefore can be used to show the progress that has been made. Both machines were tested under the same Fortran 77 compiler (as of December 9, 1988).

Table 4.8: **Performance of the Convex C-1 and C-210**

Loop	Convex C-1		Convex C-210	
Number	r_∞	$n_{1/2}$	r_∞	$n_{1/2}$
1	6.0	14	16	26
2	10.7	22	23	15
3	1.8	13	4.0	1
4	4.0	12	9.0	20
5	.87	1	1.5	1
6	5.9	28	18	36
7	7.0	17	13	22

The r_∞ values for the C-210 (shown in Table 4.8) nicely reflect the reduced cycle time c with respect to the C-1 ($c = 100$ nanoseconds for the C-1 and $c = 40$ nanoseconds for the C-210).

4.4.7 Alliant FX/80

An Alliant FX/80 may be equipped with up to 8 processors (ACEs). Table 4.9 gives timing results collected for different numbers of ACEs. The compiler FORTRAN V3.1.33 (D.20D52) was used for all the timings.

Table 4.9: **Performance of Alliant FX/80**

Loop Number	1 ACE r_∞	1 ACE $n_{1/2}$	2 ACEs r_∞	2 ACEs $n_{1/2}$	4 ACEs r_∞	4 ACEs $n_{1/2}$	6 ACEs r_∞	6 ACEs $n_{1/2}$
1	3.45	18	6.9	55	13.8	96	17.1	117
2	5.6	6	12.2	25	23.0	47	32.6	74
3	1.1	11	2.2	29	4.3	59	6.4	94
4	2.8	15	4.5	28	9.75	71	10.85	77
5	1.6	22	1.6	22	2.14	43	3.0	87
6	5.1	31	10.2	78	20.4	160	30.5	254
7	3.6	5	6.2	8	13.3	26	16.8	31

Before analyzing the results, we note the following points:

- The timings for 1 ACE have been obtained with other compiler options, including switching parallelism off to make the code for 1 ACE a little more efficient. The timings for multiple ACEs have been obtained by explicitly parallel code. Hence, it becomes difficult to compare the results for 1 ACE with those for multiple ACEs.
- No special care has been taken to let the loop lengths be exact multiples of the numbers of ACEs. Occasionally, therefore, the performance is slightly degraded; and hence the least-squares fit for r_∞ leads to slightly less than optimal values.
- Because of the cache, we sometimes experience degradation in performance, especially for relatively simple loops (see the performance for the LINPACK code in Section 4.5.3). For more complicated DO-loops (e.g., Loop 2), the cache can be better exploited. This may partially explain irregularities in the performance parameters for multiple ACEs.

From the results in Table 4.9 we conclude that, indeed, in many cases r_∞ is proportional to the number of ACEs. We also see that, at least for some loops, the $n_{1/2}$-values for multiple ACEs grow roughly linearly with the number of ACEs (e.g., Loops 2, 3, and 6). In those cases the synchronization overhead apparently is small.

Finally, we note that the performance for combined updates, as in Loop 2, is much better than for a single update (Loop 1); hence the Level 2 BLAS approach (Section 5.1.2) is an improvement. A closer look at our experiments reveals that the performance for such combined loops is even better for loop lengths that fit the cache size; this can be realized through the use of the Level 3 BLAS (Section 5.1.3).

4.4.8 General Observations

From the preceding tables we conclude that, for most architectures, the simple DO-loops do not lead to anything close to peak performance. Only those machines capable of moving at least three vectors concurrently to and from main memory achieve peak performance for some of the loops (e.g., CRAY X-MP, CRAY Y-MP, ETA, and CYBER).

An additional bottleneck is caused by memory hierarchy that involves a cache (e.g., on the IBM and Alliant systems). This leads to a further degradation in the actual performance in many cases.

One should not conclude from our findings that it is impossible to come close to peak performance for important algorithms on all those machines. In fact, our results indicate that we have to look for possibilities of combining loops, in order to save on memory transfer. We discuss loop combining further in Chapter 5, where we show what can be done by using a set of operations over two and three loops.

We note, however, that in sparse matrix applications we are often restricted in the possibilities for loop combining. Then the $(r_\infty, n_{1/2})$-parameters may help to understand or model the performance of more complex algorithms.

4.5 LINPACK Benchmark

Simple DO-loops, like those in Section 4.4, form the backbone for the LINPACK subroutines. We therefore include some discussion on how such more complicated subroutines behave on vector and parallel architectures.

As we shall see, subroutines that merely use vector operations provide hardly any improvement over a simple DO-loop when executing on a machine with a cache memory. To avoid the bottlenecks inherent in such a memory hierarchy, we need the Level 2 or Level 3 BLAS approach, which will be treated in detail in Chapter 5.

4.5.1 Description of the Benchmark

Before continuing, we note here the distinction between LINPACK [90] and the so-called LINPACK benchmark. LINPACK is a collection of subroutines to solve various systems of simultaneous linear algebraic equations. The package was designed to be machine independent and fully portable and to run efficiently in many operating environments. The LINPACK benchmark [89] measures the performance of two routines from the LINPACK collection of software. These routines are

DGEFA and DGESL (these are double-precision versions; SGEFA and SGESL are their single-precision counterparts).

DGEFA performs the LU decomposition with partial pivoting, and DGESL uses that decomposition to solve the given system of linear equations. Most of the time is spent in DGEFA. Once the matrix has been decomposed, DGESL is used to find the solution; this process requires $O(n^2)$ floating-point operations, as opposed to the $O(n^3)$ floating-point operations of DGEFA.

4.5.2 Calls to the BLAS

DGEFA and DGESL in turn call three routines from the Basic Linear Algebra Subprograms, or BLAS [225]: IDAMAX, DSCAL, and DAXPY.

While the LINPACK routines DGEFA and DGESL involve two-dimensional array references, the BLAS refer to one-dimensional arrays. The LINPACK routines in general have been organized to access two-dimensional arrays by column. In DGEFA, for example, the call to DAXPY passes an address within the two-dimensional array A, which is then treated as a one-dimensional reference within DAXPY. Since the indexing is *down* a column of the two-dimensional array, the references to the one-dimensional array are sequential with unit stride. This is a performance enhancement over, say, addressing along the row of a two-dimensional array. Since Fortran dictates that two-dimensional arrays are stored by columns in memory, accesses to consecutive elements of a column lead to simple index calculations. References to consecutive elements differ by one word instead of by the leading dimension of the two-dimensional array.

4.5.3 Asymptotic Performance

We have run the LINPACK benchmark on a number of different machines and have collected data for various orders of matrices. This information can be used to give measures of $n_{1/2}$ and r_∞ for computing the matrix solution with the LINPACK programs, as follows.

We know that the algorithm uses $2/3n^3 + n^2 + O(n)$ operations for a matrix of order n. We also know that the time to complete the problem has the form $c_3n^3 + c_2n^2 + c_1n$. Hence, we can describe the rate of execution as the ratio of the operations performed to the time required to perform those operations, or

$$rate = \frac{2/3n^3 + n^2 + O(n)}{c_3n^3 + c_2n^2 + c_1n}. \tag{4.13}$$

We can compute the coefficients c_1, c_2, and c_3 through a least-squares fit of the data. From these coefficients we can then predict the asymptotic rate. At $n = \infty$ we have

$$r_\infty = \frac{2}{3c_3}. \tag{4.14}$$

By examining the point at which we reach half the asymptotic rate, we can determine $n_{1/2}$,

$$\frac{r_\infty}{2} = \frac{1}{3c_3} = \frac{2/3n^3 + \ldots}{c_3 n^3 + c_2 n^2 + \ldots}. \tag{4.15}$$

Solving for n, we determine that

$$n_{1/2} \approx c_2/c_3. \tag{4.16}$$

We compute the coefficients r_∞ and $n_{1/2}$ from the data. We then plot the data as well as the model and indicated values for r_∞ and $n_{1/2}$. In the following figures, the estimated asymptotic rate r_∞ is the horizontal dotted line, and the $n_{1/2}$ is the vertical dashed line. The circles represent the measured performance data for the LINPACK benchmark programs. The solid curve of Mflops is based on the model.

CRAY X-MP

The performance for the CRAY X-MP is an example of typical performance for a vectorizable algorithm over a range of problem sizes (see Figure 4.5.3). The performance for small problems is low; and as the size of the problem increases, the performance increases until it reaches its asymptotic rate.

Alliant

The performance of the Alliant FX/8 provides an interesting insight into the use of memory hierarchy. As the matrix size increases from order 1, we see the expected trend of increasing performance. When the size of the matrix exceeds 200, however, the performance decreases and eventually settles down and reaches an asymptotic rate less than the rate of a matrix of order 200. This result appears surprising at first glance, but is explained by looking at the memory hierarchy of the machine.

The machine has a 512-Kbyte cache. For a small-order matrix, the cache can accommodate the complete matrix and needs to load the matrix only once, where it will be reused from the cache. Since the cache is 512 Kbytes, it can accommodate 64,000 full-precision words or a matrix of order 252. Since other variables such as loop indices, scalar variables, and program code are also passed through the cache, the actual size of the matrix that can be fully contained is smaller than 252; and as we can see from the data, the peak is achieved around a matrix of order 210.

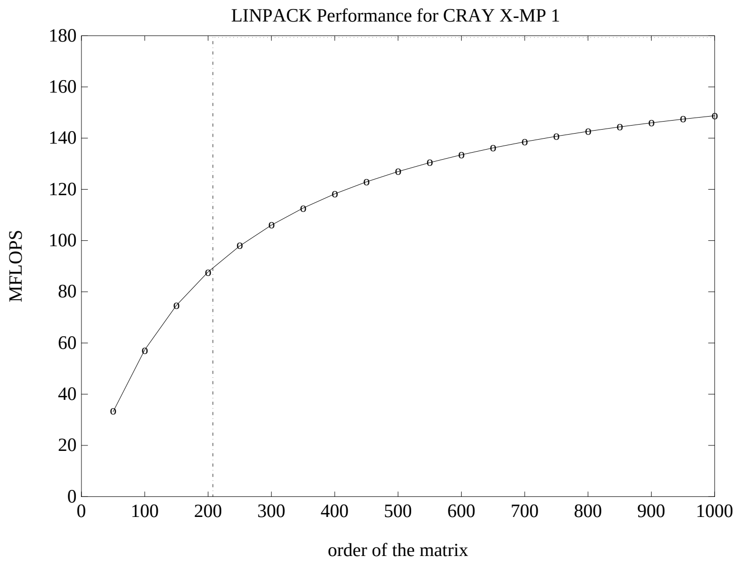

Figure 4.1: LINPACK performance for the CRAY X-MP

As the matrix becomes larger, it can no longer be fully contained in the cache, and parts must be reloaded from main memory. Thus, the rate of execution decreases as the matrix size increases past that of order 210. This trend continues until the asymptotic rate associated with moving the data fully from memory is reached (see Figure 4.3).

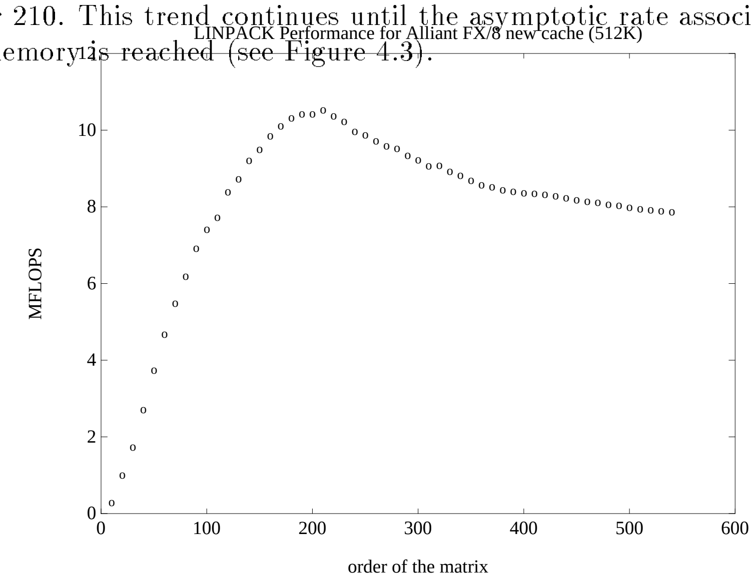

Figure 4.3: **LINPACK performance for Alliant FX/8**

Ardent Titan

The performance on the Ardent Titan provides another example, similar to that of the Alliant, of a problem with the use of the memory hierarchy. The rate of execution rises as the order of the matrix increases up to a matrix of order 550; then the performance decreases. The Ardent is a virtual-memory machine and uses a table look-aside buffer (TLB) to increase the performance of the paging system. The TLB contains the starting address of pages used in a program during execution. This configuration reduces the time required to resolve references to variables. At the same time, the TLB does not have enough locations to map all of physical memory. Thus, if a program references more bytes of data than the TLB holds, performance can be degraded by TLB misses. This situation occurs, for example, for a matrix of size 600 (see Figure 4.4).

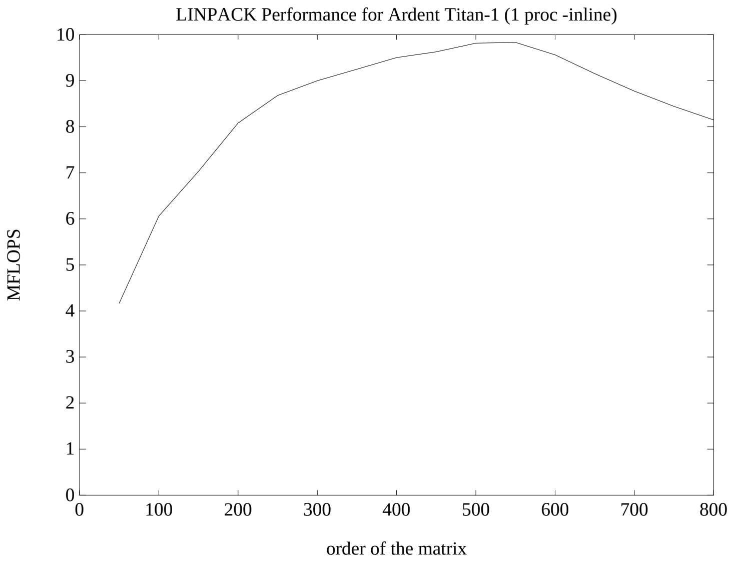

Figure 4.4: **LINPACK performance for the Ardent**

Summary of Machines

We have carried out the analysis for the asymptotic performance for running the LINPACK benchmark on a number of different machines. The results are summarized in Table 4.10.

On scalar machines, $n_{1/2}$ is effectively one, since there is no startup of vectors, and thus no penalty for short vectors. On machines that have a large value for $n_{1/2}$, we might expect to have difficulty in coming close to peak performance for general problems when the dominant operation is a simple vector operation such as a SAXPY.

Table 4.10: **Summary of $n_{1/2}$ and r_∞ for LINPACK Benchmark**

Machine	LINPACK $n = 100$, Mflops	$n_{1/2}$ Problem Size	r_∞ Mflops	Peak Mflops
Alliant FX/8 (512K cache)	6.9	160	7.3	94
Alliant FX/8 (128K cache)	5.9	34	7.6	94
Convex C-210	17	19	21	50
Convex C-220	21	82	42	100
Convex C-230		132	61	150
Convex C-240	27	145	74	200
CRAY X-MP/SE	53	87	99	210
CRAY X-MP/1	70	179	208	235
CRAY X-MP/4	178	179	456	940
CRAY Y-MP/1	144	100	208	333
CRAY Y-MP/8	275	434	1375	2667
CRAY-2S/1	44	305	138	488
CRAY-2S/4	101	297	362	1951
Cydrome Cydra 5	14	29	22	25
ETA-10P	27	484	167	167
ETA-10Q	34	484	211	210
ETA-10E	62	458	377	381
Fujitsu VP-400E	20	1522	316	1142
Fujitsu VP-400	20	1506	316	1142
Hitachi 810/20	17	2456	983	840
Hitachi 820/80	107	1693	1318	3000
Intel iPSC/2 (1 proc.)	.92	183	2.6	
NEC SX/2	43	1017	570	1300
Sequent Symmetry (1 proc.)	.21	24	.22	
Sun 3 w/FPA	.11	5	.13	
VAX 11/780 w/FPA UNIX	.12	-	.09	

The performance of the LINPACK benchmark is typical for applications where the basic operation is based on vector primitives. Some improvements may be possible by reorganizing or rearranging loops. However, in order to use the machine effectively and to obtain close to peak performance, such simple strategies as loop interchanging are not enough. In the next chapter, we discuss an approach that will achieve higher performance for the same number of floating-point operations.

Chapter 5

Building Blocks in Linear Algebra

One of the important aspects to utilizing a high-performance computer effectively is to avoid unnecessary memory references. In most computers, data flows from memory into and out of registers and from registers into and out of functional units, which perform the given instructions on the data. Performance of algorithms can be dominated by the amount of memory traffic, rather than the number of floating-point operations involved. The movement of data between memory and registers can be as costly as arithmetic operations on the data. This cost provides considerable motivation to restructure existing algorithms and to devise new algorithms that minimize data movement.

5.1 Basic Linear Algebra Subprograms

One way of achieving efficiency in the solution of linear algebra problems is through the use of the Basic Linear Algebra Subprograms. In 1973, Lawson et al. [225] described the advantages of adopting a set of basic routines for problems in linear algebra. The BLAS, as they are now commonly called, have been very successful and have been used in a wide range of software, including LINPACK and many of the algorithms published by the *ACM Transactions on Mathematical Software.* They are an aid to clarity, portability, modularity, and maintenance of software, and they have become a *de facto* standard for the elementary vector operations. The motivation for the BLAS is described in [225] and by Dodson and Lewis [84]. Here we review their purpose and their advantages. We also discuss two recent extensions to the BLAS. A complete list of routines, including calling sequence, and operations can be found in Appendix D.

The BLAS promote modularity by identifying frequently occurring operations of linear algebra and by specifying a standard interface to these operations. Efficiency may be achieved through optimization within the BLAS without altering the higher-level code that has referenced them. Obviously, it is important to identify and define a set of basic operations that are both rich enough to enable an expression of important high-level algorithms and simple enough to admit a very high level of optimization on a variety of computers. Such optimizations can often be achieved through modern compilation techniques, but hand coding in assembly language is also an option. Use of these optimized operations can yield dramatic reductions in computation time on some computers.

The BLAS also offer several other benefits:

- *Robustness* of linear algebra computations is enhanced by the BLAS, since they take into consideration algorithmic and implementation subtleties that are likely to be ignored in a typical application programming environment, such as treating overflow situations.

- Program *portability* is improved through standardization of computational kernels without giving up efficiency, since optimized versions of the BLAS can be used on those computers for which they exist, yet compatible standard Fortran is available for use elsewhere.

- Program *readability* is enhanced. The BLAS are a design tool; that is, they are a conceptual aid in coding, allowing one to visualize mathematical operations rather than the particular detailed coding required to implement the operations. By associating widely recognized mnemonic names with mathematical operations, the BLAS improve the self-documenting quality of code.

5.1.1 Level 1 BLAS

The original set of BLAS perform low-level operations such as dot-product and the adding of a multiple of one vector to another. We refer to these vector-vector operations as Level 1 BLAS. They provide a standard interface to a number of commonly used vector operations, and they instituted a useful naming convention that consists of a four- or five-letter mnemonic name preceded by one of the letters s, d, c, z to indicate precision type. Details are given in Appendix D.

The following types of basic vector operations are performed by the Level 1 BLAS:

$y \leftarrow \alpha x + y$	$dot \leftarrow x^T y$
$x \leftarrow \alpha x$	$nrm2 \leftarrow \|\|x\|\|_2$
$x \leftrightarrow y$	$y \leftarrow x$
$asum \leftarrow \|\|re(x)\|\|_1 + \|\|im(x)\|\|_1$	$amax \leftarrow 1^{st} k \ni \|re(x_k)\| + \|im(x_k)\| = \|\|x\|\|_\infty$
Generating plane rotations	Applying plane rotations

Typically these operations involve $O(n)$ floating-point operations and $O(n)$ data items moved (loaded or stored), where n is the length of the vectors.

The Level 1 BLAS permit efficient implementation on scalar machines, but the ratio of floating-point operations to data movement is too low to achieve effective use of most vector or parallel hardware. Even on a scalar machine, the cost of a subroutine call can be a performance issue when vector lengths are short and there is a limitation to the possible use of register allocations to improve the ratio of floating-point operations to data movement. The simplest and most central example is the computation of a matrix-vector product. This may be coded as a sequence of n SAXPY ($y \leftarrow \alpha x + y$) operations, but this obscures the fact that the result vector y could have been held in a vector register. Nevertheless, the clarity of code and the modularity contribute greatly to the ability to quickly construct correct, readable, and portable programs.

5.1.2 Level 2 BLAS

The principles behind the Level 1 BLAS are indeed sound and have served well in practice. However, with the advent of high-performance computers that use vector processing, it was soon recognized that additional BLAS would have to be created, since the Level 1 BLAS do not have sufficient *granularity* to admit optimizations such as reuse of data in registers and reduction in memory access. One needs to optimize at least at the level of matrix-vector operations in order to approach the potential efficiency of the machine; and the Level 1 BLAS inhibit this optimization, because they hide the matrix-vector nature of the operations from the compiler.

Thus, an additional set of BLAS, called the Level 2 BLAS, was designed for a small set of matrix-vector operations that occur frequently in the implementation of many of the most common algorithms in linear algebra [92]. The Level 2 BLAS involve $O(mn)$ scalar operations, where m and n are the dimensions of the matrix involved. The following three types of basic operation are performed by the Level 2 BLAS:

(a) Matrix-vector products of the form

$$y = \alpha A x + \beta y, \\ y = \alpha A^T x + \beta y, \text{ and} \\ y = \alpha \bar{A}^T x + \beta y$$

where α and β are scalars, x and y are vectors, and A is a matrix; and

$$x = Tx, \\ x = T^T x, \text{ and} \\ x = \bar{T}^T x,$$

where x is a vector and T is an upper or lower triangular matrix.

(b) Rank-one and rank-two updates of the form

$A = \alpha x y^T + A,$
$A = \alpha x \bar{y}^T + A,$
$H = \alpha x \bar{x}^T + H,$ and
$H = \alpha x \bar{y}^T + \bar{\alpha} y \bar{x}^T + H,$

where H is a Hermitian matrix.

(c) Solution of triangular equations of the form

$x = T^{-1} x,$
$x = T^{-T} x,$ and
$x = \bar{T}^{-T} x,$

where T is a nonsingular upper or lower triangular matrix.

Where appropriate, the operations are applied to general, general band, Hermitian, Hermitian band, triangular, and triangular band matrices in both real and complex arithmetic and in single and double precision.

Many of the frequently used algorithms of numerical linear algebra can be coded so that the bulk of the computation is performed by calls to Level 2 BLAS routines; efficiency can then be obtained by using tailored implementations of the Level 2 BLAS routines. On vector-processing machines, one of the aims of such implementations is to keep the vector lengths as long as possible, and in most algorithms the results are computed one vector (row or column) at a time. In addition, on vector register machines, performance is increased by reusing the results of a vector register and not storing the vector back into memory.

Unfortunately, this approach to software construction is often not well suited to computers with a hierarchy of memory (such as global memory, cache or local memory, and vector registers) and true parallel-processing computers.

5.1.3 Level 3 BLAS

Experience with machines [102, 163] having a memory hierarchy, such as RISC processors or shared memory parallel processors from Cray or SGI computers, indicated that the Level 2 BLAS did not have a ratio of floating point to data movement that was high enough to make efficient reuse of data that resided in cache or local memory. For those architectures, it is often preferable to partition the matrix or matrices into blocks and to perform the computation by matrix-matrix operations on the blocks. By organizing the computation in this fashion, we provide for full reuse of data while the block is held in the cache or local memory. This approach avoids excessive movement of data to and from memory. In fact, it is often possible to obtain $O(n^3)$ floating-point operations while creating only $O(n^2)$ data movement. This phenomenon is often called the *surface-to-volume* effect for the ratio of operations to data movement. In addition, on architectures that allow parallel processing, parallelism can be exploited in two ways: (1) operations on distinct blocks may be performed in parallel; and (2) within the operations on each block, scalar or vector operations may be performed in parallel.

The Level 3 BLAS [91] are targeted at the matrix-matrix operations required for these purposes. The routines are derived in a fairly obvious manner from some of the Level 2 BLAS, by replacing the vectors x and y with matrices B and C. The advantage in keeping the design of the software as consistent as possible with that of the Level 2 BLAS is that it is easier for users to remember the calling sequences and parameter conventions.

In real arithmetic the operations for the Level 3 BLAS have the following forms:

(a) Matrix-matrix products:

$$C = \alpha AB + \beta C$$
$$C = \alpha A^T B + \beta C$$
$$C = \alpha AB^T + \beta C$$
$$C = \alpha A^T B^T + \beta C$$

These operations are more accurately described as matrix-matrix multiply-and-add operations; they include rank-k updates of a general matrix.

(b) Rank-k updates of a symmetric matrix:

$$C = \alpha AA^T + \beta C$$
$$C = \alpha A^T A + \beta C$$
$$C = \alpha A^T B + \alpha B^T A + \beta C$$
$$C = \alpha AB^T + \alpha BA^T + \beta C$$

(c) Multiplying a matrix by a triangular matrix:

$$B = \alpha TB$$
$$B = \alpha T^T B$$
$$B = \alpha BT$$
$$B = \alpha BT^T$$

(d) Solving triangular systems of equations with multiple right-hand sides:

$$B = \alpha T^{-1} B$$
$$B = \alpha T^{-T} B$$
$$B = \alpha BT^{-1}$$
$$B = \alpha BT^{-T}$$

Here α and β are scalars; A, B, and C are rectangular matrices (in some cases, square and

Table 5.1: Speed in Mflop/s of Level 2 and Level 3 BLAS operations on a CRAY C90

(all matrices are of order 1000; U is upper triangular)

Number of processors:	1	2	4	8	16
Level 2: $y \leftarrow \alpha Ax + \beta y$	899	1780	3491	6783	11207
Level 3: $C \leftarrow \alpha AB + \beta C$	900	1800	3600	7199	14282
Level 2: $x \leftarrow Ux$	852	1620	3063	5554	6953
Level 3: $B \leftarrow UB$	900	1800	3574	7147	13281
Level 2: $x \leftarrow U^{-1}x$	802	1065	1452	1697	1558
Level 3: $B \leftarrow U^{-1}B$	896	1792	3578	7155	14009

symmetric); and T is an upper or lower triangular matrix (and nonsingular in (d)).

Analogous operations are in complex arithmetic: conjugate transposition is specified instead of simple transposition, and in (b) C is Hermitian and α and β are real.

The results of using the different levels of BLAS on the IBM RS/6000 Model 550 are shown in Figure 5.1. The same effect can be seen in the CRAY C90 in Table 5.1.

The rates of execution quoted in Table 5.1 for the Level 2 and 3 BLAS represent the asymptotic rates for operations implemented in assembler language.

Table 5.2 illustrates the advantage of the Level 3 BLAS through a comparison of the ratios of floating-point operations to data movement for three closely related operations from the Level 1, 2, and 3 BLAS. The second column counts the number of loads and stores required, while the third column counts the number of floating-point operations required to complete the operation. The fourth column reports the ratio of these two.

5.2 Levels of Parallelism

Today's advanced computers exhibit different levels of parallelism. In this section, we discuss these different levels and briefly outline how algorithms may be recast to achieve efficiency.

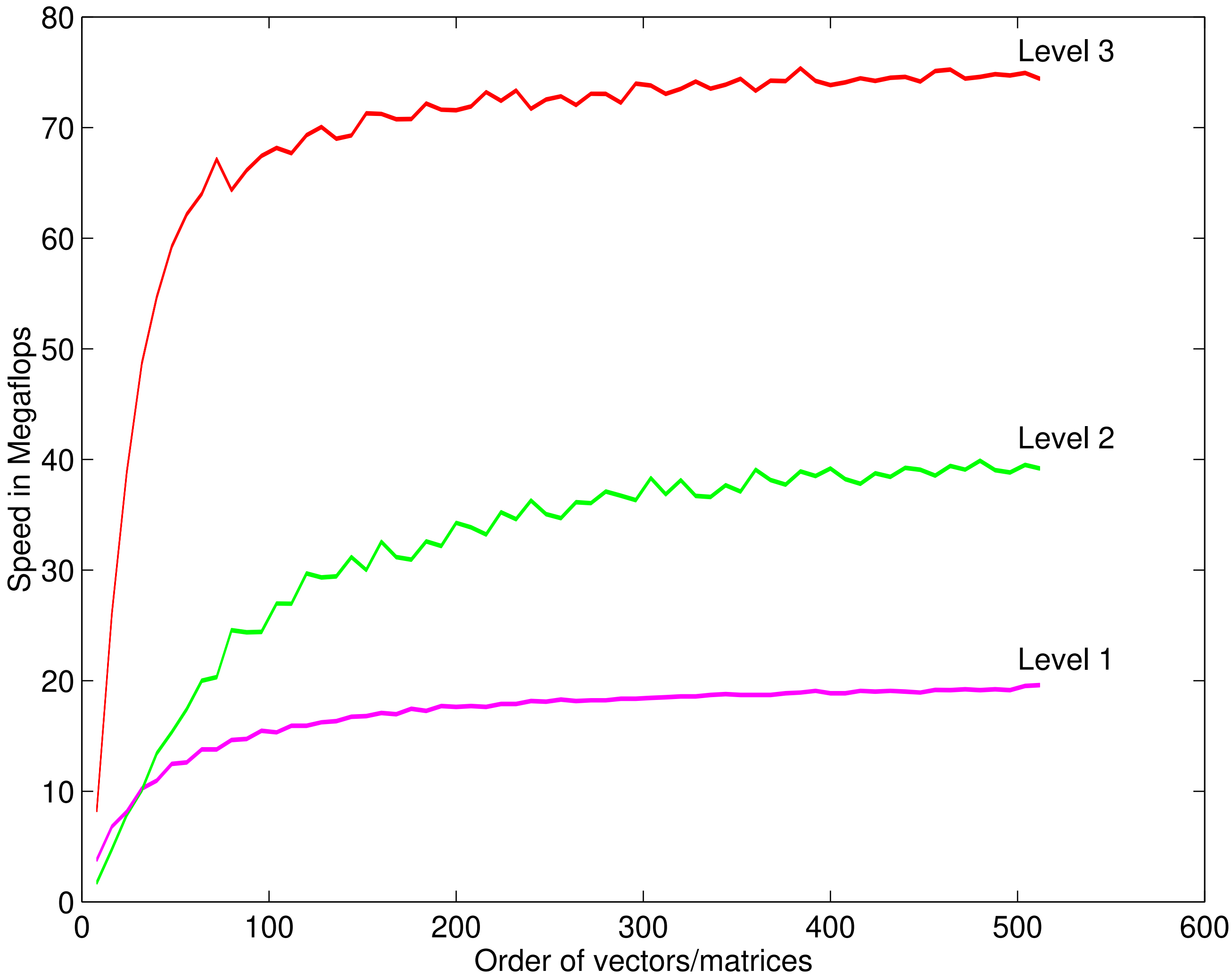

Figure 5.1: Level 1, 2, and 3 BLAS Performance on IBM RS/6000-550

Table 5.2: Advantage of the Level 3 BLAS

BLAS	Loads and Stores	Floating-Point Operations	Ref:Ops (ratio) $n = m = k$
Level 1 SAXPY $y = y + \alpha x$	$3n$	$2n$	$3 : 2$
Level 2 SGEMV $y = \beta y + \alpha Ax$	$mn + n + 2m$	$2mn$	$1 : 2$
Level 3 SGEMM $C = \beta C + \alpha AB$	$2mn + mk + kn$	$2mnk$	$2 : n$

5.2.1 Vector Computers

Vector architectures exploit parallelism at the lowest level of computation. They require regular data structures (i.e., rectangular arrays) and large amounts of computation in order to be effective.

Most algorithms in linear algebra can be easily vectorized. However, to gain the most out of such machines, simple vectorization is usually not enough. Some vector computers are limited in the sense that there is only one path between memory and the vector registers. This creates a bottleneck if a program loads a vector from memory, performs some arithmetic operations, and then stores the results. In order to achieve top performance, the scope of the vectorization must be expanded to facilitate chaining and minimize data movement, in addition to using vector operations. Recasting the algorithms in terms of matrix-vector operations makes it easy for a vectorizing compiler to achieve these goals.

Let us look more closely at how the recasting of an algorithm is performed. Many of the algorithms in linear algebra can be expressed in terms of a SAXPY operation. This results in three vector memory references for each two vector floating-point operations. If this operation constitutes the body of an inner loop that updates the same vector y many times, a considerable amount of unnecessary data movement will occur.

Usually, if a SAXPY occurs in an inner loop, the algorithm may be recast in terms of some matrix-vector operation, such as $y = y + M * x$, which is just a sequence of SAXPYs involving the columns of the matrix M and the corresponding components of the vector x (this is a matrix-vector multiple, GAXPY or generalized SAXPY operation). Thus it is relatively easy to recognize automatically that only the columns of M need be moved into registers while accumulating the result y in a vector register, avoiding two of the three memory references in the innermost loop. The rewritten algorithm also allows chaining to occur on vector machines.

Since that time, a considerable improvement has been made in the ability of an optimizing compiler to vectorize and in some cases even to automatically parallelize the nested loops that occur in the BLAS. Usually, modern optimizing compilers are sophisticated enough to achieve near-optimal performance of the BLAS at all levels because of their regularity and simplicity. However, if the compiler is not successful, it is still quite reasonable to hand tune these operations, perhaps in assembly language, since there are so few of them and since they involve simple operations on regular data structures.

Higher-level codes may also be constructed from these modules. The resulting codes have achieved super-vector performance levels on a wide variety of vector architectures. Moreover, the same codes have also proved effective on parallel architectures. In this way portability has been achieved without suffering a serious degradation in performance.

5.2.2 Parallel Processors with Shared Memory

The next level of parallelism involves individual scalar processors executing serial instruction streams simultaneously on a shared-data structure. A typical example would be the simultaneous execution of a loop body for various values of the loop index. This is the capability provided by a parallel processor. With this increased functionality, however, comes a burden. This communication requires synchronization: if these independent processors are to work together on the same computation, they must be able to communicate partial results to each other. Such synchronization introduces overhead. It also requires new programming techniques that are not well understood at the moment. However, the simplicity of the basic operations within the Level 2 and Level 3 BLAS does allow the encapsulated exploitation of parallelism.

Typically, a parallel processor with globally shared memory must employ some sort of interconnection network so that all processors may access all of the shared memory. There must also be an arbitration mechanism within this memory access scheme to handle cases where two processors attempt to access the same memory location at the same time. These two requirements obviously have the effect of increasing the memory access time over that of a single processor accessing a dedicated memory of the same type.

Again, memory access and data movement dominate the computations in these machines. Achieving near-peak performance on such computers requires devising algorithms that minimize data movement and reuse data that has been moved from globally shared memory to local processor memory.

5.2.3 Parallel-Vector Computers

The two types of parallelism we have just discussed are combined when vector rather than serial processors are used to construct a parallel computer. These machines are able to execute independent loop bodies that employ vector instructions. Some of the most powerful computers today are of this type. They include the CRAY T90 line and the NEC SX-4 and Fujitsu VPP-300. The problems with using such computers efficiently are, of course, more difficult than those encountered with each type individually. Synchronization overhead becomes more significant for a vector operation than a scalar operation, blocking loops to exploit outer-level parallelism may conflict with vector length, and so on.

5.2.4 Clusters Computing

A further complication is added when computers are interconnected to achieve yet another level of parallelism. This is the case for the SGI Power Challenge. Such a computer is intended to solve large applications problems that naturally split up into loosely coupled parts which may be solved efficiently on a cluster of parallel processors.

5.3 Basic Factorizations of Linear Algebra

In this section we develop algorithms for the solution of linear systems of equations and linear least-squares problems. Such problems are basic to all scientific and statistical calculations. Our intent here is to briefly introduce these basic notions. Thorough treatments may be found in [294, 175].

First, we consider the solution of the linear system

$$Ax = b, \tag{5.1}$$

where A is a real $n \times n$ matrix and x and b are both real vectors of length n. To construct a numerical algorithm, we rely on the fundamental result from linear algebra that implies that a permutation matrix P exists such that

$$PA = \mathrm{LU},$$

where L is a unit lower triangular matrix (ones on the diagonal) and U is upper triangular. The matrix U is nonsingular if and only if the original coefficient matrix A is nonsingular. Of course, this factorization facilitates the solution of equation (5.1) through the successive solution of the triangular systems

$$Ly = Pb, \; Ux = y.$$

The well-known numerical technique for constructing this factorization is called Gaussian elimination with partial pivoting. The reader is referred to [175] for a detailed discussion. As a point of reference, a brief development of this basic factorization will be given here. Variants of the factorization will provide a number of block algorithms.

5.3.1 Point Algorithm: Gaussian Elimination with Partial Pivoting

Let P_1 be a permutation matrix such that

$$P_1 A e_1 = \begin{pmatrix} \delta \\ c \end{pmatrix},$$

where $|\delta| \geq |c^T e_j|$ for all j (i.e., δ is the element of largest magnitude in the first column). If $\delta \neq 0$, put $l = c\delta^{-1}$; otherwise put $l = 0$. Then

$$P_1 A = \begin{pmatrix} \delta & u^T \\ c & \hat{A} \end{pmatrix} = \begin{pmatrix} 1 & \\ l & I \end{pmatrix} \begin{pmatrix} \delta & u^T \\ 0 & \hat{A} - lu^T \end{pmatrix}.$$

Suppose now that $\hat{L}\hat{U} = \hat{P}\left(\hat{A} - lu^T\right)$. Then

$$\begin{pmatrix} 1 & 0 \\ 0 & \hat{P} \end{pmatrix} P_1 A = \begin{pmatrix} 1 & 0 \\ \hat{P}l & \hat{L} \end{pmatrix} \begin{pmatrix} \delta & u^T \\ 0 & \hat{U} \end{pmatrix},$$

and

$$PA = LU,$$

where P, L, and U have the obvious meanings in the above factorization.

This discussion provides the basis for an inductive construction of the factorization $PA = LU$. However, a development that is closer to what is done in practice may be obtained by repeating the basic factorization step on the "reduced matrix" $\hat{A} - lu^T$ and continuing in this way until the final factorization has been achieved in the form

$$A = \left(P_1^T L_1 P_2^T L_2 ... P_{n-1}^T L_{n-1}\right) U,$$

with each P_i representing a permutation in the (i, k_i) positions where $k_i \geq i$. A number of algorithmic consequences are evident from this representation. One is that the permutation matrices may be represented by a single vector p with $p_i = k_i$. To compute the action $P_i b$, we simply interchange elements i and k_i. This representation leads to the following numerical procedure for the solution of (5.1).

for $j = 1$ step 1 until n
    $b \leftarrow L_j^{-1} P_j b$; (1)
end ;
Solve $Ux = b$;

Since $L_j^{-1} = I - l_j e_j^T$, (1) amounts to a SAXPY operation.

5.3.2 Special Matrices

Often matrices that arise in scientific calculations will have special structure. There are good reasons for taking advantage of such structure when it exists. In particular, storage requirements can be reduced, the number of floating-point operations can be reduced, and more stable algorithms can be obtained.

An important class of such special matrices is symmetric matrices. A matrix is *symmetric* if $A = A^T$. A symmetric matrix A is *positive semidefinite* if $x^T Ax \geq 0$ for all x, and it is *indefinite* if $(x^T Ax)$ takes both positive and negative values for different vectors x. It is called *positive definite* if $x^T Ax > 0$ for all $x \neq 0$. A well-known result is that a symmetric matrix A is positive definite if and only if

$$A = LDL^T,$$

where L is unit lower triangular and D is diagonal with positive diagonal elements.

This factorization results from a modification of the basic Gaussian elimination algorithm to take advantage of the fact that a matrix is symmetric and positive definite. This results in the Cholesky factorization.

Cholesky Factorization. The computational algorithm for Cholesky factorization can be developed as follows. Suppose that A is symmetric and positive definite. Then $\delta = e_1^T Ae_1 > 0$. Thus, putting $l = a\delta^{-1}$ gives

$$A = \begin{pmatrix} \delta & a^T \\ a & \hat{A} \end{pmatrix} = \begin{pmatrix} 1 & \\ l & I \end{pmatrix} \begin{pmatrix} \delta & \\ & \hat{A} - la^T \end{pmatrix} \begin{pmatrix} 1 & l^T \\ & I \end{pmatrix}.$$

Since $l = a\delta^{-1}$, the submatrix $\hat{A} - la^T = \hat{A} - a(\delta^{-1})a^T$ is also symmetric and is easily shown to be positive definite as well. Therefore, the factorization steps may be continued (without pivoting) until the desired factorization is obtained. Details about numerical stability and variants are given in [251].

Symmetric Indefinite Factorization. When A is symmetric but indefinite, pivoting must be done to factor the matrix. The example

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

shows that there is no general factorization of the form

$$A = LDL^T,$$

where D is diagonal and L is lower triangular when A is indefinite. There is, however, a closely related factorization

$$PAP^T = LDL^T,$$

where P is a permutation matrix, L is a unit lower triangular matrix, and D is a block diagonal matrix with one-by-one and two-by-two diagonal blocks.

Bunch and Kaufman [45] have devised a clever partial pivoting strategy to construct such a factorization. For a single step, the pivot selection is completely determined by examining two columns of the matrix A, with the selection of the second column depending on the location of the maximum element of the first column of A. Let j be the index of the entry in column 1 of largest magnitude. The pivot matrix is the identity if no pivot is required; an interchange between row and column 1 with row and column j if entry α_{jj} is to become a 1×1 pivot; and an interchange between row and column 2 with row and column j if the submatrix

$$\begin{pmatrix} \alpha_{11} & \alpha_{j1} \\ \alpha_{j1} & \alpha_{jj} \end{pmatrix}$$

is to become a 2×2 pivot. The pivot strategy devised by Bunch and Kaufman [45] is described in the following pseudo-code. In this code $\theta = (1 + (17)^{1/2})/8$ is chosen to optimally control element growth in the pivoting strategy:

> $j = index\ of\ max_i(|\alpha_{i1}|)$;
> $\mu = max_j(\ |\alpha_{jl}|)$;
> if ($|\alpha_{11}| \geq \theta|\alpha_{j1}|\ or\ |\alpha_{11}|\mu \geq \theta|\alpha_{j1}|^2$) then
> $j = 1$; (1×1 pivot; with no interchange);
> else
>   if ($|\alpha_{jj}| \geq \theta\mu$) then
>   (1×1 pivot; interchange 1 with j);
>   else
>   (2×2 pivot; interchange 2 with j);

```
    endif;
  endif;
  end;
```

This scheme is actually a slight modification of the original Bunch-Kaufman strategy. It has the same error analysis and has a particularly nice feature for positive definite matrices. We emphasize two important points. First, the column $j > 1$ of A—which must be examined in order to determine the necessary pivot—is determined solely by the index of the element in the first column that is of largest magnitude. Second, if the matrix A is positive definite, then *no pivoting will be done*, and the factorization reduces to the Cholesky factorization. To see this, note the following. If A is positive definite, then the submatrix

$$\begin{pmatrix} \alpha_{11} & \alpha_{j1} \\ \alpha_{j1} & \alpha_{jj} \end{pmatrix}$$

must be positive definite. Thus,

$$|\alpha_{11}|\mu = \alpha_{11}\mu \geq \alpha_{11}\alpha_{22} > \alpha_{j1}^2 > \theta\alpha_{j1}^2,$$

which indicates that α_{11} will be a 1×1 pivot and no interchange will be made according to the pivot strategy. Further detail concerning this variant is available in [292].

Now, assume that the pivot selection for a single step has been made. If a 1×1 pivot has been indicated, then a symmetric permutation is applied to bring the selected diagonal element to the (1,1) position, and the factorization step is exactly as described above for the Cholesky factorization. If a 2×2 pivot is required, then a symmetric permutation must be applied to bring the element of largest magnitude in the first column into the (2,1) position. Thus

$$P_1 A P_1^T = \begin{pmatrix} D & C \\ C^T & \hat{A} \end{pmatrix} = \begin{pmatrix} I & \\ CD^{-1} & I \end{pmatrix} \begin{pmatrix} D & \\ & \hat{A} - CD^{-1}C^T \end{pmatrix} \begin{pmatrix} 1 & D^{-1}C^T \\ & I \end{pmatrix}.$$

In this factorization step, let $D = \begin{pmatrix} \delta_1 & \beta \\ \beta & \delta_2 \end{pmatrix}$ and $C = (c_1, c_2)$. When a 2×2 pivot has been selected, it follows that $|\delta_1\delta_2| < \theta^2\beta^2$, so that $det(D) < (\theta^2 - 1)\beta^2 < 0$ and D is a 2×2 indefinite matrix.

Note that with this strategy every 2×2 pivot matrix D has a positive and a negative eigenvalue. Thus one can read off the inertia (the number of positive, negative, and zero eigenvalues) of the factorized matrix by counting the number of positive 1×1 pivots plus the number of 2×2 pivots to get the number of positive eigenvalues, and the number of negative 1×1 pivots plus the number of 2×2 pivots to get the number of negative eigenvalues.

As with the other factorizations, this one may be continued to completion by repeating the basic step on the reduced submatrix

$$\hat{\hat{A}} = \hat{A} - CD^{-1}C^T.$$

Exploitation of symmetry reduces the storage requirement and also the computational cost by half.

An alternative to this factorization is given by Aasen in [1]. A blocked version of that alternative algorithm is presented in [292].

5.4 Blocked Algorithms: Matrix-Vector and Matrix-Matrix Versions

The basic algorithms just described are the core subroutines in LINPACK for solving linear systems. The original version of LINPACK was designed to use vector operations. However, the algorithms did not perform near the expected level on the powerful vector machines that were developed just as the package had been completed. The key to achieving high performance on these advanced architectures has been to recast the algorithms in terms of matrix-vector and matrix-matrix operations to permit reuse of data.

To derive these variants, we examine the implications of a block factorization in progress. One can construct the factorization by analyzing the way in which the various pieces of the factorization interact. Let us consider the decomposition of the matrix A into its LU factorization with the matrix partitioned in the following way. Let us suppose that we have factored A as $A = \mathrm{LU}$. We write the factors in block form and observe the consequences.

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix}$$

Multiplying L and U together and equating terms with A, we have

$$\begin{array}{lll} A_{11} = L_{11}U_{11}, & A_{12} = L_{11}U_{12}, & A_{13} = L_{11}U_{13}, \\ A_{12} = L_{21}U_{11}, & A_{22} = L_{21}U_{12} + L_{22}U_{22}, & A_{23} = L_{21}U_{13} + L_{22}U_{23}, \\ A_{31} = L_{31}U_{11}, & A_{32} = L_{31}U_{12} + L_{32}U_{22}, & A_{33} = L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33}. \end{array}$$

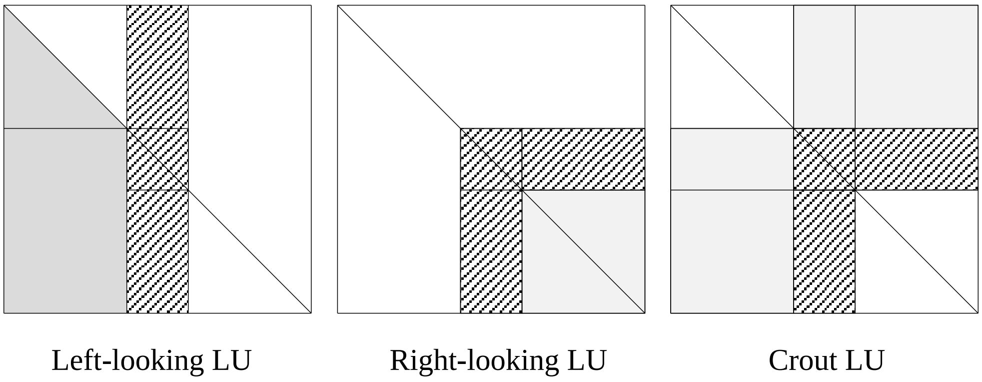

Figure 5.2: Memory access patterns for variants of LU decomposition

With these simple relationships we can develop variants by postponing the formation of certain components and also by manipulating the order in which they are formed. A crucial factor for performance is the choice of the *blocksize,* k (i.e., the column width) of the second block column. A blocksize of 1 will produce matrix-vector algorithms, while a blocksize of $k > 1$ will produce matrix-matrix algorithms. Machine-dependent parameters such as cache size, number of vector registers, and memory bandwidth will dictate the best choice for the blocksize.

Three natural variants occur: right-looking, left-looking, and Crout (see Figure 5.4). The left-looking variant computes one block column at a time, using previously computed columns. The right-looking variant (the familiar recursive algorithm) computes a block row and column at each step and uses them to update the trailing submatrix. The terms right and left refer to the regions of data access, and Crout represents a hybrid of the left- and right-looking version.

The shaded parts indicate the matrix elements accessed in forming a block row or column, and the darker shading indicates the block row or column being computed. The left-looking variant computes one block column at a time, using previously computed columns. The right-looking variant (the familiar recursive algorithm) computes a block row and column at each step and uses them to update the trailing submatrix. The Crout variant is a hybrid algorithm in which a block row and column are computed at each step, using previously computed rows and previously computed columns. These variants have been called the *i,j,k variants* owing to the arrangement of loops in the algorithm. For a more complete discussion of the different variants, see [96, 251].

Each of these can be modified to produce a variant of the Cholesky factorization when the matrix is known to be symmetric and positive definite. These modifications are straightforward and will not be presented. The symmetric indefinite case does provide some difficulty, however, and this will be developed separately.

Let us now develop these block variants of Gaussian elimination with partial pivoting.

5.4.1 Right-Looking Algorithm

Suppose that a partial factorization of A has been obtained so that the first k columns of L and the first k rows of U have been evaluated. Then we may write the partial factorization in block partitioned form, with square blocks along the leading diagonal, as

$$PA = \begin{pmatrix} L_{11} & & \\ L_{21} & I & \\ L_{31} & 0 & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & \hat{A}_{22} & \hat{A}_{23} \\ & \hat{A}_{32} & \hat{A}_{33} \end{pmatrix}, \tag{5.2}$$

where L_{11} and U_{11} are $k \times k$ matrices, and P is a permutation matrix representing the effects of pivoting. Pivoting is performed to improve the numerical stability of the algorithm and involves the interchange of matrix rows. The blocks labeled $\hat{A}_{ij}$ in Eq. 5.2 are the updated portion of A that has not yet been factored, and will be referred to as the *active submatrix*.

We next advance the factorization by evaluating the next block column of L and the next block row of U, so that

$$\begin{pmatrix} I & \\ & P_2 \end{pmatrix} PA = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & \hat{A}_{33} \end{pmatrix}. \tag{5.3}$$

where P_2 is a permutation matrix of order $M - k$. Comparing Eqs. 5.2 and 5.3 we see that the factorization is advanced by first factoring the first block column of the active submatrix which will be referred to as the *current column*,

$$P_2 \begin{pmatrix} \hat{A}_{22} \\ \hat{A}_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22} \tag{5.4}$$

This gives the next block column of L. We then pivot the active submatrix to the right of the current column and the partial L matrix to the left of the current column,

$$\begin{pmatrix} \hat{A}_{23} \\ \hat{A}_{33} \end{pmatrix} \Leftarrow P_2 \begin{pmatrix} \hat{A}_{23} \\ \hat{A}_{33} \end{pmatrix}, \quad \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \Leftarrow P_2 \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \tag{5.5}$$

and solve the triangular system

$$U_{23} = L_{22}{}^{-1} \hat{A}_{23} \tag{5.6}$$

to complete the next block row of U. Finally, a matrix-matrix product is performed to update $\hat{A}_{33}$,

$$\hat{A}_{33} \Leftarrow \hat{A}_{33} - L_{32} U_{23}. \tag{5.7}$$

Now, one simply needs to relabel the blocks to advance to the next block step.

The main advantage of the block partitioned form of the LU factorization algorithm is that the updating of $\hat{A}_{33}$ (see Eq. 5.7) involves a matrix-matrix operation if the block size is greater than 1. Matrix-matrix operations generally perform more efficiently than matrix-vector operations on high performance computers. However, if the block size is equal to 1, then a matrix-vector operation is used to perform an outer product — generally the least efficient of the Level 2 BLAS [92] since it updates the whole submatrix.

Note that the original array A may be used to store the factorization, since the L is unit lower triangular and U is upper triangular. Of course, in this and all of the other versions of LU factorization, the additional zeros and ones appearing in the representation do not need to be stored explicitly.

5.4.2 Left-Looking Algorithm

As we shall see, from the standpoint of data access, the left-looking variant is the best of the three. To begin, we assume that

$$PA = \begin{pmatrix} L_{11} & & \\ L_{21} & I & \\ L_{31} & 0 & I \end{pmatrix} \begin{pmatrix} U_{11} & A_{12} & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & A_{32} & A_{33} \end{pmatrix}. \tag{5.8}$$

and that we wish to advance the factorization to the form

$$\begin{pmatrix} I & \\ & P_2 \end{pmatrix} PA = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & A_{13} \\ 0 & U_{22} & A_{23} \\ 0 & 0 & A_{33} \end{pmatrix}. \tag{5.9}$$

Comparing Eqs. 5.8 and 5.9 we see that the factorization is advanced by first solving the triangular system

$$U_{12} = L_{11}^{-1} A_{12} \tag{5.10}$$

and then performing a matrix-matrix product to update the rest of the middle block column of U,

$$\begin{pmatrix} \hat{A}_{22} \\ \hat{A}_{32} \end{pmatrix} \Leftarrow \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} U_{12}. \tag{5.11}$$

Next we perform the factorization

$$P_2 \begin{pmatrix} \hat{A}_{22} \\ \hat{A}_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22} \tag{5.12}$$

and lastly the pivoting

$$\begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} \Leftarrow P_2 \begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} \text{ and } \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \Leftarrow P_2 \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix}. \tag{5.13}$$

Observe that data accesses all occur to the left of the block column being updated. Moreover, the only write access occurs within this block column. Matrix elements to the right are referenced only for pivoting purposes, and even this procedure may be postponed until needed with a simple rearrangement of the above operations. Also, note that the original array A may be used to store the factorization, since the L is unit lower triangular and U is upper triangular. Of course, in this and all of the other versions of triangular factorization, the additional zeros and ones appearing in the representation do not need to be stored explicitly.

5.4.3 Crout Algorithm

The Crout variant is best suited for vector machines with enough memory bandwidth to support the maximum computational rate of the vector units. Its advantage accrues from avoiding the solution of triangular systems and also from requiring fewer memory references than the right-looking algorithm. Here, it is assumed that

$$PA = \begin{pmatrix} L_{11} & & \\ 0 & I & \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & U_{12} & U_{13} \\ L_{21} & A_{22} & A_{23} \\ L_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} U_{11} & 0 & 0 \\ & I & 0 \\ & & I \end{pmatrix}.$$

To advance this factorization, we solve

$$(A_{22}, A_{23}) \leftarrow (A_{22}, A_{23}) - L_{21}\,(U_{12}, U_{13}) \ \ \text{(matrix-matrix product)},$$

then

$$A_{32} \leftarrow A_{32} - L_{31}U_{12} \ \text{(matrix-matrix product)},$$

and then factor

$$P_2 \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22}.$$

Now we replace

$$\begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} \leftarrow P_2 \begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix}, \quad \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \leftarrow P_2 \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix}$$

and put

$$U_{23} = L_{21}^{-1} A_{23}.$$

At this point,

$$P_2PA = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & & U_{13} \\ 0 & I & U_{23} \\ L_{31} & L_{32} & A_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & 0 \\ & U_{22} & 0 \\ & & I \end{pmatrix}.$$

Again, a repartitioning and relabeling will allow the factorization to advance to completion.

With this presentation, the storage scheme may be somewhat obscure. Note that the lower triangular factor L_{11} and the upper triangular factor U_{11} may occupy the upper left-hand corner of the original $n \times n$ array. Thus, the original array A may be overwritten with this factorization as it is computed.

5.4.4 Typical Performance of Blocked LU Decomposition

Most of the computational work for the LU variants is contained in three routines: the matrix-matrix multiply, the triangular solve with multiple right-hand sides, and the unblocked LU factorization for operations within a block column. Table 5.4.4 shows the distribution of work among these three routines and the average performance rates on one processor of a CRAY-2 for a sample matrix of order 500 and a blocksize of 64. In these examples, each variant calls its own unblocked variant, and the pivoting operation uses about 2% of the total time. The average speed of the matrix-matrix multiply on the CRAY-2 is over 400 Mflops for all three variants, but the average speed of the triangular solve depends on the size of the triangular matrices. For the left-looking variant, the triangular matrices at each step range in size from k to $n-k$, where k is the blocksize and n the order of the matrix, and the average performance is 268 Mflops, while for the right-looking and Crout variants, the triangular matrices are always of order k and the average speed is only 105 Mflops. Clearly the average performance of the Level 3 BLAS routines in a blocked routine is as important as the percentage of Level 3 BLAS work.

Despite the differences in the performance rates of their components, the block variants of the LU factorization tend to show similar overall performance, with a slight advantage to the right-looking and Crout variants because more of the operations are in SGEMM. This points out the importance of having a good implementation for all the BLAS.

Table 5.3: Breakdown of Operations and Times for LU Variants for $n = 500$, $k = 64$ (CRAY-2S, 1 processor)

Variant	Routine	% Operations	% Time	Avg. Mflops
Left-looking	SGEMM	49	32	438
	STRSM	41	45	268
	unblocked LU	10	20	146
Right-looking	SGEMM	82	56	414
	STRSM	8	23	105
	unblocked LU	10	19	151
Crout	SGEMM	82	57	438
	STRSM	8	24	105
	unblocked LU	10	16	189

5.4.5 Blocked Symmetric Indefinite Factorization

To derive a blocked version of this algorithm, we adopt the right-looking or rank-k update approach. The pivoting requirement apparently prevents the development of an effective left-looking algorithm.

To this end, suppose that $k-1$ columns of the matrix L have been computed and stored in an $n \times (k-1)$ array L. Then

$$PAP^T = LDL^T + \begin{pmatrix} 0 & 0 \\ 0 & \tilde{A} \end{pmatrix}$$

which may be rearranged to give

$$PAP^T - LDL^T = \begin{pmatrix} 0 & 0 \\ 0 & \tilde{A} \end{pmatrix}.$$

Now, to advance the factorization one step, we need only know the first and the jth column of $\tilde{A}$ where j is the index of the element of maximum magnitude in the first column. These two columns can be revealed without knowing $\tilde{A}$ explicitly, by noting that the ith column is given by

$$\tilde{A}e_i = \left[PAP^T - LDL^T\right] e_{k+i-1}.$$

We compute P_2 as shown above in the point algorithm to produce

$$P_2\tilde{A}P_2^T = \begin{pmatrix} I & 0 \\ L_{22} & I \end{pmatrix} \begin{pmatrix} D_2 & 0 \\ 0 & \tilde{\tilde{A}} \end{pmatrix} \begin{pmatrix} I & L_{22}^T \\ 0 & I \end{pmatrix}.$$

Now,

$$P \leftarrow P \begin{pmatrix} I & 0 \\ 0 & P_2 \end{pmatrix}, \; D \leftarrow \begin{pmatrix} D & 0 \\ 0 & D_2 \end{pmatrix}, \text{ and } L \leftarrow \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix}.$$

At this point the decomposition (5.1) has been updated with L becoming an $n \times k$ or an $n \times (k+1)$ matrix and D becoming a $k \times k$ or a $(k+1) \times (k+1)$ matrix according to the pivot requirements. Once the number of columns of L has become as large as desired, we let

$$PAP^T = \begin{pmatrix} A_{11} & A_{12}^T \\ A_{21} & A_{22} \end{pmatrix},$$

where the first block column has the same dimensions as L. Now the transformations may be explicitly applied with the trailing submatrix A_{22} being overwritten with

$$A_{22} \leftarrow A_{22} - L_{21} D L_{21}^T,$$

which can be computed by using a matrix-matrix product. Of course, symmetry will be taken advantage of here to reduce the operation count by half.

As shown above, it is possible to develop the left-looking algorithm in the case of LU decomposition, a procedure that is advantageous because it references only the matrix within a single block column for reads and writes and references only the matrix from the left for reads. The matrix entries to the right of the *front* are not referenced. An analogous version is apparently not possible for the symmetric indefinite factorization.

Let us demonstrate by trying to generate this version in the same manner as one would generate "left-looking" LU decomposition. The derivation requires a different approach to be taken. Suppose that

$$PAP^T = \begin{pmatrix} L_{11} & & \\ L_{21} & I & \\ L_{31} & 0 & I \end{pmatrix} \begin{pmatrix} DL_{11}^T & A_{12} & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & A_{32} & A_{33} \end{pmatrix}, \tag{5.14}$$

where the block

$$\begin{pmatrix} A_{12} \\ A_{22} \\ A_{32} \end{pmatrix}$$

is to be factored and used to update the existing factorization. Suppose for the moment that no additional pivoting is necessary. Then the steps required to update the factorization are as follows:

1. Note $L_{11}DL_{21}^T = A_{12}$ by symmetry.
2. Overwrite $A_{22} \leftarrow A_{22} - L_{21}DL_{21}^T$.
3. Overwrite $A_{32} \leftarrow A_{32} - L_{31}DL_{21}^T$.
4. Factor $A_{22} = L_{22}D_2L_{22}^T$.
5. Overwrite $L_{32} = A_{32} \leftarrow A_{32}D_2^{-1}L_{22}^{-T}$.

Now update (5.2) to obtain

$$PAP^T = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & I \end{pmatrix} \begin{pmatrix} DL^T{}_{11} & DL_{21}^T & A_{13} \\ 0 & D_2L_{22}^T & A_{23} \\ 0 & 0 & A_{33} \end{pmatrix}.$$

This advancement of the factorization by blocks may be continued by repartitioning and repeating these steps to adjoin the next block of columns.

Unfortunately, because of the required look-ahead, pivoting must be done one step at a time in order to fill out the block. The same look-ahead strategy described for the "right-looking" algorithm could be employed to construct the block one step at a time. However, the advantage of the apparent matrix-matrix products at steps 2 and 3 above is lost, because the pivoting may incorporate entries from the blocks A_{13} , A_{23}, and A_{33}. Thus, when pivoting has been incorporated, one is limited to matrix-vector operations. Hence, pivoting prevents the development of a "left-looking" Level 3 version of the symmetric indefinite factorization.

5.4.6 Typical Performance of Blocked Symmetric Indefinite Factorization

The performance for the symmetric indefinite factorization routine is along the same lines as LU decomposition. It has to duplicate a little of the computation in order to "look ahead" to determine the necessary row and column interchanges, but the extra work can be more than compensated for by the greater speed of updating the matrix by blocks as is illustrated in Table 5.4.

Table 5.4: Speed in megaflops of DSYTRF for matrices of order n with UPLO = 'U' on an IBM POWER2 model 590

Block size	Values of n 100	1000
1	62	86
64	68	165

Table 5.5: Performance in Mflops of Symmetric Indefinite Factorization $k = 64$

(CRAY-2/S, 1 processor)

n	Unblocked	Blocked
100	44	48
200	88	109
300	115	168
400	132	198
500	140	238
700	158	265
1000	171	314

In Table 5.5 we show the increase in performance of the blocked version over the unblocked version for various orders of matrices. As can be seen, the performance of the blocked version is superior to the matrix-vector version.

5.5 Linear Least Squares

In addition to solving square linear systems, we are also concerned with overdetermined systems. That is, we wish to "solve" systems

$$Ax = b, \tag{5.15}$$

where A is a real $m \times n$ matrix with $m > n$, x is a vector of length n, and b is a vector of length m. Of course, the system (5.3) is consistent (i.e., has a solution) if and only if b is in the range of A. Therefore, we often seek a relaxed solution to a related problem:

$$\min_x ||Ax - b||. \tag{5.16}$$

This problem is called the linear least-squares problem when the 2-norm is used. We already have enough tools to solve this problem, since the quadratic function

$$||Ax - b||^2 = x^T(A^TA)x - 2b^TAx + b^Tb \tag{5.17}$$

has a unique minimum at

$$\hat{x} = (A^TA)^{-1}A^Tb, \tag{5.18}$$

which may be verified by substitution or by taking the gradient of (5.5) with respect to the components of x and setting it to zero. It is easily verified that $\hat{x}$ is the unique minimizer of (5.4) if and only if the symmetric matrix A^TA is positive definite. Thus we can solve (5.4) by forming A^TA and computing its Cholesky factorization.

5.5.1 Householder Method

The *normal equation* approach that we have just outlined is often used and is usually successful. The approach does, however, have drawbacks stemming from the potential loss of information through the explicit formation of A^TA in finite-precision arithmetic and through the potential squaring of the *condition number* of the problem. When A has linearly independent columns, the condition number of a linear system or a linear least-squares problem with a coefficient matrix A is defined to be $||A||\ ||A^I||$, where $A^I = A^{-1}$ when A is square and $A^I = (A^TA)^{-1}A^T$ when A has more rows than columns. The size of the condition number is a measure of the sensitivity of the solution to perturbations in the data (i.e., in the matrix A or in the right-hand side b). Further details may be found in [294].

An alternative has therefore been developed that is only slightly more expensive and that avoids these problems. The method is QR factorization with Householder transformations. Given a real $m \times n$ matrix A, the routine must produce an $m \times m$ orthogonal matrix Q and an $n \times n$ upper triangular matrix R such that

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}.$$

Householder's method involves constructing a sequence of transformations of the form

$$I - \alpha uu^T, \text{ where } \alpha u^Tu = 2. \tag{5.19}$$

The vector u is constructed to transform the first column of the given matrix into a multiple of the first coordinate vector e_1. At the kth stage of the algorithm, one has

$$Q_{k-1}^TA = \begin{pmatrix} R_{k-1} & S_{k-1} \\ 0 & A_{k-1} \end{pmatrix},$$

and u_k is constructed such that

$$(I - \alpha_k u_k u_k^T)A_{k-1} = \begin{pmatrix} \rho_k & s_k^T \\ 0 & A_k \end{pmatrix}. \tag{5.20}$$

The factorization is then updated to the form

$$Q_k^T A = \begin{pmatrix} R_k & S_k \\ 0 & A_k \end{pmatrix}$$

with

$$Q_k = Q_k \begin{pmatrix} I & 0 \\ 0 & I - \alpha_k u_k u_k^T \end{pmatrix}.$$

However, the matrix Q_k is generally not explicitly formed, since it is available in product form if the vectors u are simply recorded in place of the columns they have been used to annihilate. This is the basic algorithm used in LINPACK [90] for computing the QR factorization of a matrix. The algorithm may be coded in terms of two of the Level 2 BLAS. To see this, just note that the operation of applying a transformation shown on the left-hand side of (5.8) may be broken into two steps:

$$z^T = u^T A \text{ (vector } \times \text{ matrix)} \tag{5.21}$$

and

$$\hat{A} = A - \alpha u z^T \text{ (rank-one modification)}.$$

5.5.2 Blocked Householder Method

One can develop a blocked version of the Householder method. The essential idea is to accumulate several of the Householder transformations into a single block transformation and then apply the result to annihilate several columns of the matrix at once. Both a left-looking and right-looking version of this algorithm can be constructed. We present the right-looking version.

To see how to construct such an aggregated transformation, note that (in general) it is possible to show that the product of k Householder transformations may be written as

$$\Pi(I - \alpha_j u_j u_j^T) = I - YTY^T,$$

where $Y = (u_1, u_2, ..., u_k)$ and T is an upper triangular matrix. This fact is easily verified by induction. If one has $V = I - YTY^T$, then

$$V(I - \alpha u u^T) = I - (Y, u)\begin{pmatrix} T & h \\ 0 & \alpha \end{pmatrix}(Y, u), \tag{5.22}$$

where $h = \alpha T W^T u$. One may also represent and compute this blocked transformation in a *WY representation* where $W = YT$. This latter representation, developed by Bischof and Van Loan [38], was later modified by Schreiber and Van Loan [278] into the *compact WY representation* presented above. The compact representation has the advantage of requiring less storage. The formation of

$$Vw = w - YT(Y^T w)$$

is rich in matrix-vector operations. One may accumulate the desired number of Householder transformations and then apply this block orthogonal transformation to obtain

$$(I - YTY^T)A = A - YT(Y^T A) = \begin{pmatrix} R_k & S_k^T \\ 0 & A_k \end{pmatrix}. \tag{5.23}$$

Of course, only the last $n-k$ columns of A would be involved in this computation, with the first k columns computed as in the point algorithm. An interesting, but minor point in practice is that the block algorithm will require more floating-point operations than the point algorithm since it requires $O(mk^2)$ additional operations to accumulate the block factorization and $k^2/2$ extra storage locations for the matrix T. Typically, however, for all but the final stages of the factorization, m is much greater than k. Therefore, the contribution of these terms is of low order in comparison to the primary factorization costs. This situation does introduce some partitioning and load-balance problems in terms of parallel processing. Typically, the later stages of any triangular factorization will create a serial bottleneck, given the low order of the matrix that remains to be factored. Thus, the vector lengths will be short and the matrix sizes small for the final few factorization steps. This situation brings up the question of when to discontinue the block algorithm in favor of the point algorithm. It also brings up the problem of varying the block size throughout the factorization. These issues are discussed in some detail in [38].

5.5.3 Typical Performance of the Blocked Householder Factorization

Table 5.5.3 shows the performance in Mflops of four variants of the QR decomposition on one processor of a CRAY-2. The two block variants are SQRBR, a block right-looking algorithm in which a block Householder matrix is computed and immediately applied to the rest of the matrix as a rank-k update, and SQRBL, a block left-looking variant in which the previous updates are first applied to the current block column before the next block Householder matrix is computed. SQR2 is the unblocked Level 2 BLAS variant, and SQRDC is the Level 1 BLAS variant from LINPACK. We see that the blocked variants surpass the unblocked variant in performance only for matrices of order greater than 200. This is a result of a larger operation count in the blocked version.

Extra work is required to compute the elements of T, but once again this is compensated for by the greater speed of applying the block form. Table 5.7 summarizes results obtained with the LAPACK routine SGEQRF.

Table 5.6: Performance in Mflops of QR Variants, $k = 64$ (CRAY-2/S, 1 processor)

	Matrix size $m = n$				
QR variant	100	200	300	400	500
LINPACK – SQRDC (vector version)	24	41	55	66	76
SQR2 (matrix-vector version)	144	215	242	251	255
SQRBR (matrix version - right looking)	106	209	269	306	328
SQRBL (matrix version - left looking)	102	198	258	293	316

Table 5.7: Speed in megaflops of SGEQRF/DGEQRF for square matrices of order n

	No. of	Block	Values of n	
	processors	size	100	1000
CONVEX C-4640	1	64	81	521
CONVEX C-4640	4	64	94	1204
CRAY C90	1	128	384	859
CRAY C90	16	128	390	7641
DEC 3000-500X Alpha	1	32	50	86
IBM POWER2 model 590	1	32	108	208
IBM RISC Sys/6000-550	1	32	30	61
SGI POWER CHALLENGE	1	64	61	190
SGI POWER CHALLENGE	4	64	39	342

5.6 Organization of the Modules

In this section we discuss the organization of code within the modules that make up the BLAS. A complete description would require far more detail than is appropriate for this book. Therefore, we shall limit ourselves to a discussion of matrix-vector and matrix-matrix products. Our objective is to give a generic idea of the considerations required for enhanced performance.

5.6.1 Matrix-Vector Product

We begin with some of the considerations that are important when coding the matrix-vector product module. The other modules require similar techniques. For vector machines such as the Cray series, the matrix-vector operation should be coded in the form

$$y(1:m) = y(1:m) + M(1:m,j)x(j) \text{ for } j = 1,2,...,n. \tag{5.24}$$

The $1 : m$ in the first entry implies that this is a column operation. The intent here is to reserve a vector register for the result while the columns of M are successively read into vector registers, multiplied by the corresponding component of x, and then added to the result register in place. In terms of ratios of data movement to floating-point operations, this arrangement is most favorable. It involves one vector move for two vector floating-point operations. The advantage of using the matrix-vector operation is clear when we compare the result to the three vector moves required to get the same two floating-point operations when a sequence of SAXPY operations is used. This operation is sometimes referred to as a GAXPY operation [96].

This arrangement is perhaps inappropriate for a parallel machine, however, because one would have to synchronize the access to y by each of the processes, and such synchronization would cause busy waiting to occur. One might do better to partition the vector y and the rows of the matrix M into blocks:

$$\begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_k \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_k \end{pmatrix} + \begin{pmatrix} M_1 \\ M_2 \\ . \\ . \\ . \\ M_k \end{pmatrix} \times x$$

and to self-schedule individual vector operations on each of the blocks in parallel:

$$y_i = y_i + M_i x \text{ for } i = 1,2,...,k.$$

That is, the subproblem indexed by i is picked up by a processor as it becomes available, and the entire matrix-vector product is reported done when all of these subproblems have been completed.

If the parallel machine has vector capabilities on each of the processors, this partitioning introduces shorter vectors and defeats the potential of the vector capabilities for small- to medium-size matrices. A better way to partition in this case is

$$y = y + (\hat{M}_1, \hat{M}_2, ..., \hat{M}_k) \begin{pmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_k \end{pmatrix}.$$

Again, subproblems are computed by individual processors. However, in this scheme, we must either synchronize the contribution of adding in each term $\hat{M}_i x_i$ or write each of these into temporary locations and hold them until all are complete before adding them to get the final result.

This scheme does prove to be effective for increasing the performance of the factorization subroutines on the smaller (order less than 100) matrices. During the final stages of the LU factorization, vector lengths become short regardless of matrix size. For the smaller matrices, subproblems with vector lengths that are below a certain performance level represent a larger percentage of the calculation. This problem is magnified when the rowwise partitioning is used.

5.6.2 Matrix-Matrix Product

Matrix-matrix operations offer the proper level of modularity for performance and transportability across a wide range of computer architectures including parallel machines with memory hierarchy. This enhanced performance is primarily due to a greater opportunity for reuse of data. There are numerous ways to accomplish this reuse of data to reduce memory traffic and to increase the ratio of floating-point operations to data movement. Here we present one way.

Consider the operation

$$Y \leftarrow Y + MX,$$

where Y is an $m \times n$ matrix, M is an $m \times p$ matrix, and X is a $p \times n$ matrix. Let us assume that $p+1$ vector registers are available (typically $p+1 = 8$) . The idea will be to hold p columns of M in p vector registers and successively compute the columns of Y:

$$Y(1:m, j) = Y(1:m, j) + sum_{k=1}^{p} M(1:m, k) X(k, j) \text{ for } j = 1, 2, ..., n. \tag{5.25}$$

One register is used to hold successive columns of Y. As the computation proceeds across the n columns, there are asymptotically $2p$ flops per 2 vector data accesses (reading and writing the jth column of Y).

If a cache is involved and we can rely on p columns of M remaining in cache once they are referenced, then it might be advantageous to set p to the length of a vector register (typically 32 or 64 words). In this case one would achieve a ratio of $2p$ flops per 3 vector data accesses (reading and writing the jth column of Y and reading the jth column of X), assuming that the rate of access to data in cache is comparable to that in registers.

Within this scheme, blocking of M in order to maintain a favorable cache *hit ratio* is important. We may exploit parallelism through blocking Y and M by rows and assigning each block to a separate processor; this requires only a *fork-join* synchronization. Alternatively, one may block Y and X by columns and assign each of these blocks to a separate processor. As in the matrix-vector operation, either fine-grain synchronization or temporary storage will be required when the blocking for parallelism is by columns.

5.6.3 Typical Performance for Parallel Processing

Blocked algorithms can exploit the parallel-processing capabilities of many high-performance machines, as described in the previous section. The basic idea is to leave the blocked algorithms unchanged and to introduce parallel processing at the Level 3 BLAS layer. The Level 3 BLAS offer greater scope than the Level 1 or 2 BLAS for exploiting parallel processors on shared-memory machines, since more operations are performed in each call.

Tables 5.6.3 and 5.6.3 give results for the CRAY Y-MP/8 and CRAY-2/4 for various numbers of processors when parallel processing is being exploited at the Level 3 BLAS layer only. Here the blocksize is 64 for LU decomposition and 16 for the QR factorization. The maximum speed of a single processor of a CRAY Y-MP is 333 Mflops. Thus, we see that for large-enough matrix dimensions, the single-processor code runs at 90% efficiency. When all 8 processors are used, the code attains 73% to 80% efficiency.

As can be seen, using parallel processing for small-order matrices results in a “slow down” in performance over the single-processor run. For sufficiently large matrices, however, the gain can be substantial—approximately 2 Gflops.

5.6.4 Benefits

Abundant evidence has already been given to verify the viability of these schemes for a variety of parallel and vector architectures. In addition to the computational evidence, several factors support

Table 5.8: Timing Results for a CRAY Y-MP/8 (Mflops)

Algorithm		Matrix size n 32	64	128	256	512	1024
LU	(1 proc)	40	108	195	260	290	304
	(2 proc)	32	91	229	408	532	588
	(4 proc)	32	90	260	588	914	1097
	(8 proc)	32	90	205	375	1039	1974
QR	(1 proc)	54	139	225	275	294	301
	(2 proc)	50	134	256	391	505	562
	(4 proc)	50	136	292	612	891	1060
	(8 proc)	50	133	328	807	1476	1937

Table 5.9: Timing Results for a CRAY 2-S/4 (Mflops)

Algorithm		Matrix size n 32	64	128	256	512	1024
LU	(1 proc)	28	81	152	238	320	381
	(2 proc)	17	54	145	327	540	715
	(4 proc)	17	55	160	394	865	1290
QR	(1 proc)	43	115	195	211	258	265
	(2 proc)	35	98	195	385	576	689
	(4 proc)	35	97	220	528	941	1122

the use of a modular approach. We can easily construct the standard algorithms in linear algebra from these types of module. The operations are simple and yet encompass enough computation that they can be vectorized and also parallelized at a reasonable level of granularity [102]. Finally, the modules can be constructed in such a way that they hide all of the machine intrinsics required to invoke parallel computation, thereby shielding the user from being concerned with any machine-specific changes to the library.

5.7 LAPACK

A collaborative effort is under way to develop a transportable linear algebra library in Fortran 77 [12]. The library provides a uniform set of subroutines to solve the most common linear algebra problems and runs efficiently on a wide range of high-performance computers.

Specifically, the LAPACK library (shorthand for Linear Algebra Package) provides routines for solving systems of simultaneous linear equations, least-squares solutions of overdetermined systems of equations, and eigenvalue problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) will also be provided, as will related computations such as reordering of the factorizations and condition numbers (or estimates thereof). Dense and banded matrices are provided for, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices.

This library is based on the successful EISPACK [290, 164] and LINPACK [90] libraries, integrating the two sets of algorithms into a unified, systematic library. A great deal of effort was expended to incorporate design methodologies and algorithms that make the LAPACK codes more appropriate for today's high-performance architectures. The LINPACK and EISPACK codes were written in a fashion that, for the most part, ignored the cost of data movement. As seen in Chapters 3 and 4, most of today's high-performance machines, however, incorporate a memory hierarchy [88, 208, 295] to even out the difference in speed of memory accesses and vectorized floating-point operations. As a result, codes must be careful about reusing data in order not to run at memory speed instead of floating-point speed. LAPACK codes were carefully restructured to reuse as much data as possible in order to reduce the cost of data movement. A further improvement is the incorporation of new and improved algorithms for the solution of eigenvalue problems [81, 103].

LAPACK is efficient and transportable across a wide range of computing environments, with special emphasis on modern high-performance computers. It improves efficiency in two ways. First, the developers have restructured most of the algorithms in LINPACK and EISPACK in terms of calls to a small number of extended BLAS each of which implements a block matrix operation such as matrix multiplication, rank-k matrix updates, and the solution of triangular systems. These block operations can be optimized for each architecture, but the numerical algorithms that call them will be portable. Second, the developers have implemented a class of recent divide-and-conquer algorithms for various eigenvalue problems [103, 211, 13].

The target machines are high-performance architectures with one or more processors, usually with a vector-processing facility or RISC based processors. The class includes all of the most powerful shared machines currently available and in use for general-purpose scientific computing: CRAY T90, CRAY J90, Fujitsu VPP, Hitachi, HP-Convex, IBM Power 2, SGI Power Challenge, NEC SX-4. It is assumed that the number of processors is modest (no more than 100, say). In cases where an algorithm can be restructured into block form in several different ways, all with different performance characteristics, the LAPACK project has chosen the structure that provides the best "average" performance over the range of target machines. It is also hoped that the library will perform well on a wider class of shared-memory machines, and the developers are actively exploring the applicability of their approach to distributed-memory machines.

LAPACK contains a variety of linear algebra algorithms that are much more accurate than their predecessors. In general, they replace absolute error bounds (either on the backward or forward error) with relative error bounds, and hence better respect the sparsity and scaling structure of the original problems.

The library is in the public domain, freely distributed over *netlib* [95], just like LINPACK, EISPACK, and many other libraries.

5.8 ScaLAPACK

ScaLAPACK is a library of high performance linear algebra routines for distributed memory MIMD computers. It is a continuation of the LAPACK project, which designed and produced analogous software for workstations, vector supercomputers, and shared memory parallel computers. The goals of the project are efficiency (to run as fast as possible), scalability (as the problem size and number of processors grow), reliability (including error bounds), portability (across all important parallel machines), flexibility (so users can construct new routines from well-designed parts), and ease-of-use (by making LAPACK and ScaLAPACK look as similar as possible). Many of these goals, particularly portability, are aided by developing and promoting *standards*, especially for low-level communication and computation routines. We have been successful in attaining these goals, limiting most machine dependencies to two standard libraries called the BLAS, or Basic Linear Algebra Subroutines, and BLACS, or Basic Linear Algebra Communication Subroutines. ScaLAPACK will run on any machine where both the BLAS and the BLACS are available.

The ScaLAPACK software library is extending the LAPACK library to run scalably on MIMD, distributed memory, concurrent computers [55]. For such machines the memory hierarchy includes the off-processor memory of other processors, in addition to the hierarchy of registers, cache, and

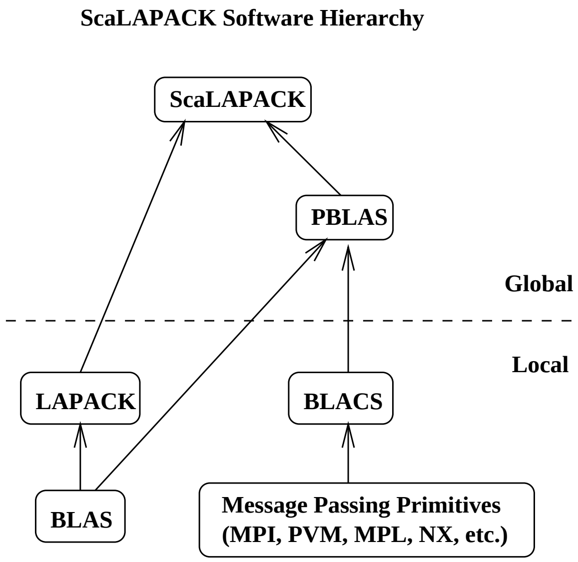

Figure 5.3: ScaLAPACK Software Hierarchy

local memory on each processor. Like LAPACK, the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. The fundamental building blocks of the ScaLAPACK library are a set of Basic Linear Algebra Communication Subprograms (BLACS) [87] for communication tasks that arise frequently in parallel linear algebra computations, and the Parallel Basic Linear Algebra Subprograms (PBLAS), which are a distributed memory version of the sequential BLAS. In the ScaLAPACK routines, all interprocessor communication occurs within the PBLAS and the BLACS, so the source code of the top software layer of ScaLAPACK looks very similar to that of LAPACK.

Figure 5.3 describes the ScaLAPACK software hierarchy. The components below the dashed line, labeled Local, are called on a single processor, with arguments stored on single processors only. The components above the line, labeled Global, are synchronous parallel routines, whose arguments include matrices and vectors distributed in a 2D block cyclic layout across multiple processors.

5.8.1 The Basic Linear Algebra Communication Subprograms (BLACS)

The **BLACS** (Basic Linear Algebra Communication Subprograms) [87] are a message passing library designed for linear algebra. The computational model consists of a one or two dimensional grid of processes, where each process stores matrices and vectors. The BLACS include synchronous send/receive routines to send a matrix or submatrix from one process to another, to broadcast submatrices to many processes, or to compute global reductions (sums, maxima and minima). There are also routines to construct, change, or query the process grid. Since several ScaLAPACK algorithms require broadcasts or reductions among different subsets of processes, the BLACS permit a process to be a member of several overlapping or disjoint process grids, each one labeled by a *context*. Some message passing systems, such as MPI [253], also include this context concept. The BLACS provide facilities for safe inter-operation of system contexts and BLACS contexts.

5.8.2 PBLAS

In order to simplify the design of ScaLAPACK, and because the BLAS have proven to be very useful tools outside LAPACK, we chose to build a Parallel BLAS, or PBLAS [56], whose interface is as similar to the BLAS as possible. This decision has permitted the ScaLAPACK code to be quite similar, and sometimes nearly identical, to the analogous LAPACK code. Only one substantially new routine was added to the PBLAS, matrix transposition, since this is a complicated operation in a distributed memory environment [57].

We hope that the PBLAS will provide a distributed memory standard, just as the BLAS have provided a shared memory standard. This would simplify and encourage the development of high performance and portable parallel numerical software, as well as providing manufacturers with a small set of routines to be optimized. The acceptance of the PBLAS requires reasonable compromises among competing goals of functionality and simplicity.

The PBLAS operate on matrices distributed in a 2D block cyclic layout. Since such a data layout requires many parameters to fully describe the distributed matrix, we have chosen a more object-oriented approach, and encapsulated these parameters in an integer array called an *array descriptor*. An array descriptor includes

(1) the descriptor type,
(2) the BLACS context (see Section 5.8.1),
(3) the number of rows in the distributed matrix,
(4) the number of columns in the distributed matrix,
(5) the row block size (r in Section),
(6) the column block size (c in Section),
(7) the process row over which the first row of the matrix is distributed,
(8) the process column over which the first column of the matrix is distributed,
(9) the leading dimension of the local array storing the local blocks.

By using this descriptor, a call to a PBLAS routine is very similar to a call to the corresponding BLAS routine.

```
CALL DGEMM ( TRANSA, TRANSB, M, N, K, ALPHA,
             A( IA, JA ), LDA,
             B( IB, JB ), LDB, BETA,
             C( IC, JC ), LDC )

CALL PDGEMM( TRANSA, TRANSB, M, N, K, ALPHA,
             A, IA, JA, DESC_A,
             B, JB, DESC_B, BETA,
             C, IC, JC, DESC_C )
```

`DGEMM` computes `C = BETA * C + ALPHA * op( A ) * op( B )`, where `op(A)` is either `A` or its transpose depending on `TRANSA`, `op(B)` is similar, `op(A)` is `M`-by-`K`, and `op(B)` is `K`-by-`N`. `PDGEMM` is the same, with the exception of the way in which submatrices are specified. To pass the submatrix starting at `A(IA,JA)` to `DGEMM`, for example, the actual argument corresponding to the formal argument `A` would simply be `A(IA,JA)`. `PDGEMM`, on the other hand, needs to understand the global storage scheme of `A` to extract the correct submatrix, so `IA` and `JA` must be passed in separately. `DESC_A` is the array descriptor for `A`. The parameters describing the matrix operands `B` and `C` are analogous to those describing `A`. In a truly object-oriented environment matrices and `DESC_A` would be synonymous. However, this would require language support, and detract from portability.

The presence of a context associated with every distributed matrix provides the ability to have separate "universes" of message passing. The use of separate communication contexts by distinct libraries (or distinct library invocations) such as the PBLAS insulates communication internal to the library from external communication. When more than one descriptor array is present in the argument list of a routine in the PBLAS, it is required that the individual BLACS context entries must be equal. In other words, the PBLAS do not perform "inter-context" operations.

We have not included specialized routines to take advantage of packed storage schemes for symmetric, Hermitian, or triangular matrices, nor of compact storage schemes for banded matrices [56].

5.8.3 ScaLAPACK sample code

Given the infrastructure described above, the ScaLAPACK version (PDGETRF) of the LU decomposition is nearly identical to its LAPACK version (DGETRF).

SEQUENTIAL LU FACTORIZATION CODE

```
      DO 20 J = 1, MIN( M, N ), NB
         JB = MIN( MIN( M, N )-J+1, NB )

         Factor diagonal and subdiagonal blocks and test for exact
         singularity.

         CALL DGETF2( M-J+1, JB, A( J, J ), LDA, IPIV( J ),
     $                IINFO )

         Adjust INFO and the pivot indices.

         IF( INFO.EQ.0 .AND. IINFO.GT.0 ) INFO = IINFO + J - 1
         DO 10 I = J, MIN( M, J+JB-1 )
            IPIV( I ) = J - 1 + IPIV( I )
   10    CONTINUE

         Apply interchanges to columns 1:J-1.

         CALL DLASWP( J-1, A, LDA, J, J+JB-1, IPIV, 1 )

         IF( J+JB.LE.N ) THEN

            Apply interchanges to columns J+JB:N.

            CALL DLASWP( N-J-JB+1, A( 1, J+JB ), LDA, J, J+JB-1,
     $                   IPIV, 1 )

            Compute block row of U.

            CALL DTRSM( 'Left', 'Lower', 'No transpose', 'Unit',
     $                  JB, N-J-JB+1, ONE, A( J, J ), LDA,
     $                  A( J, J+JB ), LDA )
            IF( J+JB.LE.M ) THEN

               Update trailing submatrix.

               CALL DGEMM( 'No transpose', 'No transpose',
     $                     M-J-JB+1, N-J-JB+1, JB, -ONE,
     $                     A( J+JB, J ), LDA, A( J, J+JB ), LDA,
     $                     ONE, A( J+JB, J+JB ), LDA )
            END IF
         END IF
   20 CONTINUE
```

PARALLEL LU FACTORIZATION CODE

```
      DO 10 J = JA, JA+MIN(M,N)-1, DESCA( 5 )
         JB = MIN( MIN(M,N)-J+JA, DESCA( 5 ) )
         I = IA + J - JA

         Factor diagonal and subdiagonal blocks and test for exact
         singularity.

         CALL PDGETF2( M-J+JA, JB, A, I, J, DESCA, IPIV, IINFO )

         Adjust INFO and the pivot indices.

         IF( INFO.EQ.0 .AND. IINFO.GT.0 )
     $      INFO = IINFO + J - JA

         Apply interchanges to columns JA:J-JA.

         CALL PDLASWP( 'Forward', 'Rows', J-JA, A, IA, JA, DESCA,
     $                 J, J+JB-1, IPIV )

         IF( J-JA+JB+1.LE.N ) THEN

            Apply interchanges to columns J+JB:JA+N-1.

            CALL PDLASWP( 'Forward', 'Rows', N-J-JB+JA, A, IA,
     $                    J+JB, DESCA, J, J+JB-1, IPIV )

            Compute block row of U.

            CALL PDTRSM( 'Left', 'Lower', 'No transpose', 'Unit',
     $                   JB, N-J-JB+JA, ONE, A, I, J, DESCA, A, I,
     $                   J+JB, DESCA )
            IF( J-JA+JB+1.LE.M ) THEN

               Update trailing submatrix.

               CALL PDGEMM( 'No transpose', 'No transpose',
     $                      M-J-JB+JA, N-J-JB+JA, JB, -ONE, A,
     $                      I+JB, J, DESCA, A, I, J+JB, DESCA,
     $                      ONE, A, I+JB, J+JB, DESCA )
            END IF
         END IF
   10 CONTINUE
```

Future releases of the ScaLAPACK library will extend the flexibility of the PBLAS and increase the functionality of the library to include routines for the solution of general banded linear systems, general and symmetric positive definite tridiagonal systems, rank-deficient linear least squares problems, generalized linear least squares problems, and the singular value decomposition. A draft of the ScaLAPACK Users' Guide is currently available on *netlib* and we plan to have the final draft ready for publication at the end of 1996.

A Fortran 90 interface for the LAPACK library is currently under development. This interface will provide an improved user-interface to the package by taking advantage of the considerable simplifications of the Fortran 90 language such as assumed-shape arrays, optional arguments, and generic interfaces.

As HPF compilers have recently become available, work is currently in progress to produce an HPF interface for the ScaLAPACK library. HPF provides a much more convenient distributed memory interface than can be provided in processor-level languages such as Fortran77 or C. This interface to a subset of the ScaLAPACK routines will be available at the time of the next ScaLAPACK release. With the increased generality of the PBLAS to operate on partial first blocks, ScaLAPACK will be fully compatible with HPF [219].

Also underway is an effort to establish a BLAS Technical Forum to consider expanding the BLAS in a number of ways in light of modern software, language, and hardware developments. The goals of the forum are to stimulate thought and discussion, and define functionality and future development of a set of standards for basic matrix and data structures. Dense and sparse BLAS are considered, as well as calling sequences for a set of low-level computational kernels for the parallel and sequential settings. For more information on the BLAS Technical Forum refer to the URL:

http://www.netlib.org/utk/papers/blast-forum.html

The EISPACK, LINPACK, LAPACK, BLACS and SCALAPACK linear algebra libraries are in the public domain. The software and documentation can be retrieved from *netlib* (`http://www.netlib.org`).

Chapter 6

Direct Solution of Sparse Linear Systems

In this chapter, we discuss the direct solution of linear systems

$$Ax = b, \tag{6.1}$$

where the coefficient matrix A is large and sparse. Sparse systems arise in very many application areas. We list just a few such areas in Table 6.1.

Table 6.1: **A list of some application areas for sparse matrices**

acoustic scattering	4	demography	3	network flow	1
air traffic control	1	economics	11	numerical analysis	4
astrophysics	2	electric power	18	oceanography	4
biochemical	2	electrical engineering	1	petroleum engineering	19
chemical eng.	16	finite elements	50	reactor modeling	3
chemical kinetics	14	fluid flow	6	statistics	1
circuit physics	1	laser optics	1	structural engineering	95
computer simulation	7	linear programming	16	survey data	11

This table, reproduced from [122], shows the number of matrices from each discipline present in the Harwell-Boeing Sparse Matrix Test Collection. This standard set of test problems is currently being upgraded to a new Collection called the Rutherford-Boeing Sparse Matrix Test Collection [124] that will include far larger systems and matrices from an even wider range of disciplines. This new Collection will be available from `netlib` and the Matrix Market (`http://math.nist.gov/MatrixMarket`).

Some of the algorithms that we will describe may appear complicated, but it is important to remember that we are primarily concerned with methods based on Gaussian elimination. That is, most of our algorithms compute an LU factorization of a permutation of the coefficient matrix A, so that $PAQ = LU$, where P and Q are permutation matrices, and L and U are lower and upper triangular matrices, respectively. These factors are then used to solve the system (6.1) through the forward substitution $Ly = P^T b$ followed by the back substitution $U(Q^T x) = y$. When A is symmetric, this fact is reflected in the factors, and the decomposition becomes $PAP^T = LL^T$ (Cholesky factorization) or $PAP^T = LDL^T$ (needed for an indefinite matrix). This last decomposition is sometimes called root-free Cholesky. It is common to store the inverse of D rather than D itself in order to avoid divisions when using the factors to solve linear systems. Note that we have used the same symbol L in all three factorizations although they each represent a different lower triangular matrix.

The solution of (6.1) can be divided naturally into several phases. Although the exact subdivision will depend on the algorithm and software being used, a common subdivision is given by:

1. A preordering phase that exploits structure, for example a preordering to block triangular or bordered block diagonal form (see [120], for example).
2. An analysis phase where the matrix structure is analysed to produce a suitable ordering and data structures for efficient factorization.
3. A factorization phase where the numerical factorization is performed.
4. A solve phase where the factors are used to solve the system using forward and back substitution.

Some codes combine phases 2 and 3 so that numerical values are available when the ordering is being generated. Phase 3 (or the combined phase 2 and 3) usually requires the most computing time, while the solve phase is generally an order of magnitude faster. Note that the concept of a separate factorization phase, which may be performed on a different matrix to that originally analysed, is peculiar to sparse systems. In the case of dense matrices, only the combined analysis and factorization phase exists and, of course, phase 1 does not apply.

As we have seen in the earlier chapters, the solution of (6.1) where A, of order n, is considered as a dense matrix requires $O(n^2)$ storage and $O(n^3)$ floating-point arithmetic operations. Since we will typically be solving sparse systems that are of order several thousand or even several tens or hundreds of thousands, dense algorithms quickly become infeasible on both grounds of work and storage. The aim of sparse matrix algorithms is to solve equations of the form (6.1) in time and space proportional to $O(n) + O(\tau)$, for a matrix of order n with τ nonzeros. It is for this reason that sparse codes can become very complicated. Although there are cases where this linear target cannot be met, the complexity of sparse linear algebra is far less than in the dense case.

The study of algorithms for effecting such solution schemes when the matrix A is large and sparse is important not only for the problem in its own right, but also because the type of computation required makes this an ideal paradigm for large-scale scientific computing in general. In other words, a study of direct methods for sparse systems encapsulates many issues that appear widely in computational science and that are not so tractable in the context of really large scientific codes.

The principal issues can be summarized as follows:

1. Much of the computation is integer.
2. The data-handling problem is significant.
3. Storage is often a limiting factor, and auxiliary storage is frequently used.
4. Although the innermost loops are usually well defined, often a significant amount of time is spent in computations in other parts of the code.
5. The innermost loops can sometimes be very complicated.

Issues 1–3 are related to the manipulation of sparse data structures. The efficient implementation of techniques for handling these is of crucial importance in the solution of sparse matrices, and we discuss this in Section 6.1. Similar issues arise when handling large amounts of data in other large-scale scientific computing problems. Issues 2 and 4 serve to indicate the sharp contrast between sparse and non-sparse linear algebra. In code for large dense systems, well over 98 percent of the time (on a serial machine) is typically spent in the innermost loops, whereas a substantially lower fraction is spent in the innermost loops of sparse codes. The lack of dominance of a single loop is also characteristic of a wide range of large-scale applications.

Specifically, the data handling nearly always involves indirect addressing (see Section 3.5). This clearly has efficiency implications particularly for vector or parallel architectures. Much of our discussion on suitable techniques for such platforms is concerned with avoiding indirect addressing in the innermost loops of the numerical computation.

Another way in which the solution of sparse systems acts as a paradigm for a wider range of scientific computation is that it exhibits a hierarchy of parallelism that is typical of that existing in the wider case. This hierarchy comprises three levels:

- *System level.* This involves the underlying problem which, for example, may be a partial differential equation (PDE) or a large structures problem. In these cases, it is natural (perhaps even before discretization) to subdivide the problem into smaller subproblems, solve these independently, and combine the independent solutions through a small (usually dense) interconnecting problem. In the PDE case, this is done through domain decomposition; in the structures case, it is called substructuring. An analogue in discrete linear algebra is partitioning and tearing.

- *Matrix level.* At this level, parallelism is present because of sparsity in the matrix. A simple example lies in the solution of tridiagonal systems. In a structural sense no (direct) connection exists between variable 1 in equation 1 and variable n in equation n (the system is assumed to have order n), and so Gaussian elimination can be applied to both of these "pivots" simultaneously. The elimination can proceed, pivoting on entry 2 and $n-1$, then 3 and $n-2$ simultaneously so that the resulting factorization (known in LINPACK as the BABE algorithm) has a parallelism of two. This is not very exciting, although the amount of arithmetic is unchanged from the normal sequential factorization. However, sparsity allows us to pivot simultaneously on every other entry; and when this is continued in a nested fashion, the cyclic reduction (or nested dissection in this case also) algorithm results. Now we have only $\log n$ parallel steps, although the amount of arithmetic is about double that of the serial algorithm. This sparsity parallelism can be automatically exploited for any sparse matrix, as we discuss in Section 6.5.

- *Submatrix level.* This level is exploited in an identical way to Chapter 5, since we are here concerned with eliminations within dense submatrices of the overall sparse system. Thus, the techniques of the dense linear algebra case (e.g., Level 3 BLAS) can be used. The only problem is how to organize the sparse computation to yield operations on dense submatrices. This is easy with band, variable band, or frontal solvers (Section 6.4) but can also be extended, through multifrontal methods, to any sparse system (Section 6.5).

It is important to stress that the Basic Linear Algebra Subprograms (BLAS) that we envisage using at the submatrix level are just the dense matrix BLAS discussed in the earlier chapters of this book. There is much discussion of BLAS standards for sparse matrices, see for example [126], but this is aimed at the iterative methods that we discuss in the next chapter rather than for use in algorithms or software for direct methods.

Throughout this chapter, we illustrate our points by the results of numerical experiments using computer codes, most commonly from the Harwell Subroutine Library (HSL) [205]. Most of the matrix codes in the HSL can be identified by the first two characters "MA" in the subroutine or package name. We sometimes use artificially generated test problems, but most are taken from the Harwell-Boeing Sparse Matrix Test Collection [123]. These problems can be obtained by anonymous ftp from the directory pub/harwell_boeing of the machine matisa.cc.rl.ac.uk (130.246.8.22) at the Rutherford Appleton Laboratory. We have supplemented this test set by some matrices collected by Tim Davis, available by anonymous ftp from `ftp://ftp.cise.ufl.edu/pub/faculty/davis/matrices`.

6.1 Introduction to Direct Methods for Sparse Linear Systems

The methods that we consider for the solution of sparse linear equations can be grouped into four main categories: general techniques, frontal methods, multifrontal approaches, and supernodal algorithms. In this section, we introduce the algorithms and approaches and examine some basic operations on sparse matrices. In particular, we wish to draw attention to the features that are important in exploiting vector and parallel architectures. For background on these techniques, we recommend the book by Duff, Erisman, and Reid [120].

The study of sparse matrix techniques is in great part empirical, so throughout we illustrate points by reference to runs of actual codes. We discuss the availability of codes in the penultimate section of this chapter.

6.1.1 Four Approaches

We first consider (Section 6.2) a general approach typified by the HSL code MA48 [131] or Y12M [331]. The principal features of this approach are that numerical and sparsity pivoting are performed at the same time (so that dynamic data structures are used in the initial factorization) and that sparse data structures are used throughout—even in the inner loops. As we shall see, these features must be considered drawbacks with respect to vectorization and parallelism. The strength of the general approach is that it will give a satisfactory performance over a wide range of structures and is often the method of choice for very sparse unstructured problems. Some gains and simplification can be obtained if the matrix is symmetric or banded. We discuss these methods and algorithms in Section 6.3.

Frontal schemes can be regarded as an extension of band or variable-band schemes and will perform well on systems whose bandwidth or profile is small. The efficiency of such methods for solving grid-based problems (for example, discretizations of partial differential equations) will depend crucially on the underlying geometry of the problem. One can, however, write frontal codes so that any system can be solved; sparsity preservation is obtained from an initial ordering, and numerical pivoting can be performed within this ordering. A characteristic of frontal methods is that no indirect addressing is required in the innermost loops and so dense matrix kernels can be used. We use the HSL code MA42 [134, 137] to illustrate this approach in Section 6.4.

The class of techniques that we study in Section 6.5, is an extension of the frontal methods termed *multifrontal.* The extension permits efficiency for any matrix whose nonzero pattern is symmetric or nearly symmetric and allows any sparsity ordering techniques for symmetric systems to be used. The restriction to nearly symmetric patterns arises because the initial ordering is performed on the sparsity pattern of the Boolean sum of the patterns of A and A^T. The approach can, however, be used on any system. The first example of this was the HSL code MA37 [129] which was the basis for the later development of a code that uses the Level 3 BLAS and is also designed for shared memory parallel computers. This new HSL code is called MA41 [10]. As in the frontal method, multifrontal methods use dense matrices in the innermost loops so that indirect addressing is avoided. There is, however, more data movement than in the frontal scheme, and the innermost loops are not so dominant. In addition to the use of direct addressing, multifrontal methods differ from the first class of methods because the sparsity pivoting is usually separated from the numerical pivoting. However, there have been very recent adaptations of multifrontal methods for general unsymmetric systems [73] which combine the analysis and factorization phases.

Another way of avoiding or amortizing the cost of indirect addressing is to combine nodes into supernodes. This technique is discussed in Section 6.6.

6.1.2 Description of Sparse Data Structure

We continue this introductory section by describing the most common sparse data structure, which is the one used in most general-purpose codes. The structure for a row of the sparse matrix is illustrated in Figure 6.1. All rows are stored in the same way, with the real values and column indices in two arrays with a one-to-one correspondence between the arrays so that the real value in position k, say, is in the column indicated by the entry in position k of the column index array. A sparse matrix can then be stored as a collection of such sparse rows in two arrays; one integer, the other real. A third integer array is used to identify the location of the position in these two arrays of the data structure for each row. If the ith position in this third array held the position marked "pointer" in Figure 6.1, then entry a_{i,j_1} would have value ξ. Clearly, access to the entries in a row is straightforward, although indirect addressing is required to identify the column index of an entry.

We illustrate this scheme in detail using the example in Figure 6.2.

If we consider the matrix in Figure 6.2, we can hold each row as a packed sparse vector and the matrix as a collection of such vectors. For each member of the collection, we normally store an integer pointer to its start and the number of entries. Since we are thinking in terms of Fortran 77, a pointer is simply an array subscript indicating a position in an array. Thus, for example, the matrix of Figure 6.2 may be stored in Fortran arrays as shown in Table 6.2. Here LEN(i) contains the number of entries in row i, while IPTR(i) contains the location in arrays ICN and VALUE of

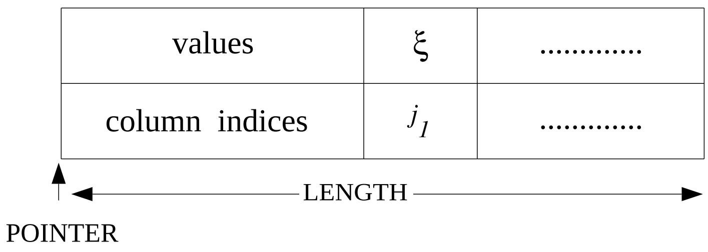

Figure 6.1: **Storage scheme for row of sparse matrix**

$$\begin{pmatrix} 1 & 0 & 0 & 4 \\ -1 & 0 & 3 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & -2 & 0 & -4 \end{pmatrix}$$

Figure 6.2: **A** 4×4 **sparse matrix**

the first entry in row i. For example, row 2 starts in position 3; referring to position 3 in ICN and VALUE, we find the (2,3) entry has value 3. Since LEN(2) = 2, the fourth position is also in row 2; specifically the (2,1) entry has value -1. Note that, with this scheme, the columns need not be held in order. This is important because, as we will see in Section 6.2, more entries will be added to the pattern of the matrix during the elimination process (called *fill-in*) and this will be facilitated by this scheme. A detailed discussion of this storage scheme is given in [122].

Note that there is redundancy in holding both row lengths (LEN) and row pointers (IPTR) when the rows are held contiguously in order, as in Table 6.2. In general, however, operations on the matrix will result in the rows not being in order and thus both arrays are required.

6.1.3 Manipulation of Sparse Data Structures

To give a flavor of the issues involved in the design of sparse matrix algorithms, we now examine a manipulation of sparse data structures that occurs commonly in LU factorization. The particular manipulation we consider is the addition of a multiple of one row (the pivot row) of the matrix to other rows (the non-pivot rows) where the matrix is stored in the row pointer/column index scheme

Table 6.2: **Matrix of Figure 6.2 stored as a collection of sparse row vectors**

Subscripts	1	2	3	4	5	6	7
LEN	2	2	1	2			
IPTR	1	3	5	6			
ICN	4	1	3	1	2	2	4
VALUE	4.	1.	3.	-1.	2.	-2.	-4.

just described. Assume that an integer array, of length n, IQ say, containing all positive entries is available (for example, this may be a permutation array), and that there is sufficient space to hold temporarily a second copy of the pivot row in sparse format. Assume that the second copy has been made. Then a possible scheme is as follows:

1. Scan the pivot row, and for each entry determine its column index from the ICN value. Set the corresponding entry in IQ to the negation of the position of that entry within the compact form of the pivot row. The original IQ entry is held in the ICN entry in the second copy of the pivot row.

   For each non-pivot row, do steps 2 and 3:

   2. Scan the non-pivot row. For each column index, check the corresponding entry in IQ. If it is positive, continue to the next entry in the non-pivot row. If the entry in IQ is negative, set it positive, and update the value of the corresponding entry in the non-pivot row (using the entry from the pivot row identified by the negation of the IQ entry).

   3. Scan the unaltered copy of the pivot row. If the corresponding IQ value is positive, set it negative. If it is negative, then there is fill-in to the non-pivot row. The new entry (a multiple of the pivot row entry identified by the negated IQ value) is added to the end of the non-pivot row, and we continue to the next entry in the pivot row.

4. Finally, reset IQ to its original values using information from both copies of the pivot row.

Figure 6.3 illustrates the situation before each step. Note that it is not necessary to keep each row in column order when using this scheme. Conceptually simpler sparse vector additions can result when they are kept in order, but the extra work to keep the columns in order can lead to inefficiencies.

Before 1

IQ $\quad i_1 \quad i_2 \quad i_3 \quad i_4 \quad i_5 \quad i_6 \quad i_7 \quad \ldots\ldots. \quad i_{n-1} \quad i_n$

Pivot row	A	α_1	α_2	α_3		+	second copy
	ICN	j_1	j_2	j_3			
Non-pivot row	A	β_1	β_2	β_3	β_4		
	ICN	j_2	j_3	j_4	j_5		

Before 2

IQ $\quad -2 \quad -1 \quad i_3 \quad -3 \quad i_5 \quad i_6 \quad i_7 \quad \ldots\ldots. \quad i_{n-1} \quad i_n$

Pivot row unchanged

Second copy of pivot row	A	α_1	α_2	α_3
	ICN	i_2	i_1	i_4

No change to non-pivot row

Before 3

IQ $\quad 2 \quad -1 \quad i_3 \quad 3 \quad i_5 \quad i_6 \quad i_7 \quad \ldots\ldots. \quad i_{n-1} \quad i_n$

Pivot row unchanged

Non-pivot row	A	$\beta_1 + \zeta\alpha_2$	$\beta_2 + \zeta\alpha_3$	β_3	β_4
	ICN	j_2	j_3	j_4	j_5

Before 4

IQ $\quad -2 \quad -1 \quad i_3 \quad -3 \quad i_5 \quad i_6 \quad i_7 \quad \ldots\ldots. \quad i_{n-1} \quad i_n$

Pivot row unchanged

Non-pivot row	A	$\beta_1 + \zeta\alpha_2$	$\beta_2 + \zeta\alpha_3$	β_3	β_4	$\zeta\alpha_1$
	ICN	j_2	j_3	j_4	j_5	j_1

Figure 6.3: **Sketch of steps involved when adding one sparse vector to another**

The important point about the rather complicated algorithm just described is that there are no scans of length n vectors. Thus, if this computation is performed at each major step of Gaussian elimination on a matrix of order n, there are (from this source) no $O(n^2)$ contributions to the overall work. Since our target in sparse calculations is to develop algorithms that are linear in the matrix order and number of nonzero entries, $O(n^2)$ calculations are a disaster and will dominate the computation if n is large enough.

In addition to avoiding such scans and permitting the indices to be held in any order, no real array of length n is needed. Indeed, the array IQ in Figure 6.3 can be used for any other purpose; our only assumption was that its entries were all positive. Another benefit is that we never check numerical values for zero (which might be the case if we expanded into a full-length real vector), so no confusion arises between explicitly held zeros and zeros not within the sparsity pattern. This point can be important when solving several systems with the same structure but different numerical values.

6.2 General Sparse Matrix Methods

We now discuss some fundamental aspects of sparse matrix manipulation and the use of sparsity exploiting techniques in Gaussian elimination on general sparse matrices.

6.2.1 Fill-in and sparsity ordering

A major concern when the matrix A is sparse is that the factors L and U will generally be denser than the original A.

After k steps of elimination on a matrix of order n, the reduced matrix is the lower $n-k$ by $n-k$ matrix modified from the original matrix according to the first k pivots steps. If we denote the entries of the original matrix by $a_{ij}^{(1)}$ and those of the reduced matrix after k stages of Gaussian elimination by $a_{ij}^{(k+1)}$, then fill-in is caused in Gaussian elimination if, in the basic operation

$$a_{ij}^{(k+1)} \leftarrow a_{ij}^{(k)} - a_{ik}^{(k)} \left[a_{kk}^{(k)}\right]^{-1} a_{kj}^{(k)}, \tag{6.2}$$

the entry in location (i,j) of the original A was zero. The ordering of the rows and columns of A can be important in preserving sparsity in the factors. Figure 6.4 gives an example of a case where ordering the rows and columns to preserve sparsity in Gaussian elimination is extremely effective. If pivots are chosen from the diagonal in the natural order, the reordered matrix preserves all zeros in the factorization, but the original order preserves none.

```
x x x x x x x x                       x             x
x x                                     x           x
x   x                                     x         x
x     x                                     x       x
x       x                                     x     x
x         x                                     x   x
x           x                                     x x
x             x                       x x x x x x x x

Original matrix                       Reordered matrix
```

Figure 6.4: **Original and reordered matrix**

Table 6.3: **Benefits of Sparsity on Matrix of Order 2021 with 7353 entries**

Procedure	Total storage (Kwords)	Flops (10^6)	Time (secs) CRAY J90
Treating system as dense	4084	5503	34.5
Storing and operating only on nonzero entries	71	1073	3.4
Using sparsity pivoting	14	42	0.9

Table 6.3 illustrates the gains one can obtain in the sparse case by ignoring all or most of the zero entries both in the original matrix and in the ensuing calculation. The second row of the table gives figures for sparse elimination without any sparsity-exploiting reordering, while the third row indicates that further substantial gains can be made when sparsity-exploiting orderings are used. In larger problems, because of the $O(n^3)$ and $O(n^2)$ complexity of dense codes for work and storage respectively, we might expect even more significant gains.

Another difference between dense and sparse systems arises when we consider the common case where a subsequent solution is required with a matrix of the same sparsity structure as a previously factored system. Whereas this has no relevance in the dense case, it does have a considerable influence in the sparse case since information from the first factorization can be used to simplify the second. Indeed, frequently most or all of the ordering and data organization can be done before any numerical factorization is performed.

Although it would not normally be sensible to use explicit inverses in the solution of dense systems of equations, it makes even less sense in the sparse case because (at least in the sense of the sparsity pattern) the computed inverse of an irreducible sparse matrix is always dense [119], whereas the factors can often be very sparse. Examples are a tridiagonal matrix and the arrowhead matrix shown in Figure 6.4. In short, it is particularly important that one does not use explicit inverses when dealing with large sparse matrices.

We illustrated the effect of ordering on sparsity preservation in Figure 6.4. A simple but effective strategy for maintaining sparsity is due to Markowitz [233]. At each stage of Gaussian elimination, he selects as a pivot the nonzero entry of the remaining reduced submatrix with the lowest product of number of other entries in its row and number of other entries in its column.

More precisely, before the kth major step of Gaussian elimination, let $r_i^{(k)}$ denote the number of entries in row i of the reduced $(n-k+1) \times (n-k+1)$ submatrix, Similarly let $c_j^{(k)}$ be the number of entries in column j. The Markowitz criterion chooses the entry $a_{ij}^{(k)}$ from the reduced submatrix to minimize the expression

$$(r_i^{(k)} - 1)(c_j^{(k)} - 1), \tag{6.3}$$

where $a_{ij}^{(k)}$ satisfies some numerical criteria also.

This strategy can be interpreted in several ways, for example, as choosing the pivot that modifies the least number of coefficients in the remaining submatrix. It may also be regarded as choosing the pivot that involves least multiplications and divisions. Finally, we may think of (6.3) as a means of bounding the fill-in at this stage, because it would be equal to the fill-in if all $(r_i^{(k)} - 1)(c_j^{(k)} - 1)$ modified entries were previously zero.

In general, for the Markowitz ordering strategy in the unsymmetric case, we need to establish a suitable control for numerical stability. In particular, we restrict the Markowitz selection to those pivot candidates satisfying the inequality

$$|a_{kk}^{(k)}| \geq u\ |a_{ik}^{(k)}|, \quad i \geq k, \tag{6.4}$$

where u is a preset threshold parameter in the range $0 < u \leq 1$.

If we look back at equation (6.2), we see that the effect of u is to restrict the maximum possible growth in the numerical value of a matrix entry in a single step of Gaussian elimination to $(1+1/u)$. Since it is possible to relate the entries of the backward error matrix E (that is, the LU factors are exact for the matrix $A+E$) to such growth by the formula

$$|e_{ij}| \leq 3.01\ \epsilon\ n\ maxa$$

where ϵ is the machine precision and $maxa$ the modulus of the largest entry encountered in the Gaussian elimination process then, since the value of $maxa$ can be affected by the value of u, changing u can affect the stability of our factorization. We illustrate this effect in Table 6.4 and remark that, in practice, a value of u of 0.1 has been found to provide a good compromise between maintaining stability and having the freedom to reorder to preserve sparsity.

Table 6.4: **Effect of Variation in Threshold Parameter** u
(matrix of order 541 with 4285 entries)

u	Entries in Factors	Error in solution
1.0	16767	3×10^{-9}
0.25	14249	6×10^{-10}
0.10	13660	4×10^{-9}
0.01	15045	1×10^{-5}
10^{-4}	16198	1×10^{2}
10^{-10}	16553	3×10^{23}

The results in Table 6.4 are without any iterative refinement, but even with such a device no meaningful answer is obtained in the case with u equal to 10^{-10}. Note that, at very low values of the threshold, the number of entries in the factors increases with decreasing threshold. This is somewhat counter-intuitive but indicates the difficulty in choosing later pivots because of poor choices made earlier. If in the current vogue for inexact factorizations, we consent to sacrifice accuracy of factorization for increase in sparsity, then this is done not through the threshold parameter u but rather through a drop tolerance parameter *tol*. Entries encountered during the factorization with a value less than *tol*, or less than *tol* times the largest in the row or column, are dropped from the structure, and an inexact or partial factorization of the matrix is obtained. We consider this use of partial factorizations as preconditioners in Chapter **??**.

6.2.2 Indirect addressing ... its affect and how to avoid it

Most recent vector processors have a facility for hardware indirect addressing, and one might be tempted to believe that all problems associated with indirect addressing have been overcome. For example, on the CRAY J90, the loop shown in Figure 6.5 (a sparse SAXPY) ran asymptotically at only 5.5 Mflop/s when hardware indirect addressing was inhibited but ran asymptotically at over

80 Mflop/s when it was not. (For further information on the behavior of a sparse SAXPY on a range of machines, see Section 4.5.)

```
      DO 100 I=1,M
         A(ICN(I)) = A(ICN(I)) + AMULT * W(I)
  100 CONTINUE
```

Figure 6.5: **Sparse SAXPY loop**

On the surface, the manufacturers' claims to have conquered the indirect addressing problem would seem vindicated, and we might believe that our sparse general codes would perform at about half the rate of a highly tuned dense matrix code. This reasoning has two flaws. The first lies in the $n_{1/2}$ value [199] for the sparse loops (i.e., the length of the loop required to attain half the maximum performance). This measure is directly related to the startup time. For the loop shown in Figure 6.5, the $n_{1/2}$ value on the CRAY J90 is about 50, which—relative to the typical order of sparse matrices being solved by direct methods (greater than 10,000)—is insignificant. However, the loop length for sparse calculations depends not on the order of the system but rather on the number of entries in the pivot row. We have done an extensive empirical examination of this number using the MA48 code on a wide range of applications. We show a representative sample of our results in Table 6.5. Except for the example from structural analysis (the second matrix of order 1224 in the table), this length is very low and, even in this extreme case, is only about equal to the $n_{1/2}$ value mentioned above. Thus the typical performance rate for the sparse inner loop is far from the asymptotic performance.

It should be added that there are matrices where the number of entries in each row is very high, or becomes so after fill-in. This would be the case in large structures problems and for most discretizations of elliptic partial differential equations. For example, the factors of the five-diagonal matrix from a five-point star discretization of the Laplace operator on a q by q grid could have about $6 \log_2 q$ entries in each row of the factors. Unfortunately, such matrices are very easy to generate and are thus sometimes overused in numerical experiments.

The second problem with the use of hardware indirect addressing in general sparse codes is that the amount of data manipulation in such a code means that a much lower proportion of the time is spent in the innermost loops than in code for dense matrices. Again we have performed an empirical study on MA48; we show these results in the last column of Table 6.5. The percentage given in that table is for the total time of three innermost loops in MA48, all at the same depth of nesting. We see that typically around 40-50 percent of the overall time on a CRAY J90 is spent in the innermost loops. Thus, even if these loops were made to run infinitely fast, a speedup of only at most a factor of two would be obtained: a good illustration of Amdahl's law.

The lack of locality of reference inherent in indirect addressing means that performance can also

Table 6.5: **Statistics from MA48 on a CRAY J90 for various matrices from different disciplines**

Matrix Order	Matrix Entries	Application area	Av. Length of Pivot Row	% Time in Inner Loops
680	2646	Atmospheric pollution	3.4	35
838	5424	Aerospace	24.5	49
1005	4813	Ship design	23.0	48
1107	5664	Computer simulation	19.2	53
1176	9874	Electronic circuit analysis	6.5	34
1224	9613	Oil reservoir modeling	12.7	49
1224	56126	Structural analysis	51.4	47
1374	8606	Nuclear engineering	16.7	49
1454	3377	Power systems networks	3.5	33
2021	7353	Chemical engineering	2.8	32
2205	14133	Oil reservoir modeling	4.0	52
2529	90158	Economic modeling	3.4	49

be handicapped or degraded on cache-based computers and parallel machines. One should mention, however, that more recent work effectively blocks the operations so that the indirect addressing is removed from the inner loops. If this approach (similar in many ways to the one we discuss in Section 6.5) is used, much higher computational rates can be achieved.

We conclude, therefore, that for general matrices vector indirect addressing is of limited assistance for today's general sparse codes. Even for general systems, however, advantage can be taken of vectorization by using a hybrid approach, where a dense matrix routine is used when fill-in has caused the reduced matrix to become sufficiently dense. That is, at some stage, it is not worth paying attention to sparsity. At this point, the reduced matrix can be expanded as a full matrix, any remaining zero entries are held explicitly, and a dense matrix code can be used to effect the subsequent decomposition. The resultant hybrid code should combine the advantages of the reduction in operations resulting from sparsity in the early stages with the high computational rates of a dense linear algebra code in the latter. The point at which such a switch is best made will, of course, depend both on the vector computer characteristics and on the relative efficiency of the sparse and dense codes.

6.2.3 Comparison with sparse codes

Every time there are improvements in algorithms or computer hardware that greatly improve the efficiency of dense matrix solvers, there are claims that sparse direct solvers are now obsolete and that these dense solvers should be used for sparse systems as well. There are two main reasons why this argument is severely flawed. The first is that, although the performance, as we illustrated in Table **??**, is remarkable, even for fairly modest values of order n of the matrix, the $O(n^3)$ complexity of a dense solver makes such calculations prohibitively expensive, if the $O(n^2)$ storage requirements have not already made solution impossible. However, a far more telling rebuttal is provided by current sparse direct solvers like the HSL code MA48. We see from the results in Table 6.6 that MA48 comfortably outperforms the LAPACK code SGESV on the CRAY Y-MP even for sparse matrices of fairly modest order. Clearly, the MA48 performance is dependent on the matrix structure, as is evident from the two runs on different matrices of order 1224, whereas the LAPACK code depends on the order and shows the expected $O(n^3)$ behavior, slightly tempered by the better performance of the Level 3 BLAS on larger matrices. However, even at its worst, MA48 comfortably outperforms SGESV. It should also be pointed out that subsequent factorizations of a matrix of the same pattern are much cheaper for the sparse code, for example the matrix BCSSTK27 can be refactorized by MA48 in 0.33 seconds, using the same pivot sequence as before.

The final nail in the coffin of dense matrix solvers lies in the fact that MA48 monitors the density of the reduced matrix as the elimination proceeds and switches to dense mode as indicated in Section 6.2.2. Thus, the highly efficient dense solvers, discussed in Chapter **??**, are now incorporated within the sparse code and so *a fortiori* the dense solvers can never beat the sparse code. This is not entirely true because there are some overheads associated with switching, but experiments with such an approach indicate that the switchover density for overall time minimization can often be low (typically 20 percent dense) and that gains of a factor of over four can be obtained even with unsophisticated dense matrix code [112]. Naturally, if we store the zeros of a reduced matrix, we might expect an increase in the storage requirements for the decomposition. Although the luxury of large memories on some current computers gives us ample scope for allowing such an increase, it is interesting to note that the increase in storage for floating-point numbers is to some extent compensated for by the reduction in storage for the accompanying integer index information.

6.2.4 Other approaches

In this section, we have concentrated on the approach typified by the codes MA48 [131] and Y12M [331]. While these are possibly the most common codes used to solve general sparse systems arising in a wide range of application areas, there are other algorithmic approaches and codes for solving general unsymmetric systems. One technique is to preorder the rows, then to choose pivots from each row in turn, first updating the incoming row according to the previous pivot steps and then

Table 6.6: **Comparison between MA48 and LAPACK (SGESV) on range of matrices from the Harwell-Boeing Sparse Matrix Test Collection.** Times for factorization and solution are in seconds on one processor of a CRAY Y-MP

Matrix	Order	Entries	MA48	SGESV
FS 680 3	680	2646	0.06	0.96
PORES 2	1224	9613	0.54	4.54
BCSSTK27	1224	56126	2.07	4.55
NNC1374	1374	8606	0.70	6.19
WEST2021	2021	7353	0.21	18.88
ORSREG 1	2205	14133	2.65	24.25
ORANI678	2529	90158	1.17	36.37

choosing the pivot by using a threshold criterion on the appropriate part of the updated row. If the columns are also preordered for sparsity, then an attempt is first made to see whether the diagonal entry in the reordered form is suitable. This approach is used by the codes NSPFAC and NSPIV of Sherman [280]. It is similar to the subsequent factorizations using the main approach of this section, and so is simple to code. It can, however, suffer badly from fill-in if a good initial ordering is not given or if numerical pivoting forbids keeping close to that ordering. Some of the multifrontal and supernodal approaches to sparse system solution also suffer in this way.

Another approach is to generate a data structure that, within a chosen column ordering, accommodates all possible pivot choices [173]. It is remarkable that this is sometimes not overly expensive in storage and has the benefit of good stability but within a subsequent static data structure. There are, of course, cases when the storage penalty is high.

Methods using a sparse QR factorization can be used for general unsymmetric systems. These are usually based on work of George and Heath [167] and first obtain the structure of R through a symbolic factorization of the structure of the normal equations matrix A^TA. It is common not to keep Q but to solve the system using the semi-normal equations

$$R^TRx = A^Tb,$$

with iterative refinement usually used to avoid numerical problems. A QR factorization can, of course, be used for least-squares problems and implementations have been developed using ideas similar to those discussed later in Section 6.5 for Gaussian elimination. Further discussion of QR and least squares is outside the scope of this chapter.

Another approach particularly designed for unsymmetric matrices has been developed by Davis and Yew [74]. Here a set of pivots is chosen simultaneously using Markowitz and threshold criteria so that if the set is permuted to the upper part of the matrix, the corresponding block will be diagonal and all operations corresponding to these pivots can be performed simultaneously. Indeed, it is possible to design an algorithm to perform the pivot search in parallel also. Subsequent sets of independent pivots are chosen in a similar manner until the reduced matrix becomes dense enough to switch to dense code, as discussed in Section 6.2. Alaghband [2] proposes a similar type of scheme.

One possibility for exploiting parallelism by general sparse direct methods is to use partitioning methods and, in particular, the bordered block triangular form [120]. This approach is discussed by Arioli and Duff [15], who indicate that reasonable speedups on machines with low levels of parallelism (4–8) are obtained fairly easily even on very difficult systems.

6.3 Methods for Symmetric Matrices and Band Systems

Elementary graph theory has been used as a powerful tool in the analysis and implementation of diagonal pivoting on symmetric matrices (see, for example, [170]). Although graph theory is sometimes overused in sparse Gaussian elimination, in certain instances it is a useful and powerful tool. We shall now explore one such area. For this illustration, we consider finite *undirected graphs*, G(V, E), which comprise a finite set of distinct *vertices* (or `nodes`) V and an *edge set* E consisting of unordered pairs of vertices. An edge $e \in$ E will be denoted by (u, v) for some $u, v \in$ V. The graph is *labeled* if the vertex set is in 1-1 correspondence with the integers $1, 2, ..., |\mathrm{V}|$, where $|\mathrm{V}|$ is the number of vertices in V. In this application of graph theory, the set E, by convention, does not include self-loops (edges of the form (u, u), $u \in$ V) or multiple edges. Thus, since the graph is undirected, edge (u, v) is equal to edge (v, u), and only one is held. A subgraph H(U, F) of G(V, E) has vertex set U $\subseteq$ V, and edge set F $\subseteq$ E. H(U, F) is fully connected if $(u, v) \in$ F for all $u, v \in$ U. With any symmetric matrix, of order n say, we can associate a labeled graph with n vertices such that there is an edge between vertex i and vertex j (edge (i, j)) if and only if entry a_{ij} (and, by symmetry, a_{ji}) of the matrix is nonzero. We give an example of a matrix and its associated graph in Figure 6.6. If the pattern of A is symmetric and we can be sure that diagonal pivots produce a stable factorization (the most important example is when A is symmetric and positive definite), then two benefits occur. We do not have to carry numerical values to check for stability, and the search for the pivot is simplified to finding i such that

$$r_i^{(k)} = \min_t r_t^{(k)}$$

and using $a_{ii}^{(k)}$ as pivot, where $r_t^{(k)}$, is the number of entries in row t of the reduced matrix, as in Equation (6.3). This scheme was introduced by Tinney and Walker [299] as their Scheme 2 and

$$\begin{pmatrix} \times & \times & & & \times & \\ \times & \times & & & & \times \\ & & \times & & & \\ & & & \times & \times & \\ \times & & & \times & \times & \\ & \times & & & & \times \end{pmatrix}$$

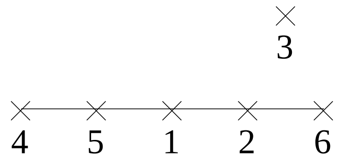

Figure 6.6: **Symmetric matrix and associated graph**

is normally termed the *minimum degree algorithm* because of its graph theoretic interpretation: in the graph associated with a symmetric sparse matrix, this strategy corresponds to choosing for the next elimination the vertex that has the fewest edges connected to it. Surprisingly (as we shall see later in this section), the algorithm can be implemented in the symmetric case without explicitly updating the sparsity pattern at each stage, a situation that greatly improves its performance.

The main benefits of using such a correspondence can be summarized as follows:

1. The structure of the graph is invariant under symmetric permutations of the matrix (they correspond merely to a relabeling of the vertices).
2. For mesh problems, there is usually a correspondence between the mesh and the graph of the resulting matrix. We can thus work more directly with the underlying structure.
3. We can represent *cliques* (fully connected subgraphs) in a graph by listing the vertices in a clique without storing all the interconnection edges.

6.3.1 The Clique Concept in Gaussian Elimination

To illustrate the importance of the clique concept in Gaussian elimination, we show in Figure 6.7 a matrix and its associated graph (also the underlying mesh, if, for example, the matrix represents the five-point discretization of the Laplacian operator). If the circled diagonal entry in the matrix were chosen as pivot (admittedly not a very sensible choice on sparsity grounds), then the resulting reduced matrix would have the dashed (pivot) row and column removed and have additional entries (fill-ins) in the checked positions and their symmetric counterparts. The corresponding changes to the graph cause the removal of the circled vertex and its adjacent edges and the addition of all the dashed edges shown.

$$
\begin{array}{cccccccccccc}
-4 & 1 & & & 1 & & & & & & & \\
1 & -4 & 1 & & & 1 & & & & & & \\
 & 1 & -4 & 1 & & \checkmark & 1 & \checkmark & & & \checkmark & \\
 & & 1 & -4 & & & & 1 & & & & \\
1 & & & & -4 & 1 & & & 1 & & & \\
 & 1 & & & 1 & -4 & 1 & \checkmark & & 1 & \checkmark & \\
 & & 1 & & & 1 & -4 & 1 & & & 1 & \\
 & & & 1 & & & 1 & -4 & & & \checkmark & 1 \\
 & & & & 1 & & & & -4 & 1 & & \\
 & & & & & 1 & & & 1 & -4 & 1 & \\
 & & & & & & 1 & & & 1 & -4 & 1 \\
 & & & & & & & 1 & & & 1 & -4
\end{array}
$$

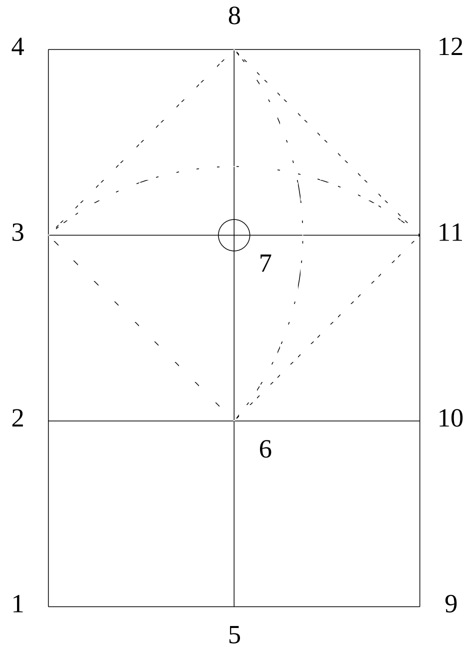

Figure 6.7: **Use of cliques in Gaussian elimination**

Thus, after the elimination of vertex 7, the vertices 3,6,8,11 form a clique because all vertices of the graph of the reduced matrix that were connected to the vertex associated with the pivot will become pairwise connected after that step of the elimination process. Although this clique formation has been observed for some time [258], it was more recently that techniques exploiting clique formation have been used in ordering algorithms. Our example clique has 4 vertices and 6 interconnecting edges; but, as the elimination progresses, this difference will be more marked, since a clique on q vertices has $q(q-1)/2$ edges corresponding to the off-diagonal entries in the associated dense submatrix.

If we are using the clique model for describing the elimination process then, when a vertex is selected as pivot, the cliques in which it lies are amalgamated. We illustrate clique amalgamation in Figure 6.8 where the circled element is being used as a pivot and the vertices in each rectangle are all pairwise connected. We do not show the edges internal to each rectangle, because we wish to reflect the storage and data manipulation scheme that will be used.

The two cliques (sometimes called elements or generalized elements by analogy with elements in the finite-element method) are held only as lists of constituent vertices (1,2,3,4,7,8,10,11,12,13)

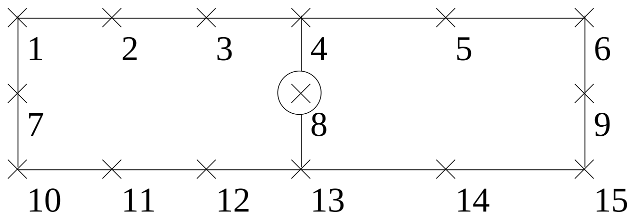

Figure 6.8: **Clique amalgamation**

and (4,5,6,8,9,13,14,15). The result of eliminating (that is pivoting on) vertex 8 is to amalgamate these two cliques. That is, after the elimination of vertex 8, the variables in both of these cliques will be pairwise connected to form a new clique given by the list (1,2,3,4,5,6,7,9,10,11,12,13,14,15). Not only is a list merge the only operation required, but the storage after elimination (for the single clique) will be less than before the elimination (for the two cliques). Since such a clique amalgamation can be performed in a time proportional to the number of vertices in the cliques concerned, both work and storage are linear rather than quadratic in the number of vertices in the cliques.

To summarize, two important aspects of this clique-based approach make it very significant and exciting. Earlier codes for computing minimum degree orderings mimicked the Gaussian elimination process. That is, they performed the elimination symbolically without doing any floating-point arithmetic. With the clique-based approach, we do not have to mimic the Gaussian elimination operations during this ordering phase, and finding the ordering can be significantly faster than the actual elimination. The second—and in many ways more important—aspect is that of storage, where the ordering can be implemented so that it requires only slightly more storage (a small multiple of n) over that for the matrix itself. Since the storage required for computing the ordering is independent of the fill-in and is known in advance, we can ensure that sufficient storage is available for successful completion of this phase. Thus, even for very large problems whose decomposition must necessarily be out-of-core, it may be possible to do an in-core ordering. When computing the ordering we can determine the storage and work required for subsequent factorization. This forecast could be crucial if we wish to know whether the in-core solution by a direct method is feasible or whether an out-of-core solver or iterative method must be used.

The dramatic effect of using a clique amalgamation approach in the implementation of minimum degree can be seen very clearly in the performance of the two HSL codes MA17 and MA27. The former code, written before clique amalgamation techniques were developed, runs two orders of magnitude slower than the HSL code MA27 which uses clique amalgamation. The improvement in minimum degree codes over the decade 1970 to 1980 is shown by [120].

6.3.2 Further comments on ordering schemes

Researchers have added numerous twists to the minimum degree scheme. Many of these are implementation refinements; they are discussed in a review by George and Liu [171]. Others are developed to compromise a strict minimum degree so that some other property is enhanced—for example, so that the ordering produced retains the good reduction in work afforded by minimum degree while, at the same time, yields an ordering more amenable to parallel implementation than strict minimum degree [121], [227].

Additionally, techniques have been developed to use an approximate minimum degree to further reduce the computation times for finding an ordering. The approximate minimum degree (AMD) algorithm of [7] is often much faster than the best implementation of minimum degree and produces an ordering of comparable quality.

There has been much work very recently in developing dissection techniques that produce orderings resulting in lower fill-in and operation count for a subsequent Cholesky factorization [267] over a wide range of problems. Much of this is motivated by finding good orderings for the elimination on parallel computers.

6.4 Frontal Methods

Frontal methods have their origins in the solution of finite-element problems from structural analysis. One of the earliest computer programs implementing the frontal method was that of Irons [209]. He considered only the case of symmetric positive-definite systems. The method can, however, be extended to unsymmetric systems [204] and need not be restricted to finite-element applications [110].

The usual way to describe the frontal method is to view its application to finite-element problems where the matrix A is expressed as a sum of contributions from the elements of a finite-element discretization. That is,

$$A = \sum_{l=1}^{m} A^{[l]}, \tag{6.5}$$

where $A^{[l]}$ is nonzero only in those rows and columns that correspond to variables in the lth element. If a_{ij} and $a_{ij}^{[l]}$ denote the (i,j)th entry of A and $A^{[l]}$, respectively, the basic assembly operation when forming A is of the form

$$a_{ij} \Leftarrow a_{ij} + a_{ij}^{[l]}. \tag{6.6}$$

It is evident that the basic operation in Gaussian elimination

$$a_{ij} \Leftarrow a_{ij} - a_{ip}[a_{pp}]^{-1}a_{pj} \tag{6.7}$$

may be performed as soon as all the terms in the triple product (6.7) are *fully summed* (that is, are involved in no more sums of the form (6.6)). The assembly and Gaussian elimination processes can therefore be interleaved and the matrix A is never assembled explicitly. This allows all intermediate working to be performed in a dense matrix, termed the *frontal matrix*, whose rows and columns correspond to variables that have not yet been eliminated but occur in at least one of the elements that have been assembled.

For non-element problems, the rows of A (equations) are added into the frontal matrix one at a time. A variable is regarded as fully summed whenever the equation in which it last appears is assembled. The frontal matrix will, in this case, be rectangular. A full discussion of the equation input can be found in [111].

We now describe the method for element input. After the assembly of an element, if the k fully summed variables are permuted to the first rows and columns of the frontal matrix, we can partition the frontal matrix F in the form

$$F = \begin{pmatrix} B & C \\ D & E \end{pmatrix}, \tag{6.8}$$

where B is a square matrix of order k and E is of order $r \times r$. Note that $k+r$ is equal to the current size of the frontal matrix, and $k \ll r$, in general. Typically, B might have order 10 to 20, while E is of order 200 to 500. The rows and columns of B, the rows of C, and the columns of D are fully summed; the variables in E are not yet fully summed. Pivots may be chosen from anywhere in B. For symmetric positive-definite systems, they can be taken from the diagonal in order but in the unsymmetric case, pivots must be chosen to satisfy a threshold criteria. This is discussed in [111].

It is also possible to view frontal methods as an extension of band or variable-band schemes. We do this in the following section.

6.4.1 Frontal methods ... link to band methods and numerical pivoting

A common method of organizing the factorization of a band matrix of order n with semibandwidth b is to allocate storage for a dense $b \times 2b-1$ matrix, which we call the frontal matrix, and to use this as a window that runs down the band as the elimination progresses. Thus, at the beginning, the frontal matrix holds rows 1 to b of the band system. This configuration enables the first pivotal step to be performed (including pivoting if this is required); and, if the pivot row is then moved out of the frontal matrix, row $b+1$ of the band matrix can be accommodated in the frontal matrix.

One can then perform the second pivotal step within the frontal matrix. Typically, a larger frontal matrix is used since greater efficiency can be obtained by moving blocks of rows at a time. It is then usually possible to perform several pivot steps within the one frontal matrix. The traditional reason for this implementation of a banded solver is for the solution of band systems by out-of-core methods since only the frontal matrix need be held in main storage. This use of auxiliary storage is also one of the principal features of a general frontal method.

This "windowing" method can be easily extended to variable-band matrices. In this case, the frontal matrix must have order at least $\max_{a_{ij}\neq 0}\{|i-j|\}$. Further extension to general matrices is possible by observing that any matrix can be viewed as a variable-band matrix. Here lies the main problem with this technique: for any arbitrary matrix with an arbitrary ordering, the required size for the frontal matrix may be very large. However, for discretizations of partial differential equations (whether by finite elements or finite differences), good orderings can usually be found (see, for example, [133] and [289]). In fact, recent experiments [136] show that there is a reasonably wide range of problems for which a frontal method could be the method of choice, particularly if the matrix is expressed as a sum of element matrices in unassembled form.

We limit our discussion here to the implementation of Gaussian elimination on the frontal matrix. The frontal matrix is of the form (6.8). The aim at this stage is to perform k steps of Gaussian elimination on the frontal matrix (choosing pivots from B), to store the factors $L_B U_B$ of B, DB^{-1}, and C on auxiliary storage, and to generate the Schur complement $E - DB^{-1}C$ for use at the next stage of the algorithm.

As we mentioned earlier, in the unsymmetric case, the pivots can be chosen from anywhere within B. In our approach [111], we use the standard sparse matrix technique of threshold pivoting, where $b_{ij} \in B$ is suitable as pivot only if

$$|b_{ij}| \geq u \max(\max_s |b_{sj}|, \max_s |d_{sj}|), \tag{6.9}$$

where u is a preset parameter in the range $0 < u \leq 1$.

Notice that, although we can only choose pivots from the block B, we must test potential pivot candidates for stability by comparing their magnitude with entries from B and from D. This means that large entries in D can prevent the selection of some pivots from B. Should this be the case, $k_1 \leq k$ steps of Gaussian elimination will be performed, and the resulting Schur complement $E - DB_1^{-1}C$, where B_1 is a square submatrix of B of order k_1, will have order $r + k - k_1$. Although this can increase the amount of work and storage required by the algorithm, the extra cost is typically very low, and all pivotal steps will eventually be performed since the final frontal matrix has a null E block (that is, $r = 0$).

An important aspect of frontal schemes is that all the elimination operations are performed within a dense matrix, so that kernels and techniques (including those for exploiting vectorization or parallelism) can be used on dense systems. It is also important that k is usually greater than 1, in which case more than one elimination is performed on the frontal matrix and Level 2 and Level 3 BLAS can be used as the computational kernel. Indeed, on some architectures (for example the SGI Power Challenge), it can be beneficial to increase the size of the B block by doing extra assemblies before the elimination stage, even if more floating-pont operations are then required to effect the factorization [61].

6.4.2 Vector Performance

We now illustrate some of the points just raised with numerical experiments. For the experimental runs, we use as our standard model problem an artificial finite-element problem designed to simulate those actually occurring in some CFD calculations. The elements are nine-node rectangular elements with nodes at the corners, mid-points of the sides, and center of the element. A parameter to the generation routine determines the number of variables per node which has been set to five for the runs in this chapter. The elements are arranged in a rectangular grid whose dimensions are given in the tables.

A code for frontal elimination, called MA32, was included in the Harwell Subroutine Library in 1981 [110]. This code was used extensively by people working with finite-elements particularly on the CRAY-2, then at Harwell. The MA32 code was substantially improved to produce the HSL code MA42 [134] that has a very similar interface and functionality to its predecessor but makes more extensive use of higher Level BLAS than the earlier routine. On a machine with fast Level 3 BLAS, the performance can be impressive. We illustrate this point by showing the performance of our standard test problem on one processor of a CRAY Y-MP, whose peak performance is 333 Mflop/s and on which the Level 3 BLAS matrix-matrix multiply routine SGEMM runs at 313 Mflop/s on sufficiently large matrices. It is important to realize that the Megaflop rates given in Table 6.7 include all overheads for the out-of-core working used by MA42.

Table 6.7: **Performance (in Mflop/s) of MA42 on one processor of a CRAY Y-MP on our standard test problem**

Dimension of element grid	16 x 16	32 x 32	48 x 48	64 x 64	96 x 96
Max order frontal matrix	195	355	515	675	995
Total order of problem	5445	21125	47045	83205	186245
Mflop/s	145	208	242	256	272

Although, as we see in Table 6.7, very high Megaflop rates can be achieved with the frontal scheme, this is often at the expense of more flops than are necessary to perform the sparse factorization. The usefulness of the frontal scheme is very dependent on this trade off between flops and Mflop/s. It may be possible to reorder the matrix to reduce the flop count and extend the applicability of frontal methods, using reordering algorithms similar to those for bandwidth reduction of assembled matrices [133]. As an example of the effect of reordering, the matrix LOCK3491 from the Harwell-Boeing Sparse Matrix Test Collection can be factorized by MA42 at a rate of 118 Mflop/s on a CRAY Y-MP. If the matrix is first reordered using the HSL subroutine MC43, the frontal code then runs at only 89 Mflop/s but the number of floating-point operations is reduced substantially so that only 1.07 seconds are now required for the factorization compared with 9.69 seconds for the unordered problem. This is a graphic illustration of the risk of placing too much emphasis on computational rates (Mflop/s) when evaluating the performance of algorithms. Additionally, although the effect of reordering can be quite dramatic, it is very similar to bandwidth reduction and so will not enable the efficient use of frontal schemes on general systems.

In addition to the problem that far more arithmetic can be done than is required for the numerical factorization, a second problem with the frontal scheme is that there is little scope for parallelism other than that which can be obtained within the higher level BLAS. Indeed, if we view the factorization in terms of a computational tree where nodes correspond to factorizations of frontal matrices of the form (6.8) and edges correspond to the transfer of the Schur complement data, then the computational tree of the method just described is a chain. Since all the data must be received from the child before work at the parent node can complete, parallelism cannot be exploited at the tree level. These deficiencies can be at least partially overcome through allowing the use of more than one front. This permits pivot orderings that are better at preserving sparsity and also gives more possibility for exploitation of parallelism through working simultaneously on different fronts.

6.4.3 Parallel implementation of frontal schemes

This way of exploiting parallelism when using frontal methods is similar to domain decomposition, where we partition the underlying "domain" into subdomains, perform a frontal decomposition on each subdomain separately (this can be done in parallel) and then factorize the matrix corresponding to the remaining "boundary" or "interface" variables (the Schur complement system), perhaps by also using a frontal scheme, as in [135], or by any suitable solver. This strategy corresponds to a bordered block diagonal ordering of the matrix and can be nested. More recently, a class of algorithms have been designed [21] that combine different ordering strategies on the subdomain and the interface variables.

Although the main motivation for using multiple fronts is usually to exploit parallelism, it is also important that the amount of work can be significantly changed by partitioning the domain, and in some cases can be much reduced. For example, suppose we have a 5-point finite-difference discretization on a $2N$ x $2N$ grid, resulting in a matrix of order $4N^2$ and semibandwidth $2N$. Then straightforward solution using a frontal scheme requires $32N^4 + O(N^3)$ floating-point operations whereas, if the domain is partitioned into four subdomains each of size N x N, the operation count can be reduced to $18.6N^4 + O(N^3)$. Note that the ordering within each subdomain is important in achieving this performance.

We illustrate this change in the number of floating-point operations by running a multiple-front code on our model problem on a 48 x 48 element grid on a single processor of a CRAY Y-MP [135]. With a single domain, the CPU time is 69.4 secs and there are 16970 million floating-point operations in the factorization, while the corresponding figures using four subdomains are 48.4 and 10350. If we partition further, the figures reduce to 40.3 and 8365, when using 8 subdomains.

We have examined the performance of this approach in two parallel environments: on an eight processor shared-memory CRAY Y-MP8I and on a network of five DEC Alpha workstations using PVM [135]. In each case, we divide the original problem into a number of subdomains equal to the number of processors being used. It is difficult to get meaningful times in either environment because we cannot guarantee to have a dedicated machine. The times in Table 6.8 are, however, for lightly loaded machines.

The results on the CRAY are encouraging and show good speedup. Those on the Alpha farm are quite comparable and indicate that the overheads of PVM and communication costs do not dominate and good speedups are possible. We should add that it is important that disk i/o is local to each processor. The default on our Alpha system was to write all files centrally and this increased data traffic considerably, gave poor performance, and varied greatly depending on system loading.

We also observe that the speedup obtained when increasing the number of subdomains from 2 to 4 is greater than 2. This apparent superlinear behavior is caused by the reduction in the number of floating-point operations for the four subdomain partitioning as discussed earlier.

Other work on the parallel implementation of a frontal elimination has been carried out by Benner et al. [34] using large-grain parallelism on a CRAY X-MP and an ELXSI, and by Lucas et al. [229] on a hypercube.

6.5 Multifrontal Methods

If one takes the techniques discussed in the previous section to their logical conclusion, then many separate fronts can be developed simultaneously and can be chosen using a sparsity preserving ordering such as minimum degree. We call such methods "multifrontal" methods. The idea is to

Table 6.8: **Multiprocessor performance of MA42 on CRAY Y-MP and DEC Alpha farm on 48 x 48 grid for our standard test problem**

	CRAY Y-MP		DEC Alpha farm	
Number of subdomains	Wall clock time (secs)	Speedup	Wall clock time (secs)	Speedup
1	98.8	-	1460.3	-
2	64.6	1.5	1043.0	1.4
4	30.7	3.2	457.5	3.2
8	15.3	6.5	-	-

couple a sparsity ordering with the efficiency of a frontal matrix kernel so allowing good exploitation of high performance computers. Multifrontal methods are described in some detail in [120], and their potential for parallelism is discussed in [113, 114, 115].

In this section, we shall work through a small example to give a flavor of the important points and to introduce the notion of an elimination tree. An elimination tree, discussed in detail in [113] and [228], is used to define a precedence order within the factorization. The factorization commences at the leaves of the tree, and data is passed towards the root along the edges in the tree. To complete the work associated with a node, all the data must have been obtained from the children of the node, otherwise work at different nodes is independent. We use the example in Figure 6.9 to illustrate both the multifrontal method and its interpretation in terms of an elimination tree.

$$
\begin{array}{cccc}
\times & & \times & \times \\
 & \times & \times & \times \\
\times & \times & \times & \\
\times & \times & & \times
\end{array}
$$

Figure 6.9: **Matrix used to illustrate multifrontal scheme**

The matrix shown in Figure 6.9 has a nonzero pattern that is symmetric. (Any system can be considered, however, if we are prepared to store explicit zeros.) The matrix is ordered so that pivots will be chosen down the diagonal in order. At the first step, we can perform the elimination corresponding to the pivot in position (1,1), first "assembling" row and column 1 to get the submatrix shown in Figure 6.10. By "assembling", we mean placing the entries of row and column 1 into a submatrix of order the number of entries in row and column 1. Thus the zero

$$\begin{array}{ccc} \times & \times & \times \\ \times & & \\ \times & & \end{array}$$

Figure 6.10: **Assembly of first pivot row and column**

entries a_{12} and a_{21} cause row and column 2 to be omitted in Figure 6.10, and so an index vector is required to identify the rows and columns that are in the submatrix. The index vector for the submatrix in Figure 6.10 would have entries (1,3,4) for both the rows and the columns. Column 1 is then eliminated by using pivot (1,1) to give a reduced matrix of order two with associated row (and column) indices 3 and 4. In conventional Gaussian elimination, updating operations of the form

$$a_{ij} \Leftarrow a_{ij} - a_{i1}[a_{11}]^{-1}a_{1j} \tag{6.10}$$

would be performed immediately for all (i, j) such that $a_{i1}a_{1j} \neq 0$. However, in this formulation, the quantities

$$a_{i1}[a_{11}]^{-1}a_{1j} \tag{6.11}$$

are held in the reduced submatrix, and the corresponding updating operations are not performed immediately. These updates are not necessary until the corresponding entry is needed in a later pivot row or column. The reduced matrix can be stored until that time.

Row (and column) 2 is now assembled, the (2,2) entry is used as pivot to eliminate column 2, and the reduced matrix of order two—with associated row (and column) indices of 3 and 4—is stored. These submatrices are called frontal matrices. More than one frontal matrix generally is stored at any time (currently we have two stored). This is why the method is called "multifrontal". Now, before we can perform the pivot operations using entry (3,3), the updating operations from the first two eliminations (the two stored frontal matrices of order two) must be performed on the original row and column 3. This procedure is effected by summing or assembling the reduced matrices with the original row and column 3, using the index lists to control the summation. Note that this gives rise to an assembled submatrix of order 2 with indices (3,4) for rows and columns. The pivot operation that eliminates column 3 by using pivot (3,3) leaves a reduced matrix of order one with row (and column) index 4. The final step sums this matrix with the (4,4) entry of the original matrix. The sequence of major steps in the elimination is represented by the tree in Figure 6.11.

The same storage and arithmetic are needed if the (4,4) entry is assembled at the same time as the (3,3) entry, and in this case the two pivotal steps can be performed on the same submatrix. This procedure corresponds to collapsing or amalgamating nodes 3 and 4 in the tree of Figure 6.11 to yield the tree of Figure 6.12. On typical problems, node amalgamation produces a tree with about half

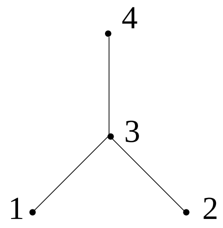

Figure 6.11: **Elimination tree for the matrix of Figure 6.9**

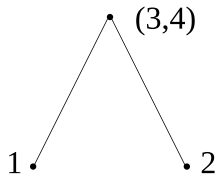

Figure 6.12: **Assembly tree for the matrix of Figure 6.9 after node amalgamation**

as many nodes as the order of the matrix. We call this tree the *assembly* tree. Additional advantage can be taken of amalgamations which do not preserve the number of arithmetic operations, that is where there are variables in the child node that are not present in the parent. This is sometimes called "relaxed amalgamation". Duff and Reid [129] employ node amalgamation to enhance the vectorization of a multifrontal approach, and much subsequent work has also used this strategy for better vectorization and exploitation of parallelism (for example, [19] and [228]).

The computation at a node of the tree is simply the assembly of information concerning the node, together with the assembly of the reduced matrices from its children, followed by some steps of Gaussian elimination. Each node corresponds to the formation of a frontal matrix of the form (6.8) followed by some elimination steps, after which the Schur complement is passed on for assembly at the parent node.

Viewing the factorization using an assembly tree has several benefits. Since only a partial ordering is defined by the tree, the only requirement for a numerical factorization with the same amount of arithmetic is that the calculations must be performed for all the children of a node before those at the parent node can complete. Thus, many different orderings with the same number of floating-point operations can be generated from the elimination tree. In particular, orderings can be chosen for economic use of storage, for efficiency in out-of-core working, or for parallel implementation (see Section 6.5.3). Additionally, small perturbations to the tree and the number of floating-point operations can accommodate asymmetry, numerical pivoting, or enhanced vectorization.

Liu [228] presents a survey of the role of elimination trees in Gaussian elimination and discusses the efficient generation of trees (in time effectively linear in the number of entries in the original matrix) as well as the effect of different orderings of the tree and manipulations of the tree that preserve properties of the elimination. For example, the ordering of the tree can have a significant effect on the storage required by the intermediate frontal matrices generated during the elimination. Permissible manipulations on the tree include tree rotations, by means of which the tree can, for example, be made more suitable for driving a factorization that exploits parallelism better [281].

The multifrontal method can be used on indefinite systems and on matrices that are symmetric in pattern but not in value. The HSL codes MA27 and MA47 are designed for symmetric indefinite systems [127] and provide a stable factorization by using a combination of 1×1 and 2×2 pivots from the diagonal [46]. The HSL code MA41 [9] will solve systems for which the matrix is unsymmetric but it bases its original analysis on the pattern formed from the Boolean summation of the matrix with its transpose. Although this strategy could be poor, it even works for systems whose pattern is quite unsymmetric, as we show in Table 6.15. For matrices which are very structurally unsymmetric (like those in the last two columns of Table 6.15), Davis and Duff [73] have developed an approach that generalizes the concept of elimination trees and performs a Markowitz style analysis on the unsymmetric pattern while retaining the efficiency of a multifrontal method. We will discuss this further in Section 6.5.5.

6.5.1 Performance on Vector Machines

Because multifrontal methods have identical kernels to the frontal methods, one might expect them to perform well on vector machines. However, the dense matrices involved are usually of smaller dimension. There is also a greater amount of data handling outside the kernel, and it is important to consider not only the performance of the kernel but also the assembly operations involved in generating the assembled frontal matrix.

Since a characteristic of frontal schemes is that, even for a matrix of irregular structure, the eliminations are performed using direct addressing and indirect addressing is only performed at the assembly stage, it is possible to affect the balance of these operations by performing more assemblies before performing eliminations. This node amalgamation was first suggested in the original paper of Duff and Reid [128] where it was recommended as an aid to performance on vector machines. They parameterized this amalgamation by coalescing a parent node with its eldest child if the number of eliminations originally scheduled for the parent was less than a specified *amalgamation parameter*. Naturally, as this amalgamation parameter increases, the number of nodes will monotonically decrease although, because there is not a complete overlap of the variables in the parent and child, the operations for factorization and number of entries in the factors will

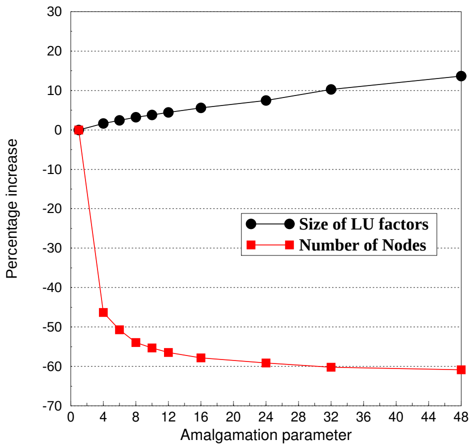

Figure 6.13: **Effect of amalgamation on size of factors and number of nodes**

increase. We show this effect in Figure 6.13 where the runs were performed by Amestoy using the MA41 code on test matrix BCSSTK15.

The intended effect of node amalgamation with respect to increased efficiency on vector machines is to reduce the number of costly indirect addressing operations albeit at the cost of an increase in the number of elimination operations. We show this effect in Figure 6.14 where we note that, as the amalgamation parameter is increased, there is a much greater reduction in indirect operations than the increase in direct operations so that we might expect amalgamation to be very beneficial on machines where indirect operations are more costly than directly addressed operations. This we illustrate in Figures 6.15 and 6.16. The effect is very marked on the CRAY C90 vector computer (Figure 6.15) but is also noticeable on RISC workstations (Figure 6.16) where the use of indirect addressing causes problems for the management of the cache.

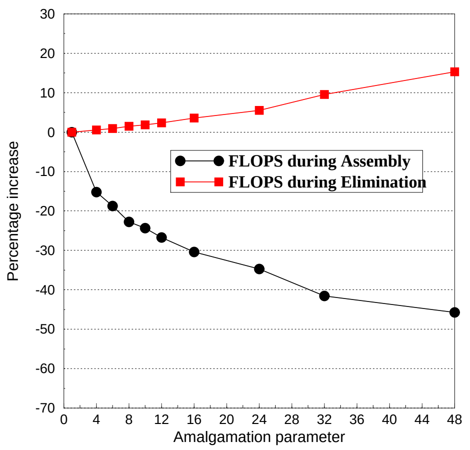

Figure 6.14: **Effect of amalgamation on number of indirect and direct operations**

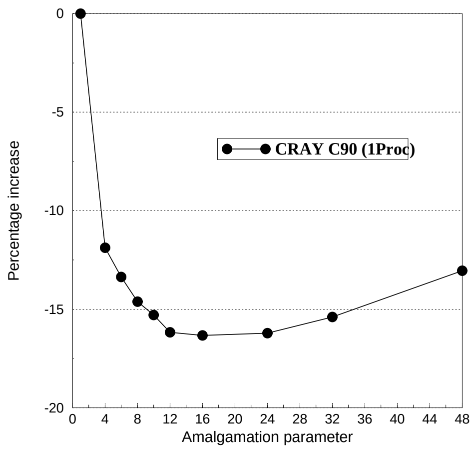

Figure 6.15: **Effect of amalgamation on factorization time on CRAY C90**

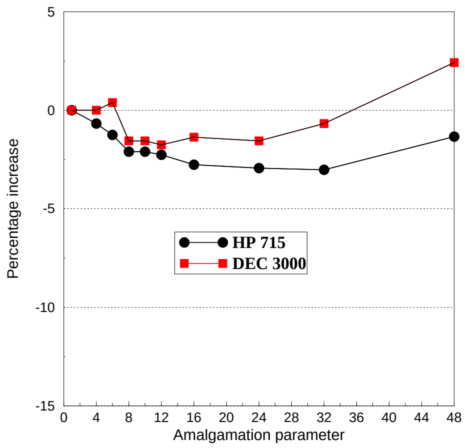

Figure 6.16: **Effect of amalgamation on factorization time on RISC workstations**

Amestoy and Duff [9] have used BLAS kernels in both assembly and elimination operations to achieve over 0.75 Gflop/s on one processor of the CRAY C98 for the numerical factorization of problems from a range of applications. This high computational rate is achieved without incurring a significant increase in arithmetic operations. Ashcraft [19] also reports on high computational rates for the vectorization of a multifrontal code for symmetric systems.

6.5.2 Performance on RISC machines

When we were discussing indirect addressing in Section 6.2, we remarked that its use in the innermost loops can be inefficient on machines with cache memory. Conversely, dense kernels allow good use of memory hierarchies of the kind found in the modern generation of RISC architecture based machines. We illustrate this with the results shown in Table 6.9.

Table 6.9: **Performance in Mflop/s of multifrontal code MA41 on matrix BCSSTK15 on range of RISC processors**

Computer	Peak	DGEMM	MA41
DEC 3000/400 AXP	133	49	34
HP 715/64	128	55	30
IBM RS6000/750	125	101	64
IBM SP2 (Thin node)	266	213	122
MEIKO CS2-HA	100	43	31

In this table, we show the performance of a tuned version of DGEMM on the machine. This is the Level 3 BLAS routine for (dense) matrix-by-matrix multiplication and is usually the fastest non-trivial kernel for any machine. We feel this is a more meaningful yardstick by which to judge performance than the peak speed, although we show this also in Table 6.9. The MA41 performance ranges from 50 to 70% of the DGEMM performance showing that good use is being made of the Level 3 BLAS.

6.5.3 Performance on Parallel Machines

Duff [128] considered the implementation of multifrontal schemes on parallel computers with shared memory. In this implementation, there are really only three types of tasks, the assembly of information from the children, the selection of pivots, and the subsequent eliminations, although he also allows the eliminations to be blocked so that more than one processor can work on the

Table 6.10: **Speedup on Alliant FX/8 of Five-Point Discretization of Laplace on a** 30×30 **grid**

No. processors	Time	Speedup
1	2.59	-
2	1.36	1.9
4	.74	3.5
6	.57	4.5
8	.46	5.6

elimination operations from a single pivot. He chooses to store all the tasks available for execution in a single queue with a label to identify the work corresponding to the task. When a processor is free, it goes to the head of the queue, selects the task there, interprets the label, and performs the appropriate operations. This process may in turn generate other tasks to be added to the end of the queue. This model was used by Duff in designing a prototype parallel code from the HSL code MA37 and we show, in Table 6.10, speedup figures that he obtained on the Alliant FX/8 [116]. It was this prototype code that was later developed into the code MA41.

The crucial aspect of multifrontal methods that enables exploitation of parallelism is that work at different nodes of the assembly tree is independent and the only synchronization required is that data from the child nodes must be available before the computation at a node can be completed. The computation at any leaf node can proceed immediately and simultaneously. Although this provides much parallelism near the leaf nodes of the tree, there is less towards the root and, of course, for an irreducible problem there is only one root node. If the nodes of the elimination tree are regarded as atomic, then the level of parallelism reduces to one at the root and usually increases only slowly as we progress away from the root. If, however, we recognize that parallelism can be exploited within the calculations at each node (corresponding to one or a few steps of Gaussian elimination on a dense submatrix), much greater parallelism can be achieved. This loss of parallelism by regarding nodes as atomic is compounded by the fact that most of the floating-point arithmetic is performed near the root so that it is vital to exploit parallelism also within the computation at each node. We illustrate the amount of parallelism available from the assembly tree by the results in Table 6.11, where we show the size and number of tree nodes at the leaves and near the root of the tree. Thus we see that while there is much parallelism at the beginning of the factorization, there is much less near the root, if the tree nodes are regarded as atomic. Furthermore, about 75% of the work in the factorization is performed within these top three levels. However, the size of the frontal matrices are much larger near the root so we can exploit parallelism at this stage of the factorization by, for example, using parallel variants of the Level 3 BLAS for the elimination operations within a node. The effect of this is seen clearly in the results from [8] shown in Table 6.12, where the increased speedups in columns (2) are due to exploiting parallelism within the node.

Table 6.11: **Statistics on front sizes in tree**

			Leaf nodes		Top 3 levels	
Matrix	Order	Tree nodes	Number	Av. size	Number	Av. size
BCSSTK15	3948	576	317	13	10	376
BCSSTK33	8738	545	198	5	10	711
BBMAT	38744	5716	3621	23	10	1463
GRE1107	1107	344	250	7	12	129
SAYLR4	3564	1341	1010	5	12	123
GEMAT11	4929	1300	973	10	112	148

Table 6.12: **Performance summary of the multifrontal factorization using MA41 on 7 processors of a CRAY C98.** In columns (1) we exploit only parallelism from the tree; in columns (2) we combine the two levels of parallelism

			(1)		(2)	
Matrix	order	entries	Mflop/s	(speedup)	Mflop/s	(speedup)
WANG3	26064	177168	1062	(1.42)	3718	(4.98)
WANG4	26068	177196	1262	(1.70)	3994	(5.39)
BBMAT	38744	1771722	2182	(3.15)	3777	(5.46)

The code used in the experiments in Table 6.12 was the MA41 code [10, 11] which was developed for shared memory parallel computers. It should be emphasised that, because of the portability provided through the use of the BLAS, the MA41 code is essentially identical for all shared memory machines. MA41 will also run with relatively minor changes on a virtual shared memory machine although it is beneficial to make more significant changes for efficiency. This is seen in Figure 6.17, where a significant modification to the virtual shared memory implementation was to perform some explicit copying of subarrays into the private memories and to use BLAS locally on each processor [8].

Multifrontal codes have also been developed for distributed memory machines. An efficient

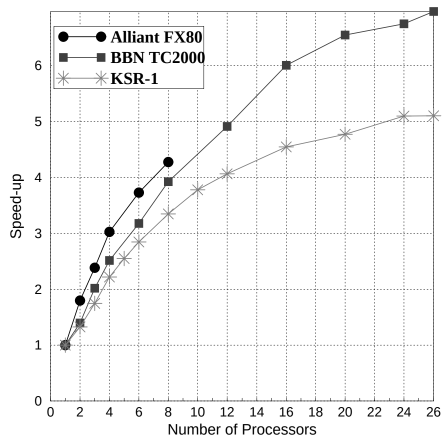

Figure 6.17: **Speedup graph**

data parallel version of a multifrontal code has been implemented by [65]. The most impressive times and speedups of which we are aware have been obtained by Gupta and Kumar [184] for an implementation of a multifrontal code on a 1024-processor nCUBE-2 machine. We include some of their results for the matrix BCSSTK15 in Table 6.13. For larger matrices, they show even better performance with a speedup of over 350 on 1024 processors for matrix BCSSTK31 of order 35588 with about 6.4 million entries. In subsequent experiments, Gupta, Karypis, and Kumar [185] have implemented versions of their algorithms on a CRAY T3D, using SHMEM for message passing, and have obtained a high performance and good speedup on this machine also.

Table 6.13: **Performance of code of Gupta and Kumar on nCUBE on matrix BCSSTK33.** Times in seconds. Results taken from Gupta and Kumar [184]

Number of processors	1	4	16	64	256
Time	103.7	26.7	8.3	3.2	1.5
Speedup	1.00	3.9	12.5	32.4	67.8
Efficiency %	100	97	78	51	27
Load balance %	100	98	91	87	84

Clearly the parallelism available through the elimination tree depends on the ordering of the matrix. In general, short bushy trees are preferable to tall thin ones since the number of levels determines the inherent sequentiality of the computation. Two common ordering strategies for general sparse symmetric systems are minimum degree and nested dissection (see, for example, [120] and [170]). Although these orderings are similar in behavior for the amount of arithmetic and storage, they give fairly different levels of parallelism when used to construct an elimination tree. We illustrate this point in Table 6.14, where the maximum speedup is computed from a simulation of the elimination as the ratio of the number of operations in the sequential algorithm to the number of sequential operations in the parallel version, with account taken of data movement as well as floating-point calculations [125].

Table 6.14: **Comparison of two orderings for generating an elimination tree for multifrontal solution** (the problem is generated by a 5-point discretization of a 10×100 grid)

Ordering	Minimum degree	Nested dissection
Number of levels in tree	52	15
Number of pivots on longest path	232	61
Maximum speedup	9	47

6.5.4 Exploitation of structure

The multifrontal methods that we have just described generate the assembly tree and corresponding guiding information for the numerical factorization without access to numerical values and assume that the pivot order thus selected will be numerically suitable. For matrices which are not positive definite, this is not necessarily the case, so the numerical factorization must be able to tolerate enforced changes to the predicted sequence by performing additional pivoting. For general symmetric systems, this is usually not much of a problem and the overheads of the additional pivoting are fairly low. However, symmetric systems of the form

$$\begin{pmatrix} H & A \\ A^T & 0 \end{pmatrix}, \tag{6.12}$$

called augmented systems, occur frequently in many application areas [117]. It is necessary to exploit the structure during the symbolic analysis so that pivots are not chosen from the zero block and it is preserved during the factorization. The effect of taking this structure into account can be very dramatic. For example, on a matrix of the form (6.12) with H a diagonal matrix of order 1028 with 504 ones and 524 zeros and A the 1028 x 524 matrix FFFFF800 from the collection in [165], an earlier multifrontal code which does not exploit the zero-block structure (namely HSL code MA27) forecasts 1.5 million floating-point operations but requires 16.5 million. The HSL code MA47 [130, 132], however, forecasts and requires only 7,954 flops. Unfortunately, the new code is much more complicated and involves more data movement since frontal matrices are not necessarily absorbed by the parent and can percolate up the tree. Additionally, the penalty for straying from the forecast pivot sequence can be very severe.

6.5.5 Unsymmetric multifrontal methods

Although the multifrontal schemes that we have just described are designed for structurally symmetric matrices, structurally unsymmetric matrices can be handled by explicitly holding zero entries. It is more complicated to design an efficient multifrontal scheme for matrices that are asymmetric in structure. The main difference is that the elimination cannot be represented by an (undirected) tree but a directed acyclic graph (DAG) is required [146]. The frontal matrices are, of course, no longer square and, as in the case discussed in the previous section, they are not necessarily absorbed at the parent node and can persist in the DAG. Finally the complication of *a posteriori* numerical pivoting is even more of a problem with this scheme so that the approach adopted is normally to take account of the real values when computing the DAG and the pivot order.

An unsymmetric multifrontal code by Davis and Duff, based on this approach, is included as subroutine MA38 in the Harwell Subroutine Library. We show a comparison, taken from [73], of this code with the "symmetric" HSL code MA41 in Table 6.15.

The problems in Table 6.15 are arranged in order of increasing asymmetry, where the asymmetry index is defined as

$$\frac{\text{Number of pairs such that } a_{ij} = 0, a_{ji} \neq 0}{\text{Total number of off-diagonal entries}}, \tag{6.13}$$

so that a symmetric matrix has asymmetry index 0.0. These results show clearly the benefits of treating the asymmetry explicitly.

We show a comparison of the HSL MA38 code with HSL codes MA41 and MA48 in Table 6.16, taken from [72]. These results show that the new code can be very competitive, sometimes outperforming the other codes.

Table 6.15: **Comparison of "symmetric" (MA41) and "unsymmetric" (MA38) code**

Order	13535	62424	26064	22560	120750	36057
Entries	390607	1717792	177168	1014951	1224224	227628
Index of asymmetry (6.13)	0.00	0.00	0.00	0.36	0.76	0.89
Floating-point ops (10^9)						
MA41	0.3	2.3	10.4	2.9	38.2	0.6
MA38	3.8	5.3	62.2	9.0	7.0	0.2
Factorization time						
(*seconds on Sun ULTRA*)						
MA41	8	46	174	81	809	19
MA38	85	127	1255	226	220	11

Table 6.16: **Comparison of MA38 with MA48 and MA41 on a few test matrices.** Times in seconds on a Sun ULTRA-1 workstation

Matrix	Order	No entries	Factorization Time		
			MA38	MA48	MA41
AV41092	41092	1683902	1618	1296	254
PSMIGR_1	3140	543162	198	179	188
ONETONE1	36057	341088	57	110	189
LHR71	70304	1528092	93	287	996

6.6 Other Approaches for Exploitation of Parallelism

Although we feel that the multifrontal approach is very suitable for exploitation of parallelism, it is certainly not the only approach being pursued. Indeed, the Cholesky algorithm viewed as a left-looking algorithm (Section 5.4.2) can be implemented for sparse systems and can also be blocked by using a supernodal formulation similar to the node amalgamation that we discussed in Section 6.5. A code based on this approach attained very high performance on some structural analysis and artificially generated problems on a CRAY Y-MP [281]. A variant of the standard column-oriented sparse Cholesky algorithm has also been implemented on hypercubes ([168] and [169]). Highly efficient codes based on a supernodal factorization for MIMD machines, in particular for an INTEL Paragon, have been developed by [266].

The supernodal concept has recently been extended to unsymmetric systems by [83]. It is now not possible to use Level 3 BLAS efficiently. However, Demmel et al [83] have developed an implementation that performs a dense matrix multiplication of a block of vectors and, although these cannot be written as another dense matrix, they show that this Level 2.5 BLAS has most of the performance characteristics of Level 3 BLAS since the repeated use of the same dense matrix allows good use of cache and memory hierarchy. In Table 6.17 we compare their code, SUPERLU, with the multifrontal approach on a range of examples. The multifrontal code MA41 was used if the asymmetry index of the matrix was less than 0.5 and the code MA38 was used otherwise. This seems to support the premise that the simpler multifrontal organization performs better although it is important to use the *very* unsymmetric code, MA38, when the matrix is very structurally asymmetric.

Table 6.17: **Comparison of multifrontal against supernodal approach.** Times in seconds on a Sun ULTRA-1 workstation

Matrix	Order	Entries	Analyse and Factorize Time (secs)		Entries in LU factors (10^6)	
			SUPERLU	Multif	SUPERLU	Multif
ONETONE2	36057	227628	9	11	1.3	1.3
TWOTONE	120750	1224224	758	221	24.7	9.8
WANG3	26024	177168	1512	174	27.0	11.4
VENKAT50	62424	1717792	172	46	18.0	11.9
RIM	22560	1014951	78	80	9.7	7.4
GARON2	13535	390607	60	8	5.1	2.4

A quite different approach to designing a parallel code is more related to the general approach discussed in Section 6.2. In this technique, when a pivot is chosen all rows with entries in the pivot column and all columns with entries in the pivot row are marked as ineligible and a subsequent pivot can only be chosen from the eligible rows and columns. In this way, a set of say k independent pivots is chosen. This set of pivots not affect each other and can be used in parallel. In other words, if the pivots were permuted to the beginning of the matrix, this pivot block would be diagonal. The resulting elimination operations are performed in parallel using a rank k update. This is similar to the scheme of Davis and Yew discussed in Section 6.2.4. We show some results of this approach taken from [303] in Table 6.18, where it is clear that good speedups can be obtained.

Table 6.18: **Results from [303] on a PARSYTEC SuperCluster FT-400.** Times in seconds for LU factorization.

Matrix	Order	Entries	Number of processors			
			1	16	100	400
SHERMAN 2	1080	23094	1592	108	32.7	15.6
LNS 3937	3937	25407	2111	168	37.9	23.7

6.7 Software

Although much software is available that implements direct methods for solving sparse linear systems, little is within the public domain. There are several reasons for this situation, the principal ones being that sparse software often is encapsulated within much larger packages (for example, for structural analysis) and that much work on developing sparse codes is funded commercially so that the fruits of this labor often require licenses.

Among the public domain sparse software are some routines from the Collected Algorithms of ACM (available from `netlib`), mostly for matrix manipulation (for example, bandwidth reduction, ordering to block triangular form) rather than for equation solution, although the NSPIV code from Sherman [280] is available as Algorithm 533.

Both Y12M and the HSL code MA28, referenced in this chapter, are available from `netlib`, although people obtaining MA28 in this way are still required to sign a license agreement, and use of the newer HSL code MA48 is recommended. A freely available research version of MA38, called UMFPACK, which includes a version for complex matrices, is available in `netlib` as is the C code SUPERLU implementing the supernodal factorization of [83]. There is also a skeleton sparse LU code from Banks and Smith in the *misc* collection in `netlib`, and Joseph Liu distributes his multiple minimum-degree code upon request.

Among the codes available under license are those from the Harwell Subroutine Library that were used to illustrate many of the points in this chapter, a subset of which is also marketed by NAG under the title of the Harwell Sparse Matrix Library. Contact details for these organizations can be found in Appendix A.2. IMSL is also developing its own code for solving sparse systems, and a sparse LU code is available in the current release of ESSL for the IBM RS/6000 and SP2 computers. Sparse linear equation codes are also available to users of Cray computers upon request to Cray Research Inc.

Other packages include the SPARSPAK package, primarily developed at the University of Waterloo [172], which solves both linear systems and least-squares problems, and routines in the

PORT library from Bell Labs [216], details of which can be obtained from `netlib`. Versions of the package YSMP, developed at Yale University [145], can be obtained from Scientific Computing Associates at Yale, who also have several routines implementing iterative methods for sparse equations.

Although work is in progress to produce a public domain version of SUPERLU for shared memory parallel computers, there is even less software available for the parallel solution of sparse equations. The only public domain software we are aware of is the code by Heath and Raghavan which is included in the SCALAPACK package. Bisseling et al [222] plan a public release of their parallel Markowitz/threshold solver, SPLU, designed for message passing architectures, while the EU LTR Project PARASOL (http://192.129.37.12/parasol/) plans to develop a suite of direct and iterative sparse solvers for message passing architectures that are primarily targeted at finite-element applications.

It should be stressed that we have been referring to fully supported products. Many other codes are available that are either at the development stage or are research tools (for example, the SMMS package of Fernando Alvarado at Wisconsin [3]) and the SPARSKIT package of Youcef Saad at Minneapolis [273].

6.8 Brief Summary

We have discussed several approaches to the solution of sparse systems of equations, with particular reference to their suitability for the exploitation of vector and parallel architectures. We see considerable promise in both frontal and multifrontal methods on vector machines and reasonable possibilities for exploitation of parallelism by supernodal and multifrontal methods. A principal factor in attaining high performance is the use of dense matrix computational kernels, which have proved extremely effective in the dense case.

Finally, we have tried to keep the narrative flowing in this presentation by avoiding an excessive number of references. For such information, we recommend the recent review [118], where 215 references are listed.

Chapter 7

Krylov Subspaces - Projection

7.0.1 Notations

We will formulate the iterative methods in such a way that they can be used for real-valued as well as complex-valued variables. In particular, the innerproduct of two n-vectors x and y will be denoted as

$$x^* y = \sum_{i=1}^{i=n} \overline{x}_i y_i .$$

The norm $\|x\|_2$ of a vector x is defined by

$$\|x\|_2 \equiv \sqrt{x^* x}.$$

For an m by n matrix A with elements $a_{i,j}$, the matrix A^* denotes the n by m matrix with elements $\overline{a}_{j,i}$ (the complex conjugate of A). For real-valued matrices A^* will sometimes be denoted as A^T (the transpose of A).
If one works with complex variables then, of course, all parameters in the iterative schemes (except for the norms and the integer values) have to be specified as complex values.
Vectors will be denoted by lower case symbols, like x, y. Approximations for vectors in an iterative process will be denoted by upper indices, like $x^{(i)}$. In particular, the initial guess for the solution x of $Ax = b$ will be denoted as $x^{(0)}$, with corresponding residual $r^{(0)} \equiv b - Ax^{(0)}$.
Vectors that belong to a set, for instance a basis for some subspace, are denoted by a simple superscript, like v^1, v^2, ...

7.0.2 Basic iteration methods: Richardson iteration, Power method

A very basic idea, that leads to many effective iterative solvers, is to split the matrix A of a given linear system as the sum of two matrices, one of which a matrix that would have led to a system that can easily be solved. For the linear system $Ax = b$, the splitting $A = I - (I - A)$ leads to the well-known *Richardson iteration*:

$$x^{(i)} = b + (I - A)x^{(i-1)} = x^{(i-1)} + r^{(i-1)}, \quad r^{(i-1)} \equiv b - Ax^{(i-1)} \tag{7.1}$$

Multiplication by $-A$ and adding b gives

$$b - Ax^{(i)} = b - Ax^{(i-1)} - Ar^{(i-1)}$$

or

$$r^{(i)} = (I - A)r^{(i-1)} = (I - A)^i r^{(0)} = P_i(A)r^{(0)}. \tag{7.2}$$

This expression shows us that we may expect (fast) convergence if $||I - A||_2$ is (much) smaller than 1.
We have expressed the residual as a polynomial in A times the initial residual and for the standard Richardson iteration this polynomial is a special one, namely $P_i(A) = (I - A)^i$. As we will see, many popular iteration methods have the property that the residual r_i at the i-th iteration step can be expressed as some polynomial, of degree i in A, times the initial residual $r^{(0)}$. This property will be exploited in several ways, for instance, for getting bounds on the speed of convergence.

We have shown the Richardson iteration in its simplest form. A more general form of the iteration is

$$x^{(i)} = x^{(i-1)} + \alpha_i r^{(i-1)}, \tag{7.3}$$

where α_i might be chosen, for example, to minimize $||r^{(i)}||$. This leads to

$$r^{(i)} = b - Ax^{(i)} = b - Ax^{(i-1)} - \alpha_i Ar^{(i-1)} = (I - \alpha_i A)r^{(i-1)}.$$

We easily see that

$$r^{(i)} = P_i(A)r(0), \quad \text{with} \quad P_i(A) = \prod_{j=1}^{i}(I - \alpha_j A).$$

The Richardson iteration with respect to a specific choice of the parameters $\{\alpha_j\}$ leads to the polynomial $P_i(t) = \prod_{j=1}^{i}(1 - \alpha_j t)$. As we shall see, most other methods amount to selecting these parameters and hence selecting a specific family of polynomials. These polynomials are almost never constructed in explicit form, but they are used in formulas to express the special relation between r_i and r_0 and specific properties of the polynomials are often used to analyze convergence.

Modern methods (such as GMRES) often select these parameters indirectly through a minimization or projection condition.

For the sake of simplicity, we shall continue to discuss the choice $\alpha_j = 1$.
In the derivation of an iterative scheme we have used a very simple splitting. In many cases approximations K, better than I, to the given matrix A are available. For instance, if one skips in an LU-decomposition certain small elements, in an attempt to make the decomposition computationally cheaper, then the resulting decomposition may not be good enough to represent the solution sufficiently well, but it may be used to improve the solution as in iterative refinement. It is simple to formulate this as an iterative process similar to the Richardson iteration. We write $A = K - (K - A)$, and the corresponding iteration method becomes

$$Kx^{(i)} = b + (K - A)x^{(i-1)} = Kx^{(i-1)} + b - Ax^{(i-1)}. \tag{7.4}$$

Solving for $x^{(i)}$ leads formally to

$$x^{(i)} = x^{(i-1)} + K^{-1}(b - Ax^{(i-1)}).$$

In practice we almost never form K^{-1} explicitly, but the above expression helps to make the relation with the standard Richardson iteration more clear. If we define $B \equiv K^{-1}A$ and $c = K^{-1}b$, then the K-splitting is equivalent with the standard splitting for $Bx = c$. In fact, working with the iteration method for K, is formally equivalent with standard Richardson for the *preconditioned* matrix $K^{-1}A$ and preconditioned right-hand side. The matrix K^{-1} may be viewed as an operator that helps to transform the given matrix to a matrix that is closer to I, and we have seen allready that this is a highly desirable property for the standard iteration process.
Hence, for our further considerations with respect to deriving and analysing iteration schemes, it is no loss of generality if we restrict ourselves to the standard splitting. Any other splitting can be incorporated easily in the different schemes by replacing the quantities A and b, by the preconditioned quantities B and c, respectively. As we will see, in actual implementations we always work with expressions as in (7.4), and solve $x^{(i)}$ from such expressions.

From now on we will assume that $x^{(0)} = 0$. This also does not mean a loss of generality, for the situation $x^{(0)} \neq 0$ can, through a simple linear translation $z = x - x^{(0)}$, be transformed to the system

$$Az = b - Ax^{(0)} = \tilde{b},$$

for which obviously $z^{(0)} = 0$.

For the Richardson iteration it follows that

$$x^{(i+1)} = r^{(0)} + r^{(1)} + r^{(2)} + \cdots + r^{(i)} = \sum_{j=0}^{i} (I - A)^j r^{(0)},$$

$$x^{(i+1)} \in \{r^{(0)}, Ar^{(0)}, \ldots, A^i r^{(0)}\} \equiv \mathcal{K}_{i+1}(A; r^{(0)}).$$

The subspace $\mathcal{K}_i(A; r^{(0)})$ is called the *Krylov subspace* of dimension i, generated by A *and* $r^{(0)}$. Apparently, the Richardson iteration delivers elements of Krylov subspaces of increasing dimension. Including local iteration parameters in the iteration leads to other elements of the same Krylov subspaces, and hence to other polynomial expressions for the error and the residual. A natural question to ask now is "what is the best approximate solution that we can select from the Krylov subspace?" Posed mathematically, "best" might be defined as to minimize the norm of the residual, or as to have the residual orthogonal to the Krylov subspace. We will see different interpretations of "best" in Section 8.0.4.

Although it may not be immediately apparant, Krylov subspaces play also a big role in eigenvalue calculations. To iteratively solve an eigenvalue problem $Aw = \lambda w$ (with $\|w\| = 1$) we might observe that

$$A^i w^{(0)} = \sum_{j=1}^{n} \gamma_j \lambda_j^i q^{(j)},$$

where $Aq^{(j)} = \lambda_j q^{(j)}$ and $w^{(0)} = \sum_{j=1}^{n} \gamma_j q^{(j)}$. Thus if $|\lambda_1| > |\lambda_j|$ for $j > 1$ we have

$$\frac{1}{\lambda_1^i} A^i w^{(0)} = q^{(1)} + \sum_{j=2}^{n} \gamma_j q^{(j)} (\frac{\lambda_j}{\lambda_1})^i \rightarrow q^{(1)} \quad \text{as} \quad i \rightarrow \infty.$$

Of course we don't have λ_1 available but virtually any homogeneous scaling of $A^i w^{(0)}$ that is proportional to λ_1^i will do. It is convenient to scale by the largest element at each iteration and the resulting form of the *classic power method* is given in Fig. 7.1. If the largest (in modulus) eigenvalue of A, and its eigenvector, are real then the algorithm can be carried out in real arithmetic and the pair $(\theta^{(j)}, w^{(j)})$ converges to the dominant eigenpair.

There are two major drawbacks to the power method. First, the rate of convergence is linear with a convergence factor proportional to $|\frac{\lambda_2}{\lambda_1}|$ so it may be arbitrarily slow. Second, only one eigenvector can be found. It is possible to use certain deflation schemes [270] to find a subsequent eigenvector once the first one has converged. However, it is not difficult to see that various linear combinations of iterates $\{\widehat{w}^{(i)}\}$ could be used to approximate additional eigenvectors using the same sequence $\{w^{(i)}\}$. For example, $\widehat{w}^{(i)} = (w^{(i)} - \lambda_1 w^{(i-1)})/\lambda_2$ will converge to a multiple of $q^{(2)}$. Once again we see that our approximate solutions are naturally members of a Krylov Subspace $\mathcal{K}_m(A; w^{(0)})$ and we might again ask, "What is the best approximate eigenvector that can be constructed from this subspace?"
Note that the power method exploits only the last two vectors of the Krylov subspace. Later on we will see more powerful methods that exploit the whole Krylov subspace.

$v = w^{(0)} / \max\_elt(w^{(0)})$
for $j = 1, 2, 3, ..,$**until** convergence
  $w^{(j)} = Av$
  $\theta^{(j)} = \max\_elt(w^{(j)})$
  $v = w^{(j)}/\theta^{(j)}$
end

Figure 7.1: The Power Method

7.0.3 Orthogonal basis (Arnoldi, Lanczos)

The standard iteration method (7.1), as well as the Power Method (Fig. 7.1), generate approximate solutions that belong to Krylov subspaces of increasing dimension. Modern iteration methods construct better approximations in these subspaces with some modest computational overhead as compared with the standard iterations.

In order to identify better approximations in the Krylov subspace we need a suitable basis for this subspace, one that can be extended in a meaningful way for subspaces of increasing dimension. The obvious basis $r^{(0)}$, $Ar^{(0)}$, ..., $A^{i-1}r^{(0)}$, for $\mathcal{K}^i(A; r^{(0)})$, is not very attractive from a numerical point of view, since the vectors $A^j r^{(0)}$ point more and more in the direction of the dominant eigenvector for increasing j (the power method !), and hence the basis vectors become dependent in finite precision arithmetic.

Instead of the standard basis one usually prefers an orthonormal basis, and Arnoldi [16] suggested to compute this basis as follows. Assume that we have already an orthonormal basis v^1, ..., v^j for $K^j(A; r^{(0)})$, then this basis is expanded by computing $t = Av^j$, and by orthonormalizing this vector t with respect to v^1, ..., v^j. We start this process with $v^1 \equiv r^{(0)}/\|r^{(0)}\|_2$. In principle the orthonormalization process can be carried out in different ways, but the most commonly used approach is to do this by a modified Gram-Schmidt procedure [178]. This leads to the algorithm in Fig. 7.2, for the creation of an orthonormal basis for $K^m(A; r^{(0)})$.

It is easily verified that the v^1, ..., v^m form an orthonormal basis for $\mathcal{K}^m(A; r^{(0)})$ (that is, if the construction does not terminate at a vector $t = 0$). The orthogonalization leads to relations between the v^j, that can be formulated in a compact algebraic form. Let V_j denote the matrix

$v^1 = r^{(0)}/\|r^{(0)}\|_2$;
for $j = 1, ..., m-1$
  $t = Av^j$;
  for $i = 1, ..., j$
    $h_{i,j} = (v^i)^* t$;
    $t = t - h_{i,j} v^i$;
  end;
  $h_{j+1,j} = \|t\|_2$;
  $v^{j+1} = t/h_{j+1,j}$;
end

Figure 7.2: Arnoldi's method with modified Gram–Schmidt orthogonalization

with columns v^1 up to v^j, then it follows that

$$AV_{m-1} = V_m H_{m,m-1}. \tag{7.5}$$

The m by $m-1$ matrix $H_{m,m-1}$ is upper Hessenberg, and its elements $h_{i,j}$ are defined by the Arnoldi algorithm.
We see that this orthogonalization becomes increasingly expensive for increasing dimension of the subspace, since the computation of each $h_{i,j}$ requires an inner product and a vector update.

Parallel computing aspects. The orthogonalization procedure is composed with three basic elements: a matrix vector product with A, innerproducts, and updates. On vector computers or shared memory parallel computers these kernels can often be efficiently implemented. However, from a computational performance point of view the basis vectors generated by the standard Richardson method or the Power method are attractive, since orthogonalization of a set of vectors offers more possibilities for scheduling and combination of operations than generating the basis vectors and orthonormalizing them one by one. Since, as we have seen, the standard basis tends to be increasingly ill-conditioned, an alternative would be to generate a better conditioned basis first, and to orthogonalize the complete set of vectors afterwards. In terms of computer arithmetic this is, of course, more expensive, but since this approach offers more parallelism, there may be a trade-off on some computers.

We discuss two alternatives for the construction of a reasonably well-conditioned set of basis vectors.

It was suggested in [30] to construct recursions with polynomials with known zeros d_i. The basis is then generated by

$$y^{i+1} = y^i - \frac{1}{d_i} A y^i, \quad i = 2, \ldots, m-1, \tag{7.6}$$

starting with $y^1 = v^1/||v^1||_2$.
In [30] it is suggested to use a Leja ordering for the d_i in order to improve numerical stability of the recursion in (7.6). The zeros d_i should be distributed over the field of values of A, which can be roughly estimated from the spectrum of $H_{m,m}$. The problem here is that we need to have some idea of the spectrum, but in many circumstances one only generates basis sets of limited dimension in order to restrict the computational costs. After the limit dimension has been reached, the iteration, that is the construction of a new Krylov subspace with A, is restarted. So the eigenvalue knowledge obtained from previous Krylov subspaces can be exploited to construct a rather well conditioned basis for the next one.

The second alternative is to use for the basis of the new Krylov subspace the (implicitly generated) orthogonal polynomials of the previous one. This is surprisingly simple: we carry out the same algorithm in Fig. 7.2 as before, acting on the new starting vector v^1, but we skip the computations of the elements $h_{i,j}$ and $h_{j+1,j}$ (we simply use the old values). This procedure has been suggested in [285]:section 6. The two alternatives are closely related. If we choose the d_i's to be the Ritz values of $H_{m-1,m-1}$, then the vectors y^i are up to a scaling factor equal to the vectors generated in the second alternative.
The second alternative is conceptually simple: no problems with ordering of Ritz values, but it is also more expensive. In the approach in [30], the basis is generated with $m-2$ vector updates, whereas in the second alternative this requires $\frac{1}{2}m(m-1)$ updates. Note that these costs for generating a non-orthogonal but well-conditioned basis come in addition to the costs of generating an orthogonal basis for the Krylov subspace. The only advantage of generating an nonorthogonal basis first, is in reducing the costs for communication overhead of the innerproducts. Our advice is to follow the approach in [30], with the Ritz values for the d_i's.

Once we have a set of basis vectors for the Krylov subspace, we can orthogonalize them (if necessary) in the following way [80]. Suppose the basis set is given by

$$\left\{v^1, y^2, \ldots, y^m\right\}$$

The subsequent orthogonalization is done by the algorithm in Fig. 7.3.

This algorithm is a version of modified Gram-Schmidt in which the loops have been interchanged. It is obvious that all innerproducts in the innerloop can be executed in parallel, so that all message

```
for i = 2, ..., m
    for j = i, ..., m
        r_{i-1,j} = (v^{i-1})^* y^j
        y^j = y^j - r_{i-1,j} v^{i-1}
    end
    r_{i,i} = ||y^i||_2
    v^i = y^i / r_{i,i}
end
```

Figure 7.3: Orthogonalization of a given set

passing for these operations can be combined. In particular, this leads to less start-up time, which is often the dominating part in communication. See [80] for an analysis of the parallel performance of this approach and for illustrative experiments.

Note that we need to construct the reduced matrix $H_{m,m-1}$ associated with the orthogonalized set $v^1, \ldots, v^m$. We will now assume that this set has been generated by (7.6). For the set $y^1 = v^1$, y^2, $\ldots$, y^m, we have by construction that

$$[y^2, y^3, \ldots, y^m] = Y_{m-1} - AY_{m-1}D_{m-1}^{-1},$$

where D_{m-1} is the diagonal matrix with diagonal entries $d_1, \ldots, d_{m-1}$, and $Y_j = [y^1, \ldots, y^j]$.

It is easily verified that the relation between the y^j's and the v^j's, after the algorithm in Fig. 7.3, is given by

$$Y_m = V_m R_{m,m},$$

where the elements $r_{i,j}$ of the upper-triangular matrix $R_{m,m}$ are defined in the algorithm.

This leads to

$$\begin{aligned} AY_{m-1}D_{m-1} &= Y_{m-1} - [y^2, \ldots, y^m] \\ &\equiv V_m \widehat{H}_{m,m-1}, \end{aligned}$$

where the elements of the m by $m-1$ upper Hessenberg matrix $\widehat{H}_{m,m-1}$ are given by

$$\widehat{h}_{i,j} = r_{i,j} - r_{i,j+1}.$$

This leads to

$$AV_{m-1}R_{m-1,m-1}D_{m-1}^{-1} = V_m \widehat{H}_{m,m-1}. \tag{7.7}$$

Note that every $x \in K^{m-1}(A; v_1)$ can be expressed as

$$x = V_{m-1} R_{m-1,m-1} D_{m-1}^{-1} y,$$

where y is a vector of length $m - 1$.
This leads to obvious adaptations in the solution methods that are based on the Arnoldi algorithm (like, for instance, GMRES).

Note that if A is symmetric, then so is $H_{m-1,m-1} = V_{m-1}^* A V_{m-1}$, so that in this situation $H_{m-1,m-1}$ is tridiagonal. This means that in the orthogonalization process, each new vector has to be orthogonalized with respect to the previous two vectors only, since all other innerproducts vanish. The resulting three term recurrence relation for the basis vectors of $K_m(A; r^{(0)})$ is known as the *Lanczos method* and some very elegant methods are derived from it. In the symmetric case the orthogonalization process involves constant arithmetical costs per iteration step: one matrix vector product, two innerproducts, and two vector updates. This leaves little freedom for parallelism. In our discussion of the Conjugate Gradients method, which is based on this short-term recurrence, we will show some opportunities for reduction of the communication overhead of the innerproducts on distributed parallel computers.

Chapter 8

Iterative Methods for Linear Systems

8.0.4 Krylov Subspaces Solution methods: basic principles

Since we are looking for a solution $x^{(k)}$ in $\mathcal{K}^k(A; r^{(0)})$, that vector can be written as a combination of the basis vectors of the Krylov subspace, and hence

$$x^{(k)} = V_k y. \tag{8.1}$$

Note that y has k components.
There are four projection-type of approaches to find a suitable approximation for the solution x of $Ax = b$:

1. The *Ritz-Galerkin* approach:
   require that $b - Ax^{(k)} \perp \mathcal{K}^k(A; r^{(0)})$.

2. The *minimum residual* approach:
   require $\|b - Ax^{(k)}\|_2$ to be minimal over $\mathcal{K}^k(A; r^{(0)})$.

3. The *Petrov-Galerkin* approach:
   require that $b - Ax^{(k)}$ is orthogonal to some other suitable k-dimensional subspace.

4. The *minimum error* approach:
   require $\|x - x^{(k)}\|_2$ to be minimal over $A^* \mathcal{K}^k(A^*; r^{(0)})$.

The Ritz-Galerkin approach leads to such popular and well-known methods as Conjugate Gradients, the Lanczos method, FOM, and GENCG. The minimum residual approach leads to methods like GMRES, MINRES, and ORTHODIR.

If we select the k-dimensional subspace, in the third approach, as $\mathcal{K}^k(A^*; s^{(0)})$, then we obtain the Bi-CG, and QMR methods, which both fall in the class of Petrov-Galerkin methods. More recently, hybrids of the these approaches have been proposed, like CGS, Bi-CGSTAB, BiCGSTAB(ℓ), TFQMR, FGMRES, and GMRESR.
The fourth approach may come as a surprise at first sight, since it is not so obvious that we will be able to minimize the error. However, it will turn out that the special form of the subspace makes this possible. In this class we find SYMMLQ and GMERR.

The Ritz-Galerkin approach: FOM and CG

Let us assume, for simplicity again, that $x^{(0)} = 0$, and hence $r^{(0)} = b$. The Ritz-Galerkin conditions imply that $r^{(k)} \perp \mathcal{K}^k(A; r^{(0)})$, and this is equivalent with

$$V_k^*(b - Ax^{(k)}) = 0. \tag{8.2}$$

Since $b = r^{(0)} = \|r^{(0)}\|_2 v^1$, it follows that $V_k^* b = \|r^{(0)}\|_2 e^1$ with e^1 the first canonical unit vector in $I\!R^k$. With $x^{(k)} = V_k y$ we obtain

$$V_k^* A V_k y = \|r^{(0)}\|_2 e^1. \tag{8.3}$$

This system can be interpreted as the system $Ax = b$ projected onto $\mathcal{K}^k(A; r^{(0)})$.

Obviously we have to construct the $k \times k$ matrix $V_k^* A V_k$, but this is, as we have seen readily available from the orthogonalization process:

$$V_k^* A V_k = H_{k,k}, \tag{8.4}$$

so that the $x^{(k)}$ for which $r^{(k)} \perp \mathcal{K}^k(A; r^{(0)})$ can be easily computed by first solving $H_{k,k} y = \|r^{(0)}\|_2 e^1$, and forming $x^{(k)} = V_k y$. This algorithm is known as FOM or GENCG.

When A is symmetric, then $H_{k,k}$ reduces to a tridiagonal matrix $T_{k,k}$, and the resulting method is known as the *Lanczos* method [224]. In clever implementations it is avoided to store all the vectors v^j.
When A is in addition positive definite then we obtain, at least formally, the *Conjugate Gradient* method. In commonly used implementations of this method one forms directly a LU factorization for $T_{k,k}$ and this leads to very elegant short recurrencies for the $x^{(j)}$ and the corresponding $r^{(j)}$, see Section 8.0.5. The positive definiteness is necessary to guarantee the existence of the LU factorization.
Note that for some $j \leq n-1$ the construction of the orthogonal basis must terminate. In that case we have that $AV_{j+1} = V_{j+1} H_{j+1,j+1}$. Let y be the solution of the reduced system $H_{j+1,j+1} y = \|r^{(0)}\| e^1$, and $x^{(j+1)} = V_{j+1} y$. Then it follows that $x^{(j+1)} = x$, i.e., we have arrived at the exact solution,

since $Ax^{(j+1)} - b = AV_{j+1}y - b = V_{j+1}H_{j+1,j+1}y - b = \|r^{(0)}\|V_{j+1}e^1 - b = 0$ (we have assumed that $x^{(0)} = 0$).
The Conjugate Gradients (**CG**) method [197], for symmetric positive definite matrices A, is merely a variant on the Ritz-Galerkin approach, that saves storage and computational effort. A related variant for symmetric indefinite linear systems is known as SYMMLQ [254].

The minimum residual approach: GMRES and MINRES

The creation of an orthogonal basis for the Krylov subspace, in Section 7.0.3, has led to

$$AV_i = V_{i+1}H_{i+1,i}. \tag{8.5}$$

We will still assume that $x^{(0)} = 0$, and hence $r^{(0)} = b$.
We look for an $x^{(i)} \in \mathcal{K}^i(A; r^{(0)})$, that is $x^{(i)} = V_i y$, for which $\|b - Ax^{(i)}\|_2$ is minimal. This norm can be rewritten as

$$\begin{aligned}
\|b - Ax^{(i)}\|_2 &= \|b - AV_i y\|_2 \\
&= \|b - V_{i+1}H_{i+1,i}y\|_2 \\
&= \|V_{i+1}(\|r^{(0)}\|_2 e^1 - H_{i+1,i}y)\|_2.
\end{aligned}$$

Now we exploit the fact that V_{i+1} is an orthonormal transformation with respect to the Krylov subspace $\mathcal{K}^{i+1}(A; r^{(0)})$:

$$\|b - Ax^{(i)}\|_2 = \|\|r^{(0)}\|_2 e^1 - H_{i+1,i}y\|_2,$$

and this final norm can simply be minimized by determining the minimum norm least squares solution of:

$$H_{i+1,i}y = \|r^{(0)}\|_2 e^1. \tag{8.6}$$

In GMRES [275] this is done with Givens rotations, that annihilate the subdiagonal elements in the upper Hessenberg matrix $H_{i+1,i}$.
Note that when A is symmetric the upper Hessenberg matrix $H_{i+1,i}$ reduces to a tridiagonal system. This simplified structure can be exploited in order to avoid storage of all the basis vectors for the Krylov subspace, in a way similar as has been pointed out for CG. The resulting method is known as MINRES [254].

The Petrov-Galerkin approach: Bi-CG and QMR

For unsymmetric systems we can, in general, not reduce the matrix A to a symmetric matrix in a lower-dimensional subspace, by orthogonal projections. The reason is that we can not create an

orthogonal basis for the Krylov subspace by a 3-term recurrence relation [150]. However, we can try to obtain a suitable non-orthogonal basis with a 3-term recurrence, by requiring that this basis is orthogonal to some other basis.
We start by constructing an arbitrary basis for the Krylov subspace:

$$h_{i+1,i}v^{i+1} = Av^i - \sum_{j=1}^{i} h_{j,i}v^j, \tag{8.7}$$

which can be rewritten in matrix notation as $AV_i = V_{i+1}H_{i+1,i}$.

Clearly, we can not use V_i for the projection, but suppose we have a W_i for which $W_i^*V_i = D_i$ (an i by i diagonal matrix with diagonal entries d_i), and for which $W_i^*v^{i+1} = 0$. The coefficients $h_{i+1,i}$ can be chosen, we suggest to choose them such that $\|v^{i+1}\|_2 = 1$.
Then

$$W_i^*AV_i = D_iH_{i,i}, \tag{8.8}$$

and now our goal is to find a W_i for which $H_{i,i}$ is tridiagonal. This means that $V_i^*A^*W_i$ should be tridiagonal also. This last expression has a similar structure as the right-hand side in (8.8), with only W_i and V_i reversed. This suggests to generate the w^i with A^*.
We start with an arbitrary $w^1 \neq 0$, such that $(w^1)^*v^1 \neq 0$. Then we generate v^2 with (8.7), and orthogonalize it with respect to w^1, which means that $h_{1,1} = (w^1)^*Av^1/((w^1)^*v^1)$. Since $(w^1)^*Av^1 = (v^1)^*A^*w^1$, this implies that w^2, generated with

$$h_{2,1}w^2 = A^*w^1 - h_{1,1}w^1,$$

is also orthogonal to v^1.
This can be continued, and we see that we can create bi-orthogonal basis sets $\{v^j\}$, and $\{w^j\}$, by making the new v^i orthogonal to w^1 up to w^{i-1}, and then by generating w^i with the same recurrence coefficients, but with A^* instead of A.
Now we have that $W_i^*AV_i = D_iH_{i,i}$, and also that $V_iA^*W_i = D_iH_{i,i}$. This implies that $D_iH_{i,i}$ is symmetric, and hence $H_{i,i}$ is a tridiagonal matrix, which gives us the desired 3-term recurrence relation for the v^j's, and the w^j's.

We may proceed in a similar way as in the symmetric case:

$$AV_i = V_{i+1}T_{i+1,i}, \tag{8.9}$$

but here we use the matrix $W_i = [w^1, w^2, ..., w^i]$ for the projection of the system

$$W_i^*(b - Ax^{(i)}) = 0,$$

or

$$W_i^* A V_i y - W_i^* b = 0. \tag{8.10}$$

Using (8.9), we find that y satisfies

$$T_{i,i} y = \|r^{(0)}\|_2 e^1,$$

and $x^{(i)} = V_i y$.

This method is known as the Bi-Lanczos method [224]. Similar as for the symmetric case, there is a variant with short recurrences. This method is known as Bi-CG, and it will be further discussed in Section 8.0.5. Bi-CG can be interpreted as the nonsymmetric variant of CG, and one might wonder whether there is also such a nonsymmetric variant of MINRES. Unfortunately, it is not possible to compute, with short recurrences, a minimum residual solution, but the QMR method comes close to it. This method will also be discussed in Section 8.0.5.

The Bi-CG method has seen some offspring in the form of hybrid methods, methods that can be viewed as combinations of Bi-CG with other methods. These methods include CGS, TFQMR, Bi-CGSTAB, and others. They will be discussed in the next Chapter.

The minimum error approach: SYMMLQ and GMERR

We explain these methods for symmetric real matrices. The extension to the complex Hermitian cas is obvious.

In SYMMLQ we minimize the norm of the error $x - x^{(k)}$, for $x^{(k)} = x^{(0)} + AV_k y(k)$, which means that $y^{(k)}$ is the solution of the normal equations

$$V_k^T A A V_k y^{(k)} = V_k^T A(x - x^{(0)}) = V_k^T r^{(0)} = ||r^{(0)}||_2 e^1.$$

This system can be further simplified by exploiting the Lanczos relations (see Sect. 8.0.4), with $T_{k+1,k} \equiv H_{k+1,k}$:

$$V_k^T A A V_k = T_{k+1,k}^T V_{k+1}^T V_{k+1} T_{k+1,k} = T_{k+1,k}^T T_{k+1,k}.$$

A stable way of solving this set of normal equations is based on an $L\widetilde{Q}$ decomposition of $T_{k+1,k}^T$, but note that this is equivalent with the transpose of the $Q_{k+1,k} R_k$ decomposition of $T_{k+1,k}$, which is constructed for MINRES (by Givens rotations), and where R_k is an upper tridiagonal matrix (only the diagonal and the first two codiagonals in the upper triangular part contain non-zero elements):

$$T_{k+1,k}^T = R_k^T Q_{k+1,k}^T.$$

This leads to

$$T_{k+1,k}^T T_{k+1,k} y^{(k)} = R_k^T R_k y^{(k)} = ||r^{(0)}||_2 e^1,$$

from which the basic generating formula for SYMMLQ is obtained:

$$\begin{aligned} x^{(k)} &= x^{(0)} + AV_k R_k^{-1} R_k^{-T} \|r^{(0)}\|_2 e^1 \\ &= x^{(0)} + V_{k+1} Q_{k+1,k} R_k^{-1} R_k^{-T} \|r^{(0)}\|_2 e^1 \\ &= x^{(0)} + (V_{k+1} Q_{k+1,k}) \; (L_k^{-1} \|r(0)\|_2 e^1), \end{aligned} \tag{8.11}$$

with $L_k \equiv R_k^T$. The actual implementation of SYMMLQ [254] is based on an update procedure for $W_k \equiv V_{k+1} Q_{k+1,k}$, and on a three term recurrence relation for $\|r^{(0)}\|_2 L_k^{-1} e^1$.
This approach can be generalized tot unsymmetric systems and then it leads to an algorithm with long recursions: GMERR [325], in a similar way as GMRES leads to long recursions. The error is minimized over the Krylov subspace $A^T \mathcal{K}^k(A^T, r^{(0)})$. For the symmetric case it is shown in [287] that the additional factor A in front of the Krylov subspace may lead to a delay in convergence, with respect to MINRES, by a number of iterations that is necessary to reduce the residual norm by a factor proportional to the condition number of A. In the unsymmetric case we can not simply transfer this observation since the corresponding residual minimizing method GMRES works with a Krylov space in A, while GMERR works with A^T. It seems obvious though that for nearly symmetric systems GMERR may not be expected to have an advantage over GMRES. For highly unsymmetric systems anything may happen. For more information the reader is referred to [325].

8.0.5 Iterative Methods in more detail

In the coming sections we shall describe a number of the most popular and powerful iterative methods, and we shall discuss implementation aspects on vector and parallel computers.

Our focus here is on five groups of methods:

- Conjugate gradient (CG) method, MINRES, and LSQR. CG can be used for linear systems of which the matrix is positive definite symmetric. The method is based on the Lanczos recurrence, which still can be exploited if the matrix is only symmetric. In that case we can determine a mimimum residual solution with MINRES.

  Of course, the conjugate gradient method can also be applied for general linear systems by applying it to the (not explicitly formed) normal equations. A stable way of doing this is achieved by the so-called LSQR method.

- GMRES and GMRES(m). For general nonsymmetric systems one can construct a full orthogonal basis for the solution space. Unfortunately this leads to a demand in memory space which grows linearly with the number of iteration steps. The problem is circumvented by restarting the iteration procedure after m steps, which leads to GMRES(m), but this may lead to less than optimal convergence behavior. Another interesting approach for keeping the length of recurrencies limited is realized in the methods FGMRES and GMRESR.

- Bi-Conjugate Gradient (Bi-CG) method and QMR. This method can be used for the solution of nonsymmetric linear systems. It finds a solution from the Krylov subspace, using the bi-orthogonal basis approach. This has the disadvantage that two matrix vector multiplications are required per iteration step. A well-known variant, and for different reasons to be preferred over Bi-CG, is QMR.

- CGS. A method that has become quite popular over recent years is the so-called conjugate gradient-squared method. It can be seen as a variant of the Bi-CG method insofar as it also exploits the work carried out with both matrix vector multiplications for improving the convergence behavior. Recent improvements over CGS include TFQMR, and GCGS.

- Bi-CGSTAB and Bi-CGSTAB(ℓ). The two matrix vector operations in Bi-CG can also be used to form, at virtually the same computational costs as for Bi-CG itself, a method that can be viewed as the product of Bi-CG and repeated minimal residual steps (or GMRES(1)-steps). The obvious extension Bi-CGSTAB(ℓ) method, in wich Bi-CG is combined with GMRES(ℓ), offers further possibilities for parallelism, besides better convergence properties.

Other methods such as SSOR [192], SOR[318], SIP[296], and the Chebyshev iteration [231, 17, 178] are not treated here. It will be clear from our discussions, however, that most of the vectorization and parallelization approaches easily carry over to those methods.

Preconditioning:
The speed of convergence of the Krylov subspace methods depends on spectral properties of the given matrix, such as the distribution of the eigenvalues, and (in the unsymmetric case) angles between eigenvectors, and on the given right-hand side. These properties are commonly unknown to the user, but nevertheless one often attempts to improve these properties by a concept called *preconditioning*. The idea is to construct a *preconditioning* matrix K, with the property that $Kx = b$ is much easier to solve than for the given matrix A, and K should be close to A in some sense. The latter is rather undefined and in many cases we do not even know exactly what it means, but the general approach is to construct a cheap K, so that K^{-1} approximates the inverse of A. This is supported by the observation for the standard Richardson iteration that convergence is fast when $||K^{-1}A - I|| \ll 1$. In reality, the situation is much more complicated, and often it is already sufficient if $Kz \approx Az$ for a subspace with which the right-hand side makes a small angle. We will devote a complete chapter to the subject (Chapter 9, and as we will see, the choice for a good preconditioner is mainly a matter of trial and error, despite our knowledge of what a good preconditioner should do.
In this chapter we will formulate the algorithms so that a given preconditioner can be included.

The Conjugate Gradient Method

When A is real symmetric, the matrix of the projected system is a symmetric tridiagonal matrix which can be generated by a three-term recurrence relation between the residual vectors. When A is, in addition, positive definite, this tridiagonal system can be solved without any problem. This leads to the conjugate gradient algorithm [197, 178]. The method can be described by the scheme, given in Fig. 8.1. In that scheme, the preconditioner K may be any symmetric positive definite matrix.

Compute $r_0 = b - Ax_0$ for some initial guess x_0
for $i = 1, 2, ...$
  Solve z_{i-1} from $Kz_{i-1} = r_{i-1}$
  $\rho_{i-1} = r_{i-1}^T z_{i-1}$
  if $i = 1$
    $p_1 = z_0$
  else
    $\beta_{i-1} = \frac{\rho_{i-1}}{\rho_{i-2}}$;
    $p_i = z_{i-1} + \beta_{i-1} p_{i-1}$
  endif
  $q_i = Ap_i$
  $\alpha_i = \frac{\rho_{i-1}}{p_i^T q_i}$
  $x_i = x_{i-1} + \alpha_i p_i$
  $r_i = r_{i-1} - \alpha_i q_i$
  check convergence; continue if necessary
end;

Figure 8.1: The CG algorithm

If we choose $K = I$, then the unpreconditioned, or standard, conjugate gradient method results. For $K \neq I$, we have that, when writing $K = LL^T$, this scheme is equivalent to applying the conjugate gradient method to the (preconditioned) equation $L^{-1}AL^{-T}y = L^{-1}b$, with $x = L^{-T}y$. Because of a transformation of variables, the above implementation delivers the x_i and r_i corresponding to $Ax = b$, but note that the r_i are not orthogonal (in fact, the vectors $L^{-1}r_i$ are orthogonal).
Stopping criteria are often based upon the norm of the current residual vector r_i. One could

terminate the iteration procedure, e.g., when ρ_i/ρ_0 is less than some given value *eps*. More robust stopping criteria are based upon estimating the error in x_i, with respect to x, with information obtained from the iteration parameters α_i and β_i. For details see [33].
The CG method minimizes the A-norm of the error over the current subspace, i.e., the current iterate x_i is such that $(x_i - x, A(x_i - x))$ is minimal over all x_i in the Krylov subspace which is spanned by $L^{-1}r_0, L^{-1}r_1, ..., L^{-1}r_{i-1}$. Note that this is accomplished with the aid of only three current vectors r, p, and w. In fact, the CG algorithm can be rewritten as a three-term recurrence relation for the residual vectors r_i, and these vectors form an orthogonal basis for the Krylov subspace.

It can be shown that an upperbound for the number of iteration steps, required to get the A-norm of the error below some prescribed *eps*, is proportional to the square root of the condition number of A (or the matrix $K^{-1}A$, in the preconditioned case). This follows from the well-known upperbound on the residual [22, 178]

$$\|x - x_i\|_A \leq (\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1})^i \|x - x_0\|_A,$$

in which κ is the condition number of $L^{-1}AL^{-T}$.

In practice, the speed of convergence can be considerably faster, most notably in situations in which the extreme eigenvalues are relatively well separated from the rest of the spectrum. The local effects in the convergence pattern of CG have been analyzed in [301].

Since the speed of convergence of the conjugate gradient method depends strongly on the spectrum of the matrix of the system, it is often advantageous to select a "cheap" matrix K (cheap in the sense that $Kw = r$ is easily and efficiently solved) in order to improve the spectral properties.

More references for CG can be found in [24, 33, 43, 178, 274].
Barrett et al [33] gives an easy introduction to the practical usage of iterative methods; it also provides software for some iterative methods in Fortran, C, and Matlab.

Parallelism in the CG method: general aspects

The possibilities in selecting efficient preconditioners K, which allow for a sufficient degree of parallelism and vectorization, will be discussed in a separate chapter, since these aspects are similar for all the methods which follow.
With respect to the different steps in the conjugate gradient method, we make the following remarks.

1. The computation of $p_i = z_{i-1} + \beta_{i-1}p_{i-1}$, $q_i = Ap_i$, and $p_i^T q_i$ can be carried out in combination on successive parts of the vectors. For example, the computation of a segment of the vector q_i can be combined with the contribution from this segment and the corresponding (also

available) segment of p_i to the innerproduct. How well the update for p_i can be combined with these two operations, depends on the bandwidth of the matrix A.
Thus we can exploit the presence of each segment of p_i and q_i in the highest level of memory (vector registers and/or cache memory) by using them more than once. This may help to reduce on expensive memory traffic.

2. The same holds for $Kz_{i-1} = r_{i-1}$ and $r_{i-1}^T z_{i-1}$.
   The equation $Kz_{i-1} = r_{i-1}$ is usually solved in two steps. With $K = LL^T$, we first solve for w from $Lw = r_{i-1}$, followed by solving $L^T z_{i-1} = w$. Note that $r_{i-1}^T z_{i-1}) = w^T w$, so that the computation of parts of the innerproduct can be combined with solving segments of w from $Lw = r_{i-1}$.
   For matrices K with a special structure, e.g., block diagonal, these two operations can further be combined with the computation of a section of $r_i = r_{i-1} - \alpha_i q_i$.

3. The performance for the updating of the approximate solutions x_{i-1} can sometimes be improved. Note that the x_i are not necessary for the iteration process itself, and updating can be carried out only intermittently by storing a number of successive p_i's. Let $P_{i,j}$ denote the matrix with columns $p_i, p_{i+1}, ..., p_{i+j}$. If $\alpha_{i,j}$ is defined as the vector with components $\alpha_i, \alpha_{i+1}, ..., \alpha_{i+j}$, then the vector x_{i+j} can be formed after j iteration steps as $x_{i+j} = x_{i-1} + P_{i,j}\alpha_{i,j}$.

4. The computation of α_i has to be completed before r_{i-1} can be updated. In order to avoid such unwanted synchronization points, alternative conjugate gradient schemes have been proposed, e.g., s-step conjugate gradient [59]. These schemes seem to be more subject to rounding errors [269, 241]. We will discuss the effects of synchronization in more detail in section 8.0.5.

Parallelism in the CG method: Communication overhead

Since on a distributed memory machine communication is required to assemble the innerproduct, it would be nice if we could proceed with other useful computation while the communication takes place. However, as is easily seen, the given CG scheme offers no possibility to overlap this communication time with useful computation. The same observation can be made for the updating of z_{i-1}, which can only take place after the completion of the innerproduct for β_{i-1}.
Several authors ([59, 240, 241]) have attempted to reduce the number of synchronization points. In our formulation of CG there are two such synchronization points, namely the computation of both innerproducts.
Meurant [240] (see also [268]) has proposed a variant in which there is only one synchronization point, however at the cost of a possibly reduced numerical stability, and one additional innerproduct. In this scheme the ratio between computations and memory references is about 2, which is better

than for the standard scheme.
We show a slightly different variant, proposed by Chronopoulos and Gear [59], in Fig. 8.2.

x_0= initial guess; $r_0 = b - Ax_0$
$q_{-1} = p_{-1} = 0; \beta_{-1} = 0$
Solve $Kw_0 = r_0$
$s_0 = Aw_0$
$\rho_0 = r_0^T w_0; \mu_0 = s_0^T w_0$
$\alpha_0 = \rho_0 / \mu_0$
for $i = 0, 1, 2, ...$
    $p_i = w_i + \beta_{i-1} p_{i-1}$
    $q_i = s_i + \beta_{i-1} q_{i-1}$
    $x_{i+1} = x_i + \alpha_i p_i$
    $r_{i+1} = r_i - \alpha_i q_i$
    check convergence; continue if necessary
    Solve $Kw_{i+1} = r_{i+1}$
    $s_{i+1} = Aw_{i+1}$
    $\rho_{i+1} = r_{i+1}^T w_{i+1}$
    $\mu_{i+1} = s_{i+1}^T w_{i+1}$
    $\beta_i = \frac{\rho_{i+1}}{\rho_i}$
    $\alpha_{i+1} = \frac{\rho_{i+1}}{\mu_{i+1} - \rho_{i+1}\beta_i/\alpha_i}$
end

Figure 8.2: The CG algorithm; Chronopoulos-Gear variant

In this scheme all vectors need only be loaded once per pass of the loop, which leads to a better exploitation of the data (improved data locality). However, the price is that we need an additional $2n$ flops per iteration step. Chronopoulos and Gear [59] claim stability, based upon their numerical experiments.
Instead of 2 synchronization points, as in the standard version of CG, we have now only one synchronization point, as the next loop can only be started when the innerproducts at the end of the previous loop have been assembled. Another advantage is that these innerproducts can be computed in parallel.
Chronopoulos and Gear [59] propose to further improve data locality and parallelism in CG by

combining s successive steps. The main drawback in this approach seems to be the potential numerical instability. Depending on the spectral properties of A, the set r_j, ..., $A^{s-1}r_j$ may tend to converge to a vector in the direction of a dominating eigenvector, or, in other words, may tend to dependence for increasing values of s. The authors report successful experiments with this approach, with no serious stability problems, for small values of s. Nevertheless, it seems that s-step CG, because of these problems, has a bad reputation (see also [269]). A similar approach, suggested by Chronopoulos and Kim [60] for restarted processes such as GMRES(m), seems to be more promising.

In [82] another variant of CG was suggested, in which there is more possibility to overlap all of the communication time with useful computations. This variant is nothing but a rescheduled version of the original CG scheme, and is therefore equally stable. The key trick in this approach is to delay the updating of the solution vector by one iteration step. Another advantage is that no additional operations are required.
It is assumed that the preconditioner K can be written as $K = (LL^T)^{-1}$. Using the fact that $r_i^T K^{-1} r_i = (L^{-1}r_i)^T L^{-1} r_i$, we can compute this innerproduct after the solution for the lower triangular factor of the preconditioner, and the communication for the innerproduct may be partly overlapped with computations for the upper triangular part. Furthermore, it is assumed that the preconditioner has a block structure, corresponding to the gridblocks assigned to the processors, so that communication (if necessary) can be overlapped with computation. This leads to the variant of the CG scheme, given in Fig. 8.3.

For a further discussion of this approach, see [82].

MINRES

If the matrix A is real symmetric, but not positive definite, then we can still exploit the Lanczos reduction to tridiagonal form. However, we can not rely on a flawless LU decomposition of the reduced tridiagonal matrix, which is necessary to obtain the coupled two-term recurrencies in CG. An alternative, specially attractive for indefinite systems, is to solve $T_{i+1,i}y = \|r_0\|_2 e_1$ in a least squares sense, similar to the procedure for GMRES. This leads to a minimum residual solution and this approach has become well-known under the name MINRES [254].
Similar as for GMRES, we apply Givens rotations to eliminate the nonzero elements in the lower triangular part of $T_{i+1,i}$, which leads to a QR decomposition, in which $R_{i,i}$ is banded upper triangular, with a bandwidth 3.
The solution y of the reduced system can be written as

$$y = R_{i,i}^{-1} Q_i^T ||r_0||_2 e_1,$$

x_0= initial guess; $r_0 = b - Ax_0$
$p_{-1} = 0; \beta_{-1} = 0; \alpha_{-1} = 0$
Solve $Ls = r_0$
$\rho_0 = s^T s$
for $i = 0, 1, 2, ...$
    Solve $L^T w_i = s$
    $p_i = w_i + \beta_{i-1} p_{i-1}$
    $q_i = A p_i$
    $\gamma = p_i^T q_i$
    $x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$
    $\alpha_i = \frac{\rho_i}{\gamma}$
    $r_{i+1} = r_i - \alpha_i q_i$
    $s = L^{-1} r_{i+1}$
    $\rho_{i+1} = s^T s$
    check convergence
    continue if necessary **else**
        $x_{i+1} = x_i + \alpha_i p_i$
        quit
    $\beta_i = \frac{\rho_{i+1}}{\rho_i}$
end

Figure 8.3: A better parallelizable variant of CG

and for the iteratively approximated solution x_i, we have that

$$\begin{aligned} x_i &= (V_i R_{i,i}^{-1}) Q_i^T ||r_0||_2 e_1 \\ &= W_i Q_i^T ||r_0||_2 e_1. \end{aligned}$$

The columns of W_i can be obtained with a simple three term recurrence, because of the 3-diagonal banded structure of $R_{i,i}$. The remaining part of the expression, $Q_i^T ||r_0||_2 e_1$, can also easily be updated in each iteration. In this way, the approximate solution can be computed at each iteration step, without the necessity to store the full basis for the Krylov subspace, as is done in GMRES. The above described coupled 3-term recurrence relations lead to the algorithmic form of MINRES as represented in Fig. 8.4. This form has been inspired by a MATLAB routine published in [151].

Compute $v_1 = b - Ax_0$ for some initial guess x_0
$\beta_1 = ||v_1||_2$; $\eta = \beta_1$;
$\gamma_1 = \gamma_0 = 1$; $\sigma_1 = \sigma_0 = 0$;
$v_0 = 0$; $w_0 = w_{-1} = 0$;
for $i = 1, 2, ...$
The Lanczos recurrence:
$v_i = \frac{1}{\beta_i} v_i$; $\alpha_i = v_i^T A v_i$;
$v_{i+1} = Av_i - \alpha_i v_i - \beta_i v_{i-1}$
$\beta_{i+1} = ||v_{i+1}||_2$
QR part:
old Givens rotations on new column of T:
$\delta = \gamma_i \alpha_i - \gamma_{i-1}\sigma_i\beta_i$; $\rho_1 = \sqrt{\delta^2 + \beta_{i+1}^2}$
$\rho_2 = \sigma_i\alpha_i + \gamma_{i-1}\gamma_i\beta_i$; $\rho_3 = \sigma_{i-1}\beta_i$
New Givens rotation for subdiag element:
$\gamma_{i+1} = \delta/\rho_1$; $\sigma_{i+1} = \beta_{i+1}/\rho_1$
Update of solution (with $W_i = V_i R_{i,i}^{-1}$)
$w_i = (v_i - \rho_3 w_{i-2} - \rho_2 w_{i-1})/\rho_1$
$x_i = x_{i-1} + \gamma_{i+1}\eta w_i$
$||r_i||_2 = |\sigma_{i+1}| \; ||r_{i-1}||_2$
check convergence; continue if necessary
$\eta = -\sigma_{i+1}\eta$
end

Figure 8.4: The unpreconditioned MINRES algorithm

The usage of the 3-term recurrence relation for the columns of W_i makes MINRES very vulnerable for rounding errors, as has been shown in [287]. It has been shown that rounding errors are propagated to the approximate solution with a factor proportional to the square of the condition number of A, whereas in GMRES these errors depend only on the condition number itself. Therefore, MINRES should not be used for ill-conditioned systems. If storage is no problem then GMRES should be preferred for ill-conditioned systems; if storage is a problem then one might consider the usage of SYMMLQ [254]. SYMMLQ, however, may converge a good deal slower than MINRES for ill-conditioned systems. For more details on this, see [287].

For a symmetric positive definite preconditioner of the form LL^T, the MINRES algorithm can be applied to the explicitly preconditioned system

$$L^{-1}AL^{-T}A\widetilde{x} = L^{-1}b, \quad \text{with} \quad x = L^{-1}\widetilde{x}.$$

Explicit inversions with L and L^T can be avoided, all one has to do is to solve linear systems with these matrices.
We can not, without risk, apply MINRES to $K^{-1}Ax = K^{-1}b$, for symmetric K and A, when the matrix $K^{-1}A$ is unsymmetric. If K is positive definite then we may replace the innerproduct by the bilinear form (x, Ky), with respect to which the matrix $K^{-1}A$ is symmetric.
Symmetric positive definite preconditioners are often not very effective for indefinite systems. The construction of effective preconditioners for symmetric indefinite A is largely an open problem.

With respect to parallelism, or other implementation aspects, MINRES can be treated as CG. Note that most of the variables in MINRES may overwrite old ones that are obsolete.

Least Squares Conjugate Gradients

Symmetry of the real matrix A (and the preconditioner) is required for the generation of the basis vectors r_i, for the Krylov subspace, by a three-term recurrence relation. Positive definiteness of A is necessary in order to define a suitable norm in which the error has to be minimized. If one of these properties is missing, then the CG algorithm is likely to fail.

A robust but often rather slowly converging iterative method for general linear systems is obtained by applying conjugate gradient to the normal equations for the linear system

$$A^*Ax = A^*b.$$

For overdetermined linear systems this leads to the minimum norm least squares solution of $Ax = b$. For a discussion of this approach and a comparison with other iterative techniques for overdetermined linear systems, see [302].

For linear systems with a square nonsingular matrix A, the normal equations approach has obvious disadvantages when compared to CG applied to the symmetric positive definite case. First, we have to compute either two matrix vector products per iteration step (with A and A^*) or only one (with A^*A), but the latter is usually less sparse than A, so that the amount of computational work almost doubles. Second, and even more serious, the condition number of A^*A is equal to the square of the condition number of A, so that we may expect a large number of iteration steps.

Nevertheless, in some situations, for example with indefinite systems, the least squares conjugate gradient approach may be useful. It has been observed that straightforward application of CG to the normal equations may suffer from numerical instability, especially when the matrix of the linear system itself is ill-conditioned. According to Paige and Saunders [255] this effect is, to a large extent, due to the explicit computation of the vector A^*Ap_i. However, the equality $(p^i)^*A^*Ap^i = (Ap^i)^*Ap^i$ allows us to adjust the scheme slightly. The adjusted scheme, proposed originally by Björck and Elfving [39], is reported to produce better results and runs as depicted in Fig. 8.5.

$x^{(0)}$ is an initial guess; $s^{(0)} = b - Ax^{(0)}$;
$r^{(0)} = A^*s^{(0)}; p^{(0)} = r^{(0)}$;
$\rho_0 = (r^{(0)})^* r^{(0)}; p^{(-1)} = 0; \beta_{-1} = 0$;
for $i = 0, 1, 2, ...$
  $p^{(i)} = r^{(i)} + \beta_{i-1}p^{(i-1)}$;
  $w^{(i)} = Ap^{(i)}$;
  $\alpha_i = \frac{\rho_i}{(w^{(i)})^* w^{(i)}}$;
  $x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$;
  $s^{(i+1)} = s^{(i)} - \alpha_i w^{(i)}$;
  $r^{(i+1)} = A^*s^{(i+1)}$;
  if $x^{(i+1)}$ is accurate enough then quit;
  $\rho_{i+1} = (r^{(i+1)})^* r^{(i+1)}$;
  $\beta_i = \frac{\rho_{i+1}}{\rho_i}$;
end

Figure 8.5: The CGNE algorithm

Note that $r^{(i)}$ is just the residual of the normal equations and $s^{(i)} = b - Ax^{(i)}$ is the residual of the given linear system itself. This algorithm differs from the CG scheme in Section 8.0.5 by the fact that at each step the matrices A and A^* are involved. For general sparsity structures we have to find storage schemes such that both Ap^i and $A^*s^{(i+1)}$ can be computed efficiently.

Another way of reducing the loss of accuracy that accompanies the normal equations approach, is to apply the Lanczos algorithm (which is closely related to the conjugate gradient algorithm) to the linear system

$$\begin{pmatrix} I & A \\ A^* & 0 \end{pmatrix} \begin{pmatrix} r \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}$$

This forms the basis for the LSQR algorithm, proposed by Paige and Saunders [255]. With respect to vector and parallel computing, however, LSQR can be treated similar as the Björck and Elfving scheme. Software for LSQR has been published in [255].

Preconditioning can be applied in two different ways:

1. Form the normal equations for $KAx = Kb$:

$$A^*K^*KAx = A^*K^*Kb$$

   or

$$B^*Bx = B^*Kb,$$

   with

$$B = KA.$$

2. With $K = LU$ form the normal equations for $L^{-1}AU^{-1}y = L^{-1}b$, where $x = U^{-1}y$:

$$U^{-*}A^*L^{-*}L^{-1}AU^{-1}y = U^{-*}A^*L^{-*}L^{-1}b$$

   or

$$C^*Cy = C^*L^{-1}b,$$

   with

$$C = L^{-1}AU^{-1}$$

In both cases we solve the normal equations for some linear system related to $Ax = b$.

GMRES and GMRES(m)

As we have seen in Section 7.0.3, the Arnoldi method can be used to make an orthonormal basis for the Krylov subspace for an unsymmetric matrix. This leads to a reduced matrix which has upper Hessenberg form. This reduced system is exploited in the GMRES method [275] in order to construct approximate solutions for which the norm of the residual is minimal with respect to the Krylov subspace. In Section 8.0.4 we have pointed out how that can be done.

A main practical disadvantage of GMRES (and of related methods as well) is that we have to store all the successive residual vectors and that the construction of the projected system becomes increasingly complex as well. An obvious way to alleviate this disadvantage is to restart the method after each m steps, where m is a suitably chosen integer value. Thi algorithm is referred

to as GMRES(m); the not-restarted version is often called 'full' GMRES.
The choice of m requires some skill and experience with the type of problems that one wants to solve. Taking m too small may result in rather poor convergence or even in no convergence at all. In [206] experiments with the choice of m are reported and discussed.

We present in Fig.8.6 the modified Gram-Schmidt version of GMRES(m) for the solution of the linear system $Ax = b$. The application to preconditioned systems, for instance, $K^{-1}Ax = K^{-1}b$ is straightforward.

$r = b - Ax^{(0)}$, for a given initial guess $x^{(0)}$
for $j = 1, 2, \ldots.$
  $\beta = ||r||_2$, $v^1 = r/\beta$; $\widehat{b} = \beta e^1$;
  for $i = 1, 2, \ldots, m$
    $w = Av^i$;
    for $k = 1, \ldots, i$
      $h_{k,i} = (v^k)^* w$; $w = w - h_{k,i} v^k$;
    $h_{i+1,i} = ||w||_2$; $v^{i+1} = w/h_{i+1,i}$;
    for $k = 2, .., i$
      $r_{k-1,i} = c_{k-1} h_{k-1,i} + s_{k-1} h_{k,i}$
      $r_{k,i} = -s_{k-1} h_{k-1,i} + c_{k-1} h_{k,i}$
    $\delta = \sqrt{h_{i,i}^2 + h_{i+1,i}^2}$; $c_i = h_{i,i}/\delta$; $s_i = h_{i+1,i}/\delta$
    $r_{i,i} = c_i h_{i,i} + s_i h_{i+1,i}$
    $\widehat{b}_{i+1} = -s_i \widehat{b}_i$; $\widehat{b}_i = c_i * \widehat{b}_i$
    $\rho = |\widehat{b}_{i+1}|$ $(= ||b - Ax^{((j-1)m+i)}||_2)$
    if ρ is small enough **then**
      ($n_r = i$; goto *SOL*);
  $n_r = m$, $y_{n_r} = \widehat{b}_{n_r}/r_{n_r,n_r}$
SOL: **for** $k = n_r - 1, \ldots, 1$
    $y_k = (\widehat{b}_k - \sum_{i=k+1}^{n_r} r_{k,i} y_i)/r_{k,k}$
  $x = \sum_{i=1}^{n_r} y_i v^i$; **if** ρ small enough quit
  $r = b - Ax$

Figure 8.6: unpreconditioned GMRES(m) with modified Gram-Schmidt

For modest values of m, the following components require most of the CPU time:

- the orthogonalization of w against the previous v^j's
- the computation of one matrix vector product with A per iteration
- the preconditioning step, that is one operation with K per iteration
- the assembly of x after the solution of the reduced system

These components are essentially the same as for the other iterative methods. The efficient implementation on various computers is rather obvious, except for the orthogonalization step. In Section 7.0.3 we have shown how this part of the algorithm can be reorganized in order to obtain more efficiency on distributed memory computers.

In GMRES the vector $x^{(i)}$ is constructed to minimize $||b-Ay||_2$ over all y with $y-x^{(0)} \in \mathcal{K}_i(A; r^{(0)})$. Obviously the correction of $x^{(i)}$ with respect to $x^{(0)}$ minimizes the residual over this Krylov subspace, so that the residual does not increase as the iteration process proceeds. Since the dimension of the Krylov subspace is bounded by n, the method terminates in at most n steps if rounding errors are absent. In practice, this finite termination property is of no importance, since these iterative methods are attractive, with respect to direct methods, only when they deliver a suitable approximation to the solution in far less than n steps. This does not imply that we may forget about the effect of rounding errors. They can easily spoil a potentially nice convergence behavior, because of an early loss of orthogonality in the construction of the projected system.
Walker [321] has proposed using Householder transformations for the construction of the orthogonal basis in the Arnoldi-part of GMRES, and this approach may be expected to lead to better results in some cases. On the other hand, this method is quite expensive, and in most practical situations the modified Gram-Schmidt orthogonalization method will do almost as well—while being about three times cheaper in arithmetic.
In any case, in the inner loop of GMRES(m) the upper Hessenberg matrix $H_{i+1,i}$, corresponding to the projected system, is constructed by orthogonalizing the basis for the Krylov subspace. The matrix $H_{i+1,i}$ is represented by the elements $h_{k,j}$. In order to solve the resulting small linear least squares system (see Section 8.0.4), this matrix is reduced by Givens rotations to an upper triangular matrix $R_{i,i}$, represented by the elements $r_{k,j}$. This upper triangular system is solved in the solution phase, indicated by 'SOL' in Fig. 8.6, and the solution is used to form the approximate solution x from a combination of basis vectors of the Krylov subspace.

If we solve the reduced system with the $R_{k,k}$ matrix that we have just before the application of the k-th Givens rotation, that is the rotation that should annihilate the element in the $(k+1, k)$ position of $H_{k+1,k}$, then we obtain the FOM method. As we have seen in Section 8.0.4, FOM can

be viewed as a Galerkin method.
There are various different implementations of FOM and GMRES. Among those equivalent with GMRES are: Orthomin [319], Orthodir [210], Axelsson's method [23], and GENCR [147]. These methods are often more expensive than GMRES per iteration step. Orthomin seems to be still popular, since this variant can be easily truncated (Orthomin(s)), in contrast to GMRES. The truncated and restarted versions of these algorithms are not necessarily mathematically equivalent. Methods that are mathematically equivalent with FOM are: Orthores [210] and GENCG [62, 326]. In these methods the approximate solutions are constructed such that they lead to orthogonal residuals (which form a basis for the Krylov subspace; analogously to the CG method). A good overview of all these methods and their relations is given in [274].

There is an interesting and simple relation between the Ritz-Galerkin approach (FOM and CG) and the minimum residual approach (GMRES and MINRES). In GMRES the projected system matrix $H_{i+1,i}$ is transformed by Givens rotations to an upper triangular matrix (with last row equal to zero). So, in fact, the major difference between FOM and GMRES is that in FOM the last $((i+1)$-th row is simply discarded, while in GMRES this row is rotated to a zero vector. Let us characterize the Givens rotation, acting on rows i and $i+1$, in order to zero the element in position $(i+1,i)$, by the sine s_i and the cosine c_i. Let us further denote the residuals for FOM with an subcript F and those for GMRES with subcript G. Then it has been shown in [69] that, if $c_k \neq 0$, the residuals of FOM and GMRES are related as

$$\|r_F^{(k)}\|_2 = \frac{\|r_G^{(k)}\|_2}{\sqrt{1-(\|r_G^{(k)}\|_2/\|r_G^{(k-1)}\|_2)^2}}. \tag{8.12}$$

The condition $c_k \neq 0$ means that $\|r_G^{(k)}\|_2 \neq \|r_G^{(k-1)}\|_2$ (in which case there is a local stagnation and the FOM solution is not defined for that value of k).
From this relation we see that if for some value of k GMRES leads to a significant reduction in the norm of the residual with respect to iteration step $k-1$, then FOM gives about the same result as GMRES. On the other hand when GMRES locally stagnates ($c_k = 0$), then FOM has a (recoverable) breakdown at the same iteration step.
Because of this relation, we can link the convergence behaviour of GMRES with the convergence of *Ritz values* (the eigenvalues of the "FOM" part of the upper Hessenberg matrix). This has been exploited in [314], for the analysis and explanation of local effects in the convergence behaviour of GMRES. It has been shown that as soon as a Ritz value of $H_{i,i}$ has converged in modest accuracy to some eigenvalue of A, then the convergence is from then on determined by the remaining eigenvalues (and eigenvectors). In practical cases this often implies an increase in the speed of convergence: the so-called *super-linear convergence behavior* of GMRES.

GMRES with variable preconditioning

In many practical situations it is attractive or even necessary to work with a different preconditioner per iterations step, which is prohibited in the classical Krylov subspace methods. One such situation arises in *domain decomposition* contexts. The solution process is then decomposed over the different subdomains, and per subdomain one might wish to use an iteration method to get an approximate solution for that domain. The coupled approximated solutions over the entire domain may be viewed as the result of the application of an approximating operator for A^{-1}, and this is just what a preconditioner does. However, since iterative processes have been used for the subdomains, the approximating operator is very likely to be different for successive approximating steps over the whole domain.

Apart from this motivation for working with variable preconditioners, it is also an attractive idea to try to get better approximations for $A^{-1}r$, given the vector r, during the iteration process. There are different approaches to accomplish this. One is to try to improve the preconditioner with updates from the Krylov subspace. This has been suggested first by Eirola and Nevanlinna [143]. Their approach leads to iterative methods that are related to Broyden's method [42], which is a Newton type of method. The Broyden method can be obtained from this update-approach if we do not restrict ourselves to Krylov subspaces. See [320] for a discussion on the relation of these methods. A more straightforward approach, mathematically related to the Eirola-Nevanlinna approach, is to compute approximations for $A^{-1}r$ by a few steps of GMRES again. The combined process can be viewed as a repeated GMRES scheme, although the approximations for $A^{-1}r$ may also be computed with any other method of choice. Nevertheless, if GMRES is the method of choice then there may be little reason not to choose GMRES again for approximating $A^{-1}r$. This repeated GMRES scheme is referred to as GMRESR and we will describe it in some more detail.

The updated preconditioners can not be applied right away in the GMRES method, since the preconditioned operator now changes from step to step, and we are not forming a regular Krylov subspace. We will discuss two different ways to circumvent this problem. The first approach is a subtle variant of GMRES, called FGMRES [271], and described by the scheme given in Fig. 8.7:

If we apply FGMRES with a fixed preconditioner, that is if we always compute the approximation for $A^{-1}v^j$ as $K^{-1}v^j$, for a fixed preconditioner K, the FGMRES(m) is equivalent with GMRES(m) applied to the system $AK^{-1}y = b$; $x = K^{-1}y$, which is called right preconditioned GMRES(m). We see that by a simple adjustment to GMRES we obtain FGMRES. The price we have to pay is additional storage for the vectors z^j. A serious drawback of this appealing scheme is that it may be break down, in contrast to GMRES, since $h_{i+1,i}$ may be zero which prevents the computation of a new vector v_{i+1}. In GMRES this is no problem because it indicates that we have arrived at the subspace that contains the exact solution, but for FGMRES this is not necessarily the case.

$r = b - Ax^{(0)}$, for a given initial guess $x(0)$
for $j = 1, 2, \ldots.$
  $\beta = ||r||_2$, $v^1 = r/\beta$; $\widehat{b} = \beta e^1$;
  for $i = 1, 2, \ldots, m$
    Compute z^j as an approx. to $A^{-1}v^j$; $w = Az^i$;
    for $k = 1, \ldots, i$
      $h_{k,i} = (v^k)^* w$; $w = w - h_{k,i}v^k$;
    $h_{i+1,i} = ||w||_2$; $v_{i+1} = w/h_{i+1,i}$;
    for $k = 2, .., i$
      $r_{k-1,i} = c_{k-1}h_{k-1,i} + s_{k-1}h_{k,i}$
      $r_{k,i} = -s_{k-1}h_{k-1,i} + c_{k-1}h_{k,i}$
    $\delta = \sqrt{h_{i,i}^2 + h_{i+1,i}^2}$; $c_i = h_{i,i}/\delta$; $s_i = h_{i+1,i}/\delta$
    $r_{i,i} = c_i h_{i,i} + s_i h_{i+1,i}$
    $\widehat{b}_{i+1} = -s_i \widehat{b}_i$; $\widehat{b}_i = c_i * \widehat{b}_i$
    $\rho = |\widehat{b}_{i+1}|$ $(= ||b - Ax^{((j-1)m+i)}||_2))$
    if ρ is small enough **then**
      $(n_r = i$; goto $SOL)$;
  $n_r = m$, $y_{n_r} = \widehat{b}_{n_r}/r_{n_r,n_r}$
SOL: **for** $k = n_r - 1, \ldots, 1$
    $y_k = (\widehat{b}_k - \sum_{i=k+1}^{n_r} r_{k,i}y_i)/r_{k,k}$
  $x = \sum_{i=1}^{n_r} y_i z^i$; **if** ρ small enough quit
  $r = b - Ax$

Figure 8.7: FGMRES(m)

We will now discuss another variant, GMRESR [315], which does not have this drawback, but which forces us to rewrite the GMRES algorithm as a (much simpler) GCR-scheme. In [315] it has been shown how the GMRES-method, or more precisely, the GCR-method, can be combined effectively with other iterative schemes. The iteration steps of GMRES (or GCR) are called outer iteration steps, while the iteration steps of the preconditioning iterative method are referred to as inner iterations. The combined method is called GMRES$\star$, where $\star$ stands for any given iterative scheme; in the case of GMRES as the inner iteration method, the combined scheme is called GMRESR [315].

The GMRES$\star$ algorithm can be represented as in Fig. 8.8. Note that the inner iteration, in order to obtain the approximated solution, can be carried out by any method available, for instance with a preconditioned iterative solver.

$x^{(0)}$ is an initial guess; $r^{(0)} = b - Ax^{(0)}$;
for $i = 0, 1, 2, 3, ...$
  Let $z^{(m)}$ be the approximate solution
    of $Az = r^{(i)}$, obtained after m steps of
    an iterative method.
  $c = Az^{(m)}$ (often available from the
      iteration method)
  for $k = 0, ..., i-1$
    $\alpha = (c^k)^* c$
    $c = c - \alpha c^k$
    $z^{(m)} = z^{(m)} - \alpha u^k$
  $c^i = c/\|c\|_2$; $u^i = z^{(m)}/\|c\|_2$
  $x^{(i+1)} = x^{(i)} + ((c^i)^* r^{(i)}) u^i$
  $r^{(i+1)} = r^{(i)} - ((c_i)^* r^{(i)}) c^i$
  if $x^{(i+1)}$ is accurate enough then quit
end

Figure 8.8: The GMRES$\star$ algorithm

A sufficient condition to avoid breakdown in this method ($\|c\|_2 = 0$) is that the norm of the residual at the end of an inner iteration is smaller than the norm of the right-hand side residual: $\|Az^{(m)} - r^{(i)}\|_2 < \|r^{(i)}\|_2$. This can easily be controlled during the inner iteration process. If stagnation occurs, i.e. no progress at all is made in the inner iteration, then it is suggested in [315] to do one (or more) steps of the LSQR method, which guarantees a reduction (but this reduction is often only small).

The idea behind these flexible iteration schemes is that we explore parts of high-dimensional Krylov subspaces, hopefully localizing almost the same approximate solution that full GMRES would find over the entire subspace, but now at much lower computational costs. For the inner iteration we may select any appropriate solver, for instance, one cycle of GMRES(m), since then we have also locally an optimal method, or some other iteration scheme, like for instance Bi-CGSTAB.

In [79] it is proposed to keep the Krylov subspace, that is built in the inner iteration, orthogonal with respect to the Krylov basis vectors generated in the outer iteration. Under various circumstances this helps to avoid inspection of already investigated parts of Krylov subspaces in future inner iterations. The procedure works as follows.
In the outer iteration process the vectors c^0, ..., c^{i-1} build an orthogonal basis for the Krylov subspace. Let C_i be the n by i matrix with columns c^0, ..., c^{i-1}. Then the inner iteration process at outer iteration i is carried out with the operator A_i instead of A, and A_i is defined as

$$A_i = (I - C_i C_i^T)A. \tag{8.13}$$

It is easily verified that $A_i z \perp c^0, ..., c^{i-1}$ for all z, so that the inner iteration process takes place in a subspace orthogonal to these vectors. The additional costs, per iteration of the inner iteration process, are i inner products and i vector updates. In order to save on these costs, one should realize that it is not necessary to orthogonalize with respect to all previous c-vectors, and that "less effective" directions may be dropped, or combined with others. In [248, 30, 79] suggestions are made for such strategies. Of course, these strategies are only attractive in cases where we see too little residual reducing effect in the inner iteration process in comparison with the outer iterations of GMRES$\star$.

Note that if we carry out the preconditioning by doing a few iterations of some other iteration process, then we have inner-outer iteration schemes; these have been discussed earlier in [176, 174].

The idea of variable preconditioning has been pursued by different authors. Axelsson and Vassilevski [28] have proposed a Generalized Conjugate Gradient method with variable preconditioning, Saad [271, 274] has proposed a scheme very similar to GMRES, called Flexible GMRES (FGMRES), and Van der Vorst and Vuik have published a GMRESR scheme. FGMRES has received most attention, presumably because it is so easy to implement: only the update directions in GMRES have to be preconditioned, and each update may be preconditioned differently. This means that only one line in the GMRES algorithm has to be adapted. The price to be paid is that the method is not longer robust; it may break down. The GENCG and GMRESR schemes are slightly more expensive in terms of memory requirements and in computational overhead per iteration step. The main difference between the two schemes is that GENCG in [28] works with Gram-Schmidt orthogonalization, whereas GMRESR makes it possible to use modified Gram-Schmidt. This may give GMRESR an advantage in actual computations. In exact arithmetic GMRESR and GENCG should produce the same results for the same preconditioners. Another advantage of GMRESR over algorithm 1 in [28] is that only one matrix vector product is used per iteration step in GMRESR;

GENCG needs two matrix vector products per step.
In GMRESR the residual vectors are preconditioned and if this gives a further reduction then GMRESR does not break down. This gives more control over the method in comparison with FGMRES. In most cases though, the results are about the same, and then the efficient scheme for FGMRES has an advantage. It should be noted, however, that if GMRESR is carried out with GMRES in the inner-loop, then the combined method can be coded just as efficiently with respect to computational overhead as FGMRES [79, 154]. Finally, an interesting aspect of GMRESR is that, unlike FGMRES, it is easy to make a version with truncated orthogonality relations.

Bi-CG and QMR

In Section 8.0.4 we have seen how projections with respect to a space generated with A^* led to a projected system

$$W_i^* A V_i y = W_i^* b,$$

or

$$T_{i,i} y = ||r_0||_2 e^1, \quad \text{with } x^{(i)} = V_i y.$$

The basis vectors v^j for $\mathcal{K}^i(A; r^{(0)})$ are orthogonal with respect to the basis vectors w^k for the adjoint Krylov subspace $\mathcal{K}^i(A^*, w^1)$, for $j \neq k$.
In the construction of these dual bases, we have assumed that $d_i \neq 0$, that is $(w^i)^* v^i \neq 0$. The generation of the bi-orthogonal basis breaks down if for some i the value of $(w^i)^* v^i = 0$, this is referred to in literature as a *serious breakdown*. Likewise, when $(w^i)^* v^i \approx 0$, we have a near-breakdown. The way to get around this difficulty is the so-called *Look-ahead strategy*, which comes down to taking a number of successive basis vectors for the Krylov subspace together and to make them blockwise bi-orthogonal. This has been worked out in detail in [257], [158], [159], and [160].
Another way to avoid breakdown is to restart as soon as a diagonal element gets small. Of course, this strategy looks surprisingly simple, but one should realise that at a restart the Krylov subspace, that has been built up so far, is thrown away, which destroys possibilities for faster (i.e., superlinear) convergence.

We can try to construct an LU-decomposition, without pivoting, of $T_{i,i}$. If this decomposition exists, then, similar as for CG, it can be updated from iteration to iteration and this leads to a recursive update of the solution vector, which avoids to save all intermediate r and w vectors. This variant of Bi-Lanczos is usually called *Bi-Conjugate Gradients*, or shortly Bi-CG [152]. In Bi-CG, the d_i are chosen such that $v^i = r^{(i-1)}$, similar as for CG. The correspondingly rescaled w^i will be denoted as $\widehat{r}^{(i-1)}$, in order to reflect that they are computed in the same way as the residuals. The only difference is that the starting vector may be different, and that A^* is used in the recurrence, rather than A.

In Fig. 8.9, we show schematically the algorithm for Bi-CG, for a real-valued unsymmetric system $Ax = b$, and with preconditioner K.

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
Choose $s^{(0)}$ (for instance, $s^{(0)} = r^{(0)}$)
for $i = 1, 2, \ldots.$
  Solve z^{i-1} from $Kz^{i-1} = r^{(i-1)}$
  solve $\widetilde{z}^{i-1}$ from $K^T\widetilde{z}^{i-1} = s^{(i-1)}$
  $\rho_{i-1} = (z^{i-1})^* s^{i-1})$
  if $i = 1$
    $p^1 = z^0$
    $\widetilde{p}^1 = \widetilde{z}^0$
  else
    $\beta_{i-1} = \frac{\rho_{i-1}}{\rho_{i-2}}$;
    $p^i = z^{i-1} + \beta_{i-1} p^{i-1}$
    $\widetilde{p}^i = \widetilde{z}^{i-1} + \beta_{i-1}\widetilde{p}^{i-1}$
  endif
  $q^i = Ap^i$
  $\widetilde{q}^i = A^*\widetilde{p}^i$
  $\alpha_i = \frac{\rho_{i-1}}{(\widetilde{p}^i)^* q^i}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^i$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^i$
  $s^{(i)} = s^{(i-1)} - \alpha_i \widetilde{q}^i$
  check convergence; continue if necessary
end;

Figure 8.9: The Bi-CG algorithm

Of course one can in general not be certain that an LU decomposition (without pivoting) of the tridiagonal matrix $T_{i,i}$ exists, and this may lead also to breakdown (a breakdown of the *second kind*), of the Bi-CG algorithm. Note that this breakdown can be avoided in the Bi-Lanczos formulation of the iterative solution scheme, e.g., by making an LU-decomposition with 2 by 2 block diagonal elements [32]. It is also avoided in the QMR approach (see below).

Note that for symmetric matrices Bi-Lanczos generates the same solution as Lanczos, provided that $w^1 = r^{(0)}$, and under the same condition Bi-CG delivers the same iterands as CG for positive definite matrices. However, the Bi-orthogonal variants do so at the cost of two matrix vector operations per iteration step.

The QMR method [160] relates to Bi-CG in a similar way as MINRES relates to CG.
We start from the recurrence relations for the v^j:

$$AV_i = V_{i+1}T_{i+1,i}.$$

We would like to identify the $x^{(i)}$, with $x^{(i)} \in \mathcal{K}^i(A; r^{(0)})$, or $x^{(i)} = V_i y$, for which

$$\|b - Ax^{(i)}\|_2 = \|b - AV_i y\|_2 = \|b - V_{i+1}T_{i+1,i}y\|_2$$

is minimal, but the problem is that V_{i+1} is not orthogonal. However, pretend that the columns of V_{i+1} are orthogonal, then

$$\begin{aligned} \|b - Ax^{(i)}\|_2 &= \|V_{i+1}(\|r^{(0)}\|_2 e^1 - T_{i+1,i}y)\|_2 \\ &= \|(\|r_0\|_2 e^1 - T_{i+1,i}y)\|_2 \equiv q_Q^k, \end{aligned}$$

and in [160] it is suggested to solve the projected miniminum norm least squares problem at the right-hand side. Since, in general, the columns of V_{i+1} are not orthogonal, the computed $x^{(i)} = V_i y$ does not solve the minimum residual problem, and therefore this approach is referred to as a Quasi-minimum residual approach [160]. Also, when the columns of V_{i+1} are not orthogonal, then the *quasi-residual norm* q_Q^k is not equal to the norm of the QMR-residual $\|b - Ax^{(k)}\|_2$, but note that $\|b - Ax^{(k)}\|_2 \leq \sqrt{k} q_Q^k$.
The above derivation leads to the simplest form of the QMR method. For the algorithm, as well as simple codes, we refer to [33]. More professional code for QMR, including Look-ahead strategies, can be found in netlib. From a parallel point of view, QMR behaves virtually the same as Bi-CG (and CG), and it needs similar approaches for making its performance better on parallel computers.

In [160] the QMR method is carried out on top of a look-ahead variant of the bi-orthogonal Lanczos method, which makes the method more robust. Experiments indicate that although QMR has a much smoother convergence behaviour than Bi-CG, it is not essentially faster than Bi-CG. This is confirmed explicitly by the following relation for the Bi-CG residual $r_B^{(k)}$ and the quasi-residual q_Q^k (in exact arithmetic):

$$\|r_B^{(k)}\|_2 = \frac{q_Q^k}{\sqrt{1 - (q_Q^k / q_Q^{k-1})^2}},$$

see [69]: Theorem 4.1.
This relation, which is similar to the relation for GMRES and FOM, shows that when QMR gives a significant reduction at step k, then Bi-CG and QMR have arrived at residuals of about the same norm, provided, of course, that the same set of starting vectors has been used, and provided that Bi-CG has not suffered from a breakdown of the second kind. Because of its smoother convergence and being more robust, QMR has to be preferred over Bi-CG.

It is tempting to compare QMR with GMRES, but this is difficult. GMRES really minimizes the 2-norm of the residual, but at the cost of increasing the work of keeping all residuals orthogonal and increasing demands for memory space. QMR does not minimize this norm, but often it has a comparable fast convergence as GMRES, at the cost of twice the amount of matrix vector products per iteration step. However, the generation of the basis vectors in QMR is relatively cheap and the memory requirements are limited and modest. Several variants of QMR, or rather Bi-CG, have been proposed, which increase the effectiveness of this class of methods in certain circumstances. These variants will be discussed in subsequent sections.

In a recent paper, Zhou and Walker [329] have shown that the Quasi-Minimum Residual approach can be followed for other methods, such as CGS and Bi-CGSTAB, as well. The main idea is that in these methods the approximate solution is updated as

$$x^{(i+1)} = x^{(i)} + \alpha_i p^i,$$

and the corresponding residual is updated as

$$r^{(i+1)} = r^{(i)} - \alpha_i A p^i.$$

This means that $AP_i = W_i R_{i+1}$, with W_i a lower bidiagonal matrix. The $x^{(i)}$ are combinations of the p^i, so that we can try to find the combination $P_i y_i$ for which $\|b - AP_i y_i\|_2$ is minimal. If we insert the expression for AP_i, and ignore the fact that the $r^{(i)}$ are not orthogonal, then we can minimize the norm of the residual in a quasi-minimum least squares sense, similar to QMR.

CGS, GCGS, and TFQMR

For Bi-CG, as for the other methods, we have that $r^{(j)} \in \mathcal{K}^{j+1}(A; r^{(0)})$, and hence it can be written formally as $r^{(j)} = P_j(A) r^{(0)}$, with P_j a polynomial of degree j. Since the basis vectors $\widehat{r}^{(j)}$, the scaled vectors w^j, for the dual subspace are generated by the same recurrence relation, we have that $\widehat{r}^{(j)} = P_j(A^*) \widehat{r}^{(0)}$.
The iteration coefficients for the recurrence relations follow from bi-orthogonality conditions, and

may look like:

$$\begin{aligned}\alpha_j &= (\widehat{r}^{(j)}, Ar_j) \\ &= (P_j(A^*)\widehat{r}^{(0)}, AP_j(A)r^{(0)}) \\ &= (\widehat{r}^{(0)}, AP_j^2(A)r^{(0)}).\end{aligned}$$

Sonneveld [291] observed that we could also construct vectors $\widetilde{r}^{(j)}$ with the property that they can be formally written as $\widetilde{r}^{(j)} = P_j^2(A)r^{(0)}$, and that this can be done with short term recurrences without ever computing explicitly the Bi-CG vectors $r^{(j)}$ and $\widehat{r}^{(j)}$. The Bi-CG iteration coefficients can then be recovered using the latter form of the innerproduct.
This has two advantages:

1. Since we do not have to form the vectors $\widehat{r}^{(j)}$, we can avoid to work with the matrix A^*. This is an advantage for those applications where A^* is not readily available.

2. An even more appealing thought is that, in case of Bi-CG convergence, the formal operator $P_j(A)$ describes how $r^{(0)}$ is reduced to $r^{(j)}$, and naively thinking we might expect that $P_j(A)^2$ would lead to a double reduction. Surprisingly this is indeed often the case. This is certainly a surprise when we come to think of it: the operator $P_j(A)$ depends very much on the starting vector $r^{(0)}$, and delivers a vector $r^{(j)}$ that is optimal, in the sense that it is orthogonal to some well-defined subspace. There is no reason why the vector $\widetilde{r}^{(j)} \equiv P_j^2(A)r^{(0)}$ should be orthogonal to the subspace $\mathcal{K}^{2j}(A^*; \widehat{r}^{(0)})$, or in other words, the optimal polynomial for the starting vector $P_j(A)r^{(0)}$ may not be expected to be $P_j(A)$ again.

The resulting method has become well-known under the name CGS, which means Conjugate Gradients Squared. The adjective 'Squared' is obvious from the polynomial expressions, but it would have been more correct to use the name Bi-CGS. CGS can be represented by the algorithm given in Fig. 8.10, in which K defines a suitable preconditioner).

In this scheme the $\widetilde{r}^{(i)}$ have been represented by $r^{(i)}$.

CGS has become a rather popular method over the years, because of its fast convergence, compared with Bi-CG, and also because of the modest computational costs per iteration step, as compared with GMRES. Comparisons of these three methods have been appeared in many studies (see, e.g., [263, 44, 260, 247]).
However, CGS usually shows a very irregular convergence behaviour, which can be understood from the observation that all kinds of irregular convergence effects in Bi-CG are squared. This behaviour can even lead to cancellation and a spoiled solution [312], and one should always check the value of $||b - Ax^{(j)}||_2$ for the final approximation $x^{(j)}$. In [288] it is shown how CGS can be implemented, with some small additional overhead, so that excessive cancellation is avoided.

$x^{(0)}$ is an initial guess; $r(0) = b - Ax(0)$;
$\tilde{r}^{(0)}$ is an arbitrary vector, such that
$(\tilde{r}^{(0)})^* r^{(0)} \neq 0$,
e.g., $\tilde{r}^{(0)} = r^{(0)}; \rho_0 = (\tilde{r}^{(0)})^* r^{(0)}$;
$\beta_0 = \rho_0; p^{-1} = q^0 = 0$;
for $i = 0, 1, 2, ...$
    $u^i = r^{(i)} + \beta_i q^i$;
    $p^i = u^i + \beta_i(q^i + \beta_i p^{i-1})$;
    solve $\hat{p}$ from $K\hat{p} = p^i$;
    $\hat{v} = A\hat{p}$;
    $\alpha_i = \frac{\rho_i}{(\tilde{r}^{(0)})^* \hat{v}}$;
    $q^{i+1} = u^i - \alpha_i \hat{v}$;
    solve $\hat{u}$ from $K\hat{u} = u^i + q^{i+1}$
    $x^{(i+1)} = x^{(i)} + \alpha_i \hat{u}$;
    if $x^{(i+1)}$ is accurate enough then quit;
    $r^{(i+1)} = r^{(i)} - \alpha_i A\hat{u}$;
    $\rho_{i+1} = (\tilde{r}^{(0)})^* r^{(i+1)}$;
    if $\rho_{i+1} = 0$ then method fails to converge !;
    $\beta_{i+1} = \frac{\rho_{i+1}}{\rho_i}$;
end

Figure 8.10: The CGS algorithm

When the matrix A is symmetric positive definite, Bi-CG produces the same $x^{(i)}$ and $r^{(i)}$ as CG, and hence CGS does not break down then.

In the case of nonsymmetry of A, a thorough convergence analysis has not yet been given. Surprisingly enough the method, applied to a suitably preconditioned linear system, does often not really suffer from nonsymmetry.

Note that the computational costs per iteration are about the same for Bi-G and CGS, but CGS has the advantage that only the matrix A itself (and its preconditioner) is involved and not its transpose. This avoids the necessity for complicated data-structures to ensure that matrix vector products Ax and A^*y can be evaluated equally fast.

For practical implementations in a parallel environment, it might be a slight disadvantage that Ap^i and $A(u^i + q^{i+1})$ cannot be computed in parallel, whereas both the matrix-vector products can be done in parallel in the Bi-G method.

Freund [161] suggested a squared variant of QMR, which was called TFQMR. His experiments show that TFQMR is not necessarily faster than CGS, but it has certainly a much smoother convergence behavior.

Bi-CGSTAB

Bi-CGSTAB [312] is based on the observation that the Bi-CG vector $r^{(i)}$ is orthogonal to the entire subspace $\mathcal{K}^i(A^*, w^1)$, so that we can construct $r^{(i)}$ by requiring it to be orthogonal to other basis vectors for $\mathcal{K}^i(A^*, w^1)$. As a result, we can, instead of squaring the Bi-CG polynomial, construct iteration methods, by which $x^{(i)}$ are generated so that $r^{(i)} = \tilde{P}_i(A)P_i(A)r^{(0)}$ with other i^{th} degree polynomials $\tilde{P}$. An obvious possibility is to take for $\tilde{P}_j$ a polynomial of the form

$$Q_i(x) = (1 - \omega_1 x)(1 - \omega_2 x)...(1 - \omega_i x), \tag{8.14}$$

and to select suitable constants ω_j. This expression leads to an almost trivial recurrence relation for the Q_i.
In Bi-CGSTAB ω_j in the j^{th} iteration step is chosen as to minimize $r^{(j)}$, with respect to ω_j, for residuals that can be written as $r^{(j)} = Q_j(A)P_j(A)r^{(0)}$.
The preconditioned Bi-CGSTAB algorithm for solving the linear system $Ax = b$, with preconditioning K is given in Fig. 8.11.

The matrix K in this scheme represents the preconditioning matrix and the way of preconditioning [312]. The above scheme carries out the Bi-CGSTAB procedure for the explicitly postconditioned linear system

$$AK^{-1}y = b,$$

but the vectors $y^{(i)}$ and the residual have been backtransformed to the vectors $x^{(i)}$ and $r^{(i)}$ corresponding to the original system $Ax = b$. Compared with CGS, two extra innerproducts need to be calculated.
In exact arithmetic, the α_j and β_j have the same values as those generated by Bi-CG and CGS.
Bi-CGSTAB can be viewed as the product of Bi-CG and GMRES(1). Of course, other product methods can be formulated as well. Gutknecht [191] has proposed BiCGSTAB2, which is constructed as the product of Bi-CG and GMRES(2).

Bi-CGSTAB(ℓ) and variants

The residual $r^{(k)} = b - Ax^{(k)}$ in Bi-CG, when applied to $Ax = b$ with start $x^{(0)}$ can be written formally as $P_k(A)r^{(0)}$, where P_k is a k-degree polynomial. These residuals are constructed with one operation with A and one with A^* per iteration step. It was pointed out in [291] that with about the same amount of computational effort one can construct residuals of the form $\widetilde{r}^{(k)} = P_k^2(A)r^{(0)}$, which is the basis for the CGS method. This can be achieved without any operation with A^*. The idea behind the improved efficiency of CGS is that if $P_k(A)$ is viewed as a reduction operator in Bi-CG, then one may hope that the square of this operator will be a twice as powerful reduction operator. Although this is not always observed in practice, one typically has that CGS converges faster than Bi-CG. This, together with the absence of operations with A^*, explains the success of the CGS method. A drawback of CGS is that its convergence behavior can look quite irregular, that is the norms of the residuals converge quite irregularly, and it may easily happen that $\|r^{(k+1)}\|_2$ is much larger than $\|r^{(k)}\|_2$ for certain k (for an explanation of this see [310]).

In [312] it was shown that by a similar approach as for CGS, one can construct methods for which $r^{(k)}$ can be interpreted as $r^{(k)} = P_k(A)Q_k(A)r^{(0)}$, in which P_k is the polynomial associated with Bi-CG and Q_k can be selected free under the condition that $Q_k(0) = 1$. In [312] it was suggested to construct Q_k as the product of k linear factors $I - \omega_j A$, where ω_j was taken to minimize locally a residual. This approach leads to the Bi-CGSTAB method. Because of the local minimization, Bi-CGSTAB displays a much smoother convergence behavior than CGS, and more surprisingly, it often also converges (slightly) faster. A weak point in Bi-CGSTAB is that we get breakdown if an ω_j is equal to zero. One may equally expect negative effects when ω_j is small. In fact, BiCGSTAB can be viewed as the combined effect of Bi-CG and GCR(1), or GMRES(1), steps. As soon as the GCR(1) part of the algorithm (nearly) stagnates, then the Bi-CG part in the next iteration step cannot (or only poorly) be constructed. For an analysis, as well as for suggestions to improve the situation, see [283].

Another dubious aspect of Bi-CGSTAB is that the factor Q_k has only real roots by construction. It is well-known that optimal reduction polynomials for matrices with complex eigenvalues may have complex roots as well. If, for instance, the matrix A is real skew-symmetric, then GCR(1) stagnates forever, whereas a method like GCR(2) (or GMRES(2)), in which we minimize over two combined successive search directions, may lead to convergence, and this is mainly due to the fact that then complex eigenvalue components in the error can be effectively reduced.

This point of view was taken in [191] for the construction of the BiCGSTAB2 method. In the odd-numbered iteration steps the Q-polynomial is expanded by a linear factor, as in Bi-CGSTAB, but in the even-numbered steps this linear factor is discarded, and the Q-polynomial from the previous even-numbered step is expanded by a quadratic $I - \alpha_k A - \beta_k A^2$. For this construction the information from the odd-numbered step is required. It was anticipated that the introduction of quadratic factors in Q might help to improve convergence for systems with complex eigenvalues, and, indeed, some improvement was observed in practical situations (see also [259]).

However, our presentation suggests a possible weakness in the construction of BiCGSTAB2, namely in the odd-numbered steps the same problems may occur as in Bi-CGSTAB. Since the even-numbered steps rely on the results of the odd-numbered steps, this may equally lead to unnecessary break-downs or poor convergence. In [286] another, and even simpler approach was taken to arrive at the desired even-numbered steps, without the necessity of the construction of the intermediate Bi-CGSTAB-type step in the odd-numbered steps. Hence, in this approach the polynomial Q is constructed straight-away as a product of quadratic factors, without ever constructing a linear factor. As a result the new method Bi-CGSTAB(2) leads only to significant residuals in the even-numbered steps and the odd-numbered steps do not lead necessarily to useful approximations.
In fact, it is shown in [286] that the polynomial Q can also be constructed as the product of ℓ-degree factors, without the construction of the intermediate lower degree factors. The main idea is that ℓ successive Bi-CG steps are carried out, where for the sake of an A^*-free construction the already available part of Q is expanded by simple powers of A. This means that after the Bi-CG part of the algorithm vectors from the Krylov subspace $s, As, A^2s, ..., A^\ell s$, with $s = P_k(A)Q_{k-\ell}(A)r^{(0)}$ are available, and it is then relatively easy to minimize the residual over that particular Krylov subspace. There are variants of this approach in which more stable bases for the Krylov subspaces are generated [285], but for low values of ℓ a standard basis satisfies, together with a minimum norm solution obtained through solving the associated normal equations (which requires the solution of an ℓ by ℓ system). In most cases Bi-CGSTAB(2) will already give nice results for problems where Bi-CGSTAB or BiCGSTAB2 may fail. Note, however, that, in exact arithmetic, if no breakdown situation occurs, BiCGSTAB2 would produce exactly the same results as Bi-CGSTAB(2) at the even-numbered steps. Bi-CGSTAB(2) can be represented as in Fig. 8.12. For more general Bi-CGSTAB(ℓ) schemes see [286, 285].

Another advantage of Bi-CGSTAB(2) over BiCGSTAB2 is in its efficiency. The Bi-CGSTAB(2) algorithm requires 14 vector updates, 9 innerproducts and 4 matrix vector products per full cycle. This has to be compared with a combined odd-numbered and even-numbered step in BiCGSTAB2, which requires 22 vector updates, 11 innerproducts, and 4 matrix vector products, and with two steps of Bi-CGSTAB which require 12 vector updates, 8 innerproducts, and 4 matrix vector products. The numbers for BiCGSTAB2 are based on an implementation described in [259].
Also with respect to memory requirements, Bi-CGSTAB(2) takes an intermediate position: it requires 2 n-vectors more than Bi-CGSTAB and 2 n-vectors less than BiCGSTAB2.

For distributed memory machines the innerproducts may cause communication overhead problems (see, e.g., [67]). We note that the Bi-CG steps are very similar to conjugate gradient iteration steps, so that we may consider all kind of tricks that have been suggested to reduce the number of synchronization points caused by the 4 innerproducts in the Bi-CG parts. For an overview of these approaches see [33]. If on a specific computer it is possible to overlap communication with communication, then the Bi-CG parts can be rescheduled as to create overlap possibillities:

1. The computation of ρ_1 in the even Bi-CG step may be done just before the update of u at the end of the GCR part.
2. The update of $x^{(i+2)}$ may be delayed until after the computation of γ in the even Bi-CG step.
3. The computation of ρ_1 for the odd Bi-CG step can be done just before the update for x at the end of the even Bi-CG step.
4. The computation of γ in the odd Bi-CG step has already overlap possibillities with the update for u.

For the GCR(2) part we note that the 5 innerproducts can be taken together, in order to reduce start-up times for their global assembling. This gives the method Bi-CGSTAB(2) a (slight) advantage over Bi-CGSTAB. Furthermore, we note that the updates in the GCR(2) may lead to more efficient code than for BiCGSTAB, since some of them can be combined.

8.0.6 Other issues

We have restricted ourselves to a description of the basic algorithms for various iterative methods, but of course there is much more to be said. For many linear problems the described methods can be applied in a straightforward manner and it is always easy to check the eventual result x_i by computing the residual $b - Ax_i$. In fact, this is the first point of concern. In many of the discussed iteration methods a vector r_i is computed by an updating process (see for instance Figure 8.1), and in exact arithmetic the relation $r_i = b - Ax_i$ holds. However, in finite precision floating point arithmetic there may be a huge difference between the *updated residual* r_i and the *true residual* $b - Ax_i$. In [288] it is shown that

$$||b - Ax_i - r_i||_2 \sim \mathbf{u} \max_{0\leq j\leq i} ||r_j||_2,$$

in which $\mathbf{u}$ denotes the relative machine precision.
This means that in algorithms in which r_i is computed by an updating process, the results should be suspect as soon as r_i is at the level of $\mathbf{u}$ times the norm of the largest intermediate updated residual.
In [288] an easy fix for this problem is suggested. The idea is to collect a number of updates for x_i and r_i for a number of successive steps and to add these collected updates to the last updated x_i and r_i at a point where the new r_i is smaller in norm than the last updated r_i. This can be implemented by only a few lines of additional code. The remarkable result is that not only the final result has more significance, but often we observe that the iteration process converges faster with these *reliable* updated vectors. Better and faster; there is no argument to avoid including this strategy in any actual code. For details see [288].

Some iteration processes may produce residual vectors that seem to converge in a rather irregular manner: successive vectors vary in norm by orders of magnitude. As we have argued above, this should be avoided since large intermediate updated residual vectors have a disastrous effect on the eventual approximation for the solution. With reliable updating we can diminish this effect. Another way of creating a more nicely converging process is known as residual smoothing, but here we have to be careful. The easiest way to obtain a smoothly converging iteration process is to introduce an auxiliary vector $\widetilde{r}_i$ and to take for this vector the r_j that has smallest norm for $0 \leq j \leq i$. For sure this leads to a very smooth convergence curve but it is obvious that this has only psychological effects. Zhou and Walker [329] consider residual smoothing processes in which a minimization process is carried out over a small number of successive residual vectors. This leads effectively to new residual vectors and these may have a smaller norm than the regular ones. In this sense residual smoothing may indeed lead to a better process. However, if some of the residual vectors of the underlying process (that is the process for which smoothing is applied) are large then we still have the accuracy spoiling effects that we discussed before, irrespective of the smoothness of the corrected residual vectors. Hence, also in this case the user should be very careful with the interpretation of the shown residuals and *reliable updating* strategies, as those discussed in [288], are still a wise option. In any case, the user should check the *true residual* $b - Ax_i$.

Another problem that the user of Bi-CG based methods, like Bi-Cg itself, CGS, TFQMR, and Bi-CGSTAB, will encounter is the possible breakdown of the three-term recursions by which the bi-orthogonal bases for Bi-CG are constructed. Exact breakdown will very seldomly occur, but a near breakdown is almost as problematic because of the numerical instabilities that affect the approximated solution. A way to circumvent the (near) breakdown is to replace the three-term recurrence relation temporarily by block three-term relation for a number of vectors simultaneously. That is, instead of expanding the Krylov subspace with one new vector, the subspace is expanded by k vectors. These k vectors increase the dimension of the subspace by k, and the size of k is chosen in such a way that the block of k vectors can be made bi-orthogonal in a block-wise sense. Such an expansion is referred to as a *look-ahead step* of length k. This technique was suggested first in [257] for length 2, and it was further refined for arbitrary lengths in connection with QMR in [160]. In [41] similar techniques were used to cure near-breakdowns in CGS. The disadvantage of all these techniques is that we have to store more vectors, and there is an increased computational for the block-orthogonalization, so that there is a limited class of problems for which this technique is still worthwhile in favor of, for instance, GMRES. An obvious alternative is to restart the Bi-CG based methods as soon as a (near) breakdown situation is detected. This happens in Bi-CG, see Figure 8.9, when the innerproduct of $\widetilde{p}^i$ and q^i is small compared to the product of the norms of these vectors. Another simple manner to detect (near) breakdown is the occurrence of a residual r_i with large norm. The disadvantage of this approach is that it may spoil the overal convergence behavior of the method, since by restarting we replace the entire Krylov subspace by one single vector.

Until now we have said nothing on how to terminate the iteration process. Many users check the norm of the residual vector and stop the process as soon as this norm arrives below some preset tolerance. Especially when the matrix is ill-conditioned a small residual this does not say much about the accuracy in the solution. Another problem is that by multiplying the given system by a constant, we can get any residual as small in norm as we want, without any effect on the approximate solution. Hence, it is necessary to relate the norm of the residual to the initial right-hand side or to the norm of the matrix. Various ways of doing this, with different interpretations for the quality of the solution, have been discussed in [33]: Chapter 4.2.

8.0.7 How to test iterative methods

The convergence behavior of iterative methods depends on many parameters, such as the distribution of the eigenvalues, components of the starting vector in eigenvector directions and even on angles between eigenvectors. Often these parameters are unknown but for relevant situations important properties may be known, for instance that all eigenvalues are positive real, or in the right-half plane, or that the contribution of slowly varying eigenvectors to the solution is dominant. With respect to efficiency it is also very important to exploit structural properties of the given system, such as sparseness, or special sparsity patterns that may help to improve performance on modern computer architectures.

All these different aspects make it very difficult to select relevant test problems. In this section we will give some hints on the selection of test problems together with remarks on what we may learn from the numerical results obtained from such problems.

A good starting point for investigating appropriate iterative methods is to consider the class of problems that one wishes to solve. The first problem that arises is how big we should choose the size of our testing problems. Most of us want to do their experiments in MATLAB on a working station and we want to have the results quickly. However, selecting a small problem size may lead to very misleading conclusions. A number of 70 iterations may seem quite modest, but if observed for a system of order 100, then it may be alarming. If the 70 iterations correspond more or less to the eigenvalue distribution, for instance, and if the larger systems have similar eigenvalue distributions then the iteration number may be representative. Small-sized systems often arise from coarse grids, but in advection-diffusion problems the convection term is much more dominant in the resulting linear system for coarse grids; the spectral properties may change completely for finer grids.

While possibly appropriate for testing purposes, small-sized systems are not the kind of systems to be solved by iterative methods in a production environment, unless very special circumstances are present (such as exceptionally good starting vectors, and very modest accuracy requirements).

Usually, small-sized systems can be solved much better by direct methods.
The effectiveness of preconditioners may depend critically on the mesh-size. Again, for advection-diffusion problems the resulting matrix may be an M-matrix for small gridsizes, but not for coarse grids. Incomplete LU-factorizations exist for M-matrices, but not necessarily for other types of matrices.
Therefore, there is not much that one can say about the selection of good testing examples other than that this is a very tricky bussiness. Nevertheless, we will discuss strategies for testing and selecting iterative methods. In our experience, it is often helpful to inspect the Ritz values that can be computed from iteration parameters. All Krylov subspace methods discussed in this book generate, either implicitly or explicitly, a so-called reduced matrix, formed by certain iteration constants. This is most clearly the case for GMRES, where the matrix $H_{i+1,i}$ takes this role. The eigenvalues of the leading i by i part of this matrix are called the *Ritz values*. Usually these Ritz values, converge, for increasing values of i, to eigenvalues of the (preconditioned) matrix, and valuable information can be deduced from their behavior. If for some value of i one or more Ritz values have converged to eigenvalues then the iteration proceeds as for a system in which the corresponding eigenvectors are not present [301, 314]; or in other words, at that stage of the iteration the components of the solution in the converged eigendirections have been discovered by the iteration method. If these eigendirections components constitute the major part of the solution then this indicates effectiveness of the method. If there are good reasons to believe that for the larger systems the solution is also composed from eigendirections that converge in early stages of the iteration process, then this is a hopeful sign. The question is whether this is a natural situation to be expected. As a rule of thumb, the exterior eigenvalues of the (preconditioned) matrix of the system to be solved are first well-approximated by Ritz values. For grid oriented problems, for instance discretized elliptic partial differential equations, some of the exterior eigenvalues correspond to eigenvectors that vary only slowly over groups of neigbouring gridcells. If the desired solution is also expected to display such behavior, then this may be an indication for success of the iterative solver. If, on the other hand, the solution is expected to behave quite irregularly with respect to the imposed grid then one may not expect much effectiveness of the iterative solver. By the way, one might well question the relevance of solving grid oriented problems for which solutions show irregular behavior for small steps across the grid.
Situations where the solution may have large variations are not so uncommon for systems associated with (electrical) networks, for instance a network representing an electric circuit. Depending on the inputvalues (the voltages imposed on the network), parts of the network my be active or not and this may involve all sorts of eigenvectors, also those associated with interior eigenvalues. For such types of systems the convergence behavior of any iterative method may depend critically on the right-hand side, and this behavior can only be improved by changing the preconditioner. Unfortunately, the matrices involved often do not have special known properties that can be exploited for the construction of effective preconditioners. For electrical networks there is some hope to construct effective preconditioners from nearby 'easier-to-solve' systems that are related to networks with

fewer connections and that have more or less similar behavior. The construction of such nearby networks requires quite some physical insight; a good example of this is the so-called *capacitance method*, described by Nelis et al [249].

So far we have not said much about the selection of the iteration method itself, or the preconditioner. For unsymmetric systems, it may be good to start the experiments with GMRES(m) with a value of m as high as possible, since this leads to valuable information on the Ritz values. If the origin is in the interior of the convex hull of the Ritz values for increasing values of the iteration count, then we can pretty much be sure that the matrix of the given system is indefinite indeed, and then GMRES(m) may be the method to continue with. A rule of thumb is to take m so large that the convex hull of the Ritz values does not change much by increasing m. The only hope to imrove the effectiveness of the iteration method is now to improve the preconditioner, so that the matrix becomes 'less indefinite', that is that the origin becomes located more to the exterior of the convex hull, or even better, outside the convex hull. Indefinite systems are almost always difficult to solve for iterative methods.
If the convex hull of Ritz values is in one half plane of the complex plane, and if the imaginary parts of the Ritz values are not large relative to the real parts, and if the overhead costs of GMRES(m) become prohibitive, then time has come to consider other methods such as QMR and Bi-CGSTAB. In our experience Bi-CGSTAB seems to be faster if the convex hull is contained in an ellipse which is located closely around the real axis. If there are Ritz values converging towards eigenpairs with relatively large imaginary part then Bi-CGSTAB may suffer from convergence problems and QMR should be considered. It may also be worthwhile to consider Bi-CGSTAB(ℓ) in such cases, since they converge potentially faster if the complex parts of the spectrum can be captured reasonably well by the GMRES(ℓ) part of these methods.

A completely different category of persons who want to have good testing examples, is formed by those who are primarily interested in the analysis and design of such methods. In this case one may want to have control over parameters that may or may not influence the convergence behavior and structural properties of the given systems may be less of concern (unless one wants to study preconditioning or performance aspects). Since most of the iteration methods use the matrix only in matrix vector products, they do not take advantage of the structure of the matrix. Therefore, it may be appropriate to carry out convergence studies with diagonal matrices, silly as it may seem from the problem-solving point of view. The iteration methods display similar convergence behavior for diagonal matrices as for matrices that are orthogonally equivalent to diagonal matrices (assuming that the right-hand side is transformed orthogonally as well). There may be a difference though with respect to rounding errors: in the matrix vector products the rounding errors made with each diagonal element can be attributed to the corresponding unit vector eigendirection, whereas for non-diagonal matrices rounding errors, even when the involved vector is an eigenvector, are distributed

over more eigendirections. This is particularly the case for MINRES, which is much more sensitive to rounding errors for ill-conditioned non-diagonal matrices as for orthogonally equivalent diagonal matrices [287].
For the testing of methods that are designed for unsymmetric matrices, diagonal matrices are not very representative, and one could then test with blockdiagonal matrices, of which the 2 by 2 blocks along the diagonal have known complex conjugate eigenpairs. The eigenvectors of such systems are blockwise orthogonal, which is a special property that is often not encountered in actual problems. Therefore it may be wise to transform such systems to spectrical equivalent systems with a non-unitary matrix. Widely different convergence behaviors can be observed for different transformations applied to the same (block-)diagonal matrix, see for more details on this [180].
Finally, one can test iterative methods and preconditioners on systems contained in public test-sets, such as the Harwell-Boeing test-set. This gives some idea on how robust the method is, but it is questionable whether this tells us much about the usefulness of the investigated method for particular classes of problems. Testing iterative methods on general test-sets often seems to express a quest for the 'ultimate robust general iterative solver'. Because the problems to be solved live in high dimensional spaces, there is no hope that one single method will detect for all sorts of systems where the solution is by inspecting only low-dimensional subspaces with modest computational effort (compared with direct solution methods).

When it come to the testing of the effectiveness of preconditioners, then little can be said in general. So far, we have not seen preconditioning techniques that work well for all sorts of matrices, i.e, nor is much theoretically known about he effectiveness of preconditioners. It appears that any preconditioning technique is only effective for a restricted class of matrices, and, hence, testing results for preconditioners over an arbitrary test set are not that useful, unless the experiments contribute to our insight on the possible effects of preconditioning for specific matrices. In general it is wise to do experiments with preconditioners for linear systems that are representative for the problems that one wants to solve.

8.0.8 Vector and Parallel Aspects

The basic time-consuming computational elements of the methods presented are as follows:

1. vector updates (SAXPY and generalized SAXPY).

2. inner products (SDOT).

3. matrix vector products like Ap (and A^*p).

4. determination of the solution q of systems $Kq = y$, for given y (the preconditioning part of the algorithm), and for systems with K^* (Bi-CG and LSQR).

Usually, the SAXPY and SDOT stages can—especially for large systems—be computed efficiently on both vector and parallel computers (or mixed). Also, in many situations the matrix vector products can be computed with high computational speeds as well, in particular when the matrices have a regular structure.

A common characteristic of these operations is that the amount of data transfer is relatively high with respect to the number of floating-point operations involved. For SAXPY, we observe 2 flops per 3 data transfer operations; for SDOT, the ratio is 2:2; and for Ap, when A has the regular five-diagonal structure and has been stored diagonally, the ratio is 9:11. (A more elaborate analysis is given in [306]).

For most vector computers the optimal ratio is 2:1. In dense matrix computations such a ratio can often be achieved by rearranging the computations (see, e.g., Dongarra and Eisenstat [94]).
In case of a cache memory, the bandwidth between cache and main memory is a limiting factor; in such a situation one must perform many flops per data transfer between cache and main memory. These aspects have for dense matrices led to the design of the so-called Level 3 BLAS (see Chapter **??**).

In the sparse matrix situation, things are much less rosy. For example, a SAXPY where the lengths of the vectors exceed the size of the cache will be executed with a performance significantly below the maximum speed of the computer. This can be illustrated by results obtained for the IBM 3090-VF (as well as any other vector machine with a memory hierarchy). Its clock cycle (18.5 ns) suggests a maximum speed of about 108 Mflops. Since there is only one vector load/store path (as for CRAY-1, CONVEX, Alliant, NEC,...), the performance for DAXPY, when the vectors are available in fastest memory (cache), could be 36 Mflops at most. Because of stripmining effects (see Section **??**) the actual performance is even 20% less: about 29 Mflops (a reduction of 20% from stripmining is also common for other vector computers). However, when the vectors are not already available in cache (which is more or less the standard situation in sparse matrix computations), the performance of DAXPY drops even further, to a modest 19.5 Mflops. Similar performance-reducing effects for SAXPY or DAXPY, as well as for the other computational elements, are observed on most vector computers, and the negative effects in the reported results should in no way be seen as characteristic or special for the IBM 3090-VF.

Occasionally, some of the basic operations in a given iterative method can be grouped together, and then the memory hierarchy can be dealt with more effectively by rearranging the computations. As an example, see our discussions on variants of CG (Section 8.0.5) and the Bi-CGSTAB(2) method (Section 8.0.5).

The main bottleneck to achieving reasonable performance is the preconditioning part of the algorithms (or the solution of $Kq = y$), since the most popular choices for K do not immediately lead to vectorizable or parallel code. Therefore we shall discuss this problem in separate sections.

We shall now discuss the implementation of the computational elements in some more detail.

Sparse Matrix-Vector Multiplication
First we shall consider sparse matrix-vector multiplication for the case that A has a very regular structure. Usually the sparse linear systems that arise from finite difference or finite element discretizations of partial differential equations have a structure in which the nonzero elements are located only in certain (off-)diagonals of the matrix. As an example we discuss the matrix that results from standard five-point finite-difference approximation of a second-order elliptic PDE over a rectangular grid imposed over a rectangular region in two dimensions:

$$A = \left(\begin{array}{cccccccccccccccccccc}
* & * & & & & & & * & & & & & & & & & & & & \\
* & * & * & & & & & & * & & & & & & & & & & & \\
& * & * & * & & & & & & * & & & & & & & & & & \\
& & * & * & * & & & & & & * & & & & & & & & & \\
& & & * & * & * & & & & & & * & & & & & & & & \\
& & & & * & * & * & & & & & & * & & & & & & & \\
& & & & & * & * & & & & & & & * & & & & & & \\
* & & & & & & & * & * & & & & & & * & & & & & \\
& * & & & & & & * & * & * & & & & & & * & & & & \\
& & * & & & & & & * & * & * & & & & & & * & & & \\
& & & & & & & . & . & . & . & . & . & & & & & & & \\
& & & & & & & & & & . & . & . & . & . & . & & & & \\
& & & & & & & & & & * & & & & & & & * & * & * \\
& & & & & & & & & & & * & & & & & & & * & *
\end{array} \right)$$

For the derivation of such matrices and their properties, see [318]. When the nonzero entries of the matrix form such a regular systematic pattern, one can choose between storing the nonzeros in a two-dimensional array, in row, column, or diagonal fashion. Often the last way is preferred [230], because it leads to larger vector lengths for vector register computers and it avoids memory strides. For the problem sketched above, we have given the performances for many supercomputers in the tables in Section **??** (loop number 14). We see from these tables that this operation leads to a performance usually in the range of 50-100% of the peak performance of the machine.

For some vector-register machines better performance can be obtained when using assembler language. For instance, when we compute $y = Ax$, with A as in the figure above, then we need, amongst others, the values x_{i-1}, x_i, and x_{i+1} for the computation of the element y_i. On Cray vector computers there exists a vector shift instruction that can create the vector streams x_{i+1} and x_{i-1} from x_i, by shifting the contents of a vector register one position forward or backward. This shift operation can be chained with other vector operations, and using these techniques, Jordan [214] has shown that this approach can lead to impressive improvements for the CRAY-1.

For computers with a cache memory (and also for some vector register machines) it is sometimes better to exploit the block structure of the matrix. If, in the above situation, the nonzero diagonals of the blocks are stored consecutively, excessive cache refreshment is avoided. However, this depends on the type of computer and the compiler. On most vector register computers the computation of a segment of y is completed before the computation of the next segment of y starts. This implies that the required elements of x for the next segment of y are usually still available in cache.

The performance can be further improved by combining the computation of $y = Ax$ with other suitable parts of the particular method, as, e.g., in CG where $q^i = Ap^i$ can be combined with the computation of the innerproduct $((q^i)^* p^i)$. Often these situations can be fully exploited in assembler language only, but sometimes the performance may also be improved in standard Fortran.

As an example we compare two possibilities for the Fortran code for the IBM 3090-VF and we report on some observed performances. We will assume that A is real symmetric and that the elements of the upper triangular part of A have been stored by diagonals in 3 arrays: `A(⋆,1)`, `A(⋆,2)`, and `A(⋆,3)`. The first possibility is to complete first the computation of $q^i = Ap^i$ in a diagonalwise fashion by code that has as a typical statement

```
      DO 10 J = N1,N2
         Q(J) = A(J-M,3)*P(J-M)+A(J-1,2)*P(J-1)+A(J,1)*P(J)+
     1          A(J,2)*P(J+1)+A(J,3)*P(J+M)
   10 CONTINUE
```

After completion of the matrix vector product we compute the innerproduct as

```
      PQ = DDOT(N,P,1,Q,1)
```

For matrices of order $N = 40000$ and bandwidth $M = 200$ this leads to a computational speed of 27 Mflops for both elements together. Note that the performance for the innerproduct is negatively influenced by the fact that the elements of `P` and `Q` have to be transported all the way through cache again for the computation of `PQ`.

The next possibility is to combine the two operations. This leads to a code with a typical statement like

```
      DO 10 J=N1,N2
         Q(J)=A(J-M,3)*P(J-M)+A(J-1,2)*P(J-1)+A(J,1)*P(J)+
     1        A(J,2)*P(J+1)+A(J,3)*P(J+M)
         SUM=SUM+Q(J)*P(J)
   10 CONTINUE
```

The computational speed observed for this way of computing was 33 Mflops. Since the performance of the Fortran coded innerproduct is significantly less than for the assembler coded DDOT, we believe that, with a careful assembler implementation of the above DO-loop, the performance can be further increased by exploiting the contents of the vector registers and by combining these operations with the computation of p^i (in CG).

Next we consider the case that the nonzero entries are more or less scattered over the matrix A. The storage scheme to be chosen then will depend on the application at hand, and on the computer type. A survey of suitable storage schemes for sparse matrices is given in by Duff, Erisman, and Reid[108], Duff[107], and Reid [264].

As an example, suppose that the matrix has at most five nonzero entries in each row, and that one is interested in forming the matrix-vector product. Then the nonzero elements could be stored in an array `AA(N,5)`, whereas the column indices of these elements are supplied in the array `IND(N,5)`. The computation of the required product then reads as follows:

```
      DO 20 J=1,5
         DO 10 I=1,N
            Y(I)=Y(I)+AA(I,J)*X(IND(I,J))
   10    CONTINUE
   20 CONTINUE
```

As a result of the indirect addressing, the performance is much worse than for the regularly structured situation. Dongarra and Duff[104] report a loss of at least a factor of 2 in performance for many common linear algebra constructions as a result of the use of indirect addressing. Consequently, indirect addressing should be avoided as much as possible.

Also for more general sparse matrix structures the negative cache effects can be reduced by selecting an appropriate storage scheme. Radicati di Brozolo and Vitaletti [262] compare the performance of the matrix-vector product on the IBM 3090-VF for two different storage schemes. They show that a compressed matrix scheme (as in ESSL [149], ITPACK [218] and ELLPACK [265]) may reduce the CPU time by a factor of over 2 as compared with the more familiar rowwise storage scheme (as in ESSL [149] and also ITPACK [181]).

$x^{(0)}$ is an initial guess; $r^{(0)} = b - Ax(0)$;
$\bar{r}^{(0)}$ $(= w^1)$ is an arbitrary vector, such that
  $(\bar{r}^{(0)})^* r^{(0)} \neq 0$, e.g., $\bar{r}^{(0)} = r^{(0)}$;
$\rho_{-1} = \alpha_{-1} = \omega_{-1} = 1$;
$v^{-1} = p^{-1} = 0$;
for $i = 0, 1, 2, ...$
  $\rho_i = (\bar{r}^{(0)})^* r^{(i)}; \beta_{i-1} = (\rho_i/\rho_{i-1})(\alpha_{i-1}/\omega_{i-1})$;
  $p^i = r^{(i)} + \beta_{i-1}(p^{i-1} - \omega_{i-1} v^{i-1})$;
  Solve $\hat{p}$ from $K\hat{p} = p^i$;
  $v^i = A\hat{p}$;
  $\alpha_i = \rho_i / (\bar{r}^{(0)})^* v^i$;
  $s = r^{(i)} - \alpha_i v^i$;
  if $\|s\|$ small enough then
    $x^{(i+1)} = x^{(i)} + \alpha_i \hat{p}$; quit;
  Solve z from $Kz = s$;
  $t = Az$;
  $\omega_i = s^* t / t^* t$;
  $x^{(i+1)} = x^{(i)} + \alpha_i \hat{p} + \omega_i z$;
  if $x^{(i+1)}$ is accurate enough then quit;
  $r^{(i+1)} = s - \omega_i t$;
end

Figure 8.11: The Bi-CGSTAB algorithm

	$x^{(0)}$ is an initial guess; $r(0) = b - Ax^{(0)}$;
	$\hat{r}^{(0)}$ is an arbitrary vector, such that $(\hat{r}^{(0)})^* r^{(0)} \neq 0$,
	e.g., $\hat{r}^{(0)} = r^{(0)}$;
	$\rho_0 = 1; u = 0; \alpha = 0; \omega_2 = 1;$
	for $i = 0, 2, 4, 6, ...$
	$\rho_0 = -\omega_2 \rho_0$
even BiCG step:	$\rho_1 = (\hat{r}^{(0)})^* r^{(i)}; \beta = \alpha \rho_1 / \rho_0; \rho_0 = \rho_1$
	$u = r^{(i)} - \beta u;$
	$v = Au$
	$\gamma = (\hat{r}^{(0)})^* v; \alpha = \rho_0 / \gamma;$
	$r = r^{(i)} - \alpha v;$
	$s = Ar$
	$x = x^{(i)} + \alpha u;$
odd BiCG step:	$\rho_1 = (\hat{r}^{(0)})^* s; \beta = \alpha \rho_1 / \rho_0; \rho_0 = \rho_1$
	$v = s - \beta v;$
	$w = Av$
	$\gamma = (\hat{r}^{(0)})^* w; \alpha = \rho_0 / \gamma;$
	$u = r - \beta u$
	$r = r - \alpha v$
	$s = s - \alpha w$
	$t = As$
GCR(2)-part:	$\omega_1 = r^* s; \mu = s^* s; \nu = s^* t; \tau = t^* t;$
	$\omega_2 = r^* t; \tau = \tau - \nu^2 / \mu; \omega_2 = (\omega_2 - \nu \omega_1 / \mu) / \tau;$
	$\omega_1 = (\omega_1 - \nu \omega_2) / \mu$
	$x^{(i+2)} = x + \omega_1 r + \omega_2 s + \alpha u$
	$r^{(i+2)} = r - \omega_1 s - \omega_2 t$
	if $x^{(i+2)}$ accurate enough then quit
	$u = u - \omega_1 v - \omega_2 w$
	end

Figure 8.12: The Bi-CGSTAB(2) algorithm

Chapter 9

Preconditioning and Parallel Preconditioning

There are many occasions and applications where iterative methods fail to converge or converge very slowly. In Chapter 8.0.5 we briefly described the concept of preconditioning. We introduced methods of preconditioning systems so that their subsequent solution by iterative methods is made more computationally feasible. In this chapter, we consider preconditioning in more detail and also discuss parallel aspects. We refer the reader to [52] for further discussions on this.

9.0.9 The purpose of preconditioning

The general problem of finding a preconditioner for a linear system $Ax = b$ is to find a matrix K (the *preconditioner* or *preconditioning matrix*) with the properties that

1. K is a good approximation to A in some sense.
2. The cost of the construction of K is not prohibitive.
3. The system $Kx = b$ is much easier to solve than the original system.

The idea is then to split the matrix as $A = K - (K - A)$, and to iterate with this splitting. As we have seen in Chapter 7.0.2, this amounts to standard Richardson iteration for the preconditioned system $K^{-1}Ax = K^{-1}b$. The Richardson method for this preconditioned system leads to a Krylov subspace, generated by the preconditioned matrix, and we can select applicable Krylov subspace methods.

There are different ways of implementing preconditioning; for the same preconditioner these different implementations lead to the same eigenvalues for the preconditioned matrices. However, the convergence behavior is also dependent on the eigenvectors or, more specifically, on the components of the starting residual in eigenvector directions. Since the different implementations can have quite different eigenvectors, we might thus believe that their convergence behavior might be quite different. Three different implementations are:

1. **Left-preconditioning**: apply the iterative method to $K^{-1}Ax = K^{-1}b$. We note that symmetry of A and K does not imply symmetry of $K^{-1}A$, so that we cannot use CG in this case. Also, if we are using a minimal residual method (GMRES or MINRES), we should realize that with left-preconditioning we are minimizing the preconditioned residual $K^{-1}(b - Ax_k)$, which may be quite different from the residual $b - Ax_k$. This could have consequences for stopping criteria that are based on the norm of the residual.

2. **Right-preconditioning**: apply the iterative method to $AK^{-1}y = b$, with $x = K^{-1}y$. This form of preconditioning also does not lead to a symmetric product when A and K are symmetric. With right-preconditioning we have to be careful with stopping criteria that are based upon the error: $||y-y_k||_2$ may be much smaller than the error-norm $||K^{-1}(y-y_k)||_2$, that we are interested in. Right-preconditioning has the advantage that it only affects the operator, and it leaves the right-hand side untouched, which may be useful in the design of software.

3. **Central preconditioning**: For a preconditioner K with $K = K_1K_2$, the iterative method is applied to $K_1^{-1}AK_2^{-1}z = K_1^{-1}b$, with $x = K_2^{-1}z$. This form of preconditioning may be used for preconditioners that come naturally in factored form, like incomplete LU factorizations ($K_1 = L$, $K_2 = U$). If A is symmetric positive definite, and K is constructed by an incomplete Cholesky process, then central preconditioning preserves symmetry of the preconditioned operator, which is a prerequisite for methods such as CG. For near-symmetric systems one may hope to maintain some kind of near-symmetry with appropriate central preconditioning. It turns out that some popular methods, for instance the CG method, can be reformulated with a different choice of inner product so that one can work directly with approximations for the vector x instead of z. Also, in this case, the actions of A and the preconditioner $K = LL^T$ can be kept separate. This has been done in our formulation of preconditioned CG (see Chapter 8.0.5).

The choice of K varies from purely "black box" algebraic techniques which can be applied to general matrices to "problem dependent" preconditioners which exploit special features of a particular problem class. Although problem dependent preconditioners can be very powerful, there is still a practical need for efficient preconditioning techniques for large classes of problems. One should

realize that working with a preconditioner adds to the computational complexity of an iterative method, and the use of a preconditioner only pays if there is a sufficient reduction in the number of iterations. To get some idea of this, we make the following assumptions.
First we consider iterative methods with a fixed amount of computational overhead per iteration step, independent of the iteration number: CG, MINRES, Bi-CG, CGS, QMR, etc. We denote by t_A the computing time for the matrix-vector product with A, and the computational overhead per iteration step (for inner products, vector updates, etc) by t_O.
The computing time for k iteration steps is then given by

$$T_U = k(t_A + t_O).$$

For the preconditioned process we make the following assumptions with respect to computing time:
(a) the action of the preconditioner, for instance the computation of $K^{-1}w$ takes αt_A.
(b) the construction costs for the preconditioner are given by t_C.
(c) preconditioning reduces the number of iterations by a factor f.
The computing time for the preconditioned process, to obtain an approximation comparable to the unpreconditioned process, can be expressed as:

$$T_P = \frac{k}{f}\left((\alpha+1)t_A + t_O\right) + t_C.$$

The goal of preconditioning is that $T_P < T_U$, which is the case if

$$f > \frac{(\alpha+1)t_A + t_O}{t_A + t_O - \frac{t_C}{k}}.$$

We see that the construction of expensive preconditioners is meaningless if the number of iterations k for the unpreconditioned process is low. It is thus realistic to consider only cases where k is so large that the initial costs t_C play no role: $t_A + t_O \gg t_C/k$.
Furthermore, in many cases the matrix-vector products are the most expensive part of the computation: $t_A \geq t_O$, so that we only profit from preconditioning if the reduction f in the number of iteration steps is significantly bigger than $\alpha + 1$.
In view of the fact that many popular preconditioners are difficult to parallelize, this requirement is certainly not trivially fulfilled in many situations.

For methods such as GMRES and FOM, the situation is slightly more complicated because of the fact that the overhead costs increase quadratically with the number of iterations. Let us assume that we use GMRES(m), and that we characterize the computational time as:
t_A for the matrix-vector product with A,
t_O for the costs of one inner product plus one SAXPY.

Then the computing time for k cycles of unpreconditioned GMRES(m) is given roughly by

$$T_U = k(m\,t_A + \frac{1}{2}m^2 t_O).$$

Again we assume that km is so large that the time for constructing a preconditioner can be ignored, and that with preconditioning we need f times fewer iterations. The computing time per action of the preconditioner is again given by αt_A. Then the computing time for preconditioned GMRES(m) is given by

$$T_P = \frac{k}{f}(m(\alpha+1)t_A + \frac{1}{2}m^2 t_O),$$

and, after some manipulation, we find that preconditioning only helps to reduce the computational time if

$$f > \frac{(\alpha+1)t_A + \frac{1}{2}mt_O}{t_A + \frac{1}{2}mt_O}.$$

We see that, if m is small and if t_A dominates, then the reduction f has to be (much) bigger than $\alpha + 1$, in order to make preconditioning practical. If m is so large that $\frac{1}{2}mt_O$ dominates over t_A, then obviously a much smaller f may be sufficient to amortize the additional costs for the preconditioner.

We now say something about the effects of preconditioning. There is very little theory for what one can expect *a priori* with a specific type of preconditioner. It is well known that incomplete LU decompositions exist if the matrix A is an M-matrix, but that does not say anything about the potential reduction in the number of iterations. For the discretized Poisson equation, it has been proved [283] that the number of iterations will be reduced by a factor larger than 3.

For systems that are not positive definite, almost anything can happen. For instance, let us consider a symmetric matrix A that is indefinite. The goal of preconditioning is to approximate A by K, so that the preconditioned matrix $K^{-1}A$ has its eigenvalues clustered near 1 as much as possible. Now imagine some preconditioning process in which we can improve the preconditioner continuously from $K = I$ to $K = A$. For instance, one might think of incomplete LU-factorization with a drop-tolerance criterion. For $K = I$, the eigenvalues of the preconditioned matrix are clearly those of A and thus are at both sides of the origin. Since eventually when the preconditioner is equal to A all eigenvalues are exactly 1, the eigenvalues have to move gradually in the direction of 1, as the preconditioner is improved. The negative eigenvalues, on their way towards 1 have to pass the origin, which means that while improving the preconditioner the preconditioned matrix may from time to time have eigenvalues very close to the origin. In our chapter on iterative methods, we explained that the residual in the i-th iteration step can be expressed as

$$r_i = P_i(B)r_0,$$

where B represents the preconditioned matrix. Since the polynomial P_i has to satisfy $P_i(0) = 1$, and since the values of P_i should be small for the eigenvalues of B, this may help to explain that there may not be much reduction for components in eigenvector directions corresponding to eigenvalues close to zero, if i is still small. This means that, when we improve the preconditioner, in the sense that the eigenvalues are getting more clustered towards 1, its effect on the iterative method may be dramatically worse for some "improvements".
This explains what we have observed many times in practice. By increasing the number of fill-in entries in ILU, sometimes the number of iterations increases. In short, the number of iterations may be a very irregular function of the level of the incomplete preconditioner. For other types of preconditioners similar observations may be made.

Is there any good strategy for designing effective preconditioners for a particular class of linear problems? We do not know of such a strategy, but the following approach may help to build up one's insight into what is happening in particular cases. For a representative linear system, one starts with unpreconditioned GMRES(m), with m as high as possible. After one cycle of GMRES(m), one computes the eigenvalues of H_m: the Ritz values. These Ritz values usually give a fairly good impression of the most relevant parts of the spectrum of A. Then one does the same with the preconditioned system and inspects the effect on the spectrum. If there is no specific trend of improvement in the behavior of the Ritz values, when we try to improve the preconditioner, then obviously we have to look for another class of preconditioner. If there is an effect on the Ritz values, then this may give us some insight as to how much more the preconditioner has to be further improved in order to be effective. At all times, we have to keep in mind the rough analysis that we made in this chapter, and check whether the construction of the preconditioner and its costs per iteration are still inexpensive enough to be amortized by an appropriate reduction in the number of iterations.

9.0.10 Incomplete LU-decompositions

As we have seen in our discussion in Chapter **??**, iterative methods converge very fast if the matrix A is close to the identity matrix in some sense, and the main goal of preconditioning is to obtain a matrix $K^{-1}A$ which is close to I. The phrase "in some sense" may have different meanings for different iterative methods: for the standard Richardson matrix we want $||I - A||_2$ to be (much) smaller than 1; for many Krylov subspace methods it is desirable that the condition number of $K^{-1}A$ is (much) smaller than that of A, or that the eigenvalues of $K^{-1}A$ are strongly clustered around some point (usually 1). In all these situations the preconditioning operator K approximates A.
It is quite natural to start looking at a direct solution method for $Ax = b$, and to see what variations we can make if the direct approach becomes too expensive. The most common direct technique is to

factorize A as $A = LU$, if necessary with permutations for pivoting. One of the main problems with LU-factorization of a sparse matrix is that often the number of entries in the factors is substantially greater than in the original matrix so that, even if the original matrix can be stored, the factors can not.

In *Incomplete LU factorization* we keep the factors artificially sparse, in order to save computer time and storage for the decomposition. The incomplete factors are used for preconditioning in the following way. First note that for all iterative methods discussed we never need the matrices A or K explicitly, but we only need to be able to compute the result of Ay for any given vector y. The same holds for K, and typically we see in codes that these operations are performed by calls to appropriate subroutines. For K, in particular, we need to be able to compute efficiently the result of $K^{-1}y$ for any given vector y. In the case of an incomplete LU factorization, K is given in the form $K = \widetilde{L}\widetilde{U}$, and $z = K^{-1}y$ is then computed by solving z from $\widetilde{L}\widetilde{U}z = y$. This is done in two steps: first solve w from $\widetilde{L}w = y$ and then compute z from $\widetilde{U}z = w$. Note that these solution steps are simple backsubstitutions and if the right-hand sides are not required further in the iterative process, the solution of either back-substitution may overwrite the corresponding right-hand side, in order to save memory. We hope that it is not necessary to stress this, but one should **never** compute the inverse of K, or of its factors, explicitly, **unless** the inverse has some very convenient sparse form (for instance when K is a diagonal matrix).

We shall illustrate the above sketched process for a popular preconditioner for sparse positive definite symmetric matrices, namely, the *incomplete Cholesky factorization* [178, 238, 239] with no fill-in. We will denote this preconditioner as IC(0). CG in combination with IC(0) is often referred to as ICCG(0). We shall consider IC(0) for the matrix with five nonzero diagonals, that arises after the 5-point finite-difference discretization of the 2-dimensional Poisson equation over a rectangular region. If the entries of the three nonzero diagonals in the upper triangular part of A are stored in three arrays $a(\cdot,1)$ for the main diagonal, $a(\cdot,2)$ for the first co-diagonal, and $a(\cdot,3)$ for the n_x-th co-diagonal, then a section of the matrix looks like that shown in (9.1).

$$A = \begin{pmatrix} \ddots & & & \ddots & \ddots & \ddots & & \ddots \\ & a_{i-n_x,3} & & a_{i-1,2} & a_{i,1} & a_{i,2} & & a_{i,3} \\ & \ddots & & & \ddots & \ddots & \ddots & & \ddots \end{pmatrix} \tag{9.1}$$

This corresponds to the unknowns over a grid as shown below:

$$\begin{array}{ccccccccc}
\cdot & \cdot & \cdot & & \cdot & & \cdot & \cdot \\
\\
\cdot & \cdot & \cdot & & \cdot & & \cdot & \cdot \\
 & & & & \downarrow & & & \\
\cdot & \cdot & \cdot & \rightarrow & \cdot & \leftarrow & \cdot & \cdot \\
 & & & & \uparrow & & & \\
\cdot & \cdot & \cdot & & \cdot & & \cdot & \cdot \\
\\
\cdot & \cdot & \cdot & & \cdot & & \cdot & \cdot \\
\\
\cdot & \cdot & \cdot & & \cdot & & \cdot & \cdot
\end{array}$$

If we write A as $A = L + diag(A) + L^T$, in which L is the strictly lower triangular part of A, then the IC(0)-preconditioner can be written as

$$K = (L + D)D^{-1}(L^T + D).$$

In this expression, K represents the preconditioner to be used in the conjugate gradient scheme (Chapter 8.0.5). For IC(0), the entries d_i of the diagonal matrix D can be computed from the relation

$$\mathrm{diag}(K) = \mathrm{diag}(A).$$

For the five-diagonal A this leads to the following relations for the d_i:

$$d_i = a_{i,1} - a_{i-1,2}^2/d_{i-1} - a_{i-n_x,3}^2/d_{i-n_x}.$$

Obviously this is a recursion in both directions over the grid. This aspect will be discussed later when considering the application of the preconditioner in the context of parallel and vector processing.
The so-called *modified incomplete decompositions* [140, 188] follow from the requirement that

$$\mathrm{rowsum}(K) = \mathrm{rowsum}(A).$$

In our context this amounts to an additional correction to the diagonal entries d_i.
Axelsson and Lindskog [26] describe a relaxed form of this modified incomplete decomposition that, for the five-diagonal A, leads to the following relations for the d_i:

$$\begin{aligned} d_i &= a_{i,1} - a_{i-1,2}(a_{i-1,2} + \alpha a_{i-1,3})/d_{i-1} \\ &\quad - a_{i-n_x,3}(a_{i-n_x,3} + \alpha a_{i-n_x,2})/d_{i-n_x}. \end{aligned}$$

Note that, for $\alpha = 0$ we have the standard IC(0) decomposition, whereas for $\alpha = 1$ we have the modified incomplete Cholesky decomposition MIC(0) proposed by Gustafsson [188]. It has been observed that, in many practical situations, $\alpha = 1$ does not lead to a reduction in the number of iteration steps, with respect to $\alpha = 0$, but in our experience, taking $\alpha = .95$ almost always reduces the number of iteration steps significantly [309]. The only difference between the IC(0) and MIC(0) is the choice of the diagonal D; in fact, the off-diagonal entries of the triangular factors are identical.

For the solution of systems $Kw = r$, given by

$$K^{-1}r = (L^T + D)^{-1}D(L + D)^{-1}r,$$

it will almost never be advantageous to determine the matrices $(L^T + D)^{-1}$ and $(L + D)^{-1}$ explicitly, since these matrices are usually dense triangular matrices.
Instead, for the computation of, say, $y = (L + D)^{-1}r$, y is solved from the linear lower triangular system $(L + D)y = r$. This step then leads typically to relations for the entries y_i, of the form

$$y_i = (r_i - a_{i-1,2}y_{i-1} - a_{i-n_x,3}y_{i-n_x})/d_i,$$

which again represents a recursion in both directions over the grid, of the same form as the recursion for the d_i.

For more general matrices, we can also often make incomplete LU factorizations and frequently they can be optimized, from the implementation point of view, in a similar way as for incomplete Cholesky factorizations. In fact, many other approaches lead to precisely the same recurrence relations as for Incomplete LU and Incomplete Cholesky: Gauss-Seidel, SOR, SSOR [192], and SIP [296]. Hence these methods can often be made vectorizable or parallel in the same way as for incomplete Cholesky preconditioning.
Since vector and parallel computers do not lend themselves well to recursions in a straightforward manner, the recursions just discussed the may seriously degrade the effect of preconditioning on a vector or parallel computer, if carried out in the form given above. This sort of observation has led to different types of preconditioners, including diagonal scaling, polynomial preconditioning, and truncated Neumann series. Such approaches may be useful in certain circumstances, but they tend to increase the computational complexity (by requiring more iteration steps or by making each iteration step more expensive). On the other hand, various techniques have been proposed

to vectorize the recursions, mainly based on reordering the unknowns or changing the order of computation. For regular grids, such approaches lead to highly vectorizable code for the standard incomplete factorizations (and consequently also for Gauss-Seidel, SOR, SSOR, and SIP). Before discussing these techniques, we shall first present a way to reduce the computational complexity of preconditioning.

Efficient implementations of ILU(0) preconditioning

Suppose that the given matrix A is written in the form $A = L + \text{diag}(A) + U$, in which L and U are the strictly lower and upper triangular part of A, respectively. Eisenstat [144] has proposed an efficient implementation for preconditioned iterative methods, when the preconditioner K can be represented as

$$K = (L + D)D^{-1}(D + U), \tag{9.2}$$

in which D is a diagonal matrix. Some simple Incomplete Cholesky, incomplete LU, and the modified versions of these factorizations, as well as SSOR can be written in this form. For the incomplete factorizations, we have to ignore all the Gaussian elimination corrections to off-diagonal entries [239]; the resulting decomposition is referred to as ILU(0) in the unsymmetric case, and IC(0) for the incomplete Cholesky situation. For the 5-point finite-difference discretized operator over rectangular grids in 2D, this is equivalent to the incomplete factorizations with no fill-in, since in these situations there are no Gaussian eliminations to non-zero off-diagonal elements.
The first step to make the preconditioning more efficient is to eliminate the diagonal D in (9.2). We rescale the original linear system $Ax = b$ to obtain

$$D^{-1/2}AD^{-1/2}\tilde{x} = D^{-1/2}b, \tag{9.3}$$

or $\tilde{A}\tilde{x} = \tilde{b}$, with $\tilde{A} = D^{-1/2}AD^{-1/2}$, $\tilde{x} = D^{1/2}x$, and $\tilde{b} = D^{-1/2}b$. With $\tilde{A} = \tilde{L} + \text{diag}(\tilde{A}) + \tilde{U}$, we can easily verify that

$$\widetilde{K} = (\widetilde{L} + I)(I + \widetilde{U}). \tag{9.4}$$

Note that the corresponding triangular systems, like $(\widetilde{L} + I)r = w$, are more efficiently solved, since the division by the entries of D is avoided.
We assume that this scaling has been carried out, and the scaled system will again be denoted as $Ax = b$ (note that this scaling does not necessarily have the effect that $\text{diag}(A) = I$).
The key idea in Eisenstat's approach (also referred to as *Eisenstat's trick*) is to apply standard iterative methods (that is, in their formulation with $K = I$) to the explicitly preconditioned linear system

$$(\widetilde{L} + I)^{-1}A(I + \widetilde{U})^{-1}y = (\widetilde{L} + I)^{-1}b, \tag{9.5}$$

where $y = (I + \widetilde{U})x$. This explicitly preconditioned system will be denoted by $Py = c$. Then it follows that

$$A = \widetilde{L} + I + \mathrm{diag}(A) - 2I + I + \widetilde{U}. \tag{9.6}$$

This expression, as well as the special form of the preconditioner given by (9.4), is used to compute the vector Pz for given z:

$$Pz = (\widetilde{L} + I)^{-1}A(I + \widetilde{U})^{-1}z = (\widetilde{L} + I)^{-1}(z + (\mathrm{diag}(A) - 2I)t) + t, \tag{9.7}$$

with

$$t = (I + \widetilde{U})^{-1}z. \tag{9.8}$$

Note that the computation of Pz is equivalent to solving two triangular systems plus the multiplication of a vector by a diagonal matrix $(\mathrm{diag}(A) - 2I)$ and an addition of this result to z. Therefore the matrix-vector product for the preconditioned system can be computed virtually at the cost of the matrix-vector product of the unpreconditioned system. This fact implies that the preconditioned system can be solved by any of the iterative methods for practically the same computational cost per iteration step as the unpreconditioned system. That is to say, the preconditioning comes essentially for free, in terms of computational complexity.
In most situations we see, unfortunately, that while we have avoided the fastest part of the iteration process (the matrix-vector product Ap), we are left with the most problematic part of the computation, namely, the triangular solves. However, in some cases, as we shall see, these parts can also be optimized to about the same level of performance as the matrix-vector multiplies.

General Incomplete Decompositions

We have discussed at some length the incomplete Cholesky decomposition for the matrix corresponding to a regular 5-point discretization of the Poisson operator in 2D. It was shown in [238] that incomplete LU factorizations exist for M-matrices with an arbitrary sparsity structure, where fill-in is only accepted for specified indices.
Let the allowable fill-in positions be given by the index set S, that is

$$l_{i,j} = 0 \quad \text{if} \quad j > i \quad \text{or} \quad (i,j) \notin S; \qquad u_{i,j} = 0 \quad \text{if} \quad i > j \quad \text{or} \quad (i,j) \notin S. \tag{9.9}$$

In the previous section, we considered incomplete factorizations with no fill-in outside the sparsity pattern of A; the corresponding S would have been:

$$S = \{(i,j) \quad | \quad a_{i,j} \neq 0\}. \tag{9.10}$$

Since we want the product K, of the incomplete factors of A, to resemble A as much as possible, a typical strategy is to require the entries of $K = LU$ to match those of A on the set S:

$$k_{i,j} = a_{i,j} \quad \text{if} \quad (i,j) \in S. \tag{9.11}$$

The factors L and U, that satisfy the conditions (9.10) and (9.11), can be computed by a simple modification of the Gaussian elimination algorithm; see Figure 9.1, following [24]. The main difference from the usual Gaussian elimination algorithm is in the innermost j-loop where an update to $a_{i,j}$ is computed only if it is allowed by the constraint set S. Although we describe the incomplete decomposition for a full matrix (by referring to all elements $a_{i,j}$, it may be clear that we will almost never do this in reality for a given sparse matrix. In practical cases we only address the elements that belong to a given data structure.

```
for r := 1 step 1 until n − 1 do
    d := 1/a_{r,r}
    for i := (r + 1) step 1 until n do
        if (i, r) ∈ S then
          e := d a_{i,r}; a_{i,r} := e;
          for j := (r + 1) step 1 until n do
              if (i, j) ∈ S and (r, j) ∈ S then
                a_{i,j} := a_{i,j} − e a_{r,j}
              end if
          end (j-loop)
        end if
    end (i-loop)
end (r-loop)
```

Figure 9.1: **ILU** for an n by n matrix A

After the completion of the algorithm, the incomplete LU factors are stored in the corresponding lower and upper triangular parts of the array A, which means that we have to make a copy of the array A, since we need this array also for matrix-vector products. In special cases this can be avoided, in particular for situations where we have neglected all elimination corrections to off-diagonal elements in A. This is sometimes done in order to be able to apply Eisenstat's trick, see Section 9.0.10.

If all fill-ins are allowed, that is if S consists of all possible index pairs, then the above algorithm is simply the usual Gaussian elimination algorithm.

Although the ILU preconditioner works quite well for many problems, it can be easily improved for some PDE problems. For example, when ILU is applied to elliptic problems, its asymptotic (as the mesh size h becomes small) convergence rate is only a fixed factor better than that of the unpreconditioned A (for an analysis of the fixed improvement factor on IC(0), see [316]). This was already observed by Dupont, Kendall and Rachford [140] and, for elliptic PDEs, they proposed a simple modification which dramatically improves the performance as h tends to zero We shall next describe the generalization of this *modified* ILU (MILU) preconditioner to a general matrix A due to Gustafsson [188].

The basic idea is extremely simple: in the condition (9.11) for ILU, the condition $k_{i,i} = a_{i,i}$ is removed and a new row sum condition is added. That is, (9.11) is replaced by:

$$\sum_{j=1}^{n} k_{i,j} = \sum_{j=1}^{n} a_{i,j} \quad \forall i \quad \text{and} \quad k_{i,j} = a_{i,j} \quad \text{if} \quad i \neq j \quad \text{and} \quad (i,j) \in S. \tag{9.12}$$

Again, for certain classes of matrices, the conditions (9.12) and (9.9) are sufficient to determine the LU factors in MILU directly. However, in practice it is easier to compute these LU factors by a simple modification of the ILU algorithm: instead of dropping the forbidden fill-ins in the ILU algorithm, these terms are added to the main diagonal of the same row; see Figure 9.2. Again, it can be shown that the computed LU factors satisfy (9.12). Note that only the j-loop is different from the ILU algorithm.

Again, the LU factors overwrite the lower and upper triangular parts of the array A respectively, and A has to be saved prior to this decomposition operation, since it is required in the iterative process.

Even though MILU produces a better asymptotic condition number bound than ILU for elliptic problems, in practice MILU does not always perform better than ILU. This may have to do with a higher sensitivity of MILU to round-off errors [310]. This provides motivation for an *interpolated* version between ILU and MILU, see for example [20, 26]. The idea is that in the MILU algorithm, the update of $a_{i,i}$ in the innermost loop is replaced by:

$$a_{i,i} := a_{i,i} - \omega e a_{r,j},$$

where $0 \leq \omega \leq 1$ is a user specified relaxation parameter. Obviously, $\omega = 0$ and 1 correspond to ILU and MILU respectively. It was observed empirically in [310], and verified using the Fourier analysis method [50], that a value of $\omega = 1 - ch^2$ gives the best results for some classes of matrices coming from elliptic problems. The optimal value of c can be estimated and is related to the optimal value of c in the DKR method in [140]. Notay [250] gave strategies for choosing $\omega = \omega_{i,j}$ dynamically in order to improve the robustness and performance for anisotropic problems.

```
for r := 1 step 1 until n − 1 do
    d := 1/a_{r,r}
    for i := (r + 1) step 1 until n do
        if (i, r) ∈ S then
          e := a_{i,r} d; a_{i,r} := e;
          for j := (r + 1) step 1 until n do
              if (r, j) ∈ S then
                if (i, j) ∈ S then
                  a_{i,j} := a_{i,j} − e a_{r,j}
                else
                  a_{i,i} := a_{i,i} − e a_{r,j}
                end if
              end if
          end (j-loop)
        end if
    end (i-loop)
end (r-loop)
```

Figure 9.2: Algorithm **MILU** for a general matrix A

Variants of ILU preconditioners

Many variants of the basic ILU and MILU methods have been proposed in the literature. These variants are designed to either reduce the total computational work or to improve the performance on vector or parallel computers. We will describe some of the more popular variants and give references where more details can be found for other variants.

A natural approach is to allow more fill-in in the LU factor (that is a larger set S), than those allowed by the condition (9.10). Several possibilities have been proposed.
The most obvious variant is to allow more fills in specific locations in the LU factors, for example allowing more nonzero bands in the L and U matrices (that is larger stencils) [25, 188, 239]. The most common location-based criterion is to allow a set number of levels of fill-in, where original entries have level zero, original zeros have level ∞ and a fill-in in position (i, j) has level $Level_{ij}$

determined by

$$\min_{1 \leq k \leq \min(i,j)} \{Level_{ik} + Level_{kj} + 1\}.$$

In the case of simple discretizations of partial differential equations, this gives a simple pattern for incomplete factorizations with different levels of fill-in. For example, if the matrix is from a five-point discretization of the Laplacian in two-dimensions, level 1 fill-in will give the original pattern plus a diagonal inside the outermost band (for instance, see [239]).

The other main criterion for deciding which entries to omit is to replace the *drop-by-position* strategy in (9.10) by a *drop-by-size* one. That is, a fill-in entry is discarded if its absolute value is below a certain threshold value. This *drop tolerance* strategy was proposed by [27, 252, 330]. For application to fluid flow problems, see [76, 77, 328].
For the regular problems just mentioned, it is interesting that the level fill-in and drop strategies give a somewhat similar incomplete factorization, because the numerical value of successive fill-in levels decreases markedly, reflecting the characteristic decay in the entries of the inverse matrix. For general problems, however, the two strategies can be significantly different. Since it is usually not known *a priori* how many entries will be above a selected threshold, the dropping strategy is normally combined with restricting the number of fill-ins allowed in each column [272]. When using a threshold criterion, it is possible to change it dynamically during the factorization to attempt to achieve a target density of the factors [246].

Although the notation is not yet fully standardized, the nomenclature commonly adopted for incomplete factorizations is ILU(k), when k levels of fill-in are allowed and ILUT(α, f), for the threshold criterion when entries of modulus less than α are dropped and the maximum number of fill-ins allowed in any column is f. There are many variations on these strategies and the criteria are sometimes combined. In some cases, constraining the row sums of the incomplete factorization to match those of the matrix can help, as in MILU [188].

Shifts can be introduced to prevent break down of the incomplete factorization process. It was proved in [238] that incomplete decompositions exist for general M-matrices. It is well known that they may not exist if the matrix is positive definite, but does not have the M-matrix property. Manteuffel [232] considered incomplete Cholesky factorizations of diagonally shifted matrices. He proved that if A is symmetric positive definite, then there exists a constant $\alpha > 0$, such that the incomplete Cholesky factorization of $A + \alpha I$ exists. Since we make an incomplete factorization for $A + \alpha I$, instead of A, it is not necessarily the case that this factorization is also efficient as a preconditioner; the only purpose of the shift is to avoid breakdown of the decomposition process. Whether there exist suitable values for α such that the preconditioner exists and is efficient is a matter of trial and error.

Another point of concern is that for non M-matrices the incomplete factors of A may be very ill-conditioned. For instance, it has been demonstrated in [304] that if A comes from a 5-point finite-difference discretization of $\Delta u + \beta(u_x + u_y) = f$ then, for β sufficiently large, the incomplete LU factors may be very ill conditioned even though A has a very modest condition number. Remedies for reducing the condition numbers of L and U have been discussed in [148, 304].

Some general comments on ILU

The use of incomplete factorizations as preconditioners for symmetric systems has a long pedigree [238] and good results have been obtained for a wide range of problems. An incomplete Cholesky factorization where one level of fill-in is allowed (IC(1)) has been shown to provide a good balance between reducing the number of iterations and the cost of computing and using the preconditioning. Although it may be thought that a preordering that would result in low fill-in for a complete factorization (for example, minimum degree) might be advantageous for an incomplete factorization, it is not true in general ([109] and [142]) and sometimes the number of iterations of ICCG(0) (=CG+IC(0)-preconditioning) can double if a minimum degree ordering is used. This effect of reordering is not apparent for ILUT preconditioners.

The situation with symmetric systems has been analysed and is quite well understood. Much recent work for symmetric systems has been to develop preconditioners that can be computed and used on parallel computers. Most of this work has, however, been applicable to highly structured problems from discretizations of elliptic partial differential equations in two and three dimensions, see for example [309]. Experiments with unstructured matrices have been reported in [196, 213], with reasonable speed-ups being achieved in [213].
The situation for unsymmetric systems is, however, much less clear. Although there have been many experiments on using incomplete factorizations and there have been studies of the effect of orderings on the number of iterations [36, 76, 141], there is very little theory governing the behavior for general systems and indeed the performance of ILU preconditioners is very unpredictable. Allowing high levels of fill-in can help but again there is no guarantee, as we have argued in Section 9.0.9.

9.0.11 Some other forms of preconditioning

Sparse approximate inverse (SPAI)

Of course, the LU and LQ factorizations are ways of representing the inverse of a sparse matrix in a way that can be economically used to solve linear systems. The main reason why explicit inverses are not used is that, for irreducible matrices, the inverse will always be dense. However, this need not be a problem if we follow the flavor of ILU factorizations and compute and use a sparse approximation to the inverse. Perhaps the most obvious technique for this is to solve the problem

$$\min_K ||I - AK||_F,^1 \tag{9.13}$$

where K has some fully or partially prescribed sparsity structure. One advantage of this is that this problem can be split into n independent least-squares problems for each of the n columns of K. Each of these least-squares problems only involves a few variables (corresponding to the number of entries in the column of K) and, because they are independent, they can be solved in parallel. With these techniques it is possible [66] to successively increase the density of the approximation to reduce the value of (9.13) and so, in principle, ensure convergence of the preconditioned iterative method.

The small least-squares subproblems can be solved by the standard (dense) QR factorization [66, 179, 183]. In a further attempt to increase sparsity and reduce computational costs in the solutions of the subproblems, it has been suggested to use a few steps of GMRES to solve the subsystems [58]. A recent study indicates that the computed approximate inverse may be a good alternative for ILU [179], but it is much more expensive to compute both in terms of time and storage, at least if computed sequentially. This means that it is normally only attractive to use this technique if the computational costs for the construction can be amortized by using the preconditioner for more right-hand sides. One other problem with these approaches is that, although the residual for the approximation of a column of K can be controlled (albeit perhaps at the cost of a rather dense column in K), the nonsingularity of the matrix K is not guaranteed. Partly to avoid this, it was proposed to approximate the triangular factors of the inverse [221]. The non-singularity of the factors can be easily controlled and, if necessary, the sparsity pattern of the factors may also be controlled. Following this approach, it has been suggested to generate sparse approximations to an A-biconjugate set of vectors using drop tolerances [35]. In a scalar or vector environment, it is also much cheaper to generate the factors in this way than to solve the least-squares problems for the columns of the approximate inverse.

One of the main reasons for the interest in sparse approximate inverse preconditioners is the difficulty of parallelizing ILU preconditioners, not only in their construction but also in their use, which requires a sparse triangular solution. However, although almost every paper on approximate inverse preconditioners states that the authors are working on a parallel implementation, there are relatively few such studies available. For highly structured matrices, some experiences have been reported in [182]. Gustafsson and Lindskog [190], have implemented a fully parallel preconditioner based on truncated Neumann expansions [305] to approximate the inverse SSOR factors of the matrix. Their experiments (on a CM-200) show a worthwhile improvement over a simple diagonal scaling.

[1] We recall that $\| \|_F$ denotes the Frobenius norm of a matrix viz. $\|A\|_F \equiv \sqrt{\sum_{i,j} a_{i,j}^2}$.

Note that, because the inverse of the inverse of a sparse matrix is sparse, there are classes of dense matrices for which a sparse approximate inverse might be a very appropriate preconditioner. This may be the case for matrices that arise from inverse problems.

For some classes of problems, it may be attractive to construct the explicit inverses of the LU factors, even if these are considerably less sparse than the factors L and U, because such a factorization can be more efficient in parallel [5]. An incomplete form of this factorization for use as a preconditioner has been proposed in [4].

Polynomial preconditioning

Of course, it is, in theory, possible to represent the inverse by a polynomial in the matrix and one could use this polynomial as a preconditioner. However, one should realize that the iterative solvers, considered in this book, form approximate solutions in the Krylov subspace. This means that the solutions can be interpreted as polynomials in the (preconditioned) matrix, applied to the right-hand side. Since the Krylov methods construct such solutions with certain optimality properties (for instance minimal residual), it is not so obvious why an additional polynomial might be effective as a preconditioner. The main motivation for considering polynomial preconditioning is to improve the parallel performance of the solver, since the matrix-vector product is often more parallelizable than other parts of the solver (for instance the inner products). The main problem is to find effective low degree polynomials. One approach, reported in [105], is to use the low order terms of a Neumann expansion of $(I - B)^{-1}$, if A can be written as $A = I - B$ and the spectral radius of B is less than 1. It was suggested in [105] to use a matrix splitting $A = K - N$ and a truncated power series for $K^{-1}N$ when the condition on B is not satisfied. More general polynomial preconditioners have also been proposed (see, for example, [18, 212, 268]).

Because the iterative solvers implicitly construct (optimal) polynomial approximations themselves, using spectral information obtained during the iterations, it is not easy to find effective alternatives without knowing such spectral information explicitly. This may help explain why the experimental results are not generally very encouraging and have been particularly disappointing for unsymmetric problems.

Preconditioning by blocks or domains

Another whole class of preconditioners that use direct methods are those where the direct method, or an incomplete version of it, is used to solve a subproblem of the original problem. This is often used in a domain decomposition setting, where problems on subdomains are solved by a direct method but the interaction between the subproblems is handled by an iterative technique.

If the system is reducible and the matrix is block diagonal, then the solution to the overall problem is just the union of the solution of the subproblems corresponding to the diagonal blocks. Although

the overall problem may be very large, it is possible that the subproblems are small enough to be solved by a direct method. This solution is effected by a block Jacobi factorization and the matrix that is preconditioned by this block Jacobi factorization is just the identity. In general, our system will not be reducible but it might still be appropriate to use the block Jacobi method as a preconditioner.
For general systems, one could apply a block Jacobi preconditioning to the normal equations which would result in the block Cimmino algorithm [14]. A similar relationship exists between a block SOR preconditioning and the block Kacmarz algorithm [40].
Block preconditioning for symmetric systems is discussed in [63]; in [64] incomplete factorizations are used within the diagonal blocks. Attempts have been made to preorder matrices to put large entries into the diagonal blocks so that the inverse of the matrix is well approximated by the block diagonal matrix whose block entries are the inverses of the diagonal blocks [54].

In a domain decomposition approach, the physical domain or grid is decomposed into a number of overlapping or non-overlapping subdomains on each of which an independent complete or incomplete factorization can be computed and applied in parallel. The main idea is to obtain more parallelism at the subdomain level rather than at the grid-point level. Usually, the interfaces or overlapping region between the subdomains must be treated in a special manner. The advantage of this approach is that it is quite general and can be used with different methods used within different subdomains.

Radicati and Robert [263] used an algebraic version of this approach by computing ILU factors within overlapping block diagonals of a given matrix A. When applying the preconditioner to a vector v, the values on the overlapped region are averaged from the two values computed from the two overlapping ILU factors. The approach of Radicati and Robert has been further refined by De Sturler [78], who studies the effects of overlap from the point of view of geometric domain decomposition. He introduces artificial mixed boundary conditions on the internal boundaries of the subdomains. In [78]:Table 5.8, experimental results are shown for a decomposition into 20×20 slightly overlapping subdomains of a 200×400 mesh for a discretized convection-diffusion equation (5-point stencil). Using an ILU preconditioning on each subdomain, it is shown that the complete linear system can be solved by GMRES on a 400-processor distributed memory Parsytec system with an efficiency in the order of 80% (that means that with this domain adapted preconditioner the process is 320 times faster than ILU preconditioned GMRES for the unpartitioned linear system on one single processor).

In [298], Tan studied the interface conditions along boundaries of subdomains and forced continuity for the solution and some low order derivatives at the interface. He proposed to include also mixed derivatives in these relations, in addition to the conventional tangential and normal derivatives. The parameters involved are determined locally by means of normal mode analysis, and they are adapted to the discretized problem. It is shown that the resulting domain decomposition method defines a standard iterative method for some splitting $A = K - N$, and the local coupling aims to minimize the largest eigenvalues of $I - AM^{-1}$. Of course this method can be accelerated, and impressive results for GMRES acceleration are shown in [298]. Some attention is paid to the case where the solutions for the subdomains are obtained with only modest accuracy per iteration step.

Chan and Govaerts [51] showed that the domain decomposition approach can actually lead to *improved* convergence rates, at least when the number of subdomains is not too large. This is because of the well known divide and conquer effect when applied to methods with superlinear complexity such as ILU: it is more efficient to apply such methods to smaller problems and piece the global solution together.

Recently, Washio and Hayami [324] employed a domain decomposition approach for a rectangular grid in which one step of SSOR is performed for the interior part of each subdomain. In order to make this domain-decoupled SSOR more like global SSOR, the SSOR iteration matrix for each subdomain is modified by premultiplying it by a matrix $(I - X_L)^{-1}$ and postmultiplying it by $(I - X_U)^{-1}$. The matrices X_L and X_U depend on the couplings between adjacent subdomains. In order to further improve the parallel performance, the inverses are approximated by low-order truncated Neumann series. A similar approach is suggested in [324] for a block modified ILU preconditioner. Experimental results have been shown for a 32-processor NEC-Cenju distributed memory computer.

Element by element preconditioners

In finite-element problems it is not always possible or attractive to assemble the entire matrix, and hence, preconditioners are required that can be constructed at the element level. The first to propose such *element by element* preconditioners were Hughes et al [207]. The main idea in these element by element preconditioners is that the element matrices are LU-decomposed and that the back and forward sweeps associated with the (incomplete) LU-factorizations are replaced by a series of minisweeps for the element factorizations. We will explain this in some more detail. Let $Ax = b$ denote the global system in the finite-element model. The matrix A is assembled from the local element matrices A_e, and we have

$$A = \sum_{e=1}^{n_e} A_e,$$

where n_e denotes the number of elements.
In many codes, the assembling of A is avoided in order to prevent computer memory problems, and instead all computations are done with the local A_e. The preconditioners that we discussed earlier are, however, based on the structure of the global matrix A. Hughes et al proposed a local element preconditioning matrix P_e as follows:

$$P_e = I + D^{-1/2}(A_e - D_e)D^{-1/2},$$

in which D denotes the diagonal of A, and D_e denotes the diagonal of A_e. It is easy to construct D from the A_e, and it is not necessary to assemble A completely for this. The idea is that

$$P \equiv \prod_{e=1}^{n_e} P_e,$$

may be viewed as an approximation for the scaled matrix $D^{-1/2}AD^{-1/2}$, and this matrix P is taken as the element-by-element preconditioner.
In iterative solvers, we then have to solve systems like $Pw = z$, and this is simply done by computing w from

$$w = \prod_{e=1}^{n_e} P_e^{-1} z,$$

that is, a small system with P_e is solved for each relevant section of the right-hand side z (in fact the elements of z corresponding to the e^{th} element).
For symmetric positive definite problems, there is a problem, since the product of symmetric matrices is not necessarily symmetric, so that P cannot be used in combination with, for instance, Conjugate Gradients. Hughes et al suggest circumventing this problem by first making a Cholesky decomposition of P_e:

$$P_e = L_e L_e^T.$$

The preconditioner is then taken as $\widetilde{P} = LL^T$, defined by

$$L \equiv \prod_{e=1}^{n_e} L_e.$$

In [317], the parallel implementation of the element-by-element preconditioner has been discussed. Note that we are free to choose the order in which we number the elements; although each ordering leads formally to a different preconditioner. We may use this 'freedom' to select an ordering that admits some degree of parallelism, and in [317] it is proposed to subdivide the set of elements into n_g groups, each of n_{eg} nonadjacent elements. It is easily verified that the subproduct of element preconditioning matrices for each group $(\prod_{eg=1}^{n_{eg}} P_{eg}^{-1})z$ can be written as a sum:

$$\left(\prod_{eg=1}^{n_{eg}} P_{eg}^{-1}\right) z = \sum_{eg=1}^{n_{eg}} P_{eg}^{-1} z,$$

and each term in this sum can be processed in parallel.

A slightly different idea is suggested in [189]. In that paper it is suggested, for symmetric positive-definite A, to decompose each A_e as $A_e = L_e L_e^T$, and to construct the preconditioner as $K = LL^T$, with

$$L = \sum_{e=1}^{n_e} L_e.$$

Since L is not necessarily a lower triangular matrix, this has to be forced explicitly by performing the local node numbering so that increasing local node numbers correspond to increasing global node numbers. Since the L_e may be singular, it is further suggested to improve the numerical stability of L (by increasing the values of the diagonal entries relative to the off-diagonal entries), by replacing L with L_ξ:

$$L_\xi = \frac{1}{1+\xi h}\widetilde{L} + (1+\xi h)D_L,$$

where $\widetilde{L}$ denotes the off-diagonal part of L, D_L represents the diagonal of L, and h is a measure for the size of the finite elements. Also for this approach, we can treat non-adjacent elements in parallel.

There may be other reasons for considering element by element inspired preconditioners. In many realistic models there is some local problem, for instance the forming of cracks under the influence of point forces in concrete or other materials. It may then seem logical to assemble the elements, around the place where some particular effect is expected, into some super-element. We can then form either complete or incomplete decompositions of these super-elements, depending on their size or complexity, and repeat the above procedure with the mix of super-elements and remaining regular elements. This approach bridges the gap between element by element preconditioning and the (incomplete) LU factorization of the fully assembled matrix, and it seems plausible that the effect of the preconditioning based on super-elements will more and more resemble the preconditioning based on the fully assembled matrix as the super-elements grow in size. Some promising results have been obtained by this super-element technique (see, for instance, [75, 317], but the selection of appropriate super-elements seems to be an art rather than a science at the moment.

9.0.12 Vector and parallel implementation of preconditioners

Partial vectorization

A common approach for the vectorization of the preconditioning part of an algorithm is known as *partial vectorization*. In this approach, the nonvectorizable loops are split into vectorizable parts and nonvectorizable remainders. Schematically, this approach can be explained as follows.
If we assume that the preconditioner is written in the form $K = LU$, where L is lower triangular and U is upper triangular then, as we discussed before, solving w from $Kw = r$ consists in solving $Ly = r$ and $Uw = y$ successively. Both systems lead to similar vectorization problems, and therefore we consider only the partial vectorization of the computation of y from $Ly = r$.
The first step is to regard L as a block matrix with blocks of suitably chosen sizes (not all the blocks need to have equal size):

$$L = \begin{pmatrix} L_{1,1} & & & & \\ L_{2,1} & L_{2,2} & & & \\ L_{3,1} & L_{3,2} & L_{3,3} & & \\ . & . & . & . & \\ . & . & . & . & \\ L_{n,1} & L_{n,2} & . & . & L_{n,n} \end{pmatrix}.$$

Next, we partition the vectors y and r conformally in subvectors y_i and r_i, so that the vector length of the ith subvector is equal to the block size of $L_{i,i}$. The subvector y_i is then the solution of

$$L_{i,i}y_i = r_i - (L_{i,1}y_1 + L_{i,2}y_2 + \cdots + L_{i,i-1}y_{i-1}).$$

Note that the amount of work for computing all the partitions of y successively is equal to the amount of work for solving $Ly = r$ in a straightforward manner. However, by rearranging the loops for the subblocks, we see that computations for the right-hand side, for each subvector y_i, can be vectorized.

For the five-point finite-difference matrix A, we take the block size equal to n_x, the number of grid points in the x-direction. In that case the standard incomplete decomposition of A leads to a factor L for which the $L_{i,i}$ are lower bidiagonal matrices, the $L_{i,i-1}$ are diagonal matrices, and all of the other $L_{i,j}$ vanish. Hence, the original nonvectorizable three-term recurrence relations are now replaced by two-term recurrence relations of length n_x, and vectorizable statements of length n_x also.

We have thus vectorized half of the work in the preconditioning step, so that the performance of this part almost doubles. In practice the performance is often even better because, for many machines, optimized software is available for two-term recurrence relations and this type of computation is often automatically replaced in a Fortran code by the optimized code.

We illustrate the effect of partial vectorization by an example. If our five-diagonal model problem is solved by the preconditioned CG algorithm, the operation count per iteration step is roughly composed as follows: $6N$ flops for the three vector updates, $4N$ flops for the two inner products, $9N$ flops for the matrix-vector product, and $8N$ flops for solving $Kw = r$, if we assume that A has been scaled such that the factors of K have unit diagonal. Assuming that the first $19N$ flops are executed at a very high vector speed and that the preconditioning part is not vectorized and runs at a speed of S Mflop/s, we conclude, using *Amdahl's law* (see S Mflop/s, we conclude, using *Amdahl's law* (see Chapter 4.1), that the Mflop/s rate for one preconditioned CG iteration step is approximately given by

$$27/(8/S) \simeq 3.4S \text{ Mflop/s}.$$

Since for most existing vector computers S is rather modest, the straightforward coded preconditioned CG algorithm (as well as other iterative methods) has a disappointingly low performance. Note that applying Eisenstat's trick (Section 9.0.10) does not lower the CPU time noticeably in this case, since the preconditioning is really the bottleneck. With partial vectorization, we find that the Mflop/s rate will be approximately

$$27/(4/S_1) \simeq 6.8 S_1 \text{ Mflop/s},$$

where S_1 is the Mflop/s rate for a two-term recursion. For many computers, S_1 can be twice as large as S. In practice, the modest blocksize of the subblocks $L_{i,i-1}$ will also often inhibit high Mflop/s rates for the vectorized part of the preconditioning step. Nevertheless, it is not uncommon to observe that partial vectorization more than doubles the performance.
For most parallel computers and many preconditioners, the performance of the preconditioned CG process is so low that the reduction in the number of iteration steps (because of preconditioning) is not reflected by a comparable reduction in CPU time, with respect to the unpreconditioned process. In other words, we have to seek better parallelizable or vectorizable preconditioners in order to beat the unpreconditioned CG process with respect to CPU time (see also Section 9.0.9).

Reordering the Unknowns

A standard trick is to select all unknowns that have no direct relationship with each other and to number them first. This is repeated for the remaining unknowns. For the five-point finite-difference discretization over rectangular grids, this approach is known as a *red-black ordering*. For more complicated discretizations, graph coloring techniques can be used to decouple the unknowns in large groups. In either case, the effect is that the matrix is permuted correspondingly and can be written, after reordering, in block form as

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} & . & . & A_{1,s} \\ A_{2,1} & A_{2,2} & A_{2,3} & . & . & A_{2,s} \\ A_{3,1} & A_{3,2} & A_{3,3} & . & . & A_{3,s} \\ . & . & . & . & . & \\ . & . & . & . & . & \\ A_{s,1} & A_{s,2} & A_{s,3} & . & . & A_{s,s} \end{pmatrix}$$

such that all the block matrices $A_{j,j}$ are diagonal matrices. For example, for red-black ordering we have $s = 2$. Then the incomplete LU factorization K of the form

$$K = (L + D) D^{-1} (U + D)$$

with L and U equal to the strict lower and strict upper triangular part of A, respectively, leads to factors $L + D$ and $U + D$ that can be represented by the same nonzero structure, for example

$$L = \begin{pmatrix} D_{1,1} & & & & \\ A_{2,1} & D_{2,2} & & & \\ A_{3,1} & A_{3,2} & D_{3,3} & & \\ . & . & . & . & \\ . & . & . & . & \\ A_{s,1} & A_{s,2} & . & . & D_{s,s} \end{pmatrix}.$$

The corresponding triangular system $(L + D)y = r$ can be solved in an obvious way by exploiting the block structure on a vector or parallel computer. The required matrix-vector products for the subblocks $A_{i,j}$ can be optimized as in Section 9.0.12 (with, of course, smaller vector lengths than for the original system).

For the five-point finite-difference matrix A, the red-black ordering leads to a very vectorizable (and parallel) preconditioner. The performance of the preconditioning step is as high as the performance of the matrix-vector product. This implies that the preconditioned processes, when applying Eisenstat's implementation, can be coded so that an iteration step of the preconditioned algorithm takes approximately the same amount of CPU time as for the unpreconditioned method. Hence any reduction in the number of iteration steps, resulting from the preconditioner, translates immediately to almost the same reduction in CPU time.

One should realize, however, that in general the factors of the incomplete LU factorization of the permuted matrix A are not equal to the similarly permuted incomplete factors of A itself. In other words, changing the order of the unknowns leads in general to a different preconditioner. This fact should not necessarily be a drawback, but often the reordering appears to have a rather strong effect on the number of iterations, so that it can easily happen that the parallelism or vectorizability obtained is effectively degraded by the increase in (iteration) work. Of course, it may also be the other way around—that reordering leads to a decrease in the number of iteration steps as a free bonus to the obtained parallelism or vectorizability.
For standard five-point finite-difference discretizations of second-order elliptic PDEs, Duff and Meurant [109] report on experiments that show that most reordering schemes (including nested dissection and red-black orderings) lead to a considerable increase in iteration steps (and hence in computing time) compared with the standard lexicographical ordering. An exception seems to be a class of parallel orderings introduced in [307], which will be described in Section 9.0.12. As noted before, this may work out differently in other situations, but one should be aware of these possible adverse effects.

Meier and Sameh [236] report on the parallelization of the preconditioned CG algorithm for a multivector processor with a hierarchical memory (for example the Alliant FX series). Their approach is based on a red-black ordering in combination with forming a reduced system (Schur complement).

Changing the Order of Computation

In some situations it is possible to change the order of the computations (by implicitly reordering the unknowns) without changing the results. This means that bitwise the same results are produced, with the same roundoff effects. The only effect is that the order in which the results are produced may differ from the standard lexicographical ordering. A prime example is the incomplete LU preconditioner for the five-point finite-difference operator over a rectangular grid.

We now number the vector and matrix entries according to the position in the grid, that is $y_{i,j}$ refers to the component of y corresponding to the i,j-th gridpoint (i in x-direction and j in y-direction). The typical expression in the solution for the lower triangular system $Ly = r$ is

$$y_{i,j} = (r_{i,j} - a_{i-1,j,2}\, y_{i-1,j} - a_{i,j-1,3}\, y_{i,j-1})/d_{i,j},$$

where $y_{i,j}$ depends only on its previously computed neighbors in the west and south directions over the grid:

$$\begin{array}{ccccc} \cdot & \cdot & & \cdot & \cdot \\ \cdot & \underset{y_{i-1,j}}{\cdot} & \rightarrow & \underset{y_{i,j}}{\cdot} & \cdot \\ & & & \uparrow & \\ \cdot & \cdot & & \underset{y_{i,j-1}}{\cdot} & \cdot \\ \cdot & \cdot & & \cdot & \cdot \end{array} \tag{9.14}$$

Hence the unknowns $y_{i,j}$ corresponding to the grid points along a diagonal of the grid, that is $i+j = constant$, depend only on the values of y corresponding to the previous diagonal. Therefore, if we compute the unknowns in the order corresponding to these diagonals over the grid, for each diagonal the y-values can be computed independently, or in vector mode.

A number of complications arise, however. If we do not wish to rearrange the unknowns in memory explicitly, then a non-unit stride is involved in the vector operations; typically this non-unit stride is $n_x - 1$, where n_x denotes the number of gridpoints in x-direction. For some computers a non-unit stride is not attractive (for instance, for machines with a relatively small cache), while on others one might encounter a severe degradation in performance because of memory bank conflicts.

The other problem is that the vectorizable length for the preconditioning part is only $\min(n_x, n_y)$ at most, and many of the loops are shorter. Thus, the average vector length may be quite small, and it really depends on the $n_{1/2}$ value whether the diagonal approach is profitable on a given architecture. Moreover, there are typically more diagonals in the grid than there are grid lines, which means that there are about $n_x + n_y - 1$ vector loops, in contrast to only n_y (unvectorized) loops in the standard lexicographical approach. Some computers have well-optimized code for recursions over one grid line. Again, it then depends on the situation whether the additional overhead for the increased number of loops offsets the advantage of having (relatively short) vector loops. Therefore, it may be advisable to explicitly reorder the unknowns corresponding to grid diagonals.

In three-dimensional problems, there are even more possibilities to obtain vectorizable or parallel code. For the standard seven-point finite-difference approximation of elliptic PDEs over a rectangular regular grid, the obvious extension to the diagonal approach in two dimensions is known as the hyperplane ordering. We now explain this in more detail.
From now on, the unknowns as well as the matrix coefficients will be indicated by three indices i, j, k, so that i refers to the index of the corresponding grid point in the x-direction, and j and k likewise in the y and z-directions. The typical relation for solving the lower triangular system in three dimensions is as follows:

$$\begin{aligned} y_{i,j,k} = (r_{i,j,k} &- a_{i-1,j,k,2}\, y_{i-1,j,k} - a_{i,j-1,k,3}\, y_{i,j-1,k} \\ &- a_{i,j,k-1,4}\, y_{i,j,k-1})/d_{i,j,k}. \end{aligned} \tag{9.15}$$

The hyperplane H^m is defined as the collection of grid points for which the triples (i, j, k) have equal sum $i + j + k = m$. Then it is obvious that all unknowns corresponding to H^m can be computed independently (that is in vector mode or in parallel) from those corresponding to H^{m-1}. This approach leads to vector lengths of $\mathcal{O}(N^{2/3})$, but the difficulty is that the unknowns required for two successive hyperplanes are not located as vectors in memory, and indirect addressing is the standard technique to identify the unknowns over the hyperplanes.

On most supercomputers, indirect addressing degrades the performance of the computation. Sometimes this is because of the overhead in computing indices, in other cases it is because of the fact that cache memories cannot be used effectively.
In [309], the reported performance for ICCG in three dimensions shows that this method, with the hyperplane ordering, can hardly compete with standard conjugate gradient applied to the

diagonally scaled system, because of the adverse effects of indirect addressing.
However, in [277, 308] ways are presented that may help to circumvent these degradations in performance. The main idea is to rearrange the unknowns explicitly in memory, corresponding to the hyperplane ordering, where the ordering within each hyperplane is chosen suitably. The more detailed description that follows has been taken from [277].

With respect to the hyperplane ordering, equation (9.15) is replaced by the set of equations in Figure 9.3.

for $m = 4, 5, 6, ..., n_x + n_y + n_z$
    for $(i, j, k) \in H^m$:
(a) $\quad y_{i,j,k} = r_{i,j,k} - a_{i,j,k-1,4}\, y_{i,j,k-1}$
(b) $\quad y_{i,j,k} = y_{i,j,k} - a_{i,j-1,k,3}\, y_{i,j-1,k}$
(c) $\quad y_{i,j,k} = y_{i,j,k} - a_{i-1,j,k,2}\, y_{i-1,j,k}$
(d) $\quad y_{i,j,k} = y_{i,j,k}/d_{i,j,k}$

Figure 9.3: Hyperplane dependencies

We have separated step (d). In practical implementations, it is advisable to scale the given linear system such that $d_{i,j,k} = 1$ for all (i, j, k). We shall discuss only part (c); the others can be vectorized similarly. Part (c) is rewritten as in Figure 9.4.

for $i = \max(2, m - n_y - n_z), ..., \min(n_x, m - 2)$
    for $j = \max(1, m - i - n_z), ..., \min(n_y, m - i - 1)$
        $k = m - i - j$
        $y_{i,j,k} = y_{i,j,k} - a_{i-1,j,k,2}\, y_{i-1,j,k}$
    end j
end i

Figure 9.4: Hyperplane dependencies, step (c)

This scheme defines the ordering of the unknowns within one hyperplane. The obvious way to implement the algorithm in Figure 9.4 is to store $y_{i,j,k}$ and $a_{i-1,j,k,2}$ in the order in which they are required over H^m. This has to be done only once at the start of the iteration process. Of course, this suggests that we have to store the matrices twice, but this is not really necessary, as we have shown in Chapter **??**.

Although the entries of y and $a(\star, 2)$ have been reordered, indirect addressing is still required for the entries $y_{i-1,j,k}$ corresponding to H^{m-1}. Schematically the algorithm in Figure 9.4 can be implemented by the following steps:

1. The required entries $y_{i-1,j,k}$ are gathered into an array V, in the order in which they are required to update the $y_{i,j,k}$ over H^m. The "gaps" in V, when H^m is larger than H^{m-1}, are left zero.
2. The entries of V are multiplied by the $a_{i-1,j,k,2}$, which are already in the desired order.
3. The result from the previous step is subtracted component-wise from the $y_{i,j,k}$ corresponding to H^m.

It was reported in [277, 308] that this approach can lead to a satisfactory performance; such a performance has been demonstrated for machines that gave a bad Megaflop rate for the standard hyperplane approach with indirect addressing. In [37] a similar approach was developed for the CM-5 computer.

Some Other Vectorizable Preconditioners

Of course, many suggestions have been made for the construction of vectorizable preconditioners. The simplest is diagonal scaling, where the matrix A is scaled symmetrically so that the diagonal of the scaled matrix has unit entries. This is known to be quite effective, since it helps to reduce the condition number [156, 300] and often has a beneficial influence on the convergence behavior. On some vector computers, the computational speed of the resulting iterative method (without any further preconditioning) is so high that it is often competitive with many of the approaches that have been suggested in previous sections [193, 309].

Nevertheless, in many situations more powerful preconditioners are needed, and many vectorizable variants of these have been proposed. One of the first suggestions was to approximate the inverse of A by a truncated Neumann series [105]. When A is diagonally dominant and scaled such that $\mathrm{diag}(A) = I$, then it can be written as $A = I - B$, and A^{-1} can be evaluated in a Neumann series as

$$A^{-1} = (I - B)^{-1} = I + B + B^2 + B^3 + \cdots. \tag{9.16}$$

Dubois et al. [105] suggest taking a truncated Neumann series as the preconditioner, that is approximating A^{-1} by

$$K^{-1} = I + B + B^2 + \cdots + B^p. \tag{9.17}$$

By observing that this preconditioner, K^{-1}, can be written as a pth degree polynomial P in A, it is obvious that all the iterative methods now lead to iteration vectors x_i in the Krylov subspace that is formed with $P(A)A$ (instead of A, as for the unpreconditioned methods). That is, after m iteration steps we arrive at a Krylov subspace of restricted form: it contains only powers of $P(A)A$ times the starting residual. This is in contrast with the regular Krylov subspace that is obtained after $m(p+1)$ iteration steps with the unpreconditioned method, and that contains also all intermediate powers of A. In both cases, the amount of work spent in matrix-vector multiplications is the same; hence, at the cost of more iterations, the unpreconditioned process can lead, in theory, to a better approximation for the solution, since it has a larger subspace at its disposal. Therefore it is not plausible that polynomial preconditioning leads to big savings in general. Any possible gain is due to the fact that the overhead in the polynomial preconditioned case may be smaller.

More sophisticated polynomial preconditioners are obtained when arbitrary coefficients are allowed in the polynomial expansion for A^{-1} [212, 268]. Apparently they still suffer from the same disadvantage in that they generate approximate solutions in Krylov subspaces of a restricted form, at the cost of the same number of matrix-vector products for which the unpreconditioned method generates a "complete" Krylov subspace. However, they can certainly be of advantage in a parallel environment, since they reduce the effect of synchronization points in the method. Another advantage is that they may lead to an "effective" Krylov subspace (that is: containing only the powers of A that really matter) in fewer iteration steps with less loss of orthogonality. As far as we know, this point has not yet been investigated.

Obviously, the inverse of A is better approximated by a truncated Neumann series of a fixed degree when A is more diagonally dominant. This is the idea behind a truncated Neumann series approach suggested in [305]. First, an incomplete factorization of A is constructed. To simplify the description, we assume that A has been scaled such that the diagonal entries in the factors of the incomplete decomposition are equal to 1 (see Section 9.0.10):

$$K = (L + I)(I + U). \tag{9.18}$$

Then the factors are written in some suitable block form, as in Section 9.0.12, viz.

$$(L+I) = \begin{pmatrix} L_{1,1} & & & & \\ L_{2,1} & L_{2,2} & & & \\ L_{3,1} & L_{3,2} & L_{3,3} & & \\ . & . & . & . & . \\ . & . & . & . & . \\ L_{m,1} & L_{m,2} & . & . & L_{m,m} \end{pmatrix}.$$

If the computation of $L_{i,i}^{-1} r_i$ were an efficient vectorizable operation, then the complete process of solving $(L+I)z = r$ could be vectorized, for segments z_i of z, corresponding to the size of $L_{i,i}$:

$$z = L_{i,i}^{-1}(r_{i,i} - L_{i,i-1}z_i - ... - L_{i,1}z_1) \tag{9.19}$$

(assuming that the operations $L_{i,j}z_j$ are vectorizable operations). In many relevant situations it happens that the factors $L+I$ and $I+U$ are diagonally dominant when A is diagonally dominant, and one may then expect that the subblocks $L_{i,i}$ are even more diagonally dominant. Van der Vorst [305] has proposed using truncated Neumann series only for the inversion of these diagonal blocks of L (and U). He has shown both theoretically and experimentally that only a few terms in the Neumann series, say 2 or 3, suffice to get an efficient (and vectorizable) process, for problems that come from five-point finite-difference approximations in two dimensions.
In most situations, there is a price to be paid for this vectorization, in that the number of iteration steps increases slightly and also the number of floating-point operations per iteration step increases by $4N$ (for the 2-term truncated variant). Van der Vorst [305] has shown for a model problem that the increase in iteration steps is modest when only 2 terms in the Neumann series are used.
In [309] this approach is extended to the three-dimensional situation, where $I+L$ can be viewed as a nested block form. The resulting method, which has the name "nested truncated Neumann series", leads to rather long vector lengths and can be attractive for some special classes of problem on some parallel computers.

Finally, we comment on a vectorizable preconditioner that has been suggested by Meurant [240]. The starting point is a so-called block preconditioner, that is a preconditioner of the form

$$K = (L+B)B^{-1}(B+U), \tag{9.20}$$

in which B itself is a block diagonal matrix. This type of preconditioner has been suggested by many authors [63, 217, 237]. Most of these block preconditioners differ in the choice of B. They are reported to be quite effective in two-dimensions (in which A is block tridiagonal) in that they significantly reduce the number of iteration steps for many problems. However, in three-dimensions, experience leads to less favorable conclusions (see, for example, [217]). Moreover, for

vector computers they share the drawback that the inversion of the diagonal blocks of B (which are commonly tridiagonal matrices) can lead to rather poor performance.
Meurant [240] has proposed a variant to a block preconditioner introduced in [63], in which he approximates the inverses of these tridiagonal blocks of B by some suitably chosen band matrices. He reports on results for some vector computers (CRAY-1, CRAY X-MP, and CYBER 205) and shows that this approach leads often to lower CPU times than the truncated Neumann series approach.

Parallel aspects of reorderings

By reordering the unknowns, a matrix structure can be obtained that allows for parallelism in the triangular factors representing the incomplete decomposition. The red-black ordering, for instance, leads to such a highly parallel form. As has been mentioned before, this reordering often leads to an increase in the number of iteration steps, with respect to the standard lexicographical ordering.

Of more interest is the effort that has been put into constructing a parallel preconditioner, since these attempts are also relevant for the other iterative methods.

Let us write the triangular factors of K in block bidiagonal form:

$$L = \begin{pmatrix} L_{1,1} & & & & \\ L_{2,1} & L_{2,2} & & & \\ & L_{3,2} & L_{3,3} & & \\ & & \cdot \ \ \cdot & \cdot & \\ & & & \cdot \ \ \cdot & \cdot \\ & & & L_{p,p-1} & L_{p,p} \end{pmatrix}$$

For p not too small, Seager [279] suggests setting some of the off-diagonal blocks of L to zero (and to do so in a symmetrical way also in the upper triangular factor U). Then the back substitution process is decoupled into a set of independent back substitution processes. The main disadvantage of this approach is that often the number of iteration steps increases, especially when more off-diagonal blocks are discarded.

Radicati and Robert [261] suggest constructing, in parallel, incomplete factorizations of slightly overlapping parts of the matrix. They report that this can lead to a decrease in the number of iteration steps (in experiments carried out on a six-processor IBM 3090/VF).

Still another approach, suggested by Meurant [241], exploits the idea of the two-sided (or twisted) Gaussian elimination procedure for tridiagonal matrices [29, 307]. This is generalized for the incomplete factorization. In this approach, K is written as ST, where S takes the (twisted) form

$$
\begin{pmatrix}
S_{1,1} & & & & & & & \\
S_{2,1} & S_{2,2} & & & & & & \\
 & S_{3,2} & S_{3,3} & & & & & \\
 & & \cdot & \cdot & & & & \\
 & & & S_{p-1,p} & S_{p,p} & S_{p,p+1} & & \\
 & & & & & S_{p+1,p+1} & S_{p+1,p+2} & \\
 & & & & & & \cdot & \cdot \\
 & & & & & & & S_{q,q}
\end{pmatrix}
$$

(and T has a block structure similar to S^T). This approach can be viewed as starting the (incomplete) factorization process simultaneously at both ends of the matrix A. This factorization is discussed in Chapters 7.4 and 7.5 of the LINPACK Users' Guide [90] where it is attributed to Jim Wilkinson. It is colloquially referred to as the BABE algorithm (for **B**urn **A**t **B**oth **E**nds).

Van der Vorst [307] has shown how this procedure can be done in a nested way for the diagonal blocks of S (and T) too. For the two-dimensional five-point finite-difference discretization over a rectangular grid, the first approach comes down to reordering the unknowns (and the corresponding equations) as

$$
\begin{array}{cccccc}
1 & 3 & 5 & 7 & 9 & 11 \\
13 & 15 & 17 & 19 & 21 & 23 \\
\rightarrow & & & & & \\
 & \cdot & \cdot & \cdot & \cdot & \\
 & & & & & \\
\rightarrow & & & & & \\
14 & 16 & 18 & 20 & 22 & 24 \\
2 & 4 & 6 & 8 & 10 & 12
\end{array}
\tag{9.21}
$$

while the nested (twisted) approach is equivalent to reordering the unknowns as

$$
\begin{array}{cccccccc}
1 & 5 & 9 & 13 & 14 & 10 & 6 & 2 \\
17 & 21 & 25 & 29 & 30 & 26 & 22 & 18 \\
33 & 37 & 41 & 45 & 46 & 42 & 38 & 34 \\
 & & & & & & & \\
35 & 39 & 43 & 47 & 48 & 44 & 40 & 36 \\
19 & 23 & 27 & 31 & 32 & 28 & 24 & 20 \\
3 & 7 & 11 & 15 & 16 & 12 & 8 & 4
\end{array}
\tag{9.22}
$$

That is, we start to number from the four corners of the grid in an alternating manner. It is obvious that the original twisted approach leads to a process that can be carried out almost entirely in two

parallel parts, while the nested form can be done almost entirely in four parallel parts. Similarly, in three dimensions the incomplete decomposition, as well as the triangular solves, can be done almost entirely in eight parallel parts.
Van der Vorst [307, 308] reports a slight decrease in the number of iteration steps for these parallel versions, with respect to the lexicographical ordering. Duff and Meurant [109] have compared the preconditioned conjugate gradient method for a large number of different orderings, such as nested dissection and red-black, zebra, lexicographical, Union Jack, and nested parallel orderings. The nested parallel orderings are among the most efficient; thus, they are good candidates even for serial computing, and parallelism here comes as a free bonus.

At first sight there might be some problems for parallel vector processors since, in the orderings we have just sketched, the subsystems are in lexicographical ordering and hence not completely vectorizable. Of course, these subgroups could be reordered diagonally:

$$\begin{array}{ccccccccccc}
1 & 5 & 13 & . & . & . & . & . & 14 & 6 & 2 \\
9 & 17 & . & . & . & . & . & . & . & 18 & 10 \\
21 & . & . & . & . & . & . & . & . & . & 22 \\
. & . & . & . & . & . & . & . & . & . & . \\
23 & . & . & . & . & . & . & . & . & . & 24 \\
11 & 19 & . & . & . & . & . & . & . & 20 & 12 \\
3 & 7 & 15 & . & . & . & . & . & 16 & 8 & 4
\end{array} \tag{9.23}$$

which then leads to vector code as shown in Section 9.0.12. The disadvantage is that in practical situations the vector lengths will only be small on average. In [308], alternative orderings are suggested, based on carrying out the twisted factorization in a diagonal fashion. For example, in the two-dimensional situation the ordering could be

$$\begin{array}{cccccccccc}
1 & 3 & 7 & 13 & . & . & . & . & . & . \\
5 & 9 & 15 & . & . & . & . & . & . & . \\
11 & 17 & . & . & . & . & . & . & . & . \\
19 & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & 20 \\
. & . & . & . & . & . & . & . & 18 & 12 \\
. & . & . & . & . & . & . & 16 & 10 & 6 \\
. & . & . & . & . & . & 14 & 8 & 4 & 2
\end{array} \tag{9.24}$$

which leads to a process that can be done almost entirely in parallel (except for the grid diagonal in the middle, which is coupled to both groups), and each group can be done in vector mode, just as shown in Section 9.0.12. Of course, this can be generalized to three dimensions, leading to four parallel processes, each vectorizable. The twisted factorization approach can also be combined with the hyperplane approach in Section 9.0.12.

In [307] it has been mentioned that these twisted incomplete factorizations can be implemented in the efficient manner proposed by Eisenstat [144] (see also Section 9.0.10), since they satisfy the requirement that the entries in corresponding locations in A be equal to the off-diagonal entries of $\tilde{S}$ and $\tilde{T}$ in

$$K = ((\tilde{S} + D)D^{-1})(D + \tilde{T}), \tag{9.25}$$

with $\tilde{S} + D = SD^{-1/2}$, $D^{-1/2}T = D + \tilde{T}$.

Experiences with Parallelism

Although the problem of finding efficient parallel preconditioners has not been solved at all, it may be helpful to discuss some experimental results for some of the previously discussed approaches. All of the results have been reported for the nicely structured systems coming from finite-difference discretizations of elliptic PDEs over two-dimensional and three-dimensional rectangular grids.

Radicati and Robert [261] suggest partitioning the given matrix A in (slightly) overlapping blocks along the main diagonal, as in Figure 9.5.

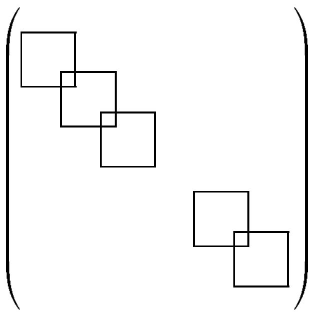

Figure 9.5: Overlapping blocks in A

Note that a given nonzero entry of A is not necessarily contained in one of these blocks. However, experience suggests that this approach is more successful if these blocks cover all the nonzero entries of A. The idea is to compute in parallel local preconditioners for all of the blocks, for example

$$A_i = L_i D_i^{-1} U_i - R_i. \tag{9.26}$$

Then, when solving $Kw = r$ in the preconditioning step, we partition r in (overlapping) parts r_i, according to A_i, and we solve the systems $L_i D_i^{-1} U_i w_i = r_i$ in parallel. Finally we define the components of w to be equal to corresponding components of the w_i's in the nonoverlapping parts and to the average of them in the overlapped parts.

Radicati and Robert [261] report on timing results obtained on an IBM 3090-600E/VF for GMRES preconditioned by overlapped incomplete LU decomposition for a two-dimensional system of order 32,400 with a bandwidth of 360. For p processors $(1 \leq p \leq 6)$ they subdivide A in p overlapping parts, the overlap being so large that these blocks cover all the nonzero entries of A. They found experimentally an overlap of about 360 entries to be optimal for their problem. This approach led to a speedup of roughly p. In some cases the parallel preconditioner was slightly more effective than the standard one, so that this method is also of interest for applications on serial computers.

Meurant [242] reports on timing results obtained with a CRAY Y-MP/832, using an incomplete repeated twisted block factorization for two-dimensional problems. In his experiments the L of the incomplete factorization has a block structure as shown below.

$$\begin{pmatrix}
X &&&&&&&&&&&&&&&&& \\
X & X &&&&&&&&&&&&&&&& \\
& X & X &&&&&&&&&&&&&&& \\
&&& X & X &&&&&&&&&&&&& \\
&&&& X & X &&&&&&&&&&&& \\
&&&&& X &&&&&&&&&&&& \\
&&&&&& X &&&&&&&&&&& \\
&&&&&& X & X &&&&&&&&&& \\
&&&&&&& X & X &&&&&&&&& \\
&&&&&&&&& X & X &&&&&&& \\
&&&&&&&&&& X & X &&&&&& \\
&&&&&&&&&&& X &&&&&& \\
&&&&&&&&&&&& X &&&&& \\
&&&&&&&&&&&& X & X &&&& \\
&&&&&&&&&&&&& X & X &&& \\
&&&&&&&&&&&&&&& X & X & \\
&&&&&&&&&&&&&&&& X & X \\
&&&&&&&&&&&&&&&&& X
\end{pmatrix}$$

Specifically, L has alternatingly a lower block diagonal, an upper one, a lower one, and (finally) an upper one. For this approach Meurant reports a speedup, for preconditioned CG, close to 6 on the 8-processor CRAY Y-MP. This speedup has been measured relative to the same repeated twisted factorization process executed on a single processor. Meurant also reports an increase in the number of iteration steps as a result of this repeated twisting. This increase implies that the effective speedup with respect to the non-parallel code is only about 4. In [78], Meurant's approach has been combined with the approach of Radicati and Robert.

For three-dimensional problems on a two-processor system we have used the blockwise twisted approach in the z-direction. That is, the (x, y)-planes in the grid were treated in parallel from bottom and top inwards. Over each plane we used ordering by diagonals, in order to achieve high vector speeds on each processor.

On a CRAY X-MP/2 this led, for a preconditioned CG, to a reduction by a factor of close to 2 in wall clock time with respect to the CPU time for the non-parallel code on a single processor. For the microtasked code, the wall clock time on the 2-processor system was measured for a dedicated system, whereas for the non-parallel code the CPU time was measured on a moderately loaded system. In some situations we even observed a reduction in wall clock time by a factor of slightly more than two, because of the better convergence properties of the twisted incomplete preconditioner.

As suggested before, we can apply the twisted incomplete factorization in a nested way. For three-dimensional problems this can be exploited by twisting also the blocks corresponding to (x, y) planes in the y-direction. Over the resulting blocks, corresponding to half (x, y) planes, we may apply diagonal ordering in order to fully vectorize the four parallel parts. With this approach we have been able to reduce the wall clock time by a factor of 3.3, for a preconditioned CG, on the 4-processor Convex C-240. In this case the total CPU time, used by all of the processors, is roughly equal to the CPU time required for single-processor execution. For more details on these and some other experiments, see [311].

As has been shown in Section 9.0.12, the hyperplane ordering can be used to realize long vector lengths in three-dimensional situations—at the expense, however, of indirect addressing. For a CYBER 205, we have demonstrated in some detail how these adverse indirect addressing effects can be circumvented. A similar approach has been followed by Berryman et al. [37] for parallelizing standard ICCG on a Connection Machine Model 2. For a 4K-processor machine they report a computational speed of 52.6 Mflop/s for the (sparse) matrix-vector product, while 13.1 Mflop/s has been realized for the preconditioner with the hyperplane approach. This reduction in speed by a factor of 4 makes it attractive to use only diagonal scaling as a preconditioner, in certain situations, for massively parallel machines like the CM-2. The latter approach has been followed by Mathur and Johnsson [234] for finite-element problems.

We have used the hyperplane ordering for preconditioned CG on an Alliant FX/4, for three-dimensional systems with dimensions $n_x = 40, n_y = 39$, and $n_z = 30$. For 4 processors this led to a speedup of 2.61, compared with a speedup of 2.54 for the CG process with only diagonal scaling as a preconditioner. The fact that both speedups are far below the optimal value of 4 must be attributed to cache effects. These cache effects can be largely removed by using the reduced system approach suggested by Meier and Sameh [236]. However, for the three-dimensional systems that we have tested, the reduced system approach led, on the average, to about the same CPU times as for the hyperplane approach on Alliant FX/8 and FX/80 computers.

Chapter 10

Linear Eigenvalue Problems $Ax = \lambda x$

In this section, we discuss the solution of the standard linear eigenvalue problem

$$Ax = \lambda x.$$

We shall need to develop some brief background material before discussing large scale problems.

10.0.13 Theoretical background and notations

A short discussion of the mathematical structure of the eigenvalue problem is necessary to fix notation and introduce ideas that lead to an understanding of the behavior, strengths and limitations of the algorithms. In this discussion, the real and complex number fields are denoted by **R** and **C** respectively. The set of numbers $\sigma(A) \equiv \{\lambda \in \mathbf{C} : rank(A - \lambda I) < n)\}$ is called the *spectrum* of A. The elements of this discrete set are the *eigenvalues* of A and they may be characterized as the n roots of the *characteristic polynomial* $p_A(\lambda) \equiv det(\lambda I - A)$.
Corresponding to each distinct eigenvalue $\lambda \in \sigma(A)$ is at least one nonzero vector x such that $Ax = \lambda x$. This vector is called a *right eigenvector* of A corresponding to the eigenvalue λ. The pair (x, λ) is called an *eigenpair.* A nonzero vector y such that $y^*A = \lambda y^*$ is called a *left eigenvector.* The multiplicity $n_a(\lambda)$ of λ as a root of the characteristic polynomial is the *algebraic* multiplicity and the dimension $n_g(\lambda)$ of $Null(\lambda I - A)$ is the *geometric* multiplicity of λ. A matrix is *defective* if $n_g(\lambda) < n_a(\lambda)$ and otherwise A is *non-defective.* The eigenvalue λ is *simple* if $n_a(\lambda) = 1$.

A subspace $\mathcal{S}$ of $\mathbf{C^{n\times n}}$ is called an *invariant subspace* of A if $A\mathcal{S} \subset \mathcal{S}$. It is straightforward to show if $A \in \mathbf{C^{n\times n}}$, $X \in \mathbf{C^{n\times k}}$ and $H \in \mathbf{C^{k\times k}}$ satisfy

$$AV = VH, \tag{10.1}$$

then $\mathcal{S} \equiv \mathrm{R}ange(V)$ is an invariant subspace of A. Moreover, if V has full column rank k then the columns of V form a basis for this subspace and $\sigma(H) \subset \sigma(A)$. If $k = n$ then $\sigma(H) = \sigma(A)$ and A is said to be *similar* to H. A is *diagonalizable* if it is similar to a diagonal matrix and this property is equivalent to A being non-defective.

The most successful numerical algorithms for eigenvalue problems are based upon the Schur decomposition. It states that every square matrix is *unitarily similar* to an upper triangular matrix. In other words, any linear operator (on a finite dimensional space) can be represented as an upper triangular matrix with an appropriate choice of basis.

Theorem 10.0.1 *(Schur Decomposition). Every square matrix A may be decomposed in the form*

$$AQ = QR, \tag{10.2}$$

where Q is unitary and R is upper triangular. The diagonal elements of R are the eigenvalues of A.

From the Schur decomposition, it is easily seen that

- A matrix A is normal ($AA^* = A^*A$) if and only if $A = Q\Lambda Q^*$ with Q unitary and Λ diagonal.
- A matrix A is Hermitian ($A = A^*$) if and only if $A = Q\Lambda Q^*$ with Q unitary and Λ is diagonal with real diagonal elements.

In either case the eigenvectors of A are the ortho-normal columns of Q and the eigenvalues are the diagonal elements of Λ.

10.0.14 Single-Vector Methods

Single vector methods are the simplest and most storage-efficient ways to compute a single eigenvalue and it's corresponding eigenvector. In fact, for most methods suitable for finding a few selected eigenvalues, the eigenvectors are the principal objects of the computation and the eigenvalues sort of "come along for the ride."

We have already seen the most fundamental algorithm of this type, the simple Power Method (Section ??). The drawbacks of this method are, as pointed out previously, potentially slow convergence, inability to compute other than the dominant eigenvalue, inability to compute more than one eigenpair at a time.

There is a simple and very effective cure for the first two ills, however it can be quite expensive. Instead of working with the original matrix A, we can introduce the *Spectral Transformation*

$$A_\sigma = (A - \sigma I)^{-1}.$$

The important fact that makes this transformation worthwhile is

$$Ax = \lambda x \quad \Longleftrightarrow \quad A_\sigma x = \frac{1}{\lambda - \sigma} x$$

and thus if ν is an eigenvalue of A_σ corresponding to eigenvector x then $\lambda = \sigma + \frac{1}{\nu}$ is an eigenvalue of the original matrix A and x is the corresponding eigenvector. It is easily seen that the dominant eigenvalue ν of A_σ will yield the eigenvalue λ of A that is closest to σ. Moreover, the rate of convergence for the power method applied to A_σ increases dramatically as the shift σ is taken to be closer and closer to a desired eigenvalue λ. In fact, for a given σ the rate of linear convergence is

$$\frac{\lambda_1 - \sigma}{\lambda_2 - \sigma},$$

where λ_1 is the closest and λ_2 is the next closest eigenvalue of A to the shift σ.

The down-side to this spectacular improvement in convergence is that an application of A_σ involves $(A - \sigma I)^{-1}$. Of course, this inverse is never explicitly computed. Instead, a sparse direct factorization of $A - \sigma I = LU$ is computed once and re-used throughout the iteration (Here, the permutations have been incorporated into the L and U). Thus, the operation $w \leftarrow A_\sigma v$ is replaced by

$$\text{Solve} \quad Lz = v; \quad \text{Solve} \quad Uw = z;$$

Occasionally, the value of σ might be updated to increase the rate of convergence once a few digits of accuracy have been obtained in the desired eigenvalue. The ultimate rate of convergence is achieved by replacing σ at each iteration with the updated approximation to the eigenvalue λ. This ultimate single vector scheme is known as *Inverse Iteration*. Its rate of convergence is *quadratic* in general and *cubic* when A is Hermitian.

$$\text{General: } |\lambda^{(j)} - \lambda| < C|\lambda^{(j-1)} - \lambda|^2, \text{Hermitian: } |\lambda^{(j)} - \lambda| < C|\lambda^{(j-1)} - \lambda|^3.$$

However, each time σ is changed, a new factorization is required.

This spectral tranformation is fundamental to the success of the most effective algorithms. However, for many practical problems it is impossible to factor the matrix $A - \sigma I$. In these cases, we may wish to substitute an iterative method to solve the linear systems. Unfortunately, there can be many pitfalls to this approach.

10.0.15 The QR Algorithm

At the other algorithmic extreme are methods for finding all of the eigenvalues and vectors of a given matrix. Typically, these are unsuitable for large problems because they involve a sequence of dense similarity transformations that quickly destroy any sparsity or structure of the original matrix A. Nevertheless, it is important to understand these methods to see if there is some sort of middle ground for algorithms that may find a selected subset of the eigenvalues and eigenvectors of A.

Certainly, when it is possible to use it, the well known Implicitly Shifted QR-Algorithm [**?**] is the method of choice as a general algorithm for finding all of the eigenvalues and vectors. This algorithm is closely tied to the Shur decomposition. In fact, it produces a sequence of unitary similarity transformations that iteratively reduce A to upper triangular form. The algorithm begins with an initial unitary similarity transformation of A to the condensed form $AV = VH$ where H is upper Hessenberg (tridiagonal in case $A = A^*$) and V is unitary. Then the iteration shown in Figure 10.1 is performed.

Factor $AV = VH$
for $j = 1, 2, 3..,$**until** convergence
    $\mu = select\_shift(H)$;
    Factor $QR = H - \mu I$;
    $H \leftarrow Q^* H Q; \quad V \leftarrow VQ;$
end

Figure 10.1: The Shifted QR Method

In this schemen Q is unitary and R is upper triangular (i.e. the QR factorization of $H - \mu I$). It is easy to see that H is unitarily similar to A throughout the course of this iteration. The iteration is continued until the subdiagonal elements of H converge to zero, i.e. until a Schur decomposition has been (approximately) obtained. In the standard implicitly shifted QR-iteration, the unitary matrix Q is never actually formed. It is computed indirectly as a product of 2×2 Givens or 3×3 Householder transformations through a “bulge chase” process. The elegant details of an efficient and stable implementation would be too much of a digression here. They may be found in [178]. The convergence behavior of this iteration is fascinating. The columns of V converge to Schur vectors at various rates. These rates are fundamentally linked to the simple power method and its rapidly convergent variant, inverse iteration (see [**?**]).

It is worth noting some detail about the convergence of the leading and trailing columns of the matrix V as the iteration proceeds. If we denote the leading column by $v_1^{(j)}$ and the trailing column by $v_n^{(j)}$, then

$$v_1^{(j+1)} = (A - \mu_j I)v_1^{(j)}/\rho_{11}^{(j)} \quad \text{and} \quad v_1^{(j+1)} = (\bar{A} - \bar{\mu_j} I)^{-1} v_1^{(j)} \bar{\rho}_{nn}^{(j)},$$

where $\rho_{11}^{(j)}$ and $\rho_{nn}^{(j)}$ are the leading and trailing diagonal elements of R. Thus, the leading column of V is converging with a (modified) power-method-like rate and with appropriate shift selection, the trailing column is converging with an inverse iteration rate. Clearly, if one can factor the entire matrix in this way, the latter situation must be exploited. However, as we shall see, the power like convergence taking place in the leading columns may also be exploited in a way that is very effective for large scale problems and parallel computation.

Despite the extremely fast rate of convergence and the efficient use of storage, the implicitly shifted QR method is not suitable for large scale problems and it has proved to be extremely difficult to parallelize. Large scale problems are typically sparse or structured so that a matrix-vector product $w \leftarrow Av$ may be computed with time and storage proportional to n rather than n^2 . A method based upon full similarity transformations quickly destroys this structure. Storage and operation counts become order n^2. Hence, there is considerable motivation for methods that only require matrix-vector products with the original A. Fortunately, as we shall soon see, the power like convergence taking place in the leading columns of the shifted QR iteration may exploited in a way that is very effective for large scale problems.

10.0.16 Subspace Projection Methods

We have already seen the rudiments of Krylov subspace projection in Chapter **??**. Certainly, some of the most effective methods available for computing a few eigen-pairs of a large matrix or matrix pencil are based on these ideas. In the case of the standard problem $Ax = \lambda x$, Krylov subspace projection results in the Lanczos/Arnoldi class of methods. We have already hinted at how these methods may be viewed as systematic ways to extract additional eigen-information from the sequence of vectors produced by a power iteration.

If one hopes to obtain additional information through various linear combinations of the power sequence, it is natural to formally consider the *Krylov* subspace

$$\mathcal{K}_k(A, v_1) = \text{Span } \{v_1, Av_1, A^2v_1, \ldots, A^{k-1}v_1\}$$

and to attempt to formulate the best possible approximations to eigenvectors from this subspace.

Approximate eigenpairs are constructed from this subspace by imposing a Galerkin condition. Given any k dimensional subspace $\mathcal{S}$ of $\mathbf{C}^n$ we define a vector $x \in \mathcal{S}$ to be a a *Ritz vector* with corresponding *Ritz value* θ if the Galerkin condition

$$< w, Ax - \theta x >= 0 \ , \quad \text{for all} \quad w \in \mathcal{S}$$

is satisfied with $< \cdot, \cdot >$ denoting some inner product on $\mathbf{C}^n$. In this setting, we are interested in $\mathcal{S} = \mathcal{K}_k(A, v_1)$. More general subspaces will be considered later.

From its definition, we see that every $w \in \mathcal{K}_k$ is of the form $w = \phi(A)v_1$ for some polynomial ϕ of degree less than k and also that $\mathcal{K}_{j-1} \subset \mathcal{K}_j$ for $j = 2, 3, \cdots, k$. Thus, if we have constructed a sequence of orthogonal bases $V_j = [v_1, v_2, \cdots, v_j]$ with $\mathcal{K}_j = Range(V_j)$ and $V_j^* V_j = I_j$, then it is fairly straightforward to see that $v_j = p_{j-1}(A)v_1$ where p_{j-1} is a polynomial of degree $j - 1$. To extend the basis for $\mathcal{K}_k$ to one for $\mathcal{K}_{k+1}$ we must construct a new vector that has a component in the direction of $A^k v_1$ and then orthogonalize this with respect to the previous basis vectors. The only basis vector available with a component in the direction of $A^{k-1}v_1$ is v_k and thus a convenient way to obtain the direction of the new vector v_{k+1} will be given by

$$\begin{aligned} f_k &= Av_k - V_k h_k, \\ v_{k+1} &= f_k / \|f_k\|, \end{aligned}$$

where the vector h_k is constructed to achieve $V_k^* f_k = 0$. Of course, the orthogonality of the columns of V_k gives the formula $h_k = V_k^* A v_k$.

This construction provides a crucial fact concerning f_k:

$$\|f_k\| = \min_h \|Av_k - V_k h\| = \min \|p(A)v_1\|,$$

where the second minimization is over all polynomials p of degree k with the same leading coefficient as p_{k-1} (i.e.$\lim_{\tau \to 0} \frac{\tau p_{k-1}(\tau)}{p(\tau)} = 1$ where $v_k = p_{k-1}(A)v_1$.)

The only opportunity for failure here is when $f_k = 0$. However, when this happens it implies that

$$AV_k = V_k H_k$$

where $H_k = V_k^* A V_k = [h_1, h_2, \cdots, h_k]$ (with a slight abuse of notation). Hence, this "good breakdown" happens precisely when $\mathcal{K}_k$ is an invariant subspace of A.

Of course, we must ask the question: When can $f_k = 0$ happen? Well, if $v1 = \sum_{i=1}^k q_j \gamma_j$ where $Aq_j = q_j \lambda_j$, then $p(A)v_1 = 0$ with $p(\tau) = \prod_{i=1}^k (\tau - \lambda_j)$. Since this polynomial can be normalized to have the same leading coefficient as p_{k-1}, the minimization property implies that $f_k = 0$. A more precise answer is

Lemma 10.0.2 *$f_k = 0$ if and only if $v_1 = Q_k y$ with $AQ_k = Q_k R_k$ a partial Schur decomposition of A. Moreover, the Ritz values of A with respect to $\mathcal{K}_k$ are eigenvalues of A and are given by the diagonal elements of R_k.*

10.0.17 The Arnoldi Factorization

The construction we have just derived provides a relation between the matrix A, the basis matrix V_k and the residual vector f_k of the form of the form

$$AV_k = V_k H_k + f_k e_k'$$

where $V_k \in \mathbf{C}^{n\times k}$ has orthonormal columns, $V_k^* f_k = 0$ and $H_k \in \mathbf{C}^{k\times k}$ is upper Hessenberg with non-negative subdiagonal elements. We shall call this a *k-step Arnoldi Factorization* of A. It is easily seen from the construction that $H_k = V_k^* A V_k$ is upper Hessenberg. When A is Hermitian this implies H_k is real, symmetric and tridiagonal and the relation is called a *k-step Lanczos Factorization* of A. The columns of V_k are referred to as the *Arnoldi vectors* or *Lanczos vectors* , respectively.

The discussion of the previous section implies that Ritz pairs satisfying the Galerkin condition are immediately available from the eigenpairs of the small projected matrix H_k.

If $H_k s = \theta s$ then the vector $x = V_k s$ satisfies

$$\|Ax - \theta x\| = \|(AV_k - V_k H_k)s\| = |\beta_k e_k' s|.$$

Observe that if (x, θ) is a Ritz pair then

$$\theta = s^* H_k s = (V_k s)^* A (V_k s) = x^* A x$$

is a Rayleigh quotient (assuming $\|s\| = 1$) and the associated Rayleigh quotient residual $r(x) \equiv Ax - x\theta$ satisfies

$$\|r(x)\| = |\beta_k e_k' s|.$$

When A is Hermitian, this relation may be used to provide computable rigorous bounds on the accuracy of the eigenvalues of H_k as approximations to eigenvalues [**?**] of A. When A is non-Hermitian, we can only say that the residual is small if $|\beta_k e_k' s|$ is small without further information. In any case, if $f_k = 0$ these Ritz pairs become exact eigenpairs of A.

The explicit steps needed to form a k-Step Arnoldi factorization are given in Figure 10.2.

The dense matrix-vector products $V_{j+1}^* w$ and $V_{j+1} h$ may be expressed with the Level 2 BLAS operation _GEMV. As discussed previously in Chapter **??**, this provides a significant performance

$$
\begin{array}{l}
v_1 = v/\|v\|; \\
w = Av_1; \;\; \alpha_1 = v_1^* w; \\
f_1 \leftarrow w - v_1\alpha_1; \\
V_1 \leftarrow (v_1);\; H_1 \leftarrow (\alpha_1); \\
\text{for } j = 1, 2, 3, \ldots \text{k-1}, \\
\quad \beta_j = \|f_j\|;\; v_{j+1} \leftarrow f_j/\beta_j; \\
\quad V_{j+1} \leftarrow (V_j, v_{j+1}); \\
\quad \widehat{H}_j \leftarrow \begin{pmatrix} H_j \\ \beta_j e_j' \end{pmatrix}; \\
\quad w \leftarrow Av_{j+1}; \\
\quad h \leftarrow V_{j+1}^* w;\; f_{j+1} \leftarrow w - V_{j+1}h; \\
\quad H_{j+1} \leftarrow (\widehat{H}_j, h); \\
\text{end}
\end{array}
$$

Figure 10.2: k-Step Arnoldi Factorization

advantage on virtually every platform from workstation to super-computer. Moreover, considerable effort has been made within the ScaLapack project to optimize these kernels for a variety of parallel machines.

The mechanism used here to orthogonalize the new information Av_k against the existing basis V_k is the Classical Gram Schmidt process (CGS). It is notoriously unstable and will fail miserably in this setting without modifcation. One remedy is to use the Modified Gram Schmidt process (MGS). Unfortunately, this will also fail to produce orthogonal vectors in the resarting situation we are about to discuss and it cannot be expressed with Level 2 Blas in this setting. Fortunately, the CGS method can be rescued through a technique proposed by Daniel, Gragg, Kaufman and Stewart (DGKS) in [?]. This scheme provides an excellent way to construct a vector f_{j+1} that is numerically orthogonal to V_{j+1}. It amounts to computing a correction

$$c = V_{j+1}^* f_{j+1}; \quad f_{j+1} \leftarrow f_{j+1} - V_{j+1}c; \quad h \leftarrow h + c;$$

just after the initial CGS step if necessary. A simple test is used to avoid this DGKS correction if it is not needed. This mechanism maintains orthogonality to full working precision at very reasonable cost. The special situation imposed by the restarting scheme we are about to discuss makes this modification essential for obtaining accurate eigenvalues and numerically orthogonal Schur vectors (eigenvectors in the Hermitian case). The schemes mentioned earlier in Section ?? based on MGS are sufficient for solving linear systems but not for the situations encountered with eigenvalue calculations.

Failure to maintain orthogonality leads to several numerical difficulties. In the Hermitian case, Paige [?] showed that the occurs precisely when an eigenvalue of H_j is close to an eigenvalue of A. In fact, the Lanczos vectors lose orthogonality in the direction of the associated approximate eigenvector. Moreover, failure to maintain orthogonality results in spurious copies of the approximate eigenvalue produced by the Lanczos method. Implementations based on selective and partial orthogonalization [?, ?, ?] monitor the loss of orthogonality and perform additional orthogonalization steps only when necessary. The methods developed in [?, ?, ?] use the with no re-orthogonalization steps. Once a level of accuracy has been achieved, the spurious copies of computed eigenvalues are located and deleted. Then the Lanczos basis vectors are re-generated from the 3-term recurrence and Ritz vectors are recursively constructed in place. This is a very comptetitive strategy when the matrix vector product $w \leftarrow Av$ is relatively inexpensive.

10.0.18 Restarting the Arnoldi Process

An unfortunate aspect of the Lanczos/Arnoldi process is that there is no way to determine in advance how many steps will be needed to determine the eigenvalues of interest within a specified accuracy. We have tried to indicate with our brief theoretical discussion that the eigen-information obtained through this process is completely determined by the choice of the starting vector v_1. Unless there is a very fortuitis choice of v_1, eigen-information of interest probably will not appear until k gets very large. Clearly, it becomes intractable to maintain numerical orthogonality of V_k. Extensive storage will be required and repeatedly finding the eigensystem of H_k also becomes intractable at a cost of $\mathcal{O}(k^3)$ flops.

The obvious need to control this cost has motivated the development of restarting schemes. Restarting means replacing the starting vector v_1 with an "improved" starting vector v_1^+ and then computing a new Arnoli factorization with the new vector. Our brief theoretical discussion about the structure of f_k serves as a guide: Our goal is to iteratively force v_1 to be a linear combination of eigenvectors of interest. A more general and, in fact, a better numerical strategy, is to force the starting vector to be a linear combination of Schur vectors that span the desired invariant subspace.

Explicit Restarting

The need for restarting was recognized early on by Karush [?] soon after the appearance of the original algorithm of Lanczos [?]. Subsequently, there were developments by Paige [?], Cullum and Donath [?], and Golub and Underwood [?]. More recently, a restarting scheme for eigenvalue computation was proposed by Saad based upon the polynomial acceleration scheme introduced by Manteuffel [?] for the iterative solution of linear systems. Saad [?] proposed to restart the factorization with a vector that has been preconditioned so that it is more nearly in a k-dimensional invariant subspace of interest. This preconditioning is in the form of a polynomial in A applied to the starting vector that is constructed to damp unwanted components from the eigenvector expansion. The idea is to force the starting vector to be a member of an invariant subspace. An iteration is defined by a repeatedly restarting until the updated Arnoldi factorization eventually contains the desired information. For more information on the selection of effective restarting vectors, see [272]. One of the more successful approaches is to use Chebyshev polynomials in order to damp unwanted eigenvector components in the available subspace. Explicit restarting techniques are easily parallelized, in contrast to the overhead involved in implicit restarting (next section). The reason is that major part of the work is in matrix vector products. In the situation that one has to solve the eigenproblem on a massively parallel computer for a matrix that involves cheap matrix vector products, this may be an attractive property.

10.0.19 Implicit Restarting

There is another approach to restarting that offers a more efficient and numerically stable formulation. This approach called *implicit restarting* is a technique for combining the implicitly shifted QR scheme with a k-step Arnoldi or Lanczos factorization to obtain a truncated form of the implicitly shifted QR-iteration.
The numerical difficulties and storage problems normally associated with Arnoldi and Lanczos processes are avoided. The algorithm is capable of computing a few (k) eigenvalues with user specified features such as largest real part or largest magnitude using $2nk + \mathcal{O}(k^2)$storage. The computed Schur basis vectors for the desired k-dimensional eigen-space are numerically orthogonal to working precision.

Implicit restarting provides a means to extract interesting information from large Krylov subspaces while avoiding the storage and numerical difficulties associated with the standard approach. It does this by continually compressing the interesting information into a fixed size k-dimensional subspace. This is accomplished through the implicitly shifted QR mechanism. An Arnoldi factorization of length $m = k + p$

$$AV_m = V_m H_m + f_m e_m', \tag{10.3}$$

is compressed to a factorization of length k that retains the eigen-information of interest. This is accomplished using QR steps to apply p shifts implicitly. The first stage of this shift process results in

$$AV_m^+ = V_m^+ H_m^+ + f_m e_m' Q, \tag{10.4}$$

where $V_m^+ = V_mQ$, $H_m^+ = Q^*H_mQ$, and $Q = Q_1Q_2\cdots Q_p$. Each Q_j is the orthogonal matrix associated with the shift μ_j used during the shifted QR algorithm. Because of the Hessenberg structure of the matrices Q_j, it turns out that the first $k-1$ entries of the vector $e_m'Q$ are zero (i.e. $e_m'Q = (\sigma e_k', \widehat{q}^*)$). This implies that the leading k columns in equation (10.4) remain in an Arnoldi relation. Equating the first k columns on both sides of (10.4) provides an updated $k-$step Arnoldi factorization

$$AV_k^+ = V_k^+H_k^+ + f_k^+e_k', \tag{10.5}$$

with an updated residual of the form $f_k^+ = V_m^+e_{k+1}\widehat{\beta}_k + f_m\sigma$. Using this as a starting point it is possible to apply p additional steps of the Arnoldi process to return to the original m-step form.

There are many ways to select the shifts $\{\mu_j\}$ that are applied by the QR steps. Virtually any explicit polynomial restarting scheme could be applied through this implicit mechanism. Considerable success has been obtained with the choice of *exact shifts* . This selection is made by sorting the eigenvalues of H_m into two disjoint sets of k "wanted" and p "unwanted" eigenvalues and using the p unwanted ones as shifts. With this selection the p shift applications result in H_k^+ having the k wanted eigenvalues as its spectrum. As convergence takes place, the subdiagonals of H_k tend to zero and the most desired eigenvalue approximations appear as eigenvalues of the leading $k \times k$ block of R in a Schur decompositon of A. The basis vectors V_k tend to orthogonal Schur vectors.

An alternate interpretation stems from the fact that each of these shift cycles results in the implicit application of a polynomial in A of degree p to the starting vector.

$$v_1 \leftarrow \psi(A)v_1 \quad \text{with} \quad \psi(\lambda) = \prod_{j=1}^{p}(\lambda - \mu_j). \tag{10.6}$$

The roots of this polynomial are the shifts used in the QR process and these may be selected to enhance components of the starting vector in the direction of eigenvectors corresponding to desired eigenvalues and damp the components in uwanted directions. Of course, this is desirable because it forces the starting vector into an invariant subspace associated with the desired eigenvalues. This in turn forces f_k to become small and hence convergence results. Full details may be found in [**?**].

An intuitive notion of how this repeated updating of the starting vector v_1 through implicit restarting results leads to convergence. If v_1 is expressed as a linear combination of eigenvectors $\{q_j\}$ of A, then

$$v_1 = \sum_{j=1}^{n} q_j\gamma_j \Rightarrow \psi(A)v_1 = \sum_{j=1}^{n} q_j\psi(\lambda_j)\gamma_j.$$

Applying the same polynomial (i.e. using the same shifts) repeatedly for ℓ iterations will result in

Compute an $m = k + p$ step Arnoldi factorization
    $AV_m = V_m H_m + f_m e_m'$.
repeat until "convergence",
    Compute $\sigma(H_m)$ and select p
    shifts $\mu_1, \mu_2, ...\mu_p$
    $q^* \leftarrow e_m'$;
    for $j = 1, 2, ..., p$,
        Factor $[Q, R] = \mathrm{q}r(H_m - \mu_j I)$;
        $H_m \leftarrow Q^* H_m Q$; $V_m \leftarrow V_m Q$;
        $q^* \leftarrow q^* Q$;
    end
    $f_k \leftarrow v_{k+1}\widehat{\beta}_k + f_m \sigma_k$;
    $V_k \leftarrow V_m(1:n, 1:k)$; $H_k \leftarrow H_m(1:k, 1:k)$;
    Beginning with the k-step Arnoldi factorization
        $AV_k = V_k H_k + f_k e_k'$,
    apply p additional steps of the Arnoldi process
    to obtain a new m-step Arnoldi factorization
        $AV_m = V_m H_m + f_m e_m'$.
end

Figure 10.3: Implicitly Restarted Arnoldi Method (IRAM)

the j-th original expansion coefficient being attenuated by a factor

$$\left(\frac{\psi(\lambda_j)}{\psi(\lambda_1)}\right)^{\ell} ,$$

where the eigenvalues have been ordered according decreasing values of $|\psi(\lambda_j))|$. The leading k eigenvalues become dominant in this expansion and the remaining eigenvalues become less and less significant as the iteration proceeds. Hence, the starting vector v_1 is forced into an invariant subspace as desired. The adaptive choice provided with the exact shift mechanism further enhances the isolation of the wanted components in this expansion. Hence, the wanted eigenvalues are approximated better and better as the iteration proceeds.
The basic iteration is listed in Figure 10.3 .

It is worth noting that if $m = n$ then $f_m = 0$ and this iteration is precisely the same as the Implicitly Shifted QR iteration. Even for $m < n$, the first k columns of V_m and the Hessenberg submatrix $H_m(1 : k, 1 : k)$ are mathematically equivalent to the matrices that would appear in the full Implicitly Shifted QR iteration using the same shifts μ_j. In this sense, the Implicitly Restarted Arnoldi method may be viewed as a truncation of the Implicitly Shifted QR iteration. The fundamental difference is that the standard Implicitly Shifted QR iteration selects shifts to drive subdiagonal elements of H_n to zero from the bottom up while the shift selection in the Implicitly Restarted Arnoldi method is made to drive subdiagonal elements of H_m to zero from the top down.

There are important implementation details concerning the deflation (setting to zero) of subdiagonal elements of H_m and the purging of unwanted but converged Ritz values. These details are quite important for a robust implementation but they are beyond the scope of this discussion. Complete details of these numerical refinements may be found in [**?**, **?**].

This implicit scheme costs p rather than the $k+p$ matrix-vector products the explicit scheme would require. Thus the exact shift strategy can be viewed both as a means to damp unwanted components from the starting vector and also as directly forcing the starting vector to be a linear combination of wanted eigenvectors. See [**?**, **?**] for information on the convergence of IRAM and [**?**, **?**] for other possible shift strategies for Hermitian A. The reader is referred to [**?**, **?**] for studies comparing implicit restarting with other schemes.

10.0.20 Lanczos' method

As previously mentioned, when A is Hermitian ($A = A^*$) then the projected matrix H is tridiagonal and the Arnoldi process reduces to the Lanczos method. Historically, the Lanczos process preceeded the Arnoldi process. In fact, the Arnoldi process was conceived primarily as a means for reducing a matrix to upper Hessenberg form through an orthogonal similarity transformation.

In this Hermitian case, if we denote the subdiagonal elements of H by $\beta_1, \beta_2, \cdots \beta_{n-1}$ and the diagonal elements by $\alpha_1, \alpha_2, \cdots \alpha_n$ then the relation

$$AV_k = V_k H_k + f_k e_k'$$

gives

$$f_k = v_{k+1}\beta_k = (A - \alpha_k)v_k - v_{k-1}\beta_{k-1}.$$

This famous three term recurrence has been studied extensively since its inception. The numerical difficulties are legendary with the two main issues being the numerical orthogonality of the sequence of basis vectors and the almost certain occurance of "spurious" copies of converged eigenvalues re-appearing as eigenvalues of the projected matrix H_k.

The most favorable form the the recurrance in the absence of any additional attempt at achieving orthogonality is

$v_1 = v/\|v\|;$
$w = Av_1; \alpha_1 = v_1^* w;$
$f_1 \leftarrow w - v_1\alpha_1;$
for $j = 1, 2, 3, ...$m-1,
    $\beta_j = \|f_j\|;$
    $v_{j+1} \leftarrow f_j/\beta_j;$
    $w \leftarrow Av_{j+1} - v_j\beta_j;$
    $\alpha_{j+1} = v_{j+1}' w;$
    $f_{j+1} \leftarrow w - v_{j+1}\alpha_{j+1};$
end

Figure 10.4: Lanczos Process

This organization amounts to the last two steps of the Modified Gram Schmidt variant of the Arnoldi process presented in Section ??. All of the other coefficients that would ordinarily appear in the Arnoldi process are zero in the Hermitian case.

Once the tridiagonal matrix H_m has been constructed, analyzed and found to posses k converged eigenvalues $\{\theta_1, \theta_2, \cdots, \theta_k\}$, with corresponding eigenvectors $Y = [y_1, y_2, \cdots, y_k]$, we may construct the eigenvectors with the following recursion.

This mechanism is quite attractive when the matrix vector product $W \leftarrow Av$ is relatively inexpensive. However, there are considerable numerical difficulties that must be overcome. Cullum and Willoughby developed schemes for analyizing the projected matrix H_m and modifying it to get rid of the spurious eigenvalue cases. Briefly, this analysis consists of deleting the first row and column of H and then comparing the Ritz values of the new H with those of the original H. Those that are the same are the spurious ones. The idea is that the 'good' Ritz values are associated with significant components in the starting vector. If you skip the first row and column, then you skip essentially the starting vector and hence you skip essential parts of the wanted eigenvectors. Consequently, if something still converges it must be spurious.

Even with this analysis, there is no assurance of numerical orthogonality of the eigenvectors.

$X = v_1 * Y(1,:);$
$w = Av_1;$
$f_1 \leftarrow w - v_1\alpha_1;$
for $j = 1, 2, 3,$ ...m-1,
  $v_{j+1} \leftarrow f_j/\beta_j;$
  $X \leftarrow X + vj + 1Y(j+1,:);$
  $f_{j+1} \leftarrow Av_{j+1} - v_{j+1}\alpha_{j+1} - v_j\beta_j;$
end

Figure 10.5: Lanczos Process

Parlett and Scott, advocate a selected orthogonalization proceedure that orthogonalizes against converged Ritz vectors as they appear. Grimes, Lewis, Simon advocate always using Shift-Invert so that the Lanczos sequence is relatively short and the separation of the transformed eigenvalues aids in the orthogonality so that a selective orthogonalization scheme is quite successful.

STILL TO DO?

- Miscellaneous: harmonic Ritz values, Monitoring convergence (Parlett-Reid), **D H**

10.0.21 Other Subspace Iteration Methods

Clearly, there is a limitation of Krylov methods when a shift-invert spectral transformation is required to accelerate convergence. If this cannot be accomplished with a sparse direct factorization then we must be prepared to obtain accurate solutions to the shift-invert equations with an iterative method. Without an accurate solution, the basic premise of the Krylov subspace is lost and along with that, all of the convergence analysis. If one desires to introduce an acceleration scheme using innacurate solutions to shift-invert equations the notion of Krylov must be abandoned. In this section we develop alternative subpace iteration ideas that admit these innacurate solves.

We have developed the Krylov subspace projection as a means of extracting more information form the power iteration sequence. However, there is a more straightforward generalization of the power method that is also aimed at extracting several eigenvectors simultaneously. We present the simplest variant here. The algorithm we consider is a straightforward generalization of the power

Factor $VR = W$ (with $W \in \mathbf{R}^{n \times k}$ arbitrary);
Set $H = 0$;
while ($\|W - VH\| > tol\|H\|$),
    $W \leftarrow AV$;
    $H = V^* W$;
    Factor $[V, R] = qr(W)$;
end

Figure 10.6: Generalized Power Method

method. This algorithm attempts to compute an invariant subspace rather than a single vector. When it converges, this iteration will have computed an invariant subspace of the real matrix A . Note that

$$H = V^H AV \quad \text{with} \quad V^H V = I$$

so H is a generalized Raleigh Quotient. Approximate eigenpairs for A are of the form (x, θ) where $Hy = y\theta$ and $x = Vy$. It is easily seen that the pairs (x, θ) are indeed Ritz pairs for A with respect to the subspace $\mathcal{S} = Range(V)$. The algorithm for the generalized power iteration is given in Figure 10.6.

The analogy to the shifted inverse power method is a straightforward modification of subspace iteration; it is shown in Figure 10.7.

It is easily checked that $H = V^* AV$ and that the Ritz pairs (x, θ) may be obtained as above. However, in this case the subpace $\mathcal{S} = Range(V)$ will be dominated by eigenvector directions corresponding to the eigenvalues nearest to the selected shifts μ_j.

The iteration becomes quite interesting when $k = n$. Suppose we constructed (using n-steps of Arnoldi say) an orthogonal similarity transformation of A to upper Hessenberg form. That is

$$AV = VH \quad \text{with} \quad V^T V = I, \quad H \text{upper Hessenberg.}$$

If $H = QR$ is the QR factorization of H then $W = (VQ)R$ is the QR factorization of $W = AV$. Moreover

$$\begin{aligned} (A - \mu I)(VQ) &= (VQ)(RQ) \\ AV_+ &= V_+ H_+ \quad \text{with} \quad V_+^T V_+ = I, \quad H_+ = RQ + \mu I. \end{aligned}$$

$$
\begin{array}{l}
\text{Factor } [V, R] = qr(W) \text{ (with } W \in \mathbf{R}^{n\times k} \text{ arbitrary);} \\
\text{Set } H = 0; \\
\text{while (} \|W - VH\| > tol\|H\| \text{),} \\
\quad \mu = \text{Select\_shift}(H); \\
\quad \text{Solve } (A - \mu)W = V; \\
\quad \text{Factor } [\widehat{W}, R] = qr(W) \ ; \\
\quad H = \widehat{W}^* V R^{-1} + \mu I; \\
\quad V \leftarrow \widehat{W}; \\
\text{end}
\end{array}
$$

Figure 10.7: Generalized Shifted Inverse Power Method

The amazing point here is that H_+ remains upper Hessenberg if H is originally upper Hessenberg. Moreover, the QR factorization of H by Givens' method and the associated updating $V_+ = VQ$ requires $O(n^2)$ flops rather than $O(n^3)$ for a dense Q-R factorization. This leads to directly to the QR algorithm presented in 10.1.

The important observation to make with this iteration is that the construction of the subspace is divorced from the construction of Ritz vectors. Therefore, we could just as well solve $(A-\mu)W = V$ with an iterative method (perhaps inaccurately) and then get H directly with $H \leftarrow \widehat{W}^* A\widehat{W}$. Thus, by giving up Krylov we lose efficiency of having eigen pair approximations and associated error estimates available directly from H but we gain the potentially tremendous advantage of admitting acceleration schemes without accurate solves.

We can also construct other sets of orthonormal vectors, identifying convenient subspaces for the restriction of the matrix A. Davidson's method is based upon this idea.

10.0.22 Davidson's method

The main idea behind Davidson's method is the following one. Suppose we have some subspace K of dimension k, over which the projected matrix A has a Ritz value θ_k (e.g., θ_k is the largest Ritz value) and a corresponding Ritz vector u_k. Let us assume that an orthogonal basis for K is given by the vectors v_1, v_2, ..., v_k.

Now we want to find a successful update for u_k, in order to expand our subspace. To that end we compute the defect $r = Au_k - \theta_k u_k$. Then Davidson, in his original paper [70], suggests to compute t from $(D_A - \theta_k I)t = r$, where D_A is the diagonal of the matrix A. The vector t is made orthogonal to the basis vectors v_1, ..., v_k, and the resulting vector is chosen as the new v_{k+1}, by which K is expanded.

It has been reported that this method can be quite successful in finding dominant eigenvalues of (strongly) diagonally dominant matrices. The matrix $(D_A - \theta_k I)^{-1}$ can be viewed as a preconditioner for the vector r. Davidson [71] suggests that his algorithm (more precisely: the Davidson-Liu variant of it) may be interpreted as a Newton-Raphson scheme, and this has been used as an argument to explain its fast convergence. It is tempting to see the preconditioner also as an approximation for $(A - \theta_k I)^{-1}$, and, indeed, this approach has been followed for the construction of more complicated preconditioners (see, e.g., [68, 244, 245]). However, note that $(A - \theta_k I)^{-1}$ would map r onto u_k, and hence it would not lead to an expansion of our search space. Clearly this is a wrong interpretation for the preconditioner.

10.0.23 The Jacobi-Davidson iteration method

In this section we admit the matrix A to be complex, and in order to express this we use the notation v^* for the complex conjugate of a vector (if complex), or the transpose (if real), and likewise for matrices.

In Davidson's method the residual vector is preconditioned in order to expand the search space. In the Jacobi-Davidson method we want to find the orthogonal complement for our current approximation u_k with respect to the desired eigenvector u of A. Therefore, we are interested to see explicitly what happens in the subspace $u_k^{\perp}$.

The orthogonal projection of A onto that space is given by $B = (I - u_k u_k^*)A(I - u_k u_k^*)$ (we assume that u_k has been normalized).

The correction v for u_k should satisfy:

$$(B - \theta I)v = -r \quad \text{and} \quad v \perp u_k. \tag{10.7}$$

Note that we are free to use any method for the (approximate) solution of (10.7) and that it is not necessary to require diagonally dominance of B (or A).

The algorithm for the improved Davidson method then becomes as displayed in Figure 10.8. We have skipped indices for variables that overwrite old values in an iteration step, e.g., u instead of u_k).

Now we will discuss convenient ways for the approximate solution of

$$(I - u_k u_k^*)(A - \theta_k I)(I - u_k u_k^*)t = -r \;\text{ and }\; t \perp u_k. \tag{10.8}$$

compute $z = v_1 = v/||v||_2$ for some initial guess v
$w_1 = Av_1$, $\theta = m_{1,1} = (w_1, v_1)$, $r = w_1 - \theta u$
    for $k = 2, ...$
        Solve (approximately) a $t \perp u$ from
            $(I - zz^T)(A - \theta I)(I - zz^T)t = r$
        for $j = 1, .., k-1$
            $t = t - (t, v_j)v_j$
        $v_k = t/||t||_2$, $w_k = Av_k$
        for $j = 1, .., k$
            $m_{j,k} = (w_k, v_j)$
        Compute the largest eigenpair θ, s
        of the symmetric matrix M_k, ($||s||_2 = 1$)
        $(s = (\sigma_1, ..., \sigma_k)^T)$
        $z = \sigma_1 v_1 + \cdots + \sigma_k v_k$
        $\widehat{z} = \sigma_1 w_1 + \cdots + \sigma_k w_k$
        $r = \widehat{z} - \theta\, z$
        Stop if $||r||_2 \leq \varepsilon$

Figure 10.8: The Jacobi-Davidson method for the computation of $\lambda_{\max}(A)$

Since $t \perp u_k$, it follows from (10.8) that

$$(A - \theta_k I)t - \varepsilon u_k = -r \tag{10.9}$$

or

$$(A - \theta_k I)t = \varepsilon u_k - r$$

When we have a suitable preconditioner $M \approx A - \theta_k I$ available, then we can compute an approximation $\tilde{t}$ for t:

$$\tilde{t} = \varepsilon M^{-1} u_k - M^{-1} r. \tag{10.10}$$

The value of ε is determined by the requirement that $\tilde{t}$ should be orthogonal with respect to u_k:

$$\varepsilon = \frac{u_k^* M^{-1} r}{u_k^* M^{-1} u_k}. \tag{10.11}$$

Equation (10.10) leads to several interesting observations:

1. If we choose $\varepsilon = 0$ then we obtain the Davidson method (with preconditioner M). In this case $\tilde{t}$ will not be orthogonal to u_k.

2. If we choose ε as in (10.11) then we have a Jacobi-Davidson method. Note that this method requires two operations with the preconditioning matrix per iteration, but this can be relaxed if we use a Krylov subspace solver for the approximate solution of the correction equation.

3. If $M = A - \theta_k I$, then (10.10) reduces to

$$t = \varepsilon (A - \theta_k I)^{-1} u_k - u_k.$$

   Since t is made orthogonal to u_k afterwards, this choice is equivalent with $t = (A - \theta_k I)^{-1} u_k$. In this case the method is just mathematically equivalent with (accelerated) shift and invert iteration (with optimal shift).

If we solve (10.8) approximately with a preconditioned iterative method, like Bi-CGSTAB or GMRES, then we only need two preconditioning operations for the first iteration step of this solver, as we will explain now.
The most common situation is that we have a preconditioner K for $A - \mu I$, where μ is a value close enough to the desired eigenvalue. Of course, we may select $\mu = \theta_k$, but the construction of a new preconditioner for each value of θ_k may not pay off. The preconditioner also has to be restricted to the subspace orthogonal to u_k, that is we have to work with the restricted preconditioner

$$(I - u_k u_k^*) K (I - u_k u_k^*).$$

If we start the Krylov subspace solution method for solving

$$I - u_k u_k^*)\,(A-\theta I)\,(I-u_k u_k^*)\,t = -r,$$

with $t_0 = 0$, then all vectors occurring in the process will be orthogonal to u_k. For the preconditioning step we have to solve a system like

$$(I-u_k u_k^*)K(I-u_k u_k^*)w = z,$$

and because of $u_k^* w = 0$, this reduces to

$$(I-u_k u_k^*)Kw = z,$$

which is equivalent with $Kw = z - \beta u_k$, where β is determined by the requirement that w is orthogonal to u_k. This last equation is easily solved by solving two equations:
1. solve $\widetilde{w}$ from $K\widetilde{w} = z$,
2. solve $\widetilde{u}$ from $K\widetilde{u} = u_k$.
Note that this second equation is identical for all iterations of the linear iterative solver and hence it has to be solved only once per application of the Krylov solver. The value of β is given by

$$\beta = \frac{\widetilde{w}^* u_k}{\widetilde{u}^* u_k}.$$

From a parallel point of view the Jacobi-Davidson method has the same computational ingredients as for instance GMRES. Successful parallel implementation largely depends on how well an effective preconditioner can be parallelized. For the preconditioners we can select any of the preconditioners discussed in Section **??**, but note that the operator $A - \theta I$ will be indefinite in general, so that one has to be careful with incomplete decomposition techniques.

JDQR

The Jacobi-Davidson method can also be used in order to find more than one eigenpair by using *deflation* techniques, similar to the Arnoldi method. If an eigenvector has converged, then we continue in a subspace spanned by the remaining eigenvectors. A problem is then how to re-use information obtained in a previous Jacobi-Davidson cycle.
In [155] an algorithm is proposed by which several eigenpairs can be computed. The algorithm is based on the computation of a partial Schur form of A:

$$AQ_k = Q_k R_k,$$

where Q_k is an $n \times k$ orthonormal matrix, and R_k is a $k \times k$ upper triangular matrix, with $k \ll n$. Note that if (x, λ) is an eigenpair of R_k, then $(Q_k x, \lambda)$ is an eigenpair of A.

We now proceed in the following way in order to obtain this partial Schur form for eigenvalues close to a target value τ.
Step I: Given an orthonormal subspace basis $v_1, \ldots, v_i$, with matrix V_i, compute the projected matrix $M = V_i^* A V_i$. For the $i \times i$ matrix M we compute the complete Schur form $M = US$, with $U^* U = I$, and S upper triangular. This can be done with the standard QR algorithm [177].
Then we orden S such that the $|s_{i,i} - \tau|$ form a nondecreasing row for increasing i. The first few diagonal elements of S then represent the eigenapproximations closest to τ, and the first few of the correspondingly reordened columns of V_i represent the subspace of best eigenvector approximations. If memory is limited then this subset can be used for restart, that is the other columns are simply discarded. The remaining subspace is expanded according to the Jacobi-Davidson method.
After convergence of this procedure we have arrived at an eigenpair (q, λ) of A: $Aq = \lambda q$. The question is how to expand this partial Schur form of dimension 1. This will be shown in step II.
Step II: Suppose we have allready a partial Schur form of dimension k, and we want to expand this by a convenient new column q:

$$A\,[Q_k, q] = [Q_k, q] \begin{bmatrix} R_k & s \\ & \lambda \end{bmatrix}$$

with $Q^* q = 0$.
After some standard linear algebra manipulations it follows that

$$(I - Q_k Q_k^*)(A - \lambda I)(I - Q_k Q_k^*) q = 0,$$

which expresses that the new pair (q, λ) is an eigenpair of

$$\widetilde{A} = (I - Q_k Q_k^*) A (I - Q_k Q_k^*).$$

This pair can be computed by applying the Jacobi-Davidson algorithm (with Schur form reduction, as in step I) for $\widetilde{A}$.

Some notes are appropriate:

1. Although we see that after each converged eigenpair the explicitly deflated matrix $\widetilde{A}$ leads to more expensive computations, it is shown in [155], by numerical experiments, that the entire procedure leads to a very efficient computational process. An explanation for this is that after convergence of some eigenvectors, the matrix $\widetilde{A}$ will be better conditioned, so that the correction equation in the Jacobi-Davidson step is more easily solved.

2. The correction equation may be solved by a preconditioned iterative solver, and it is shown in [155] that the same preconditioner can be used with great efficiency for different eigenpairs. Hence, it pays to construct better preconditioners.

3. In [155] a similar algorithm for generalized eigenproblems $Ax = \lambda Bx$ is proposed, based on partial QZ reduction citegolo89.

10.0.24 Eigenvalue Software - ARPACK, P_ARPACK

ARPACK is a collection of Fortran77 subroutines designed to solve large scale eigenvalue problems. ARPACK stands for ARnoldi PACKage. ARPACK software is capable of solving large scale non Hermitian (standard and generalized) eigenvalue problems from a wide range of application areas. Parallel ARPACK (P_ARPACK) is provided as an extension to the current ARPACK library and is targeted for distributed memory message passing systems. The message passing layers currently supported are BLACS and MPI.

This software is based upon the Implicitly Restarted Arnoldi Method (IRAM) we presented in Section ??. When the matrix A is symmetric it reduces to a variant of the Lanczos process called the Implicitly Restarted Lanczos Method (IRLM). For many standard problems, a matrix factorization is not required. Only the action of the matrix on a vector is needed.

The important features of ARPACK and P_ARPACK are:

- A reverse communication interface.
- Ability to return k eigenvalues which satisfy a user specified criterion such as largest real part, largest absolute value, largest algebraic value (symmetric case), etc.
- A fixed pre-determined storage requirement of $n \cdot \mathcal{O}(k) + \mathcal{O}(k^2)$ words will typically suffice throughout the computation. No auxiliary storage is required.
- Sample driver routines are included that may be used as templates to implement various spectral transformations to enhance convergence and to solve the generalized eigenvalue problem. Also, there is an SVD driver.
- Special consideration is given to the generalized problem $Ax = Bx\lambda$ for singular or ill-conditioned symmetric positive semi-definite M.
- A numerically orthogonal Schur basis of dimension k is always computed. These are also eigenvectors in the Hermitian case and orthogonality is to working precision. Eigenvectors are available on request in the non-Hermitian case.

- The numerical accuracy of the computed eigenvalues and vectors is user specified. Residual tolerances may be set to the level of working precision. At working precision, the accuracy of the computed eigenvalues and vectors is consistent with the accuracy expected of a dense method such as the implicitly shifted QR iteration.

- Multiple eigenvalues offer no theoretical difficulty. This is possible through deflation techniques similar to those used with the implicitly shifted QR algorithm for dense problems. With the current deflation rules, a fairly tight convergence tolerance and sufficiently large subspace will be required to capture all multiple instances. However, since a block method is not used, there is no need to "guess" the correct block size that would be needed to capture multiple eigenvalues.

Reverse Communication Interface

The reverse communication interface is one of the most important aspects of the design of ARPACK both for interfacing with user application codes and for parallel decomposition. This interface avoids having to express a matrix-vector product through a subroutine with a fixed calling sequence. This means that the user is free to choose any convenient data structure for the matrix representation. Also, the user is free (and responsible for) to partition the matrix-vector product in the most favorable way for parallel efficiency. Moreover, if the matrix is not available explicitly, the user is free to express the action of the matrix on a vector through a subroutine call or a code segment. It is not necessary to conform to a fixed format for a subroutine interface and hence there is no need to communicate data through the use of `COMMON`.

A typical usage of this interface is illustrated as follows:

```
 10  continue
     call snaupd (ido, bmat, n, which,...,workd,..., info)
     if (ido .eq. newprod) then
        call matvec ('A', n, workd(ipntr(1)), workd(ipntr(2)))
     else
        return
     endif
     go to 10
```

This shows a code segment of the routine the user must write to set up the reverse communication call to the top level ARPACK routine `snaupd` to solve a nonsymmetric eigenvalue problem. As usual, with reverse communication, control is returned to the calling program when interaction with the matrix A is required. The action requested of the calling program is simply to perform `ido`. (in this case multiply the vector held in the array `workd` beginning at location `ipntr(1)` and inserting the result into the array `workd` beginning at location `ipntr(2)`). Note that the `call` to the subroutine `matvec` in this code segment is simply meant to indicate that this matrix-vector operation is taking place. One only needs to supply the action of the matrix on the specified vector and put the result in the designated location.

Parallelizing ARPACK

The parallelization paradigm found to be most effective for ARPACK on distributed memory machines was to provide the user with a Single Program Multiple Data (SPMD) template. The reverse communication interface is one of the most important aspects in the design of ARPACK and this feature lends itself to a simplified SPMD parallelization strategy. This approach was used for previous parallel implementations of ARPACK [?] and provides a fairly straightforward interface for the user. Reverse communication allows the P_ARPACK codes to be parallelized internally without imposing a fixed parallel decomposition on the matrix or the user supplied matrix-vector product. Memory and communication management for the matrix-vector product $w \leftarrow Av$ can be optimized independent of P_ARPACK . This feature enables the use of various matrix storage formats as well as calculation of the matrix elements on the fly.

The calling sequence to ARPACK remains unchanged except for the addition of the BLACS context (or MPI communicator). Inclusion of the context (or communicator) is necessary for global communication as well as managing I/O. The addition of the context is new to this implementation and reflects the improvements and standardizations being made in message passing [?, ?].

10.0.25 Data Distribution of the Arnoldi Factorization

The numerically stable generation of the Arnoldi factorization

$$AV_k = V_k H_k + f_k e_k^T$$

where

A, $n \times n$ matrix

H_k, $k \times k$ - projected matrix (Upper Hessenberg)

V_k, $n \times k$ matrix, $k \ll n$ - Set of Arnoldi vectors

f_k, residual vector, length n, $V_k^T f_k = 0$

coupled with an implicit restarting mechanism [?] is the basis of the ARPACK codes. The simple parallelization scheme used for P_ARPACK is as follows.

- H_k replicated on every processor
- V_k is distributed across a 1-D processor grid. (Blocked by rows)
- ff_k and workspace distributed accordingly

The SPMD code looks essentially like the serial code except that the local block of the set of Arnoldi vectors, V_{loc}, is passed in place of V, and n_{loc}, the dimension of the local block, is passed instead of n.

With this approach there are only two communication points within the construction of the Arnoldi factorization inside P_ARPACK: computation of the 2-norm of the distributed vector f_k and the orthogonalization of f_k to V_k using Classical Gram Schmidt with DGKS correction [?]. Additional communication will typically occur in the user supplied matrix-vector product operation as well. Ideally, this product will only require nearest neighbor communication among the processes. Typically the blocking of V coincides with the parallel decomposition of the matrix A. The user is free to select an appropriate blocking of V to achieve optimal balance between the parallel performance of P_ARPACK and the user supplied matrix-vector product.

The SPMD parallel code looks very similar to that of the serial code. Assuming a parallel version of the subroutine matvec, an example of the application of the distributed interface is illustrated as the follows:

```
 10   continue
      call psnaupd (comm, ido, bmat, nloc, which, ...,
     *                      Vloc , ... lworkl, info)
      if (ido .eq. newprod) then
         call matvec ('A', nloc, workd(ipntr(1)), workd(ipntr(2)))
      else
         return
      endif
      go to 10
```

Where, `nloc` is the number of rows in the block `Vloc` of V that has been assigned to this node process.

The blocking of V is generally determined by the parallel decomposition of the matrix A. For parallel efficiency, this blocking must respect the configuration of the distributed memory and

$$
\begin{array}{ll}
(1) & \beta_k \leftarrow gnorm(\|f_k^{(*)}\|\,); \;\; v_{k+1}^{(j)} \leftarrow f_k^{(j)} \cdot \frac{1}{\beta_k}; \\[2ex]
(2) & w^{(j)} \leftarrow (Aloc) v_{k+1}^{(j)}\,; \\[2ex]
(3) & \begin{pmatrix} h \\ \alpha \end{pmatrix}^{(j)} \leftarrow \begin{pmatrix} V_k^{(j)T} \\ v_{k+1}^{(j)T} \end{pmatrix} w^{(j)}\,; \begin{pmatrix} h \\ \alpha \end{pmatrix} \leftarrow gsum \left[\begin{pmatrix} h \\ \alpha \end{pmatrix}^{(*)} \right] \\[2ex]
(4) & f_{k+1}^{(j)} \leftarrow w^{(j)} - (V_k, v_{k+1})^{(j)} \begin{pmatrix} h \\ \alpha \end{pmatrix}\;; \\[2ex]
(5) & H_{k+1} \leftarrow \begin{pmatrix} H_k & h \\ \beta_k & e_k^T \leftarrow \end{pmatrix}\;; \\[2ex]
(6) & V_{k+1}^{(j)} \leftarrow (V_k, v_{k+1})^{(j)}\,;
\end{array}
$$

Table 10.1: The explicit steps of the process responsible for the j block.

interconnection network. Logically, the V matrix be partitioned by blocks

$$V^T = (V^{(1)^T}, V^{(2)^T}, ..., V^{(nproc)^T})$$

with one block per processor and with H replicated on each processor. The explicit steps of the CGS process taking place on the j-th processor are shown in Table 10.1.

Note that the function $gnorm$ at Step 1 is meant to represent the global reduction operation of computing the norm of the distributed vector f_k from the norms of the local segments $f_k^{(j)}$ and the function $gsum$ at Step 3 is meant to represent the global sum of the local vectors $h^{(j)}$ so that the quantity $h = \sum_{j=1}^{nproc} h^{(j)}$ is available to each process on completion. These are the only two communication points within this algorithm. The remainder is perfectly parallel. Additional communication will typically occur at Step 2. Here the operation $(Aloc)v$ is meant to indicate that the user supplied matrix-vector product is able to compute the local segment of the matrix-vector product Av that is consistent with the partition of V. Ideally, this would only involve nearest neighbor communication among the processes.

Since H is replicated on each processor, the implicit restart mechanism described in Section ?? remains untouched. The only difference is that the local block $V^{(j)}$ is in place of the full matrix V. Operations associated with implicit restarting are perfectly parallel with this strategy. They consist of the following steps that occur independently on each processor:

for $i = 1, 2, ..., p$,
  Factor $[Q_i, R_i] = \mathrm{q}r(H_m - \mu_i I)$;
  $H_m \leftarrow Q_i^* H_m Q_i$;
  $Q \leftarrow QQ_i$;
end
$V_m^{(j)} \leftarrow V_m^{(j)} Q$;

All operations on the matrix H are replicated on each processor. Thus there is no communication overhead The replication of H and the shift selection and application to H on each processor represents a serial bottleneck that limits the scalability of this scheme when k grows with n. However, if k is fixed as n increases then this scheme scales linearly with n as we shall demonstrate with some computational results. In the actual implementation, separate storage is not required for the Q_i. Instead, it is represented as a product of 2×2 Givens or 3×3 Householder transformations that are applied directly to update Q. On completion of this accumulation of Q, the operation $V_m^{(j)} \leftarrow V_m^{(j)} Q$ independently on each processor j using the level 3 BLAS operation _GEMM.

The main benefit of this approach is that the changes to the serial version of ARPACK were very minimal. Since the change of dimension from matrix order n to its local distributed blocksize `nloc` is invoked through the calling sequence of the subroutine `psnaupd`, there is no fundamental algorithmic change within the code. Only eight routines were affected in a minimal way. These routines either required a change in norm calculation to accomodate distributed vectors (Step 1), modification of the distributed dense matrix-vector product (Step 4), or inclusion of the context or communicator for I/O (debugging/tracing). More specifically, the commands are changed from

```
rnorm = sdot (n, resid, 1, workd, 1)
rnorm = sqrt(abs(rnorm))
```

to

```
rnorm = sdot (n, resid, 1, workd, 1)
call sgsum2d(comm,'All',' ',1, 1, rnorm, 1, -1, -1 )
rnorm = sqrt(abs(rnorm))
```

where `sgsum2d` is the BLACS global sum operator. The MPI implementation uses the

MPI_ALLREDUCE global operator. Similarly, the computation of the matrix-vector product operation $h \leftarrow V^T w$ requires a change from

```
   call sgemv ('T', n, j, one, v, ldv, workd(ipj), 1,
  *            zero, h(1,j), 1)
```

to

```
   call sgemv ('T', n, j, one, v, ldv, workd(ipj), 1,
  *            zero, h(1,j), 1)
   call sgsum2d( comm, 'All', ' ', j, 1, h(1,j), j,
  *              -1, -1 )
```

Another strategy which was tested was to use Parallel BLAS (PBLAS) [?] software developed for the ScaLAPACK project to achieve parallelization. The function of the PBLAS is to simplify the parallelization of serial codes implemented on top of the BLAS. The ARPACK package is very well suited for testing this method of parallelization since most of the vector and matrix operations are accomplished via BLAS and LAPACK routines.

Unfortunately this approach required additional parameters to be added to the calling sequence (the distributed matrix descriptors) as well as redefining the workspace data structure. Although there is no significant degradation in performance, the additional code modifications, along with the data decomposition requirements, make this approach less favorable. As our parallelization is only across a one dimensional grid, the functionality provided by the PBLAS was more sophisticated than we required. The current implementation of the PBLAS (ScaLAPACK version 1.1) assumes the matrix operands to be distributed in a block-cyclic decomposition scheme.

10.0.26 Message Passing

One objective for the development and maintenance of a parallel version of the ARPACK [?] package was to construct a parallelization strategy whose implementation required as few changes as possible to the current serial version. The basis for this requirement was not only to maintain a level of numerical and algorithmic consistency between the parallel and serial implementations, but also to investigate the possibility of maintaining the parallel and serial libraries as a single entity.

On many shared memory MIMD architectures, a level of parallelization can be accomplished via compiler options alone without requiring any modifications to the source code. This is rather ideal for the software developer. For example, on the SGI Power Challenge architecture the MIPSpro F77 compiler uses a POWER FORTRAN Accelerator (PFA) preprocessor to automatically uncover the parallelism in the source code. PFA is an optimizing Fortran preprocessor that discovers parallelism in Fortran code and converts those programs to parallel code. A brief discussion of implementation details for ARPACK using PFA preprocessing may be found in [?]. The effectiveness of this preprocessing step is still dependent on how suitable the source code is for parallelization. Since most of the vector and matrix operations for ARPACK are accomplished via BLAS and LAPACK routines, access to efficient parallel versions of these libraries alone will provide a reasonable level of parallelization.

Unfortunately, for distributed memory architectures the software developer is required to do more work. For distributed memory implementations, message passing between processes must be explicitly addressed within the source code and numerical computations must take into account the distribution of data. In addition, for the parallel code to be portable, the communication interface used for message passing must be supported on a wide range of parallel machines and platforms. For /small P_ARPACK, this portability is achieved via the Basic Linear Algebra Communication Subprograms (BLACS) [?] developed for the ScaLAPACK project and Message Passing Interface (MPI) [?].

10.0.27 Parallel Performance

To illustrate the potential scalability of Parallel ARPACK on distributed memory architectures some example problems have been run on the Maui HPCC SP2. The results shown in Table 1 attempt to illustrate the potential internal performance of the of the P_ARPACK routines independent of the users implementation of the matrix vector product.

In order to isolate the performance of the ARPACK routines from the performance of the user's matrix-vector product and also to eliminate the effects of a changing problem characteristic as the problem size increases, a test was comprised of replicating the same matrix repeatedly to obtain a block diagonal matrix. This completely contrived situation allows the workload to increase linearly with the number of processors. Since each diagonal block of the matrix is identical, the algorithm should behave as if *nproc* identical problems are being solved simultaneously (provided an appropriate starting vector is used). For this example we use a starting vector of all "1's". The only obstacles which prevent ideal speedup are the communication costs involved in the global operations and the "serial bottleneck" associated with the replicated operations on the projected matrix H. If neither of these were present then one would expect the execution time to remain constant as the problem size and the number of processors increase.

The matrix used for testing is a diagonal matrix of dimension $100,000$ with uniform random elements between 0 and 1 with four of the diagonal elements separated from the rest of the spectrum by adding an additional 1.01 to these elements. The problem size is then increased linearly with the number of processors by adjoining an additional diagonal block for each additional processor. For

Number of Nodes	Problem Size	Total Time (s)	Efficiency
1	100,000 * 1	40.53	
4	100,000 * 4	40.97	0.98
8	100,000 * 8	42.48	0.95
12	100,000 * 12	42.53	0.95
16	100,000 * 16	42.13	0.96
32	100,000 * 32	46.59	0.87
64	100,000 * 64	54.47	0.74
128	100,000 * 128	57.69	0.70

Table 10.2: Internal Scalability of P_ARPACK

these timings we used the non-symmetric P_ARPACK code pdnaupd with the following parameter selections: mode is set to 1, number of Ritz values requested is 4, portion of the spectrum is "LM", and the maximum number of columns of V is 20.

10.0.28 Availability

The codes are available by anonymous ftp from

`ftp.caam.rice.edu`

or by connecting directly to the `URL`

`http://www.caam.rice.edu/software/ARPACK`

To get the software by anonymous ftp, connect by ftp to `ftp.caam.rice.edu` and `login` as `anonymous`. Then change directories to

```
software/ARPACK
```

or connect directly to the URL as described above and follow the instructions in the `README` file in that directory. The ARPACK software is also available from Netlib in the directory `ScaLAPACK`.

10.0.29 Summary

The implementation of the P_ARPACK is portable across a wide range of distributed memory platforms. The portability of P_ARPACK is achieved by utilization of the BLACS and MPI. We have been quite satisfied with how little effort it takes to port P_ARPACK to a wide variety of parallel platforms. So far we have tested P_ARPACK on a SGI Power Challenge cluster using PVM-BLACS and MPI, on a CRAY T3D using Cray's implementation of the BLACS, on an IBM SP2 using MPL-BLACS and MPI, on a Intel paragon using NX-BLACS and MPI, and on a network of Sun stations using MPI and MPI-BLACS.

Chapter 11

The Generalized Eigenproblem

The generalized eigenproblem ($Ax = \lambda Bx$) occurs frequently and naturally in many large scale applications. Often in large scale problems, the matrix B arises from inner products of basis vectors in a finite-element discretization of a continuous problem. As such, it is symmetric and positive (semi-) definite.

11.0.30 Arnoldi and Lanczos

Several schemes have been developed to extend the Krylov subspace idea to the generalized problem (**??**). These extensions are generally based upon a conversion of the generalized problem to a standard one. Perhaps the most successful variant [**?**] is to use the *spectral transformation*

$$(A - \sigma B)^{-1} Bx = x\nu.$$

An eigenvector x of this transformed problem is also an eigenvector of the original problem (**??**) with the corresponding eigenvalue given by $\lambda = \sigma + \frac{1}{\nu}$. When the matrix B is symmetric and positive (semi-) definite, it is often helpful to work with the B (semi-) inner product in the Lanczos/Arnoldi process [**?**, **?**, **?**]. With this transformation, convergence of the Lanczos/Arnoldi iteration is very rapid to eigenvalues near the shift σ because they are transformed to extremal well-separated eigenvalues and also since eigenvalues far from σ are damped (mapped near zero).

To utilize this transformation in a Lanczos/Arnoldi process, the repeated operation $w \leftarrow Av$ is replaced by repeated solutions of a shift invert equation $(A - \sigma B)w = Bv$ at each step of the iteration. If a sparse-direct factorization of the shifted matrix $(A - \sigma B)$ is possible then this single factorization may be re-used at each step of the iteration. This approach is certainly the method of choice but may not be practical or even possible in many important applications.

In some cases, it may be effective to use a pre-conditioned iterative method to solve the shift-invert equations but there are a number of pitfalls to this approach. Typically, the shifted matrix is very ill-conditioned because σ will be chosen to be near an interesting eigenvalue. Moreover, this shifted matrix will usually be indefinite (or have indefinite symmetric part). These are the conditions that are most difficult for iterative solution of linear systems. Finally, these difficulties are exacerbated by the fact that each linear system must be solved to a considerably greater accuracy than the desired accuracy of the eigenvalue calculation. Otherwise, each step of the Lanczos/Arnoldi process will essentially involve a different matrix operator. The Jacobi-Davidson method can be generalized in order to make inexact solves with $(A - \sigma B)$ allowable. This method will be discussed in the next section.

11.0.31 The Jacobi-Davidson QZ algorithm

We will describe how the Jacobi-Davidson method can be applied directly to the generalized eigenproblem, that is without transforming this problem first to a standard eigenproblem. This means that explicit inversion of any of the involved matrices, or combinations thereof, is avoided. The standard methods of Arnoldi and Lanczos are not well-suited for this. In order to be able to work with orthogonal bases throughout the algorithm, we will focus on a reduction to Schur form of the matrices, instead of diagonal form.

In the Jacobi-Davidson method a subspace is generated onto which the given eigenproblem is projected. The much smaller projected eigenproblem is solved by standard direct methods, and this leads to approximations for the eigensolutions of the given large problem. This is the 'Davidson' part of the algorithm. Then, a correction equation for a selected eigenpair is set up. The solution of the correction equation defines an orthogonal correction for the current eigenvector approximation; this is the 'Jacobi' part of the algorithm. The correction is used for the expansion of the subspace and the procedure is repeated.

In the algorithm the large system is projected on a suitable subspace, in a way that may be viewed as a truncated and inexact form of the QZ factorization. The projected system itself is reduced to Schur form by the QZ algorithm. For this reason we refer to the algorithm as JDQZ.

We consider the generalized eigenproblem $Ax = \lambda Bx$, and our aim is to compute a few eigenvalues close to a given target $\tau \in \mathbb{C}$. Suppose that we have a suitable low-dimensional subspace, spanned by v_1, ..., v_m, and we want to identify good eigenvector approximations in this subspace. Let V_m denote the $n \times m$ matrix with v_i as its i-th column. Let us further assume that the vectors v_i form an orthonormal basis. Then the Petrov-Galerkin approach is to find a vector $y \in \mathbb{R}^m$, and a θ, such that

$$AV_m y - \theta BV_m y \perp \{w_1, \ldots, w_m\},$$

for some convenient set of vectors $\{w_j\}$. We will assume that these vectors are also orthonormal and that they form the columns of a matrix W_m. This leads to an eigenproblem for projected matrices of order m:

$$W_m^* A V_m y - \theta W_m^* B V_m y = 0, \tag{11.1}$$

where W_m^* denotes the adjoint of W_m. In applications we have that $m \ll n$.
The solutions θ are referred to as the (Petrov-)Ritz values (approximations for eigenvalues of $A - \lambda B$), and $V_m y$ is the corresponding (Petrov-)Ritz vector, with respect to the subspace spanned by $v_1, \ldots, v_m$.

The main problem is to find the expansion vectors v_{m+1} and w_{m+1} for the next iteration step in the process for improving the eigen approximations. In [284], it is shown how the expansion vectors can be selected for the standard eigenproblem $Ax = \lambda x$, and in [155] the same approach is followed for generalized eigenproblems. For this approach it is not necessary to transform the given generalized eigenproblem to a standard one (for instance, by inverting one of the matrices A or B, or a combination).

We will briefly describe this inverse-free Jacobi-Davidson approach, for details we refer to [155]. First we use the QZ algorithm [178] to reduce (11.1) to a generalized Schur form. With m the dimension of span$\{v_j\}$, this QZ computation delivers orthogonal $(m \times m)$ matrices U_R and U_L, and upper triangular $(m \times m)$ matrices S_A and S_B, such that

$$U_L^* \left(W_m^* A V_m\right) U_R = S_A, \tag{11.2}$$

$$U_L^* \left(W_m^* B V_m\right) U_R = S_B. \tag{11.3}$$

The decomposition is ordered so that the leading diagonal elements of S_A and S_B represent the eigenvalue approximation closest to the target value τ. The approximation for the eigenpair is then taken as

$$(\tilde{q}, \theta) \equiv (V_m U_R(:,1), S_B(1,1)/S_A(1,1)\,), \tag{11.4}$$

assuming that $S_A(1,1) \neq 0$ (see [155] for further details). With this approximation for the eigenvector and the eigenvalue, we compute the *residual vector*:

$$r \equiv A\tilde{q} - \theta B \tilde{q}.$$

We also compute an auxiliary vector $\gamma \tilde{z} = A\tilde{q} - \tau B\tilde{q}$, where γ is such that $||\tilde{z}||_2 = 1$.
With these two vectors we can define the correction equation for $\tilde{q}$, that is so typical for the Jacobi-Davidson approach: we want to find the correction v for $\tilde{q}$ with

$$\tilde{q}^* v = 0 \quad \text{and} \quad (I - \tilde{z}\tilde{z}^*)(A - \theta B)(I - \tilde{q}\tilde{q}^*)v = -r\,. \tag{11.5}$$

In practice, we solve (11.5) only approximately by a few steps of, for instance, GMRES (with a suitable preconditioner, which is always necessary in order have the output vectors in the proper

subspace). Then we orthonormalize the approximation for v with respect to v_1, ..., v_m, and take the results as our new v_{m+1}. For the expansion vector w_{m+1}, we take the vector $Av - \tau Bv$, orthonormalized with respect to w_1, ..., w_m.
It can be shown that in this case $\tilde{z} = W_m U_L(:, 1)$.

The above sketched algorithm descibes the *Harmonic Petrov value approach* proposed in [155].

11.0.32 The Jacobi-Davidson QZ-method: restart and deflation

In some circumstances the Jacobi-Davidson method has apparent disadvantages with respect to other iterative schemes, for instance the shift-and invert Arnoldi method [293]. For instance, in many cases we see rapid convergence to one single eigenvalue, and what to do if we want more eigenvalues? For Arnoldi this is not a big problem, since the usually slower convergence towards a particular eigenvalue goes hand in hand with simultaneous convergence towards other eigenvalues. So after a number of steps Arnoldi produces approximations for several eigenvalues.
For Jacobi-Davidson the obvious approach would be to restart with a different target, with no guarantee, hoewever, that this leads to a new eigenpair. Also the detection of multiple eigenvalues is a problem, but this problem is shared with the other subspace methods.

A well-known way out of this problem is to use a technique, known as *deflation*. If an eigenvector has converged, then we continue in a subspace spanned by the remaining eigenvectors. A problem is then how to re-use information obtained in a previous Jacobi-Davidson cycle. In [155] an algorithm is proposed by which several eigenpairs can be computed. The algorithm is based on the computation of a partial generalized Schur form for the matrix pair (A, B):

$$AQ_k = Z_k S_k \quad \text{and} \quad BQ_k = Z_k T_k,$$

in which Q_k and Z_k are n by k orthonormal matrices and S_k, T_k are k by k upper triangular matrices, with $k \ll n$. Note that if (x, λ) is an eigenpair of $S_k x - \lambda T_k x = 0$, then $(Q_k x, \lambda)$ is an eigenpair of $A - \lambda B$.

We now proceed in the following way in order to obtain this partial generalized Schur form for eigenvalues close to a target value τ.

Step I: Given orthonormal subspace bases $v_1, \ldots, v_i$, with matrix V_i, and w_1, ..., w_i, with matrix W_i, we compute the projected matrices $M_A = W_i^* A V_i$ and $M_B = W_i^* B V_i$. For these $i \times i$ matrices M_A and M_B, we compute the complete generalized Schur form

$$M_A U_R = U_L S_A \quad \text{and} \quad M_B U_R = U_L S_B, \tag{11.6}$$

in which U_R and U_L are orthonormal, and S_A and S_B are upper triangular. This can be done with the standard QZ algorithm [177].
Then we orden S_A, and S_B such that the $|S_B(i,i)/S_A(i,i) - \tau|$ form a nondecreasing row for increasing i. The first few diagonal elements of S_A and S_B then represent the eigenapproximations closest to τ, and the first few of the correspondingly reordered columns of V_i represent the subspace of best eigenvector approximations. If memory is limited then this subset can be used for restart, that is the other columns are simply discarded. The remaining subspace is expanded according to the Jacobi-Davidson method.
After convergence of this procedure we have arrived at an eigenpair (q, λ) of $A - \lambda B$: $Aq = \lambda Bq$. The question is how to expand this partial generalized Schur form of dimension 1. This will be shown in step II.

Step II: Suppose we have already a partial Schur form of dimension k, and we want to expand this by an appropriate new column q:

$$A\,[Q_k, q] = [Z_k, z] \begin{bmatrix} S_k & s \\ & \alpha \end{bmatrix}$$

and

$$B\,[Q_k, q] = [Z_k, z] \begin{bmatrix} T_k & t \\ & \beta \end{bmatrix}.$$

After some standard linear algebra manipulations it follows that

$$(I - Z_k Z_k^*)(\beta A - \alpha B)(I - Q_k Q_k^*)q = 0 \quad \text{and} \quad Q_k^* q = 0,$$

which expresses that the new pair $(q, \lambda \equiv \alpha/\beta)$ is an eigenpair of the matrix pair

$$(I - Z_k Z_k^*)A(I - Q_k Q_k^*), \;\; (I - Z_k Z_k^*)B(I - Q_k Q_k^*). \tag{11.7}$$

Approximations for an eigenpair of this matrix pair can be computed by applying the Jacobi-Davidson algorithm (with Schur form reduction, as in step I), with appropriate W and V. Expansion vectors for V and W are computed by solving approximately the Jacobi-Davidson correction equation:

$$(I - \tilde{z}\tilde{z}^*)(I - Z_k Z_k^*)(A - \lambda B)(I - Q_k Q_k^*)(I - \tilde{q}\tilde{q}^*)v = -r, \tag{11.8}$$

with $Q_k v = 0$, and $\tilde{q}^* v = 0$. This leads to projected matrices M_A and M_B, for which it is easy to prove that:

$$M_A = W^*(I - Z_k Z_k^*)A(I - Q_k Q_k^*)V = W^* A V, \tag{11.9}$$

and

$$M_B = W^*(I - Z_k Z_k^*)B(I - Q_k Q_k^*)V = W^* B V. \tag{11.10}$$

For these matrices we construct again, as in step I, the generalized Schur decomposition:

$$M_A U_R = U_L S_A \quad \text{and} \quad M_B U_R = U_L S_B.$$

Upon convergence of the Jacobi-Davidson iteration, we obtain the expansion vector $q = VU_R(:,1)$ for Q_k, and the expansion vector z for Z_k is computed with the relation $z = WU_L(:,1)$.

Some notes are appropriate:

1. In our computations, we have used the explicitly deflated matrix $A - \lambda B$ in (11.8), whereas we could have exploited the much less expensive expressions (11.9) and (11.10). We have made this choice for stability reasons. Despite these more expensive computations, it is shown in [155], by numerical experiments, that the entire procedure leads to a very efficient computational process. An explanation for this is that after convergence of some eigenvectors, the pair M_A, M_B will be better conditioned, so that the correction equation (11.8) in the Jacobi-Davidson step is more easily solved.

2. The correction equation (11.8) may be solved (approximately) by a preconditioned iterative solver, and it is shown in [155] that the same preconditioner can be used with great efficiency for different eigenpairs. Hence, the costs for constructing accurate preconditioners can be spread over many systems to be solved.

11.0.33 Parallel aspects

The Jacobi-Davidson method lends itself in an excellent way for parallel computing because of its explicit nature. We will discuss the computational elements of the algorithm in view of parallel processing.

- In the initialization step subspaces V and W are chosen and orthonormal basis for these subspaces are constructed. This involves inner products and vector updates.
- In the first step of the iteration loop, the matrices M_A and M_B have to be computed. The projections with W and V, as in equations (11.9) and (11.10), also involve only inner products and vector updates. Note that it is sufficient to compute only the last recent row and colums of these matrices. This again involves a number of innerproducts: one per new matrix element.
- In the second step the eigenpairs of the projected pair are computed. The projected system is of low dimension: the projected matrices are typically of order less than 30, and hence the number of computations in this step is limited and is often negligible with respect to the number of computations in the other steps. Therefore, we let all processors do this computation, in order to avoid further communication.

- In the third step the selected Ritz vector is computed with (11.4) by taking a linear combination of the columns of V. Hence, if cleverly implemented, no multiplications with the matrices A and B are needed in this step, but only vector updates.
- In the fourth step the convergence criterion is checked. This typically involves the computation of the norm of the residual r.
- The computation of the approximate solution of the correction equation (11.8) is usually the most expensive step in the algorithm. If this is done by a method like GMRES, then the only time-consuming operations are matrix-vector multiplications, vector updates, and inner products. Preconditioning for GMRES requires special attention with respect to parallelizability; we will not discuss this aspect in this paper.
Note that we explicitly deflate with Z_k and Q_k in (11.8). Multiplication with the entire operator can be carried out in five steps. First multiply with the two projectors on the right, then multiply with the matrices A and B, and take a linear combination of the results, and finally multiply with the projectors on the left. A multiplication with a projector requires only inner products and vector updates.
- In the final step of the algorithm, the subspace V is expanded with the solution of the correction equation and this new basis vector is orthogonalized against the others. This process also requires only inner products and vector updates.

From this global analysis we learn that the main CPU time-consuming operations are multiplications with matrices, vector updates, and inner products, and we only need to parallelize these operations. The inner products may require global communication (in distributed processing mode), and this may need further attention for large numbers of processors.

In [**?**] it has been decribed how to implement this algorithm on the Cray T3D; a distributed memory computer. The communication steps have been implemented with the fast SHMEM_GET and SHMEM_PUT routines. The observed speed-ups for a slightly different formulation of the algorithm, were very satisfactory: in [**?**] a problem of order 274625 is described for which a linear speed-up was observed when increasing the number of processors from 16 to 64 (without preconditioning for (11.8)).

Appendix A

Acquiring Mathematical Software

A1. Netlib

We have collected in *netlib* much of the software described or used in this book. The *netlib* service provides quick, easy, and efficient distribution of public-domain software to the scientific computing community on an as-needed basis.

A user sends a request by electronic mail to *netlib@ornl.gov* on the Internet. A request has one of the following forms:

```
send index

send index from {library}

send {routines} from {library}

find {keywords}
```

The *netlib* service provides its users with features not previously available:

- There are no administrative channels to go through.
- Since no human processes the request, it is possible to get software at any time, even in the middle of the night.
- The most up-to-date version is always available.

- Individual routines or pieces of a package can be obtained instead of a whole collection.

Below is a list of software available when this book went to print.

```
This directory contains software described in the book,
Linear Algebra Computations on Vector and Parallel Computers,
by Jack Dongarra, Iain Duff, Danny Sorensen, and Henk Van der Vorst.

slpsubhalf  software to measure the performance of routines SGEFA and
            SGESL from the Linpack package.

linpacks    software to run the "Linpack Benchmark"

sblas1 single precision Level 1 BLAS
dblas1 double precision Level 1 BLAS
cblas1 complex precision Level 1 BLAS
zblas1 double complex precision Level 1 BLAS

sblas2 single precision Level 2 BLAS
dblas2 double precision Level 2 BLAS
cblas2 complex precision Level 2 BLAS
zblas2 double complex precision Level 2 BLAS

sblas3 single precision Level 3 BLAS
dblas3 double precision Level 3 BLAS
cblas3 complex precision Level 3 BLAS
zblas3 double complex precision Level 3 BLAS

slus.f single precision versions of different blocked
       LU decomposition algorithms
schol.f single precision versions of different blocked
        Cholesky decomposition algorithms
sqrs.f single precision versions of different blocked
       QR decomposition algorithms

benchm a benchmark program for performance test for Fortran loops.
      The program executes a number of Fortran DO-loops and lists
      the execution times for different loop lengths, the Mflops-rates
      and the performance parameters R-inf and n-half.
```

```
pcg3d a preconditioned conjugate gradient code for 3D problems.
```

To get an up-to-date listing, send electronic mail to

```
netlib@ornl.gov
```

In the mail message, type

```
send index from DDSV
```

A1.1 Mathematical Software

In addition netlib contains various items of software and test problems which maybe useful. Listed below are some of these items. (To obtain an item type: *send ma28ad from harwell*, for example.)

```
(harwell/ma28ad) lu,general sparse matrix,pivot for sparsity and stability
(harwell/ma28bd) lu,sparse,different values,factored by (harwell/ma28ad)
(hompack/fixpds) nonlinear equations,f(x)=0,sparse,ode based
(hompack/fixpds) nonlinear equations,rho(a,lambda,x)=0,sparse,ode based
(hompack/fixpds) nonlinear equations,x=f(x),sparse,ode based
(hompack/fixpns) nonlinear equations,f(x)=0,sparse,normal flow
(hompack/fixpns) nonlinear equations,rho(a,lambda,x)=0,sparse,normal flow
(hompack/fixpns) nonlinear equations,x=f(x),sparse,normal flow
(hompack/fixpqs) nonlinear equations,f(x)=0,sparse,augmented jacobian
(hompack/fixpqs) nonlinear equations,rho(a,lamb,x)=0,sparse,augmented jacobian
(hompack/fixpqs) nonlinear equations,x=f(x),sparse,augmented jacobian
(laso/dilaso) all eigenvalue and eigenvector,sparse symmetric,lanczos
(laso/dnlaso) few eigenvalue and eigenvector,sparse symmetric,lanczos
(misc/bsmp) bank and smiths sparse lu made simple
(odepack/lsodes) stiff and nostiff odes,sparse jacobian
(port/chk/prac) test of the port sparse matrix package,[cp]
```

```
(port/chk/prad) test of the port sparse matrix package,[dp]
(port/chk/prsa) test of the port sparse matrix package,[sp]
(port/ex/prea) example of (port/spfor),row and column ordering,sparse matrix
(port/ex/prma) example of (port/spmsf),symbolic lu decomposition,sparse matrix
(port/ex/prs1) example of (port/spmml),sparse matrix vector multiplication
(port/ex/prs3) example of (port/spmnf),numerical lu decomposition,sparse matrix
(port/ex/prsa) example of (port/spmle),sparse linear system solution
(port/ex/prsf) example of (port/spfle),sparse linear system solution
(port/ex/prsj) example of (port/spmce),lu decomposition,sparse matrix
(port/ex/prsm) example of (port/spfce),lu decomposition,sparse matrix
(port/ex/prsp) example of (port/spmlu),lu decomposition of a sparse matrix
(port/ex/prst) example of (port/spflu),lu decomposition of a sparse matrix
(port/ex/prsy) example of (port/spfnf),lu decomposition,sparse matrix
(port/ex/prsz) example of (port/spfml),sparse matrix vector multiplication
(toms/508) bandwidth reduction,profile reduction,sparse matrix
(toms/509) bandwidth reduction,king algorithm,profile reduction,sparse matrix
(toms/529) symmetric permutations,block triangular,depth first search,sparse
(toms/533) sparse matrix,simultaneous linear equations,partial pivoting
(toms/538) eigenvalue,eigenvector,sparse,diagonalizable,simultaneous iteration
(toms/570) eigenvalue,eigenvector,iteration,real sparse nonsymmetric matrix
(toms/586) iterative methods,sparse matrix
(toms/601) sparse matrix
(toms/618) estimating sparse jacobian matrices
(toms/636) estimating sparse hessian matrices,difference of gradients
(y12m/y12m) solution of linear systems,matrices are large and sparse
(y12m/y12mae) sparse matrix,solve one system,single right hand side,[sp]
(y12m/y12maf) sparse matrix,solve one system,single right hand side,[dp]
```

A2. Mathematical Software Libraries

Large mathematical libraries of mathematical software are maintained by the International Mathematical and Statistical Libraries (better known as IMSL), the Numerical Algorithms Group (better known as NAG), and the Harwell Subroutine Library. IMSL and NAG provide their libraries under a license agreement with support provided. With the Harwell Subroutine Library license agreement, no support is provided.

IMSL, Inc.
2500 Park West Tower 1
2500 City Blvd.
Houston, Texas 77042-3020
713-782-6060

NAG, Inc.
1400 Opus Place, Suite 200
Downers Grove, IL 60515-5702
708-971-2337, FAX 708-971-2706

NAG Ltd.
Wilkinson House
Jordan Hill Road
Oxford OX8 YDE
England
+44-865-511245, FAX +44-865-310139

Harwell Subroutine Library
Mr. S. Marlow
Building 8.9
Harwell Laboratory
Didcot, Oxon, OX11 0RA, England

MathWorks

In addition to the Fortran mathematical software libraries, MathWorks provides an interactive system for matrix computations called MATLAB. This matrix calculator runs on many systems from personal computers to workstations to mainframes. MATLAB provides a programming language to allow rapid algorithm design which facilitates the construction and testing of ideas and algorithms. For information on the license agreement for MATLAB, contact MathWorks:

The MathWorks, Inc.
20 North Main St.
Sherborn, MA 01770
508-653-1415

Appendix B

Glossary

address generation: During the execution of an instruction, the cycle in which an effective address is calculated by means of indexing or indirect addressing.

Amdahl's law: The relationship between performance (computational speed) and CPU time. When two parts of a job are executed at a different speed, the total CPU time can be expressed as a function of these speeds. Amdahl first pointed out that the lower of these speeds may have a dramatic influence on the overall performance.

array constant: Within a DO-loop, an array reference all of whose subscripts are invariant:

```
      DO 10 I = 1, N
         A(I) = X(J) * B (I+J) + Z(8,J,K,3)
   10 CONTINUE
```

In the preceding loop, X(J) and Z(8,J,K,3) are array constants. (A(I) and B(I+J) are not array constants since the loop index appears in the subscripts.)

associative access: A method in which access is made to an item whose key matches an access key, as distinct from an access to an item at a specific address in memory.

associative memory: Memory whose contents are accessed by key rather than by address.

attached vector-processor: A specialized processor for vector computations, designed to be connected to a general-purpose host processor. The host processor supplies input/output functions, a file system, and other aspects of a computing system environment.

automatic vectorization: A compiler that takes code written in a serial language (usually Fortran or C) and translates it into vector instructions. The vector instructions may be machine specific or in a source form such as array extensions or as subroutine calls to a vector library.

auxiliary memory: Memory that is usually large, slow, and inexpensive, often a rotating magnetic or optical memory, whose main function is to store large volumes of data and programs not currently being accessed by a processor.

bank conflict: A bank "busy-wait" situation. Since memory chip speeds are relatively slow when required to deliver a single word, supercomputer memories are placed in a large number of independent banks (usually a power of 2). A vector laid out contiguously in memory (one component per successive bank) can be accessed at one word per cycle (despite the intrinsic slowness of memory chips) through the use of pipelined delivery of vector-component words at high bandwidth. When the number of banks is a power of 2, then vectors requiring strides of a power of 2 can run into a bank conflict.

bank cycle time: The time, measured in clock cycles, taken by a memory bank between the honoring of one request to fetch or store a data item and accepting another such request. On most supercomputers this value is either four or eight clock cycles.

BLAS—Basic Linear Algebra Subprograms, a set of Fortran-callable subroutines that perform "kernel" linear algebra operations. Three levels of BLAS currently exist.

barrier synchronization: A means for synchronizing a set of processors in a shared-memory multiprocessor system by halting processors in that set at a specified barrier point until every processor in the set reaches the barrier. At this point the barrier is "removed," and all processors are allowed to resume execution.

cache: Small interface memory with better fetch speed than main memory. The term is more often used when this memory is required to interface with main memory. Cache memory is usually made transparent to the user. A reference to a given area of main memory for one piece of data or instruction is usually closely followed by several additional references to that same area for other data or instruction. Consequently, a cache is automatically filled by a predefined algorithm. The computer system manages the "prefetch" process.

cache coherence: The state that exists when all caches within a multiprocessor have identical values for any shared variable that is simultaneously in two or more caches.

cache hit: A cache access that successfully finds the requested data.

cache line: The unit in which data is fetched from memory to cache.

cache miss: A cache access that fails to find the requested data. The cache must then be filled from main memory at the expense of time.

chaining (linking): The ability to take the results of one vector operation and use them directly as input operands to a second vector instruction, without the need to store to memory or registers the results of the first vector operation. Chaining two vector floating-point operations, for example, could double the asymptotic Mflops rate.

chime: "Chained vector time," approximately equal to the vector length in a DO-loop. The number of chimes required for a loop dominates the time required for execution. A new chime begins in a loop each time a resource (functional unit, vector register, or memory path) must be reused.

chunksize: The number of iterations of a parallel DO-loop grouped together as a single task in order to increase the granularity of the task.

CISC (Complex Instruction Set Computer): A computer with an instruction set that includes complex (multicycle) instructions.

clock cycle: The fundamental period of time in a computer. For example, the clock cycle of a CRAY-2 is 4.1 nsec.

coarse-grain parallelism: Parallel execution in which the amount of computation per task is significantly larger than the overhead and communication expended per task.

combining switch: An element of an interconnection network that can combine certain types of requests into one request and produce a response that mimics serial execution of the requests.

common subexpression: A combination of operations and operands that is repeated, especially in a loop:

```
     DO 20 I = 1, N
        A(I) = 2.0 + B(I) * C(I) + X(I) / T(I)
        Y(I) = P(I) / (2.0 + B(I) * C(I))
        D(I) = X(I) / T(I) + U(I)
  20 CONTINUE
```

The following are common subexpressions in the preceding loop:

```
  2.0 + B(I) * C(I)
  X (I)/T(I)
```

A good compiler will not recompute the common subexpressions but will save them in a register for reuse.

compiler directives: Special keywords specified on a comment card, but recognized by a compiler as providing additional information from the user for use in optimization. For example,

CDIR$ IVDEP

specifies to a CRAY compiler that no data dependencies exist among the array references in the loop following the directive.

compress/index: A vector operation used to deal with the nonzeros of a large vector with relatively few nonzeros. The location of the nonzeros is indicated by an index vector (usually a bit vector of the same length in bits as the full vector in words). The compress operation uses the index vector to gather the nonzeros into a dense vector where they are operated on with a vector instruction. See also **gather/scatter**.

concurrent processing: Simultaneous execution of instructions by two or more processors within a computer.

critical section: A section of a program that can be executed by at most one process at a time.

crossbar (interconnection): An interconnection in which each input is connected to each output through a path that contains a single switching node.

cycle (of computer clock): An electronic signal that counts a single unit of time within a computer.

cycle time: The length of a single cycle of a computer function such as a memory cycle or processor cycle. See also **clock cycle**.

data cache: A cache that holds data but does not hold instructions.

data dependency: The situation existing between two statements if one statement can store into a location that is later accessed by the other statement. For example, in

```
S1: C = A + B
S2: Z = C * X + Y
```

S2 is data dependent on S1: S1 must be executed before S2 so that C is stored before being used in S2. A recursive data dependency involves statements in a DO-loop such that a statement in one iteration depends on the results of a statement from a previous iteration. For example, in

```
   DO 30 I = 1, N
      A(I) = B(I) * A(I-1) + C(I)
30 CONTINUE
```

a recursive data dependency exists in the assignment statement in loop 30: the value of A(I) computed in one iteration is the value A(I-1) needed in the next iteration.

deadlock: A situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.

dependency analysis: An analysis (by compiler or precompiler) that reveals which portions of a program depend on the prior completion of other portions of the program. Dependency analysis usually relates statements in a DO-loop.

direct mapping: A cache that has a set associativity of one, so that each item has a unique place in the cache at which it can be stored.

disk striping: Interleaving a disk file across two or more disk drives to enhance input/output performance. The performance gain is a function of the number of drives and channels used.

distributed memory: A form of storage in which each processor can access only a part of the total memory in the system and must explicitly communicate with other processors in the system to share data.

distributed processing: Processing on a network of computers that do not share main memory.

explicitly parallel: Language semantics that describe which computations are independent and can be performed in parallel. See also **implicitly parallel**.

fetch-and-add: A computer synchronization instruction that updates a word in memory, returns the value before the update, and (if executed concurrently by several processors simultaneously) produces a set of results as if the processors executed in some arbitrary order.

fine-grain parallelism: A type of parallelism where the amount of work per task is relatively small compared with the overhead necessary for communication and synchronization.

flops: Arithmetic floating-point operations (addition and multiplication) per second.

functional units: Functionally independent parts of the ALU each of which performs a specific function, for example, address calculation, floating-point add, and floating-point multiply.

gather/scatter: The operations related to large, sparse data structures. A full vector with relatively few nonzeros is transformed into a vector with only those nonzeros by using a gather operation. The full vector, or one with the same structure, is built from the inverse operation or scatter. The process is accomplished with an index vector, which is usually the length of the number of nonzeros, with each component being the relative location in the full vector. See also **compress/index**.

GAXPY: A generalized *SAXPY* operation, taking linear combinations of a set of columns and accumulating the columns in a vector, as in matrix-vector product.

Gflops (Gigaflops): A computation rate of one billion floating-point operations per second.

global memory: A memory accessible to all of the computer's processors.

granularity: The size of the tasks to be executed in parallel. Fine granularity is illustrated by execution of statements or small loop iterations as separate processes; coarse granularity involves subroutines or sets of subroutines as a separate process. The greater the number of processes, the "finer" the granularity and the greater the overhead required to keep track of them.

Grosch's law: An empirical rule that the cost of computer systems increases as the square root of the computational power of the systems.

hierarchy (of memory system): A multilevel memory structure in which successive levels are progressively larger, slower, and cheaper. Examples are registers, cache, local memory, main memory, secondary storage, and disk.

high-speed buffer memory: A part of the memory that holds data being transferred between a large main memory and the registers of high-speed processors.

hit ratio: the ratio of the number of times data requested is found in the cache.

hot-spot contention: An interference phenomenon observed in multiprocessors caused by memory access statistics being slightly skewed from a uniform distribution to favor a specific memory module.

hypercube architecture: A local-memory architecture where the processors are connected in a topology known as a hypercube. For a hypercube with 2^d processors, each CPU has d nearest neighbors.

implicitly parallel: Language semantics that do not allow the user to explicitly describe which computations are independent and can be performed in parallel. For an implicitly parallel language, the compiler must deduce or prove independence in order to make use of parallel hardware. The comparative difficulty of the deduction separates implicitly parallel languages from explicitly parallel languages.

instruction buffer: A small, high-speed memory that holds instructions recently executed or about to be executed.

instruction cache: A cache memory that holds only instructions, but not data.

instruction scheduling: A strategy of a compiler to analyze the outcome of the operations specified in a program and to issue instructions in an optimal manner. That is, the instructions are not necessarily issued in the order specified by the programmer, but in an order that optimally uses the registers, functional units, and memory paths of the computer—at the same time guaranteeing correct results for the computation.

instruction set: The set of low-level instructions that a computer is capable of executing. Programs expressed in a high-level language must ultimately be reduced to these.

interactive vectorizer: An interactive program to help a user vectorize source code. The program analyzes the source for loops and operation sequences that can be accomplished by using vector instructions or macros. When obstructions to vectorization are found, the user is informed. Often the user can indicate that a vector reference with a potential recursive reference is "safe," or the user can remove an IF-test, branch, or subroutine call, in order to achieve vectorization. See also **true ratio**.

interconnection network: The system of logic and conductors that connects the processors in a parallel computer system. Some examples are bus, mesh, hypercube, and Omega networks.

interprocessor communication: The passing of data and information among the processors of a parallel computer during the execution of a parallel program.

interprocessor contention: Conflicts caused when multiple CPUs compete for shared system resources. For example, memory bank conflicts for a user's code in global-memory architectures are caused by other processors running independent applications.

invariant: A variable, especially in a DO-loop, that appears only on the right side of an equals signs. The variable is read only; it is never assigned a new value.

invariant expression: An expression, especially in a DO-loop, all of whose operands are invariant or constant.

linking: See **chaining.**

load balancing: A strategy in which the longer tasks and shorter tasks are carefully allocated to different processors to minimize synchronization costs or task startup overhead.

local memory: A form of storage in which communication with the small, fast memory is under user control. Local memory is similar to cache, but under explicit program control.

locality: A pragmatic concept having to do with reducing the cost of communication by grouping together objects. Certain ways of laying out data can lead to faster programs.

lock: A shared-data structure that multiple processes access to compete for the exclusive right to continue execution only if no other processor currently holds that exclusive right. Locks are typically implemented as primitive operations.

loop unrolling: A loop optimization technique for both scalar and vector architectures. The iterations of an inner loop are decreased by a factor of two or more by explicit inclusion of the very next one or several iterations. Loop unrolling can allow traditional scalar compilers to make better use of registers and to improve overlap operations. On vector machines, loop unrolling may either improve or degrade performance; the process involves a tradeoff between overlap and register use on the one hand and vector length on the other.

Mflops (megaflops): Millions of floating-point (arithmetic) operations per second, a common rating of supercomputers and vector instruction machines.

main memory: A level of random-access memory, the primary memory of a computer. Typical sizes for large computers are 64–2048 Mbytes.

macrotasking: Dividing a computation into two or more large tasks to be executed in parallel. Typically the tasks are subroutine calls executed in parallel.

memory bank conflict: A condition that occurs when a memory unit receives a request to fetch or store a data item prior to completion of its bank cycle time since its last such request.

memory management: The process of controlling the flow of data among the levels of memory hierarchy.

microtasking: Parallelism at the DO-loop level. Different iterations of a loop are executed in parallel on different processors.

MIMD: A multiple-instruction stream/multiple-data stream architecture. Multiple-instruction streams in MIMD machines are executed simultaneously. MIMD terminology is used more generally to refer to multiprocessor machines. See also **SIMD, SISD.**

mini-supercomputer: A machine with roughly one-tenth to one-half the performance capability of supercomputer and available at roughly one-tenth the price. "Mini-supers" use a blend of minicomputer technology and supercomputer architecture (pipelining, vector instructions, parallel CPUs) to achieve attractive price and performance characteristics.

MIPS: Millions of instructions per second.

MOPS: Millions of operations per second.

multiprocessing: The ability of a computer's operating system to mix separate user jobs on one or more CPUs. See also **multitasking.**

multiprocessor: A computer system with more than one CPU. The CPUs are usually more tightly coupled than simply sharing a local network. For example, a system with CPUs that use a common bus to access a shared memory is called a multiprocessor.

multiprogramming: The ability of a computer to time share its CPU with more than one program on a CPU. See also **multitasking.**

multitasking: The execution of multiple tasks from the same program on one or more processors.

The terms multiprogramming, multiprocessing, and multitasking are often used interchangeably (with a notable lack of precision) to describe three different concepts. We use multiprogramming to refer to the ability of a computer to share more than one program on at least one CPU, and multitasking to refer the simultaneous execution of several tasks from the same program on two or more CPUs, or the ability of a computer to intermix jobs on one or more CPUs.

$n_{1/2}$: The vector length required to achieve one-half the peak performance rate. A large value of $n_{1/2}$ indicates severe overhead associated with vector startup. The rule of thumb is that if the average vector length is 3 times $n_{1/2}$, there is great efficiency; if the vector length is less than that, there is inefficiency.

optimization: A process whereby a compiler tries to make the best possible use of the target computer's hardware to perform the operations specified by a programmer. Alternatively, optimization refers to the process whereby a programmer tries to make optimal use of the target programming language to produce optimal code for the computer architecture.

optimization block: A block of code (rarely a whole subprogram, often a single DO-loop) in which a compiler optimizes the generated code. A few compilers attempt to optimize across such blocks; many work on each block independently.

page: The smallest managed unit of a virtual memory system. The system maintains separate virtual-to-physical translation information for each page.

parallel computer: A computer that can perform multiple operations simultaneously, usually because multiple processors (that is, control units or ALUs) or memory units are available. Parallel computers containing a small number (say, less than 50) of processors are called multiprocessors; those with more than 1000 are often referred to as massively parallel computers.

parallel processing: Processing with more than one CPU on a single program simultaneously.

parallelization: The simultaneous execution of separate parts of a single program.

parsing: The process whereby a compiler analyzes the syntax of a program to establish the relationships among operators, operands, and other tokens of a program. Parsing does not involve any semantic analysis.

partitioning: Restructuring a program or algorithm into independent computational segments. The goal is to have multiple CPUs simultaneously work on the independent computational segments.

percentage parallelization: The percentage of CPU expenditure processed in parallel on a single job. It is usually not possible to achieve 100 percent of an application's processing time to be shared equally on all CPUs.

percentage vectorization: The percentage of an application executing in vector mode. This percentage may be calculated as a percentage of CPU time or as the percentage of lines of code (usually Fortran) in vector instructions. The two approaches are not consistent and may give very different percentage ratings. The first calculation method leads to performance improvement as measured by CPU time, while the second method measures the "success rate" of the compiler in converting scalar code to vector code. The former is, of course, the more meaningful hardware performance measure. See also **vectorization**.

perfect-shuffle interconnection: An interconnection structure that connects processors according to a permutation that corresponds to a perfect shuffle of a deck of cards.

physical memory: The actual memory of a computer directly available for fetching or storing of data (contrast with virtual memory).

pipelining: The execution of a set of operations, by a single processor, such that subsequent operations in the sequence can begin execution before previous ones have completed execution. Pipelining is an assembly line approach to data or instruction execution. In modern supercomputers the floating-point operations are often pipelined with memory fetches and stores of the vector data sets.

primary memory: Main memory accessible by the CPU(s) without using input/output processes. See also **secondary memory.**

process: A sequential program in execution. A program by itself is not a process; a program is a passive entity, while a process is an active entity.

pseudovector: A scalar temporary.

r_∞**:** The asymptotic rate for a vector operation as the vector length approaches infinity.

recurrence: A relationship in a DO-loop whereby a computation in one iteration of the loop depends upon completion of a previous iteration of the loop. Such dependencies inhibit vectorization.

reduced instruction set computer (RISC): A philosophy of instruction set design where a small number of simple, fast instructions are implemented rather than a larger number of slower, more complex instructions.

reduction function: A special type of recurrence relationship where a vector of values is reduced to a single scalar result. The variable containing the result is referred to as a "reduction-function scalar." Most compilers recognize several common reduction functions, such as sum, dot product, the minimum (maximum) of the elements of a vectorizable expression, and variants on these themes.

scalar processing: The execution of a program by using instructions that can operate only on a single pair of operands at a time (contrast with vector processing).

scalar temporary: A compiler-created scalar variable that holds the value of a vectorizable expression on each iteration of a DO-loop.

scoreboard: A hardware device that maintains the state of machine resources to enable instructions to execute without conflict at the earliest opportunity.

secondary memory: A larger and slower memory than primary memory. Access to it often requires special instructions, such as I/O instructions. See also **primary memory**.

semaphore: A variable that is used to control access to shared data.

set associative: A cache structure in which all tags in a particular set are compared with an access key in order to access an item in cache. The set may have as few as one element or as many elements as there are lines in the full cache.

SIMD: single instruction stream/multiple data stream architecture. Currently, three such machines dominate the market: the AMT DAP, Goodyear, and the Connection Machine.

SISD: Single instruction stream/single data stream system. Instructions are processed sequentially, with the flow of data from memory to the functional unit and back to memory—the traditional configuration of computers.

speedup: The factor of performance improvement over pure scalar performance. The term is usually applied to performance of either one CPU versus multiple CPUs or vector versus scalar processing. The reported numbers are often misleading because of an inconsistency in reporting the speedup over an original or revised process running in scalar mode.

spin lock: An implementation of the LOCK primitive that causes a processor to retest a semaphore until it changes value. Busy waits will spin until the lock is free.

strength reduction: A process whereby a compiler attempts to replace instructions with less costly instructions that produce identical results. For example, X**2 becomes X*X.

stride: A term derived from the concept of walking (striding) through the data from one location to the next. The term is often used in relation to vector storage. A mathematical vector is represented in a computer as an array of numbers. Most computers use contiguous storage of vectors, locating them by the address of the first word and vector length. Many applications in linear algebra, however, require load and store of vectors with components that do not reside contiguously in memory, such as the rows of a matrix stored in column order. The row elements are spaced in memory by a distance (stride) of the leading dimension of the array representing the matrix. Some vector computers allow vector fetching and storing to occur with randomly stored vectors. An index vector locates successive components. This is often useful in storing the nonzero elements of a sparse vector. See also **vector.**

stripmining: A process used by a compiler on a register-to-register vector processor whereby a

DO-loop of long or variable iteration count is executed in "strips" which are the length of a vector register, except for a "remainder" strip whose length is less. For example, on a Cray computer with a vector length of 64, a loop of iteration count 150 is performed in one strip of length 22 (the remainder) and two strips of length 64.

supercomputer(s): At any given time, that class of general-purpose computers that are faster than their commercial competitors and have sufficient central memory to store the problem sets for which they are designed. Computer memory, throughput, computational rates, and other related computer capabilities contribute to performance. Consequently, a quantitative measure of computer power in large-scale scientific processing does not exist, and a precise definition of supercomputers is difficult to formulate.

superword: A term used on the CYBER 205 and ETA 10 to describe a group of eight 64-bit words, or, alternatively, sixteen 32-bit "half-words." The memory units on these machines generally fetch and store data in superwords (also called "swords"), regardless of the size of the data item referenced by the user program.

synchronization: An operation in which two or more processors exchange information to coordinate their activity.

test and set: An instruction that typically tests a memory location (flag/semaphore) and updates it according to the flag's value. It is atomic in that after the flag is read and before the flag is updated, the CPU executing the instruction will not allow access to the flag.

thrashing: A phenomenon of virtual memory systems that occurs when the program, by the manner in which it is referencing its data and instructions, regularly causes the next memory locations referenced to be overwritten by recent or current instructions. The result is that referenced items are rarely in the machine's physical memory and almost always must be fetched from secondary storage, usually a disk.

thread: A lightweight or small-granularity process.

translation look-aside buffer (TLB): The TLB is the memory cache of the most recently used page table entries within the memory management unit.

true ratio: The frequency with which the "true" branch in a Fortran IF-test occurs. If the true ratio is known at compile time, some compilers can take advantage of this knowledge. However, the true ratio is often data dependent and cannot effectively be dealt with automatically. See also **interactive vectorizer.**

ultracomputer: A shared-memory MIMD computer incorporating a perfect-shuffle interconnection network capable of combining colliding message packets according to simple logical and mathematical functions.

unneeded store: The situation resulting when two or more stores into the same memory location without intermediate reads occur in an optimization block, especially within a DO-loop, such that only the last store need actually be performed. For example, in the loop below, the calculation of A(I) in the first statement will be overwritten by the calculation for A(I) in the last statement. Consequently, A(I) need not be stored when calculated in the first statement but must be kept in a register for the second and third statement.

```
    DO 40 I + 1, N
       A(I) = B(I) * C(I) / (E(I) + F(I))
       X(I) = Y(I) * A(I)
       Z(I) = R(I) + Q(I) * A(I)
       A(I) = X(I) + Y(I) * Z(I)
 40 CONTINUE
```

vector: An ordered list of items in a computer's memory. A simple vector is defined as having a starting address, a length, and a stride. An indirect address vector is defined as having a relative base address and a vector of values to be applied as indexes to the base. Consider the following:

```
    DO 50 I = 1, N
       J = J * J / I
       K = K + 2
       A(I) = B(IB(I)) * C(K) + D(J)
 50 CONTINUE
```

All of these vectors have length N; A and C are vectors with strides of one and two, respectively; B is an indirect address vector with the simple vector IB holding the indexes; and the vector of indirect address indexes of D can be computed at execution time from the initial value of J.

vector processing: A mode of computer processing that acts on operands that are vectors or arrays. Modern supercomputers achieve speed through pipelined arithmetic units. Pipelining, when coupled with instructions designed to process all the elements of a vector rather than one data pair at a time, is known as vector processing.

vector register: A storage device that acts as an intermediate memory between a computer's functional units and main memory.

vectorization: The act of tuning an application code to take advantage of vector architecture. See also **percentage vectorization.**

vectorize: The process whereby a compiler or programmer generates vector instructions for a loop. Vector instructions are typically in the form of machine-specific instructions but can also be expressed as calls to special subroutines or in an array notation.

virtual memory: A memory scheme that provides a programmer with a larger memory than that physically available on a computer. As data items are referenced by a program, the system assigns them to actual physical memory locations. Infrequently referenced items are transparently migrated to and from secondary storage—often, disks. The collection of physical memory locations assigned to a program is its "working set."

virtual processor: An abstraction away from the physical processors of a computer that permits one to program the computer as if it had more processors than are actually available. The physical processors are time-shared among the virtual processors.

VLIW (very long instruction word): Reduced-instruction-set machines with a large number of parallel, pipelined functional units but only a single thread of control. Instead of coarse-grain parallelism of vector machines and multiprocessors, VLIWs provide fine-grain parallelism.

VLSI (very large-scale integration): A manufacturing process that uses a fixed number of manufacturing steps to produce all components and interconnections for hundreds of devices each with millions of transistors.

von Neumann bottleneck: The data path between the processor and memory of a computer that most constrains performance of such a computer.

von Neumann computer: An SISD computer in which one instruction at a time is decoded and performed to completion before the next instruction is decoded. A key point of a von Neumann computer is that both data and program live in the same memory.

working set: See **virtual memory.**

wrap-around scalar: A scalar variable whose value is set in one iteration of a DO-loop and referenced in a subsequent iteration and is consequently recursive. All common reduction-function scalars are wrap-around scalars and usually do not prevent vectorization. All other wrap-around scalars usually do prevent vectorization of the loop in which they appear. In the following, all scalars wrap around except S.

```
DO 60 I = 1, N
   S = T
   T = A(I) * B(I)
   SUM = SUM + T/S
   IF (T.GT.0) THEN
      Q = X(I) + Y(I) / Z(I)
```

```
      ENDIF
      R(I) = Q + P(I)
   60 CONTINUE
```

The scalar Q is a wrap around because on any iteration for which T is less than or equal to 0 is not true, the value used to compute $R(I)$ wraps around from the previous iteration.

write-in cache: A cache in which writes to memory are stored in the cache and written to memory only when a rewritten item is removed from cache. This is also referred to as write-back cache.

write-through cache: A cache in which writes to memory are performed concurrently both in cache and in main memory.

Appendix C

Information on Various High-Performance Computers

C1. The Major Supercomputers

Cray Research and Cray Computer Corp.

Machine	Cycle Time	No. of Proc.	Memory (Mwords)	Peak Perf.	Year Shipped
CRAY-1 †	12.5	1	1	160	1976
CRAY-1/S †	12.5	1	4	160	1979
CRAY X-MP †	9.5	2	4	420	1982
CRAY X-MP †	9.5	4	8	840	1984
CRAY X-MP †	9.5	2	16	840	1985
CRAY-2	4.1	4	256	194	1985
CRAY X-MP †	8.5	4	16	940	1986
CRAY X-MP †	8.5	4	32	940	1988
CRAY Y-MP	6.0	8	32	2700	1988
CRAY-3 ‡	2	16	512+	16000	1991
CRAY C-90 ‡	4	16	64+	16000	1991
CRAY-4 ‡	1	64	1000+	128000	199?

†Machines no longer manufactured.

‡Machines not yet available.

ETA Systems

Machine	Cycle Time	Memory (Mwords)	Peak Perf.	Year Released
ETA-10P †	24	16	375	1987
ETA-10E †	10.5	128	1700	1987
ETA-10Q †	19	16	475	1988
ETA-10G †	7	256	5150	1988

†Machines no longer manufactured.

Fujitsu

Machine	Cycle Time	Memory (Mwords)	Peak Perf.	Year Shipped
VP-2600/10	3.2	-	5000	1990
VP-100E	7	64	429	1984
VP-200E	7	128	857	1984
VP-50E	7	64	286	1985
VP-400E	7	128	1700	1986
VP-30E	7.5	32	133	1987

Hitachi

Machine	Cycle Time	Memory (Mwords)	Peak Perf.	Year Shipped
S-810/10	14	16	315	1983
S-810/20	14	32	620	1983
S-810/5	14	16	160	1986
S-820/60	4	32	1500	1988
S-820/80	4	64	3000	1988

IBM

Machine	Cycle Time	Memory (Mwords)	Peak Perf.	Year Shipped
3090/180 VF	18.5	8	108	1986
3090/150E VF	17.75	8	112	1987
3090/600E VF	17.2	32	696	1988
3090/500E VF	17.2	32	580	1988
3090/400E VF	17.2	32	464	1988
3090/300E VF	17.2	16	348	1988
3090/200E VF	17.2	16	232	1988
3090/280E VF	17.2	16	232	1988
3090/180E VF	17.2	8	116	1988
3090/120E VF	18.5	8	108	1988
3090/600S VF	15	32	800	1988
3090/500S VF	15	32	666	1988
3090/400S VF	15	32	533	1988
3090/300S VF	15	16	400	1988
3090/200S VF	15	16	266	1988
3090/180S VF	15	8	133	1988
3090/600J VF	14.5	64	828	1989
3090/500J VF	14.5	64	690	1989
3090/400J VF	14.5	64	552	1989
3090/300J VF	14.5	32	414	1989
3090/200J VF	14.5	32	275	1989
3090/180J VF	14.5	16	138	1989

NEC

Machine	Cycle Time	Memory (Mwords)	Peak Perf.	Year Shipped
SX2-100	6	16	285	1980
SX2-400	6	32	1300	1985
SX2-200	6	32	570	1986
SX3-44	2.9	200	22000	1990-91

C2. The Major Mini-supercomputers

Machine System	Peak Mflops	Year Intro.
Convex C1	20	1984
Alliant FX-Series	94	1985
Alliant FX-80	188	1987
Convex C2	200	1987
DEC 9000	500	1990

C3. Machine Categories

Companies Building High-Performance Computers

Available	Supers	Vector M-frames	Mini-supers
	CRAY	CDC	Alliant
	Fujitsu	Fujitsu	Convex
	Hitachi	IBM	DEC
	NEC	NAS	
		Unisys	
		Hitachi	
		Honeywell	
Being Built	SSI (Chen)		
	KSR		
	Tera		
No Longer in Business	Chopp	Amer. Super	Culler
	Denelcor	Cydrome	SAXPY
	ETA	Prisma	Astronautics
	Trilogy	SCS	Vitesse
		Key	Gould
			Multiflow
			Supertek

Available	mP	mC	SIMD	Graphics
	Arete	NCUBE	AMT	Apollo
	BBN	Cogent (t)	MasPar	SGI
	Concurrent	Intel iPSC	TMC	Stardent
	Encore	Meiko (t)		
	IP1	Paralax		
	Masscomp	Parsytec (t)		
	Myrias	Suprenum		
	Plexus			
	Sequent			
Being Built				
No Longer in Business	E&S	Symult		
	Elxsi	Topologics (t)		
	Flexible	FPS T-Series		
	Synapse			

C5. Comparison of Supercomputer Technology

Computer	Cooling Technology
CRAY Y-MP	Liquid
CRAY X-MP	Freon
CRAY-2	Liquid Immersion
ETA-10E	Liquid Nitrogen Immersion/Logic
	Chilled Air/Memory
Fujitsu VP-200	Air Cooled
Hitachi S-810/20	Air/Memory & Water/Logic
IBM 3090/VF	Liquid; water
NEC SX/2	Water

Main Memory Sizes

Computer System	No. of. Proc.	Size, MW	No. of Banks	Bank Cycles	Wait Time
CRAY-1	1	4	16	4	50
CRAY X-MP *ECL*	1-4	16	64	4	34
CRAY X-MP *MOS*	1-4	64	64	8	68
CRAY Y-MP	1-8	32	256	5	30
CRAY-2	1-4	512	128	46	184
CRAY-2S	1-4	128	128	13	53
CYBER 205	1	64	8		
ETA-10E	1-8	128	64	20	214
Fujitsu VP-200	1	32	128	8	55
Fujitsu VP-400	1	32	256	8	55
Hitachi S-820	1	32	128	17	70
IBM 3090/VF	1-6	16	16		
NEC SX/2	1	32	512	7	40

Registers and Buffer Size

Computer System	Single CPU Register Configuration (64-bit words)
CRAY-1	8×64
CRAY X-MP	8×64
CRAY Y-MP	8×64
CRAY-2	8×64 plus 16K local memory/processor
CYBER 205	buffer
ETA-10	buffer
Fujitsu VP-200	reconfigurable $8 \times 1024 \ldots 256 \times 16$
Fujitsu VP-400	reconfigurable $16 \times 1024 \ldots 512 \times 16$
Hitachi S-820	32×256
IBM 3090/VF	8×128 plus 8K cache/processor
NEC SX-2	40×256

Paths to Memory

Computer System	No. of Paths to Memory	No. of Paths/ No. of Fl. Pt.	Latency, cycles
CRAY-1	1	.5	15
CRAY X-MP	3	1.5	14
CRAY Y-MP	3	1.5	17
CRAY-2	1	.5	35-50
CYBER 205	3	1.5	50
ETA-10	3	1.5	
Fujitsu VP-200	1*	.5	31-33
Fujitsu VP-400	1	.5	31-33
IBM 3090/VF	1	.5	
NEC SX/2	12	.75	

*The Fujitsu VP-200 has 2 paths for contiguously stored vectors.

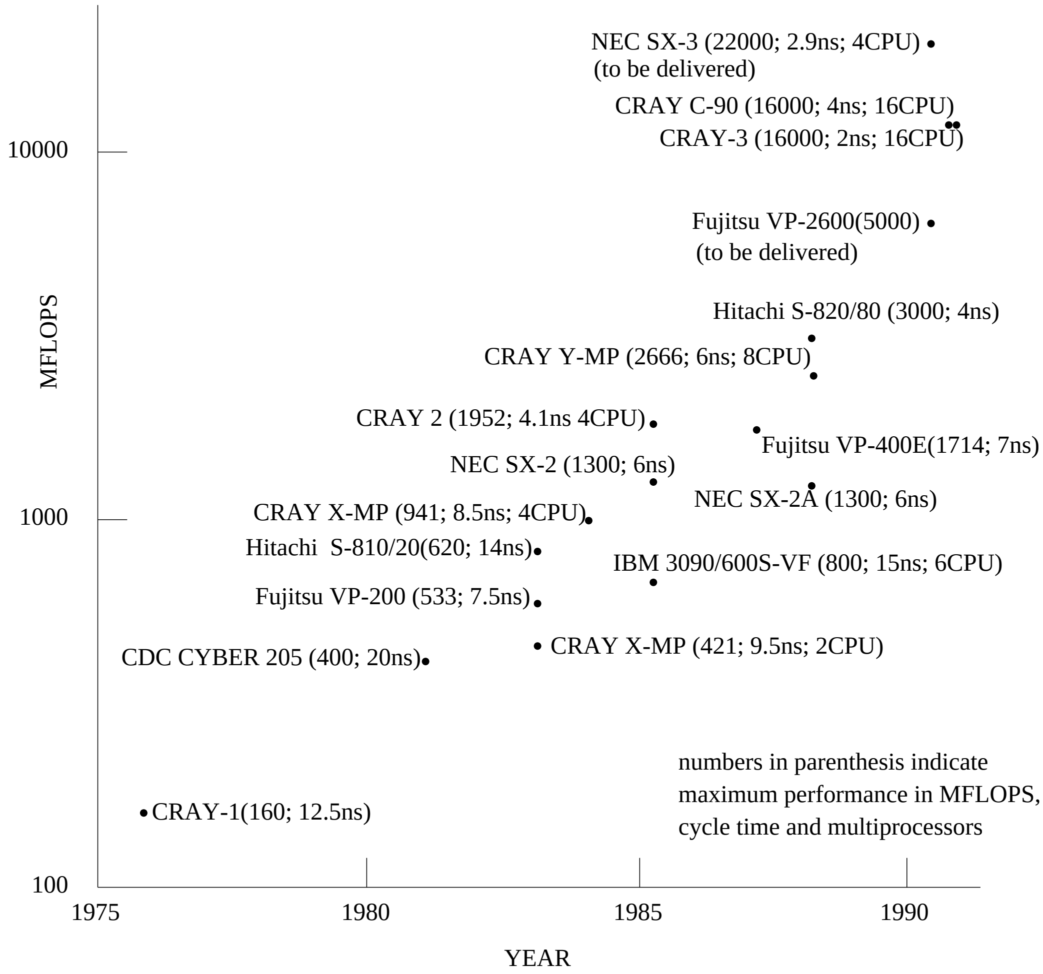

Figure C.1: **Supercomputers over Time**

Appendix D

Level 1, 2, and 3 BLAS Quick Reference

Level 1 BLAS

```
                   dim scalar vector   vector   scalars              5-element prefixes
                                                                     array

SUBROUTINE _ROTG (                                      A, B, C, S )          S, D
SUBROUTINE _ROTMG(                              D1, D2, A, B,        PARAM )  S, D
SUBROUTINE _ROT  ( N,         X, INCX, Y, INCY,               C, S )          S, D
SUBROUTINE _ROTM ( N,         X, INCX, Y, INCY,                      PARAM )  S, D
SUBROUTINE _SWAP ( N,         X, INCX, Y, INCY )                              S, D, C, Z
SUBROUTINE _SCAL ( N,  ALPHA, X, INCX )                                       S, D, C, Z, CS, ZD
SUBROUTINE _COPY ( N,         X, INCX, Y, INCY )                              S, D, C, Z
SUBROUTINE _AXPY ( N,  ALPHA, X, INCX, Y, INCY )                              S, D, C, Z
FUNCTION   _DOT  ( N,         X, INCX, Y, INCY )                              S, D, DS
FUNCTION   _DOTU ( N,         X, INCX, Y, INCY )                              C, Z
FUNCTION   _DOTC ( N,         X, INCX, Y, INCY )                              C, Z
FUNCTION   __DOT ( N,  ALPHA, X, INCX, Y, INCY )                              SDS
FUNCTION   _NRM2 ( N,         X, INCX )                                       S, D, SC, DZ
FUNCTION   _ASUM ( N,         X, INCX )                                       S, D, SC, DZ
FUNCTION   I_AMAX( N,         X, INCX )                                       S, D, C, Z
```

Level 2 BLAS

```
       options           dim   b-width scalar matrix  vector   scalar vector   prefixes

_GEMV (       TRANS,      M, N,         ALPHA, A, LDA, X, INCX, BETA,  Y, INCY ) S, D, C, Z
_GBMV (       TRANS,      M, N, KL, KU, ALPHA, A, LDA, X, INCX, BETA,  Y, INCY ) S, D, C, Z
_HEMV ( UPLO,                N,         ALPHA, A, LDA, X, INCX, BETA,  Y, INCY ) C, Z
_HBMV ( UPLO,                N, K,      ALPHA, A, LDA, X, INCX, BETA,  Y, INCY ) C, Z
_HPMV ( UPLO,                N,         ALPHA, AP,     X, INCX, BETA,  Y, INCY ) C, Z
_SYMV ( UPLO,                N,         ALPHA, A, LDA, X, INCX, BETA,  Y, INCY ) S, D
_SBMV ( UPLO,                N, K,      ALPHA, A, LDA, X, INCX, BETA,  Y, INCY ) S, D
_SPMV ( UPLO,                N,         ALPHA, AP,     X, INCX, BETA,  Y, INCY ) S, D
_TRMV ( UPLO, TRANS, DIAG,   N,                A, LDA, X, INCX )                 S, D, C, Z
_TBMV ( UPLO, TRANS, DIAG,   N, K,             A, LDA, X, INCX )                 S, D, C, Z
_TPMV ( UPLO, TRANS, DIAG,   N,                AP,     X, INCX )                 S, D, C, Z
_TRSV ( UPLO, TRANS, DIAG,   N,                A, LDA, X, INCX )                 S, D, C, Z
_TBSV ( UPLO, TRANS, DIAG,   N, K,             A, LDA, X, INCX )                 S, D, C, Z
_TPSV ( UPLO, TRANS, DIAG,   N,                AP,     X, INCX )                 S, D, C, Z
```

```
       options           dim   scalar vector   vector   matrix  prefixes

_GER  (                   M, N, ALPHA, X, INCX, Y, INCY, A, LDA ) S, D
_GERU (                   M, N, ALPHA, X, INCX, Y, INCY, A, LDA ) C, Z
_GERC (                   M, N, ALPHA, X, INCX, Y, INCY, A, LDA ) C, Z
_HER  ( UPLO,                N, ALPHA, X, INCX,          A, LDA ) C, Z
_HPR  ( UPLO,                N, ALPHA, X, INCX,          AP )     C, Z
_HER2 ( UPLO,                N, ALPHA, X, INCX, Y, INCY, A, LDA ) C, Z
_HPR2 ( UPLO,                N, ALPHA, X, INCX, Y, INCY, AP )     C, Z
_SYR  ( UPLO,                N, ALPHA, X, INCX,          A, LDA ) S, D
_SPR  ( UPLO,                N, ALPHA, X, INCX,          AP )     S, D
_SYR2 ( UPLO,                N, ALPHA, X, INCX, Y, INCY, A, LDA ) S, D
_SPR2 ( UPLO,                N, ALPHA, X, INCX, Y, INCY, AP )     S, D
```

Level 3 BLAS

```
       options                         dim      scalar matrix  matrix  scalar matrix  prefixes

_GEMM (             TRANSA, TRANSB,      M, N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC ) S, D, C, Z
_SYMM ( SIDE, UPLO,                      M, N,    ALPHA, A, LDA, B, LDB, BETA,  C, LDC ) S, D, C, Z
_HEMM ( SIDE, UPLO,                      M, N,    ALPHA, A, LDA, B, LDB, BETA,  C, LDC ) C, Z
_SYRK (       UPLO, TRANS,                  N, K, ALPHA, A, LDA,         BETA,  C, LDC ) S, D, C, Z
_HERK (       UPLO, TRANS,                  N, K, ALPHA, A, LDA,         BETA,  C, LDC ) C, Z
_SYR2K(       UPLO, TRANS,                  N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC ) S, D, C, Z
_HER2K(       UPLO, TRANS,                  N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC ) C, Z
_TRMM ( SIDE, UPLO, TRANSA,        DIAG, M, N,    ALPHA, A, LDA, B, LDB )                S, D, C, Z
_TRSM ( SIDE, UPLO, TRANSA,        DIAG, M, N,    ALPHA, A, LDA, B, LDB )                S, D, C, Z
```

Name	Operation	Prefixes
_ROTG	Generate plane rotation	S, D
_ROTMG	Generate modified plane rotation	S, D
_ROT	Apply plane rotation	S, D
_ROTM	Apply modified plane rotation	S, D
_SWAP	$x \leftrightarrow y$	S, D, C, Z
_SCAL	$x \leftarrow \alpha x$	S, D, C, Z, CS, ZD
_COPY	$y \leftarrow x$	S, D, C, Z
_AXPY	$y \leftarrow \alpha x + y$	S, D, C, Z
_DOT	$dot \leftarrow x^T y$	S, D, DS
_DOTU	$dot \leftarrow x^T y$	C, Z
_DOTC	$dot \leftarrow x^H y$	C, Z
__DOT	$dot \leftarrow \alpha + x^T y$	SDS
_NRM2	$nrm2 \leftarrow \|\|x\|\|_2$	S, D, SC, DZ
_ASUM	$asum \leftarrow \|\|re(x)\|\|_1 + \|\|im(x)\|\|_1$	S, D, SC, DZ
I_AMAX	$amax \leftarrow 1^{st} k \ni \|re(x_k)\| + \|im(x_k)\|$ $= max(\|re(x_i)\| + \|im(x_i)\|)$	S, D, C, Z

_GEMV	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
_GBMV	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
_HEMV	$y \leftarrow \alpha Ax + \beta y$	C, Z
_HBMV	$y \leftarrow \alpha Ax + \beta y$	C, Z
_HPMV	$y \leftarrow \alpha Ax + \beta y$	C, Z
_SYMV	$y \leftarrow \alpha Ax + \beta y$	S, D
_SBMV	$y \leftarrow \alpha Ax + \beta y$	S, D
_SPMV	$y \leftarrow \alpha Ax + \beta y$	S, D
_TRMV	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
_TBMV	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
_TPMV	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
_TRSV	$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$	S, D, C, Z
_TBSV	$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$	S, D, C, Z
_TPSV	$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$	S, D, C, Z

_GER	$A \leftarrow \alpha xy^T + A, A - m \times n$	S, D
_GERU	$A \leftarrow \alpha xy^T + A, A - m \times n$	C, Z
_GERC	$A \leftarrow \alpha xy^H + A, A - m \times n$	C, Z
_HER	$A \leftarrow \alpha xx^H + A$	C, Z
_HPR	$A \leftarrow \alpha xx^H + A$	C, Z
_HER2	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
_HPR2	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
_SYR	$A \leftarrow \alpha xx^T + A$	S, D
_SPR	$A \leftarrow \alpha xx^T + A$	S, D
_SYR2	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D
_SPR2	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D

Name	Operation	Prefixes
_GEMM	$C \leftarrow \alpha op(A)op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n$	S, D, C, Z
_SYMM	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H$	S, D, C, Z
_HEMM	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T$	C, Z
_SYRK	$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$	S, D, C, Z
_HERK	$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$	C, Z
_SYR2K	$C \leftarrow \alpha AB^H + \alpha BA^H + \beta C, C \leftarrow \alpha A^H B + \alpha B^H A + \beta C, C - n \times n$	S, D, C, Z
_HER2K	$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C, C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C$	C, Z
_TRMM	$B \leftarrow \alpha op(A)B, B \leftarrow \alpha Bop(A), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z
_TRSM	$B \leftarrow \alpha op(A^{-1})B, B \leftarrow \alpha Bop(A^{-1}), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z

Notes

Meaning of prefixes

S - REAL	C - COMPLEX	
D - DOUBLE PRECISION	Z - COMPLEX*16	(this may not be supported by all machines)

For the Level 2 BLAS a set of extended-precision routines with the prefixes ES, ED, EC, EZ may also be available.

Level 1 BLAS

In addition to the listed routines there are two further extended-precision dot product routines DQDOTI and DQDOTA.

Level 2 and Level 3 BLAS

Matrix types

GE - GEneral	GB - General Band	
SY - SYmmetric	SB - Symmetric Band	SP - Symmetric Packed
HE - HErmitian	HB - Hermitian Band	HP - Hermitian Packed
TR - TRiangular	TB - Triangular Band	TP - Triangular Packed

Options

The dummy options arguments are declared as CHARACTER*1 and may be passed as character strings.

TRANS_	= '**N**o transpose', '**T**ranspose', '**C**onjugate transpose' (X, X^T, X^H)
UPLO	= '**U**pper triangular', '**L**ower triangular'
DIAG	= '**N**on-unit triangular', '**U**nit triangular'
SIDE	= '**L**eft', '**R**ight' (A or op(A) on the left, or A or op(A) on the right)

For real matrices, TRANS_ = 'T' and TRANS_ = 'C' have the same meaning.
For Hermitian matrices, TRANS_ = 'T' is not allowed.
For complex symmetric matrices, TRANS_ = 'H' is not allowed.

Bibliography

[1] J. O. Aasen. On the reduction of a symmetric matrix to tridiagonal form. *BIT*, 11:233–242, 1971.

[2] G. Alaghband. Parallel pivoting combined with parallel reduction and fill-in control. *Parallel Computing*, 11:201–221, 1989.

[3] F. L. Alvarado. Manipulation and visualisation of sparse matrices. *ORSA J. Computing*, 2:186–207, 1989.

[4] F.L. Alvarado and H. Da g. Incomplete partitioned inverse preconditioners. Technical report, Department of Electrical and Computer Engineering, University of Wisconsin, Madison, 1994.

[5] F.L. Alvarado and R. Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM J. Sci. Comput.*, 14:446–460, 1993.

[6] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Computing Conference*, volume 30, pages 483–485, 1967.

[7] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Analysis and Applications*, 17(4):886–905, 1996.

[8] Patrick R. Amestoy, Michel J. Daydé, Iain S. Duff, and Pierre Morère. Linear algebra calculations on a virtual shared memory computer. *Int J. High Speed Computing*, 7(1):21–43, 1995.

[9] Patrick R. Amestoy and Iain S. Duff. Vectorization of a multiprocessor multifrontal code. *Int. J. of Supercomputer Applics.*, 3:41–59, 1989.

[10] Patrick R. Amestoy and Iain S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. Supercomputer Applics*, 7:64–82, 1993.

[11] Patrick R. Amestoy and Iain S. Duff. MA41: a parallel package for solving sparse unsymmetric sets of linear equations. Technical Report (to appear), CERFACS, Toulouse, France, 1997.

[12] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. LAPACK working note #20: LAPACK: A portable linear algebra library for high-performance computers. Computer Science Technical Report CS-90-105, University of Tennessee, May 1990.

[13] P. Arbenz and G. Golub. On the spectral decomposition of Hermitian matrices modified by row rank perturbations with applications. *SIAM J. Matrix Anal. Appl.*, 9(1):40–58, January 1988.

[14] M. Arioli, I.S. Duff, J. Noailles, and D. Ruiz. A block projection method for sparse equations. *SIAM J. Sci. Stat. Comput.*, 13:47–70, 1992.

[15] Mario Arioli and Iain S. Duff. Experiments in tearing large sparse systems. In M. G. Cox and S. Hammarling, editors, *Reliable Numerical Computation*, pages 207–226, Oxford, 1990. Oxford University Press.

[16] W. E. Arnoldi. The principle of minimized iteration in the solution of the matrix eigenproblem. *Quart. Appl. Math.*, 9:17–29, 1951.

[17] S. F. Ashby. CHEBYCODE: A Fortran implementation of Manteuffel's adaptive Chebyshev algorithm. Report UIUCDCS-R-85-1203, University of Illinois, Urbana, IL, May 1985.

[18] S.F. Ashby. Minimax polynomial preconditioning for Hermitian linear systems. *SIAM J. Matrix Analysis and Applications*, 12:766–789, 1991.

[19] C. Ashcraft. A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems. Report ETA-TR-51, ETA, 1987.

[20] C. Ashcraft and R. Grimes. On vectorizing incomplete factorizations and SSOR preconditioners. *SIAM J. Sci. Statist. Comput.*, 9(1):122–151, 1988.

[21] C. Ashcraft and J. W. H. Liu. Robust ordering of sparse matrices using multisection. Technical Report ISSTECH-96-002, Boeing Information and Support Services, Seattle, 1996. Also Report CS-96-01, Department of Computer Science, York University, Ontario, Canada.

[22] O. Axelsson. Solution of linear systems of equations: Iterative methods. In V. A. Barker, editor, *Sparse Matrix Techniques: Copenhagen*, pages 1–51, Berlin, 1977. Springer-Verlag.

[23] O. Axelsson. Conjugate gradient type methods for unsymmetric and inconsistent systems of equations. *Lin. Alg. and its Appl.*, 29:1–16, 1980.

[24] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, Cambridge, 1994.

[25] O. Axelsson and V.A. Barker. *Finite Element Solution of Boundary Value Problems. Theory and Computation*. Academic Press, New York, NY, 1984.

[26] O. Axelsson and G. Lindskog. On the eigenvalue distribution of a class of preconditioning methods. *Numer. Math.*, 48:479–498, 1986.

[27] O. Axelsson and N. Munksgaard. Analysis of incomplete factorizations with fixed storage allocation. In D. Evans, editor, *Preconditioning Methods - Theory and Applications*, pages 265–293. Gordon and Breach, New York, 1983.

[28] O. Axelsson and P. S. Vassilevski. A black box generalized conjugate gradient solver with inner iterations and variable-step preconditioning. *SIAM J. Matrix Anal. Appl.*, 12(4):625–644, 1991.

[29] I. Babuska. Numerical stability in problems of linear algebra. *SIAM J. Numer. Anal.*, 9(1):53–77, 1972.

[30] Z. Bai, D. Hu, and L. Reichel. A Newton basis GMRES implementation. *IMA J. Numer. Anal.*, 14:563–581, 1991.

[31] H.B. Bakoglu, G.F. Grohoski, and R.K. Montoye. The IBM RISC System/6000 processor: Hardware overview. *IBM Journal of Research and Development*, 34:12–23, 1990.

[32] R. E. Bank and T. F. Chan. An analysis of the composite step biconjugate gradient method. *Num. Math.*, 66:295–319, 1993.

[33] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 1994.

[34] R. E. Benner, G. R. Montry, and G. G. Weigand. Concurrent multifrontal methods: shared memory, cache, and frontwidth issues. *Int J. Supercomputer Applications*, 1:26–44, 1987.

[35] M. Benzi, C.D. Meyer, and M. Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Sci. Comput.*, 17:1135–1149, 1996.

[36] M. Benzi, D.B. Szyld, and A. van Duin. Orderings for incomplete factorization preconditioning of nonsymmetric problems. Technical Report 97-91, Temple University, Department of Mathematics, Philadelphia, PA, 1997.

[37] H. Berryman, J. Saltz, W. Gropp, and R. Mirchandaney. Krylov methods preconditioned with incompletely factored matrices on the CM-2. ICASE Report 89-54, NASA Langley Research Center, Hampton, VA, 1989.

[38] C. Bischof and C. Van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Statist. Comput.*, 8(2):s2–s13, March 1987.

[39] A. Bjorck and T. Elfving. Accelerated projection methods for computing pseudo-inverse solutions of systems of linear equations. *BIT*, 19:145–163, 1979.

[40] R. Bramley and A. Sameh. Row projection methods for large nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:168–193, 1992.

[41] C. Brezinski and M. Redivo-Zaglia. Treatment of near breakdown in the CGS algorithm. *Numerical Algorithms*, 7:33–73, 1994.

[42] G. C. Broyden. A new method of solving nonlinear simultaneous equations. *Comput. J.*, 12:94–99, 1969.

[43] A.M. Bruaset. *A Survey of Preconditioned Iterative Methods*. Longman Scientific & and Technical, Harlow, UK, 1995.

[44] G. Brussino and V. Sonnad. A comparison of direct and preconditioned iterative techniques for sparse unsymmetric systems of linear equations. *Int. J. for Num. Methods in Eng.*, 28:801–815, 1989.

[45] J. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving systems of linear equations. *Math. Comp.*, 31:163–179, 1977.

[46] J. R. Bunch, L. Kaufman, and B. N. Parlett. Decomposition of a symmetric matrix. *Numerische Mathematik*, 27:95–110, 1976.

[47] R. Butler and E. Lusk. Monitors, Messages, and Clusters: The P4 Parallel Programming System. *Parallel Computing*, 20:547–64, April 1994.

[48] D. Calahan, J. J. Dongarra, and D. Levine. Vectorizing compilers: A test suite and results. In *Supercomputer 88*, pages 98–105. IEEE Press, 1988.

[49] R. Calkin, R. Hempel, H. Hoppe, and P. Wypior. Portable Programming with the PARMACS Message–Passing Library. *Parallel Computing, Special issue on message–passing interfaces*, 20:615–32, April 1994.

[50] T.F. Chan. Fourier analysis of relaxed incomplete factorization procedures. *SIAM J. Sci. Stat. Comput.*, 12:668–680, 1991.

[51] T.F. Chan and D.Goovaerts. A note on the efficiency of domain decomposed incomplete factorizations. *SIAM J. Sci. Stat. Comp.*, 11:794–803, 1990.

[52] T.F. Chan and H.A. Van der Vorst. Approximate and incomplete factorizations. In D.E. Keyes, A. Sameh, and V. Venkatakrishnan, editors, *Parallel Numerical Algorithms*, ICASE/LaRC Interdisciplinary Series in Science and Engineering, pages 167–202. Kluwer, Dordrecht, 1997.

[53] S. C. Chen, D. J. Kuck, and A. H. Sameh. Practical parallel band triangular system solvers. *ACM Trans. Math. Softw.*, 4:270–277, 1978.

[54] H. Choi and D.B. Szyld. Threshold ordering for preconditioning nonsymmetric problems with highly varying coefficients. Technical Report 96-51, Department of Mathematics, Temple University, Philadelphia, 1996.

[55] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. "scalapack: A portable linear algebra library for distributed memory computers - design issues and performance". Technical Report UT CS-95-283, LAPACK Working Note #95, University of Tennessee, 1995.

[56] J. "Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R.C." Whaley. "A Proposal for a Set of Parallel Basic Linear Algebra Subprograms". Technical Report UT CS-95-292, LAPACK Working Note #100, University of Tennessee, 1995.

[57] J. Choi, J. Dongarra, and D. Walker. "Parallel Matrix Transpose Algorithms on Distributed Concurrent Computers". Technical Report UT CS-93-215, LAPACK Working Note #65, University of Tennessee, 1993.

[58] E. Chow and Y. Saad. Approximate inverse preconditioners for general sparse matrices. Technical Report Research Report UMSI 94/101, University of Minnesota Supercomputing Institute, Minneapolis, Minnesota, 1994.

[59] A. T. Chronopoulos and C. W. Gear. s-step iterative methods for symmetric linear systems. *J. Comput. Appl. Math.*, 25(2):153–168, 1989.

[60] A. T. Chronopoulos and S. K. Kim. s-Step Orthomin and GMRES implemented on parallel computers. Technical Report 90/43R, UMSI, Minneapolis, 1990.

[61] K. Andrew Cliffe, Iain S. Duff, and Jennifer A. Scott. Performance issues for frontal schemes on a cache-based high performance computer. Technical Report RAL-TR-97-001, Rutherford Appleton Laboratory, 1997.

[62] P. Concus and G. H. Golub. A generalized Conjugate Gradient method for nonsymmetric systems of linear equations. Technical Report STAN-CS-76-535, Stanford University, Stanford, CA, 1976.

[63] P. Concus, G. H. Golub, and G. Meurant. Block preconditioning for the conjugate gradient method. *SIAM J. Sci. Statist. Comput.*, 6(1):220–252, 1985.

[64] P. Concus and G. Meurant. On computing INV block preconditionings for the conjugate gradient method. *BIT*, pages 493–504, 1986.

[65] J. M. Conroy, S. G. Kratzer, and R. F. Lucas. Data-parallel sparse matrix factorization. In J. G. Lewis, editor, *Proceedings 5th SIAM Conference on Linear Algebra*, pages 377–381, Philadelphia, 1994. SIAM Press.

[66] J. D. F. Cosgrove, J. C. Diaz, and A. Griewank. Approximate inverse preconditionings for sparse linear systems. *Intern. J. Computer Math.*, 44:91–110, 1992.

[67] L. Crone and H. van der Vorst. Communication aspects of the conjugate gradient method on distributed-memory machines. *Supercomputer*, X(6):4–9, 1993.

[68] M. Crouzeix, B. Philippe, and M. Sadkane. The Davidson method. *SIAM J. Sci. Comp.*, 15:62–76, 1994.

[69] J. Cullum and A. Greenbaum. Relations between Galerkin and norm-minimizing iterative methods for solving linear systems. *SIAM J. Matrix Analysis and Appl.*, 17:223–247, 1996.

[70] E.R. Davidson. The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real symmetric matrices. *J. Comp. Phys.*, 17:87–94, 1975.

[71] E.R. Davidson. Monster matrices: their eigenvalues and eigenvectors. *Computers in Physics*, 7:519–522, 1993.

[72] Timothy A. Davis and Iain S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. Technical Report TR-95-020, Computer and Information Science Department, University of Florida, 1995.

[73] Timothy A. Davis and Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Analysis and Applications*, 18(1):140–158, 1997.

[74] Timothy A. Davis and P. C. Yew. A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization. *SIAM J. Matrix Analysis and Applications*, 11:383–402, 1990.

[75] M.J. Daydé, J.-Y. L'Excellent, and N.I.M. Gould. On the preprocessing of sparse unassembled linear systems for efficient solution using element-by-element preconditioners. Technical Report RT/APO/96/2, Département Informatique, ENSEEIHT-IRIT, Toulouse, 1996.

[76] E. F. D'Azevedo, F. A. Forsyth, and W. P. Tang. Ordering methods for preconditioned conjugate gradient methods applied to unstructured grid problems. *SIAM J. Matrix Analysis and Applications*, 13:944–961, 1992.

[77] E.F. D'Azevedo, P.A. Forsyth, and W.P. Tang. Drop tolerance preconditioning for incompressible viscous flow. *Int'l J. Comp. Math.*, 44:301–312, 1992.

[78] E. De Sturler. *Iterative methods on distributed memory computers.* PhD thesis, Delft University of Technology, Delft, the Netherlands, 1994.

[79] E. De Sturler and D. R. Fokkema. Nested Krylov methods and preserving the orthogonality. In N. Duane Melson, T.A. Manteuffel, and S.F. McCormick, editors, *Sixth Copper Mountain Conference on Multigrid Methods*, volume Part 1 of *NASA Conference Publication 3324*, pages 111–126. NASA, 1993.

[80] E. De Sturler and H.A. Van der Vorst. Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers. *J. Appl. Num. Math.*, 18:441–459, 1995.

[81] J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. Prospectus for the development of a linear algebra library for high-performance computers. Mathematics and Computer Science Division Report ANL-MCS-TM-97, Argonne National Laboratory, September 1987.

[82] J. Demmel, M. Heath, and H. Van der Vorst. Parallel numerical linear algebra. In *Acta Numerica 1993.* Cambridge University Press, Cambridge, 1993.

[83] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. Technical Report UCB//CSD-95-883, Computer Science Division, U. C. Berkeley, Berkeley, California, July 1995.

[84] D. Dodson and J. Lewis. Issues relating to extension of the Basic Linear Algebra Subprograms. *ACM SIGNUM Newsletter*, 20(1):2–18, 1985.

[85] J. Dongarra and T. Dunigan. Message-passing performance of various computers. *Concurrency—Practice and Experience*, To Appear:??–??, 1997.

[86] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM Framework Supports Heterogeneous Network Computing. *Computers in Physics*, 7(2):166–75, April 1993.

[87] J. Dongarra and R. C. Whaley. "A User's Guide to the BLACS v1.0". Technical Report UT CS-95-281, LAPACK Working Note #94, University of Tennessee, 1995.

[88] J. J. Dongarra, editor. *Experimental Parallel Computing Architectures*. North-Holland, New York, 1987.

[89] J. J. Dongarra. Performance of various computers using standard linear equations software in a Fortran environment. Computer Science Technical Report CS-89-85, University of Tennessee, March 1990.

[90] J. J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users' Guide*. SIAM Pub., Philadelphia, 1979.

[91] J. J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16:1–17, 1990.

[92] J. J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An extended set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 14:1–17, 1988.

[93] J. J. Dongarra and S. C. Eisenstat. Squeezing the most out of an algorithm in Cray Fortran. *ACM Trans. Math. Softw.*, 10(3):221–230, 1984.

[94] J. J. Dongarra and S. C. Eisenstat. Squeezing the most out of an algorithm in CRAY-Fortran. *ACM Trans. Math. Softw.*, 10:221–230, 1984.

[95] J. J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30(5):403–407, July 1987.

[96] J. J. Dongarra, F. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Rev.*, 26:91–112, January 1984.

[97] J. J. Dongarra and A. Hinds. Unrolling loops in Fortran. *Software—Practice and Experience*, 9:219–226, 1979.

[98] J. J. Dongarra and L. Johnsson. Solving banded systems on a parallel processor. *Parallel Computing*, 5:219–246, 1987.

[99] J. J. Dongarra, A. Karp, K. Kennedy, and D. Kuck. 1989 Gordon Bell Prize. *IEEE Software*, pages 100–110, May 1990.

[100] J. J. Dongarra and D. Sorensen. A portable environment for developing parallel Fortran programs. *Parallel Computing*, 5:175–186, July 1987.

[101] J. J. Dongarra, D. Sorensen, K. Connolly, and J. Patterson. Programming methodology and performance issues for advanced computer architectures. *Parallel Computing*, 8:41–58, 1988.

[102] J. J. Dongarra and D. C. Sorensen. Linear algebra on high-performance computers. In U. Schendel, editor, *Proceedings of Parallel Computing '85*, pages 3–32, New York, 1986. North Holland.

[103] J. J. Dongarra and D. C. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM J. Sci. Statist. Comput.*, 8(2):s139–s154, March 1987.

[104] J.J. Dongarra and I.S. Duff. Advanced architecture computers. Technical Report CS-89-90, University of Tennessee, 1989.

[105] P. Dubois, A. Greenbaum, and G. H. Rodrigue. Approximating the inverse of a matrix for use in iterative algorithms on vector processors. *Computing*, 22:257–268, 1979.

[106] P. Dubois and G. Rodrigue. An analysis of the recursive doubling algorithm. In Kuck et al., editors, *High speed computer and algorithm organization*, New York, 1977. Academic Press.

[107] I. S. Duff. Data structures, algorithms and software for sparse matrices. In D. J. Evans, editor, *Sparsity and its Applications*, pages 1–29. Cambridge University Press, Cambridge, 1985.

[108] I. S. Duff, A. M. Erisman, and J.K.Reid. *Direct methods for sparse matrices.* Oxford University Press, London, 1986.

[109] I. S. Duff and G. A. Meurant. The effect of ordering on preconditioned conjugate gradient. *BIT*, 29:635–657, 1989.

[110] Iain S. Duff. MA32 – A package for solving sparse unsymmetric systems using the frontal method. Technical Report AERE R11009, Her Majesty's Stationery Office, London, 1981.

[111] Iain S. Duff. Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core. *SIAM J. Scientific and Statistical Computing*, 5:270–280, 1984.

[112] Iain S. Duff. The solution of sparse linear equations on the CRAY-1. In Janusz S. Kowalik, editor, *Proceedings of the NATO Advanced Research Workshop on High-Speed Computation, held at Julich, Federal Republic of Germany, June 20-22, 1983*, NATO ASI series. Series F, Computer and Systems Sciences; Vol. 7, pages 293–309, Berlin, 1984. Springer-Verlag.

[113] Iain S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3:193–204, 1986.

[114] Iain S. Duff. The parallel solution of sparse linear equations. In W. Handler, D. Haupt, R. Jeltsch, W. Juling, and O. Lange, editors, *CONPAR 86: Conference on Algorithms and Hardware for Parallel Processing, Aachen, September 17–19, 1986: Proceedings*, Lecture Notes in Computer Science 237, pages 18–24, Berlin, 1986. Springer-Verlag.

[115] Iain S. Duff. Multiprocessing a sparse matrix code on the Alliant FX/8. *J. Comput. Appl. Math.*, 27:229–239, 1989.

[116] Iain S. Duff. Parallel algorithms for sparse matrix solution. In D. J. Evans and C Sutti, editors, *Parallel computing. Methods algorithms and applications*, pages 73–82, Bristol, 1989. Adam Hilger Ltd.

[117] Iain S. Duff. The solution of augmented systems. In D. F. Griffiths and G. A. Watson, editors, *Numerical Analysis 1993, Proceedings of the 15th Dundee Conference, June-July 1993*, Pitman Research Notes in Mathematics Series. **303**, pages 40–55, Harlow, England, 1994. Longman Scientific & Technical.

[118] Iain S. Duff. Sparse numerical linear algebra: direct methods and preconditioning. In I. S. Duff and G. A. Watson, editors, *The State of the Art in Numerical Analysis*, pages 27–62, Oxford, 1997. Oxford University Press.

[119] Iain S. Duff, A. M. Erisman, C. W. Gear, and John K. Reid. Sparsity structure and Gaussian elimination. *SIGNUM Newsletter*, 23(2):2–8, April 1988.

[120] Iain S. Duff, A. M. Erisman, and John K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, England, 1986.

[121] Iain S. Duff, Nicholas I. M. Gould, Marc Lescrenier, and John K. Reid. The multifrontal method in a parallel environment. In M. G. Cox and S. Hammarling, editors, *Reliable Numerical Computation*, pages 93–111, Oxford, 1990. Oxford University Press.

[122] Iain S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15(1):1–14, March 1989.

[123] Iain S. Duff, Roger G. Grimes, and John G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (Release I). Technical Report RAL 92-086, Rutherford Appleton Laboratory, 1992.

[124] Iain S. Duff, Roger G. Grimes, and John G. Lewis. The Rutherford-Boeing Sparse Matrix Collection. Technical report, Rutherford Appleton Laboratory, 1997. To appear.

[125] Iain S. Duff and S. L. Johnsson. Node orderings and concurrency in structurally-symmetric sparse problems. In Graham F. Carey, editor, *Parallel Supercomputing: Methods, Algorithms and Applications*, pages 177–189. J. Wiley and Sons, New York, 1989.

[126] Iain S. Duff, Michele Marrone, Giuseppe Radicati, and Carlo Vittoli. Level 3 Basic Linear Algebra Subprograms for sparse matrices: a user level interface. Technical Report RAL-TR-95-049 (Revised), RAL, 1997. To appear in ACM Trans Math Softw.

[127] Iain S. Duff and John K. Reid. MA27 – A set of Fortran subroutines for solving sparse symmetric sets of linear equations. Technical Report AERE R10533, Her Majesty's Stationery Office, London, 1982.

[128] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 9:302–325, 1983.

[129] Iain S. Duff and John K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM J. Scientific and Statistical Computing*, 5:633–641, 1984.

[130] Iain S. Duff and John K. Reid. MA47, a Fortran code for direct solution of indefinite sparse symmetric linear systems. Technical Report RAL 95-001, Rutherford Appleton Laboratory, 1995.

[131] Iain S. Duff and John K. Reid. The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Trans. Math. Softw.*, 22(2):187–226, 1996.

[132] Iain S. Duff and John K. Reid. Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 22(2):227–257, 1996.

[133] Iain S. Duff, John K. Reid, and Jennifer A. Scott. The use of profile reduction algorithms with a frontal code. *Int J. Numerical Methods in Engineering*, 28:2555–2568, 1989.

[134] Iain S. Duff and Jennifer A. Scott. MA42 – a new frontal code for solving sparse unsymmetric systems. Technical Report RAL 93-064, Rutherford Appleton Laboratory, 1993.

[135] Iain S. Duff and Jennifer A. Scott. The use of multiple fronts in Gaussian elimination. In J. G. Lewis, editor, *Proceedings of the Fifth SIAM Conference on Applied Linear Algebra*, pages 567–571, Philadelphia, 1994. SIAM Press.

[136] Iain S. Duff and Jennifer A. Scott. A comparison of frontal software with other sparse direct solvers. Technical Report RAL-TR-96-102, Rutherford Appleton Laboratory, 1996.

[137] Iain S. Duff and Jennifer A. Scott. The design of a new frontal code for solving sparse unsymmetric systems. *ACM Trans. Math. Softw.*, 22(1):30–45, 1996.

[138] T. H. Dunigan. Performance of the Intel i860 Hypercube. Engineering, Physics, and Mathematics Division ORNL/TM-11491, Oak Ridge National Laboratory, April 1990.

[139] T. H. Dunigan. Early experiences and performance of the Intel Paragon. Technical report, Oak Ridge National Laboratory, 1993. ORNL/TM-12194.

[140] R. P. Dupont, T. Kendall and H. Rachford. An approximate factorization procedure for solving self-adjoint elliptic difference equations. *SIAM J. Numer. Anal.*, 53:559–573, 1968.

[141] L.C. Dutto. The effect of ordering on preconditioned GMRES algorithm for solving the Navier-Stokes equations. *Int. J. Numerical Methods in Engineering*, 36:457–497, 1993.

[142] V. Eijkhout. Analysis of parallel incomplete point factorizations. *Lin. Alg. Appl.*, 154-156:723–740, 1991.

[143] T. Eirola and O. Nevanlinna. Accelerating with rank-one updates. *Lin. Alg. and its Appl.*, 121:511–520, 1989.

[144] S. C. Eisenstat. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM J. Sci. Statist. Comput.*, 2:1–4, 1981.

[145] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package, I: The symmetric codes. *Int J. Numerical Methods in Engineering*, 18:1145–1151, 1982.

[146] S. C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in unsymmetric sparse symbolic factorization. *SIAM J. Matrix Analysis and Applications*, 13:202–211, 1992.

[147] H. C. Elman. *Iterative methods for large sparse nonsymmetric systems of linear equations.* PhD thesis, Computer Science Dept., Yale University, New Haven, CT, 1982.

[148] H.C. Elman. A stability analysis of incomplete LU factorization. *Math. Comp.*, 47:191–217, 1986.

[149] Engineering, Scientific Subroutine Library-Guide, and Reference. Release 3, Order No. sc23-0184, IBM, Kingston, NY, 1988.

[150] V. Faber and T. A. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J. Numer. Analysis*, 21(2):352–362, 1984.

[151] B. Fischer. *Polynomial based iteration methods for symmetric linear systems.* Advances in Numerical Mathematics. Wiley and Teubner, Chichester, Stuttgart, 1996.

[152] R. Fletcher. *Conjugate gradient methods for indefinite systems*, volume 506 of *Lecture Notes Math.*, pages 73–89. Springer-Verlag, Berlin–Heidelberg–New York, 1976.

[153] M. Flynn. Very high speed computing systems. *Proc. IEEE*, 54:1901–1909, 1966.

[154] D.R. Fokkema. *Subspace methods for linear, nonlinear, and eigen problems.* PhD thesis, Utrecht University, Utrecht, 1996.

[155] D.R. Fokkema, G.L.G. Sleijpen, and H.A. van der Vorst. Jacobi-Davidson style QR and QZ algorithms for the partial reduction of matrix pencils. Technical Report Preprint 941, Mathematical Institute, Utrecht University, 1996.

[156] G. E. Forsythe and E. G. Strauss. On best conditioned matrices. *Proc. Amer. Math. Soc.*, 6:340–345, 1955.

[157] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard . *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), 1994. Special issue on MPI. Also available electronically, the url is `ftp://www.netlib.org/mpi/mpi-report.ps`.

[158] R. W. Freund, M. H. Gutknecht, and N. M. Nachtigal. An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices. *SIAM J. Sci. Comput.*, 14:137–158, 1993.

[159] R. W. Freund and N. M. Nachtigal. An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices, part 2. Technical Report 90.46, RIACS, NASA Ames Research Center, 1990.

[160] R. W. Freund and N. M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Num. Math.*, 60:315–339, 1991.

[161] Roland Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Comput.*, 14:470–482, 1993.

[162] K. Gallivan, W. Jalby, U. Meier, and A. H. Sameh. Impact of hierarchical memory systems on linear algebra algorithmic design. *The Int. Journal of Supercomputer Appl.*, 21:12–48, 1988.

[163] K. Gallivan, R. Plemmons, and A. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, 1990.

[164] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. *Matrix Eigensystem Routines—EISPACK Guide Extension, Lecture Notes in Computer Science*, volume 51. Springer-Verlag, New York, 1977.

[165] D. M. Gay. Electronic mail distribution of linear programming test problems. Mathematical Programming Society. COAL Newsletter, 1985.

[166] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994. The book is available electronically, the url is `ftp://www.netlib.org/pvm3/book/pvm-book.ps`.

[167] A. George and M. T. Heath. Solution of sparse linear least squares problems using Givens rotations. *Linear Algebra and its Applications*, 34:69–83, 1980.

[168] A. George, M. T. Heath, J. W. H. Liu, and E. Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM J. Scientific and Statistical Computing*, 9:327–340, 1988.

[169] A. George, M. T. Heath, J. W. H. Liu, and E. Ng. Solution of sparse positive definite systems on a hypercube. *J. Comput. Appl. Math.*, 27:129–156, 1989.

[170] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[171] A. George and J. W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 1989.

[172] A. George, J. W. H. Liu, and E. G. Ng. User's guide for SPARSPAK: Waterloo sparse linear equations package. Technical Report CS-78-30 (Revised), University of Waterloo, Canada, 1980.

[173] A. George and E. Ng. An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM J. Scientific and Statistical Computing*, 6:390–409, 1985.

[174] E. Giladi, G.H. Golub, and J.B. Keller. Inner and outer iterations for the Chebyshev algorithm. Technical Report SCCM-95-10, Computer Science Department, Stanford University, Stanford, CA, 1995.

[175] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins Press, Baltimore, Maryland, 2nd edition, 1989.

[176] G. H. Golub and M. L. Overton. The convergence of inexact Chebyshev and Richardson iterative methods for solving linear systems. *Numer. Math.*, 53:571–594, 1988.

[177] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, 1989.

[178] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, 1996.

[179] N.I.M. Gould and J.A. Scott. On approximate-inverse preconditioners. Technical Report RAL-TR-95-026, Rutherford Appleton Laboratory, 1995.

[180] A. Greenbaum and Z. Strakos. Matrices that generate the same Krylov residual spaces. In G. Golub, A. Greenbaum, and M. Luskin, editors, *Recent Advances in Iterative Methods*, The IMA Volumes in Mathematics and its Applications, 60, pages 95–118. Springer Verlag, Berlin, 1994.

[181] R. G. Grimes, D. R. Kincaid, and D. M. Young. ITPACK 2.0 user's guide. Technical report, Univ. of Texas, Austin, Texas, 1979.

[182] M. Grote and H. Simon. Parallel preconditioning and approximate inverses on the connection machine. In R.F. Sincovec, D.E. Keyes, M.R. Leuze, L.R. Petzold, and D.A. Reed, editors, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 519–523, Philadelphia, 1993. SIAM.

[183] M.J. Grote and T. Huckle. Parallel preconditionings with sparse approximate inverses. *SIAM J. Scient. Comput.*, 18:838–853, 1997.

[184] A. Gupta and V. Kumar. A scalable parallel algorithm for sparse matrix factorization. Technical Report TR-94-19, Department of Computer Science, University of Minnesota, 1994.

[185] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. Technical Report UMSI 96/97, University of Minnesota, Supercomputer Institute, 1996.

[186] J. Gustafson. Reevaluating Amdahl's law. *Comm. ACM*, 31:532–533, 1988.

[187] J. Gustafson, G. R. Montry, and R. E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Statist. Comput.*, 9:609–638, 1988.

[188] I. Gustafsson. A class of first order factorization methods. *BIT*, 18:142–156, 1978.

[189] I. Gustafsson and G. Lindskog. A preconditioning technique based on element matrix factorizations. *Comput. Methods Appl. Mech. Eng.*, 55:201–220, 1986.

[190] I. Gustafsson and G. Lindskog. Completely parallelizable preconditioning methods. *Numerical Linear Algebra with Applications*, 2:447–465, 1995.

[191] M. H. Gutknecht. Variants of BICGSTAB for matrices with complex spectrum. *SIAM J. Sci. Comput.*, 14:1020–1033, 1993.

[192] L. A. Hageman and D. M. Young. *Applied Iterative Methods.* Academic Press, New York, 1981.

[193] K. Hayami and N. Harada. The scaled conjugate gradient method on vector processors. In S. P. Kartashev and I. S. Kartashev, editors, *Supercomputing Systems, Proc. of the First International Conference, St.Petersburg*, pages 213–221, 1985.

[194] D. Heller. Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems. *SIAM J. Numer. Anal.*, 13:484–496, 1976.

[195] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[196] M. Heroux, P. Vu, and C. Yang. A parallel preconditioned conjugate gradient package for solving sparse linear systems on a CRAY Y-MP. *Applied Numerical Mathematics*, 8:93–115, 1991.

[197] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Natl. Bur. Stand.*, 49:409–436, 1954.

[198] R. Hockney and C. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms.* Adam Hilger, Ltd., Bristol, United Kingdom, 1981.

[199] R. Hockney and C. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms.* Adam Hilger, Ltd., Bristol, United Kingdom, 1981.

[200] R. Hockney and C. Jessup. *Parallel Computers 2: Architecture, Programming and Algorithms.* Adam Hilger/IOP Publishing, Bristol, 1988.

[201] R. W. Hockney and I. J. Curington. $f_{1/2}$: A parameter to characterize memory and communication bottlenecks. *Parallel Computing*, 10:277–286, 1989.

[202] Roger Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, 17:1111–1130, 1991.

[203] Roger Hockney. The communication challenge for MPP. *Parallel Computing*, 20:389–398, 1994.

[204] P. Hood. Frontal solution program for unsymmetric matrices. *Int J. Numerical Methods in Engineering*, 10:379–400, 1976.

[205] HSL. *Harwell Subroutine Library. A Catalogue of Subroutines (Release 12).* AEA Technology, Harwell Laboratory, Oxfordshire, England, 1996. For information concerning HSL contact: Dr Scott Roberts, AEA Technology, 552 Harwell, Didcot, Oxon OX11 0RA, England (tel: +44-1235-434988, fax: +44-1235-434136, email: Scott.Roberts@aeat.co.uk).

[206] Y. Huang and H. A. van der Vorst. Some observations on the convergence behaviour of GMRES. Technical Report 89-09, Delft University of Technology, Faculty of Tech. Math., 1989.

[207] T.J.R. Hughes, I. Levit, and J. Winget. An element-by-element solution algorithm for problems of structural and solid mechanics. *J. Comp. Methods in Appl. Mech. Eng.*, 36:241–254, 1983.

[208] K. Hwang and F. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, NY, 1984.

[209] B. M. Irons. A frontal solution program for finite-element analysis. *Int J. Numerical Methods in Engineering*, 2:5–32, 1970.

[210] K. C. Jea and D. M. Young. Generalized conjugate-gradient acceleration of nonsymmetrizable iterative methods. *Lin. Alg. & Appl.*, 34:159–194, 1980.

[211] E. Jessup and D. Sorensen. A parallel algorithm for computing the singular value decomposition of a matrix. Mathematics and Computer Science Division Report MCS-TM-102, Argonne National Laboratory, Argonne, IL, December 1987.

[212] O. G. Johnson, C. A. Micheli, and G. Paul. Polynomial preconditioning for conjugate gradient calculations. *SIAM J. Numer. Anal.*, 20:363–376, 1983.

[213] M.T. Jones and P.E. Plassmann. The efficient parallel iterative solution of large sparse linear systems. In A. George, J.R. Gilbert, and J.W.H. Liu, editors, *Graph Theory and Sparse Matrix Computations*, IMA Vol 56. Springer Verlag, Berlin, 1994.

[214] T. L. Jordan. A guide to parallel computation and some CRAY-1 experiences. Technical Report LANL Report LA-UR-81-247, Los Alamos National Laboratory, Los Alamos, NM, 1981.

[215] A. Karp and R. Babb. A comparison of 12 parallel Fortran dialects. *IEEE Software*, pages 52–67, 1988.

[216] L. Kaufman. Usage of the sparse matrix programs in the PORT library. Technical Report Report 105, Bell Laboratories, Murray Hill, New Jersey, 1982.

[217] R. Kettler. *Linear multigrid methods in numerical reservoir simulation.* PhD thesis, Delft University of Technology, Delft, 1987.

[218] D. R. Kincaid, T. C. Oppe, J. R. Respess, and D. M. Young. ITPACKV 2C User's Guide. Report CNA-191, Center for Numerical Analysis, Univ. of Texas, Austin, TX, 1984.

[219] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy K. Steel Jr., and Mary E. Zosel. *The High Performance Fortran Handbook.* MIT Press, Cambridge, MA, 1994.

[220] P. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill, New York, 1981.

[221] L. Yu. Kolotilina and A. Yu. Yeremin. Factorized sparse approximate inverse preconditionings. *SIAM J. Matrix Analysis and Applications*, 14:45–58, 1993.

[222] J. Koster and R. H. Bisseling. Parallel sparse LU decomposition on a distributed-memory multiprocessor, 1994. Submitted to *SIAM J. Scientific Computing.*

[223] J. J. Lambiotte and R. G. Voigt. The solution of tridiagonal linear systems on the CDC-STAR-100 computer. Technical report, ICASE-NASA Langley Research Center, Hampton, VA, 1974.

[224] C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Natl. Bur. Stand*, 49:33–53, 1952.

[225] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5:308–329, 1979.

[226] J. M. Levesque and J. W. Williamson. *A Guidebook to Fortran on Supercomputers.* Academic Press, 1989.

[227] J. W. H. Liu. The minimum degree ordering with constraints. *SIAM J. Scientific and Statistical Computing*, 10:1136–1145, 1988.

[228] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Analysis and Applications*, 11:134–172, 1990.

[229] R. Lucas, T. Blank, and J. Tiemann. A parallel method for large sparse systems of equations. *IEEE Trans. on Computer-Aided Design*, CAD-6:981–991, 1987.

[230] N. K. Madsen, G. H. Rodrigue, and J. I. Karush. Matrix multiplication by diagonals on a vector/parallel processor. *Inform. Process. Lett.*, 52:41–45, 1976.

[231] T. A. Manteuffel. The Tchebychev iteration for nonsymmetric linear systems. *Numer. Math.*, 28:307–327, 1977.

[232] T.A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Math. Comp.*, 31:473–497, 1980.

[233] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3:255–269, Apr. 1957.

[234] K. K. Mathur and S. L. Johnsson. The finite element method on a data parallel computing system. *Int. J. of High-Speed Computing*, 1(1):29–44, May 1989.

[235] U. Meier. A parallel partition method for solving banded systems of linear equations. *Parallel Computing*, 2:33–43, 1985.

[236] U. Meier and A. Sameh. The behavior of conjugate gradient algorithms on a multivector processor with a hierarchical memory. Report 758, University of Illinois, Urbana, 1988.

[237] J. A. Meijerink. Iterative methods for the solution of linear equations based on incomplete factorisations of the matrix. Technical Report 643, Shell, KSEPL, Rijswijk, 1983.

[238] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp*, 31:148–162, 1977.

[239] J. A. Meijerink and H. A. van der Vorst. Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems. *J. Comp. Physics*, 44:134–155, 1981.

[240] G. Meurant. The block preconditioned conjugate gradient method on vector computers. *BIT*, 24:623–633, 1984.

[241] G. Meurant. Numerical experiments for the preconditioned conjugate gradient method on the CRAY X-MP/2. Report LBL-18023, University of California, Berkeley, 1984.

[242] G. Meurant. The conjugate gradient method on vector and parallel supercomputers. Report CTAC-89, University of Brisbane, 1989.

[243] C. Moler. A closer look at Amdahl's law. Technical Report TN-02-687, Intel, 1987.

[244] R.B. Morgan. Computing interior eigenvalues of large matrices. *Lin. Alg. and its Appl.*, 154–156:289–309, 1991.

[245] R.B. Morgan and D.S. Scott. Preconditioning the Lanczos algorithm for sparse symmetric eigenvalue problems. *SIAM J. Sci. Comput.*, 14:585–593, 1993.

[246] N. Munksgaard. Solving sparse symmetric sets of linear equations by preconditioned conjugate gradient method. *ACM Transactions on Mathematical Software*, 6:206–219, 1980.

[247] N. M. Nachtigal, S. C. Reddy, and L. N. Trefethen. How fast are nonsymmetric matrix iterations? *SIAM J. Matrix Anal. Appl.*, 13:778–795, 1992.

[248] N. M. Nachtigal, L. Reichel, and L. N. Trefethen. A hybrid GMRES algorithm for nonsymmetric matrix iterations. Technical Report 90-7, MIT, Cambridge, MA, 1990.

[249] H. Nelis. *Sparse Approximations of Inverse Matrices.* PhD thesis, Delft University of Technology, Delft, 1989.

[250] Y. Notay. DRIC: a dynamic version of the RIC method. *Numerical Linear Algebra with Applications*, 1:511–532, 1994.

[251] J. Ortega and C. Romine. The ijk forms of factorization II. Parallel systems. *Parallel Computing*, 7(2):149–162, 1988.

[252] O. Østerby and Z. Zlatev. *Direct methods for sparse matrices.* Number 157 in Lecture Notes in Computer Science. Springer Verlag, Berlin, Heidelberg, New York, 1983.

[253] S. Otto, J. Dongarra, S. Hess-Lederman, M. Snir, and D. Walker. *Message Passing Interface: The Complete Reference.* The MIT Press, 1995.

[254] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12:617–629, 1975.

[255] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Soft.*, 8:43–71, 1982.

[256] Parasoft Corporation, Monrovia, CA. *Express User's Guide*, version 3.2.5 edition, 1992. Parasoft can be reached, electronically, at `parasoft@Parasoft.COM`.

[257] B. N. Parlett, D. R. Taylor, and Z. A. Liu. A look-ahead Lanczos algorithm for unsymmetric matrices. *Math. Comp.*, 44:105–124, 1985.

[258] S. V. Parter. The use of linear graphs in gaussian elimination. *SIAM Review*, 3:119–130, 1961.

[259] C. Pommerell. *Solution of large unsymmetric systems of linear equations.* PhD thesis, Swiss Federal Institute of Technology, Zürich, 1992.

[260] C. Pommerell and W. Fichtner. PILS: An iterative linear solver package for ill-conditioned systems. In *Supercomputing '91*, pages 588–599, Los Alamitos, CA., 1991. IEEE Computer Society.

[261] G. Radicati and Y. Robert. Vector and parallel CG-like algorithms for sparse non-symmetric systems. Report 681-M, IMAG/TIM3, Grenoble, France, 1987.

[262] G. Radicati and M. Vitaletti. Sparse matrix-vector product and storage representations on the IBM 3090 with Vector Facility. Report G513-4098, IBM-ECSEC, Rome, 1986.

[263] G. Radicati di Brozolo and Y. Robert. Parallel conjugate gradient-like algorithms for solving sparse non-symmetric systems on a vector multiprocessor. *Parallel Computing*, 11:223–239, 1989.

[264] J. K. Reid. Sparse matrices. In D. A. H. Jacobs, editor, *The State of the Art in Numerical Analysis*, pages 85–146, New York, 1977. Academic Press.

[265] J. R. Rice and R. F. Boisvert. *Solving Elliptic Problems Using ELLPACK*. Springer-Verlag, Heidelberg, 1985.

[266] E. Rothberg. Efficient sparse Cholesky factorization on distributed-memory multiprocessors. In J. G. Lewis, editor, *Proceedings 5th SIAM Conference on Linear Algebra*, page 141, Philadelphia, 1994. SIAM Press.

[267] Edward Rothberg. Exploring the tradeoff between imbalance and separator size in nested dissection ordering. Technical Report Unnumbered, Silicon Graphics Inc, 1996.

[268] Y. Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM J. Sci. Statist. Comput.*, 6:865–881, 1985.

[269] Y. Saad. Krylov subspace methods on supercomputers. Report September 19, RIACS, Moffett Field, 1988.

[270] Y. Saad. *Numerical methods for large eigenvalue problems*. Manchester University Press, Manchester, UK, 1992.

[271] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14:461–469, 1993.

[272] Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.

[273] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations. Version 2. Technical report, Computer Science Department, University of Minnesota, 1994.

[274] Y. Saad. *Iterative methods for sparse linear systems*. PWS Publishing Company, Boston, 1996.

[275] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[276] J. J. F. M. Schlichting and H. A. van der Vorst. Solving bidiagonal systems of linear equations on the CDC CYBER 205. Report NM-R8725, CWI, Amsterdam, 1987.

[277] J. J. F. M. Schlichting and H. A. van der Vorst. Solving 3D block bidiagonal linear systems on vector computers. *Journal of Comp. and Appl. Math*, 27:323–330, 1989.

[278] R. Schreiber and C. Van Loan. A storage efficient WY representation for products of Householder transformations. *SIAM J. Sci Stat Comp.*, 10(1):53–57, 1989.

[279] M. K. Seager. Parallelizing conjugate gradient for the CRAY X-MP. *Parallel Computing*, 3:35–47, 1986.

[280] A. H. Sherman. Algorithm 533. NSPIV, A Fortran subroutine for sparse Gaussian elimination with partial pivoting. *ACM Trans. Math. Softw.*, 4:391–398, 1978.

[281] H. D. Simon, P. Vu, and C. Yang. Performance of a supernodal general sparse solver on the CRAY Y-MP: 1.68 Gflops with autotasking. Report SCA-TR-117, Boeing Computer Services, Seattle, 1989.

[282] A. Skjellum and A. Leung. Zipcode: a Portable Multicomputer Communication Library atop the Reactive Kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–76. IEEE Press, 1990.

[283] G. L. G. Sleijpen and H.A. Van der Vorst. Maintaining convergence properties of BICGSTAB methods in finite precision arithmetic. *Numerical Algorithms*, 10:203–223, 1995.

[284] G. L. G. Sleijpen and H.A. Van der Vorst. A Jacobi-Davidson iteration method for linear eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 17:401–425, 1996.

[285] G. L. G. Sleijpen, H.A. Van der Vorst, and D. R. Fokkema. Bi-CGSTAB(ℓ) and other hybrid Bi-CG methods. *Numerical Algorithms*, 7:75–109, 1994.

[286] G. L. G. Sleijpen and D. R. Fokkema. BICGSTAB(ℓ) for linear equations involving unsymmetric matrices with complex spectrum. *ETNA*, 1:11–32, 1993.

[287] G. L. G. Sleijpen, H.A. Van der Vorst, and J. Modersitzki. The main effects of rounding errors in Krylov solvers for symmetric linear systems. Technical Report 1006, University Utrecht, Department of Mathematics, 1997.

[288] G.L.G. Sleijpen and H.A. Van der Vorst. Reliable updated residuals in hybrid Bi-CG methods. *Computing*, 56:141–163, 1996.

[289] S. W. Sloan and M. F. Randolph. Automatic element reordering for finite-element analysis with frontal schemes. *Int J. Numerical Methods in Engineering*, 19:1153–1181, 1983.

[290] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. Klema, and C. Moler. *Matrix Eigensystem Routines—EISPACK Guide., 2nd ed.*, volume 6. Springer-Verlag, New York, 1976.

[291] P. Sonneveld. CGS: a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 10:36–52, 1989.

[292] D. Sorensen and C. Van Loan. Block factorizations for symmetric indefinite matrices, Cornell technical report. In preparation.

[293] D. C. Sorensen. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM J. Mat. Anal. Appl.*, 13(1):357–385, 1992.

[294] G. W. Stewart. *Introduction to Matrix Computations.* Academic Press, New York, 1973.

[295] H. Stone. *High Performance Computer Architecture.* Addison-Wesley, New York, 1987.

[296] H. L. Stone. Iterative solution of implicit approximations of multidimensional partial differential equations. *SIAM J. Numer. Anal.*, 5:530–558, 1968.

[297] H. S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *JACM*, 20:27–38, 1973.

[298] K.H. Tan. *Local coupling in domain decomposition.* PhD thesis, Utrecht University, Utrecht, the Netherlands, 1995.

[299] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. of the IEEE*, 55:1801–1809, 1967.

[300] A. van der Sluis. Condition numbers and equilibration of matrices. *Numer. Math.*, 14 1:14–23, 1969.

[301] A. van der Sluis and H. A. van der Vorst. The rate of convergence of conjugate gradients. *Numer. Math.*, 48:543–560, 1986.

[302] A. van der Sluis and H. A. van der Vorst. Numerical solution of large sparse linear algebraic systems arising from tomographic problems. In G. Nolet, editor, *Seismic Tomography*, Dordrecht, 1987. Reidel.

[303] A. F. van der Stappen, R. H. Bisseling, and J. G. G. van de Vorst. Parallel sparse LU decomposition on a mesh network of transputers. *SIAM J. Matrix Analysis and Applications*, 14:853–879, 1993.

[304] H. A. Van der Vorst. Iterative solution methods for certain sparse linear systems with a non-symmetric matrix arising from PDE-problems. *J. Comp. Phys.*, 44:1–19, 1981.

[305] H. A. van der Vorst. A vectorizable variant of some ICCG methods. *SIAM J. Sci. Statist. Comput.*, 3:350–356, 1982.

[306] H. A. van der Vorst. The performance of Fortran implementations for preconditioned conjugate gradients on vector computers. *Parallel Computing*, 3:49–58, 1986.

[307] H. A. van der Vorst. Large tridiagonal and block tridiagonal linear systems on vector and parallel computers. *Parallel Computing*, 5:45–54, 1987.

[308] H. A. van der Vorst. High performance preconditioning. *SIAM J. Sci. Statist. Comput.*, 10:1174–1185, 1989.

[309] H. A. van der Vorst. ICCG and related methods for 3D problems on vector computers. *Computer Physics Communications*, 53:223–235, 1989.

[310] H. A. Van der Vorst. The convergence behaviour of preconditioned CG and CG-S in the presence of rounding errors. In O. Axelsson and L. Yu. Kolotilina, editors, *Preconditioned Conjugate Gradient Methods*, Berlin, 1990. Nijmegen 1989, Springer Verlag. Lecture Notes in Mathematics 1457.

[311] H. A. Van der Vorst. Experiences with parallel vector computers for sparse linear systems. *Supercomputer*, 37:28–35, 1990.

[312] H. A. Van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 13:631–644, 1992.

[313] H. A. van der Vorst and K. Dekker. Vectorization of linear recurrence relations. *SIAM J. Sci. Statist. Comput.*, 10:27–35, 1989.

[314] H. A. Van der Vorst and C. Vuik. The superlinear convergence behaviour of GMRES. *J. of Comp. Appl. Math.*, 48:327–341, 1993.

[315] H. A. Van der Vorst and C. Vuik. GMRESR: A family of nested GMRES methods. *Num. Lin. Alg. with Appl.*, 1:369–386, 1994.

[316] H.A. Van der Vorst. *Preconditioning by Incomplete Decompositions*. PhD thesis, Utrecht University, Utrecht, The Netherlands, 1982.

[317] M.B. van Gijzen. *Iterative solution methods for linear equations in finite element computations*. PhD thesis, Delft University of Technology, Delft, the Netherlands, 1994.

[318] R. S. Varga. *Matrix Iterative Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1962.

[319] P. K. W. Vinsome. ORTOMIN: An iterative method for solving sparse sets of simultaneous linear equations. In *Proc. Fourth Symposium on Reservoir, Simulation, Society of Petroleum Engineers of AIME*, pages 149–159, 1976.

[320] C. Vuik and H.A. van der Vorst. A comparison of some GMRES-like methods. *Lin. Alg. and its Appl.*, 160:131–162, 1992.

[321] H. F. Walker. Implementation of the GMRES method using Householder transformations. *SIAM J. Sci. Statist. Comput.*, 9:152–163, 1988.

[322] H. H. Wang. A parallel method for tridiagonal equations. *ACM Trans. Math. Softw.*, pages 170–183, 1981.

[323] W. Ware. The ultimate computer. *IEEE Spectrum*, pages 89–91, March 1973.

[324] T. Washio and K. Hayami. Parallel block preconditioning based on SSOR and MILU. *Numer. Lin. Alg. with Applic.*, 1:533–553, 1994.

[325] R. Weiss. *Parameter-Free Iterative Linear Solvers.* Akademie Verlag, Berlin, 1996.

[326] O. Widlund. A Lanczos method for a class of nonsymmetric systems of linear equations. *SIAM J. Numer. Anal.*, 15:801–812, 1978.

[327] D. T. Winter. Efficient use of memory and input/output. In J. J. te Riele, T. J. Dekker, and H. A. van der Vorst, editors, *Algorithms and Applications on Vector and Parallel Computers*, Amsterdam, 1987. North-Holland.

[328] D.P. Young, R.G. Melvin, F.T. Johnson, J.E. Bussoletti, L.B. Wigton, and S.S. Samanth. Application of sparse matrix solvers as effective preconditioners. *SIAM J. Sci. Stat. Comp.*, 10(6):1186–1199, 1989.

[329] L. Zhou and H. F. Walker. Residual smoothing techniques for iterative methods. *SIAM J. Sci. Stat. Comp.*, 15:297–312, 1994.

[330] Z. Zlatev. *Computational methods for general sparse matrices.* Kluwer Acad. Pub., Dordrecht, Boston, London, 1991.

[331] Z. Zlatev, J. Waśniewski, and K. Schaumburg. *Y12M - Solution of Large and Sparse Systems of Linear Algebraic Equations*, volume 121 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1981.

Index