TECHNICAL PAPER
Submitted to Journal of Parallel and Distributed Computing

SCALABILITY ISSUES AFFECTING THE DESIGN OF A DENSE LINEAR ALGEBRA LIBRARY[1]

Jack J. Dongarra ‡§
Robert A. van de Geijn †
David W. Walker §

‡ Department of Computer Science
University of Tennessee
107 Ayres Hall
Knoxville, TN 37996-1301

† Department of Computer Sciences
University of Texas
Austin, TX 78712

§ Mathematical Sciences Section
Oak Ridge National Laboratory
P. O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

Corresponding author:
David W. Walker
Oak Ridge National Laboratory
P. O. Box 2008
Oak Ridge, TN 37831-6367
(615) 574-7401 (office)
(615) 574-0680 (fax)
walker@msr.epm.ornl.gov (email)

[1]This work was supported in part by ARPA under contract number DAAL03-91-C-0047 administered by ARO, and in part by DOE under contract number DE-AC05-84OR21400.

SCALABILITY ISSUES AFFECTING THE DESIGN OF A DENSE LINEAR ALGEBRA LIBRARY

Jack J. Dongarra
Robert A. van de Geijn
David W. Walker

Abstract

This paper discusses the scalability of Cholesky, LU, and QR factorization routines on MIMD distributed memory concurrent computers. These routines form part of the ScaLAPACK mathematical software library that extends the widely-used LAPACK library to run efficiently on scalable concurrent computers. To ensure good scalability and performance, the ScaLAPACK routines are based on block-partitioned algorithms that reduce the frequency of data movement between different levels of the memory hierarchy, and particularly between processors. The block cyclic data distribution, that is used in all three factorization algorithms, is described. An outline of the sequential and parallel block-partitioned algorithms is given. Approximate models of algorithms' performance are presented to indicate which factors in the design of the algorithm have an impact upon scalability. These models are compared with timings results on a 128-node Intel iPSC/860 hypercube. It is shown that the routines are highly scalable on this machine for problems that occupy more than about 25% of the memory on each processor, and that the measured timings are consistent with the performance model. The contribution of this paper goes beyond reporting our experience: our implementations are available in the public domain.

1 Introduction

Massively parallel, distributed memory, concurrent computers are playing an increasingly important role in large scale scientific computing, and in particular are the primary target machines for "Grand Challenge" applications. In addition, massively parallel computers are also beginning to be more widely used in production engineering environments, and to a somewhat lesser extent in the business and finance sectors. A number of key software technologies are helping to accelerate the more widespread use of massively parallel computers in these areas. These include the development of parallelizing compilers, parallel language extensions such as Fortran D [25] and High Performance Fortran [33], the adoption of a standard message passing interface [18], support for parallel constructs and operations, such as guard layers and object migration, and a variety of tools for debugging, and visualizing/analyzing performance on massively parallel computers.

Another important and active research area is the development of re-usable software for multicomputers in the form of libraries and "tool-kits" [7, 23, 39]. Linear algebra—in particular, the solution of linear systems of equations—lies at the heart of most calculations in scientific computing. We are currently building a software library for performing linear algebra computations on multicomputers, and this paper deals primarily with the performance and scalability of the dense LU, Cholesky, and QR factorization routines that form the core of this library. In addition, we shall discuss how the design goals for the library, particularly the performance requirements, influenced the implementation of the core routines.

The scalable library we are developing for multicomputers will be fully compatible with the LAPACK library for vector and shared memory computers [1, 2, 12, 17], and is therefore called ScaLAPACK. LAPACK was designed to implement the earlier EISPACK and LINPACK linear algebra libraries efficiently on shared memory, vector supercomputers, and to improve the robustness of some of the algorithms. These types of computer have hierarchical memories, and a key concept in the design of LAPACK was to improve performance by minimizing the movement of data between the different layers of the memory. This was done by recasting the algorithms in block-partitioned form, so that the bulk of the computation is performed by matrix-matrix operations using the Level 3 Basic Linear Algebra Subprograms (BLAS) [15, 16]. This approach permits locality of reference to be preserved.

ScaLAPACK also uses block-partitioned algorithms to ensure good performance on MIMD distributed memory concurrent computers, by minimizing the frequency of data movement between different levels of the memory hierarchy [9, 10]. For such machines the memory hierarchy includes the off-processor memory of other processors, in addition to the hierarchy of registers, cache, and local memory on each processor, so the block partitioned approach is particularly useful in reducing the startup cost associated with interprocessor communication. This optimization is essential to the scalable performance of the library routines. The fundamental building blocks of the ScaLAPACK library are distributed memory versions of the Level 2 and Level 3 BLAS, and a set of Basic Linear Algebra Communication Subprograms (BLACS) [3, 21] for performing communication tasks that arise frequently in parallel linear algebra computations. In the ScaLAPACK routines all interprocessor communication takes place 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.

2 Requirements of Scalable Libraries

In developing a library of high-quality subroutines for performing dense linear algebra computations the design goals fall into three broad classes:

- performance goals
- ease-of-use goals
- range-of-use goals

These design goals will be discussed in the following three subsections.

2.1 Performance

Two important performance metrics are *concurrent efficiency* and *scalability*. We seek good performance characteristics in our algorithms by eliminating, as much as possible, overhead due to load imbalance, data movement, and algorithm restructuring. The way in which the data is distributed (or decomposed) over the memory hierarchy of a computer is of fundamental importance to these factors. Concurrent efficiency, ϵ, is defined as the concurrent speedup per processor [26], where the concurrent speedup is the execution time, $T_{\rm seq}$, for the best sequential algorithm running on one processor of the concurrent computer, divided by the execution time, T, of the parallel algorithm running on N_p processors. When direct methods are used, as in LU factorization, the concurrent efficiency depends on the problem size and the number of processors, so on a given parallel computer and for a fixed number of processors the running time should not vary greatly for problems of the same size. Thus, we may write,

$$\epsilon(N, N_p) = \frac{1}{N_p} \frac{T_{\rm seq}(N)}{T(N, N_p)} \tag{1}$$

where N represents the problem size. In dense linear algebra computations most of the computational work involves operations on floating-point numbers, so the sequential execution time is usually proportional to the floating-point operation count. Thus, the concurrent efficiency is related to the performance, G, measured in floating-point operations per second, by

$$G(N, N_p) = \frac{N_p}{t_{\rm calc}}\, \epsilon(N, N_p) \tag{2}$$

where $t_{\rm calc}$ is the time for one floating-point operation. For routines that iterate, such as eigensolvers, the number of iterations, and hence the execution time, depends not only on the problem size, but also on other characteristics of the input data, such as condition number. A parallel algorithm is said to be highly scalable [29, 36] if the concurrent efficiency depends on the problem size and number of processors only through their ratio. This ratio is simply the memory requirement of the problem per processor, often referred to as the granularity. Thus, for a highly scalable algorithm the concurrent efficiency is constant as the number of processors increases while keeping the granularity fixed. Alternatively, Eq. 2 shows that this is equivalent to saying that for a highly scalable algorithm the performance depends linearly on the number of processors for fixed granularity.

The degree of scalability may be gauged by the isoefficiency function, $\rho_\epsilon(N_p)$, which is defined to be the problem size necessary to maintain some fixed efficiency, ϵ, as the number of processors, N_p, varies. Thus, if $\rho_\epsilon(N_p)$ depends linearly on N_p then the concurrent efficiency is a function of the grain size, $g = N/N_p$. Such algorithms are highly scalable. Algorithms for which $\rho_\epsilon(N_p)$ is a rapidly increasing function of N_p are said to scale poorly.

The scalability of a parallel algorithm can be assessed by plotting the isoefficiency function for different values of ϵ, that is by plotting curves in the (N_p, N) plane on which ϵ is constant. On any particular machine we are usually interested in how an algorithm scales within a "window" in the (N_p, N) plane. This is shown schematically in Figure 1 in which the window is bounded to the right by the number of processors in the parallel machine, below by the size of the smallest problem of interest, and to the left and above by either the memory size per processor or runtime considerations. For some algorithms there may also be an upper bound on the problem size imposed by considerations of stability and/or accuracy. In Figure 1 we have assumed that the memory requirements scale linearly with the problem size. The runtime bound turns over as N_p increases as the concurrent efficiency falls off.

In designing an algorithm for a scalable library we seek one that is scalable within the windows of interest of as large a set of machines as possible. It should be noted that scalability studies conducted on small machines are of little use if the problem sizes considered are below the minimum size of interest. In the absence of an accurate performance model it is not valid to infer that an algorithm that scales well on one machine will also scale well on another machine, or even on the same machine for a larger number of processors. It is also important to note the distinction between good scalability and high efficiency. Suppose we have two algorithms with the same order of computational complexity that perform the same task. Algorithm 1 is efficient and scalable for some number of processors $N_p \leq N_*$, but loses scalability as the number of processors increases further. Algorithm 2 is less efficient than Algorithm 1 for a given problem size in the $N_p \leq N_*$ regime, but maintains good scalability for all N_p. In this case it is unclear which algorithm is "best" overall. For a sufficiently small number of processors Algorithm 1 is preferable, but for a larger machine Algorithm 2 is better. Such dilemmas raise the possibility of using polyalgorithms, that is applying different algorithms on different machines or for different

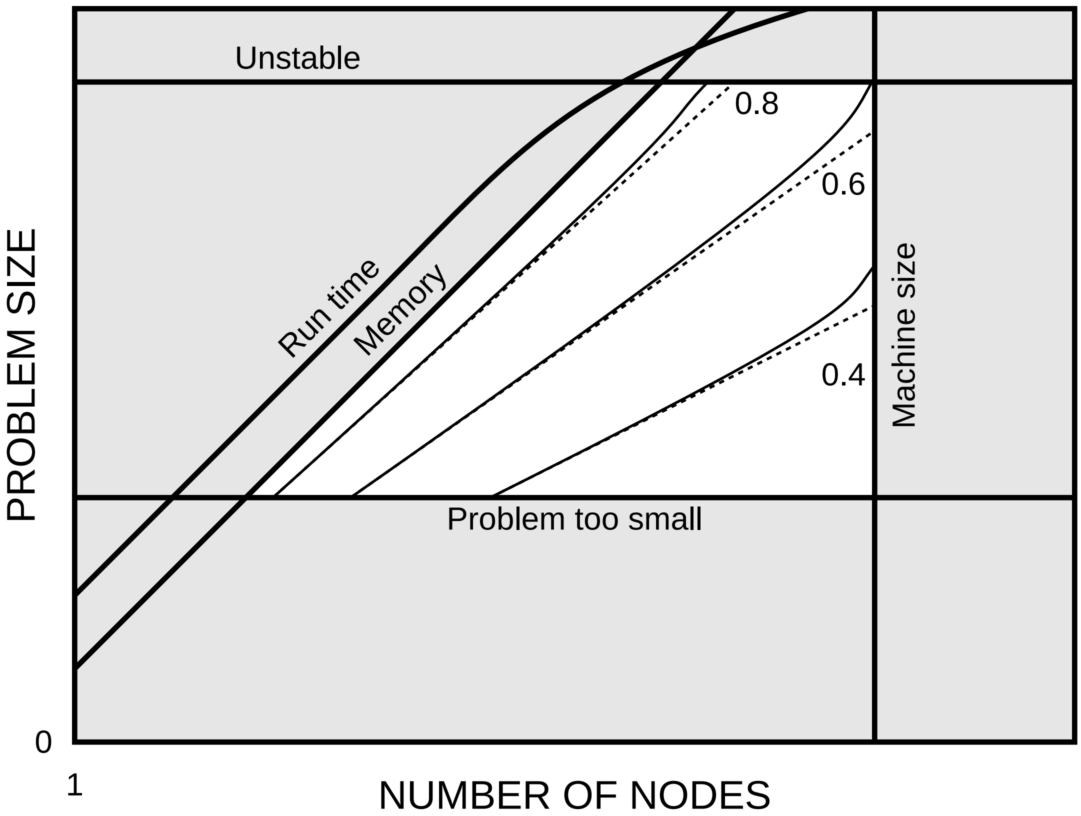

Figure 1: Schematic representation of isoefficiency curves within the window of interest, shown unshaded. The solid curves represent isoefficiency curves labeled by the efficiency. The dashed lines correspond to high scalability.

regions of the (N_p, N) plane.

The definition of scalability that we have adopted here is based on the performance per processor at fixed *memory* per processor. An alternative approach is to measure scalability in terms of performance per processor at fixed *computational work* per processor. This latter approach provides a useful scalability metric when runtime constrains the problem size [30, 31]. Large, dense linear algebra computations mostly arise in three-dimensional boundary element problems [24], and for such problems memory, rather than runtime, usually constrains the problem size [42]. As noted by Cwik, Patterson, and Scott, "simulations using integral equation methods are fundamentally limited by the amount of available memory," which has led them to develop an out-of-core solver for their electromagnetic scattering problem [11]. Moreover, the largest dense LU factorization problems run in-core on the Intel Delta system are solved in less than 12 minutes [13]. In Figure 1 memory constrains the problem size and forms the lefthand boundary of the window-of-interest. There are, of course, many problems in which the runtime constraint lies below the memory constraint, but for the problems and machines currently of interest to us we believe this not to be the case. For matrix problems the problem size is $O(N^2)$, so the memory constraint in the (N_p, N) plane is a straight line. The runtime is $O(N^3)$, so the runtime constraint is a curve of the form $N_p^{2/3}$. Thus, for a sufficiently large number of processors runtime will constrain the problem. The critical question is at what number of processors does this crossover occur? Alternatively, we could ask whether there is a significant number of dense linear algebra problems whose solution is so important that we are prepared to put up with very long runtimes. We believe that such a class of problems is of practical interest, and that for these problems the crossover takes place at machine sizes that are significantly larger than those currently available. Whether this continues to be the case in the future depends on how the technology advances. Thus, we scale the problem size with fixed memory per processor in our definition of scalability.

2.2 Ease-Of-Use

Ease-of-use is concerned with factors such as portability and the user interface to the library. Portability in its most inclusive sense means that the code is written in a standard language, such as Fortran, and that the source code can be compiled on an arbitrary machine to produce a program that will run correctly and efficiently. In our modular approach to ScaLAPACK it is assumed that the distributed Level 3 BLAS and the BLACS that form the building blocks of the library routines are available in optimized form for each target platform, and are linked in during compilation. Thus, only the upper layers of the ScaLAPACK library are fully portable in the source code sense. Ease-of-use is also related to the user interface, and is enhanced if implementation details are largely hidden from the user, for example, through the use of an object-based interface to the library. In addition to the LAPACK-compatible interface, we are also experimenting with developing interfaces for LAPACK and ScaLAPACK that are compatible with Fortran 90 [9] and C++ [20].

2.3 Range-Of-Use

Range-of-use may be gauged by how numerically stable the algorithms are over a range of input problems, and the range of data structures the library will support. For example, LAPACK deals with dense matrices stored in a rectangular array, packed matrices where only the upper or lower half of a symmetric matrix is stored, and banded matrices where only the nonzero bands are stored.

3 The ScaLAPACK Effort

3.1 Algorithms to be Included

Over the past three years the LAPACK linear algebra library has been designed and implemented for a wide range of shared memory supercomputers. When complete, ScaLAPACK will extend LAPACK to distributed memory concurrent computers. This section gives an overview of the functionality provided by LAPACK. LAPACK, which is based on the successful LINPACK [14] and EISPACK [27, 41] libraries, is portable and efficient across the range of large-scale, shared-memory, general-purpose computers. This portability and efficiency is achieved through the use in LAPACK of a set of basic linear algebra subroutines (called the Level 1, 2, and 3 BLAS) which perform basic operations such as scalar-vector, matrix-vector, and matrix-matrix multiplication. These subroutines, especially the matrix-matrix operations, can be optimized for each machine while the Fortran code that calls them remains identical and hence portable across all the machines. This approach lets us extract most of the performance that each machine has to offer, while restricting machine-dependent code to the BLAS and a few integer "tuning parameters."

LAPACK provides approximately the same functionality as LINPACK and EISPACK together, namely, solution of systems of simultaneous linear equation, least-squares solution of over-determined systems of equations, and solution of matrix eigenvalue problems (standard and generalized). The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the factorizations and condition numbers (or estimates thereof). Dense and band matrices are provided for, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices.

LAPACK includes routines for solving systems of linear equations for different types of matrices. In addition to general solvers for dense, banded, and tridiagonal matrices, special solve routines exist for symmetric/Hermitian indefinite and positive definite matrices, complex symmetric matrices, and positive definite banded and tridiagonal matrices. For the symmetric and Hermitian cases, versions exist that assume packed storage, thereby halving the memory required to store the matrix.

In addition to the linear system solvers, LAPACK includes routines for solving linear least squares problem, computing the singular value decomposition, for the eigenvalue problem (general, symmetric, and generalized problem).

3.2 Target Architectures

The EISPACK and LINPACK software libraries were designed for supercomputers in use in the 1970's and early 1980's, such as the CDC-7600, Cyber 205, and Cray-1 computers [35]. These machines featured multiple functional units that were pipelined to get good performance. The CDC-7600 was basically a high-performance scalar computer, while the Cyber 205 and Cray-1 were early vector computers.

The development of LAPACK in the late 1980's was intended to make the EISPACK and LINPACK libraries run efficiently on shared memory, vector supercomputers. The ScaLAPACK software library extends the use of LAPACK to distributed memory concurrent supercomputers. The development of ScaLAPACK began in 1991 and is expected to be completed by the end of 1994.

The underlying concept of both the LAPACK and ScaLAPACK libraries is the use of block-partitioned algorithms to minimize data movement between different levels in hierarchical memory. Thus, the ideas discussed in this paper for developing a library for performing dense linear algebra computations are applicable to any computer with a hierarchical memory that (1) imposes a sufficiently large startup cost on the movement of data between different levels in the hierarchy, and for which (2) the cost of a context switch is too great to make fine grainsize multithreading worthwhile. Our target machines are, therefore, medium and large grainsize advanced-architecture computers. These include "traditional" shared memory, vector supercomputers, such as the Cray Y-MP and C90, and MIMD distributed memory concurrent supercomputers, such as the Intel Paragon, and Thinking Machines' CM-5 [43], and the more recently announced IBM SP1 and Cray T3D concurrent systems. Since these machines have only very recently become available most of the ongoing development of the ScaLAPACK library is being performed on a 128-node Intel iPSC/860 hypercube and on the 512-node Intel Touchstone Delta system.

Future advances in compiler and hardware technologies in the mid to late 1990's are expected to make the multithreading paradigm a viable approach for masking communication costs. Since the blocks in a block-partitioned algorithm can be regarded as separate threads our approach will still be applicable on machines that exploit medium and coarse grainsize multithreading.

m	0	1	2	3	4	5	6	7	8	9
p	0	0	0	0	1	1	1	1	2	2
i	0	1	2	3	0	1	2	3	0	1

(a) Block

m	0	1	2	3	4	5	6	7	8	9
p	0	1	2	0	1	2	0	1	2	0
i	0	0	0	1	1	1	2	2	2	3

(b) Cyclic

Figure 2: Examples of block and cyclic decompositions of $M = 10$ data objects over $P = 3$ processes.

3.3 Data Distribution Schemes

The fundamental data object used in the ScaLAPACK library is the block-partitioned matrix. In this section, we describe the block-cyclic method for distributing such a matrix over a two-dimensional mesh of processes, or template. In general, each process has an independent thread of control, and with each process is associated some local memory directly accessible only by that process. The assignment of these processes to physical processors is a machine-dependent optimization issue.

An important property of the class of data distribution we shall use is that independent decompositions are applied over rows and columns. We shall, therefore, begin by considering the distribution of a vector of M data objects over P processes. This can be described by a mapping of the global index, m, of a data object to an index pair (p, i), where p specifies the process to which the data object is assigned, and i specifies the location in the local memory of p at which it is stored. We shall assume $0 \leq m < M$ and $0 \leq p < P$.

Two common decompositions are the *block* and the *cyclic* decompositions [26, 45]. The block decomposition assigns contiguous entries in the global vector to the processes in blocks.

$$m \mapsto (\lfloor m/L \rfloor, m \bmod L), \tag{3}$$

where $L = \lceil M/P \rceil$. The cyclic decomposition (also known as the wrapped or scattered decomposition) assigns consecutive entries in the global vector to successive different processes,

$$m \mapsto (m \bmod P, \lfloor m/P \rfloor) \tag{4}$$

Examples of the block and cyclic decompositions are shown in Figure 2.

The block cyclic decomposition is a generalization of the block and cyclic decompositions in which blocks of consecutive data objects are distributed cyclically over the processes. In the block cyclic decomposition the mapping of the global index, m, can be expressed as $m \mapsto (p, b, i)$, where p is the process number, b is the block number in process p, and i is the index within block b to which m is mapped. Thus, if the number of data objects in a block is r, the block cyclic decomposition may be written,

$$m \mapsto \left(\left\lfloor \frac{m \bmod T}{r} \right\rfloor, \left\lfloor \frac{m}{T} \right\rfloor, m \bmod r \right) \tag{5}$$

m	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
p	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2
b	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3
i	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

(a) $m \mapsto (p, b, i)$

p	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2
b	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3
i	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
m	0	1	6	7	12	13	18	19	2	3	8	9	14	15	20	21	4	5	10	11	16	17	22

(b) $(p, b, i) \mapsto m$

Figure 3: An example of the block cyclic decomposition of $M = 23$ data objects over $P = 3$ processes for a block size of $r = 2$. (a) shows the mapping from global index, m, to the triplet (p, b, i), and (b) shows the inverse mapping.

where $T = rP$. It should be noted that this reverts to the cyclic decomposition when $r = 1$, with local index $i = 0$ for all blocks. A block decomposition is recovered when $r = L$, in which case there is a single block in each process with block number $b = 0$. The inverse mapping of the triplet (p, b, i) to a global index is given by,

$$(p, b, i) \mapsto Br + i = pr + bT + i \tag{6}$$

where $B = p + bP$ is the global block number. The block cyclic decomposition is one of the data distributions supported by High Performance Fortran (HPF) [33], and has been previously used, in one form or another, by several researchers (see [4, 5, 6, 8, 19, 22, 32, 37, 38, 40, 44] for examples of its use). The block cyclic decomposition is illustrated with an example in Figure 3.

In decomposing an $M \times N$ matrix we apply independent block cyclic decompositions in the row and column directions. Thus, suppose the matrix rows are distributed with block size r over P processes by the block cyclic mapping $\mu_{r,P}$, and the matrix columns are distributed with block size s over Q processes by the block cyclic mapping $\nu_{s,Q}$. Then the matrix element indexed globally by (m, n) is mapped as follows,

$$\begin{aligned} m &\overset{\mu}{\longmapsto} (p, b, i) \\ n &\overset{\nu}{\longmapsto} (q, d, j) \end{aligned} \tag{7}$$

The decomposition of the matrix can be regarded as the tensor product of the row and column decompositions, and we can write,

$$(m, n) \mapsto (\,(p, q),\ (b, d),\ (i, j)\,) \tag{8}$$

The block cyclic matrix decomposition given by Eqs. 7 and 8 distributes blocks of size $r \times s$ to a mesh of $P \times Q$ processes. We shall refer to this mesh as the *process template*, and refer to processes by their position in the template. Equation 8 says that global index (m, n) is mapped to process (p, q), where it is stored in the block at location (b, d) in a two-dimensional array of blocks. Within this block it is stored at location (i, j). The decomposition is completely specified by the parameters r, s, P, and Q. In Figure 4 an example is given of the block cyclic decomposition of a 36×80 matrix for block size 3×5 and a process template 3×4.

The block cyclic decomposition can reproduce most of the data distributions commonly used in linear algebra computations on parallel computers. For example, if $Q = 1$ and $r = \lceil M/P \rceil$ the block row decomposition is obtained. Similarly, $P = 1$ and $s = \lceil N/Q \rceil$ gives a block column decomposition.

4 Core ScaLAPACK Routines

In this section we describe sequential, block-partitioned versions of the dense Cholesky, LU, and QR factorization routines that form the core of the ScaLAPACK library. The parallel algorithms for these routines will also be described.

4.1 LU factorization

We seek a factorization $A = LU$, where A and L are $M \times N$ matrices, and U is a $N \times N$ matrix. L is lower triangular with 1's on the main diagonal, and U is upper triangular. Suppose the system is partitioned as follows,

$$\left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right) = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array}\right) \left(\begin{array}{c|c} U_{00} & U_{01} \\ \hline 0 & U_{11} \end{array}\right) \tag{9}$$

where A_{00} is $r \times r$, A_{01} is $r \times (N - r)$, A_{10} is $(M - r) \times r$, and A_{11} is $(M - r) \times (N - r)$. L_{00} and L_{11} are lower triangular matrices with 1's on the main diagonal, and U_{00} and U_{11} are upper triangular matrices. Then we may write,

$$\begin{aligned} A_{00} &= L_{00}U_{00} && (10) \\ A_{10} &= L_{10}U_{00} && (11) \\ A_{01} &= L_{00}U_{01} && (12) \\ A_{11} &= L_{10}U_{01} + L_{11}U_{11} && (13) \end{aligned}$$

Equations 10 and 11 taken together perform an LU factorization on the first $M \times r$ panel of A (i.e., A_{00} and A_{10}). Once this is completed the matrices L_{00}, L_{10}, and U_{00} are known, and the lower triangular system in Eq. 12 can be solved to give U_{01}. Finally, we rearrange Eq. 13 as,

$$A'_{11} = A_{11} - L_{10}U_{01} = L_{11}U_{11} \tag{14}$$

p,q	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
1	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
2	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3
3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
4	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
5	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3
B 6	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
7	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
8	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3
9	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
10	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
11	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3

(a) Assignment of global block indices, (B, D), to processes, (p, q).

B,D	q 0				1				2				3			
p 0	0,0	0,4	0,8	0,12	0,1	0,5	0,9	0,13	0,2	0,6	0,10	0,14	0,3	0,7	0,11	0,15
	3,0	3,4	3,8	3,12	3,1	3,5	3,9	3,13	3,2	3,6	3,10	3,14	3,3	3,7	3,11	3,15
	6,0	6,4	6,8	6,12	6,1	6,5	6,9	6,13	6,2	6,6	6,10	6,14	6,3	6,7	6,11	6,15
	9,0	9,4	9,8	9,12	9,1	9,5	9,9	9,13	9,2	9,6	9,10	9,14	9,3	9,7	9,11	9,15
1	1,0	1,4	1,8	1,12	1,1	1,5	1,9	1,13	1,2	1,6	1,10	1,14	1,3	1,7	1,11	1,15
	4,0	4,4	4,8	4,12	4,1	4,5	4,9	4,13	4,2	4,6	4,10	4,14	4,3	4,7	4,11	4,15
	7,0	7,4	7,8	7,12	7,1	7,5	7,9	7,13	7,2	7,6	7,10	7,14	7,3	7,7	7,11	7,15
	10,0	10,4	10,8	10,12	10,1	10,5	10,9	10,13	10,2	10,6	10,10	10,14	10,3	10,7	10,11	10,15
2	2,0	2,4	2,8	2,12	2,1	2,5	2,9	2,13	2,2	2,6	2,10	2,14	2,3	2,7	2,11	2,15
	5,0	5,4	5,8	5,12	5,1	5,5	5,9	5,13	5,2	5,6	5,10	5,14	5,3	5,7	5,11	5,15
	8,0	8,4	8,8	8,12	8,1	8,5	8,9	8,13	8,2	8,6	8,10	8,14	8,3	8,7	8,11	8,15
	11,0	11,4	11,8	11,12	11,1	11,5	11,9	11,13	11,2	11,6	11,10	11,14	11,3	11,7	11,11	11,15

(b) Global blocks, (B, D), in each process, (p, q).

Figure 4: Block cyclic decomposition of a 36×80 matrix with a block size of 3×5, onto a 3×4 process template. Each small rectangle represents one matrix block – individual matrix elements are not shown. In (a), shading is used to emphasize the process template that is periodically stamped over the matrix, and each block is labeled with the process to which it is assigned. In (b), each shaded region shows the blocks in one process, and is labeled with the corresponding global block indices. In both figures, the black rectangles indicate the blocks assigned to process $(0, 0)$.

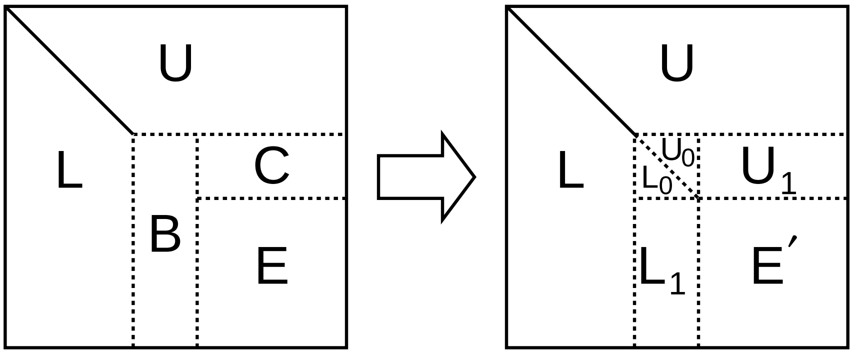

Figure 5: Stage $k+1$ of the block LU factorization algorithm showing how the panels B and C, and the trailing submatrix E are updated. The trapezoidal submatrices L and U have already been factored in previous steps. L has kr columns, and U has kr rows. In the step shown another r columns of L and r rows of U are evaluated.

From this equation we see that the problem of finding L_{11} and U_{11} reduces to finding the LU factorization of the $(M-r) \times (N-r)$ matrix A'_{11}. This can be done by applying the steps outlined above to A'_{11} instead of A. Repeating these steps K times, where

$$K = \min(\lceil M/r \rceil, \lceil N/r \rceil) \tag{15}$$

we obtain the LU factorization of the original $M \times N$ matrix A. For an in-place algorithm, A is overwritten by L and U – the 1's on the diagonal of L do not need to be stored explicitly. Similarly, when A is updated by Eq. 14 this may also be done in place.

After k of these K steps the first kr columns of L and the first kr rows of U have been evaluated, and matrix A has been updated to the form shown in Figure 5, in which panel B is $(M-kr) \times r$ and C is $r \times (N-(k-1)r)$. Step $k+1$ then proceeds as follows,

1. factor B to form the next panel of L, performing partial pivoting over rows if necessary. This evaluates the matrices L_0, L_1, and U_0 in Figure 5.

2. solve the triangular system $L_0 U_1 = C$ to get the next row of blocks of U.

3. do a rank-r update on the trailing submatrix E, replacing it with $E' = E - L_1 U_1$.

The LAPACK implementation of this form of LU factorization uses the Level 3 BLAS routines xTRSM and xGEMM to perform the triangular solve and rank-r update. We can regard the algorithm as acting on matrices that have been partitioned into blocks of $r \times r$ elements.

The parallel implementation of the block partitioned LU factorization proceeds as follows: assume A is distributed over a $P \times Q$ process template with a block cyclic distribution and a block size of $r \times r$. Assume that k panels of width r have been factored, and the remainder of the matrix has been updated accordingly, as illustrated in Figure 5.

Step $k+1$ then proceeds as follows:

1. the process in the process template that holds the column block $k+1$, i.e., panel B in Figure 5, performs an LU factorization of this panel, performing pivoting if necessary, and overwriting the corresponding entries of A with L_0, L_1, and U_0.
2. the panel (L_0 and L_1) is communicated to all other columns of the process template by broadcasting the appropriate pieces of the panel along rows of the template.
3. processes in columns of the template collaborate to apply pivoting to the portion of the matrix outside the factored panel, thereby affecting regions U_1, E, and L.
4. appropriate portions of U_1 are broadcast along columns of the process template.
5. the rank-r update $E' = E - L_1 U_1$ is performed.

4.2 Cholesky Factorization

Cholesky factorization factors a symmetric, positive-definite matrix A into the product of a lower triangular matrix, L, and its transpose, i.e., $A = LL^T$. We partition the $M \times M$ matrices A, L, and L^T, and write the system $A = LL^T$ as

$$\left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right) = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array}\right) \left(\begin{array}{c|c} L_{00}^T & L_{10}^T \\ \hline 0 & L_{11}^T \end{array}\right) \tag{16}$$

where the block A_{00} is a $r \times r$ matrix. We shall refer to r as the block size. The block-partitioned form of Cholesky factorization may be inferred inductively by writing,

$$\begin{aligned} A_{00} &= L_{00} L_{00}^T & (17) \\ A_{10} &= L_{10} L_{00}^T & (18) \\ A_{11} &= L_{10} L_{10}^T + L_{11} L_{11}^T & (19) \end{aligned}$$

If we assume that we know how to factor A_{00} as in Eq. 17, then the triangular system Eq. 18 can be solved to give L_{10}. Finally, we re-arrange Eq. 19 as,

$$A'_{11} = A_{11} - L_{10} L_{10}^T = L_{11} L_{11}^T \tag{20}$$

From this equation we see that the problem of finding L_{11} reduces to determining the Cholesky factorization of the $(M-r) \times (M-r)$ matrix A'_{11}. This can be done by recursively applying the steps outlined above to A'_{11}. Thus, to find the Cholesky factorization of the original matrix A requires $\lceil M/r \rceil$ such steps.

The parallel implementation of the block partitioned Cholesky factorization proceeds as follows: assume the lower triangle of A is distributed over a $P \times Q$ process template with a block cyclic distribution and a block size of $r \times r$. Assume that k panels of width r have been factored, and the remainder of the matrix has been updated accordingly. Figure 5 again applies, but only the lower triangular portion of the matrix is stored.

Step $k+1$ then proceeds as follows:

1. the process in the process template that holds the diagonal block $k+1$, i.e., the first block of panel B in Figure 5, performs a Cholesky factorization of this block, overwriting the corresponding entries of A with L_0.
2. L_0 is broadcast along the column of the process template holding panel $k+1$.
3. the remainder of the panel is updated in parallel by these processes, solving Eq. 18, to yield L_1.
4. the panel is communicated to all other columns of the process template by broadcasting the appropriate pieces of the panel along rows of the process template.
5. whereas in LU factorization U_1 is broadcast along columns of the process template, there is no similar communication phase in Cholesky factorization. Instead, processes of a template column collaborate to distribute the appropriate portions of the panel in the template column.
6. the rank-r update $E' = E - L_1 L_1^T$ is performed.

4.3 QR Factorization

Given an $M \times N$ matrix A, we seek the factorization $A = QR$, where Q is an $M \times M$ orthogonal matrix, and R is an $M \times N$ upper triangular matrix. We partition this factorization as follows,

$$A = \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right) = Q_1 \left(\begin{array}{c|c} I_r & 0 \\ \hline 0 & Q'_2 \end{array}\right) \left(\begin{array}{c|c} R_{00} & R_{10} \\ \hline 0 & R_{11} \end{array}\right) = Q_1 Q_2 R = QR \tag{21}$$

In this equation, both Q_1 and Q_2 are $M \times M$ orthogonal matrices, submatrix I_r denotes the $r \times r$ identity matrix, and Q'_2 donotes an $(M-r) \times (M-r)$ orthogonal matrix. Thus, Q_2 only affects the last $M-r$ rows of R. Again, r is the block size, i.e., the number of rows and columns of the matrices A_{00} and R_{00} in Eq. 21. Then we may write

$$\begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix} = Q_1 Q_2 \begin{pmatrix} R_{00} \\ 0 \end{pmatrix} = Q_1 \begin{pmatrix} R_{00} \\ 0 \end{pmatrix} \tag{22}$$

$$\begin{pmatrix} A_{01} \\ A_{11} \end{pmatrix} = Q_1 Q_2 \begin{pmatrix} R_{01} \\ R_{11} \end{pmatrix} \Rightarrow Q_1^T \begin{pmatrix} A_{01} \\ A_{11} \end{pmatrix} = \begin{pmatrix} R_{01} \\ Q_2' R_{11} \end{pmatrix} \tag{23}$$

where we have made use of the fact Q_2 has no affect on the first r rows of R. Equation 22 can be used to determine Q_1 and R_{00} from the QR factorization of the first panel of A. Multiplying the other $N-r$ columns of A by Q_1^T gives R_{01} and $A'_{11} = Q'_2 R_{11}$. Thus, the problem of finding the QR factorization of the $M \times N$ matrix A has been reduced to that of finding the QR factorization of the $(M-r) \times (N-r)$ matrix A'_{11}. Repeating this procedure K times, where K is given by Eq. 15, gives the complete QR factorization of the original matrix A.

In practice, Q is computed as a series of $\min(M, N) - 1$ Householder transformations of the form

$$H_i = I - 2\alpha_i v_i v_i^T \tag{24}$$

where $i = 1, \ldots, \min(M, N) - 1$, v_i is a vector of length M with zeroes for the first $i-1$ entries, and a one for the ith entry, and $\alpha_i = 1/(v_i^T v_i)$. In a typical QR factorization, the vectors v_i overwrite the entries of A below the diagonal, and α_i is stored in a vector. Furthermore, it can be shown that $H_1 \cdots H_r = I - VTV^T$, where T is upper triangular, and the ith column of V equals v_i. Indeed, this is how a version of the QR factorization that is rich in matrix-matrix operations is derived.

Assume that k of the above procedures have been performed, so the first kr columns of R and the first kr vectors v_i have respectively overwritten the first kr rows columns of A. Then, the last $N - kr$ columns have been updated accordingly, and A has been updated to the form shown in Figure 6, in which panel B is $(M - kr) \times r$ and C is $(M - kr) \times (N - (k+1)r)$. Step $k+1$ then proceeds as follows,

1. perform a QR factorization on panel B, overwriting the panel with the corresponding triangular factor and vectors $v_{kr+1}, \ldots, v_{kr+r}$. The factors $\alpha_{kr+1}, \ldots, \alpha_{kr+r}$ are also stored in a vector.
2. Form V_{k+1} and T_{k+1} so that $I - V_{k+1} T_{k+1} V_{k+1}^T = H_{kr+1} \cdots H_{kr+r}$.
3. Update $C' = (I - V_{k+1} T_{k+1} V_{k+1}^T) C$.

The LAPACK implementation of this form uses the Level 3 BLAS routines xGEMM and xTRMM to perform the various parts of the application of $(I - V_{k+1} T_{k+1} V_{k+1}^T)$ to C.

Turning now to the parallel implementation of the block-partitioned QR factorization algorithm, assume A is distributed over a $P \times Q$ process template with a block cyclic distribution and a block size of $r \times r$. Assume again that k panels of width r have been factored, and the remainder of the matrix has been updated accordingly. Step $k+1$ then proceeds as follows:

1. the column of the process template that holds panel $k+1$ performs a QR factorization of this panel, storing the Householder vectors as described before.

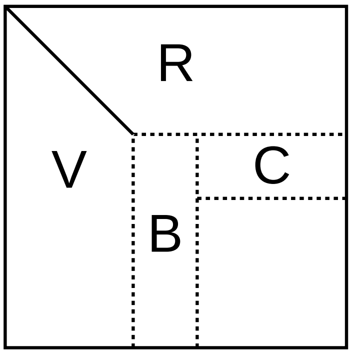

Figure 6: Stage $k+1$ of the block QR factorization algorithm, showing the panels B and C that are factored at this stage.

2. the same processes collaborate to form T_{k+1}, leaving this upper triangular matrix on the node that holds the diagonal block of A corresponding to the current panel.

3. matrix V_{k+1} is communicated to all other columns of the process template by broadcasting the appropriate pieces of this matrix along template rows.

4. matrix T_{k+1} is broadcast along the row of the process template in which it lies.

5. columns of the process template collaborate to form pieces of $V_{k+1}^T C$, leaving the result distributed over the template row that holds T_{k+1}.

6. processes in the template row that holds T_{k+1} independently update their portion of $V_{k+1}^T C$.

7. appropriate pieces of $T_{k+1} V_{k+1}^T C$ are broadcast along columns of the process template.

8. the rank-r update $C = C - V_{k+1}(T_{k+1} V_{k+1}^T C)$ is performed.

5 Performance and Scalability

In this section, we develop approximate models of the performance and scalability behavior of the parallel implementations of the three algorithms described in the previous section. We demonstrate that this behavior is observed in practice by reporting the performance attained on the 128-node Intel iPSC/860 hypercube at Oak Ridge National Laboratory.

5.1 Run Time Estimates and Scalability

Rather than concentrating on exact models for the performance of each of the algorithms, we develop approximate models by retaining only the leading terms affecting concurrent performance, efficiency, and scalability, and we shall concentrate on the information that can be extracted from these models.

It is customary to model the time for sending a message of length n items (in our case double precision numbers) between two nodes by

$$\alpha + n\beta$$

where α denotes the latency, and β the inverse of the bandwidth.

For square $N \times N$ matrices, the execution times of sequential implementations of the factorization algorithms are given by

$$T_{\text{seq}} = C_{\text{alg}} N^3 \gamma + O(N^2)$$

where γ is the time for a floating point operation, and the constant depends on the algorithm: $C_{\text{LU}} = 2/3$, $C_{\text{Chol}} = 1/3$, and $C_{\text{QR}} = 4/3$. Ideally, in the absence of concurrent overhead, we would expect the execution time for a logical mesh of $P \times Q$ processors to be

$$C_{\text{alg}} \frac{N^3}{P \cdot Q} \gamma + \frac{1}{P \cdot Q} O(N^2)$$

Instead the run time of the parallel LU factorization is estimated to be (see Appendix A),

$$\begin{aligned} \frac{2}{3} \frac{N^3}{P \cdot Q} \gamma \quad &+ \quad \frac{P+Q}{P \cdot Q} O(N^2) \gamma + \left[O(1) + O(\log_2(P)) \right] O(N) \alpha \\ &+ \quad \frac{P+Q}{P \cdot Q} \left[O(1) + O(\log_2(P)) \right] O(N^2) \beta \end{aligned} \tag{25}$$

The first term is due to the parallelization of the rank-r update. The second term is mainly due to load imbalance during the rank-r update, the panel factorization, which utilizes only P processes, and the simultaneous triangular solve, which utilizes only Q processes, all of which computations lie in the critical path of execution. The α term is due to the communications necessary for pivoting and broadcasting. The final term is due to the volume of communication that lies in the critical path of execution. In [32], it is shown theoretically that the minimum communication required for an LU factorization that balances the workload is $(P + Q)/(P \cdot Q) O(N^2) \beta$. The logarithmic factor in the last term of Eq. 25 can be removed by using a more sophisticated broadcast like the EDST algorithm in [34], at the expense of a higher cost for communication startup.

To examine scalability, first assume that $P = 1$. In this case, the matrix is column-wrapped over a one-dimensional array of processors, and the run time estimate becomes

$$T(N, 1 \times Q) = \frac{2}{3} \frac{N^3}{Q} \gamma \quad + \quad O(N^2) \gamma + O(N) \alpha + O(N^2) \beta \tag{26}$$

In this case, if we wish to maintain efficiency, ϵ, we must grow N with Q in the following way:

$$\begin{aligned} \text{Const.} &= \epsilon(N, 1 \times Q) = T_{\text{seq}}(N)/(Q \cdot T(N, 1 \times Q)) \\ &= [1 + O(Q/N) + O(Q/N^2)\alpha/\gamma + O(Q/N)\beta/\gamma]^{-1} \end{aligned} \tag{27}$$

Notice that in order to maintain efficiency, N must grow proportional to Q. Given the fact that memory requirements grow as N^2, this approach has rather poor scalability properties.

Next, assume a general $P \times Q$ process template, with $P > 1$. Then to maintain efficiency, the following must hold:

$$\begin{aligned} \text{Const.} &= \epsilon(N, P \times Q) = T_{\text{seq}}(N)/(P \cdot Q \cdot T(N, 1 \times Q)) \\ &= \Big[1 + O((P+Q)/N) + O(\log_2(P)(P \cdot Q)/N^2)\alpha/\gamma \\ &\quad + O(\log_2(P)(P+Q)/N)\beta/\gamma\Big]^{-1} \end{aligned} \tag{28}$$

Consider the case where $P = Q$. Once the number of nodes is large, $\log_2(P)$ grows very slowly and only mildly degrades the efficiency. If the variation of $\log_2(P)$ is ignored, constant efficiency can be maintained by letting N grow with P, the square root of the number of nodes.

> *The net result is that efficiency can be approximately maintained when the memory requirements for the matrix per node are kept constant.*

Thus, according to the definition of scalability given in Section 2.1, our implementation of dense LU factorization, using a two-dimensional block cyclic data distribution, is expected to exhibit good scalability.

The other algorithms lead to similar results, with different constants.

Notice that the estimated execution time is a function of many parameters. In particular, the following parameters can be chosen to optimize the performance of the parallel implementations:

1. template dimensions, $P \times Q$, which we will here assume correspond in some way to the physical dimensions of a grid that can be embedded in the given architecture.

2. block size, r, and

3. problem size, $M \times N$.

As a rule, massively parallel architectures are used to solve massive problems, so we can assume that the problem size is large enough to fill a large portion of available memory. For such problems it is always advantageous to add compute nodes. This leaves the user with the decision of how to choose the ratio P/Q, and r, that optimizes performance on a particular machine.

It is tempting to try to construct a more precise model of execution time, and hence to compute the optimal values of r and P/Q. In practice, however, the performance of the algorithm as a function of r depends much more on the size of the cache, the memory bandwidth, and the details of the CPU. Indeed, changing the blocksize by one can easily affect the performance on a single compute node by a factor of two [44]. As a result, r should be treated as a constant that depends on the implementation of the matrix-matrix multiply, and the hardware.

The optimal ratio P/Q could be predicted by the model. For our algorithms, communication within rows can be pipelined, and therefore partially hidden by computation. Computation within a column of the template is tightly coupled, making communication more difficult to hide. As a result, an optimal ratio will be attained when $P < Q$ [44].

5.2 Experiments on the iPSC/860

In this section, we show how experimental results support the theoretical scalability results by reporting performance attained by our algorithms on the Intel iPSC/860. The Intel iPSC/860 is a parallel architecture with up to 128 processing nodes. Each node consists of an Intel i860 processor, each with 8 Mbytes (on the ORNL machine). The interconnection network forms a hypercube. Logical grids of nodes can be embedded by having columns and rows form subcubes. On each node all computation was performed in double precision arithmetic, using assembly coded BLAS (Level 1, 2, and 3), which are part of a math library implemented by Kuck and Associates and provided by Intel.

As mentioned in the previous section, if the logarithmic factors in our performance model are ignored, our implementations based on two dimensional data decompositions allow efficiency to be retained when the memory use per node is held constant as the number of nodes is increased. We report performance as a function of the number of nodes in Figures 7–10. In these figures, the largest problem size per node of 6.25 Mbytes corresponds to the memory constraint in Figure 1, while the size of the Intel iPSC/860 (128 nodes) corresponds to the machine size constraint. In our timing experiments we were not constrained by run time, stability, or minimum problem size considerations, so the other constraints in Figure 1 are not relevant here.

The graphs that report performance attained per node clearly show the initial reduction, followed by a leveling, of the efficiency. This initial reduction is occurs as the efficiency falls from 1 for a single node to some approximately constant lower efficiency for more than one node. This behavior is consistent with the behavior predicted by the performance models. For the total performance graphs, the linearity of the plots when the number of nodes is increased as the memory usage is held constant, also illustrates the good scalability behavior of the algorithms on the Intel iPSC/860.

For the timing experiments reported on in Figures 7–9, the concurrent efficiencies, as defined by Eq. 1, lies in the range 0.5–0.6, 0.55–0.65, and 0.7–0.8, for the Cholesky, LU, and QR factorization algorithms, respectively. The smaller efficiencies correspond to smaller problem sizes per node. These efficiencies are quite impressive, since the bulk of the computation is being done

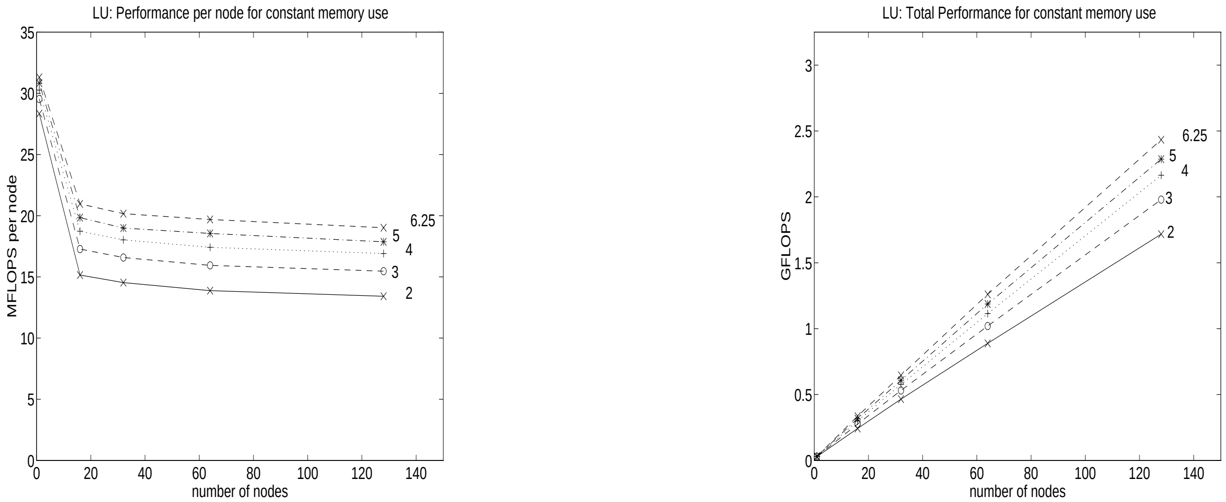

Figure 7: Performance of LU factorization as a function of the number of processors when the memory requirements per node are fixed at 2, 3, 4, 5, and 6.25 Mbytes. *Left:* Performance per node. *Right:* Total Performance.

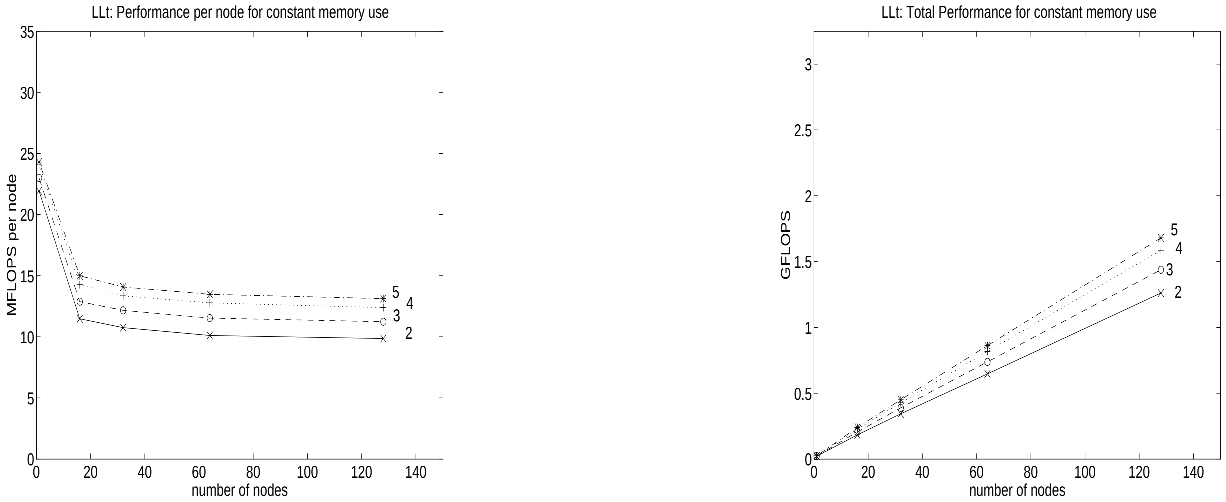

Figure 8: Performance of Cholesky factorization as a function of the number of processors when the memory requirements per node are fixed at 2, 3, 4, and 5 Mbytes. *Left:* Performance per node. *Right:* Total Performance.

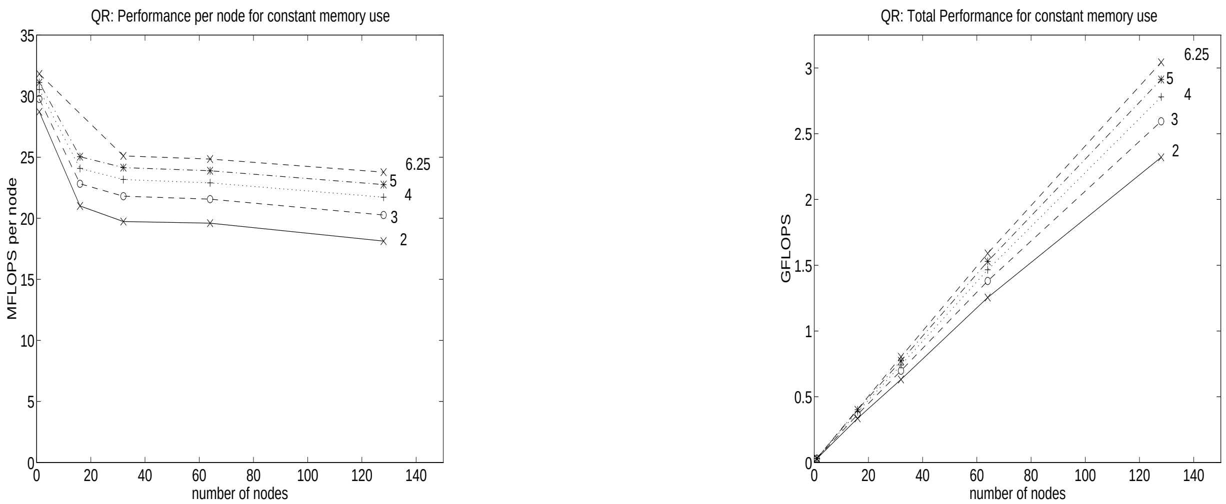

Figure 9: Performance of QR factorization as a function of the number of processors when the memory requirements per node are fixed at 2, 3, 4, 5, and 6.25 Mbytes. *Left:* Performance per node. *Right:* Total Performance.

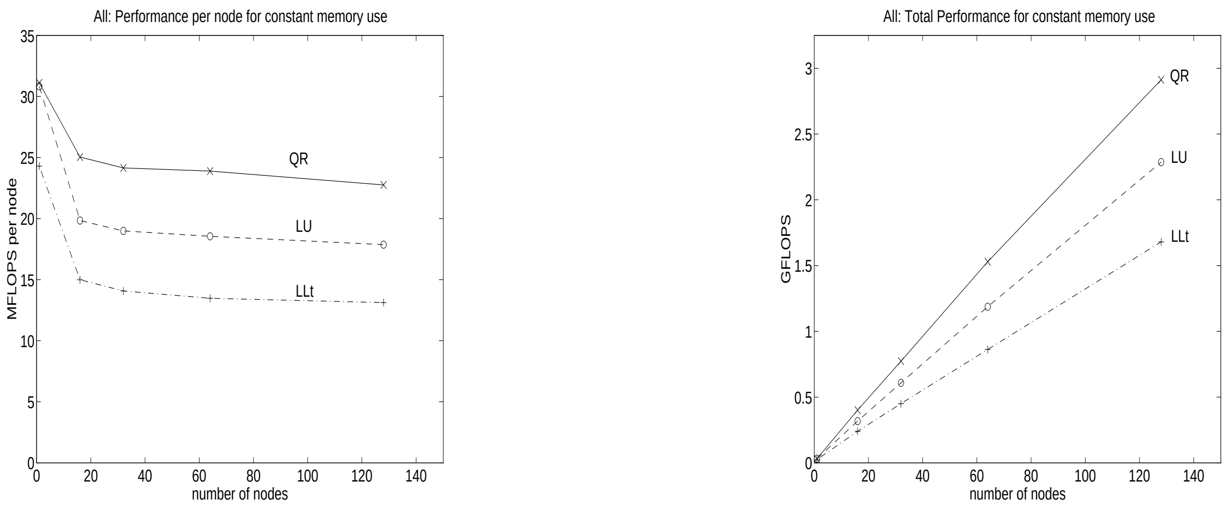

Figure 10: Performance of the different algorithms as a function of the number of processors when the memory requirements per node are fixed at 5 Mbytes. *Left:* Performance per node. *Right:* Total Performance.

by optimized, assembly-coded Level 3 BLAS routines, rather than by compiled Fortran code. Thus, the algorithms exhibit good scalability at acceptably high efficiency. As the figures show, the single node performance ranges from about 22 to 32 Mflops for these dense factorization routines. The peak performance of the Level 3 BLAS matrix-matrix multiplication routine on a single node is approximately 36.5 Mflops, which we regard as the peak performance of the i860 processor.

The bulk of the computation in all algorithms is in the rank-r update. We would expect the choice of the optimal block size to depend on the implementation of this BLAS operation. The block size for both LU and QR factorization is $r = 6$. For the Cholesky factorization, only the lower triangular portion of the matrix is updated. This poses a problem for the parallel algorithm, since the portion of the matrix on each node does not form a lower triangle, or other shape for which a single call to a Level 3 BLAS routine suffices. As a result, panels must be updated individually and, to get good performance from the BLAS, the width of the panels must increase. As a result, the optimal panel size for the Cholesky factorization is $r = 28$ for large matrices.

Since the bulk of the communication for the LU and QR factorizations is the same, the optimal grid size, $P \times Q$, for these factorization is the same: on the Intel iPSC/860 hypercube best performance is attained when $P/Q \approx 1/2$. Experimental results show the Cholesky factorization to perform best when the dimensions are (approximately) equal. This can be explained in part by the fact that the communication in Step 5 of Section 4.2 is greatly simplified when $P = Q$.

The relative performance of the algorithms is given in Figure 10. The difference in performance between the QR and LU factorizations is due to the more favorable ratio of computation to communication in the first. Similarly, the Cholesky factorization runs at a slower rate (as measured in floating-point operations per second) than both the LU and QR factorizations for the same reason. However, notice that for the single node implementation, the Cholesky factorization performs at 22–25 Mflops, compared with 28–32 Mflops for the LU and QR factorizations. Thus, even in the single node case, the Cholesky factorization performs significantly worse in terms of Mflops that the other two algorithms, although the actual runtime for Cholesky factorization, for a matrix of given size, is still less than the runtime for LU factorization, since the former involves about half as much computation as the latter. The relatively poor Mflop rate of the Cholesky factorization is due to the fact that the symmetric rank-r update is performed one panel at a time, even for the single node implementation, yielding an inferior performance of the BLAS. It should be noted that the LAPACK implementation does a single call to perform the symmetric rank-r update, yielding performance comparable to the LU and QR factorizations.

6 Conclusions and Future Work

The ScaLAPACK routines for performing linear algebra computations on distributed memory concurrent computers use distributed versions of the BLAS for performing most of the computational tasks, and the BLACS for communicating data. The use of block-partitioned algorithms improves scalability by reducing the frequency of data movement between processors. The computation performed by each processor within the distributed BLAS can itself be performed using assembly coded sequential BLAS routines, which results in good performance. The ScaLAPACK routines assume a block cyclic data distribution, and their performance can be tuned by the user on any given platform by varying the block size and the aspect ratio, P/Q, of the process template.

In this paper we have demonstrated the good performance and scalability characteristics of the Cholesky, LU, and QR dense factorization routines of the ScaLAPACK library on the Intel iPSC/860 hypercube. In addition, the ScaLAPACK library also currently includes routines for the concurrent solution of triangular systems. The message passing performed in these routines is based on the PICL interface [28], however, when the Message Passing Interface (MPI) standard [18] is complete we intend to rewrite the distributed BLAS and BLACS in terms of this, which should make the library more easily portable. We also intend to add more routines to ScaLAPACK, particularly those concerned with the estimation of condition numbers and the solution of dense eigenproblems, as discussed in Section 3.

Acknowledgments

Many individuals have been involved in the LAPACK and ScaLAPACK projects, including, E. Anderson, Z. Bai, J. Choi, C. Bischof, J. Demmel, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, J.Meyering, S. Ostrouchov, D. Sorensen, and C. Whaley. It is a pleasure to acknowledge their contributions. We are also grateful to Oak Ridge National Laboratory for use of the Intel iPSC/860 hypercube.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of Supercomputing '90*, pages 1–10. IEEE Press, 1990.

[2] E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM Press, Philadelphia, PA, 1992.

[3] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Basic Linear Algebra Communication Subprograms. In *Sixth Distributed Memory Computing Conference Proceedings*, pages 287–290. IEEE Computer Society Press, 1991.

[4] E. Anderson, A. Benzoni, J. J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. LAPACK for distributed memory architectures: Progress report. In *Parallel Processing for Scientific Computing, Fifth SIAM Conference*. SIAM, 1991.

[5] C. C. Ashcraft. The distributed solution of linear systems using the torus wrap data mapping. Engineering Computing and Analysis Technical Report ECA-TR-147, Boeing Computer Services, 1990.

[6] C. C. Ashcraft. A taxonamy of distributed dense LU factorization methods. Engineering Computing and Analysis Technical Report ECA-TR-161, Boeing Computer Services, 1991.

[7] Z. Bai and J. Demmel. Design of a parallel nonsymmetric eigenroutine toolbox, part i. In R. Sincovec, editor, *Proceedings of Sixth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM Press, 1993.

[8] R. P. Brent. The LINPACK benchmark on the AP 1000: Preliminary report. In *Proceedings of the 2nd CAP Workshop*, NOV 1991.

[9] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Computer Society Press, 1992.

[10] J. Choi, J. J. Dongarra, and D. W. Walker. The design of scalable software libraries for distributed memory concurrent computers. In *Proceedings of the CNRS-NSF Workshop on Environments and Tools for Parallel Scientific Computing*. Elsevier Science Publishers, 1993. To be published.

[11] T. Cwik, J. Patterson, and D. Scott. Electromagnetic scattering calculations on the intel touchstone delta. In *Proceedings of Supercomputing '92*, pages 538–542. IEEE Computer Society Press, 1992.

[12] J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. Prospectus for the development of a linear algebra library for high performance computers.

Technical Report 97, Argonne National Laboratory, Mathematics and Computer Science Division, September 1987.

[13] J. J. Dongarra. LINPACK benchmark: Performance of various computers using standard linear equations software. *Supercomputing Review*, 5(3):54–63, March 1992.

[14] J. J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, PA, 1979.

[15] J. J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A proposal for a set of level 3 basic linear algebra subprograms. Technical Report 88, Argonne National Laboratory, Mathematics and Computer Science Division, April 1987.

[16] J. J. Dongarra, I. Duff, J. Du Croz, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM TOMS*, 16:1–17, March 1990.

[17] J. J. Dongarra, I. S. Duff, and D. C. Sorensen H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA, 1990.

[18] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993.

[19] J. J. Dongarra and S. Ostrouchov. LAPACK block factorization algorithms on the Intel iPSC/860. Technical Report CS-90-115, University of Tennessee at Knoxville, Computer Science Department, October 1990.

[20] J. J. Dongarra, R. Pozo, and D. W. Walker. An object oriented design for high performance linear algebra on distributed memory architectures. In *Proceedings of Object Oriented Numerics Conference*, 1993. To be published.

[21] J. J. Dongarra and R. A. van de Geijn. Two-dimensional basic linear algebra communication subprograms. Technical Report LAPACK working note 37, Computer Science Department, University of Tennessee, Knoxville, TN, October 1991.

[22] J. J. Dongarra and R. A. van de Geijn. Reduction to condensed form for the eigenvalue problem on distributed memory architectures. *Parallel Computing*, 18:973–982, 1992.

[23] J. J. Dongarra, R. A. van de Geijn, and D. W. Walker. A look at scalable dense linear algebra libraries. In J. H. Saltz, editor, *Proceedings of the 1992 Scalable High Performance Computing Comference*. IEEE Press, 1992.

[24] A. Edelman. Large dense numerical linear algebra in 1993: The parallel computing influence. *International Journal of Supercomputing Applications*, 7(2), 1993.

[25] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C-W. Tseng, and M-Y. Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.

[26] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, N.J., 1988.

[27] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide Extension*, volume 51 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1977.

[28] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: a portable instrumented communication library. Technical Report TM-11616, Oak Ridge National Laboratory, October 1990.

[29] A. Gupta and V. Kumar. On the scalability of FFT on parallel computers. In *Proceedings of the Frontiers 90 Conference on Massively Parallel Computation*. IEEE Computer Society Press, 1990. Also available as technical report TR 90-20 from the Computer Science Department, University of Minnesota, Minneapolis, MN 55455.

[30] J. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[31] J. Gustafson, D. Rover, S. Elbert, and M. Carter. The design of a scalable, fixed-time computer benchmark. *Journal of Parallel and Distributed Systems*, 12:388–401, 1991.

[32] B. Hendrickson and D. Womble. The torus-wrap mapping for dense matrix computations on massively parallel computers. Technical Report SAND92-0792, Sandia National Laboratories, April 1992.

[33] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 0.4*, November 1992.

[34] C.-T. Ho and S. L. Johnsson. Distributed routing algorithms for broadcasting and personalized communication in hypercubes. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 640–648. IEEE, 1986.

[35] R. W. Hockney and C. R. Jesshope. *Parallel Computers*. Adam Hilger Ltd., Bristol, UK, 1981.

[36] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. Technical report, TR-91-18, Computer Science Department, University of Minn esota, June 1991. To appear in *Journal of Parallel and Distributed Computing*, 1994. A short

version of the paper appears in the Proceedings of the 1991 International Conference on Supercomputing, Germany, and as an invited paper in the Proceedings of 29th Annual Allerton Conference on Communuication, Control and Computing, Urbana,IL, October 1991.

[37] W. Lichtenstein and S. L. Johnsson. Block-cyclic dense linear algebra. Technical Report TR-04-92, Harvard University, Center for Research in Computing Technology, January 1992.

[38] Y. Saad and M. H. Schultz. Parallel direct methods for solving banded linear systems. Technical Report YALEU/DCS/RR-387, Department of Computer Science, Yale University, 1985.

[39] A. J. Skjellum and C. Baldwin. The multicomputer toolbox: Scalable parallel libraries for large-scale concurrent applications. Technical report, Numerical Mathematics Group, Lawrence Livermore National Laboratory, December 1991.

[40] A. J. Skjellum and A. Leung. LU factorization of sparse, unsymmetric, Jacobian matrices on multicomputers. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 328–337. IEEE Press, 1990.

[41] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1976.

[42] X.-H. Sun and L. Hi. Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Systems*, 19(1):27–37, 1993.

[43] Thinking Machines Corporation, Cambridge, MA. *CM-5 Technical Summary*, 1991.

[44] R. A. van de Geijn. Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems. Computer Science report TR-91-28, Univ. of Texas, 1991.

[45] E. F. Van de Velde. Data redistribution and concurrency. *Parallel Computing*, 16, December 1990.

A Run Time Estimate of the LU Factorization

Examine step $k+1$ of the algorithm.

1. The column of nodes that holds the $(k+1)$st panel collaborates to factor that panel. This must be broken down into r substeps, one for each of the columns. For the ith column of the panel,

   (a) determine the pivot row. This proceeds by first determining the local maximum on or below the diagonal, followed by the global maximum. Cost on a hypercube: about

   $$\left\lceil \frac{M-kr-i+1}{P} \right\rceil_r \gamma + \log_2(P)\alpha \tag{29}$$

   (assuming latency dominates the communication). Here $\lceil x \rceil_r$ denotes the smallest integer multiple of r greater than or equal to x.

   (b) pivot the portions of the pivot row that fall in this panel. Cost:

   $$\alpha + r\beta \tag{30}$$

   (Naturally, if the element is on the processor that holds the diagonal block of the matrix, this need not be done. We shall assume pivoting is always necessary.)

   (c) compute the multipliers, distributed among the column of nodes. Approximate cost:

   $$\left\lceil \frac{M-kr-i}{P} \right\rceil_r \gamma \tag{31}$$

   (d) update the rest of the panel using a rank-1 update. Approximate cost:

   $$\left\lceil \frac{M-kr-i}{P} \right\rceil_r (r-i)\gamma \tag{32}$$

2. Broadcast the pivot information within rows of nodes. This can be pipelined around an embedded ring (within the row), yielding a contribution of

   $$2(\alpha + r\beta) \tag{33}$$

3. Broadcast the factored panel (L_0 and L_1) in Figure 5):

   $$2\left(\alpha + \left\lceil \frac{M-kr}{P} \right\rceil_r r\beta\right) \tag{34}$$

4. Pivot the remainder of the rows:

$$r\left(\alpha+\left\lceil\frac{N-r}{Q}\right\rceil_r\beta\right) \tag{35}$$

5. Compute U_1 by solving $L_0U_1 = C$ on the row of nodes that holds C:

$$\left\lceil\frac{N-(k+1)r}{Q}\right\rceil_r r\gamma \tag{36}$$

6. Broadcast C:

$$\log_2(P)\left(\alpha+\left\lceil\frac{N-(k+1)r}{Q}\right\rceil_r r\beta\right) \tag{37}$$

7. Perform a rank-r update on matrix E,

$$2r\left\lceil\frac{M-(k+1)r}{P}\right\rceil_r\left\lceil\frac{N-(k+1)r}{Q}\right\rceil_r\gamma \tag{38}$$

After this step, the next column of nodes is ready to start step $k+2$.

To achieve a total estimate of the execution time, the above quantities must be summed over all steps. If for simplicity we consider only square matrices, $M = N$, and assume r is fixed and small, then summing the various contributions, we get:

$$\begin{aligned}
\sum_k\sum_i(29) &= \frac{N^2}{P}\gamma+\frac{N}{\log_2}(P)\alpha \\
\sum_k\sum_i(30) &= \frac{N}{r}\alpha+N\beta \\
\sum_k\sum_i(31) &= \frac{N^2}{2P}\gamma+O(N)\gamma \\
\sum_k\sum_i(32) &= \frac{N^2r}{4P}\gamma+O(N)\gamma \\
\sum_k(33) &= 2\frac{N}{r}\alpha+2N\beta \\
\sum_k(34) &= 2\frac{N}{r}\alpha+\frac{N^2}{P}\beta+O(N)\beta \\
\sum_k(35) &= N\alpha+\frac{N^2r}{Q}\beta+O(Nr)\beta \\
\sum_k(36) &= \frac{N^2}{2Q}\gamma+O(N)\gamma \\
\sum_k(37) &= \log_2(P)\frac{N}{r}\alpha+\log_2(P)\frac{N^2}{2Q}\beta+\log_2(P)O(N)\beta \\
\sum_k(38) &= \frac{2}{3}\frac{N^3}{P\cdot Q}\gamma+\frac{P+Q}{P\cdot Q}O(N^2)\gamma
\end{aligned}$$

This yields a total approximation of the execution time of

$$\begin{aligned}\frac{2}{3}\frac{N^3}{P \cdot Q}\gamma \quad &+ \quad \frac{P+Q}{P \cdot Q} O(N^2)\gamma + \left[O(1) + O(\log_2(P))\right] O(N)\alpha \\ &+ \quad \frac{P+Q}{P \cdot Q}\left[O(1) + O(\log_2(P))\right] O(N^2)\beta\end{aligned}$$