CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 8(7), 517–535 (SEPTEMBER 1996)

PB-BLAS: A set of parallel block basic linear algebra subprograms

JAEYOUNG CHOI
School of Computing
Soongsil University
Sangdo5-Dong, Dongjak-Ku
Seoul 156-743, Korea

JACK J. DONGARRA
Department of Computer Science
University of Tennessee at Knoxville
107 Ayres Hall
Knoxville, TN 37996-1301, USA

JACK J. DONGARRA AND DAVID W. WALKER
Mathematical Sciences Section
Oak Ridge National Laboratory
PO Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367, USA

SUMMARY

We propose a new software package which would be very useful for implementing dense linear algebra algorithms on block-partitioned matrices. The routines are referred to as block basic linear algebra subprograms (BLAS), and their use is restricted to computations in which one or more of the matrices involved consists of a single row or column of blocks, and in which no more than one of the matrices consists of an unrestricted two-dimensional array of blocks. The functionality of the block BLAS routines can also be provided by Level 2 and 3 BLAS routines. However, for non-uniform memory access machines the use of the block BLAS permits certain optimizations in memory access to be taken advantage of. This is particularly true for distributed memory machines, for which the block BLAS are referred to as the *parallel block basic linear algebra subprograms* (PB-BLAS). The PB-BLAS are the main focus of this paper, and for a block-cyclic data distribution, a single row or column of blocks lies in a single row or column of the processor template.

The PB-BLAS consist of calls to the sequential BLAS for local computations, and calls to the BLACS for communication. The PB-BLAS are the building blocks for implementing ScaLAPACK, the distributed-memory version of LAPACK, and provide the same ease-of-use and portability for ScaLAPACK that the BLAS provide for LAPACK.

The PB-BLAS consist of all Level 2 and 3 BLAS routines for dense matrix computations (not for banded matrix) and four auxiliary routines for transposing and copying of a vector and/or a block vector. The PB-BLAS are currently available for all numeric data types, i.e., single and double precision, real and complex.

1. INTRODUCTION

In 1973, Hanson, Krogh and Lawson[1] described the advantages of adopting a set of basic routines for problems in linear algebra. The first set of basic linear algebra subprograms (Level 1 BLAS)[2] defines operations on one or two vectors. LINPACK[3] and EISPACK [4] are built on top of the Level 1 BLAS. An extended set of BLAS (Level 2 BLAS) [5] was proposed to support the development of software that would be portable and efficient, particularly on vector-processing machines. These routines perform computations on a matrix and one or two vectors, such as a matrix–vector product.

Current advanced architecture computers possess hierarchical memories in which accesses to data in the upper levels of the memory hierarchy (registers, cache, and/or local

CCC 1040–3108/96/070517–19

Received April 1994

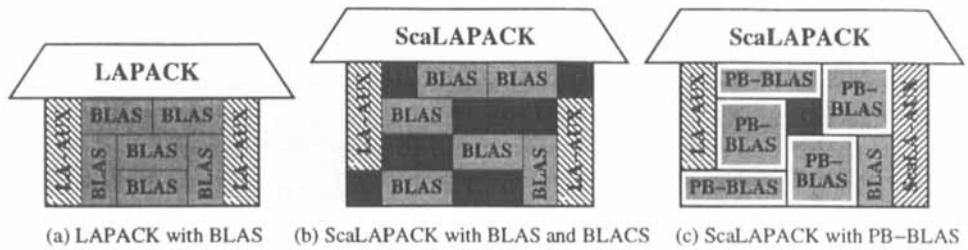


Figure 1. Building structure of LAPACK and ScaLAPACK. In the Figure, C represents BLACS communication. PB-BLAS simplifies the implementation of the ScaLAPACK by providing large blocks, which are combined with two small blocks of BLAS and BLACS

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 held in the upper levels. This can be done by partitioning the matrix or matrices into blocks and by performing the computation with matrix–matrix operations on the blocks. Another extended set of BLAS (Level 3 BLAS) [6] were proposed for that purpose. The Level 3 BLAS have been successfully used as the building blocks of a number of applications, including LAPACK [7], a software library that uses block-partitioned algorithms for performing dense and banded linear algebra computations on vector and shared memory computers. ScaLAPACK, the distributed version of the LAPACK library, also makes use of block-partitioned algorithms.

Higher performance can be attained on distributed memory computers when parallel dense matrix algorithms utilize a data distribution that views the computational nodes as a logical two-dimensional processor template [8,9]. In distributing matrix data over processors, we therefore assume a block cyclic (or scattered) distribution [10,9]. The block cyclic distribution can reproduce the most common data distributions used in dense linear algebra, as described briefly in the next Section.

There has been much interest recently in developing parallel versions of the BLAS for distributed memory concurrent computers [11–14]. Some of this research proposed parallelizing the BLAS, and some implemented a few important BLAS routines, such as matrix–matrix multiplication. There is no complete, general, parallel version of the BLAS currently available that can be used as the building blocks for implementing dense linear algebra computations on distributed-memory multiprocessors.

The basic linear algebra communication subprograms (BLACS) [15] comprise a package that provides the same ease-of-use and portability for message-passing in parallel linear algebra programs as the BLAS provide for computation. The BLACS efficiently support not only point-to-point operations between processors on a logical two-dimensional processor template, but also collective communications on such templates, or within just a template row or column.

We propose a new set of linear algebra routines for implementing ScaLAPACK on top of the sequential BLAS and the BLACS. The functionality of these routines, called the parallel block basic linear algebra subprograms (PB-BLAS), could be provided by parallel versions of the Level 2 and Level 3 BLAS; however, the PB-BLAS 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 to be made

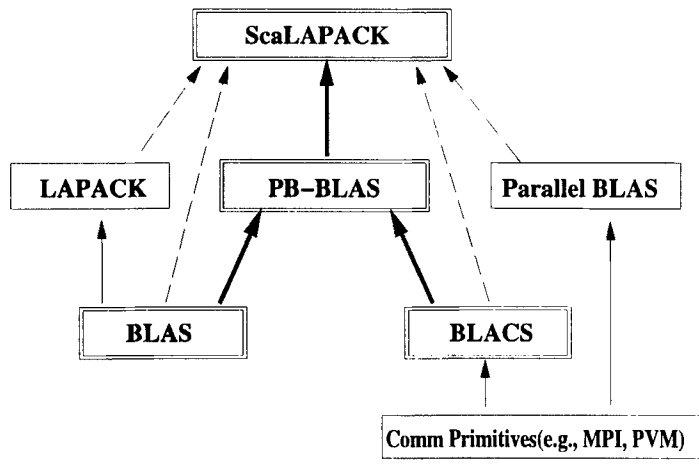


Figure 2. Hierarchical view of ScaLAPACK. The PB-BLAS plays a major role in implementing the ScaLAPACK

that would not be possible (or would be difficult) if general-purpose Level 2 and Level 3 BLAS were used. Consider the following types of matrix distributed block cyclically over a two-dimensional array of processors:

1. a matrix of $M_b \times N_b$ blocks, distributed over the whole 2-D processor template
2. a vector of L_b blocks, distributed over either a row or a column of the processor template. Clearly, this is a special case of a matrix of blocks, with either $M_b = 1$ or $N_b = 1$
3. a single block lying in a single processor in the processor template.

The restrictions that the PB-BLAS impose are as follows. No more than one of the matrices involved may be a full block matrix; the other matrices involved must be block vectors or single blocks. Computations that do not conform to these restrictions must be handled differently, for example, by using the PUMMA package [16] that has been developed for general matrix–matrix multiplication.

The PB-BLAS consist of calls to the sequential BLAS for local computations and calls to the BLACS for communication. The PB-BLAS are used as the building blocks for implementing the ScaLAPACK library, and provide the same ease-of-use and portability for ScaLAPACK that the BLAS provide for LAPACK. Figure 1 shows schematically how the PB-BLAS simplify the implementation of ScaLAPACK by combining small blocks of BLAS and BLACS and providing larger building blocks. Figure 2 shows a hierarchical view of ScaLAPACK. Main ScaLAPACK routines usually call only the PB-BLAS, but the auxiliary ScaLAPACK routines may need to call the BLAS directly for local computations and the BLACS for communication among processors.

The PB-BLAS consist of all nine Level 3 BLAS routines, four Level 2 BLAS routines (PB_GEMV, PB_HEMV, PB_SYMV, and PB_TRMV), and two auxiliary routines to transpose between a row vector of blocks and a column vector of blocks (PB_TRAN and PB_TRNV). Here we use the LAPACK naming convention in which "_" is replaced with "S" (single precision), "D" (double precision), "C" (single precision complex), or "Z" (double

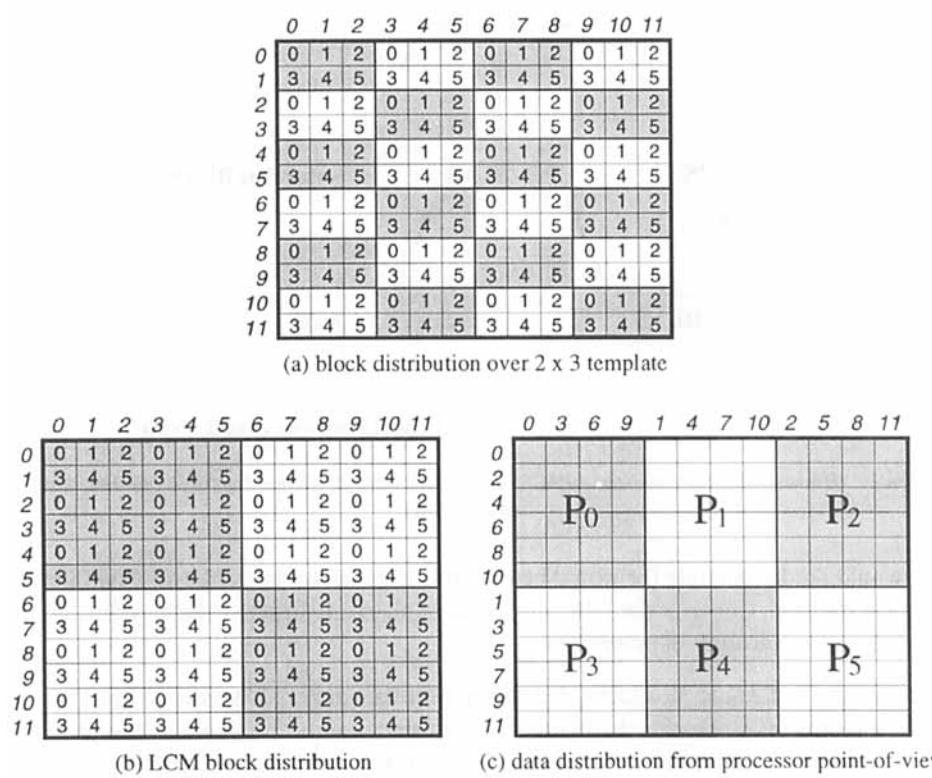


Figure 3. A matrix with 12×12 *blocks is distributed over a* 2×3 *processor template: (a) the shaded and unshaded areas represent different templates. The numbered squares represent blocks of elements, and the number indicates at which location in the processor template the block is stored – all blocks labeled with the same number are stored in the same processor. The slanted numbers, on the left and on the top of the matrix, represent indices of row of blocks and column of blocks, respectively; (b) the matrix has* 2×2 LCM *blocks. Blocks belong to the same processor if the relative locations of blocks are the same in each square LCM block. The definition of the LCM block is defined in the text: (c) it is easier to see the distribution from processor point-of-view in order to implement algorithms. Each processor has* 6×4 *blocks*

precision complex). The PB-BLAS routines have similar argument lists to the sequential BLAS routines, but contain additional parameters to specify positions of matrices in the processor template, to select destinations of broadcasting matrices, and to control communication schemes. Software developers and application programmers, who are familiar with the BLAS routines, should have no difficulty in using the PB-BLAS.

In Section 4., we will consider a simple numerical linear algebra example, Cholesky factorization, to compare a LAPACK routine with the corresponding ScaLAPACK routine, and to demonstrate the effectiveness of the PB-BLAS. In the text, specifications and an explanation of the routines are also given for the double precision real data type (and double precision complex if there is no real case).

2. DESIGN ISSUES

The way in which a matrix is distributed over the processors of a concurrent computer 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. In the block cyclic distribution, described in detail in [8,9], an $M \times N$ matrix is partitioned into blocks of size $r \times c$, and blocks separated by a fixed stride in the column and row directions are assigned to the same processor. If the stride in the column and row directions is P and Q blocks respectively, then we require that $P \cdot Q$ equal the number of processors, N_p. Thus, it is useful to imagine the processors arranged as a $P \times Q$ mesh or template. The processor at position (p,q) $(0 \leq p < P, 0 \leq q < Q)$ in the template is assigned the blocks indexed by

$$(p + i \cdot P, q + j \cdot Q), \tag{1}$$

where $i = 0, \ldots, \lfloor (M_b - p - 1)/P \rfloor$, $j = 0, \ldots, \lfloor (N_b - q - 1)/Q \rfloor$, and $M_b \times N_b$ is the size in blocks of the matrix ($M_b = \lceil M/r \rceil$, $N_b = \lceil N/s \rceil$).

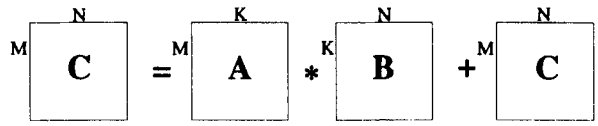
Blocks are scattered in this way so that good load balance can be maintained in parallel algorithms, such as LU factorization [10,9]. The non-scattered decomposition (or pure block distribution) is just a special case of the cyclic distribution in which the block size is given by $r = \lceil M/P \rceil$ and $c = \lceil N/Q \rceil$. A purely scattered decomposition (or two-dimensional wrapped distribution) is another special case in which the block size is given by $r = c = 1$.

We assume that a matrix is distributed over a two-dimensional processor mesh, or template, so that in general each processor has several blocks of the matrix as shown in Figure 3 (a), where a matrix with 12×12 blocks is distributed over a 2×3 template. Denoting the least common multiple of P and Q by LCM, we refer to a square of $LCM \times LCM$ blocks as an LCM block. Thus, the matrix may be viewed as a 2×2 array of LCM blocks, as shown in Figure 3 (b). Each processor has 6×4 blocks as in Figure 3 (c).

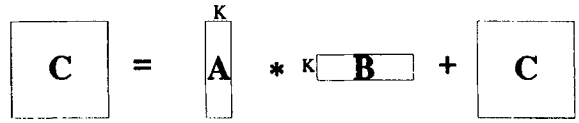
The LCM block concept was introduced in [16,21], and is very useful for implementing algorithms that use a block cyclic data distribution. Blocks belong to the same processor if their relative locations are the same in each square LCM block. All LCM blocks have the same structure and the same data distribution as the first LCM block. That is, when an operation is executed on a block of the first LCM block, the same operation can be done simultaneously on other blocks, which have the same relative location in each LCM block.

The LCM block concept is extended further in this paper to deal efficiently with symmetric and Hermitian matrices. Assume that a lower (or upper) triangular matrix A is distributed on a two dimensional processor template with the block cyclic decomposition. The locally stored matrix in each processor is not a lower (or upper) triangular matrix. The block layout of the first LCM block is the same as that of the other diagonal LCM blocks. Processors compute their own block layout of the first LCM block from their relative position on the processor template, then they can determine their own physical data distribution of the matrix A. This concept is used for updating the upper or lower triangular part of a symmetric or Hermitian matrix (PBDSYRK, PBDSYR2K, PBZHERK, and PBZHER2K), and for multiplying with it (PBDTRMM, PBDTRMV, PBZHEMM, and PBZHEMV). For details of the implementation, see Section 3.4.

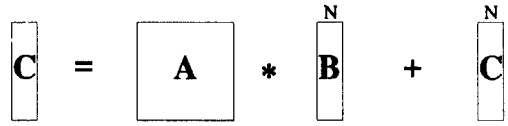
To illustrate the use of the PB-BLAS consider the matrix multiplication routine, DGEMM,



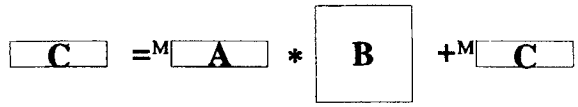
(a) General case of matrix multiplication



(b) C is a full block matrix



(c) A is a full block matrix



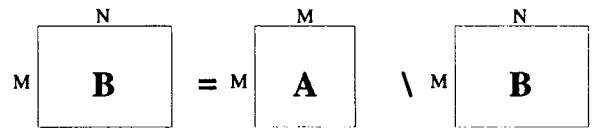
(d) B is a full block matrix

Figure 4. DGEMM: matrix multiplication when A and B are not transposed. Each case should be computed differently on distributed memory environments: (a) general case; (b) C is a full block matrix and A and B are a column and a row of blocks, respectively; (c) A is a full block matrix, (d) B is a full block matrix

for the non-transposed case: $C_{M\times N} \Leftarrow \alpha \cdot A_{M\times K} \cdot B_{K\times N} + \beta \cdot C_{M\times N}$. The PB-BLAS version handles three distinct cases depending on the sizes of the matrices involved in the computation, i.e., on whether A, B or C is a full block matrix rather than a vector of blocks. These three cases are shown in Figure 4. If K corresponds to just one block then A is a column of blocks, B is a row of blocks and C is a full block matrix as shown in Figure 4 (b). If N corresponds to just a single block then B and C are columns of blocks and A is a full block matrix (Figure 4 (c)). Finally, Figure 4 (d) shows the case in which M corresponds to a single block, so that A and C are rows of blocks and B is a full block matrix. If there is no limitation as shown in Figure 4 (a), the problem is beyond the scope of the PB-BLAS and such a problem needs to be handled with different software, such as the PUMMA package [16].

As a second example, consider the solution of triangular systems (DTRSM in the Level 3 BLAS) when the triangular matrix A is located on the left of B, as shown in Figure 5. If M corresponds to a single block, then A consists of one block, which is located on one processor, and B is a row of blocks, located on a row of the processor template. If N corresponds to a single block, then A is a full triangular matrix, distributed over all processors, and B is a column of blocks, located on a column of the processor template. The two cases are implemented separately.

In designing the PB-BLAS the following principles were followed:



(a) General case

(b) M is limited to its block size, A is a single block

(c) N is limited to its block size, A is a full block matrix

Figure 5. DTRSM: solution of triangular systems when the triangular matrix A is located on the left of B: (a) general case; (b) when M is limited to its block size, A is a single block and it is located in a single processor; (c) when N is limited to its block size, A is a full block matrix and it is distributed over 2-D processors

1. maximize the size of submatrices multiplied (or computed) in each processor
2. maximize the size of submatrices communicated among processors, thereby reducing the frequency of communication
3. minimize the size of working space required during computation.

The PB-BLAS are efficiently implemented by maximizing the size of submatrices for local computation and communication, and are implemented with minimum working space. The performance of a parallel program implemented by a novice using the PB-BLAS will generally be commensurate with that hand-coded by an experienced programmer, and in this sense the PB-BLAS attain near maximum performance.

In addition to the fundamental restrictions on matrix size stated above, the implementation of the PB-BLAS is simplified by making the following assumptions about matrix alignment:

1. The basic unit of storage in the PB-BLAS is a block of elements, thus matrices are assumed to start at the beginning of their first blocks of matrices. However, blocks in the last row or column of blocks in a matrix do not have to be full, i.e., the size of the matrix in elements does not have to be exactly divisible by the block size.
2. The PB-BLAS also makes assumptions about the alignment of matrices on the processor template. If a full block matrix begins at location (p_0, q_0) in the template, then any column vectors of blocks must also begin in row p_0, and any row vectors of blocks must begin in column q_0 if no transposition of the vector is involved in the computation. Each PB-BLAS routine is passed arguments that specify the start position of each matrix in the processor template.

ScaLAPACK makes use of block-partitioned algorithms, so it is natural to use the PB-BLAS as 'building blocks' for ScaLAPACK, and to assume that the first element of a matrix

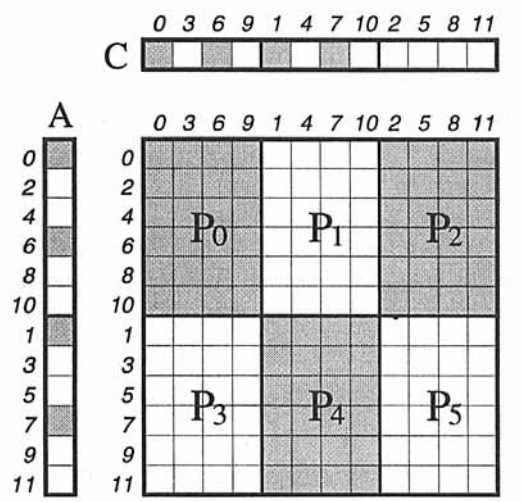


Figure 6. PBDTRAN routine: a routine for transposing a column or row of blocks (IAROW = IACOL = ICROW = ICCOL = 0)

is aligned with block boundaries. The only exceptions to this alignment constraint in the PB-BLAS are lower-level analogs of the Level 2 BLAS routines, PBDGEMV, PBDSYMV, PBDTRMV, and PBZHEMV, in which the first elements of vectors and the corresponding matrix can be located in the middle of blocks.

In Figure 4 (b), the first block of the column of blocks A and the row of blocks of B should be located at the same row and column processor as the first block of the matrix C, respectively. In Figure 4 (c), the column of blocks B needs to be transposed to multiply with the matrix A. The first block of B can be located in any processor, but the first block of the column of blocks C should be located in the same processor row as the first block of A.

3. IMPLEMENTATION OF THE PB-BLAS

The PB-BLAS routines need more arguments than the corresponding BLAS routine to specify the block sizes, positions of matrix, destinations of broadcasting matrices, communication schemes, and working space. In general, the arguments to PB-BLAS routines follow the same conventions as the BLAS and the BLACS. We now illustrate these conventions, as applied to the PB-BLAS, with a few examples.

3.1. PBDTRAN

PBDTRAN transposes a column (row) of blocks to a row (column) of blocks, so that a vector of blocks that was formerly in one column (row) of the processor template becomes redistributed to lie along one row (column) of the template.

```
      SUBROUTINE PBDTRAN( ADIST, TRFMT, M, N, NB, A, LDA, C, LDC,
     $                    IAROW, IACOL, ICROW, ICCOL, WORK )
      CHARACTER*1         ADIST, TRFMT
      INTEGER             M, N, NB, LDA, LDC
```

```
      INTEGER              IAROW, IACOL, ICROW, ICCOL
      DOUBLE PRECISION     A( LDA, * ), C( LDC, * ), WORK( * )
```

ADIST gives for the distribution of A: 'C' (columnwise) or 'R' (rowwise), and TRFMT gives the transpose format for complex data type: 'T' (transpose) or 'C' (conjugate transpose). For real data types it is ignored. M and N are number of elements in rows and columns of A, respectively. If A is distributed columnwise (ADIST = 'C'), an M × N column of blocks A is located on one column of processors, IACOL, beginning from IAROW, with a row block size NB. If IACOL = -1 it is assumed that all columns of processors have their own copies of A. The resultant N×M row of blocks C will be located on a row of processors, ICROW, beginning from ICCOL, with a column block size NB. If ICROW = -1, all rows of processors will have their own copies of C.

Figure 6 shows an example that transposes a column of blocks A to a row of blocks A over a 2 × 3 processor template. Assuming that the first blocks of A and C are located in processor P_0 (IAROW = IACOL = ICROW = ICCOL = 0), then processors P_0 and P_3 have a column of blocks A, and P_0, P_1 and P_2 will have a row of blocks C after transposing A. At first, P_0 sends $A(0)$ and $A(6)$ to itself. At the same time, P_3 sends $A(1)$ and $A(7)$ to P_1. Next, P_0 sends $A(2)$ and $A(8)$ to P_2, P_3 sends $A(3)$ and $A(9)$ to P_0, and so on. The sending processors pack the data in order to minimize the frequency of communications, and the receiving processors unpack the data as soon as they receive them in order to minimize working space. The LCM concept is used for packing and unpacking the data. The row block distance of A is 3 (= LCM/P) for packing on P_0 and P_3, and the column block distance of C is 2 (= LCM/Q) for unpacking on P_0, P_1 and P_2.

If each column of processors has its own copies of A (IACOL = -1), they operate independently to transpose A. Each column of processors sends its blocks to its own diagonal processors. In the Figure, P_0, P_4 and P_2 are diagonal processors. A diagonal processor of the first column, P_0, collects $A(0)$ and $A(6)$ from itself, and $A(3)$ and $A(9)$ from P_3. If all row of processors are to have their own copies of C (ICROW = -1), the resultant C is broadcast column-wise from the diagonal blocks.

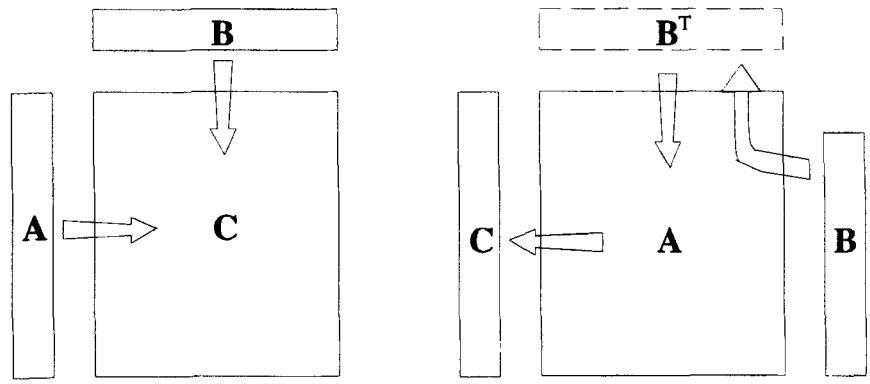
3.2. PBDGEMM

PBDGEMM is a matrix–matrix multiplication routine:

```
      SUBROUTINE PBDGEMM( MTXPOS, TRANSA, TRANSB, M, N, K, MB, NB, KB,
     $                    ALPHA, A, LDA, B, LDB, BETA, C, LDC, IAROW,
     $                    IACOL, IBROW, IBCOL, ICROW, ICCOL, BR1ST,
     $                    ACOMM, SEND2A, BCOMM, SEND2B, WORK )
      CHARACTER*1          MTXPOS, TRANSA, TRANSB, BR1ST
      CHARACTER*1          ACOMM, SEND2A, BCOMM, SEND2B
      INTEGER              M, N, K, MB, NB, KB, LDA, LDB, LDC
      INTEGER              IAROW, IACOL, IBROW, IBCOL, ICROW, ICCOL
      DOUBLE PRECISION     ALPHA, BETA
      DOUBLE PRECISION     A( LDA, * ), B( LDB, * ), C( LDC, * )
      DOUBLE PRECISION     WORK( * )
```

MTXPOS indicates which matrix is the full block matrix: 'A', 'B' or 'C'. M, N and K give the sizes of matrices in elements, and MB, NB and KB are the corresponding block sizes.

Figure 7 (a) shows a simple example of matrix–matrix multiplication, where A is a column of blocks, B is a row of blocks and C is a full block matrix. For local computation,



(a) C is a matrix (MTXPOS = "C") (b) A is a matrix (MTXPOS = "A")

Figure 7. Matrix multiplication examples when A and B are multiplied in non-transposed form: (a) C is a full block matrix; (b) A is a full block matrix

A and B need to be broadcast row-wise and column-wise, respectively. The following issues need to be decided before broadcasting A and B:

1. Which one is ready to be broadcast first, A or B ?
2. How do you send A and/or B ? That is, what communication schemes will be used to broadcast them ?
3. Where will A (or B) be sent to in other processors ? If the same layout of memory is assumed in each processor, then A (or B) could either be broadcast to same memory locations as in the root of the broadcast so A (or B) is overwritten, or it can be broadcast to working space.

These issues are addressed by extra arguments of the PB-BLAS routines. `BR1ST` determines the column (or row) block to be broadcast first when `MTXPOS` = 'C'. `ACOMM` (or `BCOMM`) controls the communication scheme of A (or B), which follows the topology definition of the BLACS [15], as discussed in Appendix A. `SEND2A` (or `SEND2B`) specifies the location of the blocks that are broadcast (either A or B, or working space). To better understand the need to specify whether the working space is to be used in the broadcasting of A or B, consider the case in which A is a column of blocks that is a submatrix of some other matrix. Clearly, if A is broadcast to the same memory location in other processors, data in the parent matrix will be incorrectly overwritten. In this case we should broadcast to the working space (`SEND2A` = 'No'). On the other hand, if A is not a submatrix, or if A lies entirely within the working space, then it can be broadcast to the same location in each processor, and extra memory and a memory-to-memory copy can be avoided (`SEND2A` = 'Yes').

The position in the processor template of a column (or row) block must be aligned with the position of the full block matrix. Figure 7 (a) shows that the first blocks of A and B are located at the same row and column of the processor template as the first block of C, respectively, If one of the blocks is misaligned, it should be moved to the appropriate position before the routine is called.

In Figure 7 (b), A is a full block matrix, and B and C are columns of blocks. The computation proceeds as follows. First, B is transposed, so that the first block of B^T is located at the same column position as the first block of A. The transposed row of blocks B^T is broadcast column-wise, and it is multiplied with the local portion of A in each processor. Then the local products are added along template rows to produce C. The first block of B may be located at any position, since the transposition of B is involved in the computation. However, the first block of C must be located at the same row position as the first block of A.

3.3. PBDTRSM

```
      SUBROUTINE PBDTRSM( MTXBLK, SIDE, UPLO, TRANSA, DIAG, M, N, NB,
     $                    ALPHA, A, LDA, B, LDB, IAROW, IACOL, IBPOS,
     $                    COMMA, SEND2A, WORK )
      CHARACTER*1         SIDE, UPLO, TRANSA, DIAG
      CHARACTER*1         MTXBLK, COMMA, SEND2A
      INTEGER             M, N, NB, LDA, LDB, IAROW, IACOL, IBPOS
      DOUBLE PRECISION    ALPHA
      DOUBLE PRECISION    A( LDA, * ), B( LDB, * ), WORK( * )
```

PBDTRSM solves a triangular system. If SIDE = 'Left' and M is limited by its block size NB (M $\leq$ NB), the triangular matrix A is just a single block (MTXBLK = 'Block'), which is located on just one processor (IAROW, IACOL) as in Figure 5 (b). The M $\times$ N row of blocks B is located on a row of processors, IAROW, starting at IBPOS,

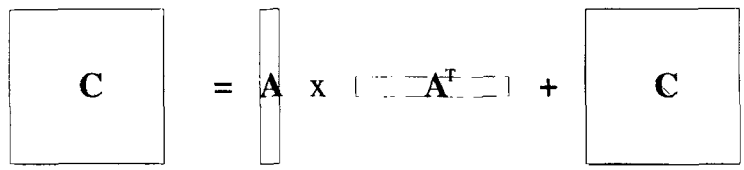
The routine is executed on one row of processors, IAROW. The triangular block A is broadcast row-wise with one of the BLACS communication topologies (COMMA), and the copies are stored either in A (SEND2A = 'Yes') or working space (SEND2A = 'No'). The row of processors compute their local portion of B by calling the Level 3 BLAS routine, DTRSM.

If SIDE = 'Left', and N is limited by its block size NB (N $\leq$ NB), the triangular matrix A is a full triangular matrix distributed over the whole two-dimensional processor template (MTXBLK = 'Matrix'), and its first block is located at (IAROW, IACOL),. The M $\times$ N column of blocks B is located on a column of processors, IBPOS, starting at IAROW, as shown in Figure 5 (c).

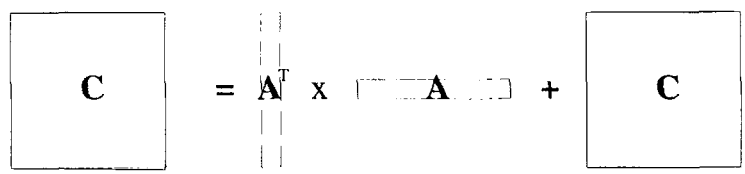
The implementation of the linear triangular system solver is a two-dimensional block version of Li and Coleman's method [21]. Since A and B are distributed block cyclically, all computations in [21] are changed to block computations using the routines DTRSM and DGEMM. If SIDE = 'Left', $(Q-1)$ blocks of B are rotated column-wise (approximately $\lceil (Q-1)/P \rceil$ blocks in each row of processors). The two arguments COMMA and SEND2A are ignored when MTXBLK = 'Matrix'.

3.4. PBDSYRK

```
      SUBROUTINE PBDSYRK( UPLO, TRANS, N, K, NB, ALPHA, A, LDA, BETA,
     $                    C, LDC, IAPOS, ICROW, ICCOL, ACOMM, SEND2A,
     $                    MULLEN, PRESV, WORK )
      CHARACTER*1         UPLO, TRANS, ACOMM, SEND2A, PRESV
      INTEGER             N, K, NB, LDA, LDC
      INTEGER             IAPOS, ICROW, ICCOL, MULLEN
      DOUBLE PRECISION    ALPHA, BETA
      DOUBLE PRECISION    A( LDA, * ), C( LDC, * ), WORK( * )
```



(a) A is not transposed (TRANS = 'N')



(b) A is transposed (TRANS = 'T')

Figure 8. Overview of PBDSYRK routine: (a) A is not transposed; (b) A is transposed

PBDSYRK performs a rank-k update on an N × N symmetric matrix C with an N × K column of blocks A (TRANS = 'No'), or with a K × N row of blocks A (TRANS = 'Trans'). That is,

$$\mathbf{C}_{N\times N} = \alpha\cdot\mathbf{A}_{N\times K}\cdot\mathbf{A}^T_{K\times N}+\beta\cdot\mathbf{C}_{N\times N}, \quad \text{or} \quad \mathbf{C}_{N\times N} = \alpha\cdot\mathbf{A}^T_{N\times K}\cdot\mathbf{A}_{K\times N}+\beta\cdot\mathbf{C}_{N\times N}$$

An overview of the routines is shown in Figure 8.

ICROW and ICCOL specify the row and column position of the first block of the matrix C, respectively. IAPOS specifies the column position of the column of blocks A if TRANS = 'No'. The row position of the first block of A is assumed to be ICROW. If TRANS = 'Trans', IAPOS specifies the row position of the row of blocks A, and the column position of the first block of A is assumed to be ICCOL.

Figure 9 (a) shows an example of PBDSYRK when TRANS = 'No' and UPLO = 'Lower'. It is assumed that 24 × 24 blocks of C are distributed over a 2 × 3 processor template, and C has 4 × 4 LCM blocks.

The computing procedure of PBDSYRK is as follows. First, the column of blocks A is broadcast row-wise from IAPOS, so that each column of processors then has its own copy of A. Each column of processors transposes A independently, and the transposed blocks in diagonal processors are broadcast column-wise. Each processor updates its own portion of C with its own portion of A and A^T.

It is often necessary to update the lower triangular matrix C without modifying data in its upper triangular part (PRESV = 'Yes'). The simplest way to do this is repeatedly to update one column of blocks of C, but if the block size (NB) is small, this updating process will not be efficient. However, it is possible to modify several columns of blocks of C. Figure 9 (b) shows this example from the point of view of the processor at P_0, where 2 ($= LCM/Q$) columns of blocks are updated at the same time. First, $A(0)$, $A(2)$, and $A(4)$ are multiplied with $A^T(0)$ and $A^T(3)$, and only the lower triangular part of 3 × 2 ($= LCM/P \times LCM/Q$) blocks are added to L_{11}. L_1 is updated by multiplying the rest of A with $A^T(0)$ and $A^T(3)$. Then L_{21} and L_2 are updated in the same way. The above scheme can be extended further. Figure 9 (c) shows the same example, where 4 ($= 2 \cdot LCM/Q$)

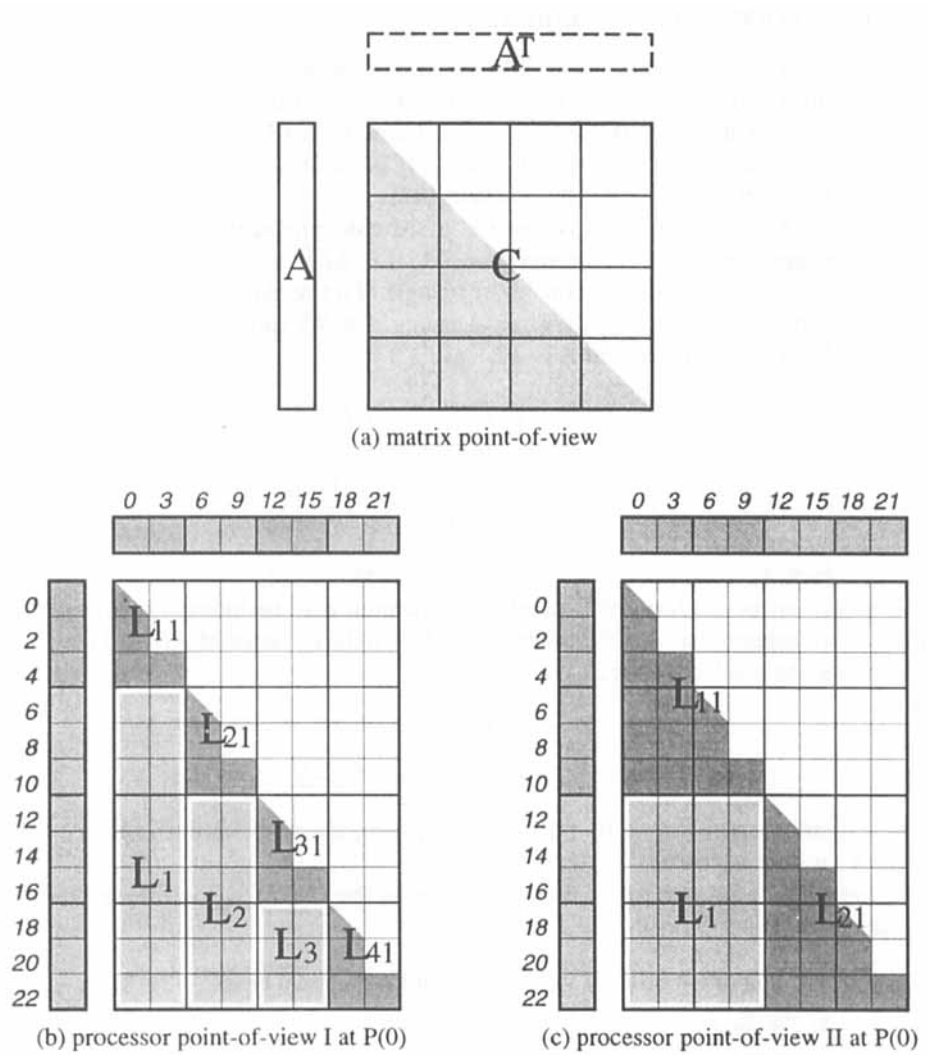


Figure 9. PBDSYRK routine: a routine for rank-k updating

columns of blocks are updated simultaneously.

It is desirable to update a multiple of LCM/Q blocks at a time from the LCM block concept. In the argument list of the PBDSYRK routine, MULLEN specifies an approximate length of multiplication to update C efficiently. The multiple factor is computed by $k = \lceil MULLEN/((LCM/Q) \cdot NB) \rceil$, and $k \cdot (LCM/Q)$ columns of blocks are updated simultaneously inside the routine. The optimum number is determined by processor characteristics as well as the numbor of processors (P and Q), the size of the matrix and the block size. The optimum number was found to be about 40 on the Intel i860 and Delta computers.

However, if it is permissible to change the data in the upper triangular part of C (PRESV = 'No'), L_{11} and L_1 can be updated with one multiplication step. This combined computation is faster.

4. APPLICATIONS OF THE PB-BLAS

In this Section we illustrate how the PB-BLAS routines can be used to implement a simple numerical linear algebra algorithm, Cholesky factorization. This is the same example as that used to demonstrate the effectiveness of the Level 3 BLAS in [6]. However, we use the right-looking version of the algorithm, since it minimizes data communication and distributes the computation across all processors [22].

Cholesky factorization factors a symmetric, positive definite matrix A into the product of a lower triangular matrix L and its transpose, i.e., $A = LL^T$. It is assumed that the lower trianglular portion of A is stored in the lower triangle of a two-dimensional array, and the computed elements of L overwrite the given elements of A. We partition the $n \times n$ matrices A, L and L^T, and write the system $A = LL^T$ as

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

where the block A_{11} is an $r \times r$ matrix, where r is the block size.

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

$$\begin{aligned} L_{21} &\Leftarrow A_{21}(L_{11}^T)^{-1} \\ A'_{22} &\Leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T \end{aligned}$$

The factorization can be done by recursively applying the steps outlined above to the $(n-r) \times (n-r)$ matrix A'_{22}.

The computation procedures of the above steps in the LAPACK routine, involve the following operations:

1. DPOTF2: compute Cholesky factorization of the diagonal block,
   $A_{11} \Rightarrow L_{11}L_{11}^T$
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,
   $A'_{22} \Leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T$

For the parallel implementation of the block-partitioned Cholesky factorization, assume that the lower triangular matrix A is distributed over a $P \times Q$ processor template with a block cyclic distribution and a block size $r \times r$. In the corresponding ScaLAPACK routine, PDPOTRF, the computation procedures outlined above are as follows:

1. PDPOTF2: a processor P_i, which has the $r \times r$ diagonal block A_{11}, performs Cholesky factorization of A_{11}. (The computation of PDPOTF2 is the same as that of DPOTF2. In PDPOTF2, P_i checks the positive definiteness of A_{11}, and broadcasts the result to the other processors so that the computation can be stopped if A_{11} is non-positive definite.)

2. PBDTRSM: L_{11} is broadcast along the column of the processors, and they compute the column of blocks of L_{21}.
3. PBDSYRK: the column of blocks L_{21} is broadcast row-wise and then transposed. Now, processors have their own portions of L_{21} and L_{21}^T. They update their local portions of the matrix A_{22}.

The Fortran code of the right-looking block Cholesky factorization, which is a variation of the LAPACK routine, DPOTRF, is given in Appendix B. The corresponding parallelized code, PDPOTRF, is included in Appendix C. PDPOTRF includes declarations to compute local indices, but overall it is very similar to the sequential version.

5. CONCLUSIONS

We have presented the PB-BLAS, a new set of block-oriented basic linear algebra subprograms for implementing ScaLAPACK on distributed memory concurrent computers. The PB-BLAS consist of calls to the sequential BLAS for local computations and calls to the BLACS for communication.

The PB-BLAS are a very useful tool for developing a parallel linear algebra code relying on the block cyclic data distribution, and provide ease-of-use and portability for ScaLAPACK. The PB-BLAS are the building blocks for implementing ScaLAPACK. A set of ScaLAPACK routines for performing LU, QR and Cholesky factorizations [19] and for reducing matrices to Hessenberg, tridiagonal and bidiagonal form have been implemented with the PB-BLAS [20].

The PB-BLAS are currently available for all arithmetic data types, i.e. single and double precision, real and complex. The PB-BLAS routines are available through *netlib* under the `scalapack` directory. To obtain them, send the message '`send index from scalapack`' to `netlib@ornl.gov`.

6. FUTURE WORK

We are developing a new version of the parallel BLAS routines, called the *Parallel BLAS* (or *PBLAS*), on top of the PB-BLAS. These consist of C-wrappers for the PB-BLAS that simplify the calling sequence of the PB-BLAS. Since the PBLAS use the C language's ability to dynamically allocate memory, a programmer does not need to worry about passing the working space of the routines. The PBLAS will hide the PB-BLAS parameters for specifying the matrix layout by using globally declared parameters. The calling sequences of the PBLAS will be very similar to those of the BLAS. Using the PBLAS instead of the PB-BLAS may sacrifice some flexibility, but it will provide greater ease-of-use to the programmer. A programmer, even one not very familiar with parallel programming, should be able to parallelize a sequential linear algebra quite easily using the PBLAS.

7. ACKNOWLEDGEMENT

The research was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, US Department of Energy, by the Defense Advanced

Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research Office, and by the Center for Research on Parallel Computing.

APPENDIX A: BLACS COMMUNICATION TOPOLOGIES

Topologies allow the user to optimize communication patterns for particular operations. In the BLAS, the TOPOLOGY parameter controls the communication pattern of the operations. It is used to optimize the way that the BLACS performs a given broadcast or global operation to fit a user's requirements. Different topologies spread the work involved in a given operation over the nodes in different ways.

```
TOPOLOGY = 'I' : Increasing ring
         = 'D' : decreasing ring
         = 'S' : split ring
         = 'H' : hypercube
         = 'F' : fully connected
         = '1' : tree broadcast with NBRANCHES = 1
         = '2' : tree broadcast with NBRANCHES = 2
         = '3' : tree broadcast with NBRANCHES = 3
         = '4' : tree broadcast with NBRANCHES = 4
         = '5' : tree broadcast with NBRANCHES = 5
         = '6' : tree broadcast with NBRANCHES = 6
         = '7' : tree broadcast with NBRANCHES = 7
         = '8' : tree broadcast with NBRANCHES = 8
         = '9' : tree broadcast with NBRANCHES = 9
```

For global operations, ring topologies, such as 'I', 'D' and 'S', are not available. For details, see the BLACS User's Guide [23].

APPENDIX B: LAPACK CHOLESKY FACTORIZATION

```
      SUBROUTINE DPOTRF( UPLO, N, A, LDA, INFO )
*
*     A variant of LAPACK of L*L**T factorization.
*     This is a right-looking Level-3 BLAS version of the algorithm.
*
      CHARACTER          UPLO
      INTEGER            INFO, LDA, N
      DOUBLE PRECISION   A( LDA, * )
*
      INTEGER            NB
      PARAMETER        ( NB = 64 )
*
*     Use blocked code.
*
      LOWER = LSAME( UPLO, 'L' )
*
      IF( LOWER ) THEN
*
*        Compute the Cholesky factorization A = L*L'
*
         DO 10 J = 1, N, NB
*
*           Factorize the current diagonal block
*           and test for non-positive-definiteness.
*
            JB = MIN( NB, N-J+1 )
            CALL DPOTF2( 'Lower', JB, A( J, J ), LDA, INFO )
```

```
            IF( INFO.NE.0 ) GO TO 20
*
            IF( J+JB.LE.N ) THEN
*
*              Form the column panel of L using the triangular solver
*
               CALL DTRSM( 'Right', 'Lower', 'Transpose', 'Non-unit',
     $                     N-J-JB+1, JB, 1.0D0, A( J, J ), LDA,
     $                     A( J+JB, J ), LDA )
*
*              Update the trailing matrix, A = A - L*L'
*
               CALL DSYRK( 'Lower', 'No transpose', N-J-JB+1, JB,
     $                     -1.0D0, A( J+JB, J ), LDA, 1.0D0,
     $                     A( J+JB, J+JB ), LDA )

            END IF
   10    CONTINUE
      END IF
      GO TO 30
*
   20 CONTINUE
      INFO = INFO + J - 1
*
   30 CONTINUE
      RETURN
      END
```

APPENDIX C: ScaLAPACK CHOLESKY FACTORIZATION

```
      SUBROUTINE PDPOTRF( UPLO, N, NB, A, LDA, INFO, WORK )
*
*     ScaLAPACK version of L*L**T factorization.
*     This is a right-looking PB-BLAS version of the algorithm.
*
      CHARACTER*1          UPLO
      INTEGER              N, NB, LDA, INFO
      DOUBLE PRECISION     A( LDA, * ), WORK( * )
*
      INTEGER              MULLEN
      PARAMETER           ( MULLEN = 40 )
*
      CALL GRIDINFO( NPROW, NPCOL, MYROW, MYCOL )
      LOWER = LSAME( UPLO, 'L' )
*
      II = 1
      JJ = 1
      IN = 1
      JN = 1
      ICURROW = 0
      ICURCOL = 0
*
      IF( LOWER ) THEN
*
*        Compute the Cholesky factorization A = L*L'
*
         DO 10 J = 1, N, NB
*
            JB = MIN( NB, N-J+1 )
            NXTROW = MOD( ICURROW+1, NPROW )
            NXTCOL = MOD( ICURCOL+1, NPCOL )
```

```
            IF( MYROW .EQ. ICURROW ) IN = II + JB
            IF( MYCOL .EQ. ICURCOL ) JN = JJ + JB
*
*           Factorize the current diagonal block
*           and test for non-positive-definiteness.
*
            CALL PDPOTF2( 'Lower', JB, A(II,JJ), LDA, ICURROW, ICURCOL,
     $                    INFO )
            IF( INFO.NE.0 ) GO TO 20
*
            IF( J+JB.LE.N ) THEN
*
*              Form the column panel of L using the triangular solver
*
               CALL PBDTRSM( 'Block', 'Right', 'Lower', 'Transpose',
     $                       'Non-Unit', N-J-JB+1, JB, NB, 1.0D0,
     $                       A(II,JJ), LDA, A(IN,JJ), LDA, ICURROW,
     $                       ICURCOL, NXTROW, '1-Tree', 'No', WORK )
*
*              Update the trailing matrix, A = A - L*L'
*
               CALL PBDSYRK( 'Lower', 'No Transpose', N-J-JB+1, JB, NB,
     $                       -1.0D0, A(IN,JJ), LDA, 1.0D0, A(IN,JN),
     $                       LDA, ICURCOL, NXTROW, NXTCOL, 'S-Ring',
     $                       'No', MULLEN, 'Yes', WORK )
*
               ICURROW = NXTROW
               ICURCOL = NXTCOL
               II = IN
               JJ = JN
            END IF
*
   10    CONTINUE
         END IF
         GO TO 30
*
   20 CONTINUE
         INFO = INFO + J - 1
*
   30 CONTINUE
         RETURN
         END
```

REFERENCES

1. R. J. Hanson, F. T. Krogh and C. L. Lawson, 'A proposal for standard linear algebra subprograms', *ACM SIGNUM Newsl.*, **8**, (16), (1973).
2. 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).
3. J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK Users' Guide*, SIAM Press, Philadelphia, PA, 1979.
4. B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema and C. B. Moler, 'Matrix eignensystem routines – EISPACK guide', *Lect. Notes Comput. Sci.*, **6**, Springer-Verlag, Berlin, 1976, 2nd edn.
5. J. J. Dongarra, J. Du Croz, S. Hammarling and R. J. Hanson, 'An extended set of Fortran basic linear algebra subprograms', *ACM Trans. Math. Softw.*, **16**, (1), 1–17 (1988).
6. J. J. Dongarra, J. Du Croz, S. Hammarling and I. Duff, 'A set of Level 3 basic linear algebra subprograms', *ACM Trans. Math. Softw.*, **18**, (1), 1–17 (1990).
7. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Ham-

marling, A. McKenney and D. Sorensen, 'LAPACK: A portable linear algebra library for high-performance computers', in *Proceedings of Supercomputing '90*, IEEE Press, 1990, pp. 1–10.
8. J. Choi, J. J. Dongarra and D. W. Walker, 'The design of scalable software libraries for distributed memory concurrent computers', in *Proceedings of Environment and Tools for Parallel Scientific Computing Workshop*, Saint Hilaire du Touvet, France, Elsevier Science Publishers, 7–8 September 1992, pp. 3–15.
9. J. J. Dongarra, R. van de Geijn and D. W. Walker, 'A look at scalable linear algebra libraries', in *Proceedings of the 1992 Scalable High Performance Computing Conference*, IEEE Press, 1992, pp. 372–379.
10. 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, 19–21 October, 1992.
11. M. Aboelaze, N. P. Chrisochoides, E. N. Houstis and C. E. Houstis, 'The parallelization of Level 2 and 3 BLAS operations on distributed memory machines', *Technical Report CSD-TR-91-007*, Purdue University, West Lafayette, IN, 1991.
12. 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, Elsevier Science Publishers, 7–8 September, 1992, pp. 17–29.
13. A. C. Elster, 'Basic matrix subprograms for distributed memory systems', in D. W. Walker and Q. F. Stout (Ed.), *Proceedings of the Fifth Distributed Memory Computing Conference*, IEEE Press, 1990, pp. 311–316.
14. R. D. Falgout, A. Skjellum, S. G. Smith and C. H. Still, 'The multicomputer toolbox approach to concurrent BLAS', *ACM Trans. Math. Softw.*, 1993 (preprint, submitted to a journal).
15. 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*, IEEE Computer Society Press, 1991, pp. 287–290.
16. J. Choi, J. J. Dongarra and D. W. Walker, 'PUMMA : parallel universal matrix multiplication algorithms on distributed memory concurrent computers', *Concurency: Pract. Exp.*, **6**, (7), 543–570 (1994).
17. J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker and R. Whaley, 'The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines', *Sci. Program.* 1995 (submitted for publication).
18. 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', *Numer. Algorithms*, **10**, 379–399 (1995).
19. J. Choi, J. J. Dongarra and D. W. Walker, 'Parallel matrix transpose algorithms on distributed memory concurrent computers', *Technical Report TM-12309*, Oak Ridge National Laboratory, Mathematical Sciences Section, October 1993.
20. J. J. Dongarra, R. Hempel, A. J. G. Hey and D. W. Walker, 'A proposal for a user-level, message passing interface in a distributed memory environment', *Technical Report TM-12231*, Oak Ridge National Laboratory, March 1993.
21. G. Li and T. F. Coleman, 'A parallel triangular solver for a distributed-memory multiprocessor', *SIAM J. Sci. Stat. Comput.*, **9**, 485–502 (1986).
22. 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.
23. J. J. Dongarra, R. A. van de Geijn and R. C. Whaley, 'BLACS user's guide, *Technical report*, University of Tennessee, 1993 (preprint).