Parallel Computing 8 (1988) 41–58
North-Holland

Programming methodology and performance issues for advanced computer architectures *

Jack J. DONGARRA, Danny C. SORENSEN, Kathryn CONNOLLY and Jim PATTERSON

Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.

Abstract. This paper will describe some recent attempts to construct transportable numerical software for high-performance computers. Restructuring algorithms in terms of simple linear algebra modules is reviewed. This technique has proved very successful in obtaining a high level of transportability without severe loss of performance on a wide variety of both vector and parallel computers. The use of modules to encapsulate parallelism and reduce the ratio of data movement to floating-point operations has been demonstrably effective for regular problems such as those found in dense linear algebra. In other situations it may be necessary to express explicitly parallel algorithms. We also present a programming methodology that is useful for constructing new parallel algorithms which require sophisticated synchronization at a large grain level. We describe the SCHEDULE package which provides an environment for developing and analyzing explicitly parallel programs in FORTRAN which are portable. This package now includes a preprocessor to achieve complete portability of user level code and also a graphics post processor for performance analysis and debugging. We discuss details of porting both the SCHEDULE package and user code. Examples from linear algebra, and partial differential equations are used to illustrate the utility of this approach.

Keywords. Parallel programming, portable parallel algorithms, the package SCHEDULE.

1. Introduction

In this paper, some techniques for programming computers with advanced architectures will be surveyed. These computers rely upon vectorization, pipelined functional units, and multiple computational elements to provide high performance. However, it may be difficult to realize the benefits of these performance features unless certain programming techniques are adopted. Often, this goes beyond trivial reorganization of existing programs and requires significant restructuring of basic algorithms. Moreover, parallel computers (those with multiple computational elements) have motivated the development of entirely new algorithms for standard problems.

The advent of these high-performance computers has opened vistas in terms of what are now thought of as computationally tractable problems. However, they have also brought about a crisis in software development. Over the years there has emerged a recognition of the importance of developing portable high-quality computer software for basic computations. In the past, issues have largely been those of robustness, accuracy, and reliability as well as ease of maintenance and distribution. Now a variety of different architectures are being offered commercially. The desire to exploit these new powerful computers has made it tempting to

* Work supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

disregard the notion of portability in favor of performance. The techniques discussed below offer some approaches to avoiding this dangerous path. Our experience with these techniques offers some evidence that it is indeed possible to develop portable codes which obtain a reasonable fraction of the potential performance of these modern computers. Moreover, it is possible to use existing mathematical software written in FORTRAN within a parallel computing environment using the techniques we describe here.

Two basis ideas are discussed. One is the notion of recasting algorithms in terms of high-level modules. An important set of such modules, Basic Linear Algebra Subprograms, BLAS [24,14,13] has proven to be very successful in achieving performance without sacrificing portability in linear algebra libraries. Coding in terms of these modules relieves an algorithm developer from the tedium of detailed machine-dependent coding to gain performance. Such detail is encapsulated within the module and done once for each machine. The other notion is to use a standardized interface to exploit the parallel capabilities of machines with multiple processors through explicit parallel programming. A large-grain control flow model of computation is presented which has a natural graphical interpretation that is useful in designing and implementing explicit parallel algorithms. This graphical interpretation also lends itself well to a postprocessing performance analysis of a given code.

Throughout this paper, we only consider shared memory multi-processors which may or may not have vector capabilities at the processor level. Section 2 deals primarily with the use of modules to gain performance and transportability of software on a wide variety of computers. Section 3 describes the SCHEDULE package.

2. Modularization for portable performance

Encapsulation of basic matrix and vector operations into a set of high-level modules has proven to be an important idea which has been very successful in providing clarity, portability, and ease of maintenance in mathematical software. The most successful example of these is a set of specifications known as the BLAS, proposed originally in 1973 by Hanson et al. [24]. These modules describe vector operations such as adding a scalar multiple of one vector to another. More recently, there has been a need to extend these ideas to Level 2 BLAS [14] involving matrix-vector operations and to Level 3 BLAS [13] involving matrix-matrix operations. The use of these modules has been restricted primarily to developers of high-quality linear algebra software such as LINPACK [12]. It is now clear that these modules should become part of the repertoire of all users of high-performance computers and should serve as a model for the development of special purpose modules designed to perform basic operations of given application areas. An important aspect of the BLAS is standardization. Efforts have been made to arrive at a consensus of what modules belong in a given set and what their calling sequence should be [24,14,13]. This is essential to get them adopted within the user community and to motivate manufacturers to devote the resources necessary to develop highly tuned modules for specific machines.

The reason widespread use of such modules should be adopted is that they offer both high performance and portability across a wide variety of computer architectures. Moreover, coding in terms of such modules will relieve an algorithm developer from the detail and tedium involved in coding for performance on a specific computer. This has become a great concern as widely different computer architectures emerge. To gain performance, one must be aware of how to use vector processors effectively, manage data movement, utilize memory hierarchies, etc. In some cases this performance cannot be achieved in a high-level language and assembly language must be employed. Such detail should not be part of a high-level algorithm, but must be taken into account at some level if performance is to be realized. Coding in terms of modules which have standardized interfaces to a programming language such as FORTRAN allow a user to tap the performance of any computer which has had the modules tuned for it.

The effectiveness of this approach is best illustrated from experience in linear algebra. LINPACK is an example where this restructuring can lead to increased performance. This is a collection of linear algebra subroutines used for solving linear algebraic systems of various types. This package was originally structured and coded in terms of the Level 1 BLAS. However, very poor performance was observed on vector computers [16]. In an effort to capture the expected performance, the LINPACK routines were recast in terms of what are now called Level 2 BLAS involving matrix-vector operations. The motivation was to reduce the ratio of data movement to floating-point operations to increase performance. This approach has been very successful on a wide variety of vector and parallel computers. The performance results are well documented in [11,18] and the details of the restructuring are shown in [18].

As computer architectures become more sophisticated in their organization, we are required to supply even a higher level of granularity in our algorithms to take full advantage of the potential performance. We are already feeling the effects of designs involving memory hierarchies such as cache and large sets of vector registers. Experience with these machines has motivated the study of higher-level modules. The next level of modularity we naturally focus on is at the matrix-matrix level. Again, the idea is to reduce the ratio of data accesses to floating-point operations. Matrix-matrix operations such as the product of two matrices are obvious candidates. The advantage here is obvious, only $O(n^2)$ data is referenced to perform $O(n^3)$ floating-point operations when two matrices of order n are multiplied. Again the main factorization routines of LINPACK have been recast in terms of these Level 3 BLAS operations with reasonable success. The details of how this restructuring might been done has been described in [18].

Table 1 illustrates the relative effectiveness of these modules in the context of computing the LU-factorization of a matrix. In the table we compare the performance of LU-factorization when recast in terms of the various levels of BLAS just discussed. These computations were done on an Alliant FX/8 computer which has eight parallel-vector processors sharing up to 64 Mbytes of memory through a 128 kbyte cache. In the results shown in Table 1, the Level 2 and 3 BLAS have been coded is assembly language to assure full use of the vector registers. The table shows that the Level 3 BLAS modules can be very effective. Moreover, while this technique appears to be tuned to a particular situation, it is equally effective in a non-cache situation as long as there are a significant number of vector registers available. Finally, blocks of cache may be used in place of vector registers when (as with the Alliant) the memory access from cache matches the memory access from registers.

A number of researchers have considered matrix-matrix formulation of the basic factorization routines of linear algebra. A very nice treatment of a block version of the Householder QR-factorization may be found in [5]. Extensive use of this technique has been made in the development of a library of linear algebra at the Center for Supercomputer Research and Development at University of Illinois at Urbana [4]. A proposal for a formal specification of the Level 3 BLAS may be found in [13].

Table 1
LU-decomposition with partial pivoting Alliant 8 CE's

Implementation	Mflops order					
	100	200	300	400	500	600
Level 1 BLAS	2.8	4.7	5.4	5.7	5.7	5.8
Level 2 BLAS	8.2	9.8	11.2	10.7	11.6	11.3
Level 3 BLAS	6.6	11.8	15.3	17.2	18.7	19.8

3. Explicit parallel programming

Encapsulation of low-level detail in modules such as the BLAS has indeed been very successful but certainly has limited applicability. Some parallel algorithms will require explicit parallel programming in order to be implemented. Adequate tools have been provided by several vendors to support loop-based parallelism. This only provides a *fork–join* mechanism. We find the tools available for explicit parallel programming of algorithms which require multilevel parallelism and dynamic allocation of user processes to be less than adequate.

Many new parallel computers are now emerging as commercial products [15]. Exploitation of the parallel capabilities requires either extensions to an existing language such as FORTRAN or development of an entirely new language. A number of activities [29,32] are under way to develop new languages that promise to provide the ability to exploit parallelism without the considerable effort that may be required in using an inherently serial language that has been extended for parallelism. We applaud such activities and expect they will offer a true solution to the software dilemma in the future. However, in the short term we feel there is a need to confront some of the software issues, with particular emphasis placed on transportability and use of existing software.

Our interests lie mainly with mathematical software typically associated with scientific computations. Therefore, we concentrate here on using the FORTRAN language. Each vendor of a parallel machine designed primarily for numerical calculations has provided a different set of parallel extensions to FORTRAN. These extensions have taken many forms already and are usually dictated by the underlying hardware and by the capabilities that the vendor wishes to supply the user. This has led to widely different extensions ranging from the ability to synchronize on every assignment of a variable with a full/empty [23] to attempts at automatically detecting loop-based parallelism with a preprocessing compiler aided by user directives [9]. The act of getting a parallel process executing on a physical processor ranges from a simple 'create' statement [23] which imposes the overhead of a subroutine call, to 'TSKSTART' [1] which imposes an overhead on the order of 10^6 machine cycles, to no formal mechanism whatsoever [9]. These different approaches reflect characteristics of underlying hardware and operating systems and to a large extent are dictated by the vendors view of which aspects of parallelism are marketable. It is too early to impose a standard on these vendors, yet it is disconcerting that there is no agreement among any of them on which extensions should be included. There is not even an agreed naming convention for extensions that have identical functionality. Program developers interested a producing implementations of parallel algorithms that will run on a different parallel machines are therefore faced with an extremely difficult task. The process of developing portable parallel packages is complicated by additional factors that lie beyond each computer manufacturer supplying different mechanism for parallel processing. A given implementation may require several different communicating parallel processes, perhaps with different levels of granularity. An efficient implementation may require the ability to dynamically start processes, perhaps many more than the number of physical processors in the system. This feature is either lacking or prohibitively expensive on most commercially available parallel computers. Instead, many of the manufacturers have limited themselves to providing one-level loop-based parallelism.

This section describes an environment for the transportable implementation of parallel algorithms in a FORTRAN setting. By this we mean that a user's code is virtually identical for each machine. The main tool in this environment is a package called SCHEDULE which has been designed to aid a programmer familiar with a FORTRAN programming environment to implement a parallel algorithm in a manner that will lend itself to transporting the resulting program across a wide variety of parallel machines. The package is designed to allow existing FORTRAN subroutines to be called through SCHEDULE, without modification, thereby permitting users access to a wide body of existing library software in a parallel setting. Machine intrinsics are invoked within the SCHEDULE package, and considerable effort may be required on our part to move SCHEDULE from one machine to another. On the other hand, the user of SCHEDULE is relieved of the burden of modifying each code he desires to transport from one machine to another.

A number of efforts are underway to provide parallel programming tools. Our work has primarily been influenced by the work of Babb [2], Browne [6], and Lusk and Overbeek [25]. We present here our approach, which aids in the programming of explicitly parallel algorithms in FORTRAN and which allows one to make use of existing FORTRAN libraries in the parallel setting. The approach taken here should be regarded as minimalist: it has a very limited scope. There are two reasons for this. First, the goal of portability of user code will be less difficult to achieve. Second, the real hope for a solution to the software problems associated with parallel programming lies with new programming languages or perhaps with the 'right' extension to FORTRAN. Our approach is expected to have a limited lifetime. Its purpose is to allow us to exploit existing hardware immediately. Nevertheless, experience gained through this effort is quite valuable. It is not unreasonable to expect this approach to mature into a very effective programming tool.

3.1. Parallel programming using SCHEDULE

The underlying idea in SCHEDULE is that parallel computations may often be represented graphically and that this is a useful way to think about and to construct parallel programs. A parallel program is derived by breaking a computation up into units of computation and execution dependencies between them. We use the term *execution dependency* rather than data dependency because these dependencies are assertions made by the user about the order in which computations may occur. This order must respect data dependencies but may be used more generally to achieve load balancing and other characteristics during execution. We find it more accurate, therefore, to call this a *control flow* graph rather than a large-grain data flow graph.

In the FORTRAN language, it is natural to associate a unit of computation (or a process) with a subroutine name together with the data upon which this subroutine will operate. The basic idea is that FORTRAN programs are naturally broken into subroutines that identify self-contained units of computations which operate on shared data structures. Naturally, this notion encompasses the use of existing library subroutines in a parallel setting. One would like to be able to call upon such routines in the usual way without modification, and without having to write an envelope around the library subroutine call in order to conform to some unusual data-passing conventions imposed by a given parallel programming environment.

To write a SCHEDULE program, a user must construct a large-grain dependency graph representing the units of computation and the execution dependencies between them in order to specify a parallel computation. Each of these units of computation will represent a subroutine call that is to be made when execution dependencies have been satisfied. The user must take the responsibility of ensuring that the dependencies represented by the graph are valid. We shall try to explain this concept through a generic example; in the following sections we shall describe the underlying concepts, the SCHEDULE mechanism and give some examples.

The first step in writing a SCHEDULE program is to express the algorithm in terms of processes and execution dependencies among the processes. A convenient way to represent this is through a computational graph. For example, the graph in Fig. 1 denotes five subroutines 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

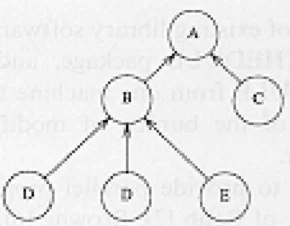

Fig. 1. Execution dependency graph.

E have completed, *B* may execute. When *B* and *C* have completed execution, *A* may start and the entire computation is finished when *A* has completed. To use SCHEDULE, one is required to specify the subroutine names and the arguments associated with each of the six units of computation, along with a representation of this dependency graph.

For each node in the graph, SCHEDULE requires two subroutine calls. One contains information about the user's routine to be called, such as the name of the routine, calling sequence parameters, and a simple tag to identify the process. The second subroutine call defines the dependency in the graph to nodes above and below the one being specified, and specifies the tag to identify the process. In this example, after an initial call to set up the environment for SCHEDULE, six pairs of calls would be made to define the relationships and data in the computational graph.

Within the context of a SCHEDULE program, execution dependencies are assertions made by the user about the correct order of computation. Responsibility for the correctness of these assertions rests with the user. SCHEDULE provides a mechanism to record the units of computation, express the dependencies between them, and then execute the computation described by the SCHEDULE program. It is very important to clearly define and understand the shared data of a problem and how it is to be partitioned for parallel processing.

3.2. *Partitioning a problem*

The first thing that must be understood is the distinction between local and global variables. Global variables are shared amongst all the processes which reference them. Local variables are defined within a subroutine and a local version of these variables is obtained each time a unit of computation based upon this subroutine is executed. As a matter of programming style, it is suggested that all shared data be grouped in named COMMON even if some of the shared parameters are passed in the call to SCHED.

Before discussing how data is partitioned it is useful to have an idea of what a SCHEDULE program will look like. The structure of a SCHEDULE program generally takes the following form:

```
Main Program:

C$SCHED START_SHARED
      common /prbdef/<array declarations >
C$SCHED END_SHARED
      external paralg
C$SCHED EXECUTE

      initialize variables
                .
                .
                .
            call sched(nprocs,paralg,<parms>)
      .
      .
      .
      output result

      stop
      end
```

Code That Records Static Execution Dependency Graph:

```
      subroutine paralg(<parms>)
      declare global variables

      declare local variables
      external subname

      do 100 j = 1,nunits
                 .
                 .
                 .
         jobtag = the identifier of this unit of computation
         icango = number of nodes jobtag depends on
         nchcks = number of nodes which depend on jobtag
         list   = list of identifiers of these nchcks dependents

         call dep(jobtag,icango,nchcks,list)
         call putq(jobtag,subname,<parms>)

  100 continue
      return
      end
```

In this generic example the *main program* declares shared data and initializes it and then makes the initial call to *sched*. This call acquires *nprocs* physical processors devoted to the execution of the computation defined by subroutine *paralg*. There can be several calls to *sched* in the main program. Each call involves the overhead of acquiring *nprocs* physical processors on the given system. Care must be exercised on some systems because the expense of acquiring a number of physical processors may be substantial. The statements beginning with CSSHED are preprocessor directives indicating the start and end of shared common and also indicate the first executable statement to the preprocessor. This aids in automatically modifying the shared common blocks on those machines that require special declaration of shared memory.

Execution of the subroutine *paralg* records the static dependency graph using calls to *dep* and *putq* for each node in the graph. In this generic example *nunits* units of computation were defined in the 100 loop of the subroutine *paralg*. Here, all of the units of computation involve the subroutine *subname* operating on different data. Generally, there may be several different subroutines associated with various units of computation being placed on the queue. They may represent very different computations and may have different calling sequences.

It is also possible to have dynamically spawned processes at run time. We use the SCHEDULE subroutine *nxtag* and *spawn* to accomplish this as shown in the following example:

Unit of Computation with Dynamic Spawning

```
      subroutine subname(<parms>)
      declare global variables
```

```
         declare local variables
         external sublow

         do 200 j = 1,nkids
            .
            .
            .
            call nxtag(mytag,jdummy)
            call spawn(mytag,jdummy,sublow,<parms>)
            .
            .
            .
     200 continue
         return
         end
```

This example shows the three levels of parallelism typically available in a SCHEDULE program. However, it should be emphasized that many layers of parallelism can be expressed using SCHEDULE and fairly intricate inter-process dependencies can be accommodated. In this generic example, the subroutine *paralg* defines the static control flow graph which is to be obeyed throughout the course of the computation, while the subroutine *subname* exploits the lowest level of parallelism utilized in this program through dynamic spawning of processes involving *sublow*. During execution, the *subname* will dynamically spawn processes that will invoke instances of *sublow* executing in parallel. This is accomplished using the calls to *nxtag* and *spawn* in the 200 loop.

More will be said about the details of this mechanism below. At this point it should be stressed that the subroutine *paralg* only records the control dependencies and the unit of computation associated with each node in the control dependency graph. None of these will execute until *paralg* completes. When *paralg* does complete (i.e. executes a return statement), the computation begins by executing those nodes which have *icango* = 0. During execution, all of the units of computation recorded by *paralg* will execute as their dependencies are satisfied. Additional units of computation may be allocated during execution through the *spawn* mechanism.

3.3. Static partition

In the generic example above, the subroutine *paralg* defines the static control flow graph. As we have just pointed out, it is important to understand that none of the units of computation defined through *dep* and *putq* will execute until a return has been executed by *paralg*. Thus the parameters passed through the calling sequence to *subname* must be global variables. That is, every variable passed through the calling sequence to *subname* with a call to *putq* must either reference a variable passed through the calling sequence of *paralg* or must be shared common.

To define the static control flow graph one must write a program that includes a call to *dep* followed by a call to *putq* corresponding to every node on the graph. Each node of the graph must be assigned a *jobtag*. These jobtags must be successive positive integers beginning at 1 and not exceeding 1000. In the program defining the static control flow graph there must be the statements

```
call dep(jobtag,icango,ncheks,list)
call putq(jobtag,subname,<parms>)
```

corresponding to each node. The call to *dep* records the control flow dependencies for the node that has been labeled *jobtag*. The dependencies are defined through specification of

icango = an integer specifying the number of nodes *jobtag* depends on,
ncheks = an integer specifying the number of nodes which depend on *jobtag*,

list = an integer array containing list of identifiers (i.e. jobtags) of the *ncheks* processes dependent upon completion of *jobtag*.

It is essential that at least one node has *icango* = 0 set. Without this condition the program cannot start. There can be several nodes with *icango* = 0. Exactly one node must have the condition *ncheks* = 0 set. The program cannot finish without this. Error messages will be output and the SCHEDULE program will execute a FORTRAN stop if either of these conditions are not met. A unit of computation is 'scheduled' for execution when its *icango* has been set to 0 indicating that it is not waiting upon the completion of any other unit of computation. The *icango* counter of each of the *ncheks* dependents on the *list* will be decremented automatically when *jobtag* completes. The unit of computation labeled *jobtag* is recorded through the call to *putq*. Once the control flow dependencies for *jobtag* have been satisfied, a

```
call subname ( <parms> )
```

will be executed where ⟨parms⟩ represents the list of parameters to be passed to the subroutine *subname*. The call to *putq* records the entry point to *subname* and the addresses of each parameter in the calling sequence ⟨parms⟩. Execution of this unit of computation only occurs after the control flow dependencies have been satisfied and a physical processor has become available. The order of execution is dictated by the order in which schedulable processes get placed on a 'ready to execute' queue called *readyq*. When the number of physical processors is 1, this order is predetermined, but when more than one processor is active, the order is generally nondeterministic. This queue is served by *nprocs* workers on a self scheduled basis. By this we mean that each worker continually gets work off the *readyq*, executes the unit of computation, updates the dependency information, and then tries to get another unit of computation off the *readyq*.

3.4. Dynamic spawning

Many interesting algorithms can be expressed using a static dependency graph. However, there are situations where, for a given problem, it is not known in advance that a code segment will warrant parallel processing until the execution of the program. In these cases the ability to dynamically spawn processes during execution is required. If one wishes to do dynamic spawning within one of the executing processes that has been defined statically, there are three options with varying levels of generality. The simplest case where no work has to be done after spawning occurs has been shown in the generic example given above. Here we show the most general case which allows multiple re-entry and multiple spawning segments:

```
      subroutine subname(<parms>)
      declare global variables

      declare local variables

      logical wait
      external sublow
c
c     resume computation at the label number returned through ientry
c
      go to (1000,2000,...,N000),ientry(mytag,N)
 1000 continue
          .
          .
          .
      do 200 j = 1,nkids
          .
```

```
                .
                .
                call nxtag(mytag,jdummy)
                call spawn(mytag,jdummy,sublow,<parms>)
                .
                .
      200 continue
          label = 2
          if(wait(mytag,label)) return
     2000 continue
                .
                .
          return
                .
                .
                .
     N000 continue
                .
                .
          return
          end
```

The logical function *wait* sets up an implied barrier which will not be crossed until all of the spawned processes have completed execution. Nevertheless, this is not a busy wait. If the reference to *wait* returns the value *true* indicating spawned processes have not completed yet, then a return is made to SCHEDULE *work* routine which then becomes available to participate in executing other schedulable units of computation. This mechanism allows the program to execute on a single processor. The second argument that is passed to *wait* is returned as the value of *ientry* when the process *mytag* is rescheduled for execution after the spawned processes have all completed.

Another construct one might use is to make *wait* a subroutine and call it at this point. Thus the *work* routine executing this process could access the SCHEDULE readyq through *wait* and participate in the execution of other processes until the spawned child processes were completed. This approach was tried but found to be unwieldy when dynamic spawning of several layers was desired. Stack overflow seemed to be the problem in this case. The construct we have shown above does not suffer from this problem because the information is stored in queues that are managed explicitly within SCHEDULE.

Let us call executing process defined by *subname* together with its arguments the parent. In the computation above, the parent process will spawn *nkids* child processes. The parameter *mytag* which appears in the calling sequence of both *nxtag* and *spawn* must contain the *jobtag* of the parent process. The parameter *mytag* must be stored in a global variable which has been passed to the parent *subname* either through the calling sequence or through named common. The calls to *nxtag* and *spawn* must be done together. The statement

call nxtag(mytag,jdummy)

returns a *jobtag* in the parameter *jdummy* which has been assigned by SCHEDULE internally. The control flow dependencies are automatically defined; the each spawned process will be forced to complete before the parent can finish. The statement

call spawn(mytag,jdummy,sublow,<parms>)

is similar in nature to a call to *putq* in the sense that will invoke a call to subroutine *sublow* (⟨parms⟩).

3.5. The SCHEDULE mechanism

The details of the underlying SCHEDULE mechanism which involves queue management and synchronization has been described in [19]. This mechanism has remained essentially the same but implementations for the various machines has been quite tedious in some cases. An important point is that the cost associated with obtaining a physical processor is only incurred with the initial call to *sched*. Moreover, there is no dependence in a SCHEDULE program on the number of physical processors used.

4. An environment for the development of explicitly parallel programs

As we mentioned at the outset, SCHEDULE is a programming aid that is intended to serve as the backbone of an environment for the development of explicitly parallel programs. Our goal is to provide a uniform interface to the parallel capacities provided by existing and impending parallel systems. Included in this are tools for debugging and analyzing the performance of parallel programs. In this section we shall describe some of our goals in this area. Preliminary attempts at achieving these goals have been implemented or are in the design stage. The package has been ported to a number of parallel machines and some experience with porting user codes has been gained. A detailed performance study of the SCHEDULE mechanism has not been undertaken yet but is in preparation. We do have some confidence that when used as intended, very good performance can be expected.

We regard SCHEDULE as a tool primarily designed as an aid to constructing an implementation of a new parallel algorithm. We do not think it is particularly well suited to converting an existing serial code to a parallel version, although this would be possible in some cases. The reason for this statement is that to program with SCHEDULE it is imperative that the large-grain control flow dependencies are well understood by the programmer. This is of course required in any successful implementation of a parallel program. However, with SCHEDULE the data partitioning and construction of the control flow dependency graph are an explicit part of the programming effort. We expect that the designer of a parallel algorithm will have this information naturally at hand. At least this has been our experience in the design of our own parallel algorithms [18]. Enforcing this programming style goes a long way towards avoiding bugs that are often associated with parallel programming.

We have found it very useful to retain the capability of executing a parallel program in serial mode. A SCHEDULE program does not depend upon the number of physical processors available in a given system. It will execute with one processor active. Others have found this to be a useful property of such a programming tool [25]. We find at least two reasons to provide it. First it allows one to develop and test code on a serial machine such as a VAX 11/780 or a workstation. This is quite important when the parallel machine is at a remote site or if the software development environment existing on the parallel machine is inadequate or difficult to use. Second, when developing a new algorithm one would like to be assured that the numerical properties of the algorithm are correct in serial mode before entering the parallel testing regime. Operating in this sequence tends to separate ordinary programming bugs from those associated with parallel programming. Moreover, due to the SCHEDULE mechanism, one can be fairly confident that a parallel bug is due either to incorrect specification of the dependency graph or to incorrect partitioning of data. Explicit synchronization is generally not a part of a SCHEDULE program and thus difficult synchronization bugs do not generally arise. When they do arise it is possible to analyze them graphically using the graphics post processing tool described below.

To aid in ascertaining a correct specification of the control flow dependency graph we envision a tool that will visually represent the units of dependency graph and the execution of the units of computation. A preliminary version of this tool has been implemented and experience with this tool is reported below. Similar ideas have been proposed by Babb [2] and others. In our view, the level of detail required in the environment developed by Babb is too fine. We expect the level of detail in such a representation of a parallel program to roughly correspond to the abstract level at which the programmer has partitioned his parallel algorithm. Let us illustrate this with a simple example. A favorite example of our's is the solution of a triangular linear system partitioned by blocks.

4.1. *Triangular solve example*

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 Fig. 2.

The first step is to solve the system $T_1 x_1 = b_1$, this will determine the solution for the 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

$$b_2 = b_2 - T_2 x_1, \quad b_3 = b_3 - T_3 x_1, \quad b_4 = b_4 - T_4 x_1, \quad b_5 = b_5 - T_5 x_1.$$

Notice that these matrix-vector multiplications can occur in parallel, as there are no dependences. However, there may be several processes attempting to update the value of a vector b_j (for example 4, 8, 11 will update b_4) and this 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:

$$b_3 = b_3 - T_7 x_2, \quad b_4 = b_4 - T_8 x_2, \quad b_5 = b_5 - T_9 x_2.$$

The process is continued until the full solution is determined. The control flow dependency graph for this can be represented as in Fig. 3.

In Fig. 4 we show the output from an executing SCHEDULE implementation of this triangular solve example with 36 rather than 15 nodes. The program was run on an Alliant FX/8. An output file was produced as the program executed which recorded the units of computation as they were defined and executed. The file was then shipped to a SUN workstation where a graphics program interpreted this output, constructed the graph and played back the execution sequence that was run on the Alliant. In the graph shown in Fig. 4

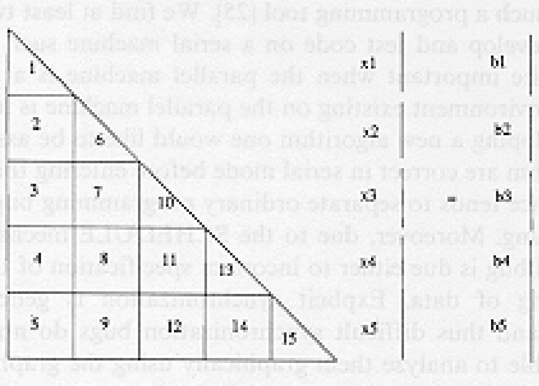

Fig. 2. Partitioning for the triangular solve.

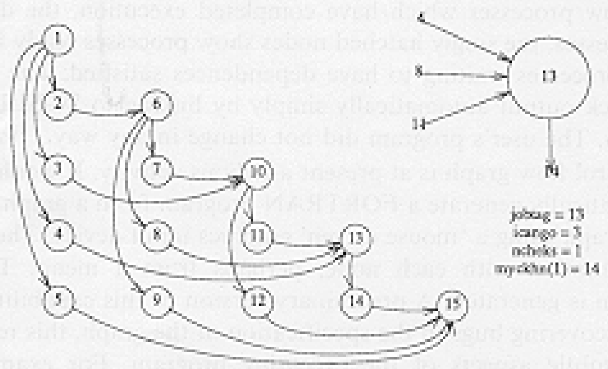

Fig. 3. Control dependency graph for triangular example.

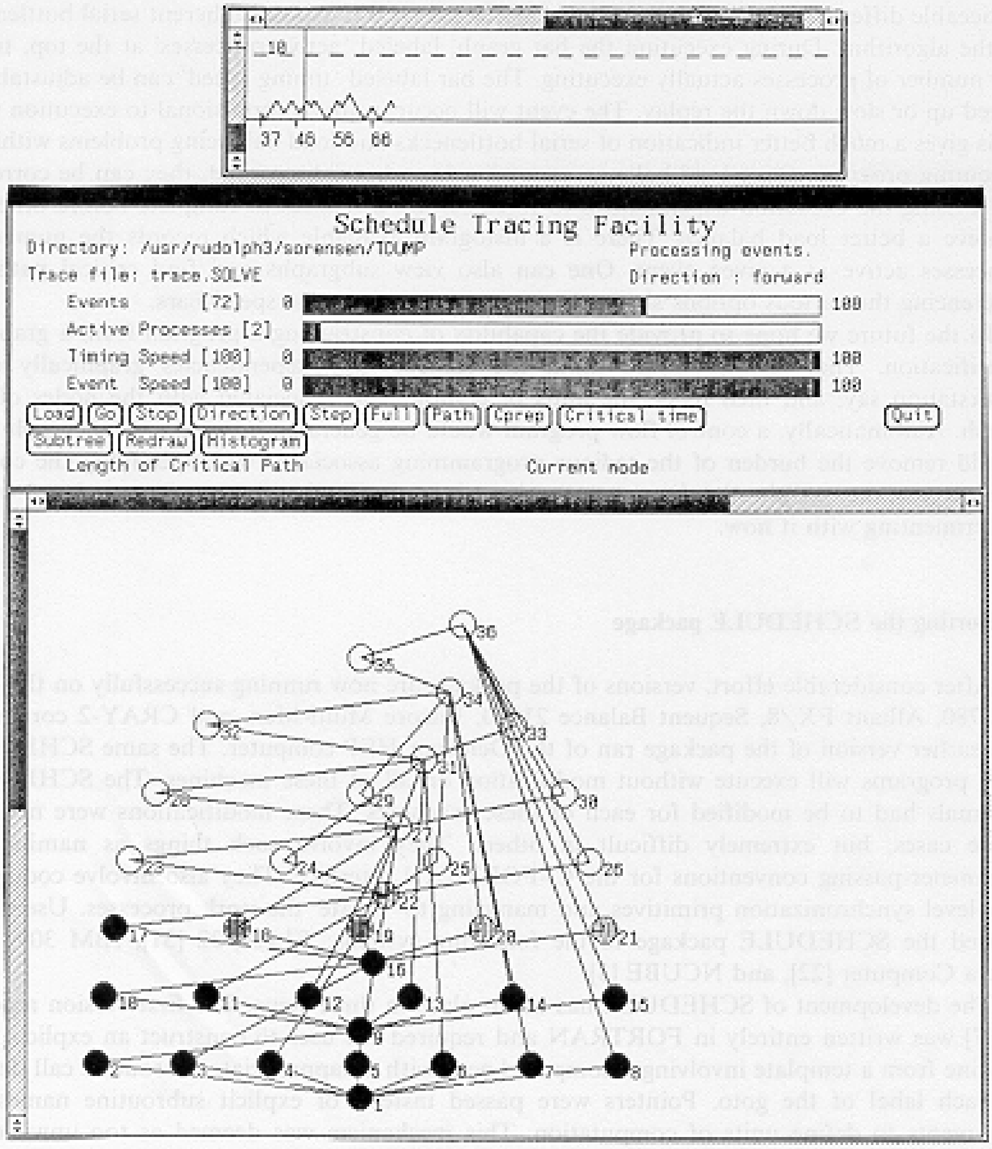
```
Schedule Tracing Facility
Directory: /usr/rudolph3/sorensen/TOUMP          Processing events.
Trace file: trace.SOLVE                          Direction : forward
      Events            [72]      0                          100
      Active Processes  [2]
      Timing Speed    [100]       0                          100
      Event  Speed    [100]       0                          100
Load  Go  Stop  Direction  Step  Full  Path  Cprep  Critical time    Quit
Subtree  Redraw  Histogram
   Length of Critical Path                    Current node
```

Fig. 4. Output from SCHEDULE for triangular example.

the black nodes show processes which have completed execution, the doubly hatched nodes show executing processes, the singly hatched nodes show processes ready for execution, and the white nodes show processes waiting to have dependences satisfied. The program was able to produce this playback output automatically simply by linking to SCHEDULE with a request for the graph option. The user's program did not change in any way.

Defining the control flow graph is at present a tedious activity. It would be desirable to have the ability to automatically generate a FORTRAN program from a graph. In this setting a user would construct a graph using a 'mouse driven' graphics input device. The user would associate each unit of computation with each node, perhaps from a menu. Then on command a FORTRAN program is generated. A preliminary version of this capability is operational.

In addition to discovering bugs in the specification of the graph, this representation is useful in exposing more subtle aspects of the executing program. For example when the graph produced by SCHEDULE is contrasted with the abstract user specified graph there are noticeable differences. The graph produced by SCHEDULE exposes inherent serial bottlenecks in the algorithm. During execution the bar graph, labeled 'active processes' at the top, tracks the number of processes actually executing. The bar labeled 'timing speed' can be adjustable to speed up or slow down the replay. The event will occur in time proportional to execution time. This gives a much better indication of serial bottlenecks and load balancing problems within an executing program. Once load balance anomalies have been discovered, they can be corrected by revising the execution dependencies to force certain processes to complete before others to achieve a better load balance. There is a histogram available which records the number of processes active at a given event. One can also view subgraphs and find critical paths by referencing the various options shown in the boxes just blow the speed bars.

In the future we hope to provide the capability of constructing a program from a graphical specification. That is, one would input the control flow dependencies graphically on a workstation say, and then name the units of computation associated with the nodes of this graph. Automatically, a control flow program would be generated based upon this graph. This would remove the burden of the tedious programming associated with specifying the control flow dependency graph. A rudimentary version of this facility has been constructed and we are experimenting with it now.

5. Porting the SCHEDULE package

After considerable effort, versions of the package are now running successfully on the VAX 11/780, Alliant FX/8, Sequent Balance 21000, Encore Multi-Max, and CRAY-2 computers. An earlier version of the package ran of the Denelcor HEP computer. The same SCHEDULE user programs will execute without modification on all of these machines. The SCHEDULE internals had to be modified for each of these machines. These modifications were minor in some cases, but extremely difficult in others. They involve such things as naming and parameter-passing conventions for the C–FORTRAN interface. They also involve coding the low-level synchronization primitives and managing to 'create' the **work** processes. Users have ported the SCHEDULE package to the following systems: FLEX/32 [31], IBM 3090 [30], Ultra Computer [22], and NCUBE [3].

The development of SCHEDULE has taken place in three steps. The first version reported in [7] was written entirely in FORTRAN and required the user to construct an explicit *work* routine from a template involving a computed goto with an appropriate subroutine call defined at each label of the goto. Pointers were passed instead of explicit subroutine names and arguments to define units of computation. This mechanism was deemed as too unwieldy to explain and use correctly so an alternative was considered. It was decided that a mechanism was needed to record the subroutine names and parameters defining a unit of computation which would look very similar to a standard FORTRAN subroutine call. A similar mechanism was devised for dynamic spawning. This version was reported in [19]. More recently, the syntax of these constructs has been revised, a preprocessor was developed, and a graphics post processing analysis tool was provided.

The new features developed during the second phase obviously required either the use of assembly language or C to be able to record entry points and addresses of memory locations. It was also necessary to be able to be able to make a generic call to a subroutine by pointing to the appropriate entry point. These capabilities were all present within the C language and since it is quite portable we decided to use it. Unfortunately, although both C and FORTRAN are portable, the interface between them is certainly not! This non-portability of the interface led to many months of tedious frustrating effort to overcome. In retrospect it probably would have been better to have written some low-level assembly language routines to record addresses and entry points.

Other difficulties stemming from the individual parallel constructs provided by each of the vendors were expected and did indeed arise. These problems involved process creation, synchronization, and memory management. The initial FORTRAN version was developed on the Denelcor HEP computer [17]. The HEP certainly provided the most convenient support for this development since it was designed at the outset to support explicit parallel programming at a very fine grain level. In fact, the SCHEDULE package represented a structured way to manage the capabilities offered by the HEP. The next machine it was put on was the Alliant FX/8 where most of the continuing development has taken place. In the case of the Alliant the package offers a considerable enhancement of the ability to express parallel algorithms. The Alliant offers a marvelous preprocessing compiler which aids in detecting and exploiting loop based parallelism. However, it offers no assistance in explicit parallel programming and synchronization.

Process creation required to get instances of the *work* routines executing on physical processors was different on each machine. On the Alliant we used the CVD$L CNCALL directive before a loop that performed *nprocs* calls to the subroutine *work*. This concept was straightforward to move to the Sequent where we used the C$DOACROSS directive. However, Sequent required additional system calls *cpus_online* and *m_set_procs* to enable the user to acquire more than half of the available processors on the Sequent. The Encore required the use of their *fork* system call. In this case *nprocs* − 1 instances of *work* were forked and one was called. On both the Sequent and Encore, these processors had to be physically released through a system call or they would remain active unix processes after the program had completed unbeknown to the user. On the CRAY-2, process creation is accomplished using **tskstart** provided by the CRAY multitasking library [1]. This construct also required *nprocs* − 1 calls to *tskstart* to invoke instances of *work* followed by one call to *work*.

For the Alliant FX/8 we coded the low-level synchronization primitives in assembly language using their test-and-set instruction since no synchronization primitives were provided by Alliant. We used the synchronization primitives provided by the system in the cases of Sequent, Encore, and Cray. All of these are versions of *lockoff*. Of course the syntax was different for each of them and we provide a uniform syntax for these primitives to the user for low-level synchronization that has been built out of the machine-dependent versions. For serial machines we provide dummy routines which resolve references to these synchronization primitives but have no functionality.

The most serious difficulties arise from memory management. The Alliant and Cray seemed to have reasonable management of shared memory. However, the Sequent and Encore were very clumsy in this regard. This difficulty stems primarily from the fact that they both rely on a modification of the notion of a UNIX process to achieve parallel processing. These UNIX processes invoked through a *fork* are allowed to share a certain portion of their memory with each other. Allocating this memory and aligning it properly seems to be very tricky. Initially, the user had to become involved in this alignment process on both of these machines. Sequent has rectified this at the price of requiring shared common blocks and C-structures to be named in the link and load command. However, Encore still requires shared common blocks to be padded to get them aligned on page boundaries and a system call must be made by the user to record the length of these blocks. Moreover, various compiler directives must be inserted in each of the machines to invoke proper I/O system calls, assure reentrant code is generated, etc. All of these things would require explicit action on the part of the user without a preprocessor. Thus, as much as we had hoped to avoid any preprocessing it was decided that one had to be provided. The preprocessor automatically inserts the things we have just mentioned in a user code. At present we are not able to provide named common that is private to process in a portable way. This is a feature we intend to provide and the preprocessor will be used to facilitate it. It is our view that common that is not explicitly declared shared should default to private and should have a scope that includes all subroutines on the calling tree below the subroutine that first names the common area. The lifetime of the common area should coincide with the lifetime of this subroutine as well. This would facilitate the use of FORTRAN libraries because an explicit change would not have to be made to the codes to invoke them in parallel. Interestingly enough, this treatment of common is valid within the *existing* FORTRAN standard although no manufacturer has chosen to implement it this way [28,27].

The greatest difficulty by far was the nonstandard C–Fortran interface on the different machines. A minor, but annoying, detail was the convention for distinguishing between C and FORTRAN routines. The Alliant and VAX seemed perfectly compatible with a C routine made FORTRAN callable by preceding the procedure name with an underscore and a FORTRAN routine made C callable simply by referring to it as a procedure invocation with underscore inserted before the subroutine name. (For example the C routine *_foo* is referred to in a FORTRAN call as *CALL FOO*.) The Encore used essentially the same convention but placed the underscore after the routine name. The CRAY-2 required all FORTRAN routines called from C routines to be referred to in capital letters and all C routines called by FORTRAN to be defined in capital letters. The Sequent had the most insidious version of this requiring the FORTRAN routine to reference a C routine by placing an underscore before the name of the routine. (For example the C routine *foo* is referred to in a FORTRAN call as *CALL _FOO*.) FORTRAN routines are made C callable by defining them with an underscore before the subroutines name. This is of course exactly opposite to all of the other machines and required intervention by our preprocessor to overcome. The Sequent also provided the difficulty of having the parameters passed in reverse order between C and FORTRAN routines. Moreover, when a procedure name is passed, the Sequent preprocessor inserts a non-user variable in the calling sequence. These things required a complete rewrite of the C routines *putq*, *spawn*, and *sched*. It was necessary to write a complicated case statement and to pass a variable with the number of passed parameters in the calling sequence to these routines in order to handle variable calling sequences. Again the preprocesser was used to insert the number of parameters. In this application there was no need to pass rectangular arrays between the C and FORTRAN routines so we did not suffer from the incompatibility of the storage schemes of these two languages.

6. Conclusions

Two concepts for achieving transportable software for high-performance computers have been presented. Experience with the module approach to encapsulating and machine-dependent code within a few modules to achieve high performance has been rewarding. This concept has been very successful within the context of linear algebra software and would be useful to try elsewhere.

The construction of SCHEDULE has evolved naturally during the course of our experience with programming various parallel machines. We are primarily motivated by the lack of uniformity and limited capabilities offered by vendors for explicit parallel programmming. We are deeply indebted to our colleagues who are active in similar pursuits and we have used many of their ideas in constructing SCHEDULE. However, although we have been greatly influenced by their work, we feel that our experience with numerical software has guided us in subtly different directions. It is our view that the user interface at this point in time is of paramount importance to the utility of such an effort. Our goal has been to provide a methodology that is within the grasp of a capable FORTRAN programmer and to provide syntax which does not represent a radical departure in appearance from FORTRAN programming. We have already modified this interface several times based upon interaction with users. The current approach has evolved from a package written entirely in FORTRAN to one requiring a C–FORTRAN interface. The implementation of SCHEDULE for some machines has been made much more difficult with this requirement due to the lack of standardization of such an interface. However, our goal of providing a clean user interface with the parallel capabilities seems to have been achieved through this mixed language approach.

A number of codes have been implementing using SCHEDULE and have been ported to the machines listed above with out change. In addition to some toy programs used for debugging SCHEDULE, several codes have been written and executed using SCHEDULE. These codes include the algorithm TREEQL for the symmetric tridiagonal eigenvalue problem [20], a domain decomposition code for singularly perturbed convection-diffusion PDE [7], an adaptive quadrature code [8], and a block preconditioned conjugate gradient code for systems arising in reservoir simulation [10], a multifrontal code for sparse elimination [21], and a code for multivariate tensor product spline approximation [26].

An important step in the development of SCHEDULE will be to provide the ability of constructing the program from a graph. At the moment it is possible to do this when the graph is small and static. However, the code produced is straight line code. It is a considerable challenge to produce a loop structure automatically.

Another interesting line of development would be to implement the package for a distributed memory computer such as the Intel iPSC, or the NCUBE machines. A version of SCHEDULE has been implemented on the NCUBE by Beguelin [3]. In this implementation one node of the NCUBE plays the role of master and manages the dependency graph and associated queues. This master process monitors the *work* routines operating on the other nodes and sends them units of computation when they are free by message passing. There are several performance issues that need to be studied in this setting. The most serious of these is how to avoid having to send the data associated with a unit of computation to the physical processor that will execute it.

Our experience with SCHEDULE has been encouraging for the most part. We do not view it as a 'solution' to the software problem we face in parallel programming. However, we do think this will be useful in the short term and perhaps will have some influence on the development of a long term solution.

References

[1] *CRAY 2 Multitasking Users Guide*, Cray Research Inc, Minn, MN, 1986.

[2] R.G. Babb, Parallel processing with large grain data flow techniques, *IEEE Comput.* 17 (7) (1984) 55–61.

[3] A. Beguelin, Private Communications, 1987.

[4] M. Berry, K. Gallivan, W. Harrod, W. Jalby, S. Lo, U. Meier, B. Philippe and A. Sameh, Parallel algorithms on the CEDAR system, CSRD Report No. 581, 1986.

[5] C. Bischoff and C. Van Loan, The WY representation for products of Householder matrices, *SIAM J. Sci. Statist. Comput.* 8 (2) (1987).

[6] J.C. Browne, Framework for formulation and analysis of parallel computation structures, *Parallel Comput.* 3 (1) (1986) 1–9.

[7] R. Chin, G. Hedstrom, F. Howes and J. McGraw, Parallel computation of multiple-scale problems, in: E.A. Wouk, ed., *New Computing Environments: Parallel, Vector, and Systolic* (SIAM, Philadelphia, PA, 1986) 134–151.

[8] S. Comer, Private Communication, 1986.

[9] Alliant Computer Systems Corp, *Alliant FX/Fortran Programmer's Handbook*, Acton, MA, 1985.

[10] J.C. Diaz, Calculating the block preconditioner on Parallel multivector processors, *Proc. Workshop on Applied Computing in The Energy Field*, Stillwater, OK (1986).

[11] J.J. Dongarra, Performance of various computers using standard linear equations software in a Fortran environment, Argonne National Laboratory MCS-TM-23 1987.

[12] J.J. Dongarra, J. Bunch, C. Moler and G. Stewart, *LINPACK Users' Guide* (SIAM, Philadelphia, PA, 1979).

[13] J.J. Dongarra, J. DuCroz, I. Duff and S. Hammarling, A proposal for a set of level 3 basic linear algebra subprograms, Argonne National Laboratory Report ANL-MCS-TM-88, 1987.

[14] J.J. Dongarra, J. DuCroz, S. Hammarling and R. Hanson, An extended set of Fortran basic linear algebra subprograms, Argonne National Laboratory Report, ANL-MCS-TM-41 (Revision 3), 1986.

[15] J.J. Dongarra and I.S. Duff, Advanced architecture computers, Argonne National Laboratory Report, ANL-MCS-TM-57 (Revision 1), 1987.

[16] J.J. Dongarra and S.C. Eisenstat, Squeezing the most out of an algorithm in Cray Fortran, *ACM Trans. Math. Software* 10 (3) (1984) 221–230.

[17] J.J. Dongarra and D. Sorensen, SCHEDULE: Tools for developing and analyzing parallel Fortran programs, Argonne National Laboratory Report, ANL-MCS-TM-86, 1986.

[18] J.J. Dongarra and D.C. Sorensen, Linear algebra on high-performance computers, in: M. Feilmeier et al., eds., *Parallel Computing 85* (North-Holland, Amsterdam, 1986) 3–32.

[19] J.J. Dongarra and D.C. Sorensen, A portable environment for developing parallel Fortran programs, *Parallel Comput.* 5 (1) (1987) 175–186.

[20] J.J. Dongarra and D.C. Sorensen, A fully parallel algorithm for the symmetric eigenvalue problem, *SIAM J. Sci. Statist. Comput.* 8 (2) (1987).

[21] I. Duff, Parallel implementation of multifrontal schemes, *Parallel Comput.* 3 (1986) 193–204.

[22] A. Greenbaum, Private Communications, 1987.

[23] H. Jordan, HEP architecture, programming and performance, in: J. Kowalik, ed., *Parallel MIMD Computation: HEP Supercomputer and Its Applications* (MIT Press, Cambridge, MA, 1985).

[24] C. Lawson, R. Hanson, D. Kincaid and F. Krogh, Basic linear algebra subprograms for Fortran usage, *ACM Trans. Math. Software* 5 (1979) 308–323.

[25] E. Lusk and R. Overbeek, Implementation of monitors with macros: A programming aid for the HEP and other parallel processors, Argonne National Laboratory Report, ANL-83-97, 1983.

[26] R. Maestro, Private Communications, 1987.

[27] B.T. Smith, Private Communications, 1987.

[28] D. Snelling, Private Communications, 1987.

[29] J. Van Rosendale and P. Mehrotra, The BLAZE language: A parallel language for scientific programming, ICASE Report #85-29, 1985.

[30] D. Vasichek, Private Communications, 1987.

[31] M.A. Vavalis, Private Communications, 1987.

[32] J.R. McGraw et al., SISAL: Streams and iterations in a single assignment language, Language Reference Manual, Ver. 1.2, Lawerence Livermore National Laboratory.