# Advances in Sparse Hypermatrix Cholesky Factorization

José R. Herrero, Juan J. Navarro*

**Abstract**

We present our work on the sparse Cholesky factorization using a hypermatrix data structure. First, we provide some background on the sparse Cholesky factorization and explain the hypermatrix data structure. Next, we present the matrix test suite used. Afterwards, we present the techniques we have developed in pursuit of performance improvements for the sparse hypermatrix Cholesky factorization of a symmetric positive definite matrix into a lower triangular factor $L$.

*Keywords: Sparse Cholesky factorization, hypermatrix structure, small matrix library, windows in data submatrices, intra-block amalgamation*

## 1 Background

Sparse Cholesky factorization is heavily used in several application domains, including finite-element and linear programming algorithms. It forms a substantial proportion of the overall computation time incurred by those applications. Consequently, there has been great interest in improving its performance [1, 2, 3]. Methods have moved from column-oriented approaches into panel or block-oriented approaches. The former use level 1 BLAS while the latter have level 3 BLAS as computational kernels [3]. Operations are thus performed on blocks (submatrices).

Columns having similar structure are taken as a group. These column groups are called *supernodes* [4]. Some supernodes may be too large to fit in cache and it is advisable to split them into *panels* [2, 3]. In other cases, supernodes can be too small to yield good performance. This is the case of supernodes with just a few columns. Level 1 BLAS routines are used in this case and the performance obtained is therefore poor. This problem can be reduced by *amalgamating* several supernodes into a single larger one [5]. Although, some null elements are then both stored and used for computation, the resulting use of level 3 BLAS routines often leads to some performance improvement.

We address the optimization of the sparse Cholesky factorization of large matrices. For this purpose, we use a *Hypermatrix* [6] block data structure.

## 2 Hypermatrix data structure

Our application uses a data structure based on a hypermatrix (HM) scheme [6, 7], in which a matrix is partitioned recursively into blocks of different sizes. Commercial packages such as NASTRAN or PERMAS have used the hypermatrix structure for solving very large systems of equations [8]. They can solve very large systems out-of-core and can work in parallel. This approach is also related to a variety of recursive/nonlinear data layouts which have been explored elsewhere for both regular [9, 10, 11, 12] and irregular [13] applications.

The HM structure consists of $N$ levels of submatrices, where N is an arbitrary number. In order to have a simple HM data structure which is easy to traverse we have chosen to have blocks at each level which are multiples of the lower levels. The top *N-1* levels hold pointer matrices which point to the next lower level submatrices. Only the last (bottom) level holds data matrices. Data matrices are stored as dense matrices and operated on as such. Hypermatrices can be seen as a generalization of quadtrees. The latter partition each matrix precisely into four submatrices [14].

Null pointers in pointer matrices indicate that the corresponding submatrix does not have any non-zero elements and is therefore unnecessary. This is useful when matrices are sparse. Figure 1 shows a sparse matrix and a simple example of corresponding hypermatrix with 2 levels of pointers.

The main potential advantage of a HM structure is the ease of use of multilevel blocks to adapt the computation to the underlying memory hierarchy. The

*Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona, (Spain). Email:{josepr,juanjo}@ac.upc.edu. This work was supported by the Ministerio de Educación y Ciencia of Spain (TIN2004-07739-C02-01)

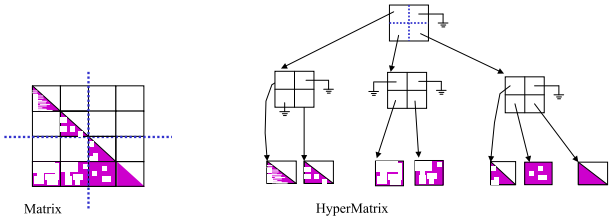


Figure 1: A sparse matrix and a corresponding hypermatrix.

operation on dense submatrices can take advantage of the BLAS3 routines. At the same time however, HM suffers from the disadvantage that zeros can be stored within data submatrices and used in computation. This is due to the fact that we do not descend down to the element level. Instead we use data submatrices of arbitrary size, considering them as dense blocks. Consequently, such data submatrices can store some zero elements. This can produce substantial overhead on sparse Cholesky. In many figures shown in this document we will present the *Effective Mflops* obtained. They refer to the number of useful floating point operations (*#flops*) performed per second. Although the time includes the operations performed on zeros, this metrics excludes nonproductive operations on zeros performed by the HM Cholesky algorithm when data submatrices contain such zeros.

$$Effective\ Mflops = \frac{\#flops(excluding\ operations\ on\ zeros) \cdot 10^{-6}}{Time\ (including\ operations\ on\ zeros)}$$

Block sizes can be chosen either statically (fixed) or dynamically. In the former case, the matrix partition does not take into account the structure of the sparse matrix. In the latter case, information from the *elimination tree* [15] is used. In this work we partition the hypermatrix statically.

## 3 Matrix characteristics

We have used several test matrices. Results presented in this document were obtained using sparse matrices corresponding to linear programming problems. QAP matrices come from Netlib [16] while others come from a variety of linear multicommodity network flow generators: A Patient Distribution System (PDS) [17], with instances taken from [18]; RMFGEN [19, 20]; GRIDGEN [21]; TRIPARTITE [22]. Table 1 shows the characteristics of several matrices obtained from such linear programming problems. Matrices were ordered with METIS [23] and renumbered by an elimination tree postorder.

## 4 Reducing overhead

In this section we present several aspects of the work we have done to improve the performance of our sparse Hypermatrix Cholesky factorization. They are based on the fact that a matrix is divided into submatrices and operations are thus performed on blocks (submatrices).

A matrix $M$ is divided into submatrices. We call $M_{br_i,bc_j}$ the data submatrix in block-row $br_i$ and block-column $bc_j$. Figure 2 shows several submatrices within a matrix. The highest cost within the Cholesky factorization process comes from the multiplication of data submatrices. It takes approximately 90% of the total factorization time. In order to ease the explanation we will refer to the three submatrices involved in a product as $A$, $B$ and $C$. For block-rows $br_1$ and $br_2$ (with $br_1 < br_2$) and block-column $bc_j$, each of these blocks is $A \equiv M_{br_2,bc_j}$, $B \equiv M_{br_1,bc_j}$ and $C \equiv M_{br_2,br_1}$. Thus, the operation performed is $C = C - A \times B^T$, which means that submatrices $A$ and $B$ are used to produce an update on submatrix $C$.

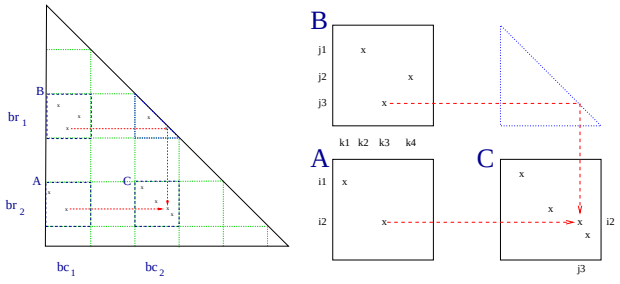


Figure 2: Static partition of a matrix: definition of blocks and example of use.

As we already introduced in section 2 the use of dense data submatrices allows for the use of level 3 Basic Linear Algebra Subroutines (BLAS3) which is the traditional way to obtain performance in a portable way. However, a certain number of zero elements can be stored in data submatrices and used during the factorization operation. This reduces the effective number of calculations performed since operations on such zeros are unnecessary. When the block size used is small, the number of zeros can be greatly reduced. However, the performance obtained from BLAS3 routines drops heavily. Consequently, there is a trade-

Table 1: Matrix characteristics: application of Interior Point Methods.

| Matrix | Dimension | NZs | NZs in L[a] | Density | Flops to factor[b] |
|---|---|---|---|---|---|
| GRIDGEN1 | 330430 | 3162757 | 130586943 | 0.002 | 278891 |
| QAP8 | 912 | 14864 | 193228 | 0.463 | 63 |
| QAP12 | 3192 | 77784 | 2091706 | 0.410 | 2228 |
| QAP15 | 6330 | 192405 | 8755465 | 0.436 | 20454 |
| RMFGEN1 | 28077 | 151557 | 6469394 | 0.016 | 6323 |
| TRIPART1 | 4238 | 80846 | 1147857 | 0.127 | 511 |
| TRIPART2 | 19781 | 400229 | 5917820 | 0.030 | 2926 |
| TRIPART3 | 38881 | 973881 | 17806642 | 0.023 | 14058 |
| TRIPART4 | 56869 | 2407504 | 76805463 | 0.047 | 187168 |
| pds1 | 1561 | 12165 | 37339 | 0.030 | 1 |
| pds10 | 18612 | 148038 | 3384640 | 0.019 | 2519 |
| pds20 | 38726 | 319041 | 10739539 | 0.014 | 13128 |
| pds30 | 57193 | 463732 | 18216426 | 0.011 | 26262 |
| pds40 | 76771 | 629851 | 27672127 | 0.009 | 43807 |
| pds50 | 95936 | 791087 | 36321636 | 0.007 | 61180 |
| pds60 | 115312 | 956906 | 46377926 | 0.006 | 81447 |
| pds70 | 133326 | 1100254 | 54795729 | 0.006 | 100023 |
| pds80 | 149558 | 1216223 | 64148298 | 0.005 | 125002 |
| pds90 | 164944 | 1320298 | 70140993 | 0.005 | 138765 |

[a] Number of non-zeros in factor L (matrix ordered using METIS).
[b] Number of floating point operations (in Millions) necessary to obtain L from the original matrix (ordered with METIS).

off in the size of data submatrices used for a sparse Cholesky factorization with the hypermatrix scheme.

### 4.1 Using Small Matrix Library (SML) routines

In [24] we introduced our library called Small Matrix Library (SML) which works on small matrices of fixed size. By fixing leading dimensions and loop trip counts at compilation time we can produce very efficient codes if a good compiler is available. Actual parameters of these routines are limited to the initial addresses of the matrices involved in the operation performed. Thus, there is one routine for each matrix size. Each one has its own name in the library. The name includes the values of the leading dimensions and loop trip counts. In the figures, however, we refer to any of them as *mxmts_fix*. Although the most important kernel is the matrix multiplication operation, we have also included in the library the SYRK (Symmetric Rank K update) and TRSM (Solve Triangular System of equations) since these operations appear within a Cholesky factorization.

The matrix multiplication performed in all routines benchmarked in this section uses the first matrix without transposition, the second matrix transposed, and subtracts the result from the destination matrix. This is the reason why we call the BLAS matrix multiplication routine *dgemm_nts*. This operation appears within a Cholesky factorization of a matrix into a lower triangular matrix $L$.

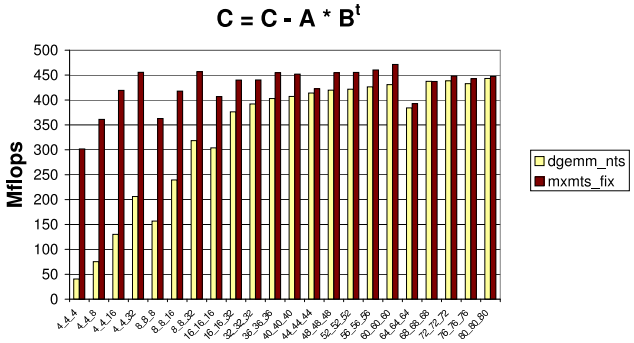


Figure 3: Performance of different $A \times B^T$ routines for several matrix sizes on an R10000.

Figure 3 shows the performance of routines *dgemm_nts* and *mxmts_fix* for several matrix sizes on a 250 MHz MIPS R10000 processor. The first

level instruction and data caches have size 32 Kbytes. There is a secondary unified instruction/data cache with size 4 Mbytes. This processor's theoretical peak performance is 500 Mflops. The vendor BLAS routine *dgemm_nts* yields very poor performance for very small matrices getting better results as matrix dimensions grow towards a size that fills the L1 cache. This is due to the overhead of passing a large number of parameters, checking for their correctness, and scaling the matrices (*alpha* and *beta* parameters in *dgemm*). This overhead is negligible when the operation is performed on large matrices but too large for operation on small matrices. Also, since the code is optimized for large matrices, further overhead can appear in the inner code by the use of techniques such as strip mining or data copying. Next, we show that this is not adequate for our hypermatrix Cholesky factorization.

Figure 4 shows results of the HM Cholesky factorization on an R10000 for matrix QAP15 from the Netlib set of sparse matrices corresponding to linear programming problems [16]; and problem pds20 from a Patient Distribution System (20 days) [18]. Ten submatrix sizes are shown: 4x4, 4x8, 4x16, ... 32x32. *Effective Mflops* are presented. They refer to the number of useful floating point operations performed per second. Two bars are shown for each submatrix size. They refer to the performance obtained for each matrix multiplication routine used internally.

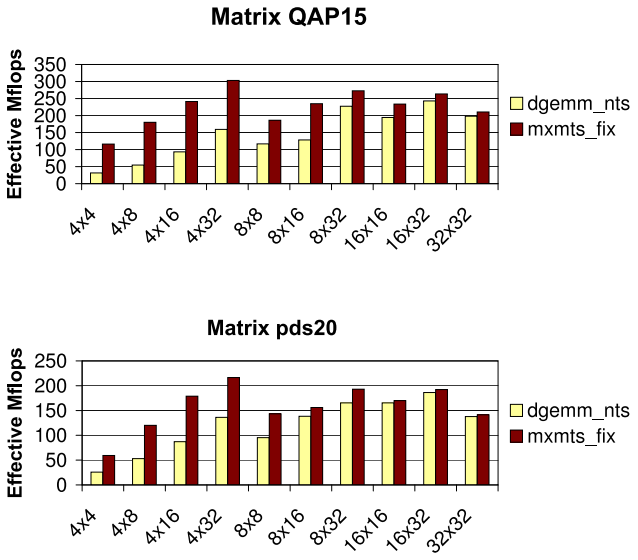


Figure 4: Impact of the matrix multiplication routine on sparse hypermatrix Cholesky. Mflops obtained using different $A \times B^T$ codes in the factorization of matrices QAP15 and pds20 on an R10000.

When *dgemm_nts* is used, the best performance is usually obtained with data submatrices of size $16 \times 16$ or $16 \times 32$. Since the amount of zeros used can be large, the effective performance is quite low. Using *mxmts_fix* however, smaller submatrix sizes usually produce better results than larger submatrix sizes. Particularly effective in this application is the use of rectangular matrices due to the fill-in produced by the Cholesky factorization. For instance, using $4 \times 16$ or $4 \times 32$ submatrix sizes, the routine used yields very good performance. Since the number of operations on zeros is considerably lower, the effective Mflops obtained are much higher than those of any other combination of size and routine.

The use of a fixed dimension routines in the SML library speeded up our Cholesky factorization an average of 12% for our matrix test suite.

Information about the creation of the SML was given in [24]. The application of SML to sparse hypermatrix Cholesky was presented in [25].

### 4.2 Rectangular data submatrices

As seen from the results shown in the preceding section, the use of rectangular data matrices can favor performance. For instance, on the R10000, we store sparse matrices with subblocks of size $4 \times 32$ and use routine *mxmts_4_4_4_4_4_32* as the internal matrix multiplication routine. The characteristics of a sparse Cholesky factorization explain this. Let us assume that the lower triangular matrix (L) is stored and used. Often an off-diagonal nonzero produces updates on other positions in the same row to its right. It is well known that fill-in can be introduced: positions which were originally zero become different from zero after the factorization. Since this updates are produced in a row-wise fashion, the rectangular shape is better suited for storing data submatrices.

### 4.3 Bit Vectors

We want to be able to avoid unnecessary matrix multiplications between matrices with elements in disjoint columns. What we need to know is whether a column within a data submatrix has any non-zero elements or not. We associate a set of bits to each data submatrix. We refer to such a set of bits as *bit vector*. Figure 5 shows a data submatrix with a bit vector associated to it.

Each bit in the vector is used to point to the existence of any non-zero in the corresponding column. For in-

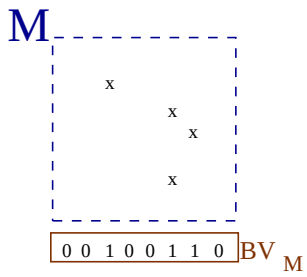


Figure 5: Bit Vectors: Definition

stance, consider matrix $B$ in figure 6a. Let us consider column indices start at 1 (Fortran indexing). There are non-zero elements only in columns $k_2 = 3$, $k_3 = 4$ and $k_4 = 7$. Thus, only bits 3, 4 and 7 in $BV_B$ will be different from 0. A bit-wise AND between bit vectors corresponding to matrices $A$ and $B$ can be used to decide whether the matrix multiplication between those matrices is necessary or not. If a single bit of the bit-wise AND results to be 1 then we need to perform the operation (figure 6a). If all bits are zero, then we can skip it (figure 6b). This test can be done in a couple of CPU cycles with an AND operation followed by a comparison to zero. The creation of the bit vectors can be done initially, when the hypermatrix structure is prepared using the symbolic factorization information. The overhead for their creation is negligible.

In [26] we showed that bit vectors can be effective to improve performance as long as windows are not used. Results can be found in section 5.

### 4.4 Windows within data submatrices

In order to reduce the storage and computation of zero values, we define *windows* of non-zeros within data submatrices in a way similar to that described in [27]: by keeping information about the actual space within a data submatrix which stores non-zeros (dense window) Figure 7a shows a window of non-zero elements within a larger block. The window of non-zero elements is defined by its top-left and bottom right corners. All zeros outside those limits are not used in the computations. Null elements within the window are still stored and computed. Storage of columns to the left of the window's leftmost column is avoided since all their elements are null. Similarly, we do not store columns to the right of the window's rightmost column. However, we do store zeros over the window's upper row and/or underneath its bottom row whenever these window's boundaries are different from the data submatrix boundaries, i.e. whole data submatrix

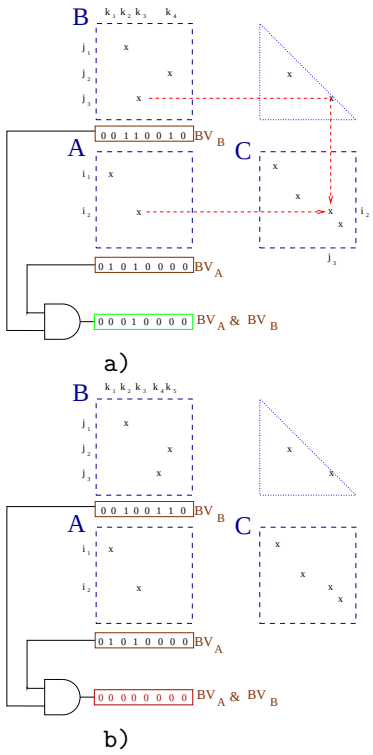


Figure 6: Using Bit Vectors. **a)** $BV_A \& BV_B \neq 0$: operation must be performed. **b)** $BV_A \& BV_B = 0$: operation can be avoided.

columns are stored from the leftmost to the rightmost columns in a window. We do this to have the same leading dimension for all data submatrices used in the hypermatrix. Thus, we can use our specialized SML routines which work on matrices with fixed leading dimensions. Actually, we extended our SML library with routines which have the leading dimensions of matrices fixed, while the loop limits may be given as parameters. Some of the routines have all loop limits fixed, while others have only one, or two of them fixed. Other routines have all the loop limits given as parameters. The appropriate routine is chosen at execution time depending on the windows involved in the operation. Thus, although zeros can be stored above or underneath a window, they are not used in computation. Zeros can still exist within the window but, in general, the overhead is greatly reduced.

The use of windows of non-zero elements within blocks allows for a larger default block size. When blocks are quite full, operations performed on them can be rather efficient. However, in those cases where only a few non-zero elements are present in a block, or the intersection of windows results in a small area, only

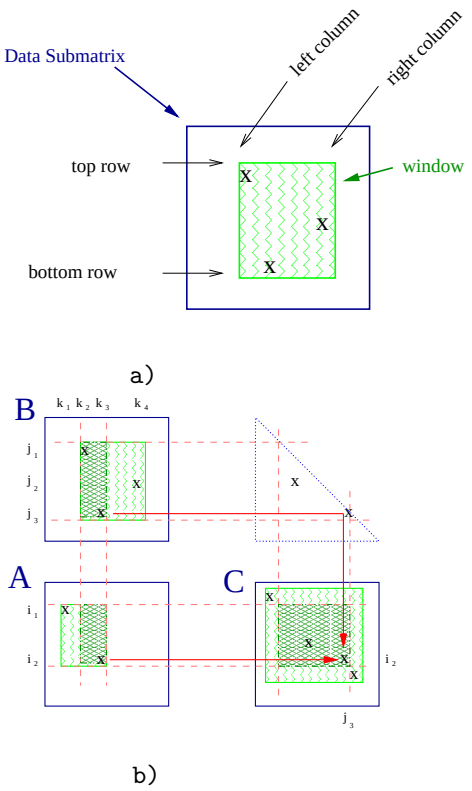


Figure 7: Windows within dense submatrices. **a)** A data submatrix and a window within it. **b)** Windows can reduce the number of operations.

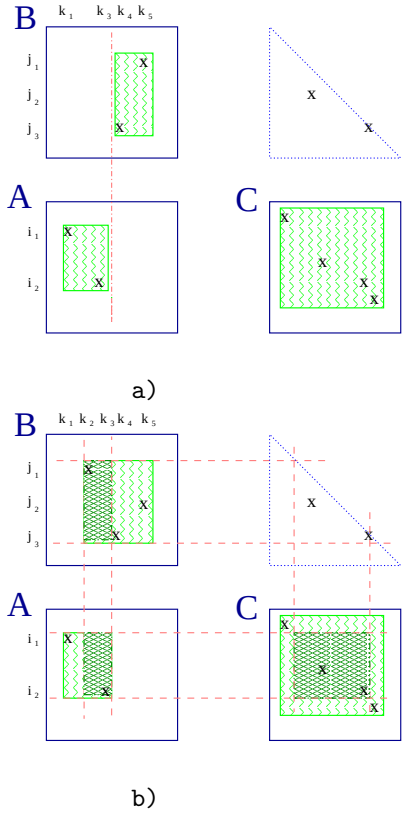


Figure 8: Windows: column-wise intersection. **a)** Disjoint windows can avoid matrix products. **b)** Windows can be ineffective to detect false intersections.

a subset of the total block is computed (dark areas within figure 7b).

When the column-wise intersection of windows in matrices $A$ and $B$ is null, we can avoid the multiplication of these two matrices (figure 8a). There are cases where the window definition we have used is not enough to avoid unnecessary operations. Consider figure 8b: there is a column-wise intersection of windows in $A$ and $B$. Thus, we would perform a product using the dark area within the three matrices involved.[1] Results will be presented in section 5.

## 5 First results and analysis

In this section we present and analyze the results we have obtained on our matrix test suite on a 250 Mhz MIPS R10000 Processor with a theoretical peak performance of 500 Mflops. Matrices come from Interior Point Methods (IPM) of linear programming. Table 1 shows their characteristics. Matrix families are separated by dashed lines in the figures. We present results for several codes. The processing done prior to the factorization is the same for all of them. The original sparsity pattern is used. Data values, however, are generated to obtain a positive definite matrix suitable for Cholesky factorization. Then, the sparse matrix is reordered using METIS [23] and renumbered by an elimination tree postorder [15].

In the sparse HM Cholesky code, after the elimination tree postorder we perform a symbolic factorization. Afterwards, we build the HM data structure. Finally, the resulting hypermatrix is factored.

Next, we present results for several variants of our code and compare their performance with that of a supernodal code. Afterwards, in section 5.1, we will analyze each variant of our code in order to explain their performance.

[1] However, if we look at the elements within those matrices we can see that the product $A \times B^T$ will produce a null update on $C$. In this case, the usage of bit vectors would be useful and could avoid this operation.

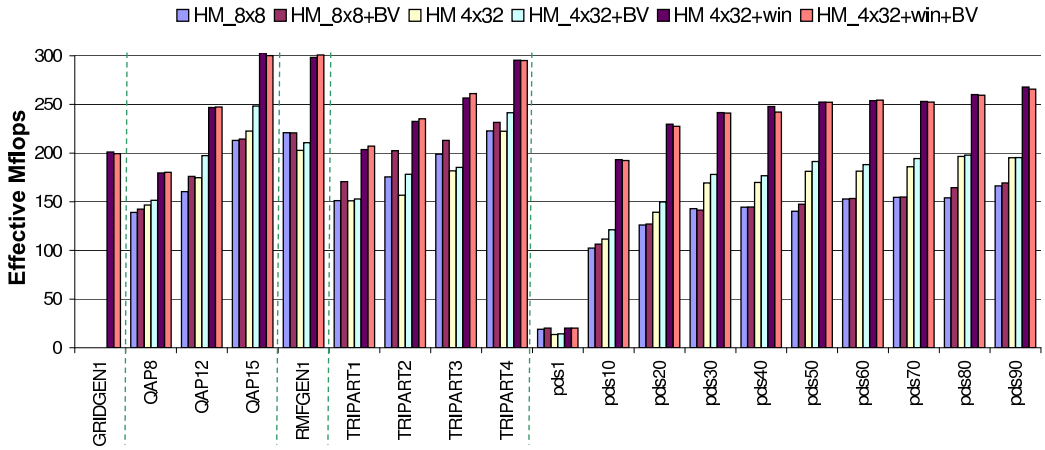


Figure 9: Performance of hypermatrix Cholesky with bit vectors (BV) and windows (win).

**HM ± windows ± BVs**

We have used SML [24] routines to improve our sparse matrix application based on hypermatrices [25]. A fixed partitioning of the matrix has been done to be able to test the impact of each overhead reduction technique used. Upper pointer levels in the HM map submatrices of size $32 \times 32$ and $512 \times 512$. This 2D layout of data submatrices exploits the cache hierarchy of the MIPS R10000 conveniently. We present results obtained with and without bit vectors for two data submatrix sizes: $8 \times 8$ and $4 \times 32$. For the latter we also introduce the usage of windows.

Figure 9 summarizes these results. Matrix families are separated by dashed lines. Defining data submatrices of size $4 \times 32$ yields better results than using size $8 \times 8$. The usage of windows clearly improves the performance of our sparse hypermatrix Cholesky algorithm. We observe that the usage of bit vectors can improve performance slightly when windows are not used. When windows are used, however, bit vectors are not effective at all.

**Supernodal Cholesky (Ng-Peyton) vs HM: 1D vs 2D layouts**

In this section we present results obtained by a supernodal (SN) block Cholesky factorization [2]. It takes as input parameters the cache size and unroll factor desired. This algorithm performs a 1D partitioning of the matrix. A supernode can be split into *panels* so that each panel fits in cache. This code has been widely used in several packages such as LIPSOL [28], PCx [29], IPM [30] or SparseM [31]. Although the test matrices we have used are in most cases very sparse, the number of elements per column is in some cases large enough so that a few columns fill the first level cache. Thus, a one-dimensional partition of the input matrix produces poor results. As the problem size gets larger, performance degrades heavily.

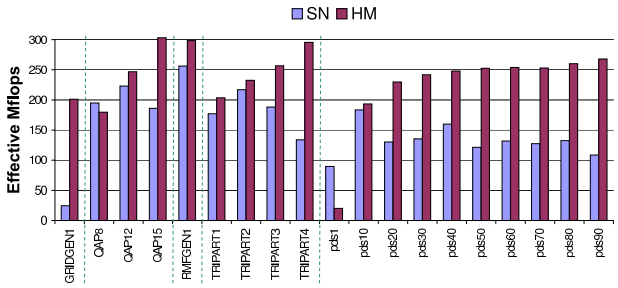


Figure 10: SN vs HM performance.

Figure 10 compares the best result obtained with each algorithm for the whole set of test matrices. We have included matrix pds1 to show that for small matrices the hypermatrix approach is usually very inefficient. This is due to the large overhead introduced by blocks which have plenty of zeros. For large matrices however, blocks are quite dense and the overhead is much lower. Performance of HM Cholesky is then much better than that of the supernodal algorithm. This is due to the better usage of the memory hierarchy: locality is properly exploited with the two dimensional partitioning of the matrix which is done in a recursive way using the HM structure.

We conclude that a two dimensional partitioning of the matrix is necessary for large sparse matrices. The overhead introduced by storing zeros within dense data blocks can be reduced by keeping information about a dense subset (window) within each data submatrix. Although some overhead still remains, the performance of our sparse hypermatrix Cholesky is up to an order of magnitude better than that of a supernodal block Cholesky which tries to use the cache memory properly by splitting supernodes into panels. Using windows and SML routines our HM Cholesky often gets over half of the processor's peak performance for medium and large size matrices factored in-core. Further details can be found in [26].

## 5.1 Analysis of sparse hypermatrix Cholesky

Next, we analyze the performance of variants of our sparse HM Cholesky implementation. In all cases the code follows a right looking scheduling with a fixed partitioning of the hypermatrix (the elimination tree is consequently not used at all). We use a $4 \times 32$ data submatrix size as stated in section 4.2.

### Usage of windows within data submatrices

Figure 11 shows the results obtained with several variants of our sparse HM Cholesky code on matrices taken from IPM. The first and second bars allow for the evaluation of the usage of windows within data submatrices (in a 2D layout of such submatrices). The usage of windows clearly improves the performance of our sparse hypermatrix Cholesky algorithm.

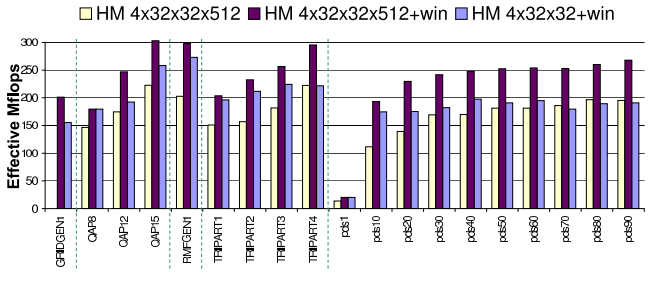


Figure 11: Sparse HM Cholesky: Usage of windows and 2D block layout.

The reason is the large reduction in operations on zeros when windows are used within data submatrices. Figure 12 shows the increase in number of floating point operations in sparse HM Cholesky with blocks of size $4 \times 32$ w.r.t. the minimum (with no operations on zero elements). The number of unnecessary operations on zeros elements is very large for all matrices. When windows are used, however, there is a large reduction in the number of operations on zeros.

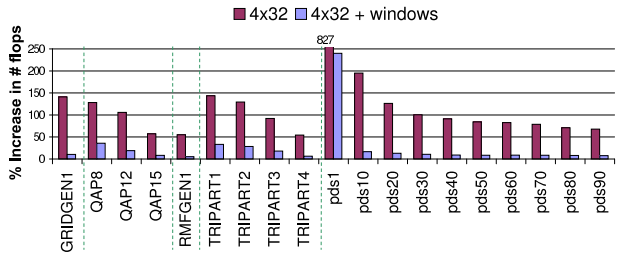


Figure 12: Increase in number of floating point operations in sparse HM Cholesky w.r.t. the minimum: windows reduce the number of operations on zeros.

### 2D layout and scheduling

In [32], we have compared 1 and 2 dimensional recursive layouts of data and computations on data blocks. The second and third bars in figure 11 show the results of 2D and 1D data layouts and scheduling of the data submatrices (windows are used in both cases). The former uses upper levels in the HM with sizes $32 \times 32$ and $512 \times 512$: each of them allows for efficient use of the corresponding level of cache. For the latter, we define only an upper level with sizes $32 \times 32$. We observe that a 2D data layout and scheduling of the computations is beneficial to the efficient computation of the Cholesky factorization of large sparse matrices.

### Overhead: operations on zeros

We have included matrix pds1 in our matrix test suite to show that for small matrices the hypermatrix approach is usually very inefficient. This is due to the large overhead introduced by blocks which have many zeros. Figure 13 shows the percentage of increase in number of flops in sparse HM Cholesky (using windows within data submatrices of size $4 \times 32$) w.r.t. the minimum (using a Compact Sparse Row storage). For large matrices however, blocks are quite dense and the overhead is much lower. Then, sparse HM Cholesky can obtain over half of the processor's peak performance.

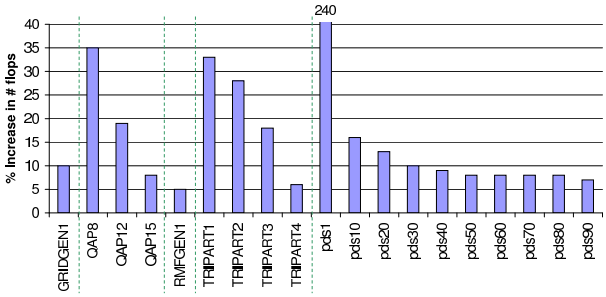


Figure 13: Sparse HM Cholesky using windows in data submatrices of size 4x32: Increase in number of floating point operations.

**Matrix Multiplication: efficiency of different routines**

We focus on matrix multiplication because it is, by far, the most expensive operation within the Cholesky factorization. We use four matrix multiplication operations. Figure 14 shows graphically the data used by each routine.

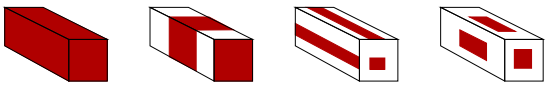


Figure 14: Four matrix multiplication routines deal with different input matrices.

*FULL* refers to the routine where all elements in data submatrices are used, i.e. no windows are used. *WIN_1DC* names the routine where windows are used for columns while *WIN_1DR* indicates the equivalent applied to rows. Finally, *WIN_2D* denotes the case where windows are used for both columns and rows.

Figure 15a shows the percentage of calls to each matrix multiplication routine for each matrix in our test suite. Figure 15b shows the percentage of multiplication operations performed by each of the four $A \times B^T$ routines we use. We can observe that the number of floating point operations is not proportional to the number of calls to each matrix multiplication subroutine. For instance, one call to routine FULL for data submatrix size $4 \times 32$ produces 512 ($4 \times 4 \times 32$) multiply-subtract operations, while one call to routine WIN_2D computes less operations (which can be as low as one).

The explanation for the differences in computation time come from the efficiency of each code. FULL refers to the routine where all matrix dimensions are fixed. This is the most efficient of the four routines and can perform matrix multiplications faster. WIN_2D denotes the case where windows are used for both columns and rows. Thus, for the latter, no dimensions are fixed at compilation time and it becomes the least efficient of all four routines. WIN_1DC computes matrix multiplications more efficiently than WIN_1DR: using (Fortran) column-wise storage the latter stores null elements within a column due to the fixed leading dimension used. These null elements are loaded (although not used) into the cache when other nonzero elements mapped into the same cache line are accessed. This fact reduces the effective memory bandwidth.

This is the reason why the performance obtained for matrices in the TRIPARTITE set is better than that obtained for matrices with similar size belonging to the PDS family. The performance obtained for the matrix with the largest number of floating point operations in our test suite, namely matrix GRIDGEN1, is less than half of the theoretical peak for the machine used. This is basically due to the dispersion of data in this matrix which leads to the usage of the FULL $A \times B^T$ routine less than 50% of the time. In addition, a large number of calls is done to the least efficient routine WIN_2D. This routine is used to compute about 10% of the operations.

## 6 Intra-block amalgamation

Amalgamation [5] has been used for a long while in sparse codes. It consists in joining supernodes with different structures allowing for the presence of zeros. This is used to produce larger blocks and thus improve the performance by a better use of the machine resources via BLAS3 routines. Next, we present the way in which we introduce amalgamation in our sparse hypermatrix Cholesky code. We refer to this technique as *Intra-block* amalgamation.

Approximately 90% of the sparse Cholesky factorization time comes from matrix multiplications. Thus, a large effort must be devoted to perform such operations efficiently. We have seen above that we use four codes specialized in the multiplication of two matrices (figure 14). The most efficient is the one on the left. Their efficiency diminishes as we move to the right.

The usage of windows reduces the number of unnec-

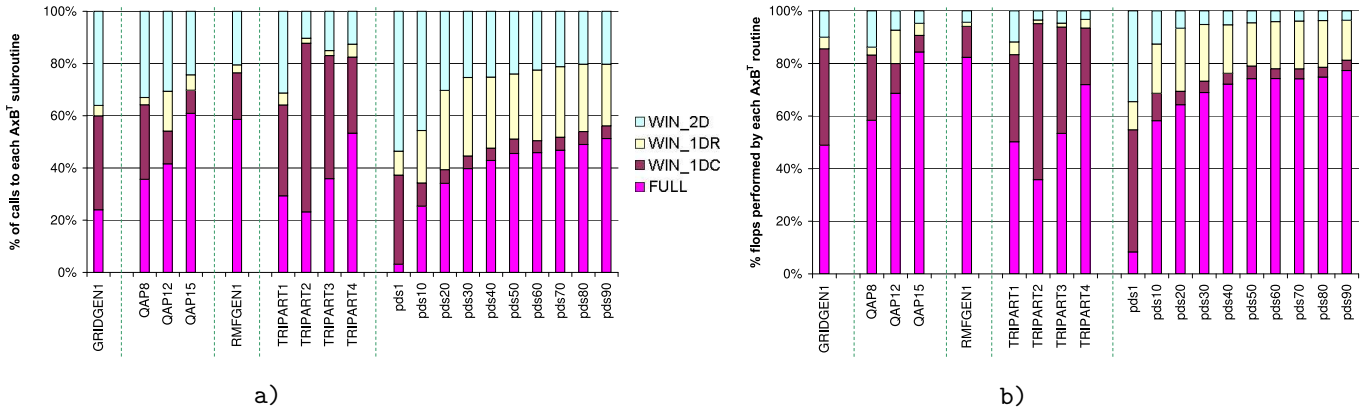


Figure 15: Sparse HM Cholesky: Percentage of **a)** calls to each $A \times B^T$ subroutine type and **b)** flops per $A \times B^T$ subroutine type.

essary operations on zeros. However, routines which work on submatrices with windows have a considerably lower performance than that of the routine for full data submatrices, where all leading dimensions and loop trip counts are fixed at compilation time (see the analysis in [32]). Performing a slightly higher number of operations with a faster routine could some times pay off. For this reason we have decided to add the possibility to extend windows with rows or columns full of zeros. This work was presented in [33].

Figure 16 shows a rectangular data submatrix with a window defined within it. That window saves operations in both columns and rows. Each time this matrix needs to be multiplied by another matrix the *WIN_2D* code will be used. Since this is the slower code amongst our 4 matrix multiplication routines, this can produce a reduction in performance.

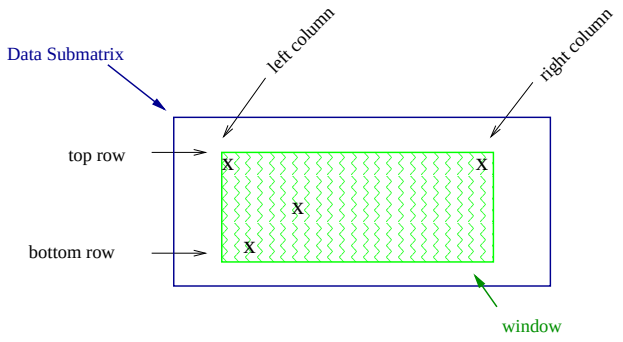


Figure 16: Original data submatrix before intra-block amalgamation.

Figure 17 shows how we can extend the window row-wise. We are aware that the resulting window will have rows full of zeros either at the top or at the bottom. This extension introduces extra overhead due to the extra number of operations on such zeros. However, this intra-block amalgamation can reduce the number of calls to routine *WIN_2D*. Instead, some new calls to *WIN_1DC* can be done if the other operand is either a full matrix or one with windows defined only for columns. Since the latter routine is more efficient than the former, this can result in a performance improvement as long as the number of unnecessary operations on zeros is not too large. We only perform such amalgamation if the dimension of the window is close to the dimension of the data submatrix. We define a threshold for rows and another one for columns.

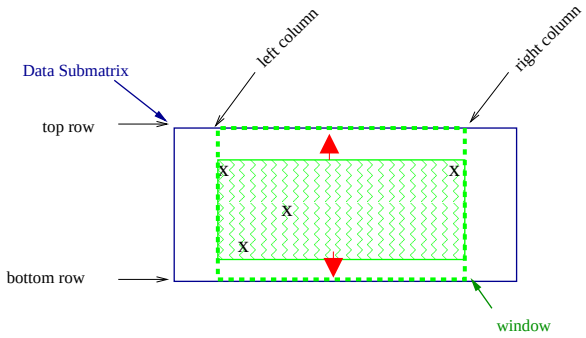


Figure 17: Data submatrix after row-wise intra-block amalgamation.

Figure 18 shows how we can extend the window column-wise. In this case, the resulting window will have columns full of zeros in at least one of its sides. Again, this can reduce the number of times in which routine *WIN_2D* is used. In this case, such calls could

be replaced by calls to *WIN_1DR*.

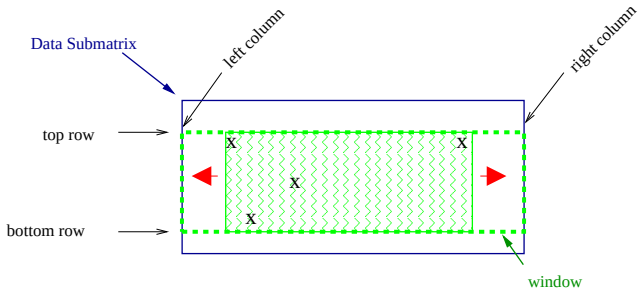


Figure 18: Data submatrix after column-wise intra-block amalgamation.

Finally, figure 19 shows how we can extend the window both row and column-wise. In this case, the resulting window matches the whole data submatrix. Again, this action can reduce the number of times routine *WIN_2D* is used. In this case, such calls could be replaced by calls to any of the 3 other matrix multiplication routines (either *FULL*, *WIN_1DC* or *WIN_1DR*) depending on the other matrix involved in the multiplication.

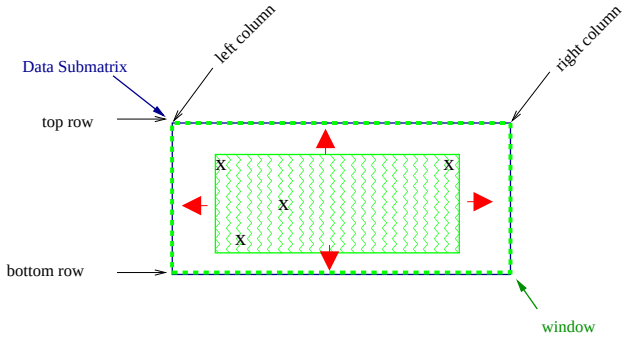


Figure 19: Data submatrix after applying both row and column-wise intra-block amalgamation.

## 6.1 Results

Given a $4 \times 32$ data block, we have introduced intra-block amalgamation in rows and columns. We have used values from 0 to 3 for row-wise amalgamation and 0 to 9 for column-wise amalgamation. A row-wise amalgamation with value 3 (amr=3) means that no routines dealing with windows in rows would be used. Next, we show details on the performance obtained using several values of row and column-wise amalgamation for some sparse matrices: QAP8 (fig. 20), TRIPART1 (fig. 21), and pds20 (fig. 22). Effective Mflops are presented. They refer to the number of useful floating point operations performed per second. This metric excludes useless operations on zeros performed by the HM Cholesky algorithm when data submatrices contain zeros.

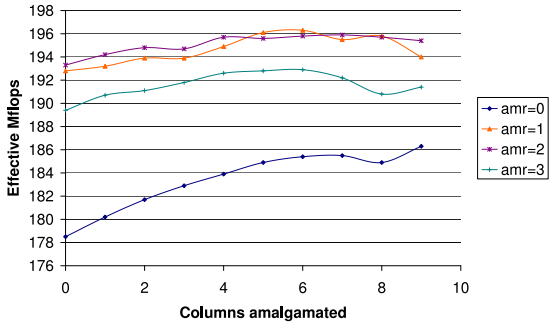


Figure 20: Intra-block amalgamation: matrix QAP8.

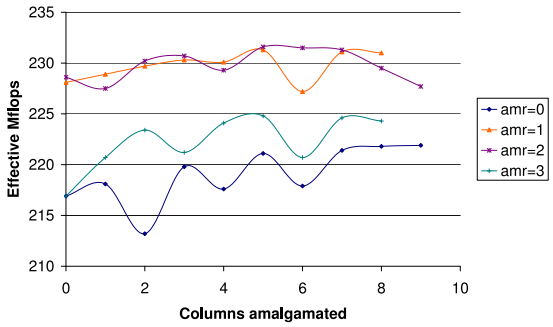


Figure 21: Intra-block amalgamation: matrix TRIPART1.

In all cases, row-wise amalgamation with values 1 and 2 obtain the best performance. There are several values for column-wise amalgamation with similar performance. We have chosen a value of 5 for the latter since this was often the best one or nearly the best.

Figure 23 shows the performance obtained with our sparse HM Cholesky code with and without intra-block amalgamation. In both cases SML routines and windows have been used. The block sizes are also the same in both cases: $4 \times 32$ as the data submatrix size; $32 \times 32$ for the next pointer level, and $512 \times 512$ as the upper pointer level.

## 6.2 Intra-block amalgamation: conclusions

A performance improvement was always achieved for row-wise amalgamation with values 1 and 2. A value around 5 was usually the best for column-wise amalgamation on the matrices tested. Using intra-block amalgamation by rows with a value of 1 (amr1), and

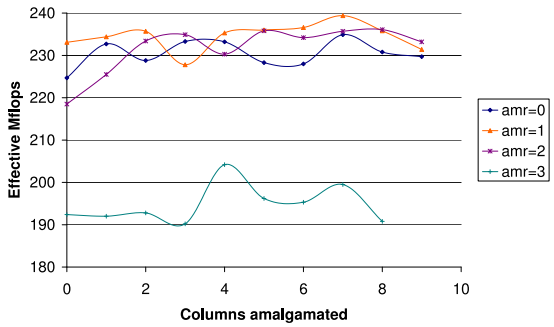


Figure 22: Intra-block amalgamation: matrix pds20.

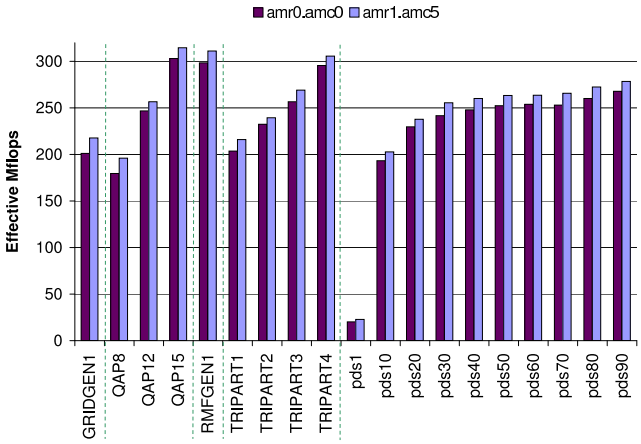


Figure 23: Performance of sparse HM Cholesky without and with intra-block amalgamation.

by columns with a value of 5 (amc5), produces a performance improvement between 3% and 12.9% with an average of 5.3% on our sparse matrix test suite on an R10000 processor.

When we introduce intra-block amalgamation we increase the overhead associated with the unnecessary operations on zeros. Thus, the next step towards performance improvements could come from a new data storage for data submatrices. In the future, we plan to add the possibility to store data submatrices as supernodes. In this way we could join several non-consecutive rows in a consecutive fashion in the same way as a code based on supernodes. We believe this could reduce the overhead since the storage and operation on zeros could be avoided while the efficient execution with BLAS3 would still remain.

# 7 Results: Comparison with other packages

Figure 24 shows the results obtained by five different sparse Cholesky factorization codes on the set of matrices introduced above on a MIPS R10000 processor. The first bar corresponds to a supernodal left-looking block Cholesky factorization (SN-LL (Ng-Peyton)) [2] already discussed above.

The second bar shows the performance obtained by the sequential version of a 2D block-oriented approach [34] as found in the SPLASH-2 [35] suite. According to [34] submatrices are kept in a two-dimensional data layout. However, this code fails to produce efficient factorizations for large matrices.

The third and fourth bars correspond to sequential versions of the supernodal left-looking (SN-LL) and supernodal multifrontal (SN-MF) codes in the TAUCS package (version 2.2) [36]. In these codes the matrix is represented as a set of supernodes. The dense blocks within the supernodes are stored in a recursive data layout matching the dense block operations. The performance obtained by these codes is quite uniform.

Finally, the fifth bar shows the performance obtained by our right looking sparse hypermatrix Cholesky code (HM). We have used windows within data submatrices and SML routines to improve our sparse matrix application based on hypermatrices. Values 1 for row-wise and 5 for column-wise amalgamation have been defined. A three-level fixed partitioning of the matrix has been used. We present results obtained for data submatrix sizes $4 \times 32$ and upper hypermatrix levels with sizes $32 \times 32$ and $512 \times 512$. In general, the sparse hypermatrix Cholesky factorization improves its efficiency as the problem dimension gets larger.

On this set of matrices, the sparse hypermatrix Cholesky code is clearly the best amongst all five codes. Its performance is considerably better than the others for large matrices.

# 8 Other considerations on sparse hypermatrix Cholesky factorization

## 8.1 Porting efficiency to a new platform

In this section we summarize the port of our application from a machine based on a MIPS R10000 processor to a platform with an Intel Itanium2 processor. We address the optimization of the sparse Cholesky

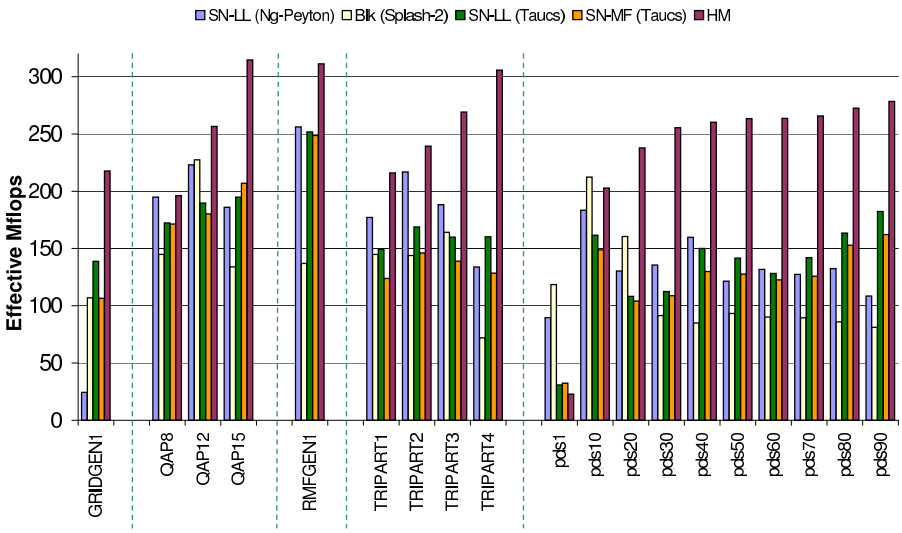


Figure 24: Performance of several sparse Cholesky factorization codes.

factorization based on a hypermatrix structure following several steps.

First we create our Small Matrix Library (SML) automatically as explained in section 4.1 . We have used $4 \times 32$ as data submatrix dimensions since these were the ones providing best performance on the R10000. Later in this document we will present the results obtained with other matrix dimensions. As we mentioned in section 4.4 we use windows within data submatrices since they have proved effective in reducing both the storage of and operation on zero elements.

Afterwards, we experiment with different intra-block amalgamation values. As an example, figure 25 shows the performance obtained using several values of intra-block amalgamation on the hypermatrix Cholesky factorization of matrix pds20 on an Itanium2. Each curve corresponds to one value of amalgamation along the rows. The curve at the top (amr=3) corresponds to the largest amalgamation threshold along the rows: three, for submatrices consisting of four rows. The one at the bottom corresponds to the case where this type of amalgamation is disabled (amr=0).

On the Itanium2 and using our matrix test suite, the worst performance is obtained when no amalgamation is done along the rows (amr=0). As we allow increasing values of the intra-block amalgamation along the rows the overall performance increases. The best performance for this matrix dimensions is obtained when amalgamation along the rows is three. This means that, for data submatrices of size $4 \times 32$, we will use no windows along the rows. This suggests that a larger number of rows could provide improved performance. We will analyze this issue in section 8.4. As we move right on the curves, we observe the performance obtained with increasing values of amalgamation threshold along the columns. The difference is often low, but the best results are obtained with values ranging from six to ten. This work was presented in [37].

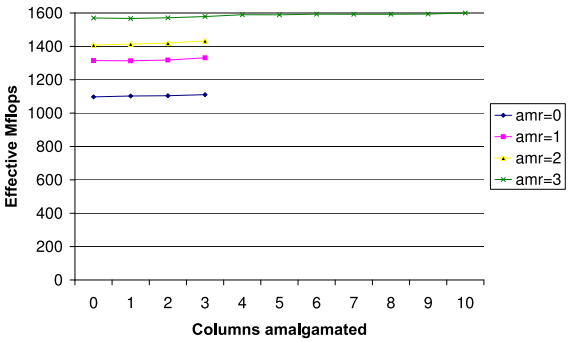


Figure 25: Sparse HM Cholesky on an Intel Itanium2: Performance obtained with different values of intra-block amalgamation on submatrices of size $4 \times 32$ on matrix pds20.

Notice that the threshold values providing best performance on the R10000 were different: one on the rows and five on the columns. The reason for this is

the different relative performance of the matrix multiplication routines. The ability of one compiler to generate efficient code for routines which take windows into account can be different from its aptitudes when dealing with routines which do not use windows at all. And these capabilities can be different from those of the compiler found on another platform. As a consequence we get different optimal values for the intra-block amalgamation thresholds on each platform.

## 8.2 Sparse matrix reordering

A sparse matrix can be reordered to reduce the amount of fill-in produced during the factorization. Also it can be reordered aiming to improve parallelism. In all the tests presented so far we have been using METIS [38] as the reorder algorithm. This algorithm is considered a good algorithm when a parallel Cholesky factorization has to be done. Also, when matrices are relatively large, graph partitioning algorithms such as METIS usually work much better than MMD, the traditional Minimum Degree ordering algorithm [39]. METIS implements a Multilevel Nested Dissection algorithm. This sort of algorithms keep a global view of the graph and partition it recursively using the Nested Dissection approach [40] splitting the graph in smaller disconnected graphs. When these subgraphs are considered small, a local ordering algorithm is used. METIS uses the MMD algorithm for the local phase. METIS changes to the local ordering strategy when the number of nodes is less than 200.

Although our current implementation is sequential, we have tried to improve the sparse hypermatrix Cholesky for the matrix orders produced by METIS. In this way, the improvements we get are potentially useful when we go parallel. However, we have also experimented with other classical algorithms. On small matrices in our matrix test suite, when the Multiple Minimum Degree (MMD) [41] algorithm was used the hypermatrix Cholesky factorization took considerably less time. However, as we use larger matrices (RMFGEN1, pds50, pds60, ...) the time taken to factor the resulting matrices became several orders of magnitude larger than that of METIS. We have also tried older methods [42] such as the Reverse Cuthill-McKee (RCM) and the Refined Quotient Tree (RQT). RCM tries to keep values in a band as close as possible to the diagonal. RQT tries to obtain a tree partitioning of a graph. These methods produce matrices with denser blocks. However the amount of fill-in is so large that the factorization time gets very large even for medium sized matrices.

### 8.2.1 Ordering for Linear Programming problems

Working with matrices which arise in linear programming problems we may use sparse matrix ordering algorithms specially targeted for these problems. The METIS sparse matrix ordering package offers some options which the user can specify to change the default ordering parameters. Following the suggestions found in its manual we have experimented with values which can potentially provide improved orderings for sparse matrices coming from linear programming problems. There are eight possible parameters. We skip the details of these parameters for brevity. However, for the sake of completeness, we include the values we have used: 1, 3, 1, 1, 0, 3, 60, and 5.

Using this configuration usually produces better sparse matrix orderings than the default configuration for the type of problems we are dealing with. This ordering results in faster sparse Cholesky numerical factorization. However, this comes at the expense of a larger ordering process which incurs in a larger ordering time. We have measured both the ordering and numerical factorization time obtained using the two configurations of METIS discussed above. We must take into account that Interior Point Methods (IPM), i.e. the methods which use the sparse Cholesky factorization on linear programming problems have an iterative nature. In each iteration a sparse Cholesky factorization is performed on matrices with different data but the very same structure. Thus, the ordering process can be performed only once, while the numerical factorization is repeated many times on different data. Until now, we have been using METIS default configuration. To evaluate the potential for the new ordering parameters we have measured the number of iterations necessary to amortize the cost of the improved matrix reordering. Figure 26 presents the number of iterations after which the specific ordering starts to be advantageous. We can see that in many cases the benefits are almost immediate. Consequently, in the rest of this work we will present results using the modified ordering process specific for matrices arising in linear programming problems. This work appeared in [37].

## 8.3 Data submatrix storage: compression

In section 8.1 we showed that on an Intel Itanium2 and using data submatrices of size $4 \times 32$ the optimal threshold for amalgamation in the rows was three. This suggests that using data submatrices with

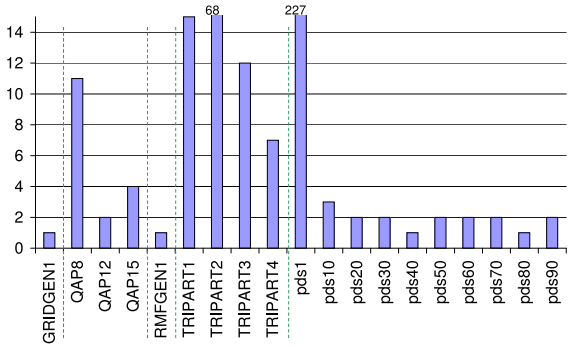


Figure 26: Number of iterations necessary to amortize cost of improved ordering.

a larger number of rows should be tried. However, the larger the blocks, the more likely it is that they contain zeros. As we have discussed in previous sections, the presence of zero values within data submatrices causes some drawbacks. Obviously, the computation on such null elements is completely unproductive. However, we allow them as long as operating on extra elements allows us to do such operations faster. On the other hand, a different aspect is the increase in memory space requirements with respect to any storage scheme which keeps only the nonzero values. Next, we present the way in which we can avoid some of this additional storage.

As we mentioned in section 4.4, we use windows to reduce the effect of zeros in the computations. However, we still keep the zeros outside of the window. Figure 27 shows two data submatrices stored contiguously. Even when each submatrix has a window we store the whole data submatrix as a dense matrix.

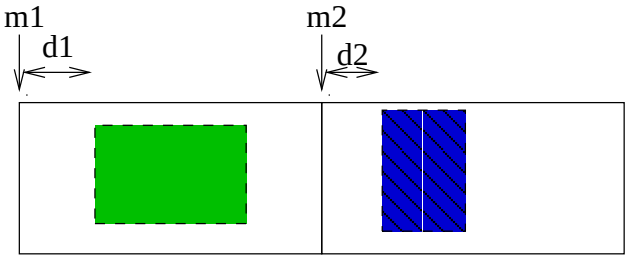


Figure 27: Data submatrices before compression.

However, we could avoid storing zeros outside of the window, i.e. just keep the window as a reduced dense matrix. This approach would reduce storage but has a drawback: by the time we need to perform the operations we need to either uncompress the data submatrix or reckon the adequate indices for a given operation. This could have a performance penalty for the numerical factorization. To avoid such overhead we store data submatrices as shown in figure 28.

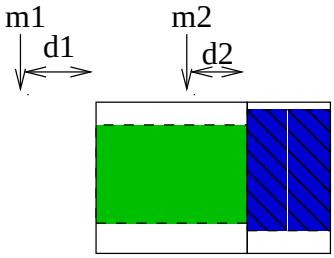


Figure 28: Data submatrices after compression.

We do not store zeros in the columns to the left and right of the window. However, we do keep zeros above and/or underneath such window. We do this for two reasons: first, to be able to use our routines in the SML which have all leading dimensions fixed at compilation time (we use Fortran, which implies column-wise storage of data submatrices); second, to avoid extra calculations of the row indices. In order to avoid any extra calculations of column indices, we keep pointers to an address which would be the initial address of the data submatrix if we were keeping zero columns on the left part of the data submatrix. Thus, if the distance from the initial address of a submatrix and its window is $d_x$ we keep a pointer to the initial address of the window with $d_x$ subtracted from it. These pointers are kept in the last level of pointers in the hypermatrix. Thus, when the numerical factorization takes place, we can take advantage of performing dense operations on data submatrices, i.e. use our efficient routines working on small matrices and avoid complex calculation of indices. Note that in the presence of windows, we will never access the zero columns to the left or right of a window, regardless of having them stored or not.

Figure 29 presents the savings in memory space obtained by this method compared to storing the whole data submatrices of size $4 \times 32$. We can observe that the reduction in memory space is substantial for all matrices. This work was presented in [37].

### 8.4 Larger data submatrices: performance

The reduction in memory space allows us to experiment with larger matrix sizes (except on the largest

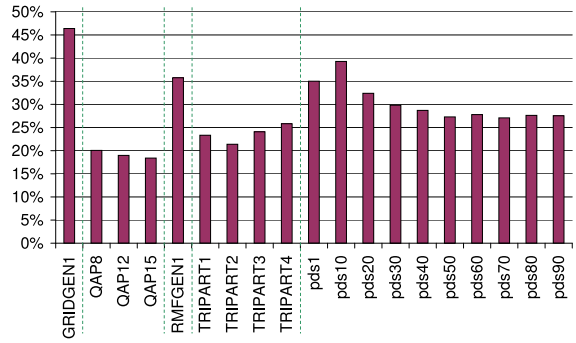


Figure 29: HM structure: reduction in space after submatrix compression.

matrix in our test suite: GRIDGEN1). Figure 30 presents the variation in execution time on an Intel Itanium2 processor when the number of rows per data submatrix was increased to 8, 16 and 32. In almost all cases the execution time increased. Only matrices of the TRIPARTITE family benefited from the use of larger submatrices.

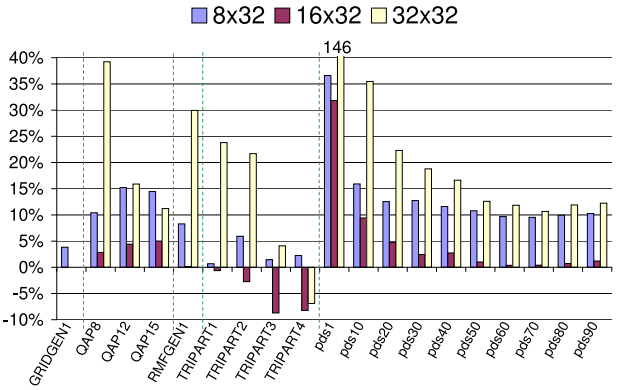


Figure 30: Sparse HM Cholesky: variation in execution time for each submatrix size relative to size $4 \times 32$.

We must note that the performance obtained with matrices of size $8 \times 32$ is worse that that obtained with submatrices of size $16 \times 32$. The reason for this is the relative performance of the routines which work on each matrix size. The one with larger impact on the overall performance of the sparse hypermatrix Cholesky factorization is the one with fixed matrix dimensions and loop trip counts. The corresponding routine for each matrix size obtains the peak performance shown in table 2. We can observe that the efficiency of the routine working on matrices with four rows is similar to the one which works on matrices with eight rows. However, the overhead, in terms of additional zeros, is much larger for the latter. This explains their relative performance. However, the improved performance of the matrix multiplication routine when matrices have 16 rows can pay off. Similarly to the comparison between codes with eight rows and four, using matrices with 32 rows produces a performance drop with respect to the usage of data submatrices with 16 rows.

| $4 \times 32$ | $8 \times 32$ | $16 \times 32$ | $32 \times 32$ |
|---|---|---|---|
| 4005 | 4080 | 4488 | 4401 |

Table 2: Performance of the $C = C - A \times B^T$ matrix multiplication routine for each submatrix size.

## 9 Conclusions

The sparse hypermatrix Cholesky factorization usually improves its performance as the problem size gets larger. There are two reasons for this. The overhead due to unnecessary operations on zeros is usually reduced. The other is that since blocks tend to be larger, more operations are performed by efficient $A \times B^T$ routines. The best performance is obtained from those cases which have few operations on zeros to cause overhead, and which make extensive use of the most efficient matrix multiplication routines.

A two dimensional partitioning of the matrix is necessary for large sparse matrices. Compared to codes which do a 1D partitioning of the matrix, our code can result in a better usage of the memory hierarchy: locality is properly exploited with the two dimensional partitioning of the matrix which is done in a recursive way using the HM structure.

The overhead introduced by storing zeros within dense data blocks in the hypermatrix scheme can be reduced by keeping information about a dense subset (window) within each data submatrix. Although some overhead still remains, using windows and SML routines our sparse HM Cholesky often obtains over half of the processor's peak performance for medium and large sparse matrices factored sequentially in-core. When windows are used bit vectors are unnecessary. In spite of the simple fixed-sized blocking used, the sparse hypermatrix Cholesky factorization which uses windows and SML routines is highly competitive.

The efficient execution of a program requires the configuration of the software to adapt it to the problem being solved and the machine used for finding a solution. We have shown the way in which we can tune our sparse hypermatrix Cholesky factorization code for high performance on a new platform. We have seen that the optimal parameters can be different for each problem type and platform. Thus, we need to adapt the code in search for performance.

Using a combination of techniques (windows within dense data submatrices and intra-block amalgamation) can be quite effective. Our code outperforms some state-of-art codes when working on some matrices taken from Linear Programming problems.

## 10 Future work

Next, we present some ideas for further improvements on sparse hypermatrix Cholesky factorization.

### 10.1 Column versus row storage

We have been using a column-wise storage for data submatrices since this is the standard Fortran order. Now, we want to see whether a row-wise storage could benefit our algorithms. The possible advantage for the sparse Cholesky factorization comes from the fact that about 90% of the time is spent in a matrix multiplication such as $C = C - A * B^T$. Having the data submatrices stored row-wise could result in streaming of all 3 matrices (accessing matrices with stride 1). This would improve spatial locality and data could be brought from upper levels of memory faster. On a Cholesky factorization, assuming Fortran column major storage, this is equivalent to performing operations on an upper triangular matrix ($U$) rather than the lower triangle ($L$).

### 10.2 Supernodes within hypermatrix data submatrices

The results obtained with the intra-block amalgamation in section 6 suggest that replacing dense data submatrices with supernodes could speed up our code. Although the code is regular and fast, using plain dense matrices we produce many unnecessary operations on zeros. We believe that we could improve the effective Mflop rate of our algorithm if we packed the full rows within those submatrices using a supernodal scheme. We purpose to modify the part of the code which corresponds to the data submatrices to be able to deal with blocks stored as supernodes.

## References


[1] Iain S. Duff. Full matrix techniques in sparse Gaussian elimination. In *Numerical analysis (Dundee, 1981)*, volume 912 of *Lecture Notes in Math.*, pages 71–84. Springer, Berlin, 1982.

[2] Esmond G. Ng and Barry W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.*, 14(5):1034–1056, 1993.

[3] Edward Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. *SIAM J. Sci. Comput.*, 17(3):699–713, 1996.

[4] J. W. Liu, E. G. Ng, and B. W. Peyton. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 14(1):242–252, January 1993.

[5] C. Ashcraft and R. G. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Software*, 15:291–309, 1989.

[6] G.Von Fuchs, J.R. Roy, and E. Schrem. Hypermatrix solution of large sets of symmetric positive-definite linear equations. *Comp. Meth. Appl. Mech. Eng.*, 1:197–216, 1972.

[7] A. Noor and S. Voigt. Hypermatrix scheme for the STAR–100 computer. *Comp. & Struct.*, 5:287–296, 1975.

[8] M. Ast, R. Fischer, H. Manz, and U. Schulz. PERMAS: User's reference manual, INTES publication no. 450, rev.d, 1997.

[9] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 13th international conference on Supercomputing*, pages 444–453. ACM Press, 1999.

[10] Jeremy D. Frens and David S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Notices*, 32(7):206–216, 1997.

[11] Vinod Valsalam and Anthony Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience*, 14(10):805–839, August 2002.

[12] David S. Wise. Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In *Euro-Par 2000,LNCS1900*, pages 774–783, September 2000.

[13] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 13th international conference on Supercomputing*, pages 425–433. ACM Press, 1999.

[14] David S. Wise. Representing matrices as quadtrees for parallel processors. *Information Processing Letters*, 20(4):195–199, May 1985.

[15] J. H. W. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1990.

[16] NetLib. Linear programming problems.

[17] W.J. Carolan, J.E. Hill, J.L. Kennington, S. Niemi, and S.J. Wichmann. An empirical evaluation of the KORBX algorithms for military airlift applications. *Oper. Res.*, 38:240–248, 1990.

[18] A. Frangioni. Multicommodity Min Cost Flow problems. Operations Research Group, Department of Computer Science, University of Pisa.

[19] Tamas Badics. RMFGEN generator., 1991.

[20] D. Goldfarb and M. D. Grigoriadis. A computational comparison of the Dinic and network simplex methods for maximum flow. In B. Simeone et al., editors, *FORTRAN Codes for Network Optimization*, Annals of Operations Research, vol. 13, pages 83–124, 1988.

[21] Y. Lee and J. Orlin. GRIDGEN generator., 1991.

[22] Andrew V. Goldberg, Jeffrey D. Oldham, Serge Plotkin, and Cliff Stein. An implementation of a combinatorial approximation algorithm for minimum-cost multicommodity flow. In *Proceedings of the 6th International Conference on Integer Programming and Combinatorial Optimization, IPCO'98 (Houston, Texas, June 22-24, 1998)*, volume 1412 of *LNCS*, pages 338–352. Springer-Verlag, 1998.

[23] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1999.

[24] José R. Herrero and Juan J. Navarro. Automatic benchmarking and optimization of codes: an experience with numerical kernels. In *Int. Conf. on Software Engineering Research and Practice*, pages 701–706. CSREA Press, June 2003.

[25] José R. Herrero and Juan J. Navarro. Improving Performance of Hypermatrix Cholesky Factorization. In *Euro-Par'03,LNCS2790*, pages 461–469. Springer-Verlag, August 2003.

[26] José R. Herrero and Juan J. Navarro. Reducing overhead in sparse hypermatrix Cholesky factorization. In *IFIP TC5 Workshop on High Performance Computational Science and Engineering (HPCSE), World Computer Congress*, pages 143–154. Springer-Verlag, August 2004.

[27] Rolf Fischer, Markus Ast, Hartmut Manz, and Jesus Labarta. A dynamic task graph parallelization approach. In *Fourth Int. Colloquium on Computation of Shell & Spatial Structures*, June 2000.

[28] Yin Zhang. Solving large-scale linear programs by interior-point methods under the MATLAB environment. *Optim. Methods Softw.*, 10(1):1–31, 1998.

[29] J. Czyzyk, S. Mehrotra, M. Wagner, and S. J. Wright. PCx User's Guide (Version 1.1). Technical Report OTC 96/01, Optimization Technology Center, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois 60439, 1997.

[30] Jordi Castro. A specialized interior-point algorithm for multicommodity network flows. *SIAM Journal on Optimization*, 10(3):852–877, September 2000.

[31] Roger Koenker and Pin Ng. SparseM: A Sparse Matrix Package for R, 2003. http://cran.r-project.org/src/contrib/PACKAGES.html#SparseM.

[32] José R. Herrero and Juan J. Navarro. Optimization of a statically partitioned hypermatrix sparse Cholesky factorization. In *Workshop on state-of-the-art in scientific computing (PARA'04),LNCS3732*, pages 798–807. Springer-Verlag, June 2004.

[33] José R. Herrero and Juan J. Navarro. Intra-block amalgamation in sparse hypermatrix Cholesky factorization. In *Int. Conf. on Computational Science and Engineering*, pages 15–22, June 2005.

[34] Edward Rothberg and Anoop Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. *SIAM J. Sci. Comput.*, 15(6):1413–1439, November 1994.

[35] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36. ACM Press, 1995.

[36] D. Irony, G. Shklarski, and S. Toledo. Parallel and fully recursive multifrontal sparse Cholesky. In *ICCS'02,LNCS2330*, pages 335–344. Springer-Verlag, April 2002.

[37] José R. Herrero and Juan J. Navarro. Sparse hypermatrix Cholesky: Customization for high performance. In *Proceedings of The International MultiConference of Engineers and Computer Scientists 2006*, June 2006.

[38] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0*, September 1998.

[39] Nicholas I. M. Gould, Yifan Hu, and Jennifer A. Scott. A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations. Technical Report RAL-TR-2005-005, Rutherford Appleton Laboratory, Oxfordshire OX11 0QX, April 2005.

[40] Alan George. Nested disection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10:345–363, September 1973.

[41] J. W. H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, 1985.

[42] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive-Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.