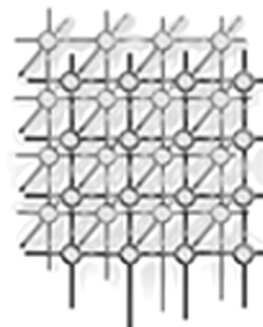


A distributed packed storage for large dense parallel in-core calculations

Marc Baboulin^{1,*}, Luc Giraud², Serge Gratton¹ and Julien Langou³



¹CERFACS, 42 avenue Gaspard Coriolis, 31057 Toulouse Cedex, France

²ENSEEIH, 2 rue Camichel, 31071 Toulouse Cedex, France

³University of Tennessee, 1122 Volunteer Boulevard, Knoxville, TN 37996-3450, U.S.A.

SUMMARY

In this paper we propose a distributed packed storage format that exploits the symmetry or the triangular structure of a dense matrix. This format stores only half of the matrix while maintaining most of the efficiency compared with a full storage for a wide range of operations. This work has been motivated by the fact that, in contrast to sequential linear algebra libraries (e.g. LAPACK), there is no routine or format that handles packed matrices in the currently available parallel distributed libraries. The proposed algorithms exclusively use the existing ScaLAPACK computational kernels, which proves the generality of the approach, provides easy portability of the code and provides efficient re-use of existing software. The performance results obtained for the Cholesky factorization show that our packed format performs as good as or better than the ScaLAPACK full storage algorithm for a small number of processors. For a larger number of processors, the ScaLAPACK full storage routine performs slightly better until each processor runs out of memory. Copyright © 2006 John Wiley & Sons, Ltd.

Received 16 January 2006; Revised 6 July 2006; Accepted 6 July 2006

KEY WORDS: scientific computing; dense linear algebra; parallel distributed algorithms; ScaLAPACK; packed storage format; Cholesky factorization; QR factorization

1. INTRODUCTION

Even though the current parallel platforms are able to provide increasing memory capacity, they are also being used to solve ever larger dense linear systems. This is the case, for instance, in geoscience or electromagnetic computations where the usual problem size is several hundreds of thousands of calculations requiring several tens of gigabytes. It is now possible to perform these calculations since the distributed memory parallel machines available today offer several gigabytes of memory

*Correspondence to: Marc Baboulin, CERFACS, 42 avenue Gaspard Coriolis, 31057 Toulouse Cedex, France.

†E-mail: baboulin@cerfacs.fr



per processor. However, when the dense matrices involved in these computations are symmetric, Hermitian or triangular, it could be worth exploiting the structure by storing only half of the matrix.

The ScaLAPACK [1] library has been designed to perform linear algebra parallel calculations on dense matrices. Contrary to the serial library LAPACK [2], ScaLAPACK does not currently support packed format for symmetric, Hermitian or triangular matrices [3]. A parallel solver has been studied in [4] that solves linear least squares problems encountered in gravity field calculations using the normal equations method. This solver also handles large symmetric linear systems in complex arithmetic resulting from Boundary Element Method (BEM) modelling of electromagnetic scattering. This solver uses about half of the memory required by ScaLAPACK and gives performance results similar to ScaLAPACK on moderately parallel platforms (up to 32 processors). Nevertheless, its parallel performance is less scalable than ScaLAPACK on higher processor counts because it uses a one-dimensional block cyclic distribution [1, p. 58]. The distributed packed storage format proposed in this paper uses ScaLAPACK and Parallel Basic Linear Algebra Subprograms (PBLAS) [5] routines. Each of these routines exploits the good load balancing of the two-dimensional block cyclic distribution [6,7] as it is implemented in ScaLAPACK [1, p. 58]. Moreover the calls to these routines in the applications that exploit this format will ensure the portability of software built upon them since these libraries are supported by all current parallel platforms. We shall see that, thanks to the set of routines we provide, building applications using matrices in distributed packed form is easy for users who are familiar with ScaLAPACK and gives good performance while saving a significant amount of memory compared with the full storage of the matrix.

This paper is organized as follows. In Section 2, we give an overview of the existing packed formats. The purpose of Section 3 is to describe the implementation of the distributed packed storage that we intend to use in parallel algorithms based on PBLAS or ScaLAPACK kernel routines. In Section 4, we explain how the Cholesky factorization can be implemented using the distributed packed format, this includes the descriptions of algorithms and a performance analysis on the IBM pSeries 690 and the CRAY XD1 cluster. Finally, some concluding comments are given in Section 5.

2. GENERALITIES ON PACKED STORAGE FORMATS

The sequential libraries LAPACK or BLAS (Basic Linear Algebra Subprograms) [8] provide a packed storage for symmetric, Hermitian or triangular matrices. This format allows us to store half of the matrix by addressing only the lower-triangular or upper-triangular part of the matrix, this part being held by columns.

For instance, the upper triangle of

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ * & a_{22} & a_{23} \\ * & * & a_{33} \end{pmatrix}$$

will be stored compactly in the matrix AP (A packed) such that $AP = [a_{11} \ a_{12} \ a_{22} \ a_{13} \ a_{23} \ a_{33}]$.

For symmetric matrices, either the lower triangle or the upper triangle can be stored. In both cases, the triangle is packed by columns, but one may notice that this is the same as storing the opposite triangle by rows. This packed storage format has been implemented in several routines of the Level-2 BLAS and LAPACK for:



- solving symmetric indefinite or symmetric/Hermitian positive definite linear systems;
- computing eigenvalues and eigenvectors for symmetric or symmetric-definite generalized eigenproblems (with condition number estimation and error bounds on the solution); and
- multiplying symmetric/Hermitian matrices and solving triangular systems.

Unfortunately, this format gives poor performance results when used in dense linear algebra computations since the algorithms are not able to make optimal use of the memory hierarchy. Blocked operations cannot be performed which prevents the use of Level-3 BLAS, and causes a dramatic loss of efficiency compared with the full storage (see, e.g., [9]). Another approach was used successfully in the IBM library ESSL [10], which consists of writing Level-3 BLAS for the packed format. Still another approach was used in [11] that defines a recursive packed Cholesky that operates on a so-called Recursive Packed Format (RPF) and requires variants of TRSM (triangular solve against a matrix) and SYRK (symmetric rank-k updates) that work on RPF.

In the framework of serial implementations, one can use blocking techniques and then store the lower-triangular or upper-triangular part of the blocked matrix. For instance, the blocked matrix

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ * & A_{22} & A_{23} \\ * & * & A_{33} \end{pmatrix}$$

can be stored in the blocked packed matrix

$$[A_{11} \quad A_{12} \quad A_{22} \quad A_{13} \quad A_{23} \quad A_{33}]$$

or

$$[A_{11} \quad A_{12} \quad A_{13} \quad A_{22} \quad A_{23} \quad A_{33}]$$

For serial implementations, the authors of [9] define a so-called upper (respectively lower) blocked hybrid format. In both formats, the blocks are ordered by columns to permit efficient operations on blocks using Level-3 BLAS (e.g. the blocked Cholesky algorithm in [9]) and the diagonal blocks are stored in a packed format so that exactly half of the matrix is stored.

Regarding parallel implementations for distributed memory architectures, out-of-core parallel solvers were implemented in several projects [12–15] where only the blocks of one triangular part are stored. However, for parallel in-core implementations, there is presently no satisfactory packed storage available for dense matrices. A packed storage for symmetric matrices distributed in a one-dimensional block cyclic column distribution is used in [4] for a least-squares solver based on the normal equations approach. In this implementation the blocks are ordered by columns and stored in a block-row array, resulting in extra storage due to the diagonal blocks that are fully stored. All blocks are manipulated using Level-3 BLAS or LAPACK blocked routines, but communication is performed by MPI primitives, whereas the distributed packed format that we propose in this paper relies on the ScaLAPACK communication layer BLACS (Basic Linear Algebra Communication Subprograms) [16].

Two algorithms using two-dimensional block cyclic data layouts are described in [17] for packed Cholesky factorization for distributed memory computing. There is currently no code available for these algorithms, but their implementation would be based on Level-3 BLAS and BLACS.

A preliminary study on a packed storage extension for ScaLAPACK has been carried out in [18]. In this format only the lower (or upper) part of each block column of the matrix is stored into a panel



considered as a separate ScaLAPACK matrix. This packed storage also stores the entire diagonal blocks. We can find in [18] experiments on the Cholesky factorization and symmetric eigensolvers. Our approach is an extension of this format.

3. DISTRIBUTED PACKED FORMAT

3.1. Definitions

ScaLAPACK proposes a data layout based on a two-dimensional block cyclic distribution. In this type of distribution, a matrix of size n is divided into blocks of size s (if we use square blocks) that are assigned to processors in a cyclic manner according to a $p \times q$ process grid. We refer to [1] for more details about this data layout. The blocks of size s that are spread among processors are called elementary blocks and the blocks of size $p.s \times q.s$ corresponding to the $p \times q$ process grid are called grid blocks.

In order to be stored in a distributed packed format, a matrix is first partitioned into larger square blocks of size b such that b is proportional to $l.s$ where l is the lowest common multiple of p and q ($b \geq l.s$). We define these blocks as ‘distributed blocks’.

In the remainder of this paper, the algorithms will be expressed in terms of distributed blocks that will be simply called ‘blocks’. Note that the distributed block performs naturally what is defined in [19] as algorithmic blocking or a tile for out-of-core implementations [13].

The following summarizes the hierarchy between the elementary block (hosted by one process), the grid block (corresponding to the process grid), and the distributed block (square block consisting of grid blocks). It shows the three kinds of block that we get when we consider a 2×3 process grid, $s = 1$, $b = 6$ and each block is labeled with the number of processes that stores it:

$$\boxed{0} : \text{elementary block, } \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline \end{array} : \text{grid block,}$$

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline \end{array} \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline \end{array} : \text{distributed block}$$

We consider here a matrix A partitioned into distributed blocks A_{ij} , and A can be either symmetric or upper triangular or lower triangular. We propose to store A compactly in a distributed packed format that consists of storing the blocks belonging to the upper or the lower triangle of A in a ScaLAPACK matrix ADP (A distributed packed).

The blocks of A will be stored horizontally in ADP so that the entries in the elementary, grid and distributed blocks are contiguous in memory and then will map better to the highest levels of cache.



Let us consider the following symmetric matrix A described using distributed blocks, that is

$$A = \begin{pmatrix} A_{11} & A_{21}^T & A_{31}^T \\ A_{21} & A_{22} & A_{32}^T \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

We provide two ways of storing A using our distributed packed format. In the lower distributed packed format, the lower triangle of A is packed *by columns* in ADP , i.e.

$$ADP = [A_{11} \quad A_{21} \quad A_{31} \quad A_{22} \quad A_{32} \quad A_{33}]$$

In the upper distributed packed format, the upper triangle of A is packed *by rows* in ADP , i.e.

$$ADP = [A_{11} \quad A_{21}^T \quad A_{31}^T \quad A_{22} \quad A_{32}^T \quad A_{33}]$$

The distributed packed storage for upper and lower triangular matrices follows from that of a symmetric matrix since the upper triangular blocked matrix

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & 0 & A_{33} \end{pmatrix}$$

is stored in a packed distributed format as

$$ADP = [A_{11} \quad A_{12} \quad A_{13} \quad A_{22} \quad A_{23} \quad A_{33}]$$

and the lower triangular blocked matrix

$$A = \begin{pmatrix} A_{11} & 0 & 0 \\ A_{21} & A_{22} & 0 \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

will be stored as

$$ADP = [A_{11} \quad A_{21} \quad A_{31} \quad A_{22} \quad A_{32} \quad A_{33}]$$

We point out that the matrix ADP corresponds 'physically' to a ScaLAPACK array that is laid out on the $p \times q$ mesh. We also specify that, in contrast to LAPACK where upper and lower triangular matrices are both packed by columns, our distributed packed format is different for upper and lower triangular matrices since they are packed by rows and columns, respectively. Note also that the diagonal blocks are full blocks and thus do not exploit the triangular or symmetric structure.

Throughout this paper we will use the following designations and notation. The distributed packed format will be simply referred to as the packed format and an implementation using this format as a packed implementation. We denote by $N = n/b$ the number of block rows in A . The packed structure ADP will be described using the A_{ij} as previously or will be denoted as the blocked matrix $[B_1 \quad B_2 \quad B_3 \quad B_4 \quad B_5 \quad B_6]$. A block $B_k = A_{ij}$ in ADP will be addressed through the indirect addressing INDGET that maps (i, j) to k (this mapping depends on whether a lower or an upper packed storage is chosen).



3.2. Tuning parameters

In order to obtain the best gigaflops performance, one may determine the dominant operation(s) involved in the computation in terms of the floating-point operation count (e.g. the matrix–matrix multiplication performed by the Level-3 routine DGEMM in the sequential Cholesky factorization). Then we try to optimize the efficiency of this dominant operation by tuning the program parameters. In parallel implementations, these user-tunable parameters are often the size s of an elementary block size and the values of p and q in the process grid.

If the DGEMM routine is the dominant operation, then s is generally determined as being the value that enables us to obtain the best sustained performance for a matrix–matrix product of size s on the target machine. This parameter is closely related to the machine characteristics and to the memory hierarchy constraints (level 1, 2, 3 cache or translation lookaside buffer (TLB)).

The optimal values for the parameters p and q generally depend on the type of algorithm that is implemented. In [20], optimal values for the ratio p/q are proposed for the LU, Cholesky and QR factorizations performed by ScaLAPACK.

Performance tuning can sometimes become more complicated when the dominant operation (i.e. the operation that must be optimized) changes with the distribution parameters or when the dominant operation in terms of flops is not the dominant operation in terms of time. In that case, a heuristic needs to be found that will often lead to a compromise (not degrading the most efficient routine while improving the less efficient one).

Finally, a parameter that influences the performance of a packed implementation is the size of b . As seen in Section 3.1, we have $b \geq l.s$. b may be chosen significantly larger than $l.s$ but in that case it would demand more memory storage because the diagonal blocks would be larger. The ratio between the memory storage required by a block size b and that required by a block size $l.s$ is given by

$$\alpha = \left(\frac{n}{b} \left(\frac{n}{b} + 1 \right) b^2 \right) / \left(\frac{n}{l.s} \left(\frac{n}{l.s} + 1 \right) (l.s)^2 \right) = \frac{n+b}{n+l.s}$$

Then the increase (in percentage) of the memory storage when using a blocking of size b instead of using a blocking of size $l.s$ is expressed by $\alpha - 1$, i.e. $(b - l.s)/(n + l.s)$. In the rest of this paper, this quantity is referred to as extra storage.

Remark 1. We did not compare the required storage using a blocking size b with the ‘ideal’ storage corresponding to the LAPACK packed storage described in Section 2, since we store the entire diagonal blocks here and our parallel implementation is based on distributed blocks whose minimum size is $l.s \times l.s$ for easy use of ScaLAPACK routines. Indeed, in our packed distributed storage, the choice $b = l.s$ is optimal from a memory point of view but, in general, an optimal packed storage would store $n(n+1)/2$ entries of the matrix.

Let ρ be the maximum extra storage that we are ready to accept. Then the maximum value of b will be such that $b \leq \rho(n + l.s) + l.s$. We will see that the choice of b will represent a trade-off between the performance (if large b improves the dominant operations) and the memory storage (b must be consistent with the maximum extra storage that we are ready to afford).

4. APPLICATION TO THE CHOLESKY FACTORIZATION

Based on the distributed packed storage defined in Section 3.1, we describe in this section how a packed distributed Cholesky factorization can be designed on top of PBLAS and ScaLAPACK kernels.



```
for  $i = 1 : N$ 
   $j = \text{INDGET}(i, i)$ 
   $B_j \leftarrow \text{chol}(B_j)$  (PDPOTRF)
   $B_{j+1:j+N-i} \leftarrow B_{j+1:j+N-i} B_j^{-T}$  (PDTRSM)
  for  $ii = i + 1 : N$ 
     $k = \text{INDGET}(ii, ii)$ 
     $B_k \leftarrow B_k - B_{j+ii-i} B_{j+ii-i}^T$  (PDSYRK rank-b update)
     $B_{k+1:k+N-ii} \leftarrow B_{k+1:k+N-ii} - B_{j+1+ii-i:j+N-i} B_{j+ii-i}^T$  (PDGEMM)
  end (ii-loop)
end (i-loop)
```

Algorithm 1. Packed right-looking Cholesky.

```
for  $i = 1 : N$ 
   $j = \text{INDGET}(i, i)$ 
  for  $ii = 1 : i - 1$ 
     $k = \text{INDGET}(ii, i)$ 
     $B_j \leftarrow B_j - B_k B_k^T$  (PDSYRK rank-b update)
     $B_{j+1:j+N-ii}^T \leftarrow B_{j+1:j+N-ii}^T - B_{k+1:k+N-i}^T B_k^T$  (PDGEMM)
  end (ii-loop)
   $B_j \leftarrow \text{chol}(B_j)$  (PDPOTRF)
   $B_{j+1:j+N-i}^T \leftarrow B_{j+1:j+N-i}^T B_j^{-T}$  (PDTRSM)
end (i-loop)
```

Algorithm 2. Packed left-looking Cholesky.

4.1. Description of the algorithms

The packed implementation of the Cholesky factorization is based on the Level-3 PBLAS routines PDGEMM (matrix–matrix product), PDSYRK (rank-k update), PDTRSM (solving triangular systems with multiple right-hand sides) and on the ScaLAPACK routine PDPOTRF (Cholesky factorization). We present in this section the packed implementations of the right-looking and left-looking variants of the Cholesky factorization that are given, respectively, in Algorithms 1 and 2. Note that Algorithm 1 operates on the lower triangular part of the matrix while Algorithm 2 operates on the upper triangular part.

The symmetric positive definite matrix partitioned into distributed blocks

$$\begin{pmatrix} B_1 & B_2^T & B_3^T \\ B_2 & B_4 & B_5^T \\ B_3 & B_5 & B_6 \end{pmatrix}$$



is stored in a lower distributed packed format as $[B_1 \ B_2 \ B_3 \ B_4 \ B_5 \ B_6]$ that we also denote by $B_{1:6}$.

We note in both algorithms that the PDGEMM and the PDTRSM instructions involve rectangular matrices. In the implementations, these instructions are performed using a loop that performs the multiplication of the $b \times b$ blocks one by one because it exhibited better performance in the experiments.

We also point out that when b is minimum ($b = l.s$), then N is maximum and thus the number of synchronizations involved in Algorithm 1 is maximum (these synchronizations occur, for instance, in the routine PDPOTF2 that performs the unblocked Cholesky factorization inside PDPOTRF due to the broadcast of the ‘INFO’ output parameter). This explains why, even if taking $b = l.s$ requires less memory storage, in practice this value of b will rarely be chosen.

4.2. Tuning

Both algorithms were implemented on an IBM pSeries 690 (two nodes of 32 processors Power-4/1.7 GHz and 64 GB memory per node) and are linked with the PBLAS and ScaLAPACK libraries provided by the vendor (in particular, the Pessl library).

As in a sequential blocked Cholesky algorithm, the matrix–matrix multiplication performed by the routines PDGEMM or PDSYRK represents the major part of the computation. Both routines call essentially the Level-3 BLAS routine DGEMM that gives good performance for $s \geq 128$ for our platform. The value of $s = 128$ will be taken in all of the following experiments.

4.2.1. Influence of the distributed block size

We now examine the influence of b on the performance on the kernel routines used in our packed implementation.

Figure 1 represents the unitary performance of each routine on a matrix of size b using a four-by-four process grid. The curves show that the performance increases with b . However, we notice that a spike occurs for $b = 8192$ and a smaller spike for $b = 4096$. Since $b = 8192$ corresponds to a local array per processor that has a leading dimension of 2048, the main spike is explained by cache misses occurring when we access to two consecutive double-precision real, in a row of an array whose leading dimension is a multiple of 2048 (due to the size of the IBM pSeries 690 L1 cache). The spike observed for $b = 4096$ corresponds to secondary cache misses. Details on this phenomenon are given in [4].

Regarding Algorithm 1, the repartition of floating-point operations among the different routines depends on the problem size n and on the block size b . However, the performance of each routine can also depend on b and on the number of processors involved in the computation.

Table I gives, for a particular example, the number of floating-point operations and the time spent in each routine for our packed implementation and for the full storage ScaLAPACK factorization. We note that the PDTRSM routine performs 8.1% of the operations and takes 41% of the time. This shows that, as observed in Figure 1, PDTRSM is far less efficient than PDGEMM and PDSYRK. We note that the number of floating-point operations corresponding to the Cholesky factorization of the diagonal blocks is negligible, as mentioned in [4]. These operations are performed by PDPOTRF in the packed implementation and by PDPOTF2 in ScaLAPACK. The corresponding figures are not mentioned in Table I because they do not represent a significant time in the global factorization. Then a heuristic for

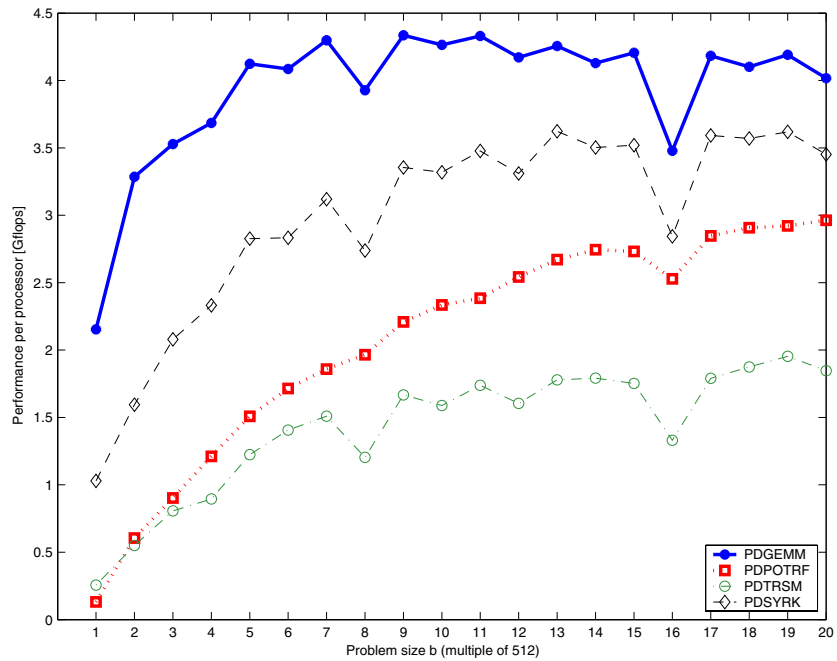


Figure 1. Performance of the PBLAS routines involved in the Cholesky factorization (16 processors).

Table I. Breakdown of operations and time for right-looking Cholesky ($n = 81920$, 64 processors).

PBLAS routine	Packed solver ($b = 5120$)		ScaLAPACK ($s = 128$)	
	Operations (%)	Time (%)	Operations (%)	Time (%)
PDGEMM and PDSYRK	91.9	58	99.8	92
PDTRSM	8.1	41	0.2	0.7

tuning the parameter b may consist of choosing an ‘acceptable’ ratio r of operations performed by the PDTRSM routine. The floating-point operations performed by PDTRSM are

$$\sum_{i=1}^{N-1} i b^3 = b^3 \frac{N(N-1)}{2}$$



Table II. Tuning parameters/heuristics for packed Cholesky implementation (IBM pSeries 690).

Parameter	Heuristic	Suggested value (IBM pSeries 690)
s	Sustained performance of DGEMM	128
p, q	$\frac{p}{q} > 1$	Depends on processor count
b	Accepted extra storage ρ Accepted PDTRSM operations r	$b \leq \min\left(\frac{4rn}{3 + \sqrt{9 - 24r}} b_{\max}, \rho(n + l.s) + l.s\right)$

Hence, since the Cholesky factorization involves $n^3/3 = (Nb)^3/3$ operations, we have

$$\frac{3(N-1)}{2N^2} \leq r$$

Thus, we get $N = n/b \geq (3 + \sqrt{9 - 24r})/4r$ and the maximum value for b is

$$b_{\max} = \frac{4rn}{3 + \sqrt{9 - 24r}}$$

As seen in Section 3.2, we have $b \leq \rho(n + l.s) + l.s$ and then the chosen value for b will be $\min(b_{\max}, \rho(n + l.s) + l.s)$.

4.2.2. Influence of the process grid

The $p \times q$ process grid can also have an influence on the performance of the code. In a packed implementation, the choice of a roughly squared grid ($\frac{1}{2} \leq p/q \leq 1$) proposed in [20] for the full storage Cholesky factorization is not necessarily the best grid choice for the packed format because the operation that slows down the global performance of the program is the PDTRSM routine. Figure 2 shows for a 16 processor grid that the PDTRSM routine applied to one block of size b is more efficient when using a rectangular grid such that $p/q > 1$ whatever the value of b . Table II summarizes the tuning parameters and the heuristics for the packed Cholesky implementation on the IBM pSeries 690.

4.3. Parallel performance results

In the following results, the processor count varies from 1 to 64 and the corresponding problem size n has been determined so that each processor uses about the same amount of memory (with $n = 10\,240$ for one processor, which corresponds to a storage of about 840 MB per processor for ScaLAPACK). For each problem size, the size b of the distributed block has been determined using the heuristic given in Section 4.2. We first compute for each problem size the maximum value of b that can be obtained by accepting a maximum of 10% of memory extra storage ρ and 15% of operations performed by the routine PDTRSM. Then b is adjusted to the nearest number that is lower to the maximum value, proportional to $l.s$ and a submultiple of n . The resulting values of b are given in Table III. The actual

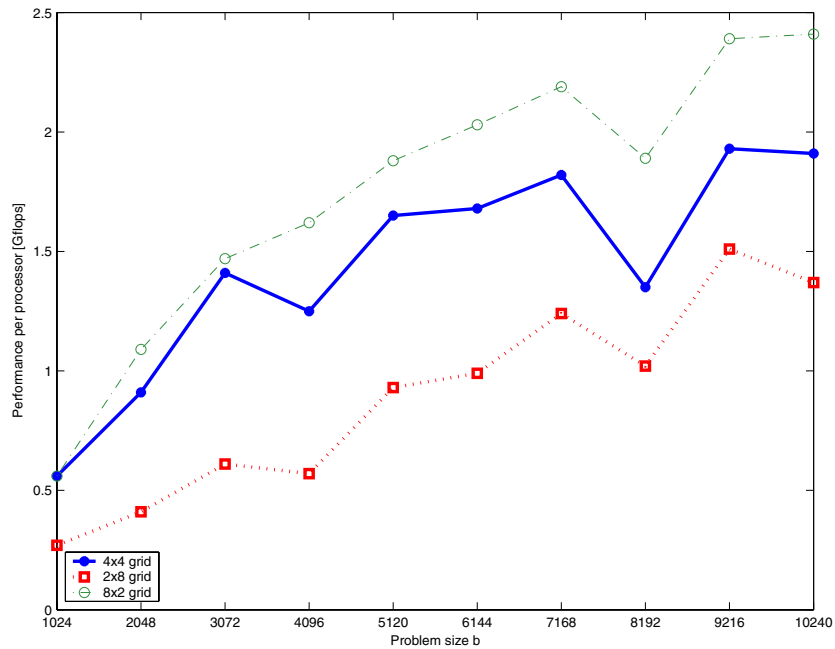


Figure 2. Performance of PDTRSM for different process grids (16 processors).

Table III. Tuned block size for $\rho = 0.1$ and $r = 0.15$.

	Problem size n						
	10 240	14 336	20 480	28 672	40 960	61 440	81 920
$p \times q$ grid	1×1	2×1	4×1	4×2	8×2	8×4	16×4
Block size b	1024	1024	2048	2048	4096	6144	10 240
Extra storage	8.6%	7%	7.3%	5.3%	7.3%	8.2%	9.7%

extra storage corresponding to the chosen value of b is the quantity $\alpha - 1 = (b - l.s)/(n + l.s)$ that has been defined in Section 3.2.

We present in Table IV performance results obtained by the right-looking implementation (Algorithm 1 described in Section 4.1) rather than the left-looking implementation (Algorithm 2) since it gives in our experiments factorization times that are slightly better for a high processor count. The selected values of b are those displayed in Table III. In accordance with Section 4.2, the grid parameters are such that $p/q > 1$ and more precisely $2 \leq p/q \leq 4$ since it provides experimentally better results. In Table IV, t_{packed} is the resulting factorization time.



Table IV. Cholesky factorization time (in seconds) for the packed solver and ScaLAPACK.

	Problem size n						
	10 240	14 336	20 480	28 672	40 960	61 440	81 920
Number of processors	1	2	4	8	16	32	64
t_{packed}	102	127	194	290	474	912	1298
$p \times q$	1	2×1	4×1	4×2	8×2	8×4	16×4
t_{scal}	106	153	219	321	471	890	1178
$p \times q$	1	1×2	2×2	2×4	4×4	4×8	8×8
Overhead	-3.8%	-17%	-11.4%	-9.7%	0.6%	2.5%	10%

Table V. Performance versus extra storage ($n = 81\,920$, 16×4 processors).

	Block size b	
	10 240	20 480
Factorization time (s)	1298	1234
Overhead with ScaLAPACK	10%	5%
Extra storage	9.7%	22%

In Table IV the performance of the packed solver is compared with that of a ScaLAPACK Cholesky factorization storing the whole matrix but performing the same number of operations (routine PDPOTRF). The corresponding factorization time t_{scal} is obtained using a $p \times q$ process grid in accordance with [20], i.e. such that $\frac{1}{2} \leq p/q \leq 1$. The difference in performance is then measured by computing the overhead $(t_{\text{packed}} - t_{\text{scal}})/t_{\text{scal}}$.

We notice in Table IV that the factorization times are better than ScaLAPACK for less than 32 processors and similar to ScaLAPACK for 32 processors. For 64 processors, there is an overhead of 10%. This overhead can be diminished by considering larger blocks. Table V shows that the performance increases with b but that it also requires more memory.

In order to evaluate the scalability of the packed solver and of the ScaLAPACK Cholesky, we plot in Figure 3 the performance in gigaflops of both algorithms. Since each algorithm maintains a constant memory use per processor, these curves measure what is referred to as isoefficiency or isogranularity in [1, p. 96]. We note that the packed solver is more efficient for a small processor count. Performance degrades for both algorithms when the number of processors increases, but the ScaLAPACK routine is slightly faster for 32 and 64 processors.

Remark 2. Since the memory required by the packed implementation depends on the chosen value for b , it is interesting to compare in detail in Table VI the memory per processor in Megabytes required by the packed solver and by the ScaLAPACK routine PDPOTRF for the experiments summarized

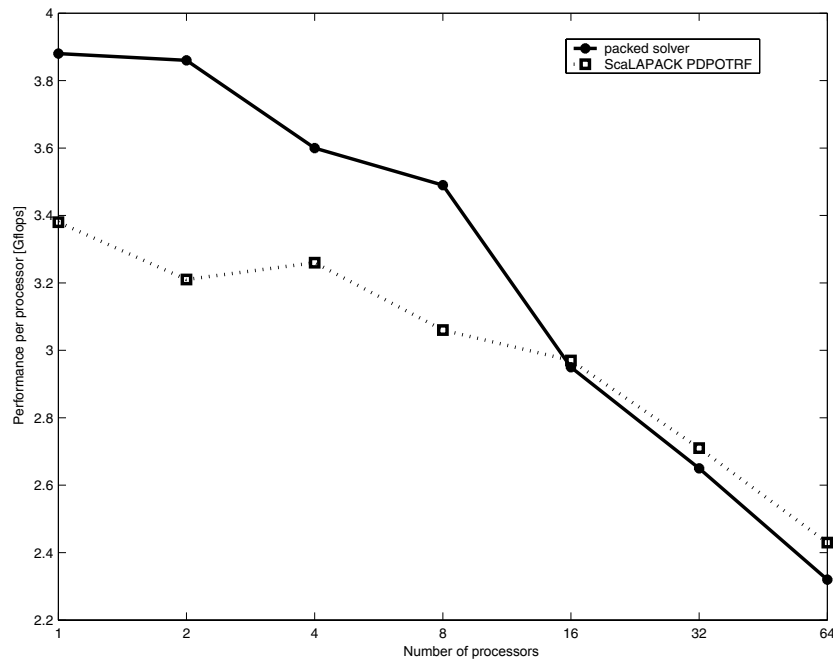


Figure 3. Isogranularity of packed Cholesky solver and ScaLAPACK on IBM pSeries 690.

Table VI. Memory required per processor by the packed solver and ScaLAPACK (in megabytes).

	Problem size n						
	10 240	14 336	20 480	28 672	40 960	61 440	81 920
Number of processors	1	2	4	8	16	32	64
Block size b	1024	1024	2048	2048	4096	6144	10 240
Memory for packed solver (MB)	461	440	461	440	461	519	472
Memory for PDPOTRF (MB)	839	822	839	822	839	943	839
Saved memory	45%	46%	45%	46%	45%	45%	44%

previously in Table IV and Figure 3. The quantities mentioned here do not include the work arrays used by ScaLAPACK computations. Note that an optimal storage that stores exactly $n(n+1)/2$ entries would require about 419 MB.

The experiments described in this section have been performed on nodes of IBM pSeries 690 with 2 GB of memory per processor. Due to the memory required by PDPOTRF (more than 850 MB), we could not achieve these comparisons on nodes having 1 GB of memory per processor (because part of



Table VII. Performance of packed implementation using 128 (16×8) processors ($n = 114\,688$).

	Block size b	
	16 384	28 672
Factorization time (s)	2741	3043
Gigaflops per processor	1.43	1.29
Extra storage	12.3%	22.8%

the memory is also used by the system). This confirms again the limitation due to the full storage for symmetric matrices.

Remark 3. By using four nodes of the IBM pSeries 690 (32-way SMP each) available at the CINES, we could evaluate how the performance of the packed solver degrades when running on 128 processors and with the same memory per processor as previously. We present in Table VII the performance obtained for $n = 114\,688$ and 16×8 processors. The best factorization time corresponds to about twice the time obtained using 64 processors. We note here that there is no interest in choosing a block size larger than 16 384 since the performance degrades and it requires more storage.

Regarding the Cholesky factorization, some other experiments were performed. They deserve to be mentioned here because they have influenced some choices made for the implementation described previously.

The first experiment investigated concerned the influence of the structure of the ScaLAPACK array for the packed structure. As mentioned in Section 3, the distributed blocks are stored row-wise in the ScaLAPACK array *ADP*. This choice is justified by the fact that the operations in Algorithm 1 are performed by column and thus the blocks are contiguous in memory. In Figure 4, we use 16 processors on the same node of the IBM machine and plot the performance in gigaflops of a packed implementation of the Cholesky factorization using either a row-wise or column-wise storage scheme for the distributed blocks. It confirms that a block-row storage provides better performance, thanks to a better data locality.

Some experiments were also performed in order to compare the right-looking and the left-looking variants of the packed implementation (using an horizontal structure for *ADP*).

In the left-looking variant, the matrix A is stored compactly using the upper distributed packed format. This enables us to have memory contiguity of the blocks belonging to a same block-row in Algorithm 2. Table VIII contains the factorization times obtained for both algorithms and shows that the left-looking implementation is slightly better when using less than 16 processors and that the right-looking implementation provides better results when using more than 32 processors.

4.4. Experiments on clusters

Several experiments were performed on the CRAY XD1 cluster at CERFACS (120 AMD Opteron 2.4 GHz, 240 GB memory, scientific library acml 3.0).

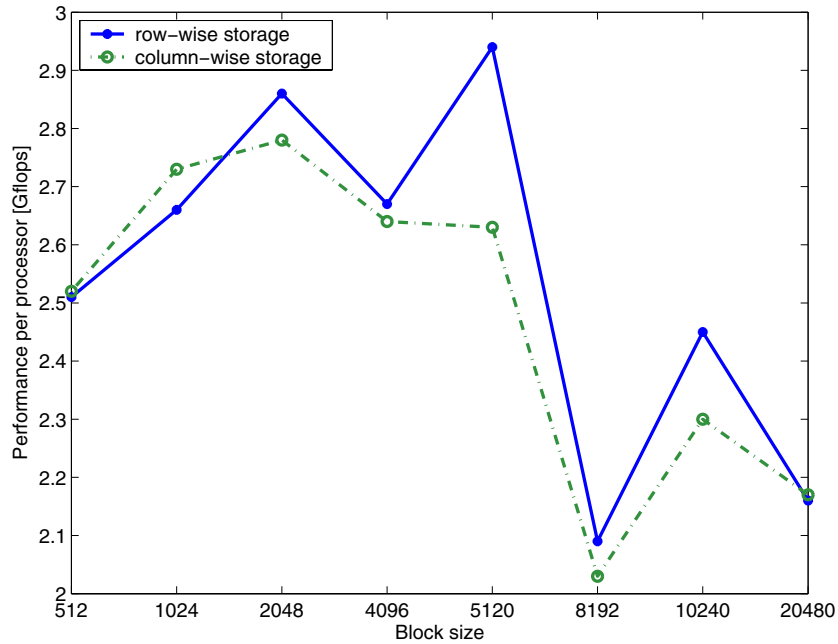


Figure 4. Row-wise storage versus column-wise storage in the distributed packed format.

Table VIII. Factorization time (s) for left-looking and right-looking variants of the packed distributed Cholesky.

	Problem size n						
	10 240	14 336	20 480	28 672	40 960	61 440	81 920
Number of processors	1	2	4	8	16	32	64
Block size b	256	256	512	1024	1024	5120	5120
Left-looking	87	123	194	283	465	966	1578
Right-looking	90	146	222	306	548	1092	1442

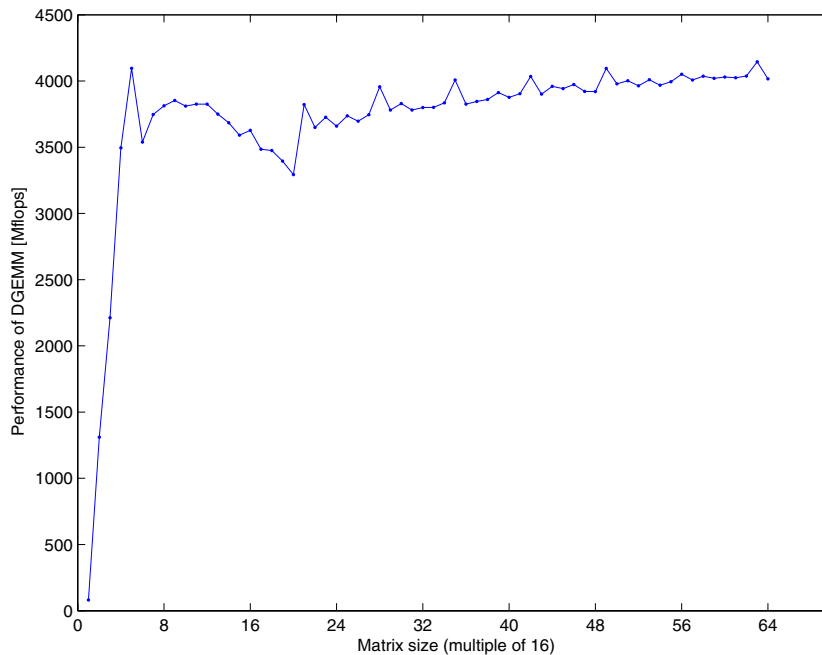


Figure 5. Performance of a serial matrix–matrix multiplication DGEMM on one processor of the CRAY Cluster XD1.

We consider problem sizes similar to those defined in Section 4.3 for the IBM pSeries 690, i.e. by considering a constant memory per processor.

As shown in Figure 5, the sustained peak rate of a serial matrix–matrix product DGEMM on this machine is about 4 Gflops. Taking $s = 128$ is here again satisfactory because it provides a DGEMM rate of 3.8 Gflops.

Since the PBLAS routine PDGEMM represents the major part of the computation in a parallel Cholesky factorization, we also tune the parameters of this routine for several problem sizes. Figure 6 shows that, in contrast to the IBM pSeries 690, the performance of PDGEMM degrades significantly when the problem size increases for a given number of processors (four here). This can be explained by the slower communication system on this platform. This encourages us to choose smaller block sizes than in Section 4.3 for the packed implementation.

We also note that a $p \times q$ rectangular grid such that $p > q$ gives better results than a square grid. This has a consequence on the choice of grid for the Cholesky factorization. Table IX contains the factorization times obtained for PDPOTRF using $p \times q$ process grids with $p = q$ and $p > q$. We observe that a rectangular grid provides a performance that is twice that of a square grid when we consider four or 16 processors. That leads us to consider the same grids for ScaLAPACK and the packed solver.

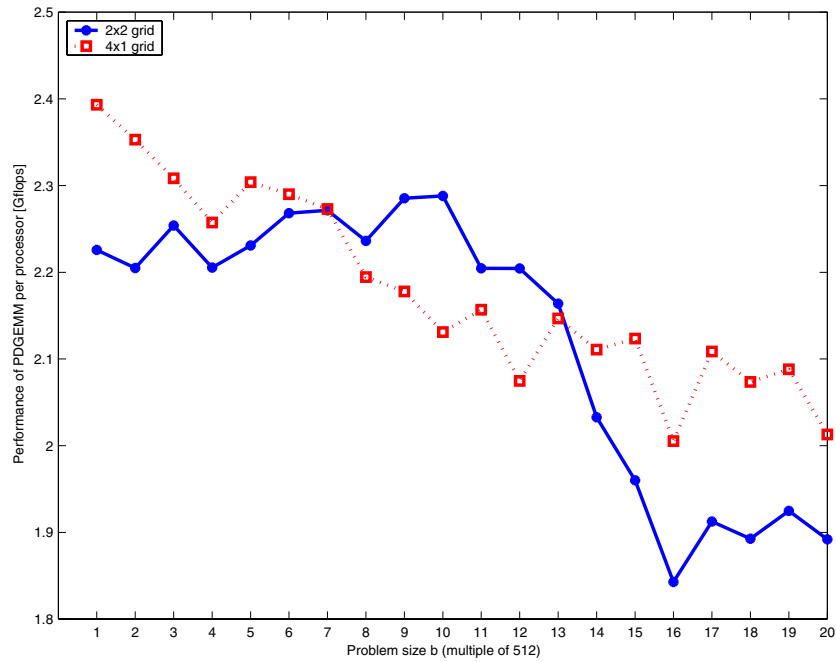


Figure 6. Performance of PDGEMM routine for two grid shapes on the CRAY Cluster XD1.

Table IX. Influence of the process grid on ScaLAPACK PDPOTRF performance (CRAY XD1).

	Problem size n		
	20 480	40 960	81 920
$p \times q$	2×2	4×4	8×8
Factorization time (s)	574	1073	2292
$p \times q$	4×1	8×2	16×4
Factorization time (s)	255	561	1622



Table X. Cholesky factorization time (s) for the packed solver and ScaLAPACK (CRAY XD1).

	Problem size n							
	10 240	14 336	20 480	28 672	40 960	61 440	81 920	107 520
Number of processors	1	2	4	8	16	32	64	112
$p \times q$	1	2×1	4×1	4×2	8×2	8×4	16×4	28×4
b	128	256	512	512	1024	1024	2048	3584
t_{packed}	113	168	270	419	616	1127	1473	2022
t_{scal}	153	184	278	370	561	1005	1399	1776
Overhead	-26%	-9%	-3%	13%	10%	12%	5%	14%

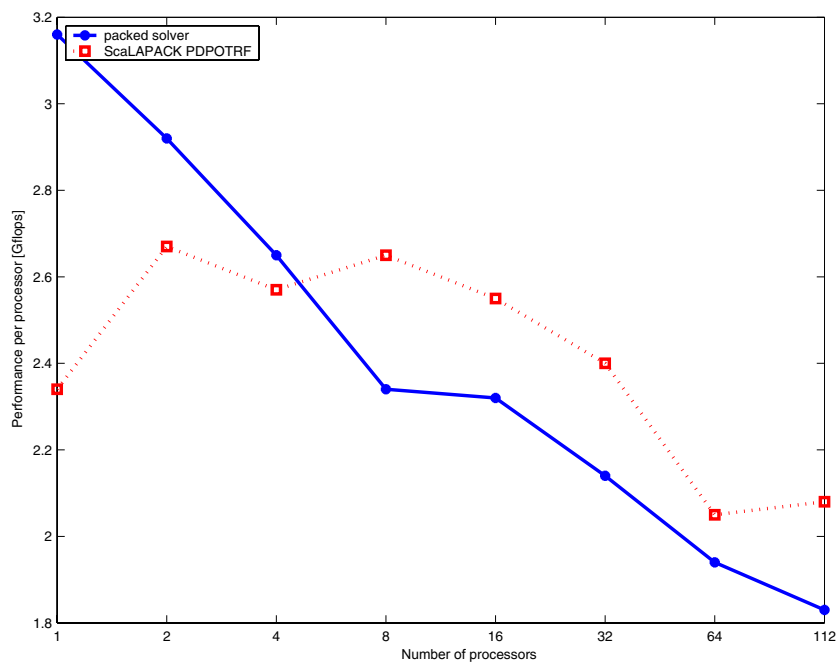


Figure 7. Isogranularity of packed Cholesky solver and ScaLAPACK on the CRAY XD1 cluster.

After tuning the grid shapes for ScaLAPACK, we now compare the performance of the packed implementation and the ScaLAPACK routine PDPOTRF. Table X shows that ScaLAPACK performance is slightly better than that of the packed solver for more than eight processors. The overhead $(t_{\text{packed}} - t_{\text{scal}})/t_{\text{scal}}$ is always lower than 14% and can be considered as acceptable.



The isogranularity of each algorithm measured in gigaflops per second is depicted in Figure 7. Similarly to the IBM pSeries 690, the packed solver is more efficient for a small processor count and the ScaLAPACK Cholesky provides better gigaflops rates when using more than eight processors while having similar behaviour when the number of processors increases.

To summarize these experiments on the cluster, we note that the tuning of the distributed block is simplified because of the monotonic decrease in performance of the PDGEMM routine (see Figure 6). This implies that we should use small blocks for the packed implementation. This has the advantage of requiring minimal memory.

On the other hand, the performance of the ScaLAPACK routine PDPOTRF is improved by considering rectangular grids that are not common for the Cholesky factorization. Then PDPOTRF gives slightly better performance than the packed solver when using more than eight processors (up to 14% for 112 processors).

5. CONCLUSION

The distributed packed storage defined in this paper allows us to handle symmetric and triangular matrices in parallel distributed environments using ScaLAPACK and Level-3 PBLAS routines. The example of the Cholesky factorization shows that choosing the optimal distributed block size leads to a trade-off between performance and memory. Some heuristics have been proposed that provide gigaflops performance similar to ScaLAPACK while requiring much less memory. In general, the performance of our packed implementations relies on the performance of the underlying ScaLAPACK kernel routines. The good results that we obtained encourage us to extend this packed storage to other linear algebra calculations involving symmetric or triangular matrices. An improvement might result from extending the rectangular full packed storage that was recently defined in [17] for serial implementations to blocked parallel implementations. Such a format could enable us to minimize the storage required by the diagonal blocks but it will be necessary to evaluate the performance of the ScaLAPACK kernel routines on this format. This will be the topic of further studies.

ACKNOWLEDGEMENTS

We would like to thank the CINES (Centre Informatique National de l'Enseignement Supérieur, Montpellier, France) for allowing us to perform experimentation on its parallel computers. We wish to express gratitude to Jack Dongarra (University of Tennessee) and to Fred Gustavson (IBM T. J. Watson Research Center) for fruitful discussions. We also wish to thank the referees for valuable comments on the manuscript.

REFERENCES

1. Blackford L *et al.* *ScaLAPACK User's Guide*. SIAM: Philadelphia, PA, 1997.
2. Anderson E *et al.* *LAPACK User's Guide* (3rd edn). SIAM: Philadelphia, PA, 1999.
3. Demmel J, Dongarra J. *LAPACK 2005 Prospectus: Reliable and scalable software for linear algebra computations on high end computers*. *LAPACK Working Note 164*, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, February 2005.



4. Baboulin M, Giraud L, Gratton S. A parallel distributed solver for large dense symmetric systems: Applications to geodesy and electromagnetism problems. *International Journal of High Performance Computing Applications* 2005; **19**(4):353–363.
5. Choi J, Dongarra J, Ostrouchov L, Petitet A, Walker D, Whaley R. A proposal for a set of parallel basic linear algebra subprograms. *LAPACK Working Note 100*, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, 1995.
6. Hendrickson BA, Womble DE. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM Journal of Scientific Computing* 1994; **15**(5):1201–1226.
7. Stewart GW. Communication and matrix computations on large message passing systems. *Parallel Computing* 1990; **16**:27–40.
8. Dongarra J, Du Croz J, Duff I, Hammarling S. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software* 1990; **16**:1–17.
9. Andersen B, Gunnels J, Gustavson F, Reid J, Waśniewski J. A fully portable high performance minimal storage hybrid Cholesky algorithm. *ACM Transactions on Mathematical Software* 2005; **31**(2):201–207.
10. IBM Engineering and Scientific Subroutine Library for AIX, version 3, Vol. 1. *Publication No. SA22-7272-0*, IBM, December 2001. Available at: http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/IBMp690/IBM/usr/lpp/essl.html.en_US/html/essl.html.
11. Andersen B, Gustavson F, Waśniewski J. A recursive formulation of Cholesky factorization of a matrix in packed storage. *ACM Transactions on Mathematical Software* 2001; **27**(2):214–244.
12. Gunter B, Reiley W, van de Geijn R. Parallel out-of-core Cholesky and QR factorizations with POOCLAPACK. *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society Press: Los Alamitos, CA, 2001.
13. Gunter B, van de Geijn R. Parallel out-of-core computation and updating of the QR factorization. *ACM Transactions on Mathematical Software* 2005; **31**(1):60–78.
14. Gustavson F, Toledo S. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computation. *Proceedings of the 4th Workshop on I/O in Parallel and Distributed Systems (IOPADS'96)*, Philadelphia, PA, 1996. IEEE Computer Society Press: Los Alamitos, CA, 1996.
15. Joffrain T, Quintana-Ortí E, van de Geijn R. Rapid development of high-performance out-of-core solvers for electromagnetics. *Proceedings of State-of-the-art in Scientific Computing: 7th International Workshop on Applied Parallel Computing (PARA 2004)*, Lyngby, Copenhagen, 20–23 June 2004 (*Lecture Notes in Computer Science*, vol. 3732), Dongarra J, Madsen K, Wasniewski J (eds.). Springer: Berlin, 2006; 414–423.
16. Dongarra J, Whaley R. A user's guide to the BLACS v1.1. *LAPACK Working Note 94*, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, 1997.
17. Gustavson FG. New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms. *Proceedings of State-of-the-art in Scientific Computing: 7th International Workshop on Applied Parallel Computing (PARA 2004)*, Lyngby, Copenhagen, 20–23 June 2004 (*Lecture Notes in Computer Science*, vol. 3732), Dongarra J, Madsen K, Wasniewski J (eds.). Springer: Berlin, 2006; 11–20.
18. D'Azevedo E, Dongarra J. Packed storage extension for ScaLAPACK. *LAPACK Working Note 135*, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, 1998.
19. van de Geijn R. *Using PLAPACK*. MIT Press: Cambridge, MA, 1997.
20. Choi J, Dongarra J, Ostrouchov L, Petitet A, Walker D, Whaley R. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming* 1996; **5**:173–184.