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 [§†]
L. 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

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

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

Abstract

This paper discusses the core factorization routines included in the ScaLAPACK library. These routines allow the factorization and solution of a dense system of linear equations via LU, QR, and Cholesky. They are implemented using a block cyclic data distribution, and are built using de facto standard kernels for matrix and vector operations (BLAS and its parallel counterpart PBLAS) and message passing communication (BLACS). 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 a significant loss in performance.

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

Key words. BLACS, distributed LAPACK, distributed memory, linear algebra, PBLAS, scalability

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) [15, 16] 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 the successor to LINPACK [14] and EISPACK [23]. 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 also use block-partitioned algorithms and be as compatible as possible with the LAPACK library for vector and shared memory computers. It is therefore called ScaLAPACK ("Scalable LAPACK") [6], and can be used to solve "Grand Challenge" problems on massively parallel, distributed-memory, concurrent computers [5, 18].

The Basic Linear Algebra Communication Subprograms (BLACS) [3] provide ease-of-use and portability for message-passing in parallel linear algebra applications. The Parallel BLAS (PBLAS) assume a block cyclic data distribution and are functionally an extended subset of the Level 1, 2, and 3 BLAS for distributed memory systems. They are based on previous work with the Parallel Block BLAS (PB-BLAS) [8]. The current model implementation relies internally on the PB-BLAS, as well as the BLAS and the BLACS. The ScaLAPACK routines consist of calls to the sequential BLAS, the BLACS, and the PBLAS modules. ScaLAPACK can therefore be ported 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. These routines have been studied on various parallel platforms by many other researchers [13, 19, 12]. We maintain compatibility between the ScaLAPACK codes and their LAPACK equivalents by isolating as much of the distributed memory operations as possible inside the PBLAS and ScaLAPACK auxiliary routines. Our goal is to simplify the implementation of complicated parallel routines while still maintaining good performance.

Currently the ScaLAPACK library contains Fortran 77 subroutines for the analysis and solution of systems of linear equations, linear least squares problems, and matrix eigenvalue problems. ScaLAPACK routines to reduce a real general matrix to Hessenberg or bidiagonal form, and a symmetric matrix to tridiagonal form are considered in [11].

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. Conclusions and future work are presented in Section 6.

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 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.

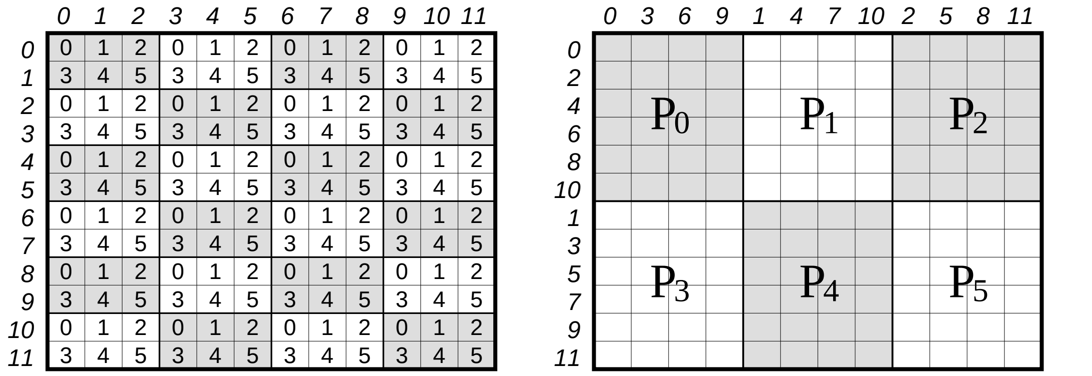
2.1. Block Cyclic Data Distribution

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, m_b, and n_b, where $P \times Q$ is the process grid and $m_b \times n_b$ 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 m_b, the block cyclic data distribution may be written as follows:

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

where $s = \lfloor m/m_b \rfloor$ and P is the number of processes. The distribution of a block-partitioned matrix can be regarded as the tensor product of two such mappings: one that distributes the



(a) block distribution over 2 x 3 grid. (b) data distribution from processor point-of-view.

Figure 1: Example of a block cyclic data distribution.

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 grid. The numbered squares represent blocks of elements, and the number indicates at which location in the process grid 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. Figure 1 (b) reflects the distribution from a process point-of-view. Each process has 6×4 blocks. 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 non-scattered decomposition (or pure block distribution) is just a special case of the cyclic distribution in which the block size is given by $m_b = \lceil M/P \rceil$ and $n_b = \lceil N/Q \rceil$. That is,

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

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

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

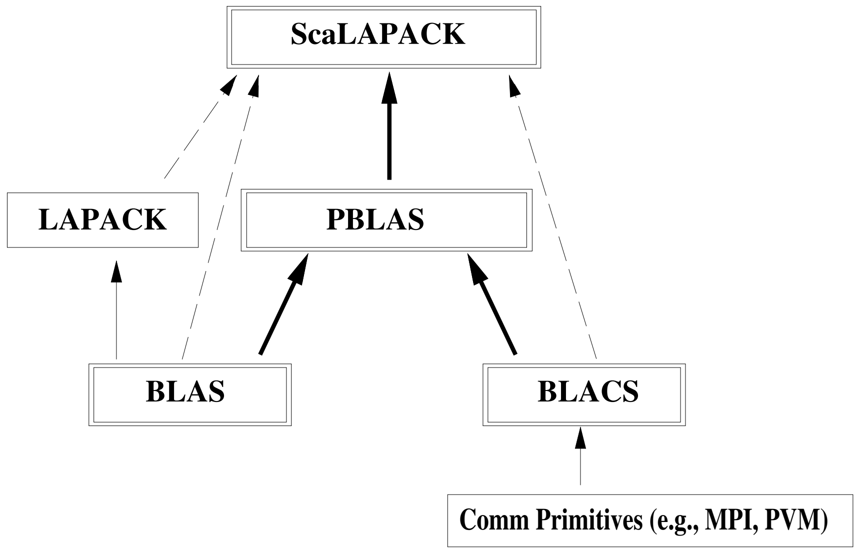


Figure 2: Hierarchical view of ScaLAPACK.

2.2. Building Blocks

The ScaLAPACK routines are composed 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 communication tasks. ScaLAPACK is portable 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 grid, but also collective communications on such grids, or within just a grid row or column.

Portable software for dense linear algebra on MIMD platforms may 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. We have implemented the BLACS for the Intel family of computers, the TMC CM-5, the IBM SP1 and SP2, and PVM. Several vendors are producing optimized versions of the BLACS (e.g. Cray, IBM, and Meiko). We plan to produce an MPI version of the BLACS in the near future.

The Parallel BLAS (PBLAS) are an extended subset of the BLAS for distributed memory computers and operate on matrices distributed according to a block cyclic data distribution scheme. These restrictions permit certain memory access and communication optimizations that would not be possible (or would be difficult) if general-purpose distributed Level 2 and Level 3 BLAS were used [7, 9].

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 computation 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 a message-passing version of LAPACK and is designed to be efficient across a wide range of architectures. The performance of the routines relies ultimately on the architectural characteristics of the machine. However, the codes are flexible in that they allow the tuning of certain parameters such as block size, and the size of the process grid. This flexibility ensures that the ScaLAPACK routines will be able to achieve good performance.

The ScaLAPACK routines, like their LAPACK equivalents, are designed to perform correctly for a wide range of inputs. Whenever practical, they behave appropriately when overflow and underflow problems are encountered or a routine is used incorrectly. Examples of error handling are given in Section 4.2 and Section 4.3.

3. Factorization Routines

In this section, we first briefly describe the sequential, block-partitioned versions of the dense LU, QR, and Cholesky factorization routines of the LAPACK library. Since we also wish to discuss the parallel factorizations, we describe the right-looking versions of the routines. The right-looking variants minimize data communication and distribute the computation across all processes [17]. After describing the sequential factorizations, the parallel versions will be discussed.

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 grid with a block cyclic distribution and a block size of $n_b \times n_b$ matching the block size of the algorithm. Thus, each n_b-wide column (or row) panel lies in one column (row) of the process grid.

In the LU, QR and Cholesky factorization routines, 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 where partial pivoting is performed over rows in a single column of the process grid 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 across processes in a single row of the process grid. The effect of this type of load imbalance can be minimized through the choice of P and Q.

3.1. LU Factorization

The LU factorization applies a sequence of Gaussian eliminations to form $A = PLU$, 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), U is upper triangular, and P is a permutation matrix, which is stored in a $\min(M, N)$ vector.

At the k-th step of the computation ($k = 1, 2, \cdots$), it is assumed that the $m \times n$ submatrix of $A^{(k)}$ ($m = M - (k-1) \cdot n_b$, $n = N - (k-1) \cdot n_b$) is to be partitioned as follows,

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

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 $A^{(k)}$ (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

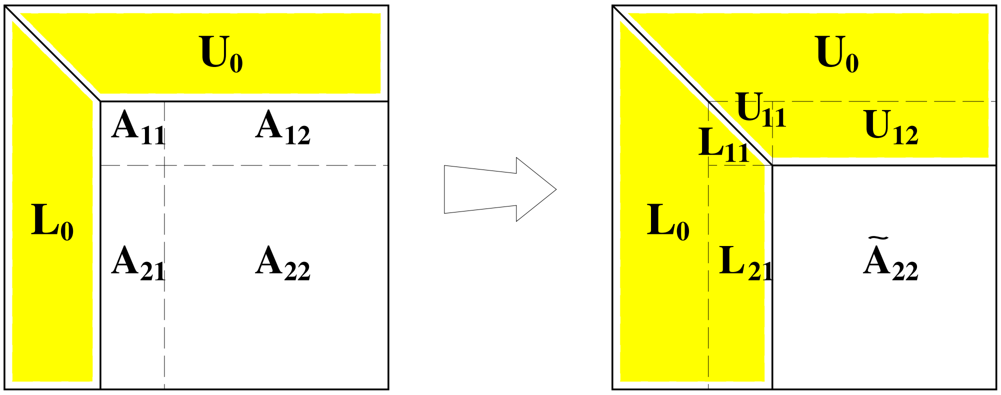


Figure 3: A snapshot of block LU factorization.

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. Later, row interchanges will be applied to L_0 and L_{21}.

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 row interchanges to the left and the right of the panel.
3. `DTRSM`: Compute the $n_b \times (n - n_b)$ row panel 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`: The current 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 holds the maximum
  - **PDSCAL**: scale the i-th column of the matrix
  - **PDGER**: broadcast the i-th row columnwise ($(n_b - i)$ elements) in the current column of processes and update the trailing submatrix
- Every process in the current process column broadcasts the same pivot information rowwise to all columns of processes.

2. **PDLASWP**: All processes apply row interchanges to the left and the right of the current panel.

3. **PDTRSM**: L_{11} is broadcast along the current row of processes, which converts the row panel A_{12} to U_{12}.

4. **PDGEMM**: The column panel L_{21} is broadcast rowwise across all columns of processes. The row panel U_{12} is broadcast columnwise down all rows of processes. Then, all 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 $A^{(k)}$ as

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

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 $A^{(k)}$, and A_2 is an $m \times (n - n_b)$ matrix, containing the last $(n - n_b)$ columns of $A^{(k)}$ (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.

A QR factorization is performed on the first $m \times n_b$ panel of $A^{(k)}$ (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

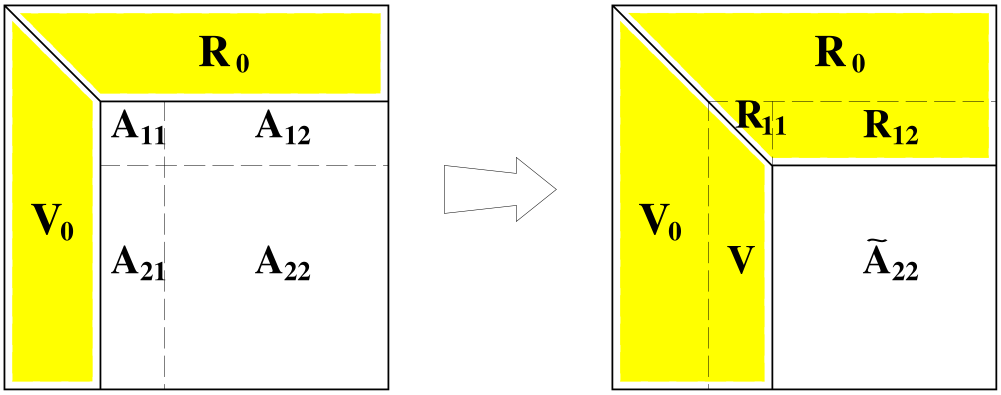


Figure 4: A snapshot of block QR factorization.

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 [4, 22], and is rich in matrix-matrix operations.

The block equation can be rearranged as

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

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 $A^{(k)}$ (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
       $\tilde{A}_1 \leftarrow H_i^T A_1 = (I - \tau_i v_i v_i^T) A_1$
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
   $$\tilde{A}_2 \leftarrow \begin{pmatrix} \tilde{R}_{12} \\ \tilde{R}_{22} \end{pmatrix} = Q^T A_2 = \left(I - VT^TV^T\right) A_2$$

- `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`: The current column of processes performs the QR factorization on an $m \times n_b$ panel of $A^{(k)}$ (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`: The current column of processes, which has a sequence of the Householder vectors V, computes T only in the current process row.
3. `PDLARFB`: Apply Q^T to the rest of the matrix from the left
   - `PDGEMM`: V is broadcast rowwise across all columns of processes. The transpose of V is locally multiplied by A_2, then the products are added to the current process row $(W \leftarrow V^T A_2)$.
   - `PDTRMM`: T is broadcast rowwise in the current process row to all columns of processes and multiplied with the sum $(W \leftarrow T^T W)$.
   - `PDGEMM`: W is broadcast columnwise down all rows of processes. Now, processes have their own portions of V and W, then they update the local portions of the matrix A_2 ($\tilde{A}_2 \leftarrow \begin{pmatrix} \tilde{R}_{12} \\ \tilde{R}_{22} \end{pmatrix} = A_2 - VW$).

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 triangular 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 $A^{(k)}$, $L^{(k)}$, and ${L^{(k)}}^T$, and write the system as

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

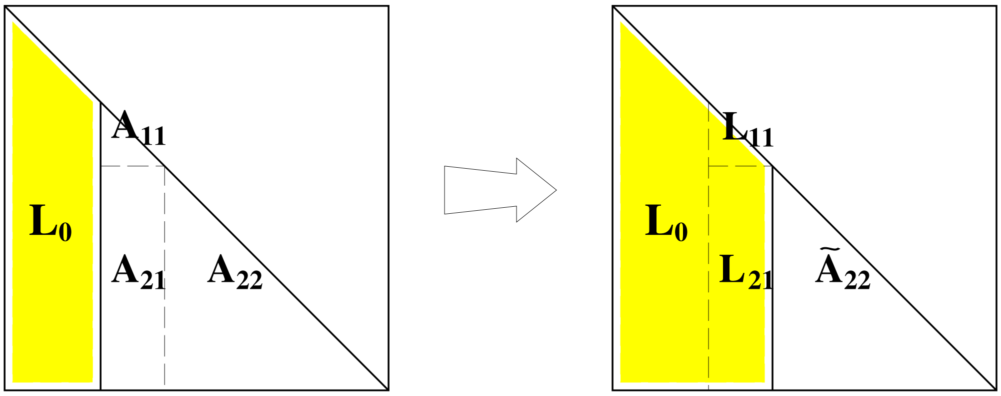


Figure 5: A snapshot of block Cholesky factorization.

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

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 $L^{(k)}$ (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 column panel L_{21},

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

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`: The 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 columnwise by P_i down all rows in the current column of processes, which computes the column of blocks of L_{21}.
3. `PDSYRK`: the column of blocks L_{21} is broadcast rowwise across all columns of processes 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. Performance Results

We have outlined the basic parallel implementation of the three factorization routines. In this section, we provide performance results on the Intel iPSC/860, Touchstone Delta, and Paragon systems. We also discuss specific implementation details to improve performance and possible variations of the routines which might yield 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 the Oak Ridge National Laboratory has 512 computational nodes, interconnected with a 2-D 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.

A good choice for the block size, $m_b \times n_b$, was determined experimentally for each factorization on the given target machines. For all performance graphs, results are presented for

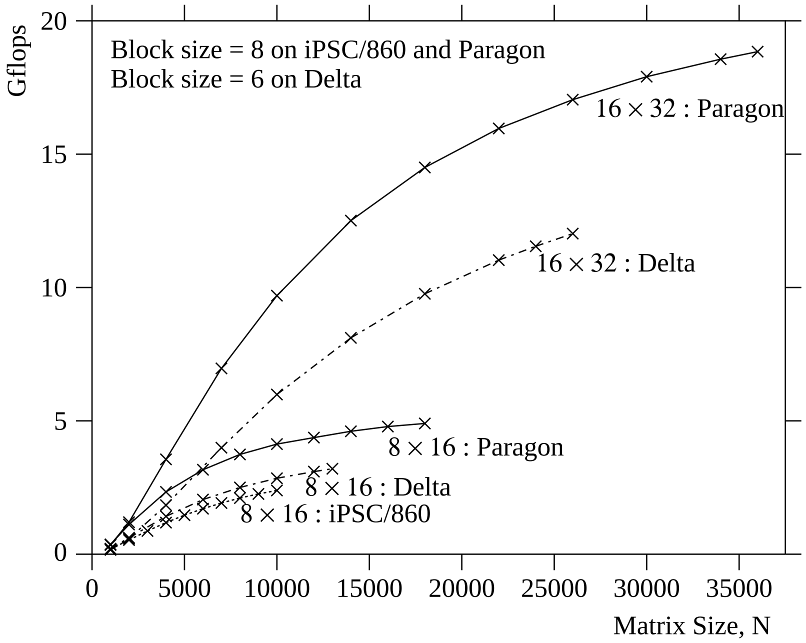


Figure 6: Performance of the LU factorization on the Intel iPSC/860, Delta, and Paragon.

square matrices with a square block size $m_b = n_b$. 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

Figure 6 shows the performance of the ScaLAPACK LU factorization routine on the Intel iPSC/860, the Delta, and the Paragon in Gflops (gigaflops or a billion floating point operations per second) as a function of the number of processes. The selected block size on the iPSC/860 and the Paragon was $m_b = n_b = 8$, and on the Delta was $m_b = n_b = 6$, 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 requires pivoting for numerical stability. Many different implementations of pivoting are possible. In the paragraphs below, we outline our implementation and some optimizations which we chose not to use in order to maintain modularity and clarity in the library.

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`. A slightly faster code may be obtained by combining the communications of `PDLASWP` and `PDGER`. That is, the pivot row is directly broadcast to other processes in the grid column, and the pivot row is replaced with the i-th row later.

The processes apply row interchanges (`PDLASWP`) to the left and the right of the column panel of A (i.e., A_{11} and A_{21}). These two row interchanges involve separate communications, which 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.

Notice that a non-negligible time is spent broadcasting the column panel of L across the process grid. It is possible to increase the overlap of communication to 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 may be increased, but in our experiments we have found the improvement to be less than 5 % and, therefore, not worth the loss of modularity.

4.2. QR Factorization

To obtain the elementary Householder vector v_i, the Euclidean norm of the vector, $A_{:i}$, is required. The sequential LAPACK routine, `DLARFG`, calls the Level-1 BLAS routine, `DNRM2`, which computes the norm while guarding against underflow and overflow. 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.

For consistency with LAPACK, we have chosen to store τ and V, and generate T when necessary. Although storing T might save us some redundant computation, we felt that consistency was more important.

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 is contained in one column of processes. Let $\bar{V}$ be a unit lower trapezoidal matrix containing the strictly lower trapezoidal part of V. $\bar{V}$ is broadcast rowwise to the other process columns so that every column of processes has its own

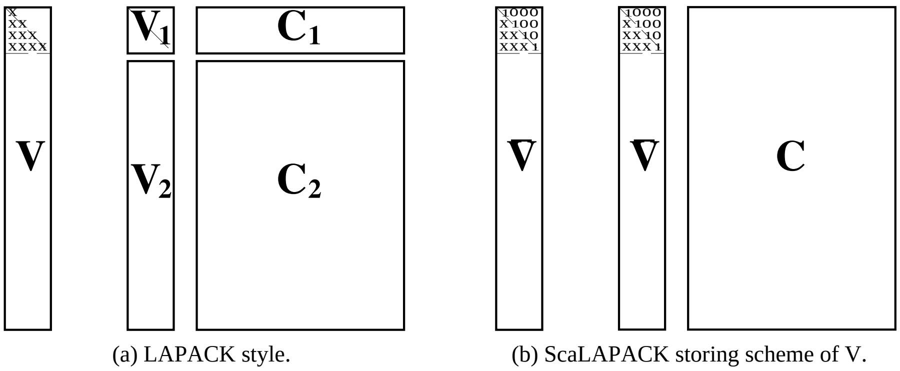


Figure 7: The storage scheme of the lower trapezoidal matrix V in ScaLAPACK QR factorization.

copy. This allows us to perform the operations involving $\bar{V}$ in one step (**DGEMM**), as illustrated in Figure 7, and not worry about the upper triangular part of V. This one step multiplication not only simplifies the implementation of the routine (**PDLARFB**), but may, depending upon the BLAS implementation, increase the overall performance of the routine (**PDGEQRF**) as well.

Figure 8 shows the performance of the QR factorization routine on the Intel family of concurrent computers. The block size of $m_b = n_b = 6$ was used on all of 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 Delta; and 21.0 Gflops for $N = 36000$ on the Paragon. Generally, the QR factorization routine has the best performance of the three factorizations 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 $(4/3\,N^3)$.

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 local portion of A_{22} with its own copy of L_{21} and L_{21}^T. The update is complicated by the fact that the globally lower triangular matrix A_{22} is not necessarily stored in the lower triangular form in the local processes. For details see [8]. The simplest way to do this is to repeatedly update one column of blocks of A_{22}. However, if the block size is small, this updating process will not be efficient. It is more efficient to update several blocks of columns at a time. The PBLAS routine, PDSYRK efficiently updates A_{22} by combining several blocks of columns at a time. For details, see [8].

The effect of the block size on the performance of the Cholesky factorization is shown in Figure 9 on 8×16 and 16×16 processors of the Intel Delta. The best performance was obtained

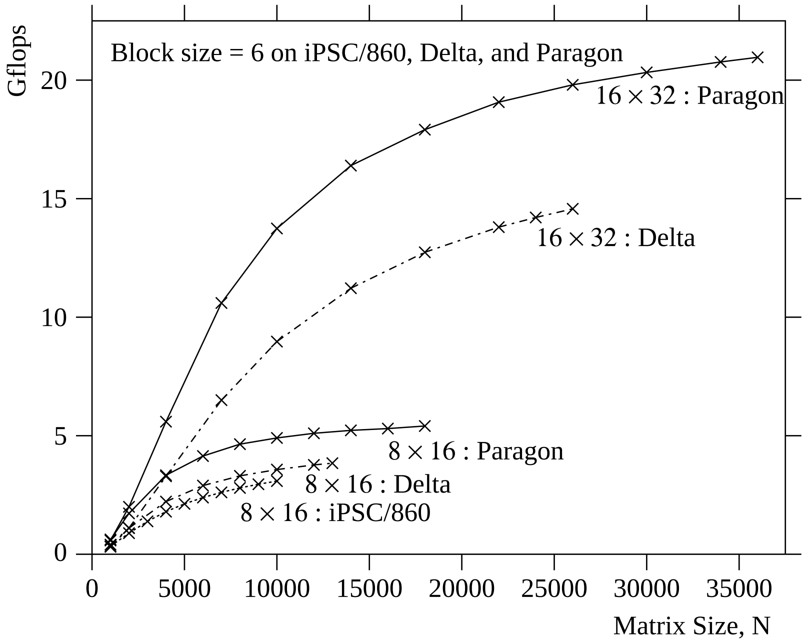


Figure 8: Performance of the QR factorization on the Intel iPSC/860, Delta, and Paragon.

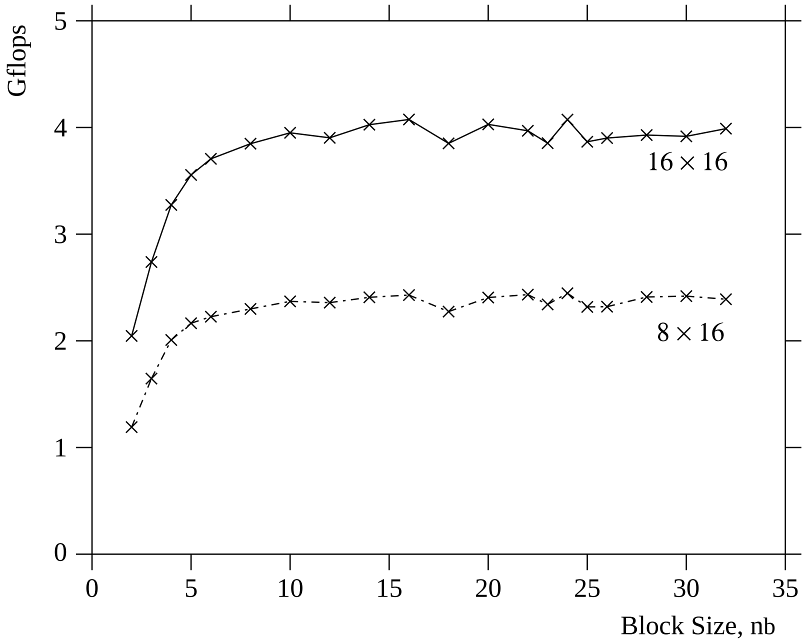


Figure 9: Performance of the Cholesky factorization as a function of the block size on 8×16 and 16×16 processes of the Intel Delta ($N = 10000$).

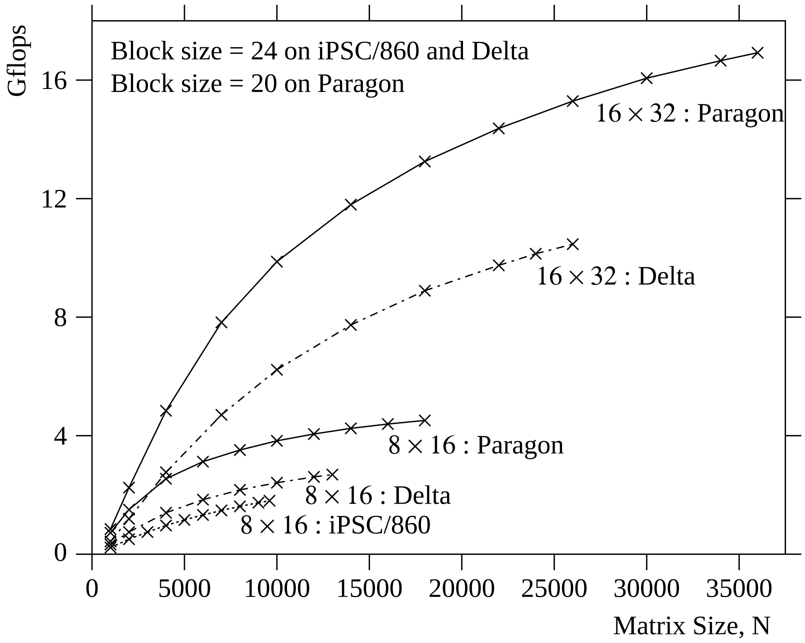


Figure 10: Performance of the Cholesky factorization on the Intel iPSC/860, Delta, and Paragon.

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.

Figure 10 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. Since it requires fewer floating point operations $(1/3\, N^3)$ than the other factorizations, it is not surprising that its flop rate is relatively poor.

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}, process P_i broadcasts a flag to all other processes to 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, leading to a corresponding increase in performance.

5. Scalability

The performance results in Figures 6, 8, and 10 can be used to assess the scalability of the factorization routines. In general, concurrent efficiency, E, is defined as the concurrent speedup

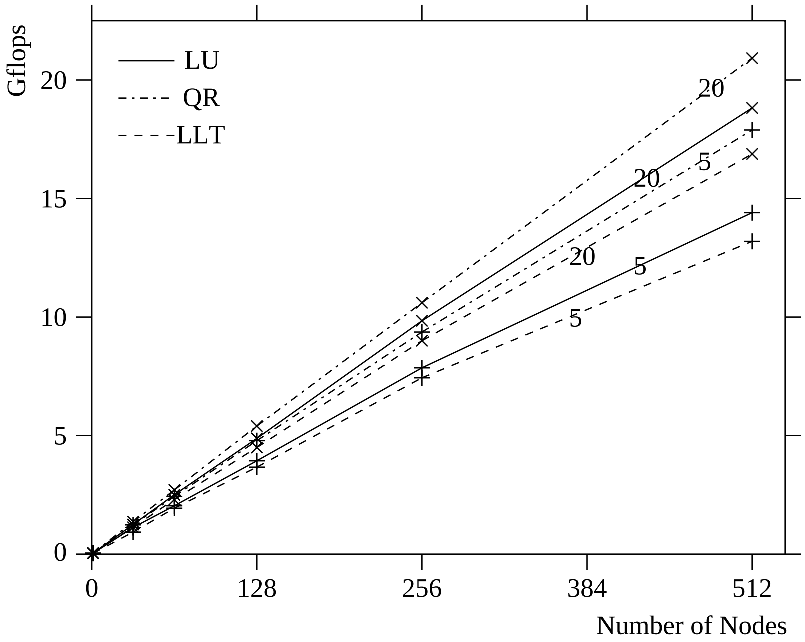


Figure 11: Scalability of factorization routines on the Intel Paragon (5, 20 Mbytes/node).

per process. That is, for the given problem size, N, on the number of processes used, N_p,

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

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 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

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

the scalability for memory-constrained problems can readily be accessed by the extent to which the isogranularity curves differ from linearity. Isogranularity was first defined in [24], and later explored in [21, 20].

Figure 11 shows the isogranularity plots for the ScaLAPACK factorization routines on the Paragon. The matrix size per process is fixed at 5 and 20 Mbytes on the Paragon. Refer to Figures 6, 8, and 10 for block size and process grid size characteristics. The near-linearity of these plots shows that the ScaLAPACK routines are quite scalable on this system.

6. Conclusions

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. We have seen that 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 obtain good performance, since the computation performed by each process within PBLAS routines can itself be performed using the assembly-coded sequential BLAS.

In designing and implementing software libraries, there is a tradeoff between performance and software design considerations, such as modularity and clarity. As described in Section 4.1, it is possible to combine messages to reduce the communication cost in several places, 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 concluded 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, Delta, and Paragon systems. Similar studies may be performed on other architectures to which the BLACS have been ported, including PVM, TMC CM-5, Cray T3D, and IBM SP1 and SP2.

The ScaLAPACK routines are currently available through *netlib* for all numeric data types, single precision real, double precision real, single precision complex and double precision complex. To obtain the routines, and the ScaLAPACK Reference Manual [10], send the message "`send index from scalapack`" to `netlib@ornl.gov`.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments to improve the quality of the paper. This research was performed in part using the Intel iPSC/860 hypercube and the Paragon computers at the 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.

7. References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, 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. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Second Edition.* SIAM Press, Philadelphia, PA, 1995.

[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] C. Bischof and C. Van Loan. The WY Representation for Products of Householder Matrices. *SIAM J. of Sci. Stat. Computing*, 8:s2–s13, 1987.

[5] 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.

[6] 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.

[7] 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.

[8] J. Choi, J. J. Dongarra, and D. W. Walker. PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subprograms. *Submitted to Concurrency: Practice and Experience*, 1994. Also available on Oak Ridge National Laboratory Technical Reports, TM-12268, February, 1994.

[9] J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurrency: Practice and Experience*, 6(7):543–570, October 1994.

[10] 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.

[11] J. Choi, J. J. Dongarra, and D. W. Walker. The Design of a Parallel, Dense Linear Algebra Software Library: Reduction to Hessenberg, Tridiagonal, and Bidiagonal Form. *Submitted to Numerical Algorithms*, 1995.

[12] K. Dackland, E. Elmroth, B. Kågström, and C. Van Loan. Parallel Block Matrix Factorizations on the Shared Memory Multiprocessor IBM 3090 VF/600J. *International Journal of Supercomputing Applications*, 6(1):69–97, Spring 1992.

[13] J. W. Demmel, M. T. Heath, and H. A. van der Vorst. Parallel Numerical Linear Algebra. *Acta Numerica*, pages 111–197, 1993.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] K. Gallivan, R. Plemmons, and A. Sameh. Parallel Algorithms for Dense Linear Algebra Computations. *SIAM Review*, 32:54–135, 1990.

[20] V. Kumar, A. Grama, A. Gupta, and G. Karypsis. *Introduction to Parallel Computing, Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[21] V. Kumar and A. Gupta. Analyzing Scalability of Parallel Algorithms and Architectures. Technical Report TR 91-18, University of Minnesota, Minneapolis, November 1992.

[22] R. Schreiber and C. Van Loan. A Storage Efficient WY Representation for Products of Householder Transformations. *SIAM J. of Sci. Stat. Computing*, 10:53–57, 1991.

[23] 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.

[24] X.-H. Sun and J. L. Gustafson. Toward a Better Parallel Performance Metric. *Parallel Computing*, 17:1093–1109, 1991.