ORNL/TM-12470

Engineering Physics and Mathematics Division

Mathematical Sciences Section

THE DESIGN AND IMPLEMENTATION OF THE SCALAPACK LU, QR, AND CHOLESKY FACTORIZATION ROUTINES

Jaeyoung Choi [§]
Jack J. Dongarra [§†]
Susan Ostrouchov [§]
Antoine P. Petitet [§]
David W. Walker [†]
R. Clint Whaley [§]

[§] Department of Computer Science
University of Tennessee at Knoxville
107 Ayres Hall
Knoxville, TN 37996-1301

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

Date Published: September 1994

Research was supported in part by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy, by the Defense Advanced Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research Office, and in part by the Center for Research on Parallel Computing

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

Contents

1 Introduction 1
2 Design Philosophy 2
  2.1 Block Cyclic Data Distribution 2
  2.2 Building Blocks 4
  2.3 Design Principles 5
3 Factorization Routines 6
  3.1 LU Factorization 6
  3.2 QR Factorization 8
  3.3 Cholesky Factorization 11
4 Results and Discussion 12
  4.1 LU Factorization 13
  4.2 QR Factorization 15
  4.3 Cholesky Factorization 17
5 Scalability and Conclusions 20
6 References 25

THE DESIGN AND IMPLEMENTATION OF THE SCALAPACK LU, QR, AND CHOLESKY FACTORIZATION ROUTINES

Jaeyoung Choi
Jack J. Dongarra
Susan Ostrouchov
Antoine P. Petitet
David W. Walker
R. Clint Whaley

Abstract

This paper presents LU, QR and Cholesky factorization routines for dense matrices, which are important components of the ScaLAPACK library. The ScaLAPACK routines are implemented assuming matrices have a block cyclic data distribution, and are built using the BLAS [12, 13, 16], the BLACS [3], and the PBLAS, which provide a simplified interface around the PB-BLAS [7]. In implementing the ScaLAPACK routines, a major objective was to parallelize the corresponding sequential LAPACK using the BLAS, BLACS, and PBLAS as building blocks, leading to straightforward parallel implementations without sacrificing performance.

We present the details of the implementation of the ScaLAPACK factorization routines, and performance and scalability results on the Intel iPSC/860, the Intel Touchstone Delta, and the Intel Paragon systems.

1. Introduction

Current advanced architecture computers are NUMA (Non-Uniform Memory Access) machines. They possess hierarchical memories, in which accesses to data in the upper levels of the memory hierarchy (registers, cache, and/or local memory) are faster than those in lower levels (shared or off-processor memory). One technique to more efficiently exploit the power of such machines is to develop algorithms that maximize reuse of data in the upper levels of memory. This can be done by partitioning the matrix or matrices into blocks and by performing the computation with matrix-vector or matrix-matrix operations on the blocks. A set of BLAS (Level 2 and 3 BLAS) [12, 13] were proposed for that purpose. The Level 3 BLAS have been successfully used as the building blocks of a number of applications, including LAPACK [1, 2], which is a successor of LINPACK [11] and EISPACK [17]. The LAPACK is a software library that uses block-partitioned algorithms for performing dense and banded linear algebra computations on vector and shared memory computers.

The scalable library we are developing for distributed-memory concurrent computers will be fully compatible with the LAPACK library for vector and shared memory computers, and is therefore called ScaLAPACK ("Scalable LAPACK") [5]. ScaLAPACK also makes use of block-partitioned algorithms. It can be used to solve the "Grand Challenge" problems on massively parallel, distributed-memory, concurrent computers [4, 15].

The Basic Linear Algebra Communication Subprograms (BLACS) [3] comprise a package that provides ease-of-use and portability for message-passing in parallel linear algebra applications. The Parallel BLAS (PBLAS), which provide a simplified interface around the Parallel Block BLAS (PB-BLAS) [7], are intermediate level routines based on the sequential BLAS and the BLACS. The PBLAS provide all the functionality supported by parallel versions of the Level 2 and Level 3 BLAS on a restricted class of matrices having a block cyclic data distribution. The ScaLAPACK routines are built using the sequential BLAS, the BLACS, and the PBLAS modules. ScaLAPACK can be ported with minimal code modification to any machine on which the BLAS and the BLACS are available.

This paper presents the implementation details, performance, and scalability of the ScaLAPACK routines for the LU, QR and Cholesky factorization of dense matrices. Throughout the implementation of ScaLAPACK, we have tried to follow the LAPACK programming style by hiding most of the communications inside of the PBLAS and the ScaLAPACK auxiliary routines. We want to demonstrate how to make it simple to implement the complicated parallel routines without sacrificing performance.

Currently ScaLAPACK includes factorization routines with their solvers, routines to refine the solution to reduce the error, and routines to estimate the reciprocal of the condition number. ScaLAPACK also includes routines to reduce a real general matrix to Hessenberg or bidiagonal

form, and a symmetric matrix to tridiagonal form. These reduction routines are considered in our separate paper [10].

The design philosophy of the ScaLAPACK library is addressed in Section 2. In Section 3, we describe the ScaLAPACK factorization routines by comparing them with the corresponding LAPACK routines. Section 4 presents more details of the parallel implementation of the routines and performance results on the Intel family of computers: the iPSC/860, the Touchstone Delta, and the Paragon. In Section 5, the scalability of the algorithms on the systems is demonstrated, and conclusions and future work are presented.

2. Design Philosophy

In ScaLAPACK, algorithms are presented in terms of *processes,* rather than the processors of the physical hardware. A process is an independent thread of control with its own nonshared, distinct memory. Processes communicate by pairwise point-to-point communication or by collective communication as necessary. In general there may be several processes on a physical processor, in which case it is assumed that the runtime system handles the scheduling of processes. For example, execution of a process waiting to receive a message may be suspended and another process scheduled, thereby overlapping communication and computation. In the absence of such a sophisticated operating system, ScaLAPACK has been developed and tested for the case of one process per processor.

2.1. Block Cyclic Data Distribution

The way in which a matrix is distributed over the processes has a major impact on the load balance and communication characteristics of the concurrent algorithm, and hence largely determines its performance and scalability. The block cyclic distribution provides a simple, yet general-purpose way of distributing a block-partitioned matrix on distributed memory concurrent computers. The block cyclic data distribution is parameterized by the four numbers P, Q, r, and c, where $P \times Q$ is the process template and $r \times c$ is the block size. Blocks separated by a fixed stride in the column and row directions are assigned to the same process.

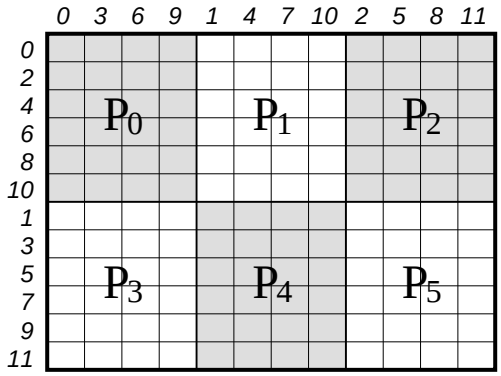
Suppose we have M objects indexed by the integers $0, 1, \cdots, M-1$. In the block cyclic data distribution the mapping of the global index, m, can be expressed as $m \longmapsto \langle p, b, i \rangle$, where p is the logical process number, b is the block number in process p, and i is the index within block b to which m is mapped. Thus, if the number of data objects in a block is r, the block cyclic data distribution may be written as follows:

$$m \longmapsto \left\langle s \bmod P,\ \left\lfloor \frac{s}{P} \right\rfloor,\ m \bmod r \right\rangle$$

where $s = \lfloor m/r \rfloor$ and P is the number of processes. The distribution of a block-partitioned

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	2	0	1	2	0	1	2	0	1	2
1	3	4	5	3	4	5	3	4	5	3	4	5
2	0	1	2	0	1	2	0	1	2	0	1	2
3	3	4	5	3	4	5	3	4	5	3	4	5
4	0	1	2	0	1	2	0	1	2	0	1	2
5	3	4	5	3	4	5	3	4	5	3	4	5
6	0	1	2	0	1	2	0	1	2	0	1	2
7	3	4	5	3	4	5	3	4	5	3	4	5
8	0	1	2	0	1	2	0	1	2	0	1	2
9	3	4	5	3	4	5	3	4	5	3	4	5
10	0	1	2	0	1	2	0	1	2	0	1	2
11	3	4	5	3	4	5	3	4	5	3	4	5

(a) block distribution over 2 x 3 template



(b) data distribution from processor point-of-view

Figure 1: A matrix with 12×12 blocks is distributed over a 2×3 process template. (a) The shaded and unshaded areas represent different templates. The numbered squares represent blocks of elements, and the number indicates at which location in the process template the block is stored – all blocks labeled with the same number are stored in the same process. The *slanted* numbers, on the left and on the top of the matrix, represent indices of a row of blocks and of a column of blocks, respectively. (b) It is easier to see the distribution from the process point-of-view in order to implement algorithms. Each process has 6×4 blocks.

matrix can be regarded as the tensor product of two such mappings: one that distributes the rows of the matrix over P processes, and another that distributes the columns over Q processes. That is, the matrix element indexed globally by (m, n) can be written as

$$(m, n) \longmapsto \langle (p, q), (b, d), (i, j) \rangle .$$

Figure 1 (a) shows an example of the block cyclic data distribution, where a matrix with 12×12 blocks is distributed over a 2×3 template. Therefore each process has 6×4 blocks as in Figure 1 (b). The block cyclic data distribution is the only distribution supported by the ScaLAPACK routines. The block cyclic data distribution can reproduce most data distributions used in linear algebra computations. For example, one-dimensional distributions over rows or columns are obtained by choosing P or Q to be 1.

The nonscattered decomposition (or pure block distribution) is just a special case of the cyclic distribution in which the block size is given by $r = \lceil M/P \rceil$ and $c = \lceil N/Q \rceil$. That is,

$$(m, n) \longmapsto \left\langle \left(\left\lfloor \frac{m}{r} \right\rfloor , \left\lfloor \frac{n}{c} \right\rfloor \right) , (0, 0), (m \bmod r, n \bmod c) \right\rangle .$$

Similarly a purely scattered decomposition (or two dimensional wrapped distribution) is another special case in which the block size is given by $r = c = 1$,

$$(m, n) \longmapsto \left\langle (m \bmod P, n \bmod Q), \left(\left\lfloor \frac{m}{P} \right\rfloor , \left\lfloor \frac{n}{Q} \right\rfloor \right) , (0, 0) \right\rangle .$$

In factorization routines, such as the LU, QR and Cholesky factorizations, in which the distribution of work becomes uneven as the computation progresses, a larger block size results in greater load imbalance, but reduces the frequency of communication between processes. There is, therefore, a tradeoff between load imbalance and communication startup cost which can be controlled by varying the block size.

In addition to the load imbalance that arises as distributed data are eliminated from a computation, load imbalance may also arise due to computational "hot spots" where certain processes have more work to do between synchronization points than others. This is the case, for example, in the LU factorization algorithm in which partial pivoting is performed over rows, and only a single column of the process template is involved in the pivot search while the other processes are idle. Similarly, the evaluation of each block row of the U matrix requires the solution of a lower triangular system which involves only processes in a single row of the process template. The details of the implementation are described in Sections 3.1 and 4.1. The effect of this type of load imbalance can be minimized through the choice of P and Q.

2.2. Building Blocks

The ScaLAPACK routines are built out of a small number of modules. The most fundamental of these are the sequential BLAS, in particular the Level 2 and 3 BLAS, and the BLACS, which perform common matrix-oriented communications tasks. ScaLAPACK can be ported with minimal code modification to any machine on which the BLAS and the BLACS are available.

The BLACS comprise a package that provides ease-of-use and portability for message-passing in a parallel linear algebra program. The BLACS efficiently support not only point-to-point operations between processes on a logical two-dimensional process template, but also collective communications on such templates, or within just a template row or column.

Future software for dense linear algebra on MIMD platforms could consist of calls to the BLAS for computation and calls to the BLACS for communication. Since both packages will have been optimized for each particular platform, good performance should be achieved with relatively little effort. The BLACS have been implemented for the Intel family of computers, the TMC CM-5, the CRAY T3D, the IBM SP1 and SP2, and for PVM.

The Parallel BLAS (PBLAS) provide a simplified interface to the Parallel Block BLAS (PB-BLAS) [7] – the PBLAS are essentially C wrappers around the PB-BLAS, which in turn are intermediate level routines based on the BLACS and the sequential BLAS. The PBLAS provide all the functionality supported by parallel, distributed versions of the Level 2 and Level 3 BLAS, however, the PBLAS can only be used in operations on a restricted class of matrices having a block cyclic data distribution. These restrictions permit certain memory access and communication optimizations that would not be possible (or would be difficult) if

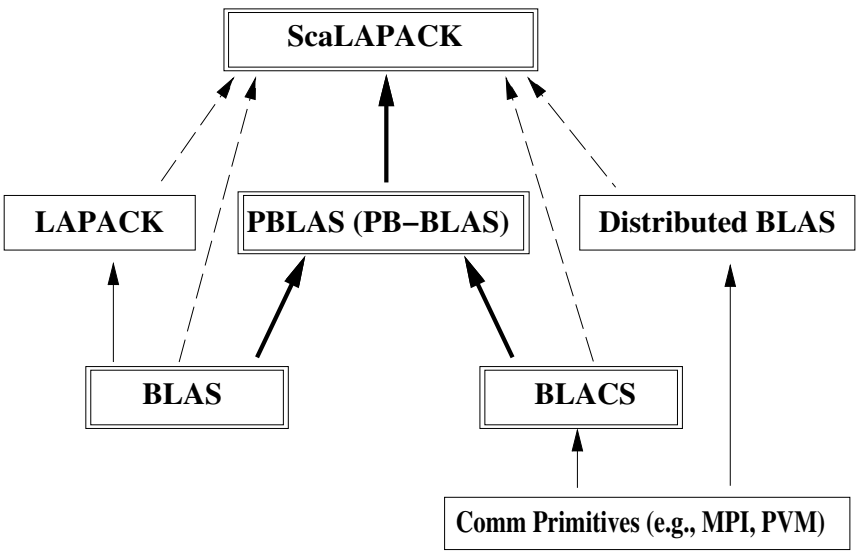


Figure 2: Hierarchical view of ScaLAPACK.

general-purpose distributed Level 2 and Level 3 BLAS were used [6, 8].

The sequential BLAS, the BLACS, and the PBLAS are the modules from which the higher level ScaLAPACK routines are built. The PBLAS are used as the highest level building blocks for implementing the ScaLAPACK library and provide the same ease-of-use and portability for ScaLAPACK that the BLAS provide for LAPACK. Most of the Level 2 and 3 BLAS routines in LAPACK routines can be replaced with the corresponding PBLAS routines in ScaLAPACK, so the source code of the top software layer of ScaLAPACK looks very similar to that of LAPACK. Thus, the ScaLAPACK code is modular, clear, and easy to read.

Figure 2 shows a hierarchical view of ScaLAPACK. Main ScaLAPACK routines usually call only the PBLAS, but the auxiliary ScaLAPACK routines may need to call the BLAS directly for local computations and the BLACS for communication among processes. In many cases the ScaLAPACK library will be sufficient to build applications. However, more expert users may make use of the lower level routines to build customized routines not provided in ScaLAPACK.

2.3. Design Principles

ScaLAPACK is designed to be the message-passing version of LAPACK. By maximizing the size of the submatrices multiplied in each process, that is, by maximizing the data reuse in the upper level of memory, it is possible to maximize the performance of the sequential BLAS. Similarly, by maximizing the size of the submatrices communicated among processes, the frequency of communication among processes can be reduced, thereby minimizing the communication startup cost. These two factors ensure that the ScaLAPACK routines have good performance

and scalability characteristics.

The ScaLAPACK routines perform correctly for a wide range of inputs. For example, in the process of computing the elementary Householder vector in the QR factorization, the Euclidean norm needs to be computed without causing overflow and underflow problems. A PBLAS routine, `PDNRM2`, takes care of the problem (see Section 4.2). Similarly if the data matrix is not positive definite in the Cholesky factorization, a process, which computes the Cholesky factorization on a diagonal block, halts its computation, yet other processes would keep waiting to finish their jobs. This problem can be avoided by broadcasting a flag to other processes to abort the computation (see Section 4.3).

3. Factorization Routines

In this section, we first briefly describe sequential, block-partitioned versions of the dense LU, QR, and Cholesky factorization routines of the LAPACK library. We use the right-looking versions of the routines for implementing them on distributed-memory concurrent computers, since this minimizes data communication and distributes the computation across all processes [14]. Then the parallel versions of these routines will be described.

For the implementation of the parallel block partitioned algorithms in ScaLAPACK, we assume that a matrix A is distributed over a $P \times Q$ process template with a block cyclic distribution and a block size of $n_b \times n_b$. Thus each column (or row) panel lies in one column (row) of the process template.

3.1. LU Factorization

The LU factorization applies a sequence of Gaussian eliminations to form $A = LU$, where A and L are $M \times N$ matrices, and U is an $N \times N$ matrix. L is unit lower triangular (lower triangular with 1's on the main diagonal), and U is upper triangular.

At the k-th step of the computation, it is assumed that the $m \times n$ submatrix of A ($m = M - k \cdot n_b$, $n = N - k \cdot n_b$) is to be partitioned as follows,

$$\begin{aligned}\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} &= \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \\ &= \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix}\end{aligned}$$

where the block A_{11} is $n_b \times n_b$, A_{12} is $n_b \times (n - n_b)$, A_{21} is $(m - n_b) \times n_b$, and A_{22} is $(m - n_b) \times (n - n_b)$. L_{11} is a unit lower triangular matrix, and U_{11} is an upper triangular matrix.

At first, a sequence of Gaussian eliminations is performed on the first $m \times n_b$ panel of A

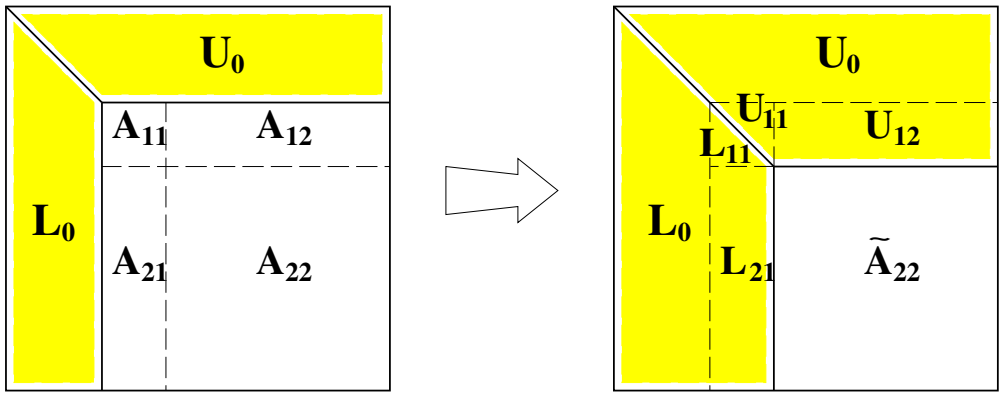


Figure 3: A snapshot of block LU factorization. It shows how the column panel, L_{11} and L_{21}, and the row panel, U_{11} and U_{12}, are computed, and how the trailing submatrix A_{22} is updated. The shaded areas represent data for which the corresponding computations are completed, that is, no more changes for these data will occur.

(i.e., A_{11} and A_{21}). Once this is completed, the matrices L_{11}, L_{21}, and U_{11} are known, and we can rearrange the block equations,

$$\begin{aligned} U_{12} &\Leftarrow (L_{11})^{-1} A_{12}, \\ \tilde{A}_{22} &\Leftarrow A_{22} - L_{21} U_{12} = L_{22} U_{22}. \end{aligned}$$

The LU factorization can be done by recursively applying the steps outlined above to the $(m - n_b) \times (n - n_b)$ matrix $\tilde{A}_{22}$. Figure 3 shows a snapshot of the block LU factorization. It shows how the column panel, L_{11} and L_{21}, and the row panel, U_{11} and U_{12}, are computed, and how the trailing submatrix A_{22} is updated. In the figure, the shaded areas represent data for which the corresponding computations are completed.

The computation of the above steps in the LAPACK routine, `DGETRF`, involves the following operations:

1. `DGETF2`: Apply the LU factorization on an $m \times n_b$ column panel of A (i.e., A_{11} and A_{21}).
   - [Repeat n_b times ($i = 1, \cdots, n_b$)]
     - `IDAMAX`: find the (absolute) maximum element of the i-th column and its location
     - `DSWAP`: interchange the i-th row with the row which holds the maximum
     - `DSCAL`: scale the i-th column of the matrix
     - `DGER`: update the trailing submatrix
2. `DLASWP`: Apply interchanges to the rest of columns.
3. `DTRSM`: Compute the subdiagonal block of U,

$U_{12} \Leftarrow (L_{11})^{-1} A_{12}$.

4. `DGEMM`: Update the rest of the matrix, A_{22},

   $\tilde{A}_{22} \Leftarrow A_{22} - L_{21}U_{12} = L_{22}U_{22}$.

The corresponding parallel implementation of the ScaLAPACK routine, `PDGETRF`, proceeds as follows:

1. `PDGETF2`: A column of processes performs the LU factorization on an $m \times n_b$ panel of A (i.e., A_{11} and A_{21}).
   - [Repeat n_b times ($i = 1, \cdots, n_b$)]
     - `PDAMAX`: find the (absolute) maximum value of the i-th column and its location (pivot information will be stored on the column of processes)
     - `PDLASWP`: interchange the i-th row with the row which hold the maximum
     - `DSCAL`: scale the i-th column of the matrix
     - `PDGER`: broadcast the i-th row columnwise ($(n-i)$ elements) and update the trailing submatrix
   - Broadcast the pivot information rowwise
2. `PDLASWP`: Apply interchanges to the rest of columns
3. `PDTRSM`: L_{11} is broadcast along a row of the processes, which compute the row panel U_{12}.
4. `PDGEMM`: The column panel L_{21} and the row panel U_{12} are broadcast rowwise and columnwise, respectively. Then, processes update their local portions of the matrix, A_{22}.

3.2. 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. At the k-th step of the computation, we partition this factorization to the $m \times n$ submatrix of A as

$$A = \begin{pmatrix} A_1 & A_2 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q \cdot \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$

where the block A_{11} is $n_b \times n_b$, A_{12} is $n_b \times (n - n_b)$, A_{21} is $(m - n_b) \times n_b$, and A_{22} is $(m - n_b) \times (n - n_b)$. A_1 is an $m \times n_b$ matrix, containing the first n_b columns of the matrix A, and A_2 is an $m \times (n - n_b)$ matrix, containing the last $(n - n_b)$ columns of A (that is, $A_1 = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ and $A_2 = \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}$). R_{11} is a $n_b \times n_b$ upper triangular matrix.

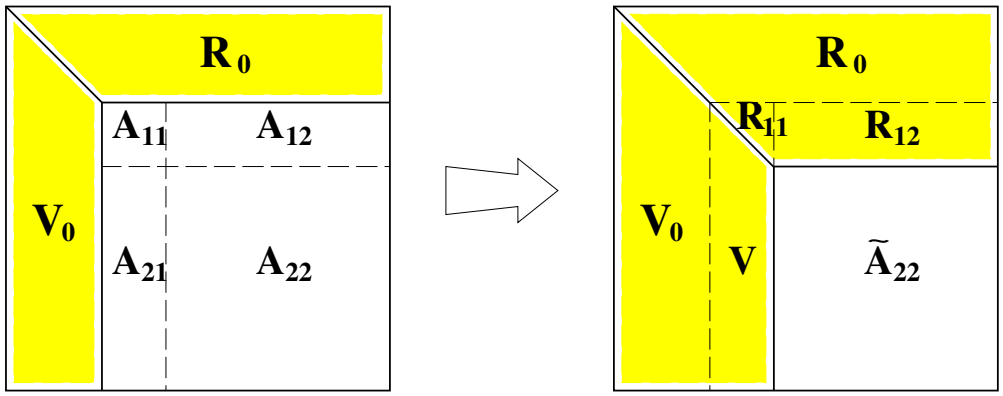


Figure 4: A snapshot of block QR factorization. During the computation, the sequence of the Householder vectors V is computed, and the row panel R_{11} and R_{12}, and the trailing submatrix A_{22} are updated.

A QR factorization is performed on the first $m \times n_b$ panel of A (i.e., A_1). In practice, Q is computed by applying a series of Householder transformations to A_1 of the form, $H_i = I - \tau_i v_i v_i^T$ where $i = 1, \cdots, n_b$. The vector v_i is of length m with 0's for the first $i-1$ entries and 1 for the i-th entry, and $\tau_i = 2/(v_i^T v_i)$. During the QR factorization, the vector v_i overwrites the entries of A below the diagonal, and τ_i is stored in a vector. Furthermore, it can be shown that $Q = H_1 H_2 \cdots H_{n_b} = I - VTV^T$, where T is $n_b \times n_b$ upper triangular and the i-th column of V equals v_i. This is indeed a block version of the QR factorization, and is rich in matrix-matrix operations.

The block equation can be rearranges as

$$\tilde{A}_2 = \begin{pmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{pmatrix} \Leftarrow \begin{pmatrix} R_{12} \\ R_{22} \end{pmatrix} = Q^T A_2 = \left(I - VT^T V^T\right) A_2 .$$

A snapshot of the block QR factorization is shown in Figure 4. During the computation, the sequence of the Householder vectors V is computed, and the row panel R_{11} and R_{12}, and the trailing submatrix A_{22} are updated. The factorization can be done by recursively applying the steps outlined above to the $(m - n_b) \times (n - n_b)$ matrix $\tilde{A}_{22}$.

The computation of the above steps of the LAPACK routine, `DGEQRF`, involves the following operations:

1. `DGEQR2`: Compute the QR factorization on an $m \times n_b$ panel of A (i.e., A_1)
   - [Repeat n_b times ($i = 1, \cdots, n_b$)]
     - `DLARFG`: generate the elementary reflector v_i and τ_i
     - `DLARF`: update the trailing submatrix

$$\tilde{A} \Leftarrow H_i^T A = (I - \tau_i v_i v_i^T) A$$

2. `DLARFT`: Compute the triangular factor T of the block reflector Q

3. `DLARFB`: Apply Q^T to the rest of the matrix from the left

$$\tilde{A}_2 \Leftarrow \begin{pmatrix} \tilde{R}_{12} \\ \tilde{R}_{22} \end{pmatrix} = Q^T A_2 = \left(I - VT^T V^T\right) A_2$$

   - `DGEMM`: $W \Leftarrow V^T A_2$
   - `DTRMM`: $W \Leftarrow T^T W$
   - `DGEMM`: $\tilde{A}_2 \Leftarrow \begin{pmatrix} \tilde{R}_{12} \\ \tilde{R}_{22} \end{pmatrix} = A_2 - VW$

The corresponding steps of the ScaLAPACK routine, `PDGEQRF`, are as follows:

1. `PDGEQR2`: A column of processes performs the QR factorization on an $m \times n_b$ panel of A (i.e., A_1)

   - [Repeat n_b times ($i = 1, \cdots, n_b$)]
     - `PDLARFG`: generate elementary reflector v_i and τ_i
     - `PDLARF`: update the trailing submatrix

2. `PDLARFT`: A column of processes, which has a sequence of the Householder vectors V, computes T.

3. `PDLARFB`: Apply Q^T to the rest of the matrix from the left

   - `PDGEMM`: The column of blocks V is broadcast rowwise and then saved in other processes. The transpose of V is locally multiplied by A_2, then the products are added to one row of processes ($W \Leftarrow V^T A_2$).
   - `PDTRMM`: T is broadcast rowwise and multiplied with the sum ($W \Leftarrow T^T W$).
   - `PDGEMM`: The row of blocks W is broadcast columnwise. Now, processes have their own portions of V and W, then they update the local portions of the matrix A ($\tilde{A}_2 \Leftarrow \begin{pmatrix} \tilde{R}_{12} \\ \tilde{R}_{22} \end{pmatrix} = A_2 - VW$).

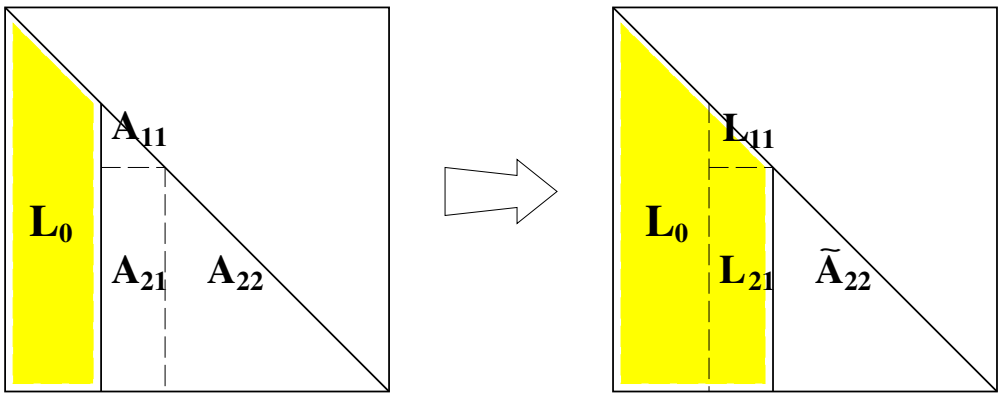


Figure 5: A snapshot of block Cholesky factorization shows how the column panel L (L_{11} and L_{21}) is computed and how the trailing submatrix A_{22} is updated.

3.3. Cholesky Factorization

Cholesky factorization factors an $N \times N$, symmetric, positive-definite matrix A into the product of a lower triangular matrix L and its transpose, i.e., $A = LL^T$ (or $A = U^T U$, where U is upper triangular). It is assumed that the lower triangler portion of A is stored in the lower triangle of a two-dimensional array and that the computed elements of L overwrite the given elements of A. At the k-th step, we partition the $n \times n$ matrices A, L, and L^T, and write the system as

$$\begin{aligned}
\begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} &= \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{pmatrix} \\
&= \begin{pmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{pmatrix}
\end{aligned}$$

where the block A_{11} is $n_b \times n_b$, A_{21} is $(n - n_b) \times n_b$, and A_{22} is $(n - n_b) \times (n - n_b)$. L_{11} and L_{22} are lower triangular.

The block-partitioned form of Cholesky factorization may be inferred inductively as follows. If we assume that L_{11}, the lower triangular Cholesky factor of A_{11}, is known, we can rearrange the block equations,

$$\begin{aligned}
L_{21} &\Leftarrow A_{21}(L_{11}^T)^{-1}, \\
\tilde{A}_{22} &\Leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T.
\end{aligned}$$

A snapshot of the block Cholesky factorization algorithm in Figure 5 shows how the column panel L (L_{11} and L_{21}) is computed and how the trailing submatrix A_{22} is updated. The factorization can be done by recursively applying the steps outlined above to the $(n - n_b) \times (n - n_b)$ matrix $\tilde{A}_{22}$.

In the right-looking version of the LAPACK routine, the computation of the above steps involves the following operations:

1. `DPOTF2`: Compute the Cholesky factorization of the diagonal block, A_{11}.

   $$A_{11} \Rightarrow L_{11}L_{11}^T$$

2. `DTRSM`: Compute the subdiagonal block of L,

   $$L_{21} \Leftarrow A_{21}(L_{11}^T)^{-1}$$

3. `DSYRK`: Update the rest of the matrix,

   $$\tilde{A}_{22} \Leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T$$

The parallel implementation of the corresponding ScaLAPACK routine, `PDPOTRF`, proceeds as follows:

1. `PDPOTF2`: A process P_i, which has the $n_b \times n_b$ diagonal block A_{11}, performs the Cholesky factorization of A_{11}.

   - P_i performs $A_{11} \Rightarrow L_{11}L_{11}^T$, and sets a flag if A_{11} is not positive definite.
   - P_i broadcasts the flag to all other processes so that the computation can be stopped if A_{11} is not positive definite.

2. `PDTRSM`: L_{11} is broadcast along a column of the processes, which compute the column of blocks of L_{21}.

3. `PDSYRK`: the column of blocks L_{21} is broadcast rowwise and then transposed. Now, processes have their own portions of L_{21} and L_{21}^T. They update their local portions of the matrix A_{22}.

4. Results and Discussion

We have outlined the basic parallel implementation of the three factorization routines. In this section, we describe a little more detail of the parallel implementation of the routines and performance results on the Intel iPSC/860, Touchstone Delta, and Paragon systems. Further we have investigated possible variations of the routines for the better performance.

The Intel iPSC/860 is a parallel architecture with up to 128 processing nodes. Each node consists of an i860 processor with 8 Mbytes of memory. The system is interconnected with a hypercube structure. The Delta system contains 512 i860-based computational nodes with 16 Mbytes /node, connected with a 2-D mesh communication network. The Intel Paragon located at Oak Ridge National Laboratory has 512 computational nodes, interconnected with a 2-D

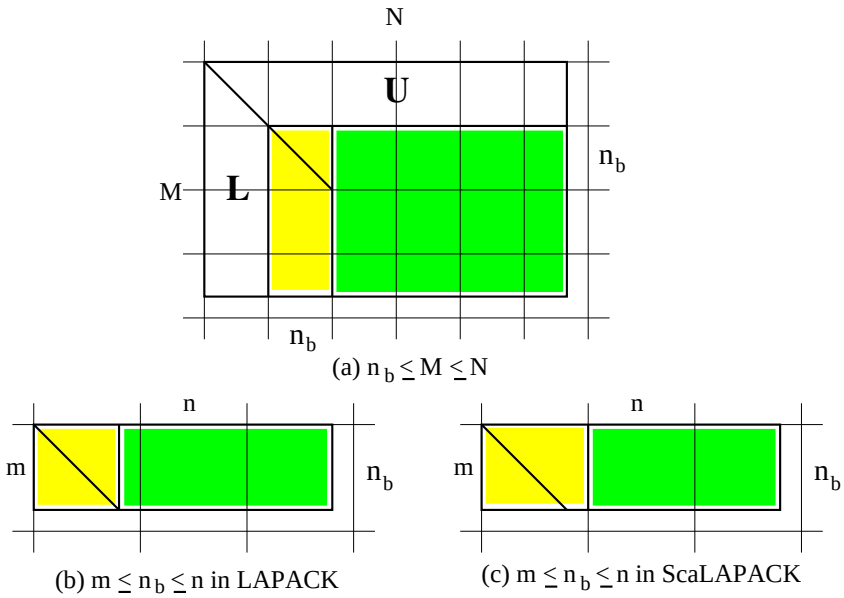


Figure 6: A snapshot of the block LU factorization when the matrix is a fat matrix ($M < N$)

mesh. Each node has 32 Mbytes of memory and two i860XP processors, one for computation and the other for communication. The Intel iPSC/860 and Delta machines both use the same 40MHz i860 processor, but the Delta has a higher communication bandwidth. Significantly higher performance can be attained on the Paragon system, since it uses the faster 50 MHz i860XP processor and has a larger communication bandwidth.

On each node all computation was performed in double precision arithmetic, using assembly-coded BLAS (Level 1, 2, and 3), provided by Intel. Communication was performed using the BLACS package, customized for the Intel systems. Most computation by the BLAS and communication by the BLACS are hidden within the PBLAS.

The optimal block size of a routine could be determined by the algorithm itself and characteristics of the target computer system, such as the ratio of computation speed over communication speed and the process mesh aspect ratio of P/Q. The block size, n_b, of the routines was selected to produce the best performance of the routines for the given target machines. The numbers of floating point operations for an $N \times N$ matrix were assumed to be $2/3\,N^3$ for the LU factorization, $4/3\,N^3$ for the QR factorization, and $1/3\,N^3$ for the Cholesky factorization.

4.1. LU Factorization

In LAPACK, the block size can be arbitrarily chosen to achieve optimal performance of a routine. But with the block cyclic data distribution in ScaLAPACK, the block size affects how the matrix is distributed over the 2-D process grid, and hence impacts load balance and

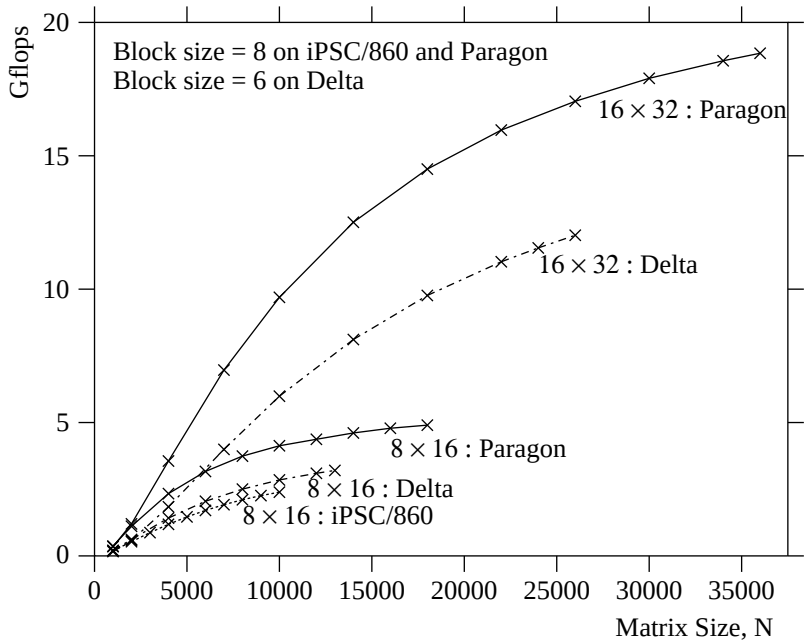


Figure 7: Performance of the LU factorization on the Intel iPSC/860, Delta, and Paragon

communication overhead. In this section we investigate how the block distribution affects the implementation of the ScaLAPACK routine.

In LAPACK, the block size can be arbitrarily chosen to achieve optimal performance of a routine. But with the block cyclic data distribution in ScaLAPACK, it has a physical meaning how the matrix is distributed over the 2-D process grid, and it affects load balance and communication overhead. In this section we investigate the effects of the physical meaning of the block size on the implementation of the ScaLAPACK.

Figure 6 (a) shows a snapshot of the block LU factorization when the matrix is a fat matrix ($M < N$). As explained in Section 3.1, `DGETF2` (or `PDGETF2`) applies the LU factorization on the $m \times n'$ column panel of A, where $n' \leq n_b$. Let us consider the computation of the last row of blocks carefully, where $m \leq n_b \leq n$.

`DGETF2` takes n' as MIN(m, n, n_b), and applies the factorization on the $m \times m$ square portion of the matrix (lightly shaded area), as shown in Figure 6 (b). The rest of the matrix (darkly shaded area) is updated later with `DTRSM`. However, it is assumed that the level-3 PBLAS routine, `PDTRSM`, cannot deal with a matrix starting from the middle of the block [7]. As illustrated in Figure 6 (c), `PDGETF2` in ScaLAPACK takes n' as MIN(n, n_b), and it computes the factorization on $m \times n_b$ portion of the matrix. The rest of the matrix satisfies the preassumption of using `PDTRSM`. Thus in ScaLAPACK, computations are generally aligned with block boundaries.

Figure 7 shows the performance of the ScaLAPACK LU factorization routine on the Intel iPSC/860, the Delta, and the Paragon in Gflops (gigaflops per second) as a function of number of processes. The optimal block size on the iPSC/860 and the Paragon was 8, and on the Delta was 6, and the best performance was attained with a process aspect ratio, $1/4 \leq P/Q \leq 1/2$. The LU routine attained 2.4 Gflops for a matrix size of $N = 10000$ on the iPSC/860; 12.0 Gflops for $N = 26000$ on the Delta; and 18.8 Gflops for $N = 36000$ on the Paragon.

The LU factorization routine is a little more complicated than the other routines because it requires column pivoting. In other words, many possible different implementations exist. We describe briefly other possible variations, which will destroy the modularity and simplicity of the implementation, but attain a slightly better performance.

In the unblocked LU factorization routine (`PDGETF2`), after finding the maximum value of the i-th column (`PDAMAX`), the i-th row will be exchanged with the pivot row containing the maximum value. Then the new i-th row is broadcast columnwise ($(n_b - i)$ elements) in `PDGER`. Instead, the communications of `PDLASWP` and `PDGER` can be combined. That is, the pivot row is directly broadcast to other processes in the column, and the pivot row is replaced with the i-th row later.

The column of processes, which has the $m \times n_b$ column panel of A (i.e., A_{11} and A_{21}), applies interchanges twice in order not to swap the data in the column panel (`PDLASWP`). These two separate communication processes also can be combined.

Finally, after completing the factorization of the column panel (`PDGETF2`), the column of processes, which has the column panel, broadcasts rowwise the pivot information for `PDLASWP`, L_{11} for `PDTRSM`, and L_{21} for `PDGEMM`. It is possible to combine the three messages to save the number of communications (or combine L_{11} and L_{21}), and broadcast rowwise the combined message.

It takes a non-negligible time to broadcast the column panel of L across the process template. It is possible to increase the overlap of communication with computation by broadcasting each column rowwise as soon as they are evaluated, rather than broadcasting all of the panel across after factoring it. With these modified communication schemes, the performance of the routine will be increased, but in our experiments we have found the improvement to be less than 5 %.

4.2. QR Factorization

It is required to compute an Euclidean norm of the vector, $A_{:i}$, to get the elementary Householder vector v_i. The sequential LAPACK routine, `DLARFG`, calls the Level-1 BLAS routine, `DNRM2`, which computes the norm without causing underflow or overflow problems. In the corresponding parallel ScaLAPACK routine, `PDLARFG`, each process in the column of processes, which holds the vector, $A_{:i}$, computes the global norm safely using the `PDNRM2` routine.

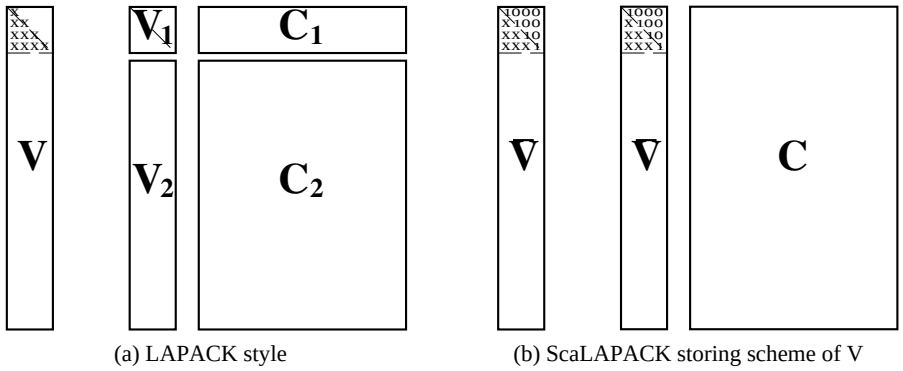


Figure 8: The storage scheme of the lower trapezoidal matrix V in ScaLAPACK QR factorization. By broadcasting $\bar{V}$, which is a copy of V, but with 0's on the upper triangular and 1's on the main diagonal, the computation involving V can be done in one step. (a) $V^T \cdot C = V_1^T \cdot C_1$ (`DTRMM`) $+ V_2^T \cdot C_2$ (`DGEMM`) (b) $V^T \cdot C = \bar{V} \cdot C$ (`DGEMM`)

The triangular factor T of a block Householder reflector V can be computed inside of `PDGEQR2`, and the routine can generate T directly, instead of τ. But keeping T might cause problems if the block size is subsequently changed by redistributing the matrix. However a separate routine `PDLARFT` needs only one communication over a column of processes to compute T on the fly, so we chose the LAPACK implementation style to compute T, i. e., we store τ and V, and generate T when necessary.

The $m \times n_b$ lower trapezoidal part of V, which is a sequence of the n_b Householder vectors, will be accessed in the form,

$$V = \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}$$

where V_1 is $n_b \times n_b$ unit lower triangular, and V_2 is $(m - n_b) \times n_b$. In the sequential routine, the multiplication involving V is divided into two steps: `DTRMM` with V_1 and `DGEMM` with V_2. However, in the parallel implementation, V will be broadcast rowwise to other processes, and all columns of processes have their own copies of V. The upper triangular part of V (including the main diagonal) will not be accessed by the other columns of processes. By sending $\bar{V}$, which is a copy of V, but with 0's on the upper triangle and 1's on the main diagonal, the multiplications involving $\bar{V}$ can be done in one step (`DGEMM`) as illustrated in Figure 8. This one step multiplication not only simplifies the implementation of the routine (`PDLARFB`), but also increases the overall performance of the routine (`PDGEQRF`).

Figure 9 shows the performance of the QR factorization routine on the Intel family of concurrent computers. The optimal block size of $n_b = 6$ was used on all the machines. Best performance was attained with an aspect ratio of $1/4 \leq P/Q \leq 1/2$. The highest performances of 3.1 Gflops for $N = 10000$ was obtained on the iPSC/860; 14.6 Gflops for $N = 26000$ on the

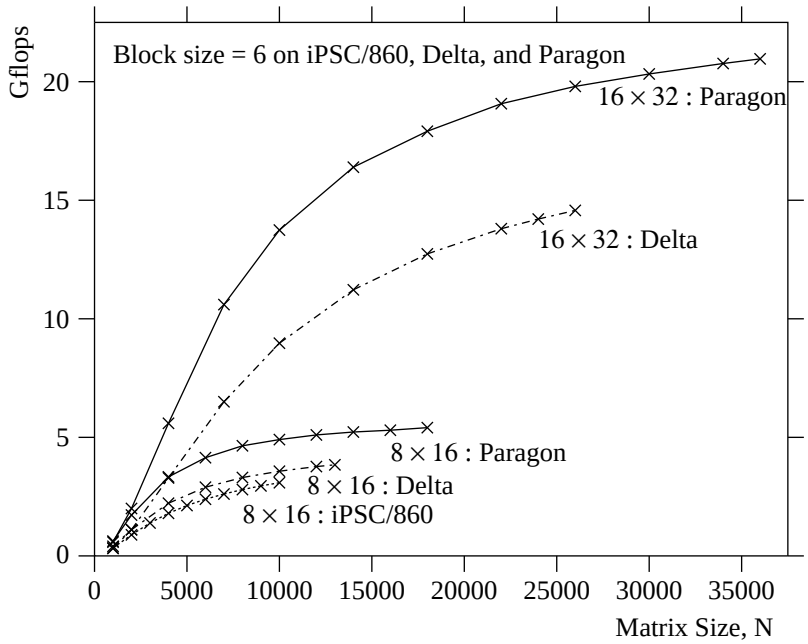


Figure 9: Performance of the QR factorization on the Intel iPSC/860, Delta, and Paragon

Delta; and 21.0 Gflops for $N = 36000$ on the Paragon.

4.3. Cholesky Factorization

The `PDSYRK` routine performs rank-n_b updates on an $(n - n_b) \times (n - n_b)$ symmetric matrix A_{22} with an $(n - n_b) \times n_b$ column of blocks L_{21}. After broadcasting L_{21} rowwise and transposing it, each process updates its own portion of A_{22} with its own portion of L_{21} and L_{21}^T. The globally lower triangular matrix A_{22} is not stored in the lower triangular form in the local processes as shown in Figure 10, thus it is complicated to update. The simplest way to do this is to repeatedly update one column of blocks of A_{22}; but if the block size is small, this updating process will not be efficient.

It is possible, and more efficient, to update several column blocks at a time. It is desirable to compute a multiple of LCM/Q blocks simultaneously since the processes easily determine their own physical data distribution of the lower triangular matrix of A, where LCM is the least common multiple of P and Q. Figures 10 (b) and (c) show how to update 2 ($= LCM/Q$) and 4 ($= 2 \cdot LCM/Q$) columns of blocks of A at a time, respectively. In the argument list of the PDSYRK routine, `MULLEN` specifies an approximate length of multiplication to update A_{22} efficiently. The multiple factor is computed by $k = \lceil MULLEN/((LCM/Q) \cdot n_b) \rceil$, and $k \cdot (LCM/Q)$ columns of blocks are updated simultaneously inside of the routine. For details, see [7]. The optimum number is determined by processor characteristics as well as the size of

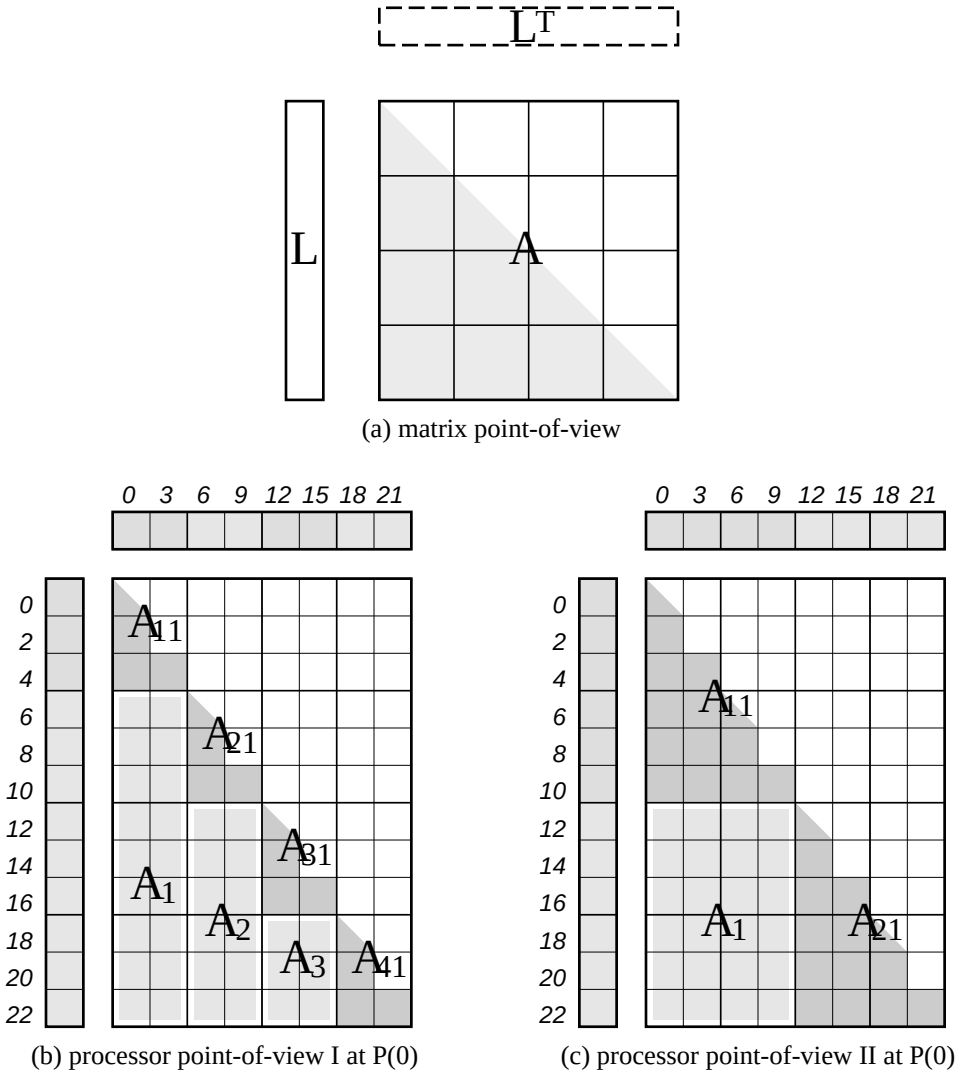


Figure 10: PDSYRK performs a rank-k update on a symmetric matrix. It is assumed that 24×24 blocks of A are distributed over a 2×3 process template. (a) A is a globally symmetric lower triangular matrix. (b) It is possible to update 2 ($= LCM/Q$) columns of blocks of A at a time. (c) It is more efficient to update 4 ($= 2 \cdot LCM/Q$) columns of blocks of A simultaneously.

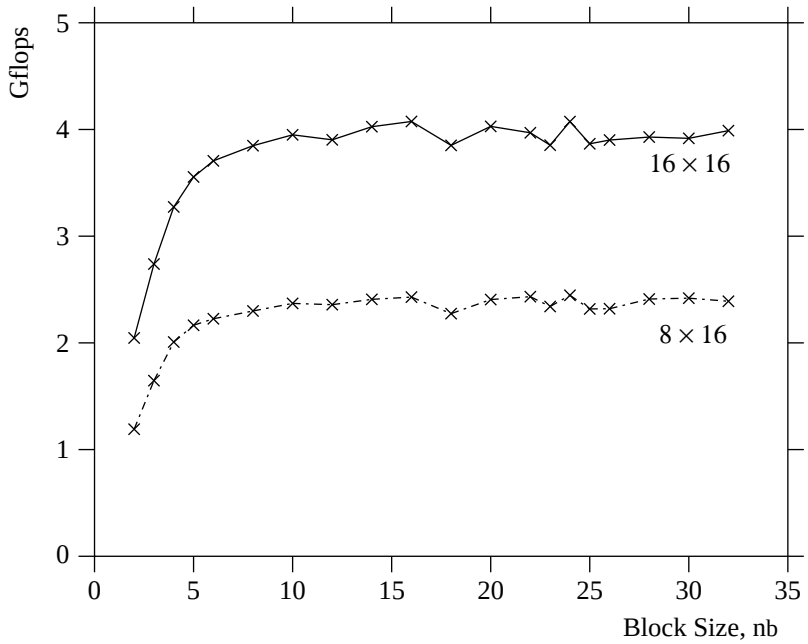


Figure 11: Performance of the Cholesky factorization as a function of the block size on the Intel Delta ($N = 10000$). The best performance was attained with the block size of $n_b = 24$ on 8×16 and 16×16 processes, but was quite insensitive to block size for $n_b \geq 8$.

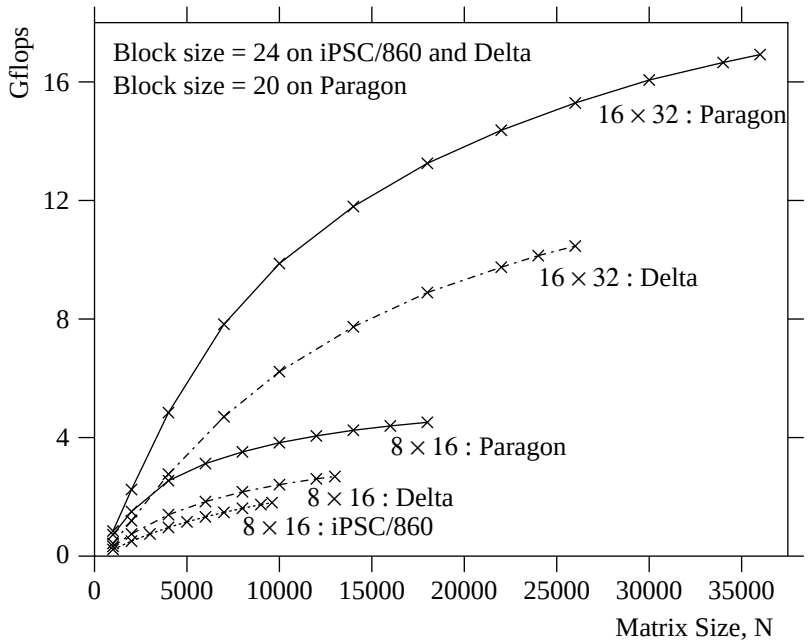


Figure 12: Performance of the Cholesky factorization on the Intel iPSC/860, Delta, and Paragon

the matrix and the block size. The optimum number was found to be about 40 on the Intel iPSC/860 and Delta computers.

The effect of the block size on the performance of the Cholesky factorization is shown in Figure 11 on 8×16 and 16×16 processors of the Intel Delta. The best performance was obtained at the block size of $n_b = 24$, but relatively good performance could be expected with the block size of $n_b \geq 6$, since the routine updates multiple column panels at a time. With $n_b = 24$, and $MULLEN = 40$, the routine updates 2 ($= \lceil MULLEN/n_b \rceil = \lceil 40/24 \rceil$, where $LCM = Q$) column panels simultaneously.

Figure 12 shows the performance of the Cholesky factorization routine. The best performance was attained with the aspect ratio of $1/2 \leq P/Q \leq 1$. The routine ran at 1.8 Gflops for $N = 9600$ on the iPSC/860; 10.5 Gflops for $N = 26000$ on the Delta; and 16.9 Gflops for $N = 36000$ on the Paragon.

If A is not positive definite, the Cholesky factorization should be terminated in the middle of the computation. As outlined in Section 3.3, a process P_i computes the Cholesky factor L_{11} from A_{11}. After computing L_{11}, P_i broadcasts a flag to other processes so that all processes stop the computation if A_{11} is not positive definite. If A is guaranteed to be positive definite, the process of broadcasting the flag can be skipped, and another performance increase can be expected.

5. Scalability and Conclusions

The performance of the three factorization routines on 128 nodes of the iPSC/860 and 512 nodes of the Delta and the Paragon were compared in Figures 13, 14, and 15, respectively. Generally the QR factorization routine has the best performance since the updating process of $Q^T A = (I - VTV^T)A$ is rich in matrix-matrix operation, and the number of floating point operations is the largest ($4/3\, N^3$). The Cholesky factorization involves operations on a symmetric matrix, and the total number of floating point operations ($1/3\, N^3$) is less than the other routines, thus its performance is poorer. The LU factorization routine seems to have more communication overhead since the routine contains column pivoting and row swapping operations. On the Delta, which has faster communication than the iPSC/860, the LU routine is slower than the Cholesky routine for small problem size ($N \leq 12000$).

The performance results in Figures 7, 9, and 12 can be used to assess the scalability of the factorization routines. In general, concurrent efficiency, ϵ, is defined as the concurrent speedup per process. That is, for the given problem size, N, on the number of processes used, N_p,

$$\epsilon(N, N_p) = \frac{1}{N_p} \frac{T_s(N)}{T_p(N, N_p)}$$

where $T_p(N, N_p)$ is the time for a problem of size N to run on N_p processes, and $T_s(N)$ is the

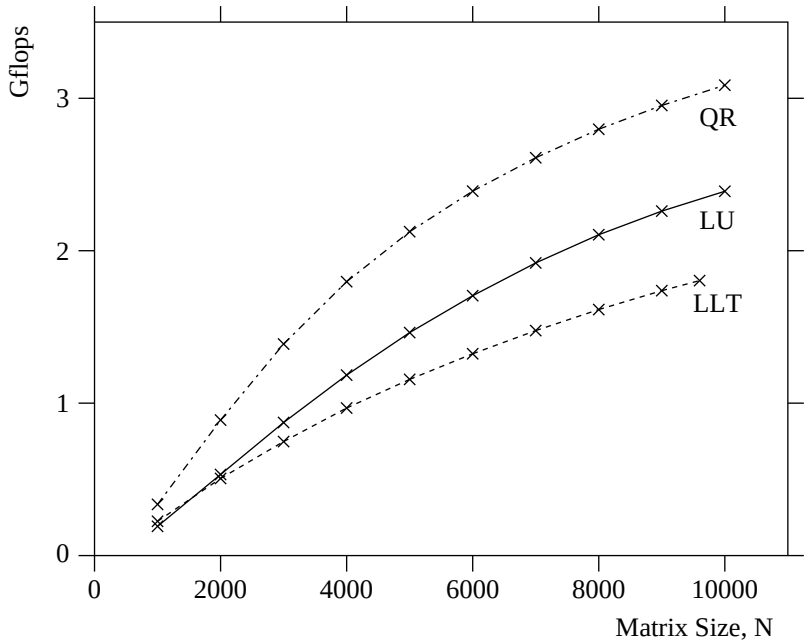


Figure 13: Performance of factorization routines on Intel iPSC/860 (128 nodes)

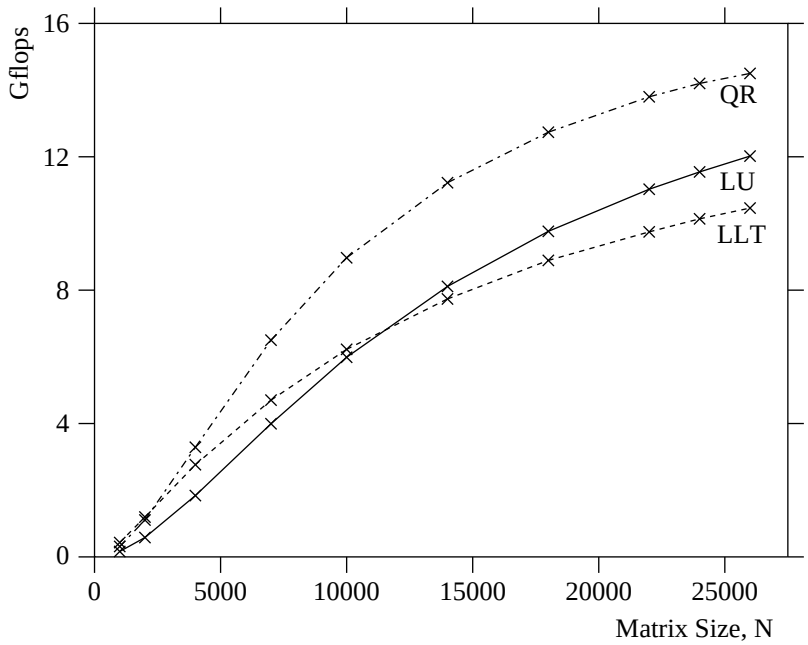


Figure 14: Performance of factorization routines on Intel Delta (512 nodes)

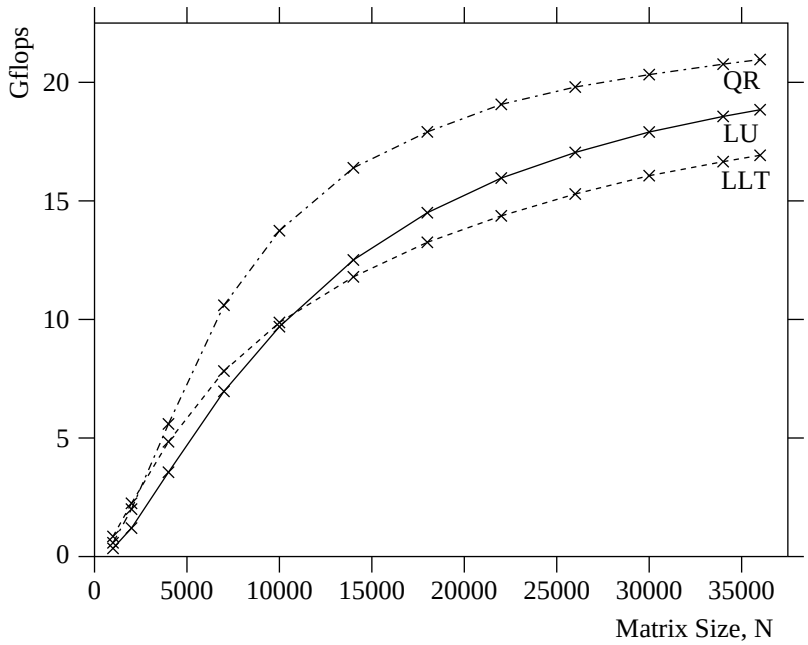


Figure 15: Performance of factorization routines on Intel Paragon (512 nodes)

time to run on one process using the best sequential algorithm. Another approach to investigate the efficiency is to see how the performance per process degrades as the number of processes increases for a fixed grain size, i. e., by plotting isogranularity curves in the (N_p, G) plane, where G is the performance. Since

$$G \propto \frac{T_s(N)}{T_p(N, N_p)} = N_p \, \epsilon(N, N_p),$$

the scalability for memory-constrained problems can readily be accessed by the extent to which the isogranularity curves differ from linearity.

Figures 16, 17, and 18 show the isogranularity plots for the ScaLAPACK factorization routines on the iPSC/860, the Delta, and the Paragon, respectively. The matrix size per process is fixed at 5 Mbytes on the iPSC/860, 9 Mbytes on the Delta, and 5 and 20 Mbytes on the Paragon. The linearity of the plots in the figures indicates that the ScaLAPACK routines have good scalability characteristics on these systems.

We have demonstrated that the LAPACK factorization routines can be parallelized fairly easily to the corresponding ScaLAPACK routines with a small set of low-level modules, namely the sequential BLAS, the BLACS, and the PBLAS. The PBLAS are particularly useful for developing and implementing a parallel dense linear algebra library relying on the block cyclic data distribution. In general, the Level 2 and 3 BLAS routines in the LAPACK code can be replaced on a one-for-one basis by the corresponding PBLAS routines. Parallel routines implemented with the PBLAS have good performance, since the computation performed by

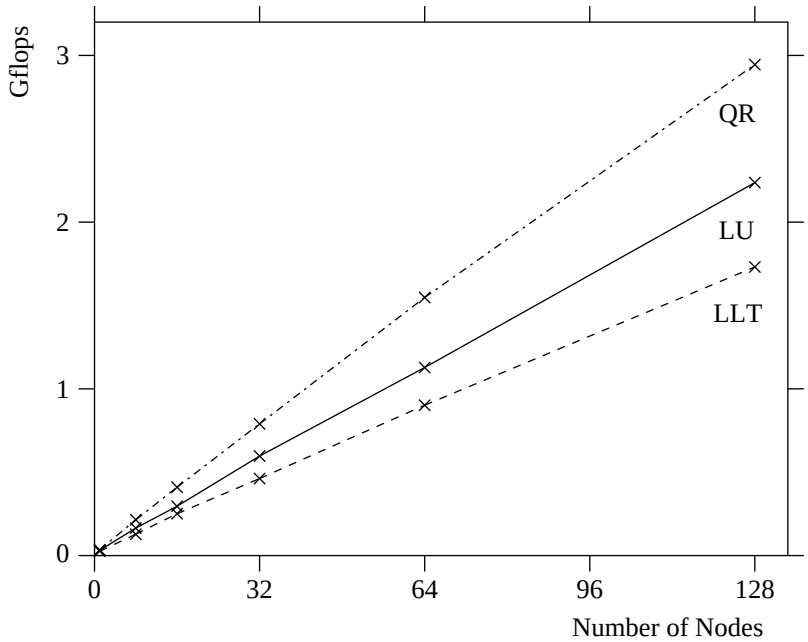


Figure 16: Scalability of factorization routines on the Intel iPSC/860 (5 Mbytes/node)

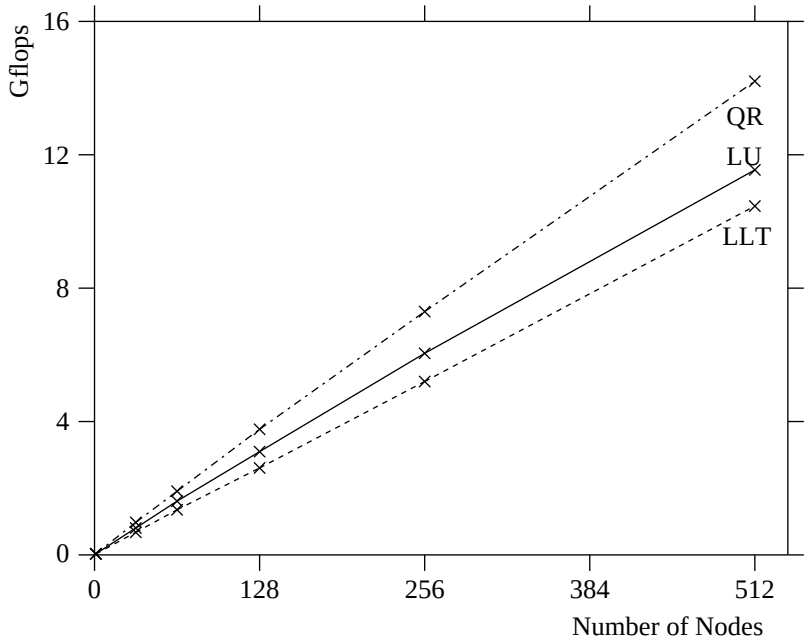


Figure 17: Scalability of factorization routines on the Intel Delta (9 Mbytes/node)

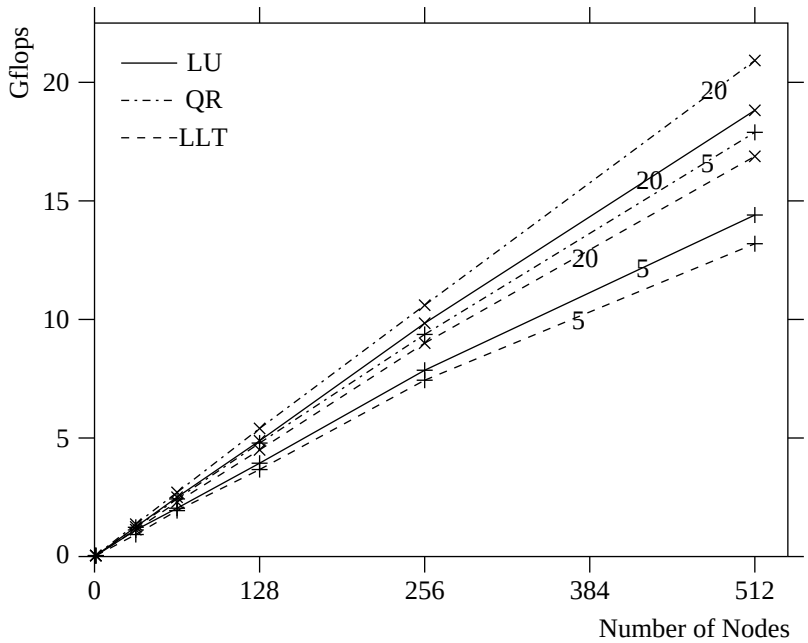


Figure 18: Scalability of factorization routines on the Intel Paragon (5, 20 Mbytes/node)

each process within PBLAS routines can be itself performed using the assembly-coded sequential BLAS.

There is a tradeoff between performance and software design considerations, such as modularity and clarity, in designing and implementing software libraries. As described in Section 4.1, it is possible to combine communications to reduce the communication costs in several places, such as in factorizing the column panel (`PDLASWP` and `PDGER`), in swapping data in the column panel (`PDGER`), and in broadcasting the column panel (`PDTRSM` and `PDGEMM`), and to replace the high level routines, such as the PBLAS, by calls to the lower level routines, such as the sequential BLAS and the BLACS. However, we have found that the performance gain is too small to justify the resulting loss of software modularity.

We have shown that the ScaLAPACK factorization routines have good performance and scalability on the Intel iPSC/860, the Delta, and the Paragon systems. Similar studies will be performed on recent machines, including the TMC CM-5, the Cray T3D, and the IBM SP1 and SP2.

Currently the ScaLAPACK library includes not only the LU, QR, and Cholesky factorization routines, but also factorization solvers, routines to refine the solutions to reduce error, and routines to estimate the reciprocal of the condition number. The ScaLAPACK routines are currently available through *netlib* only for double precision real data, but in the near future, we intend to release routines for other numeric data types, such as single precision real and complex, and double precision complex. To obtain the routines, and the ScaLAPACK Reference Manual [9], send the message "`send index from scalapack`" to `netlib@ornl.gov`.

Acknowledgements

This research was performed in part using the Intel iPSC/860 hypercube and the Paragon computers at Oak Ridge National Laboratory, and in part using the Intel Touchstone Delta system operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to the Delta system was provided through the Center for Research on Parallel Computing.

6. References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, 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] J. Choi, J. J. Dongarra, R. Pozo, D. C. Sorensen, and D. W. Walker. CRPC Research into Linear Algebra Software for High Performance Computers. *International Journal of Supercomputing Applications*, 8(2):99–118, Summer 1994.

[5] 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 Fourth Symposium on the Frontiers of Massively Parallel Computation (McLean, Virginia)*. IEEE Computer Society Press, Los Alamitos, California, October 19-21, 1992.

[6] J. Choi, J. J. Dongarra, and D. W. Walker. Level 3 BLAS for Distributed Memory Concurrent Computers. In *Proceedings of Environment and Tools for Parallel Scientific Computing Workshop, (Saint Hilaire du Touvet, France)*, pages 17–29. Elsevier Science Publishers, September 7-8, 1992.

[7] J. Choi, J. J. Dongarra, and D. W. Walker. PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subprograms. *Submitted to Concurency: Practise and Experience*, 1994.

Also available on Oak Ridge National Laboratory Technical Reports, TM-12268, February, 1994.

[8] J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurency: Practise and Experience*, 1994. Accepted for publication. Also available on Oak Ridge National Laboratory Technical Reports, TM-12252, August, 1993.

[9] J. Choi, J. J. Dongarra, and D. W. Walker. ScaLAPACK Reference Manual: Parallel Factorization Routines (LU, QR, and Cholesky), and Parallel Reduction Routines (HRD, TRD, and BRD) (Version 1.0BETA). Technical Report TM-12471, Oak Ridge National Laboratory, February 1994.

[10] J. Choi, J. J. Dongarra, and D. W. Walker. The Design of a Parallel, Dense Linear Algebra Software Library: Reduction to Hessenburg, Tridiagonal, and Bidiagonal Form. *Submitted to SIAM J. of Sci. Stat. Computing*, 1994.

[11] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide.* SIAM Press, Philadelphia, PA, 1979.

[12] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 18(1):1–17, 1990.

[13] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1988.

[14] J. J. Dongarra and S. Ostrouchov. LAPACK Block Factorization Algorithms on the Intel iPSC/860. LAPACK Working Note 24, Technical Report CS-90-115, University of Tennessee, October 1990.

[15] A. Edelman. Large Dense Linear Algebra in 1993: The Parallel Computing Influence. *International Journal of Supercomputing Applications*, 7(2):113–128, Summer 1993.

[16] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[17] 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.* Springer-Verlag, Berlin, 1976. Vol.6 of Lecture Notes in Computer Science, 2nd Ed.