Looking back, What's ahead

Yves Robert Per Stenstrom Viktor K. Prasanna

December 15, 2011

Chapter 1

Looking Back at Dense Linear Algebra Software

Piotr Luszczek
Jakub Kurzak
Jack Dongarra

Abstract

Over the years, computational physics and chemistry served as an ongoing source of problems that demanded the ever increasing performance from hardware as well as the software that ran on top of it. Most of these problems could be translated into solutions for systems of linear equations: the very topic of numerical linear algebra. Seemingly then, a set of efficient linear solvers could be solving important scientific problems for years to come. We argue that dramatic changes in hardware designs precipitated by the shifting nature of the marketplace of computer hardware had a continuous effect on the software for numerical linear algebra. The extraction of high percentages of peak performance continues to require adaptation of software. If the past history of this adaptive nature of linear algebra software is any guide then the future theme will feature changes as well – changes aimed at harnessing the incredible advances of the evolving hardware infrastructure.

1.1 Introduction

Over the decades, dense linear algebra has been an indispensible component of science and engineering. While the mathematical foundations and application methodology has changed little, the hardware has undergone a tumultous transition. The latter precipitated a number of paradigm shifts in the way the linear algebra software is implemented. Indeed, the ever evolving hardware would quickly render old code inadequate in terms of performance. The external interfaces to the numerical software routines have undergone only minor adjustment which is in line with the unchanged mathematical forumlation of the problem of solving a system of linear equations. The internal implementation of these interfaces was changing to accomodate drastic redesign of the underlying hardware technology. The internals have been modularized to ease the implementation process. These modules over time have become the building blocks of new generations of the numerical linear algebra libraries and made the effort more managable in the long run. Over time, the number and functionality of these building blocks have increased but the delegation of responsibilities between various modules inside the software stack has been retained. The most recent increase in hardware parallelism further altered the established composition process of the modules by necessitating the use of explicit scheduling mechanism which needed to be handled manually in the compositing code or externally with a software scheduler. The rapid transformation of computer hardware has not been ongoing and is expected to be continuing into the future. With it, the software will evolve further and our hope is that the design decision made in the past will allow for a smooth transition by reusing the tested and optimized libraries we have become accustomed to.

1.2 Motivation: Plasma Physics and Electronic Structure Calculation

Computational experiments of self-sustaining fusion reactions could give us an informed perspective on how to build a device capable of producing and controlling the high performance [1]. Modeling the heating response of plasma due to radio frequency (RF) waves in the fast wave time scale leads to solving the generalized Helmholtz equation. The time harmonic terms of effective approximations of the electric field, magnetic field, and distribution function as a time-averaged equilibrium satisfy the equation. The Scientific Discovery through Advanced Computing project (SciDAC) Numerical Computation of Wave Plasma-Interactions in Multi-dimensional Systems developed and implemented a simulation code that gives insight into how electromagnetic waves can be used for driving current flow, heating and controlling instabilities in the plasma. The code is called AORSA [2, 3, 4, 5, 6] and stands for All ORders Spectral Algorithm. The resulting computation requires a solution of a system of linear equations exceeding half a million unknowns [7].

In quantum chemistry, most of the scientific simulation codes result in a numerical linear algebra problem that may readily be solved with the ScaLAPACK library [8, 9]. For example, early versions of ParaGauss [10, 11, 12, 13] relied on diagonalization of the Kohn-Sham matrix and the parallelization method of choice relied on the irreducible representations of the point group. The submatrices diagonalize in parallel and the number of them depended on the symmetry group. When using one of ScaLAPACK's parallel eigensolvers it is possible to achieve speedup even for a Kohn-Sham matrix with only one block. A different use of the BLAS library occurs in UTChem [14] – an application code that collects a number of methods that allow for accurate and efficient calculations for computational chemistry of electronic structure problems. Both the ground and excited states of molecular systems are covered. In supporting a number of single-reference many-electron theories such as configuration-interaction theory, coupled-cluster theory, and Møller-Plesset perturbation theory, UTChem derives working equations using a symbolic manipulation program called Tensor Contraction Engine (TCE) [15]. It automates the process of deriving final formulas and generation of the execution program. The contraction of creation and annihilation operators according to Wick's theorem, consolidation of identical terms, and reduction of the expressions into the form of tensor contractions controlled by permutation operators are all done automatically by TCE. If tensor contractions are treated as a collection of multi-dimensional summations of the product of a few input arrays then the commutative, associative, and distributive properties of the summation allow for a number of execution orders, each of which having different execution rates when mapped to a particular hardware architecture. Also, some of the execution orders would result in calls to BLAS, which provides a substantial increase in floating-point execution rate. The current TCE implementation generates many-electron theories that are limited to non-relativistic Hartree-Fock formulation with reference wave functions but it is possible to extend it to relativistic 2- and 4-component reference wave functions.

1.3 Problem Statement in Matrix Terms

Most dense linear systems solvers rely on a decompositional approach [16]. The general idea is the following: given a problem involving a matrix A, one factors or decomposes A into a product of simpler matrices from which the problem can easily be solved. This divides the computational problem into two parts: first determine an appropriate decomposition, and then use it in solving the problem at hand. Consider the problem of solving the linear system:

$$Ax = b \tag{1.1}$$

where A is a nonsingular matrix of order n. The decompositional approach begins with the observation that it is possible to factor A in the form:

$$A = LU \tag{1.2}$$

where L is a lower triangular matrix (a matrix that has only zeros above the diagonal) with ones on the diagonal, and U is upper triangular (with only zeros below the diagonal). During the decomposition process, diagonal elements of A (called pivots) are used to divide the elements below the diagonal. If matrix A has a zero pivot, the process will break with division-by-zero error. Also, small values of the pivots excessively amplify the numerical errors of the process. So for numerical stability, the method needs to interchange rows of the matrix or make sure pivots are as large (in absolute value) as possible.

This observation leads to a row permutation matrix P and modifies the factored form to:

$$PA = LU \tag{1.3}$$

The solution can then be written in the form:

$$x = A^{-1}Pb \tag{1.4}$$

which then suggests the following algorithm for solving the system of equations:

- Factor A according to Eq. (1.3)
- Solve the system $Ly = Pb$
- Solve the system $Ux = y$

This approach to matrix computations through decomposition has proven very useful for several reasons. First, the approach separates the computation into two stages: the computation of a decomposition, followed by the use of the decomposition to solve the problem at hand. This can be important, for example, if different right hand sides are present and need to be solved at different points in the process. The matrix needs to be factored only once and reused for the different right hand sides. This is particularly important because the factorization of A, step 1, requires $O(n^3)$ operations, whereas the solutions, steps 2 and 3, require only $O(n^2)$ operations. Another aspect of the algorithm's strength is in storage: the L and U factors do not require extra storage, but can take over the space occupied initially by A. For the discussion of coding this algorithm, we present only the computationally intensive part of the process, which is step 1, the factorization of the matrix.

Decompositional technique can be applied to many different matrix types:

$$A_1 = LL^T \quad A_2 = LDL^T \quad PA_3 = LU \quad A_4 = QR \tag{1.5}$$

such as symmetric positive definite (A_1), symmetric indefinite (A_2), square non-singular (A_3), and general rectangular matrices (A_4). Each matrix type will require a different algorithm: Cholesky factorization, Cholesky factorization with pivoting, LU factorization, and QR factorization, respectively.

1.4 Introducing LU: a Simple Implementation

For the first version, we present a straightforward implementation of LU factorization. It consists of n—1 steps, where each step introduces more zeros below the diagonal, as shown in Figure 1.1.

Tools often used to teach Gaussian elimination include MATLAB and Python. They are scripting languages that make developing matrix algorithms very simple. The notation might seem very unusual to people familiar with other scripting languages because it is oriented to process multidimensional arrays. The unique features of the language that we use in the example code are:

- Transposition operator for vectors and matrices: ′ (single quote)
- Matrix indexing specified as:
  - Simple integer values: `A(m, k)`
  - Ranges: `A(k:n, k)`
  - Other matrices: `A([k m], : )`
- Built-in matrix functions such as **size** or **shape** (returns matrix dimensions), **tril** (returns the lower triangular portion of the matrix), **triu** (returns the upper triangular portion of the matrix), and **eye** (returns an identity matrix, which contains only zero entries, except for the diagonal, which is all ones).

The algorithm presented in Figure 1.1 is row-oriented, in the sense that we are taking a scalar multiple of the "pivot" row and adding it to the rows below to introduce zeros below the diagonal. The beauty of the algorithm lies in its similarity to the mathematical notation. Hence, this is the preferred way of teaching the algorithm for the first time so that students can quickly turn formulas into running code.

```
function (L,U,p) = lutx(A)
%LUTX Triangular factorization, textbook version
% (L,U,p) = lutx(A) produces a unit lower triangular matrix L,
% an upper triangular matrix U, and a permutation vector p,
% so that L*U = A(p,:)
 (n,n) = size(A);
 p = (1:n)';

 for k = 1:n-1
% Find index of largest element below diagonal in k-th column
 (r,m) = max(abs(A(k:n,k)));
 m = m+k-1;
 % Skip elimination if column is zero
 if (A(m,k) ˜= 0)
  % Swap pivot row
  if (m ˜= k)
     A((k m),:) = A((m k),:);
     p((k m)) = p((m k));
  end
  % Compute multipliers
  i = k+1:n;
  A(i,k) = A(i,k)/A(k,k);

  % Update the remainder of the matrix
  j = k+1:n;
  A(i,j) = A(i,j) - A(i,k)*A(k,j);
 end
end

% Separate result
L = tril(A,-1) + eye(n,n);
U = triu(A);
```

```
import numpy

def lutx(A):
  """
  lutx  Triangular factorization, textbook version
  """

  n, n = A.shape
  p = numpy.arange(n)

  # Find index of largest element below diagonal
  for k in range(n-1):
    r, m = max(numpy.abs(A(k:n, k)))
    m += k

    if A(m, k) != 0:
      # Swap pivot row
      if m != k:
        A((k, m), :) = A((m, k), :)
        p((k, m)) = p((m, k))

      # Compute multipliers
      i = range(k, n)
      A(i, k) /= A(k, k)

      # Update the remainder of the matrix
      j = range(k, n)
      A(i, j) -= A(i, k) * A(k, j)

  L = numpy.tril(A, -1) + numpy.eye(n, n)
  U = numpy.triu(A)

  return L, U, p
```

Figure 1.1: Simple implementation of LU in MATLAB (left) and Python (right).

The convenience has its price, however, and it is not only related to the overhead of scripting language interpretation. In the 1970s, Fortran was the language for scientific computations. Fortran stores two-dimensional arrays by column. Accessing the array in a row-wise fashion within the matrix could involve successive memory reference to locations separated from each other by a large increment, depending on the size of the declared array. The situation was further complicated by the operating system's use of memory pages to effectively control memory usage. With a large matrix and a row-oriented algorithm in a Fortran environment, an excessive number of page swaps might be generated in the process of running the software. Cleve Moler pointed this out in the 1970s [17].

To avoid this situation, one needed simply to interchange the order of the innermost nested loops on i and j. This simple change resulted in more than 30% savings in wall-clock time to solve problems of size 200 on an IBM 360/67. Expressivness of the code was thus traded for efficiency by using a less obvious ordering of loops and the use of a much more obscure (by today's standard) language.

1.5 Implementation for Vector Computers

In the second half of the seventies the introduction of vector computer systems marked the beginning of modern Supercomputing. These systems offered a performance advantage of at least one order of magnitude over conventional systems of that time. Raw performance was the main if not the only selling argument. In the first half of the eighties the integration of vector systems in conventional computing environments became more important. Only the manufacturers, which provided standard programming environments, operating systems and key applications, were successful in getting industrial customers and survived. Performance was mainly increased by improved chip technologies and

```
      subroutine dgefa(a,lda,n,ipvt,info)
      integer lda,n,ipvt(1),info
      double precision a(lda,1)
      double precision t
      integer idamax,j,k,kp1,l,nm1
c
c     gaussian elimination with partial pivoting
c
      info = 0
      nm1 = n - 1
      if (nm1 .lt. 1) go to 70
      do 60 k = 1, nm1
         kp1 = k + 1
c
c        find l = pivot index
c
         l = idamax(n-k+1,a(k,k),1) + k - 1
         ipvt(k) = l
c
c        zero pivot implies this column is already triangularized
c
         if (a(l,k) .eq. 0.0d0) go to 40
c
c           interchange if necessary
c
            if (l .eq. k) go to 10
               t = a(l,k)
               a(l,k) = a(k,k)
               a(k,k) = t
   10       continue
c
c           compute multipliers
c
            t = -1.0d0/a(k,k)
            call dscal(n-k,t,a(k+1,k),1)
c
c           row elimination with column indexing
c
            do 30 j = kp1, n
               t = a(l,j)
               if (l .eq. k) go to 20
                  a(l,j) = a(k,j)
                  a(k,j) = t
   20          continue
               call daxpy(n-k,t,a(k+1,k),1,a(k+1,j),1)
   30       continue
            go to 50
   40    continue
            info = k
   50    continue
   60 continue
   70 continue
      ipvt(n) = n
      if (a(n,n) .eq. 0.0d0) info = n
      return
      end
```

Figure 1.2: LINPACK variant of LU factorization (this is the original FORTRAN 66 code - if LINPACK was written today it would have used Fortran 2008.)

by producing shared memory multi-processor systems. They were able in one step to perform a single operation on a relatively large number of operands stored in vector registers. Expressing matrix algorithms as vector-vector operations was a natural fit for this type of machines [18]. However, some of the vector designs had a limited ability to load and store the vector registers in main memory. A technique called chaining allowed this limitation to be circumvented by moving data between the registers before accessing main memory. Chaining required recasting linear algebra in terms of matrix-vector operations.

Vector architectures exploit pipeline processing by running mathematical operations on arrays of data in a simultaneous or pipelined fashion. Most algorithms in linear algebra can be easily vectorized. Therefore, in the late 70s there was an effort to standardize vector operations for use in scientific computations. The idea was to define some simple, frequently used operations and implement them on various systems to achieve portability and efficiency. This package came to be known as the Level 1 Basic Linear Algebra Subprograms (BLAS) or Level 1 BLAS [19].

The term Level 1 denotes vector-vector operations. As we will see, Level 2 (matrix-vector operations) [20, 21], and Level 3 (matrix-matrix operations) [22, 23] play important roles as well. In the 1970s, the algorithms of dense linear algebra were implemented in a systematic way by the LINPACK project [24]. LINPACK is a collection of Fortran subroutines that analyze and solve linear equations and linear least-squares problems. The package solves linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and square tridiagonal (only diagonal, super-diagonal and sub-diagonal are present). In addition, the package computes the QR (matrix Q is unitary or hermitian and R is upper trapezoidal) and singular value decompositions of rectangular matrices and applies them to least-squares problems. LINPACK uses column-oriented algorithms, which increase efficiency by preserving locality of reference. By column orientation, we mean that the LINPACK code always references arrays down columns, not across rows. This is important since Fortran stores arrays in column-major order. This means that as one proceeds down a column of an array, the memory references proceed sequentially through memory. Thus, if a program references an item in a particular block, the next reference is likely to be in the same block.

The software in LINPACK was kept machine-independent partly through the introduction of the Level 1 BLAS routines. Calling Level 1 BLAS did almost all of the computation. For each machine, the set of Level 1 BLAS would be implemented in a machine-specific manner to obtain high performance. The Level 1 BLAS subroutines DAXPY, DSCAL, and IDAMAX are used in the routine DGEFA.

It was presumed that the BLAS operations would be implemented in an efficient, machine-specific way suitable for the computer on which the subroutines were executed. On a vector computer, this could translate into a simple, single vector operation. This avoided leaving the optimization up to the compiler and explicitly exposing a performance-critical operation.

In a sense, then, the beauty of the original code was regained with the use of a new vocabulary to describe the algorithms: the BLAS. Over time, the BLAS became a widely adopted standard and were most likely the first to enforce two key aspects of software: modularity and portability. Again, these are taken for granted today, but at the time they were not. One could have the cake of compact algorithm representation and eat it too, because the resulting Fortran code was portable.

Most algorithms in linear algebra can be easily vectorized. However, to gain the most out of such architectures, simple vectorization is usually not enough. Some vector computers are limited by having 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 operations together and to 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.

Thus, as computer architectures became more complex in the design of their memory hierarchies, it became necessary to increase the scope of the BLAS routines from Level 1 to Level 2 and Level 3.

1.6 Implementation on RISC Processors

RISC computers were introduced in the late 1980s and early 1990s. While their clock rates might have been comparable to those of the vector machines, the computing speed lagged behind due to their lack of vector registers. Another deficiency was their creation of a deep memory hierarchy with multiple levels of cache memory to alleviate the scarcity of bandwidth that was, in turn, caused mostly by a limited number of memory banks. The eventual success of this architecture is commonly attributed to the right price point and astonishing improvements in performance over time as predicted by Moore's Law [25]. With RISC computers, the linear algebra algorithms had to be redone yet again. This time, the formulations had to expose as many matrix-matrix operations as possible, which guaranteed good cache reuse.

As mentioned before, the introduction in the late 1970s and early 1980s of vector machines brought about the development of another variant of algorithms for dense linear algebra. This variant was centered on the multiplication of a matrix by a vector. These subroutines were meant to give improved performance over the dense linear algebra subroutines in LINPACK, which were based on Level 1 BLAS. In the late 1980s and early 1990s, with the introduction of RISC-type microprocessors (the "killer micros") and other machines with cache-type memories, we saw the development of LAPACK [26] Level 3 algorithms for dense linear algebra. A Level 3 code is typified by the main Level 3 BLAS, which, in this case, is matrix multiplication [27].

The original goal of the LAPACK project was to make the widely used LINPACK library run efficiently on vector and shared-memory parallel processors. On these machines, LINPACK is inefficient because its memory access patterns disregard the multilayered memory hierarchies of the machines, thereby spending too much time moving data instead of doing useful floating-point operations. LAPACK addresses this problem by reorganizing the algorithms to use block matrix operations, such as matrix multiplication, in the innermost loops (see the paper by E. Anderson and J. Dongarra under "Further Reading"). These block operations can be optimized for each architecture to account for its memory hierarchy, and so provide a transportable way to achieve high efficiency on diverse modern machines.

Here we use the term "transportable" instead of "portable" because, for fastest possible performance, LAPACK requires that highly optimized block matrix operations be implemented already on each machine. In other words, the correctness of the code is portable, but high performance is not – if we limit ourselves to a single Fortran source code.

```
      SUBROUTINE DGETRF(M, N, A, LDA, IPIV, INFO)
      INTEGER          INFO, LDA, M, N
      INTEGER          IPIV( * )
      DOUBLE PRECISION   A( LDA, * )
      DOUBLE PRECISION   ONE
      PARAMETER         ( ONE = 1.0D+0 )
      INTEGER          I, IINFO, J, JB, NB
     EXTERNAL    DGEMM, DGETF2, DLASWP, DTRSM
      EXTERNAL       XERBLA
      INTEGER          ILAENV
      EXTERNAL          ILAENV
      INTRINSIC         MAX, MIN
      INFO = 0
      IF( M.LT.0 ) THEN
        INFO = -1
      ELSE IF( N.LT.0 ) THEN
        INFO = -2
      ELSE IF( LDA.LT.MAX( 1, M ) ) THEN
        INFO = -4
      END IF
      IF( INFO.NE.0 ) THEN
        CALL XERBLA( 'DGETRF', -INFO )
        RETURN
      END IF
      IF( M.EQ.0 .OR. N.EQ.0 ) RETURN
      NB = ILAENV( 1, 'DGETRF', ' ', M, N, -1, -1 )
      IF( NB.LE.1 .OR. NB.GE.MIN( M, N ) ) THEN
        CALL DGETF2( M, N, A, LDA, IPIV, INFO )
      ELSE
         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
      END IF
      RETURN
      END
```

Figure 1.3: LAPACK's LU factorization routine DGETRF (FORTRAN 77 coding.)

LAPACK can be regarded as a successor to LINPACK in terms of functionality, although it doesn't always use the same function-calling sequences. As such a successor, LAPACK was a win for the scientific community because it could keep LINPACK's functionality while getting improved use out of new hardware.

Most of the computational work in the algorithm from Figure 1.3 is contained in three routines:

- DGEMM - Matrix-matrix multiplication
- DTRSM - Triangular solve with multiple right hand sides
- DGETF2 - Unblocked LU factorization for operations within a block column

One of the key parameters in the algorithm is the block size, called NB here. If NB is too small or too large, poor performance can result-hence the importance of the ILAENV function, whose standard implementation was meant to be replaced by a vendor implementation encapsulating machine- specific parameters upon installation of the LAPACK library. At any given point of the algorithm, NB columns or rows are exposed to a well-optimized Level-3 BLAS. If NB is 1, the algorithm is equivalent in performance and memory access patterns to the LINPACK's version.

Matrix-matrix operations offer the proper level of modularity for performance and transportability across a wide range of computer architectures, including parallel systems with memory hierarchy. This enhanced performance is primarily due to a greater opportunity for reusing 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 through the memory hierarchy. This improvement can bring a three- to ten-fold improvement in performance on modern computer architectures.

The jury is still out concerning the productivity of writing and reading the LAPACK code: how hard is it to generate the code from its mathematical description? The use of vector notation in LINPACK is

```
      SUBROUTINE PDGETRF( M, N, A, IA, JA, DESCA, IPIV, INFO )
      INTEGER            IA, INFO, JA, M, N
      INTEGER            DESCA( * ), IPIV( * )
      DOUBLE PRECISION   A( * )
      INTEGER            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTYPE_,
     $                   LLD_, MB_, M_, NB_, N_, RSRC_
      PARAMETER          ( BLOCK_CYCLIC_2D = 1, DLEN_ = 9, DTYPE_ = 1,
     $                     CTXT_ = 2, M_ = 3, N_ = 4, MB_ = 5, NB_ = 6,
     $                     RSRC_ = 7, CSRC_ = 8, LLD_ = 9 )
      DOUBLE PRECISION   ONE
      PARAMETER          ( ONE = 1.0D+0 )
      CHARACTER          COLBTOP, COLCTOP, ROWBTOP
      INTEGER            I, ICOFF, ICTXT, IINFO, IN, IROFF, J, JB, JN,
     $                   MN, MYCOL, MYROW, NPCOL, NPROW
      INTEGER            IDUM1( 1 ), IDUM2( 1 )
      EXTERNAL           BLACS_GRIDINFO, CHK1MAT, IGAMN2D, PCHK1MAT,
     $                   PB_TOPGET, PB_TOPSET, PDGEMM, PDGETF2,
     $                   PDLASWP, PDTRSM, PXERBLA
      INTEGER            ICEIL
      EXTERNAL           ICEIL
      INTRINSIC          MIN, MOD
*     Get grid parameters
      ICTXT = DESCA( CTXT_ )
      CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )
*     Test the input parameters
      INFO = 0
      IF( NPROW.EQ.-1 ) THEN
         INFO = -(600+CTXT_)
      ELSE
         CALL CHK1MAT( M, 1, N, 2, IA, JA, DESCA, 6, INFO )
         IF( INFO.EQ.0 ) THEN
            IROFF = MOD( IA-1, DESCA( MB_ ) )
            ICOFF = MOD( JA-1, DESCA( NB_ ) )
            IF( IROFF.NE.0 ) THEN
               INFO = -4
            ELSE IF( ICOFF.NE.0 ) THEN
               INFO = -5
            ELSE IF( DESCA( MB_ ).NE.DESCA( NB_ ) ) THEN
               INFO = -(600+NB_)
            END IF
         END IF
         CALL PCHK1MAT( M, 1, N, 2, IA, JA, DESCA, 6, 0, IDUM1,
     $                  IDUM2, INFO )
      END IF
      IF( INFO.NE.0 ) THEN
         CALL PXERBLA( ICTXT, 'PDGETRF', -INFO )
         RETURN
      END IF
*     Quick return if possible
      IF( DESCA( M_ ).EQ.1 ) THEN
         IPIV( 1 ) = 1
         RETURN
      ELSE IF( M.EQ.0 .OR. N.EQ.0 ) THEN
         RETURN
      END IF
*     Split-ring topology for the communication along process rows
      CALL PB_TOPGET( ICTXT, 'Broadcast', 'Rowwise', ROWBTOP )
      CALL PB_TOPGET( ICTXT, 'Broadcast', 'Columnwise', COLBTOP )
      CALL PB_TOPGET( ICTXT, 'Combine', 'Columnwise', COLCTOP )
      CALL PB_TOPSET( ICTXT, 'Broadcast', 'Rowwise', ROWBTOP )
      CALL PB_TOPSET( ICTXT, 'Broadcast', 'Columnwise', COLBTOP )
      CALL PB_TOPSET( ICTXT, 'Combine', 'Columnwise', COLCTOP )
      RETURN
      END
```

```
      CALL PB_TOPSET( ICTXT, 'Broadcast', 'Rowwise', 'S-ring' )
      CALL PB_TOPSET( ICTXT, 'Broadcast', 'Columnwise', ' ' )
      CALL PB_TOPSET( ICTXT, 'Combine', 'Columnwise', ' ' )
*     Handle the first block of columns separately
      MN = MIN( M, N )
      IN = MIN( ICEIL( IA, DESCA( MB_ ) )*DESCA( MB_ ), IA+M-1 )
      JN = MIN( ICEIL( JA, DESCA( NB_ ) )*DESCA( NB_ ), JA+MN-1 )
      JB = JN - JA + 1
*     Factor diagonal and subdiagonal blocks and test for exact
*     singularity.
      CALL PDGETF2( M, JB, A, IA, JA, DESCA, IPIV, INFO )
      IF( JB+1.LE.N ) THEN
*        Apply interchanges to columns JN+1:JA+N-1.
         CALL PDLASWP( 'Forward', 'Rows', N-JB, A, IA, JN+1, DESCA,
     $                 IA, IN, IPIV )
*        Compute block row of U.
         CALL PDTRSM( 'Left', 'Lower', 'No transpose', 'Unit', JB,
     $                N-JB, ONE, A, IA, JA, DESCA, A, IA, JN+1, DESCA )
         IF( JB+1.LE.M ) THEN
*           Update trailing submatrix.
            CALL PDGEMM( 'No transpose', 'No transpose', M-JB, N-JB,
JB,
     $                   -ONE, A, IN+1, JA, DESCA, A, IA, JN+1, DESCA,
     $                   ONE, A, IN+1, JN+1, DESCA )
         END IF
      END IF
*     Loop over the remaining blocks of columns.
      DO 10 J = JN+1, JA+MN-1, DESCA( NB_ )
         JB = MIN( MN-J+JA, DESCA( NB_ ) )
         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 )
         IF( INFO.EQ.0 .AND. IINFO.GT.0 )
     $      INFO = IINFO + J - JA
*        Apply interchanges to columns JA:J-JA.
         CALL PDLASWP( 'Forward', 'Rowwise', J-JA, A, IA, JA, DESCA,
     $                 I, I+JB-1, IPIV )
         IF( J-JA+JB+1.LE.N ) THEN
*           Apply interchanges to columns J+JB:JA+N-1.
            CALL PDLASWP( 'Forward', 'Rowwise', N-J-JB+JA, A, IA, J+JB,
     $                    DESCA, I, I+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
      IF( INFO.EQ.0 )
     $  INFO = MN + 1
      CALL IGAMN2D( ICTXT, 'Rowwise', ' ', 1, 1, INFO, 1, IDUM1, IDUM2,
     $              -1, -1, MYCOL )
      IF( INFO.EQ.MN+1 )
     $  INFO = 0
      CALL PB_TOPSET( ICTXT, 'Broadcast', 'Rowwise', ROWBTOP )
      CALL PB_TOPSET( ICTXT, 'Broadcast', 'Columnwise', COLBTOP )
      CALL PB_TOPSET( ICTXT, 'Combine', 'Columnwise', COLCTOP )
      RETURN
      END
```

Figure 1.4: ScaLAPACK variant of LU factorization (FORTRAN 77 coding makes the code overly verbose due to lack of object oriented capabilities that could have hidden much of the complexity.)

arguably more natural than LAPACK's matrix formulation. The mathematical formulas that describe algorithms are usually more complex if only matrices are used, as opposed to mixed vector-matrix notation.

1.7 Implementation on Distributed Memory Machines

Traditional design focus for Massively Parallel Processing (MPP) systems was the very high end of performance. In the early nineties, the Symmetric Multi-Processing (SMP) systems of various workstation manufacturers as well as the IBM SP series, which targeted the lower and medium market segments, gained great popularity. Their price/performance ratios were better due to the missing overhead in the design for support of the very large configurations and due to cost advantages of the larger production numbers. Due to the vertical integration of performance it was no longer economically feasible to produce and focus on the highest end of computing power alone. The design focus for new systems shifted to the market of medium performance systems.

The acceptance of MPP systems not only for engineering applications but also for new commercial applications especially for database applications emphasized different criteria for market success such as the stability of the system, continuity of the manufacturer and price/performance. Success in commercial environments became a new important requirement for a successful supercomputer business towards the end of the nineties. Due to these factors and the consolidation in the number of vendors in the market, hierarchical systems built with components designed for the broader commercial market did replace homogeneous systems at the very high end of performance. The marketplace adopted clusters of SMPs readily, while academic research focused on clusters of workstations and PCs.

At the end of the nineties clusters were common in academia but mostly as research objects and not primarily as general purpose computing platforms for applications. Most of these clusters were of comparable small scale and as a result the November 1999 edition of the TOP500 [28] listed only seven cluster systems. This changed dramatically as industrial and commercial customers started deploying clusters as soon as applications with less stringent communication requirements permitted them to take advantage of the better price/performance ratio – roughly an order of magnitude of commodity based clusters. At the same time, all major vendors in the HPC market started selling this type of cluster to their customer base. In November 2004, clusters were the dominant architectures in the TOP500 with 294 systems at all levels of performance. Companies such as IBM and Hewlett-Packard sell the majority of these clusters and a large number of them are installed at commercial and industrial customers.

In the early 2000s, clusters build with off-the-shelf components gained more and more attention not only as academic research object but also computing platforms with end-users of HPC computing systems. By 2004, these groups of clusters represent the majority of new systems on the TOP500 in a broad range of application areas. One major consequence of this trend was the rapid rise in the utilization of Intel processors in HPC systems. While virtually absent in the high end at the beginning of the decade, Intel processors are now used in the majority of HPC systems. Clusters in the nineties were mostly self-made system designed and built by small groups of dedicated scientist or application experts. This changed rapidly as soon as the market for clusters based on PC technology matured. Nowadays, the large majority of TOP500-class clusters are manufactured and integrated by either a few traditional large HPC manufacturers such as IBM or Hewlett-Packard or numerous small, specialized integrators of such systems.

In addition, there still is generally a large difference in the usage of clusters and their more integrated counterparts: clusters are mostly used for *capacity computing* while the integrated machines primarily are used for *capability computing*. The largest supercomputers are used for capability or turnaround computing where the maximum processing power is applied to a single problem. The goal is to solve a larger problem, or to solve a single problem in a shorter period of time. Capability computing enables the solution of problems that cannot otherwise be solved in a reasonable period of time (for example, by moving from a 2D to a 3D simulation, using finer grids, or using more realistic models). Capability computing also enables the solution of problems with real-time constraints (e.g., predicting weather). The main figure of merit is time to solution. Smaller or cheaper systems are used for capacity computing, where smaller problems are solved. Capacity computing can be used to enable parametric studies or to explore design alternatives; it is often needed to prepare for more expensive runs on capability systems. Capacity systems will often run several jobs simultaneously. The main figure of merit is sustained performance per unit cost. Traditionally, vendors of large supercomputer systems have learned to provide for the capacity mode of operation as the precious resources of their systems were required to be used as effectively as possible. By contrast, Beowulf clusters are mostly operated through the Linux operating system (a small minority using Microsoft Windows) where these operating systems either lack the tools or these tools are relatively immature to use a cluster effectively for capability computing. However, as clusters become on average both larger and more stable in terms of continuous operation, there is a trend to use them also as computational capability servers.

There are a number of choices of communication networks available in clusters. Of course 100 Mb/s

Ethernet or Gigabit Ethernet is always possible, which is attractive for economic reasons, but it has the drawback of a high latency ($\sim 100\mu$s) – the time it takes to send the shortest message. Alternatively, there are, for instance, networks that operate from user space, like Myrinet, Infiniband. The network speeds as shown by these networks are more or less on par with some integrated parallel systems. So, possibly apart from the speed of the processors and of the software that is provided by the vendors of traditional integrated supercomputers, the distinction between clusters and the class of custom capability machines becomes rather small and will, without a doubt, decrease further in the coming years. And the advances of the Ethernet standard into the 100 Gb/s territory with latencies well below 10 μs make it even more so.

LAPACK was designed to be highly efficient on vector processors, high-performance "superscalar" workstations, and shared-memory multiprocessors. LAPACK can also be used satisfactorily on all types of scalar machines (PCs, workstations, and mainframes). However, LAPACK in its present form is less likely to give good performance on other types of parallel architectures – for example, massively parallel Single Instruction Multiple Data (SIMD) machines, or Multiple Instruction Multiple Data (MIMD) distributed-memory machines. The ScaLAPACK effort was intended to adapt LAPACK to these new architectures.

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 distributed-memory versions of the Level-2 and Level-3 BLAS [29], and a set of Basic Linear Algebra Communication Subprograms (BLACS) [30] for communication tasks that arise frequently in parallel linear algebra computations. In the ScaLAPACK routines, all interprocessor communication occurs within the distributed BLAS and the BLACS, so the source code of the top software layer of ScaLAPACK looks very similar to that of LAPACK. The similarity may be observed by comparing Figures 1.3 and 1.4.

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 [29], 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.

It was our aim that the PBLAS would provide a distributed memory standard, just as the BLAS 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 just a small set of routines to be optimized. The acceptance of the PBLAS requires reasonable compromises between competing goals of functionality and simplicity.

The PBLAS operate on matrices distributed in a two-dimensional block cyclic layout. Because 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:

- The descriptor type
- The BLACS context (a virtual space for messages that is created to avoid collisions between logically distinct messages)
- The number of rows in the distributed matrix
- The number of columns in the distributed matrix
- The row block size
- The column block size
- The process row over which the first row of the matrix is distributed
- The process column over which the first column of the matrix is distributed
- 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 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 is simply 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.

Using message passing and scalable algorithms from the ScaLAPACK library makes it possible to factor matrices of arbitrarily increasing size, given machines with more processors. By design, the library computes more than it communicates, so for the most part, data stay locally for processing and travels only occasionally across the interconnect network.

But the number and types of messages exchanged between processors can sometimes be hard to manage. The context associated with every distributed matrix lets implementations use separate "universes" for 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, the individual BLACS context entries must be equal. In other words, the PBLAS do not perform "inter-context" operations.

In the performance sense, ScaLAPACK did to LAPACK what LAPACK did to LINPACK: it broadened the range of hardware where LU factorization (and other codes) could run efficiently. In terms of code elegance, the ScaLAPACK's changes were much more drastic: the same mathematical operation now required large amounts of tedious work. Both the users and the library writers were now forced into explicitly controlling data storage intricacies, because data locality became paramount for performance. The victim was the readability of the code, despite efforts to modularize the code according to the best software engineering practices of the day.

1.8 Shared Memory Implementation

The advent of multi-core processorss brought about a fundamental shift in the way software is produced even though comparisons have been brought up with the established coding techniques from SMPs. Rather elaborating the differences we will focus on how most of software had to be adjusted for SMPs with a special focus on dense linear algebra. The good news is that LAPACK's LU factorization runs on a multi-core system and can even deliver a modest increase of performance if multi-threaded BLAS are used. In technical terms, this is the *Block Synchronous Processing* (BSP) model [31] model of parallel computation: each call to BLAS (from a single main thread) forks a suitable number of threads (parallel units of executions that share memory and are often scheduled by the operating system), which perform the work on each core and then join the main thread of computation. This is also called a *fork-join* model and it implies a synchronization point at each join operation.

The bad news is that the LAPACK's fork-join algorithm gravely impairs scalability even on small multi-core computers that do not have the memory systems available in SMP systems. The inherent scalability flaw is the heavy synchronization in the fork-join model:only a single thread is allowed to perform the significant computation that occupies the critical section of the code, leaving other threads idle. That results in lock-step execution: all threads have to wait for the slowest one among them. It also prevents hiding of inherently sequential portions of the code behind parallel ones. In other words, the threads are forced to perform the same operation on different data. If there is not enough data for some threads, they will have to stay idle and wait for the rest of the threads that perform useful work on their data. Clearly, another version of the LU algorithm is needed such that would allow threads to stay busy all the time by possibly making them perform different operations during some portion of the execution.

The multithreaded version of the algorithm recognizes the existence of a so-called critical path in the algorithm: a portion of the code whose execution depends on previous calculations and can block

```
void
SMP_dgetrf(int n, double *a, int lda, int *ipiv, int NB,
     int tid, int tsize, int *pready, ptm *mtx, ptc *cnd) {
 int pcnt, pfctr, ufrom, uto, ifrom, p;
 double *pa = a, *pl, *pf, *lp;

 pcnt = n / NB; /* number of panels */

 /* first panel that should be factored by this thread after
  * the very first panel (number 0) gets factored */
 pfctr = tid + (tid ? 0 : tsize);

 /* this is a pointer to the last panel */
 lp = a + (size_t)(n - NB) * (size_t)lda;

 /* for each panel (that is used as source of updates) */
 for (ufrom = 0; ufrom < pcnt;
    ufrom++, pa += NB * (size_t)(lda + 1)){
  p = ufrom * NB; /* column number */

  /* if the panel to be used for updates has not been
   * factored yet; 'ipiv' does not be consulted, but it is
   * to possibly avoid accesses to 'pready' */
  if (! ipiv(p + NB - 1) || ! pready(ufrom)) {
   /* if this is this thread's panel */
   if (ufrom % tsize == tid) {
    pfactor(n-p, NB, pa, lda, ipiv+p, pready, ufrom, mtx,
cnd);

   /* if this is not the last panel */
   } else if (ufrom < pcnt - 1) {
    LOCK( mtx );
    while (! pready(ufrom)) { WAIT( cnd, mtx ); }
    UNLOCK( mtx );
   }
  }
  /* for each panel to be updated */
  for (uto = first_panel_to_update( ufrom, tid, tsize );
     uto < pcnt;   uto += tsize) {
   /* if there are still panels to factor by this thread and
    * preceding panel has been factored; test to 'ipiv'
   * could be skipped but is in there to decrease number
    * of accesses to 'pready' */
   if (pfctr < pcnt && ipiv(pfctr*NB-1) &&
                   pready(pfctr-1)) {
    /* for each panel that has to (still) update panel
     * 'pfctr' */
    for (ifrom = ufrom + (uto>pfctr ? 1 : 0); ifrom < pfctr;
                     ifrom++) {
     p = ifrom * NB;
     pl = a + (size_t)p * (size_t)(lda + 1);
     pf = pl + (size_t)(pfctr - ifrom) * (size_t)NB * lda;
     pupdate( n - p, NB, pl, pf, lda, p, ipiv, lp );
    }
    p = pfctr * NB;
    pl = a + (size_t)p * (size_t)(lda + 1);
     pfactor(n-p, NB, pl, lda, ipiv+p, pready, pfctr, mtx,
cnd);
    pfctr += tsize; /* move to this thread's next panel */
   }
   /* if panel 'uto' hasn't been factored (if it was, it
    * certainly has been updated, so no update is
    * necessary) */
   if (uto > pfctr || ! ipiv(uto * NB)) {
    p = ufrom * NB;
    pf = pa + (uto - ufrom) * (size_t)NB * lda;
    pupdate( n - p, NB, pa, pf, lda, p, ipiv, lp );
   }
  }
 }
}
```

Figure 1.5: Factorization for multi-threaded execution (C code.)

the progress of the algorithm. The LAPACK's LU does not treat this critical portion of the code in any special way: the DGETF2 subroutine is called by a single thread and doesn't allow much parallelization even at the BLAS level. While one thread calls this routine, the other ones wait idly. And since the performance of DGETF2 is bound by memory bandwidth (rather than processor speed), this bottleneck will exacerbate scalability problems as systems with more cores are introduced.

The multithreaded version of the algorithm attacks this problem head-on by introducing the notion of *look-ahead*: calculating things ahead of time to avoid potential stagnation in the progress of the computations. This of course requires additional synchronization and bookkeeping not present in the previous versions – a trade-off between code complexity and performance. Another aspect of the multithreaded code is the use of recursion in the panel factorization. It turns out that the use of recursion can give even greater performance benefits for tall panel matrices than it does for the square ones [32].

The algorithm is the same for each thread (the SIMD paradigm), and the matrix data is partitioned among threads in a cyclic manner using panels with NB columns in each panel (except maybe the last). The NB parameter corresponds to the blocking parameter NB of LAPACK. The difference is the logical assignment of panels (blocks of columns) to threads. Physically, all panels are equally accessible, because the code operates in a shared memory regime. The benefits of blocking in a thread are the same as they were in LAPACK: better cache reuse and less stress on the memory bus. Assigning a portion of the matrix to a thread seems an artificial requirement at first, but it simplifies the code and the bookkeeping data structures; most importantly, it provides better memory affinity. It turns out that multi-core chips are not symmetric in terms of memory access bandwidth, so minimizing the number of reassignments of memory pages to cores directly benefits performance.

The standard components of LU factorization are represented by the pfactor() and pupdate() functions in Figure 1.5. As one might expect, the former factors a panel, whereas the latter updates a panel using one of the previously factored panels.

The main loop makes each thread iterate over each panel in turn. If necessary, the panel is factored by the owner thread while other threads wait (if they happen to need this panel for their updates).

The look-ahead logic is inside the nested loop (prefaced by the comment for each panel to be updated) that replaces DGEMM or PDGEMM from previous algorithms. Before each thread updates one of its panels, it checks whether its already feasible to factor its first unfactored panel. This minimizes the number of times the threads have to wait because each thread constantly attempts to eliminate the potential bottleneck.

As was the case for ScaLAPACK, the multithreaded version detracts from the inherent elegance of the LAPACKs version. Also in the same spirit, performance is the main culprit: LAPACKs code will not run efficiently on machines with ever-increasing numbers of cores. Explicit control of execution threads at the LAPACK level rather than the BLAS level is critical: parallelism cannot be encapsulated in a library call. The only good news is that the code is not as complicated as ScaLAPACKs, and efficient BLAS can still be put to a good use.

1.9 Multicore Implementations

The multicore processors do not resemble the SMP systems of the past, nor do they resemble distributed memory systems. In comparison to SMPs, multicores are much more starved for memory due to the fast increase in the number of cores, which is not followed by a proportional increase in bandwidth. Owing to that, data access locality is of much higher importance in case of multicores. At the same time, they do follow to a large extent the memory model where the main memory serves as a central (not distributed) repository for data. For those reasons, the best performing algorithms or multicores happen to be parallel versions of what used to be know as “out of core” algorithms (algorithms developed in the past for situations where data does not fit in the main memory and has to be explicitly moved between the memory and the disc).

In dense linear algebra, the Tile Algorithms are direct descendants of ”out of core” algorithms. The Tile Algorithms are based on the idea of processing the matrix by square submatrices, referred to as tiles, of relatively small size. This makes the operation efficient in terms of cache and TLB use. The Cholesky factorization lends itself readily to tile formulation, however the same is not true for the LU and QR factorizations. The tile algorithms for them are constructed by factorizing the diagonal tile first and then incrementally updating the factorization using the entries below the diagonal tile. This is a very well known concept that dates back to the work of Gauss. The idea was initially used to build “out-of-core” algorithms and recently rediscovered as a very efficient method for implementing linear algebra operations on multicore processors. It is crucial to note that the technique of processing the matrix by square tiles yields satisfactory performance only when accompanied by data organization based on square tiles. The layout is referred to as Square Block layout or, simply, Tile Layout.

For parallel execution those algorithms can be scheduled either statically or dynamically. For static execution (Figure 1.6) the work for each core is predetermined and each core follows the cycle: check task dependencies (and wait if necessary), perform a task, update dependencies, transition to the next task (using a static transition function). For regular algorithms, such as dense matrix factorizations, static scheduling is straightforward and very robust.

An alternative approach, which emphasizes the ease of development, is based on writing a serial algorithm and the use of a dynamic scheduler, which traverses the code and queues tasks for parallel execution, while automatically keeping track of data dependencies (Figure 1.7). This approach relies on the availability of such a scheduler, which is not trivial to develop, but offers multiple advantages, such as pipelining/streaming of different stages of the computation (e.g. factorization and solve).

1.10 Error Analysis and Operation Count

The key aspect of all of the implementations presented in this section is their numerical properties.

```
#define A(m,n) &((PLASMA_Complex64_t*)A.mat)(A.bsiz*(m)+A.bsiz*A.lmt*(n))
#define L(m,n) &((PLASMA_Complex64_t*)L.mat)(L.bsiz*(m)+L.bsiz*L.lmt*(n))
#define IPIV(m,n) &IPIV(A.nb*(m)+A.nb*A.lmt*(n))
void plasma_pzgetrf(plasma_context_t *plasma) {
 PLASMA_desc A, L;
 int *IPIV,k, m, n, next_k, next_m, next_n, iinfo; PLASMA_Complex64_t *work;
 plasma_unpack_args_3(A, L, IPIV);
 work = (PLASMA_Complex64_t*)plasma_private_alloc(plasma, L.mb*L.nb, L.dtyp);
 ss_init(A.mt, A.nt, -1);
 k = 0; n = PLASMA_RANK;
 while (n >= A.nt) {k++; n = n-A.nt+k;}
 m = k;

 while (k < min(A.mt, A.nt) && n < A.nt) {
  next_n = n; next_m = m; next_k = k;

  next_m++;
  if (next_m == A.mt) {
   next_n += PLASMA_SIZE;
   while (next_n >= A.nt && next_k < min(A.mt, A.nt)) {
    next_k++; next_n = next_n-A.nt+next_k;} next_m = next_k;
   }
   if (n == k) {
    if (m == k) {
     ss_cond_wait(k, k, k-1);
     CORE_zgetrf( k == A.mt-1 ? A.m-k*A.nb : A.nb, k == A.nt-1 ? A.n-k*A.nb : A.nb, L.mb,
       A(k, k), A.nb, IPIV(k, k), &iinfo);
     if (PLASMA_INFO == 0 && iinfo > 0 && m == A.mt-1)
      PLASMA_INFO = iinfo + A.nb*k;
     ss_cond_set(k, k, k);
    } else {
     ss_cond_wait(m, k, k-1);
     CORE_ztstrf( m == A.mt-1 ? A.m-m*A.nb : A.nb, k == A.nt-1 ? A.n-k*A.nb : A.nb, L.mb, A.nb,
      A(k, k), A.nb, A(m, k), A.nb, L(m, k), L.mb, IPIV(m, k), work, L.nb, &iinfo);
     if (PLASMA_INFO == 0 && iinfo > 0 && m == A.mt-1)
      PLASMA_INFO = iinfo + A.nb*k;
     ss_cond_set(m, k, k);
    }
   } else {
    if (m == k) {
     ss_cond_wait(k, k, k);
     ss_cond_wait(k, n, k-1);
     CORE_zgessm( k == A.mt-1 ? A.m-k*A.nb : A.nb, n == A.nt-1 ? A.n-n*A.nb : A.nb, A.nb, L.mb,
       IPIV(k, k), A(k, k), A.nb, A(k, n), A.nb);
    } else {
     ss_cond_wait(m, k, k);
     ss_cond_wait(m, n, k-1);
     CORE_zssssm( A.nb, m == A.mt-1 ? A.m-m*A.nb : A.nb, n == A.nt-1 ? A.n-n*A.nb : A.nb, L.mb, A.nb,
      A(k, n), A.nb, A(m, n), A.nb, L(m, k), L.mb, A(m, k), A.nb, IPIV(m, k));
     ss_cond_set(m, n, k);
    }
   }
  n = next_n; m = next_m; k = next_k;
 }
 plasma_private_free(plasma, work);
 ss_finalize();
}
```

Figure 1.6: Factorization for multicore execution using the SMPD programming model with static scheduling of work (C code).

```
#define A(m,n) &((PLASMA_Complex64_t*)A.mat)[A.bsiz*(m)+A.bsiz*A.lmt*(n)]
#define L(m,n) &((PLASMA_Complex64_t*)L.mat)[L.bsiz*(m)+L.bsiz*L.lmt*(n)]
#define IPIV(m,n) &IPIV[A.nb*(m)+A.nb*A.lmt*(n)]
void plasma_pdgetrf_quark(PLASMA_desc A, PLASMA_desc L, int *IPIV) {
   plasma_context_t *plasma;
   Quark_Task_Flags task_flags = Quark_Task_Flags_Initializer;
   int k, m, n;
   int ldak, ldam;
   int tempkm, tempkn, tempmm, tempnn;
   int ib = PLASMA_IB;
   plasma = plasma_context_self();
   for (k = 0; k < min(A.mt, A.nt); k++) {
      tempkm = k == A.mt-1 ? A.m-k*A.mb : A.mb;
      tempkn = k == A.nt-1 ? A.n-k*A.nb : A.nb;
      ldak = BLKLDD(A, k);
      QUARK_Insert_task( plasma->quark, CORE_dgetrf_incpiv, &task_flags,
         sizeof(int), tempkm, VALUE,
         sizeof(int), tempkn, VALUE,
         sizeof(int), ib, VALUE,
         sizeof(int), L.nb, VALUE,
         sizeof(double)*A.mb*A.nb, A(k, k), INOUT | LOCALITY,
         sizeof(int), ldak, VALUE,
         sizeof(int)*A.mb, IPIV(k, k), OUTPUT,
         sizeof(int), k == A.mt-1, VALUE,
         sizeof(int), A.nb*k, VALUE,
         0 );
      for (n = k+1; n < A.nt; n++) {
         tempnn = n == A.nt-1 ? A.n-n*A.nb : A.nb;
         QUARK_Insert_Task( plasma->quark, CORE_dgessm, &task_flags,
            sizeof(int),      tempkm, VALUE,
            sizeof(int),      tempnn, VALUE,
            sizeof(int),      tempkm, VALUE,
            sizeof(int),      ib, VALUE,
            sizeof(int),      L.nb, VALUE,
            sizeof(int)*A.mb, IPIV(k, k), INPUT,
            sizeof(double)*A.mb*A.nb, A(k, k), NODEP,
            sizeof(int),      ldak, VALUE,
            sizeof(double)*A.mb*A.nb, A(k, n), INOUT | LOCALITY,
            sizeof(int),      ldak, VALUE,
            0 );
      }
      for (m = k+1; m < A.mt; m++) {
         tempmm = m == A.mt-1 ? A.m-m*A.mb : A.mb;
         ldam = BLKLDD(A, m);
         QUARK_Insert_Task( plasma->quark, CORE_dtstrf, &task_flags,
            sizeof(int),      tempmm, VALUE,
            sizeof(int),      tempkn, VALUE,
            sizeof(int),      ib, VALUE,
            sizeof(int),      L.nb, VALUE,
            sizeof(double)*A.mb*A.nb, A(k, k), INOUT | LOCALITY,
            sizeof(int),      ldak, VALUE,
            sizeof(double)*A.mb*A.nb, A(m, k), INOUT,
            sizeof(int),      ldam, VALUE,
            sizeof(int)*L.mb*L.nb, L(m, k), OUTPUT,
            sizeof(int),      L.mb, VALUE,
            sizeof(int)*A.mb, IPIV(m, k), OUTPUT,
            sizeof(double)*L.mb*L.nb, NULL, SCRATCH,
            sizeof(int),      m == A.mt-1, VALUE,
            sizeof(int),      A.nb*k, VALUE,
            0 );
         for (n = k+1; n < A.nt; n++) {
            tempnn = n == A.nt-1 ? A.n-n*A.nb : A.nb;
            QUARK_Insert_Task( plasma->quark, CORE_dssssm, &task_flags,
               sizeof(int),      A.nb, VALUE,
               sizeof(int),      tempnn, VALUE,
               sizeof(int),      tempmm, VALUE,
               sizeof(int),      tempnn, VALUE,
               sizeof(int),      A.nb, VALUE,
               sizeof(int),      ib, VALUE,
               sizeof(int),      L.nb, VALUE,
               sizeof(double)*A.mb*A.nb, A(k, n), INOUT | LOCALITY,
               sizeof(int),      ldak, VALUE,
               sizeof(double)*A.mb*A.nb, A(m, n), INOUT,
               sizeof(int),      ldam, VALUE,
               sizeof(int),      L(m, k), INPUT,
               sizeof(int),      L.mb, VALUE,
               sizeof(double)*A.mb*A.nb, A(m, k), INPUT,
               sizeof(int),      ldam, VALUE,
               sizeof(int)*A.mb, IPIV(m, k), INPUT,
               0 );
         }
      }
   }
}
```

Figure 1.7: Factorization for multicore execution using dynamic task scheduling (C code).

It is acceptable to forgo elegance in order to gain performance. But numerical stability is of vital importance and cannot be sacrificed, because it is an inherent part of the algorithms correctness. While these are serious considerations, there is some consolation to follow. It may be surprising to some readers that all of the algorithms presented are the same, even though its virtually impossible to make each excerpt of code produce exactly the same output for exactly the same inputs. The fundamental reason for this are the vagaries of the floating-point arithmetic in finite precision as it is implemented in virtually all hardware. In essence, only a slight change in the order in which the floating-point operations are performed causes a change in the result: the change is on the order of the, so called, machine precision. Machine precision comes from the number of decimal digits represented in the floating-point format: for double precision there are 15 digits and so the machine precision is about 10^{-15}. LINPACK and LAPACK perform the operations in different order because the latter merges the updates into a single call to BLAS. And even though ScaLAPACK merges the updates in a similar fashion as LAPACK does, the former performs its operations only on the local portion of the matrix whereas the latter treats the matrix as a single piece quantity. In other words, when LAPACK makes a single update operation, ScaLAPACK could make as many as there are processors involved in the computation.

When it comes to repeatability of results, the vagaries of floating-point representation may be captured in a rigorous way by error bounds. One way of expressing the numerical robustness of the previous algorithms is with the following formula:

$$\frac{\|A\hat{x}-b\|}{\|A\|} \le \|x-\hat{x}\| \le \|A^{-1}\|\|A\hat{x}-b\| \tag{1.6}$$

where the error vector $x-\hat{x}$ is the difference between the computed solution $\hat{x}$ and the correct solution x, and $A\hat{x}-b$ is a so-called “residual”. The previous formula basically says that the size of the error (the parallel bars surrounding a value indicate a norm – a measure of absolute size) is as small as warranted by the quality of the matrix A. Therefore, if the matrix is close to being singular in numerical sense (some entries are small with respect to machine precision and the condition number of the matrix and so they might be considered to be zero) the algorithms will not give an accurate answer. But otherwise, a relatively good quality of the result may be expected.

Another feature that is common to all the versions presented is the operation count: they all perform $O(2/3n^3)$ floating-point multiplications and/or additions. The order of these operations is what differentiates them. There exist algorithms that increase the amount of floating-point work to save on memory traffic or network transfers (especially for distributed-memory parallel algorithms.) But because the algorithms shown in this chapter have the same operation count, it is valid to compare them for performance. The computational rate (number of floating-point operations per second) may be used instead of the time taken to solve the problem, provided that the matrix size is the same. But comparing computational rates is sometimes better because it allows a comparison of algorithms when the matrix sizes differ. For example, a sequential algorithm on a single processor can be directly compared with a parallel one working on a large cluster on a much bigger matrix.

1.11 Concluding Remarks and Future Directions

In this chapter we have looked at the evolution of the design of a simple but important algorithm in computational science. The changes over the past 30 years have been necessary to follow the lead of the advances in computer architectures. In some cases these changes have been simple, such as interchanging loops. In other cases, they have been as complex as the introduction of recursion and look-ahead computations. In each case, however, a code's ability to efficiently utilize the memory hierarchy is the key to high performance on a single processor as well as shared and distributed memory systems.

The essence of the problem is the dramatic increase in complexity that software developers have had to confront, and still do. Dual-core machines are already common, and the number of cores is expected to roughly double with each processor generation. But contrary to the assumptions of the old model, programmers will not be able to consider these cores independently (i.e., multi-core is not “the new SMP”) because they share on-chip resources in ways that separate processors do not. This situation is made even more complicated by the other non-standard components that future architectures are expected to deploy, including mixing different types of cores, hardware accelerators, and memory systems.

When processor clock speeds flatlined in 2004, after more than fifteen years of exponential increases, the era of routine and near automatic performance improvements that the HPC application community had previously enjoyed came to an abrupt end. The air of crisis that followed in the wake of this new regime continues to hang over computational science. To develop software that will perform well on petascale systems with thousands of nodes and millions of cores, the list of major challenges that must now be confronted is formidable:

- Dramatic escalation in the costs of intrasystem communication between processors and/or levels of memory hierarchy;
- Increased hybridization of processor architectures (mixing CPUs, GPUs, etc.), in varying and unexpected design combinations;
- High levels of parallelism and more complex constraints means that cooperating processes must be dynamically and unpredictably scheduled for asynchronous execution;
- Software will not run at scale without much better resilience to faults and far more robustness; and
- New levels of self-adaptivity will be required to enable software to modulate process speed in order to satisfy limited energy budgets.

After the industry-wide move from single to multi-core systems, dominant mainstream computer architecture is now undergoing a second major evolution: from homogeneous to heterogeneous platforms. With increased frequency, the new systems are called Hybrid Multicores (HMCs). Todays breed of HMCs simply feature a multi-core processor and a high end GPU. In the future, the multi-core vendors are planning integration of GPU-like technology directly into the multi-core chip. From the programmer perspective, this might alleviate the problem of dealing with two separate memory address spaces: one attached to the multi-core and one attached to the GPU. If such integration is realized and the performance levels are satisfactory, then such hybrid computing device could be the prevalent hardware design. Faced with a choice of either having an external GPU or an integrated GPU-like device, the programmer would have to choose the more productive solution give the problem at hand.

In a nutshell, the high performance computing (HPC) community will soon be faced with machines supporting heterogeneities in all hardware aspects – processing elements of multiple types with different ISAs, multiple memory components with variable data transport interfaces, general and specific accelerators for various purpose, power control system infrastructure integrated throughout – and all in concurrent, simultaneous action and interaction.

Finally, the proliferation of widely divergent design ideas shows that the question of how to best combine all these new resources and components is largely unsettled. When combined, these changes produce a picture of a future in which programmers will have to overcome software design problems vastly more complex and challenging than those in the past in order to take advantage of the much higher degrees of concurrency and greater computing power that new architectures will offer. The current trends in software do not address such complexities. The message passing paradigm epitomized by the MPI (Message Passing Interface) standard quickly leads to management issues if every processing core corresponds to a single MPI process. Only a hierarchical approach could possibly address todays machine that features man hundreds of thousands of computational cores. One such approach is a mix of MPI and OpenMP. The former connects the multi-core nodes while the latter commands the computation inside each node. Such a mix could potentially reduce the programming complexity by one or two orders of magnitude in the number cores but the issue of the attached accelerator (either a GPU or GPU-like device) is still not addressed within a single programming framework. Existing PGAS (Partitioned Global Address Space) languages such as Co-Array Fortran, Titanium, and UPC have never been designed to address the hardware hybridization phenomenon. Even the new languages of the breed such as Chapel, Fortress, and X10 could potentially face the challenge of redesign to fit in the changing hardware landscape. It is still too early to tell what would come out of the current initiatives to retrofit the mainstay languages of HPC, C and Fortran, with new facilities for handling hybrid computers. At this point in time it is hard to which approach will prove to have a lasting power.

So the bad news is that none of the presented codes will work efficiently someday. The good news is that we have learned various ways to mold the original simple rendition of the algorithm to meet the ever increasing challenges of hardware designs.

Bibliography

[1] R. Aymar, V. Chuyanov, M. Huguet, and Y. Shimomura. Overview of ITER-FEAT - the future international burning plasma experiment. *Nuclear Fusion*, 41(10), 2001.

[2] E.F. Jaeger, L.A. Berry, E. DAzevedo, D.B. Batchelor, M.D. Carter MD, K.F. White, and H. Weitzner. Advances in full-wave modeling of radio frequency heated multidimensional plasmas. *Physics of Plasmas*, 9(5):1873–1881, 2002.

[3] E.F. Jaeger, L.A. Berry, J.R. Myra, D.B. Batchelor, E. DAzevedo, P.T. Bonoli, C.K. Philips, D.N. Smithe, D.A. DIppolito, M.D. Carter, R.J. Dumont, J.C. Wright, and R.W. Harvey. Sheared poloidal flow driven by mode conversion in Tokamak plasmas. *Phys. Rev. Lett.*, 90(19), 2003.

[4] E.F. Jaeger, R.W. Harvey, L.A. Berry, J.R. Myra, R.J. Dumont, C.K. Philips, D.N. Smithe, R.F. Barrett, D.B. Batchelor, P.T. Bonoli, M.D. Carter, E.F. Dazevedo, D.A. Dippolito, R.D. Moore, and J.C. Wright. Global-wave solutions with self-consistent velocity distributions in ion cyclotron heated plasmas. *Nuclear Fusion*, 46(7):S397S408, 2006.

[5] M. Fuchs, A. M. Shor, and N. Rösch. The hydration of the uranyl dication. incorporation of solvent effects in parallel density functional calculations with the program PARAGAUSS. *Int. J. Quantum Chem.*, 86:487–501, 2002.

[6] Th. Belling, Th. Grauschopf, S. Krüger, F. Nörtemann, M. Staufer, M. Mayer, V. A. Nasluzov, U. Birkenheuer, and N. Rösch. ParaGauss: A density functional approach to quantum chemistry on parallel computers. In F. Keil, M. Mackens, H. Voß, and J. Werther (Hrsg.), editors, *Scientific Computing in Chemical Engineering II*, volume 1, pages 66–73. Springer, Heidelberg, 1999.

[7] R. F. Barrett, T. H. F. Chan, E. F. D'Azevedo, E. F. Jaeger, K. Wong, and R. Y. Wong. Complex version of high performance computing LINPACK benchmark (HPL). *Concurrency and Computation: Practice and Experience*, 22(5):573–587, April 10 2010.

[8] L. Suzan Blackford, J. Choi, Andy Cleary, Eduardo F. D'Azevedo, James W. Demmel, Inderjit S. Dhillon, Jack J. Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clint Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[9] J. Choi, Jack J. Dongarra, Susan Ostrouchov, Antoine Petitet, David W. Walker, and R. Clint Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5:173–184, 1996.

[10] A. M. Shor, E. A. Ivanova Shor, V. A. Nasluzov, G. N. Vayssilov, and N. Rösch. First hybrid embedding scheme for polar covalent materials using an extended border region to minimize boundary effects on the quantum region. *J. Chem. Theor. Comput.*, 3:2290–2300, 2008. DOI: 10.1021/ct700159k.

[11] A. V. Matveev, V. A. Nasluzov, and N. Rösch. Linear response formalism for the Douglas-Kross-Hess approach to the Dirac-Kohn-Sham problem: First- and second-order energy derivatives. *Int. J. Quantum Chem.*, 107:3236–3249, 2007. DOI: 10.1002/qua.21501.

[12] N. Rösch, A. V. Matveev, V. A. Nasluzov, K. M. Neyman, Moskaleva L. V, and S. Krüger. Quantum chemistry with the douglas-kroll-hess approach to relativistic density functional theory: Efficient methods for molecules and materials. In P. Schwerdtfeger (Hrsg.), editor, *Relativistic Electronic*

Structure Theory - Applications, Theoretical and Computational Chemistry Series, volume 14, pages 656–722. Elsevier, Amsterdam, 2004.

[13] T. Kerdcharoen, U. Birkenheuer, S. Krüger, A. Woiterski, and N. Rösch. Implementation of a quantum mechanics/molecular mechanics approach in the parallel density functional program PARAGAUSS and applications to model copper thiolate clusters. *Theor. Chem. Acc.*, 109:285–297, 2003.

[14] Takeshi Yanai, Haruyuki Nakano, Takahito Nakajima, Takao Tsuneda, So Hirata, Yukio Kawashima, Yoshihide Nakao, Muneaki Kamiya, Hideo Sekino, and Kimihiko Hirao. UTChem - a program for *ab initio* quantum chemistry. In P.M.A. Sloot et al., editor, *LNCS: ICCS 2003*, volume 2660, pages 84–95. Springer-Verlag, Berlin Heidelberg, 2003.

[15] Chi-Chung Lama, Thomas Rauber, Gerald Baumgartner, Daniel Cociorva, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. *Computer Languages, Systems & Structures*, 37(2):63–75, 2011.

[16] G. W. Stewart. The decompositional approach to matrix computation. *Computing in Science & Engineering*, 2(1):50–59, Jan/Feb 2000. ISSN: 1521-9615; DOI 10.1109/5992.814658.

[17] Cleve B. Moler. Matrix computations with fortran and paging. *Communications of the ACM*, 15(4):268–270, 1972.

[18] Jack J. Dongarra, Fred G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, January 1984.

[19] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[20] Jack J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, March 1988.

[21] Jack J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, March 1988.

[22] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and S. Hammarling. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, March 1990.

[23] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:18–28, March 1990.

[24] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, 1979. ISBN 0-89871-172-X.

[25] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 19 1965.

[26] E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[27] E. Anderson and J. Dongarra. Lapack working note 19: Evaluating block algorithm variants in LAPACK. Technical Report UT-CS-90-103, University of Tennessee Computer Science, April 1990.

[28] Hans W. Meuer, Erich Strohmaier, Jack J. Dongarra, and Horst D. Simon. *TOP500 Supercomputer Sites*, 38th edition, November 2011. (The report can be downloaded from `http://www.netlib.org/benchmark/top500.html`).

[29] J. Choi. A proposal for a set of parallel basic linear algebra subprograms. Technical Report UT-CS-95-292, University of Tennessee Knoxville, 1995. LAPACK Working Note 100.

[30] Jack Dongarra and R. Clint Whaley. A user's guide to the BLACS v1.1. Technical Report UT-CS-95-281, University of Tennessee Knoxville, March 1995. LAPACK Working Note 94 updated May 5, 1997 (VERSION 1.1).

[31] L. Valiant. A bridging model for parallel computation. *Communications of ACM*, 33(8):103–111, 1990.

[32] Fred G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.