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

Marc Baboulin* Serge Gratton* Luc Giraud† Julien Langou‡

Abstract

We propose in this paper 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 to a full storage for a wide range of operations. This work has been motivated by the fact that, contrary to sequential linear algebra libraries (e.g. LAPACK [4]), there is no routine, no format that handles packed matrices in the current parallel distributed libraries available. The proposed algorithms exclusively use the existing ScaLAPACK [6] computational kernels which proves the generality of the approach, provides easy portability of the code, efficient re-use of existing software. The performance results obtained for the Cholesky factorization show that our packed format performs as good or better than the ScaLAPACK full algorithm for small numbers of processors. For larger number of processors, the ScaLAPACK full storage routine performs slightly better until each processor runs out of its memory.

Keywords: scientific computing, dense linear algebra, parallel distributed algorithms, ScaLAPACK, packed storage format, Cholesky factorization, QR factorization.

1 Introduction

Even though the current parallel platforms provide increasing memory capacity, they are also used to solve ever larger dense linear systems. This is the case for instance in geosciences or electromagnetic computations where the usual problem size is several hundreds of thousands requiring several tens of Gbytes. It is now possible to perform these calculations since the distributed memory parallel machines available today offer several Gbytes memory per processor. But when the dense matrices involved in these computations are symmetric, Hermitian or triangular, it could be worth exploiting the structure by storing only half the matrix.
The ScaLAPACK [6] library has been designed to perform linear algebra parallel calculations on dense matrices. Contrary to the serial library LAPACK [4], ScaLAPACK does not currently support packed format for symmetric, Hermitian or triangular matrices [10]. A parallel solver has been studied in [5] 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 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 [6, p. 58]. The distributed packed storage format proposed in this paper uses ScaLAPACK and PBLAS [7] routines. Each of these routines exploits the good load balancing of the two-dimensional block cyclic distribution [17, 19] as it is

*CERFACS, 42 avenue Gaspard Coriolis, 31057 Toulouse Cedex, France. Email : baboulin@cerfacs.fr - gratton@cerfacs.fr

†ENSEEIHT, 2 rue Camichel, 31071 Toulouse cedex, France. Email : giraud@n7.fr

‡University of Tennessee, 1122 Volunteer Blvd, Knoxville TN 37996-3450, USA. Email : langou@cs.utk.edu

implemented in ScaLAPACK [6, 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 [11] provide a packed storage for symmetric, hermitian or triangular matrices. This format allows us to store half 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 $AP = \left[\; a_{11}, a_{12}, a_{22}, a_{13}, a_{23}, a_{33} \; \right]$.
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),
- 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 to the full storage (see e.g [2]). Another approach was used successfully in the IBM library ESSL [1] that consists of writing Level-3 BLAS for packed format. Still another approach was used in [3] that defines a Recursive Packed Cholesky that operates on a so-called Recursive Packed Format (RPF) and requires variants of TRSM and SYRK 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

$$\left[\; A_{11}, A_{12}, A_{22}, A_{13}, A_{23}, A_{33} \; \right]$$

or

$$\left[\; A_{11}, A_{12}, A_{13}, A_{22}, A_{23}, A_{33} \; \right].$$

For serial implementations, the authors of [2] define a so-called Upper (resp. Lower) Blocked Hybrid Format. In both formats, the blocks are ordered by columns to permit efficient operations

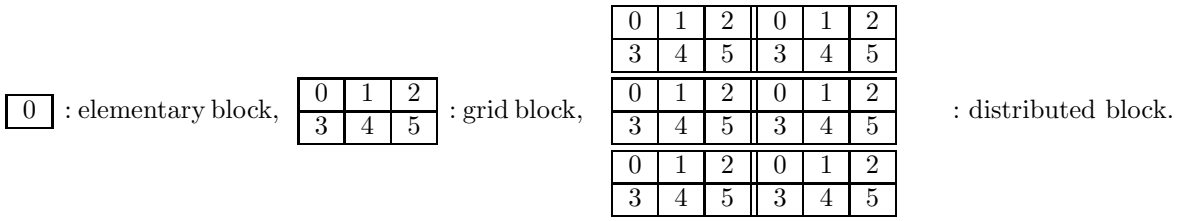
on blocks using Level-3 BLAS (e.g blocked Cholesky algorithm in [2]) and the diagonal blocks are stored in packed format so that exactly half of the matrix is stored.
Regarding parallel implementations for distributed memory architectures, out-of-core parallel solvers were implemented in several projects [13, 14, 15, 18] where only the blocks of one triangular part are stored. But for parallel in-core implementations, there is presently no satisfying packed storage available for dense matrices. A packed storage for symmetric matrices distributed in a 1-D block cyclic column distribution is used in [5] 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 [12].
Two algorithms using 2-D block cyclic data layouts are described in [16] 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 [9]. 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 stores also the entire diagonal blocks. We can find in [9] 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 cyclic manner according to a $p \times q$ process grid. We refer to [6] 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 least 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 [20] as algorithmic blocking or tile for out-of-core implementations [14].
The following figure 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 blocks that we get when we consider a 2×3 process grid, $s = 1$, $b = 6$ and each block is labeled with the number of process that stores it.



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 in 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:

$$ADP = \begin{bmatrix} A_{11} & A_{21} & A_{31} & A_{22} & A_{32} & A_{33} \end{bmatrix}.$$

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

$$ADP = \begin{bmatrix} A_{11} & A_{21}^T & A_{31}^T & A_{22} & A_{32}^T & A_{33} \end{bmatrix}.$$

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

$$ADP = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{22} & A_{23} & A_{33} \end{bmatrix}$$

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

$$ADP = \begin{bmatrix} A_{11} & A_{21} & A_{31} & A_{22} & A_{32} & A_{33} \end{bmatrix}.$$

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, contrary 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 respectively packed by rows and columns. 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 notations. 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 $N = \frac{n}{b}$ 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 $\begin{bmatrix} B_1 & B_2 & B_3 & B_4 & B_5 & B_6 \end{bmatrix}$. 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 whether a lower or an upper packed storage is chosen).

3.2 Tuning parameters

In order to obtain the best Gflops performance, one may determines the dominant operation(s) involved in the computation in terms of 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 TLB).

The optimal values for the parameters p and q generally depend on the type of algorithm that is implemented. In [8] optimal values for the ratio $\frac{p}{q}$ 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 bigger. The ratio between the memory storage required by a block size b and that required by a block size $l.s$ is given by

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

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 $\frac{b-l.s}{n+l.s}$. 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 here the entire diagonal blocks 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 improve 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.

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 $\begin{bmatrix} B_1 & B_2 & B_3 & B_4 & B_5 & B_6 \end{bmatrix}$ that we also denote by $B_{1:6}$.

We notice 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.

Algorithm 1. *:* Packed right-looking Cholesky

for $i = 1 : N$

$j = INDGET(i, i)$

$B_j \leftarrow chol(B_j)$ *(***PDPOTRF***)*

$B_{j+1:j+N-i} \leftarrow B_{j+1:j+N-i} B_j^{-T}$ *(***PDTRSM***)*

for $ii = i + 1 : N$

$k = INDGET(ii, ii)$

$B_k \leftarrow B_k - B_{j+ii-i} B_{j+ii-i}^T$ *(***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 2. *:* Packed left-looking Cholesky

for $i = 1 : N$

$j = INDGET(i, i)$

for $ii = 1 : i - 1$

$k = INDGET(ii, i)$

$B_j \leftarrow B_j - B_k B_k^T$ *(***PDSYRK** *rank-b update)*

$B^T_{j+1:j+N-ii} \leftarrow B^T_{j+1:j+N-ii} - B^T_{k+1:k+N-i} B_k^T$ *(***PDGEMM***)*

end *(ii-loop)*

$B_j \leftarrow chol(B_j)$ *(***PDPOTRF***)*

$B^T_{j+1:j+N-i} \leftarrow B^T_{j+1:j+N-i} B_j^{-T}$ *(***PDTRSM***)*

end *(i-loop)*

4.2 Tuning

Both algorithms were implemented on an IBM pSeries 690 (2 nodes of 32 processors Power-4/1.7 GHz and 64 Gbytes memory per node) and 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 multiply 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 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 4-by-4 process grid. The curves show that the performance increases with b. However we notice that a spike occurs for $b = 8192$ and a smaller one for $b = 4096$. Since $b = 8192$ corresponds to 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 [5].

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. But the performance of each routine can also depend on b and on the number of processors involved in the computation.
Table 1 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 notice 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

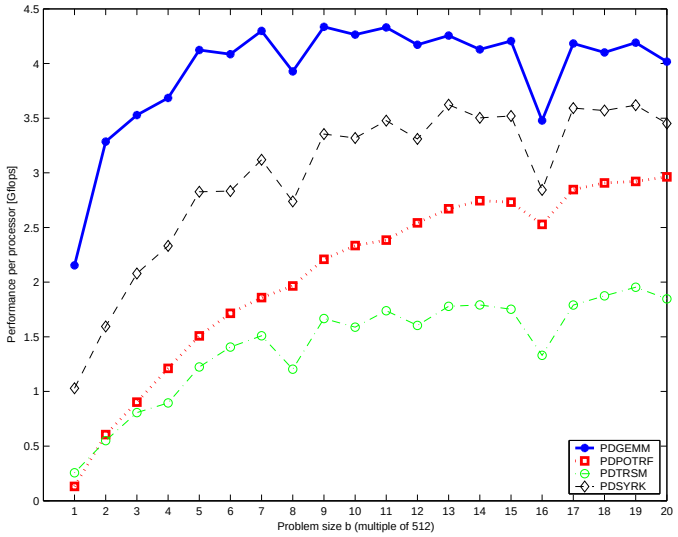


Figure 1: Performance of PBLAS routines involved in the Cholesky factorization (16 processors).

the diagonal blocks is negligible, as mentioned in [5]. These operations are performed by PDPOTRF in the packed implementation and by PDPOTF2 in ScaLAPACK. The corresponding figures are not mentioned in Table 1 because they do not represent a significant time in the global factorization. Then a heuristic for tuning the parameter b may consist in choosing an "acceptable" ratio r

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

Table 1: Breakdown of operations and time for right-looking Cholesky ($n = 81920$, 64 processors).

of operations performed by the PDTRSM routine. The floating-point operations performed by PDTRSM are:

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

Hence, since the Cholesky factorization involves $\frac{n^3}{3} = \frac{(Nb)^3}{3}$ operations, we have

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

Thus we get $N = \frac{n}{b} \geq \frac{3+\sqrt{9-24r}}{4r}$ 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 ($\frac{1}{2} \leq \frac{p}{q} \leq 1$) proposed in [8] 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 $\frac{p}{q} > 1$ whatever the value of b is. Table 2

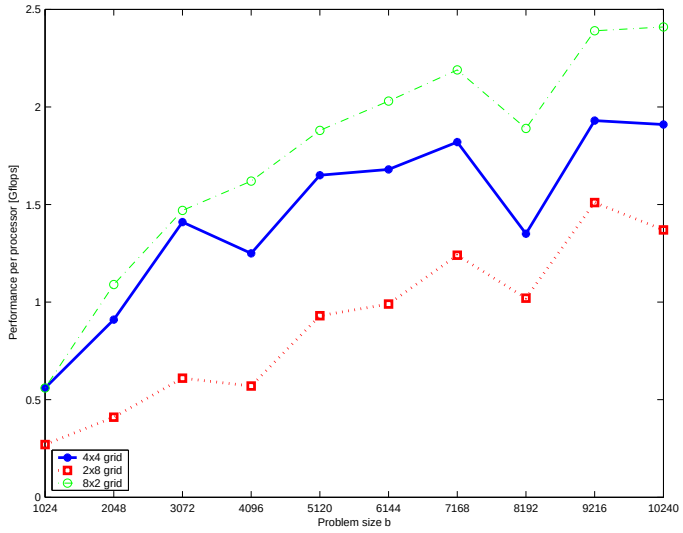


Figure 2: Performance of PDTRSM for different process grids (16 processors).

summarizes the tuning parameters and the heuristics for the packed Cholesky implementation on the 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	$b \leq \min(\frac{4rn}{3+\sqrt{9-24r}} b_{max}, \rho(n + l.s) + l.s)$

Table 2: Tuning parameters/heuristics for packed Cholesky implementation (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 $n = 10240$ for 1 processor, which corresponds to a storage of about 840 Mbytes 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 3. The actual extra-storage corresponding to the chosen value of b is the quantity $\alpha - 1 = \frac{b-l.s}{n+l.s}$ that has been defined in Section 3.2.

problem size n	10240	14336	20480	28672	40960	61440	81920
$p \times q$ grid	1×1	2×1	4×1	4×2	8×2	8×4	16×4
block size b	1024	1024	2048	2048	4096	6144	10240
extra-storage	8.6%	7%	7.3%	5.3%	7.3%	8.2%	9.7%

Table 3: Tuned block size for $\rho = 0.1$ and $r = 0.15$.

We present below performance results obtained by the right-looking implementation (Algorithm 1 described in Section 4.1) rather than the left-looking one (Algorithm 2) since it gives in our experiments factorization times that are slightly better for high processor count. The selected values of b are those displayed in Table 3. In accordance with Section 4.2, the grid parameters are such that $\frac{p}{q} > 1$ and more precisely $2 \leq \frac{p}{q} \leq 4$ since it provides experimentally better results. In that table, t_{packed} is the resulting factorization time.
In Table 4 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 [8] i.e such that $\frac{1}{2} \leq \frac{p}{q} \leq 1$. The difference in performance is then measured by computing the overhead $\frac{t_{packed}-t_{scal}}{t_{scal}}$.

n	10240	14336	20480	28672	40960	61440	81920
# procs	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 4: Cholesky factorization time (sec) for packed solver and ScaLAPACK.

We notice in Table 4 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 5 shows that the performance increases with b but that it also requires more memory.

block size b	10240	20480
factorization time (sec)	1298	1234
overhead with ScalaPACK	10%	5%
extra-storage	9.7%	22%

Table 5: Performance vs extra-storage ($n = 81920, 16 \times 4$ procs).

In order to evaluate the scalability of the packed solver and of the ScaLAPACK Cholesky, we plot in Figure 3 the Gflops performance 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 [6, p. 96]. We notice that the packed solver is more efficient for 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 6 the memory per processor in Mbytes

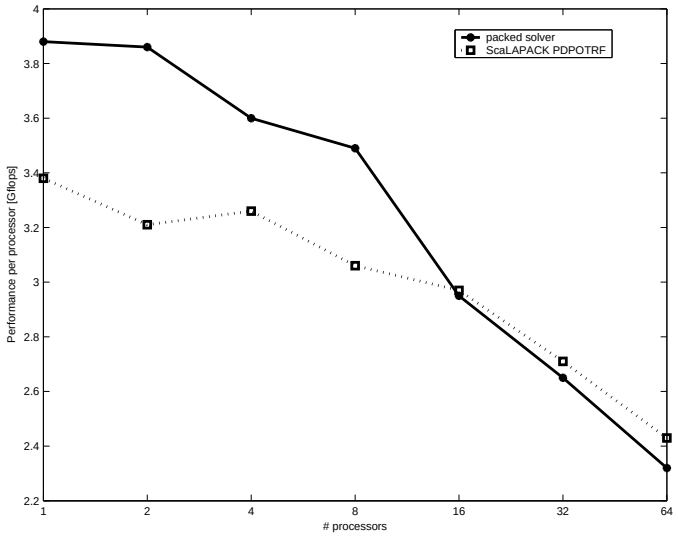


Figure 3: Isogranularity of packed Cholesky solver and ScaLAPACK on IBM pSeries 690.

required by the packed solver and by the ScaLAPACK routine PDPOTRF for the experiments summarized previously in Table 4 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 Mbytes.
The experiments described in this paragraph have been performed on nodes of IBM pSeries 690 with 2 Gbytes memory per processor. Due to the memory required by PDPOTRF (more than 850 Mbytes), we could not achieve these comparisons on nodes having 1 Gbytes memory per processor (because part of the memory is also used by the system). This confirms again the limitation due to the full storage for symmetric matrices.

n	10240	14336	20480	28672	40960	61440	81920
# procs	1	2	4	8	16	32	64
block size b	1024	1024	2048	2048	4096	6144	10240
Mbytes for packed solver	461	440	461	440	461	519	472
Mbytes for PDPOTRF	839	822	839	822	839	943	839
saved memory	45%	46%	45%	46%	45%	45%	44%

Table 6: Memory required per processor by the packed solver and ScaLAPACK (Mbytes).

Remark 3. By using 4 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 7 the performance obtained for $n = 114688$ and 16×8 processors. The best factorization time corresponds to about twice the time obtained using 64 processors. We notice that there is here no interest in choosing a block size larger than 16384 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.

block size b	16384	28672
factorization time (sec)	2741	3043
Gflops per proc.	1.43	1.29
extra-storage	12.3%	22.8%

Table 7: Performance of packed implementation using 128 (16×8) processors ($n = 114688$).

The first experiment investigated was about 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 Gflops 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.

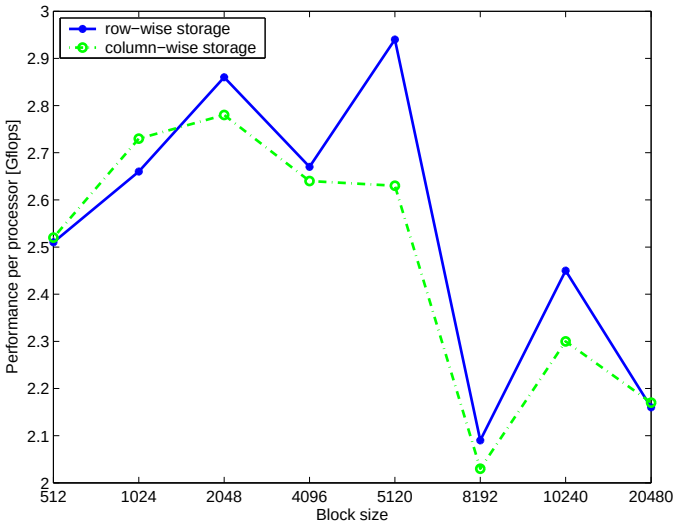


Figure 4: row-wise storage vs column-wise storage in the distributed packed format.

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 8 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 Gbytes memory, scientific library acml 3.0).
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.

n	10240	14336	20480	28672	40960	61440	81920
# procs	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

Table 8: Factorization time (sec) for left-looking and right-looking variants of the packed distributed Cholesky.

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.

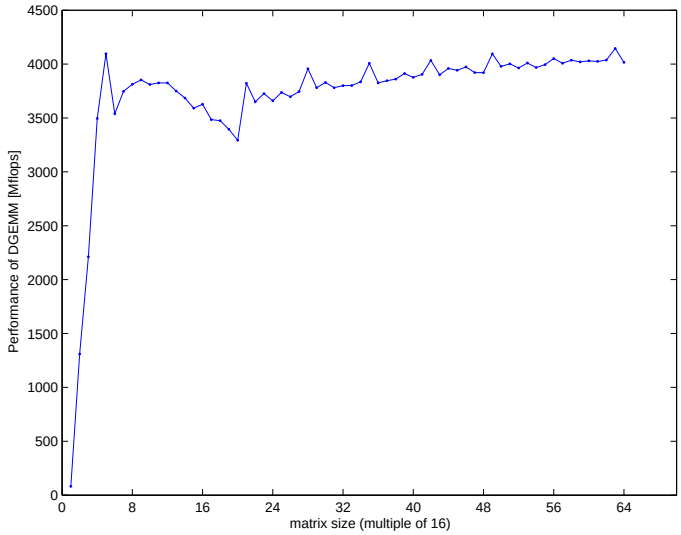


Figure 5: Performance of serial matrix-matrix multiply DGEMM on one processor of the CRAY Cluster XD1.

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 (here 4). 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 notice 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 9 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 4 or 16 processors. That leads us to consider the same grids for ScaLAPACK and the packed solver.

After tuning the grid shapes for ScaLAPACK, we now compare the performance of the packed implementation and the ScaLAPACK routine PDPOTRF. Table 10 shows that ScaLAPACK performance is slightly better than that of the packed solver for more than 8 processors. The overhead $\frac{t_{packed}-t_{scal}}{t_{scal}}$ is always lower than 14 % and can be considered as acceptable.

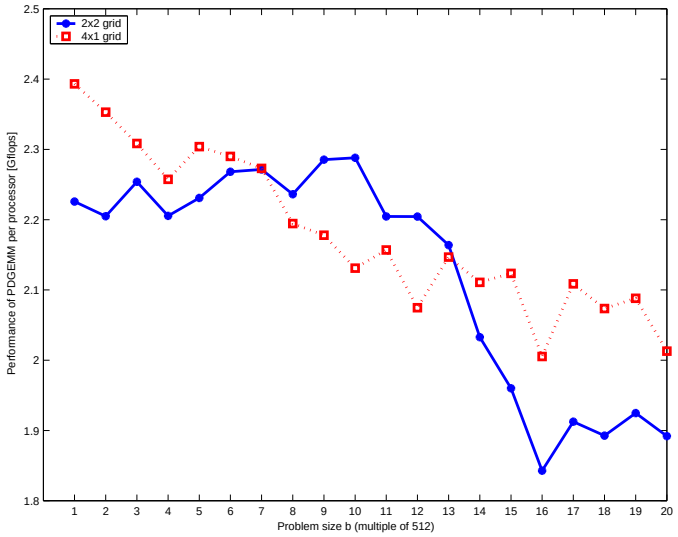


Figure 6: Performance of PDGEMM routine for 2 grid shapes on the CRAY Cluster XD1.

n	20480	40960	81920
$p \times q$	2×2	4×4	8×8
Factorization time (sec)	574	1073	2292
$p \times q$	4×1	8×2	16×4
Factorization time (sec)	255	561	1622

Table 9: Influence of the process grid on ScaLAPACK PDPOTRF performance (CRAY XD1).

n	10240	14336	20480	28672	40960	61440	81920	107520
procs	1	2	4	8	16	32	64	112
$p \times q$	1	2x1	4x1	4x2	8x2	8x4	16x4	28x4
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%

Table 10: Cholesky factorization time (sec) for packed solver and ScaLAPACK (CRAY XD1).

The isogranularity of each algorithm measured in Gflops per second is depicted in Figure 7. Similarly to the IBM pSeries 690, the packed solver is more efficient for small processor count and the ScaLAPACK Cholesky provides better Gflops rates when using more than 8 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 8 processors (up to 14% for 112 processors).

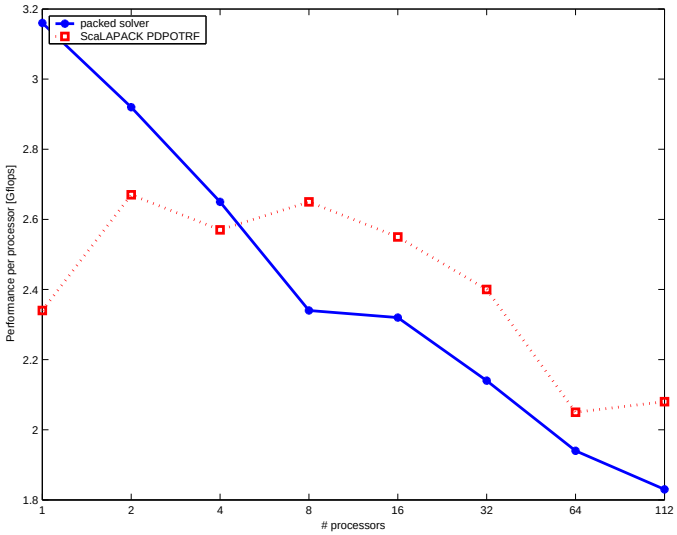


Figure 7: Isogranularity of packed Cholesky solver and ScaLAPACK on CRAY XD1 Cluster.

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 Gflops 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 to blocked parallel implementations the Rectangular Full Packed storage that was recently defined in [16] for serial 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.

Acknowledgments

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] *IBM engineering and scientific subroutine library for AIX*, version 3, volume 1. Pub. No. SA22-7272-0.

[2] B. Andersen, J. Gunnels, F. Gustavson, J. Reid, and J. Waśniewski, *A fully portable high performance minimal storage hybrid Cholesky algorithm*, ACM Trans. Math. Softw. **31** (2005), no. 2, 201–207.

[3] B. Andersen and F. Gustavson J. Waśniewski, *A recursive formulation of Cholesky factorization of a matrix in packed storage*, ACM Trans. Math. Softw. **27** (2001), no. 2, 214–244.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK user's guide*, SIAM, 1999, Third edition.

[5] M. Baboulin, L. Giraud, and S. Gratton, *A parallel distributed solver for large dense symmetric systems: applications to geodesy and electromagnetism problems*, Int. J. of High Performance Computing Applications **19** (2005), no. 4, 353–363.

[6] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley, *ScaLAPACK user's guide*, SIAM, 1997.

[7] J. Choi, J. Dongarra, L. Ostrouchov, A. Petitet, D. Walker, and R. Whaley, *A proposal for a set of parallel basic linear algebra subprograms*, Tech. report, 1995, LAPACK Working Note 100.

[8] ———, *The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines*, Scientific Programming **5** (1996), 173–184.

[9] E. D'Azevedo and J. Dongarra, *Packed storage extension for ScaLAPACK*, Tech. report, 1998, LAPACK Working Note 135.

[10] J. Demmel and J. Dongarra, *Reliable and scalable software for linear algebra computations on high end computers*, (2004), Proposal for Software and Tools for High-End Computing (ST-HEC).

[11] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Softw. **16** (1990), 1–17.

[12] J. Dongarra and R. Whaley, *A user's guide to the BLACS v1.1*, Tech. report, 1997, LAPACK Working Note 94.

[13] B. Gunter, W. Reiley, and R. van de Geijn, *Parallel out-of-core Cholesky and QR factorizations with POOCLAPACK*, IEEE Computer Society (2001), Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS).

[14] B. Gunter and R. van de Geijn, *Parallel out-of-core computation and updating of the QR factorization*, ACM Trans. Math. Softw. **31** (2005), no. 1, 60–78.

[15] F. Gustavson and S. Toledo, *The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computation.*, IEEE Computer Society (1996), In Proceedings of IOPADS'96.

[16] F. G. Gustavson, *New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms*, (June 20-23, 2004), 11–20, Proceedings of PARA'04, Workshop on state-of-the art in scientific computing.

[17] B. A. Hendrickson and D. E. Womble, *The torus-wrap mapping for dense matrix calculations on massively parallel computers*, SIAM J. Scientific Computing **15** (1994), no. 5, 1201–1226.

[18] T. Joffrain, E. Quintana-Ortì, and R. van de Geijn, *Rapid development of high-performance out-of-core solvers for electromagnetics*, (June 20-23, 2004), 414–423, Proceedings of PARA'04, Workshop on state-of-the art in scientific computing.

[19] G. W. Stewart, *Communication and matrix computations on large message passing systems*, Parallel Computing **16** (1990), 27–40.

[20] R. van de Geijn, *Using PLAPACK*, The MIT Press, 1997.