A PARALLEL DISTRIBUTED SOLVER FOR LARGE DENSE SYMMETRIC SYSTEMS: APPLICATIONS TO GEODESY AND ELECTROMAGNETISM PROBLEMS

Marc Baboulin
Luc Giraud
Serge Gratton

CERFACS, 31057 TOULOUSE CEDEX, FRANCE
BABOULIN@CERFACS.FR

Abstract

In this paper we describe the parallel distributed implementation of a linear solver for large-scale applications involving real symmetric positive definite or complex symmetric non-Hermitian dense systems. The advantage of this routine is that it performs a Cholesky factorization by requiring half the storage needed by the standard parallel libraries ScaLAPACK and PLAPACK. Our solver uses a J-variant Cholesky algorithm and a one-dimensional block-cyclic column data distribution but gives similar Gigaflops performance when applied to problems that can be solved on moderately parallel computers with up to 32 processors. Experiments and performance comparisons with ScaLAPACK and PLAPACK on our target applications are presented. These applications arise from the Earth's gravity field recovery and computational electromagnetics.

Key words: Scientific computing, parallel distributed algorithms, symmetric dense linear systems, packed storage format, Cholesky factorization, ScaLAPACK, PLAPACK

The International Journal of High Performance Computing Applications,
Volume 19, No. 4, Winter 2005, pp. 353–363
DOI: 10.1177/1094342005056134

1 Introduction

The solution of large dense linear systems appears in many engineering and scientific applications of computational sciences. This is, for instance, the case in geodesy where the calculation of the gravity field requires the solution of large linear least-squares problems that are often solved using the normal equations approach. Such dense linear systems also arise in electromagnetics applications when boundary elements are used. In these fields, the recent developments of the fast multipole method (FMM) enable us to perform dense matrix–vector products efficiently. This opens the possibility of using iterative Krylov solvers for solving these large systems. In particular, in electromagnetic computations, the FMMs are established methods providing reliable solution of linear systems up to a few tens of millions unknowns on parallel computers (Sylvand, 2002) for industrial calculations. In gravity field computations, these techniques are studied in research codes, but further investigations, mainly related to the accuracy of the solution, are needed to bring them into use in operational codes. For these latter calculations, dense factorizations are still the methods of choice and out-of-core implementations might be an alternative when the matrix does not fit into the memory (Dongarra and D'Azevedo, 1997; Gunter et al., 2000). With the advent of distributed memory parallel platforms, in-core codes are affordable and the widespread parallel libraries ScaLAPACK (Blackford et al., 1997) and PLAPACK (van de Geijn, 1997) are often selected to perform this linear algebra calculation. In particular, these libraries implement the Cholesky factorization for symmetric positive definite linear systems but they do not exploit the symmetry for the storage and the complete matrix should be allocated while the factorization only accesses half of it. This "waste" of memory cannot be afforded for operational industrial calculations since the memory of moderate size parallel computers is often the main bottleneck. This is the main reason why we develop a parallel distributed dense Cholesky factorization based on message passing interface (MPI; MPI Forum, 1994) that exploits the symmetry and stores only half of the matrix.

We choose a factorization algorithm and a data distribution that are different from the libraries ScaLAPACK and PLAPACK and these choices will be validated as correct for up to 32 processors. For higher processor counts, these choices should be reconsidered. Our algorithm implements:

- a J-variant block Cholesky factorization algorithm;
- a block-cyclic column data distribution where only the upper part of the matrix is stored, this storage being implemented in a row-wise format;

- message passing performed by non-blocking asynchronous MPI routines;
- level 3 BLAS (Dongarra et al., 1990) routines in order to account for the memory hierarchy on each processor.

This paper is organized as follows. In Section 2, we discuss the target applications that motivate the development of this solver. The purpose of Section 3 is to compare the features of the Cholesky factorization algorithm as it is implemented respectively in our solver and in ScaLAPACK or PLAPACK. For both implementations we successively describe the block algorithms in Section 3.1, the data distributions in Section 3.2, while the expected performance based on theoretical models is discussed in Section 3.3. The parallel implementation of our solver is detailed in Section 3.4. Then, in Section 4.1, we give some numerical results obtained on the target operational parallel distributed platforms in order to evaluate the parallel performance of the algorithm. These results are compared with those obtained with ScaLAPACK and PLAPACK. In Section 4.2 we present a natural extension to solve dense complex symmetric linear system arising in electromagnetics simulations. Finally, some concluding remarks are given in Section 5.

2 Target Applications

The first target application for our parallel Cholesky solver arises in the framework of the European Space Agency's Gravity field and steady-state Ocean Circulation Explorer (GOCE) mission dedicated to geodesy studies. This mission strives for a high-precision model of the Earth's static gravity field using measurements of the GPS constellation (Pail and Plank, 2002). The parameters are estimated using data recovered from daily GPS observations via a least-squares approach $\min_{x \in \mathbb{R}^n} \|Bx - c\|_2$ where $c \in \mathbb{R}^m$ is the observations vector and $B \in \mathbb{R}^{m \times n}$ is a full column rank matrix. Such problems can be solved either by using unitary transformations or by forming and then solving the normal equations:

$$B^T Bx = B^T c.$$

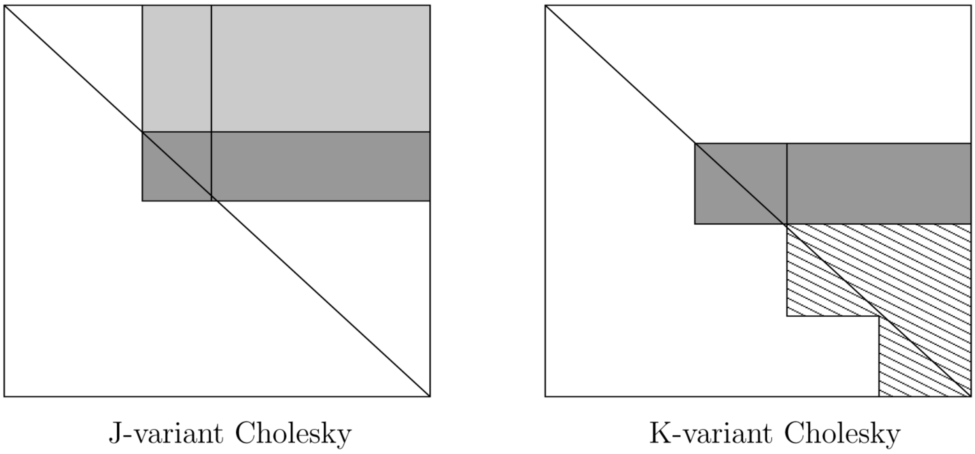
In this application, m is about 10^6 and n is slightly less than 10^5. It is known from Björck (1996) or Golub and van Loan (1996) that, if the condition number of B is large and the residual is small, the normal equations may be less accurate than the QR factorization. However the normal equations method is often favored by the users in geodesy because, when $m \gg n$, it has half the computational cost of a QR factorization (mn^2 instead of $2mn^2$). Indeed, for geodesy problems, the normal equations yield a solution that is accurate enough for the users because the estimated condition number of B is not too large. The $n \times n$ matrix $B^T B$ can be decomposed using a Cholesky factorization as $B^T B = U^T U$ where U is an upper triangular matrix. Then the normal equations become $U^T Ux = B^T c$, and the unknown vector x can be computed via forward and backward substitutions. The cost in arithmetic operations can be split as follows.

1. In the construction of the symmetric matrix $B^T B$, we only compute and store the upper triangular part of $B^T B$. Hence the cost in operations will be $\mathcal{O}(mn^2)$.
2. The Cholesky factorization algorithm involves $n^3/3$ operations.
3. The final step consists in solving two triangular systems. Hence it involves $2n^2$ operations.

We point out that steps 1 and 2 are the most time-consuming tasks and need an efficient parallel distributed implementation. The construction and storage of $B^T B$ has been implemented using MPI and the obtained performance is close to the peak performance of the computers on a matrix–matrix multiply (Parallel Algorithms Project, 2002, Section 6.3). The triangular solve, which is less critical in terms of arithmetic operations, has been implemented in a distributed manner but its performance study is beyond the scope of this paper.

In recent years, there has been a significant amount of work on the simulation of electromagnetic wave propagation phenomena, addressing various topics ranging from radar cross-section to electromagnetic compatibility, to absorbing materials, and antenna design. To address these problems, the Maxwell equations are often solved in the frequency domain. The discretization by the boundary element method (BEM) results in linear systems with dense complex symmetric matrices. With the advent of parallel processing, solving these equations via direct methods has become viable for large problems and the typical problem size in the electromagnetics industry is on the increase. Nowadays, the usual problem size is a few tens of thousands. We may notice that there is no efficient parallel solver based on compact storage for symmetric dense complex matrices suited for the modeling of electromagnetic scattering and running on moderate processor configurations (less than 32 processors).

The numerical simulations we are interested in are performed in a daily production mode at GRGS/Centre National d'Etudes Spatiales (CNES) for the geodesy application and by the Electromagnetism and Control Project at CERFACS for the electromagnetism application. For these projects, the target parallel platforms are moderately parallel computers with up to 32 processors and less than 2Gbytes memory per processor.



J-variant Cholesky

K-variant Cholesky

Fig. 1 Memory access patterns for two variants of the Cholesky factorization.

3 Parallel Implementation

3.1 BLOCK CHOLESKY FACTORIZATION

There are two main variants used for the block Cholesky factorization. The variant referred to as J-variant in Anderson and Dongarra (1990) and Ortega and Romine (1988) or left-looking algorithm in Gunter et al. (2000) computes one block row of U at a time, using previously computed lines that are located at the top of the block row being updated. The second variant is referred to as K-variant in Anderson and Dongarra (1990) or right-looking algorithm in Gunter et al. (2000). This variant performs the Cholesky decomposition $A = U^T U$ by computing a block row at each step and using it to update the trailing submatrix. The K-variant algorithm is implemented in the ScaLAPACK routine PDPOTRF (Choi et al., 1996) and the PLAPACK routine PLA_Chol (Alpatov et al., 1997). The J-variant and the K-variant are described in Figure 1, where the shaded part refers to matrix elements being accessed, the dark shaded part represents the block row being computed (current row), and the hatched part corresponds to the data being updated. In order to describe the difference between these two variants, let us consider the following symmetric block matrix

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{12}^T & \mathbf{A_{22}} & A_{23} \\ A_{13}^T & A_{23}^T & A_{33} \end{pmatrix},$$

where A_{22} is the diagonal block to be factored, assuming that the first block row has been computed and that we want to obtain the second block row.

The J-variant algorithm allows to advance the factorization as described below

$$\begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & A_{22} & A_{23} \\ & & A_{33} \end{pmatrix} \rightarrow \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & A_{33} \end{pmatrix},$$

where the second block row is computed with the following steps:

1. $U_{22} \leftarrow Chol(A_{22} - U_{12}^T U_{12})$,
2. $U_{23} \leftarrow U_{22}^{-T}(A_{23} - U_{12}^T U_{13})$.

The block A_{22} is first updated by using a matrix–matrix product and then factored. Each block belonging to the rest of the block row is updated by a matrix–matrix multiply followed by a triangular solve with multiple right-hand sides.

According to the K-variant algorithm, we advance the factorization as follows

$$\begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & \tilde{A}_{22} & \tilde{A}_{23} \\ & & \tilde{A}_{33} \end{pmatrix} \rightarrow \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & \tilde{A}_{33} \end{pmatrix},$$

Table 1
Breakdown of floating-point operations for block Cholesky algorithm.

Routine	$n = 500 - b = 64$	$n = 40{,}000 - b = 128$
DGEMM	84.6	99.5
DTRSM	14.1	0.5
DPOTRF	1.3	0.001

where the second block row is computed with the following steps:

1. $U_{22} \leftarrow Chol(\tilde{A}_{22})$,
2. $U_{23} \leftarrow U_{22}^{-T}\tilde{A}_{23}$,
3. $\tilde{A}_{33} \leftarrow \tilde{A}_{33} - U_{23}^{T}U_{23}$.

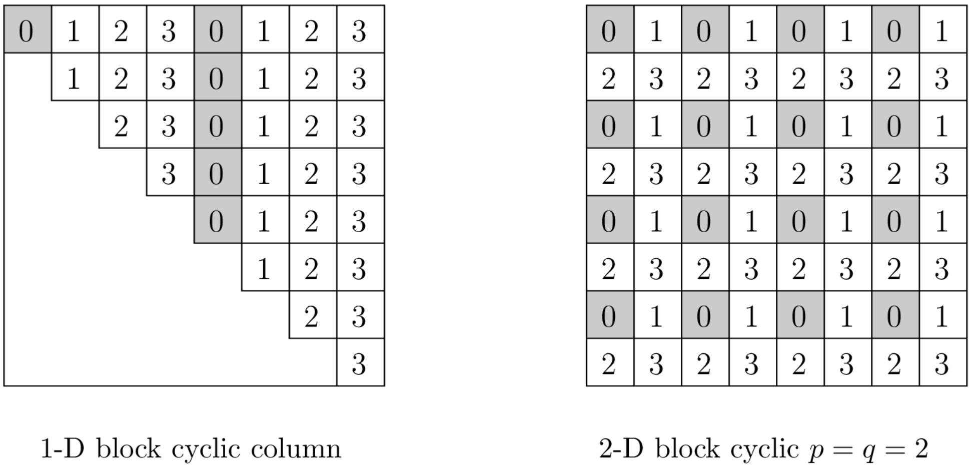
The block $\tilde{A}_{22}$ is first factored then U_{23} is computed by using a triangular solve with multiple right-hand sides. Finally, the trailing submatrix $\tilde{A}_{33}$ is updated.

We notice that in both algorithms, the computational work is contained in three routines: the matrix–matrix multiply, the triangular solve with multiple right-hand sides, and the Cholesky factorization. These numerical kernels can be respectively implemented using the Level 3 BLAS and LAPACK (Anderson et al., 1999) routines: DGEMM, DTRSM and DPOTRF. As explained in Anderson and Dongarra (1990), similar performance can be expected from both algorithms when the three dominant routines are implemented equally well. Table 1 shows for two sample matrices that the operations involved in the DGEMM routine represent the major part of the operations for matrices involved in our applications and that this percentage increases with the size of the matrix (b corresponds to the block size chosen in our algorithm).

We point out that our J-variant Cholesky algorithm that is sometimes called left-looking has to be distinguished from the algorithm referred to as left-looking LU in Dongarra et al. (1998), which performs more DTRSM triangular solves than the two other LU variants considered in that book.

3.2 DATA DISTRIBUTION

The data layout used in our solver is the one-dimensional block-cyclic column distribution. According to this layout, we choose a block size b and we divide the columns into groups of size b. Then we distribute these groups among processors in a cyclic manner column by column. Figure 2 shows an example of such a distribution for an 8 × 8 block matrix when we have four processors numbered 0, 1, 2, and 3. In this figure, each block is labeled with the number of the processor that stores it. The K-variant Cholesky algorithm implemented in ScaLAPACK or PLAPACK is based on a two-dimensional block-cyclic data distribution. In this type of distribution, the processors are arranged in a $p \times q$ rectangular array of processors. According to this choice of grid, the matrix blocks are assigned in a cyclic manner to different processors. Figure 2 shows an example of such a distribution for an 8 × 8 block matrix if we have four processors with $p = 2$ and $q = 2$. Note that a one-dimensional column distribution is a particular choice of two-dimensional distribution if $q = 1$. We refer to Blackford et al. (1997) for a more detailed description of these two types of data distribution.



0	1	2	3	0	1	2	3
	1	2	3	0	1	2	3
		2	3	0	1	2	3
			3	0	1	2	3
				0	1	2	3
					1	2	3
						2	3
							3

1-D block cyclic column

0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3

2-D block cyclic $p = q = 2$

Fig. 2 Block-cyclic data distributions.

3.3 PERFORMANCE PREDICTION

As we have chosen an algorithm and a data distribution that are different from the standard libraries, we may see the consequence of this choice on the expected factorization times. We model the theoretical performance by evaluating the elapsed computational time and assuming that the communication is perfectly overlapped by the computation. The model parameters are the problem size n, the block size b, the number of processors $p \times q$, and the peak performance γ of a matrix–matrix product DGEMM of size b on the target machine. The parameter b is usually tuned in order to obtain the best value for γ. In the following, we will consider a peak performance γ ~3.3 Gflops obtained with a block size $b = 128$, this choice being consistent with the performance of current parallel platforms. We denote n_b the number of columns of the block matrix.

As shown in Table 1, the major part of computation for both algorithms consists in performing the matrix–matrix multiply DGEMM. Hence, if s is the total number of arithmetic operations involved the Cholesky factorization, then the elapsed factorization time can be accurately approximated by s/γ.

For the J-variant algorithm using a one-dimensional block column distribution, we have

$$s = n_b \frac{b^3}{3} + \sum_{i=1}^{n_b} n_c((i-1)2b^3 + b^3),$$

where n_c is the maximum number of block columns treated per processor at step i.

For the K-variant algorithm and a two-dimensional block cyclic distribution for a $p \times q$ processors grid, we have

$$s = n_b \frac{b^3}{3} + \sum_{i=1}^{n_b} (n_s b^3 + n_u 2b^3) \quad ,$$

where n_s and n_u are the maximum number of blocks treated per processor involved respectively in the triangular solve and in the update of the trailing submatrix. We noticed experimentally that the choice of grid corresponding to $1/2 \le p/q \le 1$ ensures the best time (or close to).

The efficiency of the algorithm can be evaluated by measuring how the performance degrades as the number of processors p increases while each processor uses the same amount of memory. This measures what we define as isomemory scalability. This type of performance measurement is suitable for our target applications since we aim to use most of the memory available for each processor of the parallel machine. Using the above formula of s, we obtain a theoretical factorization time and the resulting plots evaluate what we define as theoretical isomemory scalability of the algorithm. The problem size n_p is such that each processor uses about the same amount of memory σ. In particular, σ is the memory size required to solve a problem of order n_1 on one processor. We first choose a value of n_1 that is compatible with the minimal memory capacity of current parallel machines. We take $n_1 = 10{,}000$, which corresponds to a $\sigma = 800$ Mbytes for solvers storing the whole matrix and 400 Mbytes memory storage for our symmetric solver. Then we measure the factorization time for problem of size n_p that has to be solved on p processors. The invariance of storage per processor can be written

$$\frac{n_1(n_1+1)}{2} = \frac{n_p(n_p+1)}{2p}$$

and then approximated by $n_p \approx n_1\sqrt{p}$. In Figure 3(a), we obtain the theoretical factorization times when the number of processors increases from 1 to 256 and the corresponding matrix size increases from 10,000 to 160,000. We see in Figure 3(a) that a discrepancy between the two curves occurs for 32 processors (corresponding to a problem size of 56,569), due to difference of load-balance resulting from the choice of data distribution. When more than 64 processors are used, figures given by the model confirm the well-known result that a two-dimensional block cyclic distribution is better than a one-dimensional block cyclic column distribution. However, as we can see in Figure 3(b), the scalability properties of both layouts are very similar for processor counts lower than 32.

3.4 PARALLEL IMPLEMENTATION OF THE J-VARIANT CHOLESKY ALGORITHM

3.4.1 Choice of a Data Structure As explained in Section 1, one of the main objectives of our implementation is to exploit the symmetry of the matrix by storing only about half of it. This implies that the memory of each processor will only contain the blocks assigned to this processor that belong to the upper triangular part of the symmetric matrix A. An appropriate choice for the structure containing these data must notably comply with the memory hierarchy constraints (level 1, 2 or 3 cache or TLB). More precisely, the fact that blocks are stored row-wise or column-wise in a local array might have a significant influence on performance in terms of Mflops. This will be illustrated in this paragraph by some experiments performed on IBM pSeries 690 (using the essl scientific library).

We saw in Section 3.1 that the calls to the BLAS 3 routine DGEMM and among them, the outer product between two different block columns represent the major part of the operations performed in a Cholesky factorization. If we denote (A_{ij}) the block triangular array corresponding to A, then this outer product consists in computing the operation on blocks $A_{ij} \leftarrow A_{ij} - \Sigma_{k=1}^{i-1} A_{ki}^{\mathrm{T}} A_{kj}$ where i is the current row and $A_{ij}(j > i)$ is the element of column j to

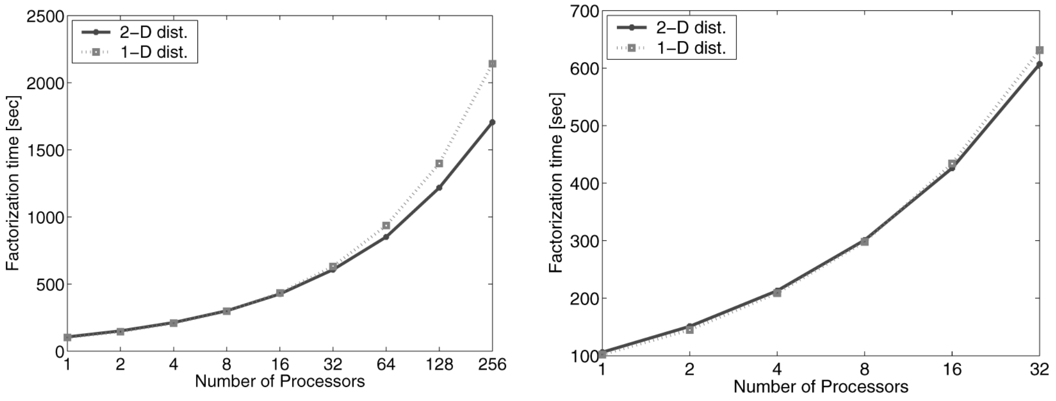


Fig. 3 Theoretical isomemory scalability (peak = 3.3 Gflops): (a) number of processors varying from 1 to 256; (b) number of processors varying from 1 to 32.

update. If $n_{\max}$ is the maximum number of blocks owned by a given processor and b is the block size defined in Section 3.3 then data blocks can be arranged in a Fortran array according to either a block-row storage or a block-column storage by using respectively an array of leading dimension b and $n_{\max} \times b$ columns or an array of leading dimension $n_{\max} \times b$ and b columns. As an example, the matrix sample given in Figure 2 leads for processor 0 to a local array that can be represented using a block-row storage by

A_{11}	A_{15}	A_{25}	A_{35}	A_{45}	A_{55}

whereas it will be represented using a block-column storage by

A_{11}
A_{15}
A_{25}
A_{35}
A_{45}
A_{55}

.

In order to evaluate the memory effect generated by each data structure, we plot in Figure 4 the performance obtained on the IBM pSeries 690 for the instruction $A_{ij} \leftarrow A_{ij} - \Sigma_{k=1}^{n_{\max}} A_{ki}^{T} A_{kj}$ where the block columns $(A_{ki})_{k=1,n_{\max}}$ and $(A_{kj})_{k=1,n_{\max}}$ are stored either row-wise or column-wise in a Fortran array. In our experiments we set $b = 128$ and $n_{\max} = 50$ but we obtained similar curves for $n_{\max} > 50$. Figure 4(a) is related to a block-row data storage and the number of columns varies from $n_{\max} \times b$ to $3 \times n_{\max} \times b$. Figure 4(b) is related to a block-column data storage and the leading dimension varies from $n_{\max} \times b$ to $3 \times n_{\max} \times b$. Since the IBM pSeries 690 has a L1 cache two-way associative of 32 KB, we obtain the worst performance when we successively access to data that are distant from a multiple of 16 KB (because they are stored in the same set of the cache memory). In a Fortran array, the distance between two consecutive data of the same line is the leading dimension of the array. Hence, cache misses are expected when we access to two consecutive double-precision real in a line of an array whose leading dimension is a multiple of 2048 ($2048 = 16K/8$). We notice in Figure 4(b) that these spikes appear exactly with the same period when performing the outer product instruction and varying the leading dimension. We also observe the secondary spikes appearing at fraction of 2048 (Goedecker and Hoisie, 2001). On the contrary, in a block-row structure, the distance between two consecutive entries in a line is b and performance obtained by computing the outer product is more stable with much less cache misses. Furthermore, if we use a block-row storage, the blocks belonging to the same block column will be contiguous in memory and then will map better to the highest levels of cache. This data contiguity is also an advantage when performing MPI operations. Taking these results into consideration, we chose the row-wise way of storing data blocks in the local memory of each processor.

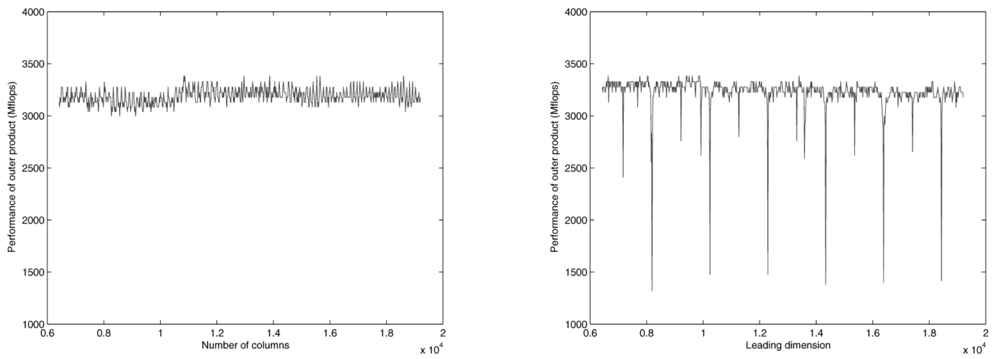


Fig. 4 Cache misses on IBM pSeries 690 for variable size arrays: (a) block-row data storage; (b) block-column data storage.

3.4.2 Parallel Algorithm To begin with, half of the symmetric matrix A is loaded into memory distributed in a block upper triangular array $(A_{ij})_{j>1}$ according to the layout described in Section 3.2 and stored into memory according to the data storage described in Section 3.4.1. We recall that n_b is the number of columns of the block matrix (A_{ij}) and we denote by p the *id* of the current processor, $(proc(i))_{i=1,n_b}$ the array containing the processor *id* for each block column i, and *nprocs* the total number of processors involved in the parallel calculation. Then the following parallel block algorithm is executed simultaneously by all processors:

Algorithm 1. Parallel block Cholesky

```
for $i = 1 : n_b$
    if $p = proc(i)$ then
        1.$proc(i)$ sends $(A_{ki}, k \leq i-1)$ to other processors (Isend1)
        2.$A_{ii} \leftarrow A_{ii} - \sum_{k=1}^{i-1} A_{ki}^{\mathrm{T}} A_{ki}$
        3.$A_{ii} \leftarrow Chol(A_{ii})$
        4.$proc(i)$ sends $A_{ii}$ to other processors (Isend2)
        5.Completion of Isend1 (Wait1)
        6.for each block column $j > i$ assigned to proc $p$:
            $A_{ij} \leftarrow A_{ij} - \sum_{k=1}^{i-1} A_{ki}^{\mathrm{T}} A_{kj}$
        7.Completion of Isend2 (Wait2)
        8.for each block column $j$ assigned to proc $p$
            $A_{ij} \leftarrow A_{ii}^{-T} A_{ij}$
    else
        9.receive $(A_{ki}, k \leq i - 1)$ from $proc(i)$(Irecv1)
        10.completion of Irecv1 (Wait1)
        11.receive $A_{ii}$ from $proc(i)$ (Irecv2)
        12.for each block column $j > i$, $A_{ij} \leftarrow A_{ij} - \sum_{k=1}^{i-1} A_{ki}^{\mathrm{T}} A_{kj}$
        13.completion of Irecv2 (Wait2)
        14.$A_{ij} \leftarrow A_{ii}^{-T} A_{ij}$
    endif
end (i-loop)
```

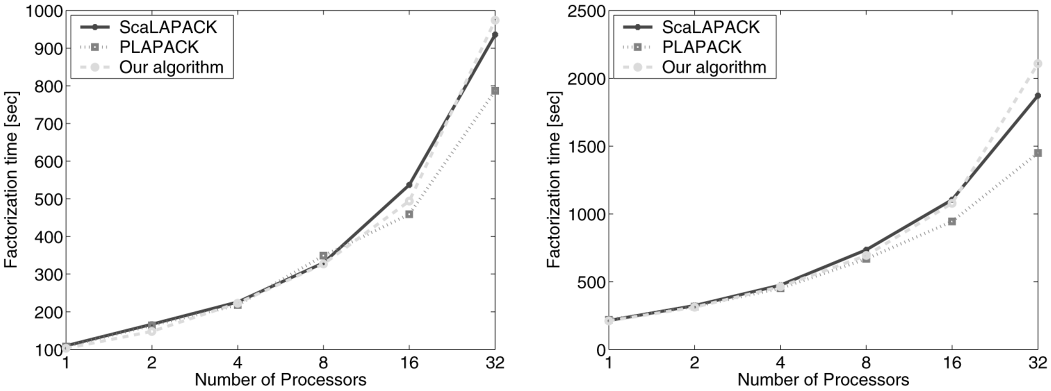


Fig. 5 Isomemory scalability of the Cholesky factorization: (a) IBM pSeries 690; (b) HP-COMPAQ Alpha Server.

As can be seen in instructions 1 and 4 of Algorithm 1, data are sent as soon as they are available by a non-blocking instruction MPI_Isend. This allows the overlapping of the communications by the computations. The update of the rest of the block row can be performed by all the processors while the communications are still ongoing. When these data are necessary for further use by the sending processor at steps 6 and 8, the communication must be completed by using MPI_Wait instruction (respectively in 5 and 7). In the same manner, messages are received as soon as possible using the non-blocking instruction MPI_Irecv in 9 and 11 and then completed by MPI_Wait just before the received data are used in the computations involved in instructions 12 and 14. These non-blocking message exchanges save about 10% of the factorization times on a matrix of dimension 24,000 on 16 processors of the IBM platform, compared with an implementation using blocking instructions MPI_Send and MPI_Recv.

4 Experiments

4.1 PERFORMANCE COMPARISONS

Experiments were conducted on the following platforms:

1. one node of an IBM pSeries 690 (32 processors Power-4/1.3 GHz and 64 Gbytes memory per node),
2. a HP-COMPAQ Alpha Server (10 SMP nodes with four processors EV68 1 GHz and 4 Gbytes memory per node) installed at CERFACS.

Computations with ScaLAPACK were performed using the pessl library on the IBM pSeries 690 and the COMPAQ ScaLAPACK V1.0.0 library on the HP-COMPAQ Alpha Server. On both machines, we used PLAPACK release 3.0.

We compare performance obtained by our J-variant Cholesky algorithm and that of the Cholesky factorization routines PDPOTRF and PLA_Chol. In Figure 5, we plot the isomemory scalability defined in Section 3.3 (i.e. constant memory allocation per processor). On this curve, the problem size varies from 10,000 (for one processor and corresponding to a memory storage per processor of about 800 Mbytes compatible with our target platforms) to 56,569 (for 32 processors). Figure 5 shows that, for up to 16 processors, our solver provides factorization times that are close to those obtained by ScaLAPACK and PLAPACK. For 32 processors, performance of the libraries using a two-dimensional block cyclic distribution are, as expected, better and this is consistent with the theoretical model described in Section 3.3.

Table 2 measures the scalability in floating-point operations of both algorithms, that we name isoflop scalability. It shows how the performance per processor (in Gflops) of the algorithms behaves when the number of processors increases and while each processor performs the same number of floating-point operations. Measuring a performance per processor enables us to compare easily with the peak performance of the machine. In the ideal case, i.e. when the communication is perfectly overlapped by the computation, we expect this performance being constant. This implies that, with the same notations as in Section 3.3, $n_p^3 = pn_1^3$, i.e. $n_p = n_1\sqrt[3]{p}$. In our experi-

Table 2
Isoflop scalability for the Cholesky factorization (Gflops).

		IBM pSeries 690			HP-COMPAQ Alpha Server		
nb procs	size	our solver	PDPOTRF	PLA_Chol	our solver	PDPOTRF	PLA_Chol
1	9449	3.3	2.9	2.9	1.5	1.5	1.5
2	11906	3.2	2.9	2.9	1.5	1.4	1.5
4	15000	2.9	2.9	2.9	1.4	1.3	1.4
8	18899	2.7	2.7	2.7	1.3	1.2	1.3
16	23811	2.5	2.5	2.5	1.1	1.1	1.2
32	30000	2	2.2	2.2	0.7	0.8	1

ments, we chose $n_1 = 9449$ and this corresponds to a memory storage per processor of about 700 Mbytes for ScaLAPACK and PLAPACK (350 Mbytes for our solver) and a fixed number of about 2.8×10^{11} floating-point operations per processor. It can be seen in Table 2 that the performance of the three routines is similar up to 16 processors and that, for 32 processors, our solver is slightly less efficient than ScaLAPACK and PLAPACK. For all routines, we notice that performance decreases significantly with the number of processors, due to the cost of the communications.

Now if we compare factorization times obtained on each machine for a given problem size, we notice that the IBM pSeries 690 is about twice as fast as the HP-COMPAQ Alpha, as it was expected since the Power 4 and the EV68 processors have a peak computation rate of respectively 5.2 Gflops and 2 Gflops.

4.2 APPLICATION TO ELECTROMAGNETISM

Contrary to the previous section, the problems we consider in this section use complex arithmetic. Large, dense and symmetric linear systems in complex arithmetic can be found in the area of electromagnetism. They result from boundary-element formulations for the solution of the three-dimensional Maxwell's equations. Many of these applications are still exploiting direct methods like LU (or sometimes LDL^T) factorization (Alléon et al., 1997; Bendali and Fares, 1999).

We propose to extend the factorization technique that we developed in real arithmetic and thereby to save 50% of the storage compared with standard software considered in Alléon et al. (1997) and Bendali and Fares (1999). Moreover, the computational cost of a U^TU factorization is half that of an LU factorization ($n^3/3$ instead of $2n^3/3$ and with complex operations). Note that a LDL^T factorization with Bunch–Kauffman diagonal pivoting could have also be used but, as shown in Béreux (2003), provided there is no need for pivoting, a U^TU factorization is as accurate and faster. Since we do not perform any pivoting in our software, similarly to Li and Demmel (2003), we implemented an iterative refinement functionality (Higham, 2002, p. 232) to improve the accuracy and stability of our solver.

Our solver enables us to switch easily the data type and to replace the LAPACK routine DPOTRF (that factors diagonal blocks) by a routine performing unblocked complex U^TU factorization. Even if this routine is less efficient than the corresponding LAPACK routine ZPOTRF (which cannot be used here because it provides a U^HU factorization), this has no significant effect on the factorization time since the operations involved in factoring the diagonal blocks represent a very small part of total operations. Experiments were conducted on eight processors of the HP-COMPAQ Alpha Server from CERFACS. The selected geometry is an aircraft with a mesh size of 18,264 represented in Figure 6. The observed elapsed times were 762 s for our U^TU factorization and 1310 s for the ScaLAPACK LU factorization used in the code CESC

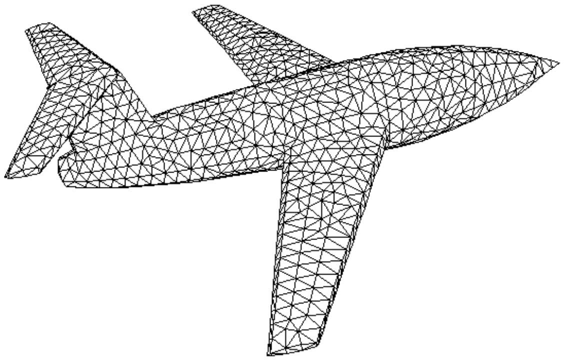


Fig. 6 Mesh of aircraft test example for electromagnetism calculation.

Table 3
Performance ratio between LU and U^TU factorization (HP-COMPAQ Alpha).

nb procs	size	LU ScaLAPACK/Our solver
1	6600	1.95
2	9334	1.91
4	13200	1.81
8	18668	1.72
16	26400	1.62

(Bendali and Fares, 1999) from the CERFACS EMC Project. We computed the corresponding scaled residual

$$\frac{\|A\tilde{x} - 1\|}{\|b\|}$$

and we obtained 2.75 × 10^{-14} (for 64-bit double precision calculation). Since the physicists require a scaled residual of order 10^{-6}, the iterative refinement was not activated.

Even if our algorithm involves half the number of operations needed for an LU factorization, we do not obtain exactly a factorization time that is half that of an LU. This relatively poor performance of a Cholesky factorization compared to an LU factorization can be explained by the fact that the ScaLAPACK LU factorization does not involve a number of message exchanges that is twice as much as our Cholesky factorization. In fact, at each step of the algorithm, the Cholesky factorization performed either by ScaLAPACK or our symmetric solver involves two communications as opposed to three for the ScaLAPACK LU. To illustrate this, we give in Table 3 the ratio between factorization times obtained by ScaLAPACK LU and our solver for other sizes of matrices arising from electromagnetism problems. We notice that the performance ratio is decreasing when the number of processors grows and thus when the impact of the communication on the global performance is more sensible. We obtain similar ratios when we compare performance of ScaLAPACK LU and that of ScaLAPACK Cholesky. Similar behavior has been observed in Choi et al. (1996).

5 Conclusion

We have described an implementation of a parallel symmetric solver involving algorithmic and distribution choices that are different from that of ScaLAPACK and PLAPACK. The behavior that one can expect from the theoretical model has been confirmed by experiments up to 32 processors. The obtained solution complies with the experimental operational constraints and we can solve problems of size corresponding to the target applications. For instance, problems of size 100,000 were solved on an IBM pSeries 690 in 1 h 20 min using 32 processors. We also solved problems of size 100,000 on HP-COMPAQ Alpha in 2 h 40 min using 32 processors on nodes that have 2 Gbytes memory per processor. Our parallel distributed symmetric solver is efficient to solve geodesy least-squares problems by a normal equations method and also to solve complex linear systems encountered in electromagnetism. From a numerical point of view, some additional work can be developed. In this respect, we are planning to add to our implementation a condition number estimate to fully assess the normal equations approach for the geodesy application. Furthermore, other approaches to design parallel linear solvers might consist in using high-level parallel routines in PLAPACK or PBLAS/ScaLAPACK. They deserve to be investigated and will be the topic of further studies.

ACKNOWLEDGMENTS

We would like to thank the Centre Informatique National de l'Enseignement Superieur (CINES) for allowing us to perform experimentation on its parallel computers. We also express gratitude to Iain Duff (Rutherford Appleton Laboratory and CERFACS) for fruitful discussions, Robert van de Geijn (University of Texas at Austin) for his help to use PLAPACK and to Messrs Georges Balmino (CNES) and M'Barek Fares (CERFACS EMC project) for their help with geodesy and electromagnetic applications, respectively. The authors wish to thank the referees for valuable comments on the manuscript.

AUTHOR BIOGRAPHIES

Marc Baboulin received an engineering degree in computer science and applied mathematics from ENSEEIHT, France. He obtained a Master of Science in signal processing in July 2003, subsequently followed by a Ph.D. in Computer Science at CERFACS. His research focuses on parallel computing and numerical linear algebra. He works more specifically on direct solvers for large dense linear systems or least-squares problems arising in engineering applications and their efficient implementation on parallel computers.

Luc Giraud received an engineering degree in applied mathematics and computer science and then a Ph.D. in computer science from the Institut National Polytechnique de Toulouse. Currently he is a senior researcher and the project leader deputy of the Parallel Algorithms Project at CERFACS. His research interests include linear algebra, iterative methods, domain decomposition and high performance computing as well as their implementations in industrial applications.

Serge Gratton received an engineering degree in applied mathematics and computer science and then a Ph.D. in computer science from the Institut National Polytechnique de Toulouse. Currently he is a senior researcher in the Parallel Algorithms Project at CERFACS. His research interests include linear algebra, methods for least-squares and high performance computing.

Reference

Alléon, G., Amram, S., Durante, N., Homsi, P., Pogarieloff, D., and Farhat, C. 1997. Massively parallel processing boosts the solution of industrial electromagnetic problems: high-performance out-of-core solution of complex dense systems. *SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 14–17.

Alpatov, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., Overfelt, J., van de Geijn, R., and Wu, Y. 1997. PLAPACK: Parallel Linear Algebra Libraries Design Overview. *Proceedings of Supercomputing 97*, San Jose, CA, November 15–21.

Anderson, E. and Dongarra, J. 1990. Evaluating block algorithm variants in LAPACK, Technical Report, LAPACK Working Note 19.

Anderson, E. et al. 1999. *LAPACK User's Guide*, 3rd edition, SIAM, Philadelphia, PA.

Bendali, A. and Fares, M'B. 1999. CERFACS electromagnetics solver code, Technical Report TR/EMC/99/52, CERFACS, Toulouse, France.

Béreux, N. 2003. A Cholesky algorithm for some complex symmetric systems, Technical Report 515, Ecole Polytechnique, Centre de Mathématiques appliquées, Palaiseau, France.

Börck, Å. 1996. *Numerical Methods for Least-Squares Problems*, SIAM, Philadelphia, PA.

Blackford, L. et al. 1997. *ScaLAPACK User's Guide*, SIAM, Philadelphia, PA.

Choi, J., Dongarra, J., Ostrouchov, L., Petitet, A., Walker, D., and Whaley, R. 1996. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming* 5:173–184.

Dongarra, J. and D'Azevedo, E. 1997. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines, Technical Report, LAPACK Working Note 118.

Dongarra, J., Du Croz, J., Duff, I., and Hammarling, S. 1990. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software* 16:1–17.

Dongarra, J., Duff, I., Sorensen, D., and van der Vorst, H. 1998. *Numerical Linear Algebra for High Performance Computers*, SIAM, Philadelphia, PA.

Goedecker, S. and Hoisie, A. 2001. *Performance Optimization of Numerically Intensive Codes*, SIAM, Philadelphia, PA.

Golub, G. and van Loan, C. 1996. *Matrix Computations*, 3rd edition, Johns Hopkins University Press, Baltimore, MD.

Gunter, B. C., Reiley, W. C., and van de Geijn, R. A. 2000. Implementation of out-of-core Cholesky and QR factorizations with POOCLAPACK, Technical Report, PLAPACK Working Note 12.

Higham, N. 2002. *Accuracy and Stability of Numerical Algorithms*, 2nd edition, SIAM, Philadelphia, PA.

Li, X. and Demmel, J. 2003. SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software* 29:110–140.

MPI Forum. 1994. MPI: a message passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing* 8(3–4).

Ortega, J. M. and Romine, C. H. 1988. The *ijk* forms of factorization methods II. Parallel systems. *Parallel Computing* 7:149–162.

Pail, R. and Plank, G. 2002. Assessment of three numerical solution strategies for gravity field recovery from GOCE satellite gravity gradiometry implemented on a parallel platform. *Journal of Geodesy* 76:462–474.

Parallel Algorithms Project. 2002. Scientific Report for 2002, Technical Report TR/PA/02/124, CERFACS, Toulouse, France, http://www.cerfacs.fr/algor/reports.

Sylvand, G. 2002. La méthode multipôle rapide en électromagnétisme: performances, parallélisation, applications. Ph.D. Thesis, Ecole Nationale des Ponts et Chaussées.

van de Geijn, R. 1997. *Using PLAPACK*, MIT Press, Cambridge, MA.