Intra-Block Amalgamation in Sparse Hypermatrix Cholesky Factorization *

José R. Herrero, Juan J. Navarro

Computer Architecture Dept., Univ. Politècnica de Catalunya
Barcelona, (Spain)
josepr,juanjo@ac.upc.edu

Abstract- **In this paper we present an improvement to our sequential in-core implementation of a sparse Cholesky factorization based on a hypermatrix storage structure. We allow the inclusion of additional zeros in data submatrices to create larger blocks and in this way use more efficient routines for matrix multiplication. Since matrix multiplication takes about 90% of the total factorization time this is an important point to optimize.**

1 Introduction

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 methods. Consequently, there has been great interest in improving its performance [6, 18, 20]. 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 [20]. Operations are thus performed on blocks (submatrices). Our work addresses the optimization of the sparse Cholesky factorization of large matrices. For this purpose, we use a *Hypermatrix* [8] block data structure with static block sizes.

Amalgamation [1] 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. We have borrowed the term to express the deliberate inclusion of zeros in data submatrices in an attempt to improve performance. In this paper we present the way we introduce extra zeros within data submatrices. Then, we show the performance improvement obtained with this technique.

1.1 Background

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. Columns having similar structure are taken as a group. These column groups are called *supernodes* [16]. Some supernodes may be too large to fit in cache and it is advisable to split them into *panels* [18, 20]. 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 work was supported by the Ministerio de Ciencia y Tecnología of Spain (TIN2004-07739-C02-01)

http://people.ac.upc.edu/josepr/

This problem can be reduced by *amalgamating* several supernodes into a single larger one [1]. 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.

1.2 Hypermatrix representation of a sparse matrix

Sparse matrices are mostly composed of zeros but often have small dense blocks which have traditionally been exploited in order to improve performance [6]. Our approach uses a data structure based on a hypermatrix (HM) scheme [8, 19]. The matrix is partitioned recursively into blocks of different sizes. The HM structure consists of N levels of submatrices. 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. Null pointers in pointer matrices indicate that the corresponding submatrix does not have any nonzero elements and is therefore unnecessary. Figure 1 shows a sparse matrix and a simple example of corresponding hypermatrix with 2 levels of pointers.

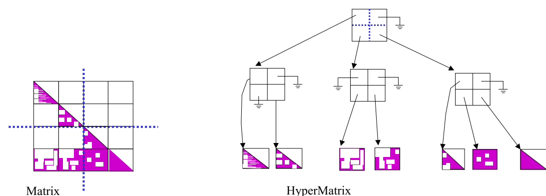

Figure 1: A sparse matrix and a corresponding hypermatrix.

The main potential advantages of a HM structure over 1D data structures, such as the Compact Row Wise structure, are: the ease of use of multilevel blocks to adapt the computation to the underlying memory hierarchy; the operation on dense matrices; and greater opportunities for exploiting parallelism. A commercial package known as PERMAS uses the hypermatrix structure [3]. It can solve very large systems out-of-core and can work in parallel. However, the disadvantages of the hypermatrix structure, namely the storage of and computation on zeros, introduce a large overhead. This problem can arise either when a fixed partitioning is used or when supernodes are amalgamated. In [2] the authors reported that a variable size blocking was introduced to save storage and to speed the parallel execution. In this way the HM was adapted to the sparse matrix being factored. The results presented in this paper, however, correspond to a static partitioning of the matrix into blocks of fixed sizes.

1.3 Machine and matrix characteristics

Execution took place 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.

We have used several test matrices. All of them are sparse matrices corresponding to linear programming problems. QAP matrices come from Netlib [17] while others come from a variety of linear multicommodity network flow generators: A Patient Distribution System (PDS) [5], with instances taken from [7]; RMFGEN [4]; GRIDGEN [14]; TRIPARTITE [9]. Table 1 shows the characteristics of several matrices obtained from such linear programming problems. Matrices were ordered with METIS [13] and renumbered by an elimination tree postorder.

Table 1: Matrix characteristics

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).

1.4 Previous work

Our previous work on sparse Cholesky factorization with the hypermatrix data structure was focused on the reduction of the overhead caused by the unnecessary operation on zeros.

We developed a set of routines which can operate very efficiently on small matrices. We create a library which we call the Small Matrix Library (SML) by fixing as many parameters as possible at compilation time [10]. Using rectangular data matrices we adapt the storage to the intrinsic structure of sparse matrices.

In [11] we showed that the use of windows within data submatrices and a 2D scheduling of the computations was necessary to improve performance. By keeping information about the actual space within a data submatrix which stores non-zeros (dense window) we can reduce both storage and computation. We define *windows* of non-zeros within data submatrices. Figure 2a 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. We also extended our SML library with routines which deal with matrices with windows. We keep the leading dimension fixed, while loop limits can be given as parameters. Some of these 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 for computation. Zeros can still exist within

the window but, in general, the overhead is greatly reduced. Routines where all parameters are fixed at compilation time are more efficient than those which use parameters passed at run time.

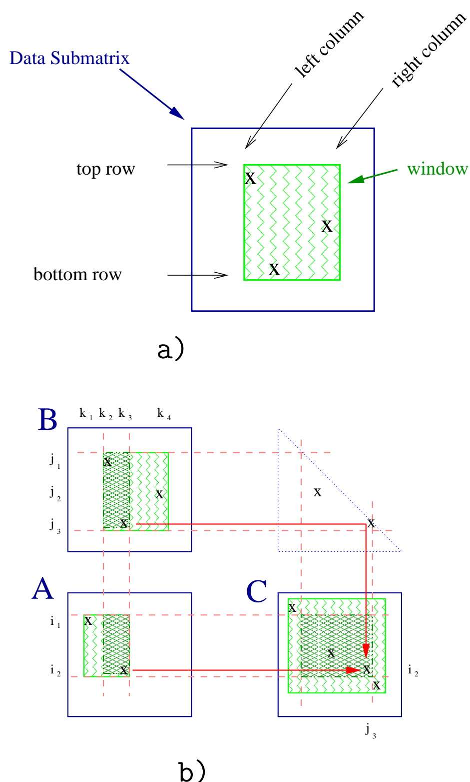

Figure 2: **a)** A data submatrix and a window within it. **b)** Windows can reduce the number of operations.

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 4 codes specialized in the multiplication of two matrices. Figure 3 shows graphically the data used by each routine. The most efficient is the one on the left. Their efficiency diminishes as we move to the right. The names and characteristics of each routine, from left to right, are: *mxmt_full* operates on the entire matrices, *mxmt_win_1dc* uses windows in columns; *mxmt_win_1dr* uses windows in rows; finally, *mxmt_win_2d* uses windows in both dimensions and is the slower code amongst all 4 codes.

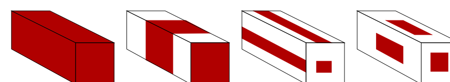

Figure 3: Four matrix multiplication routines deal with different input matrices.

2 Intra-Block Amalgamation

In this section we present an aspect of the work we have done to improve the performance of our sparse Hypermatrix Cholesky factorization.

The usage of windows reduces the number of unnecessary 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 [11]). 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.

Figure 4 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 *mxmt_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 5 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 introduces extra overhead due to the extra number of operations on such zeros. However, this can reduce the number of calls to routine *mxmt_win_2d*. Instead, some new

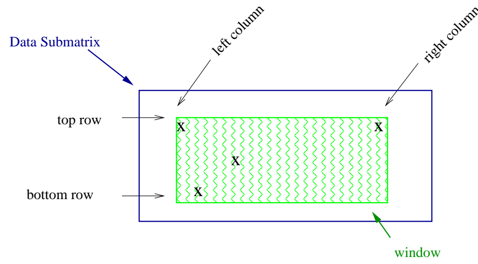

Figure 4: Original data submatrix before intra-block amalgamation.

calls to *mxmt_win_1dc* can be done. 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.

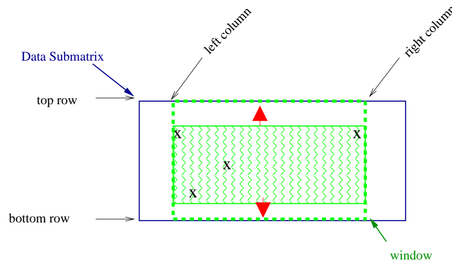

Figure 5: Data submatrix after row-wise intra-block amalgamation.

Figure 6 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 *mxmt_win_2d* is used. In this case, such calls could be replaced by calls to *mxmt_win_1dr*.

Finally, figure 7 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 *mxmt_win_2d* is used. In this case, such calls could be replaced by calls to any of the 3 other matrix multiplication routines (either *mxmt_full*, *mxmt_win_1dc* or *mxmt_win_1dr*) depending on the other matrix involved in the multiplication.

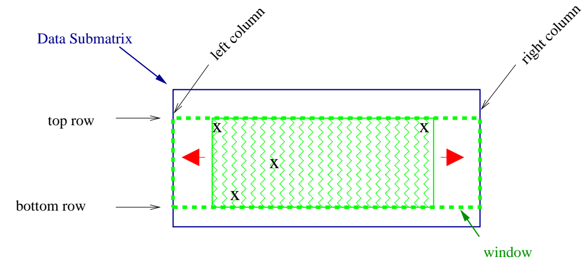

Figure 6: Data submatrix after column-wise intra-block amalgamation.

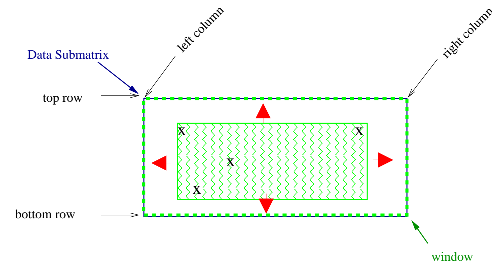

Figure 7: Data submatrix after applying both row and column-wise intra-block amalgamation.

3 Results

Given a 4×32 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 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. 8) , QAP12 (fig. 9), TRIPART1 (fig. 10), TRIPART2 (fig. 11), pds10 (fig. 12) and pds20 (fig. 13). Effective Mflops are presented.

They refer to the number of useful floating point operations performed per second. This metrics excludes useless operations on zeros performed by the HM Cholesky algorithm when data submatrices contain zeros.

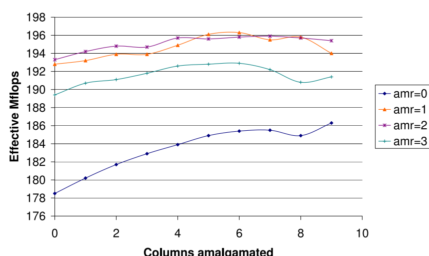

Figure 8: Intra-block amalgamation: matrix QAP8.

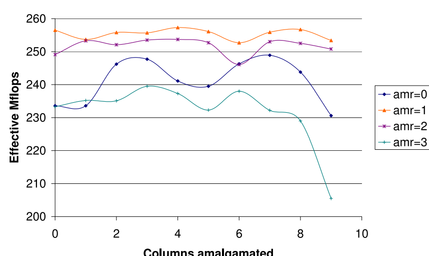

Figure 9: Intra-block amalgamation: matrix QAP12.

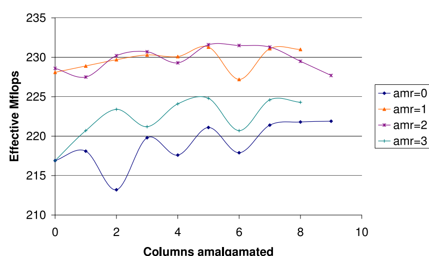

Figure 10: Intra-block amalgamation: matrix TRIPART1.

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.

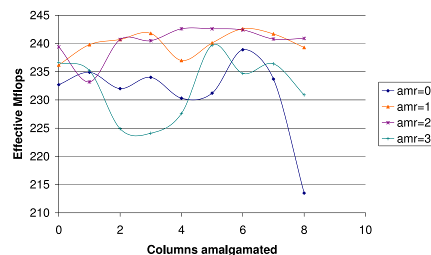

Figure 11: Intra-block amalgamation: matrix TRIPART2.

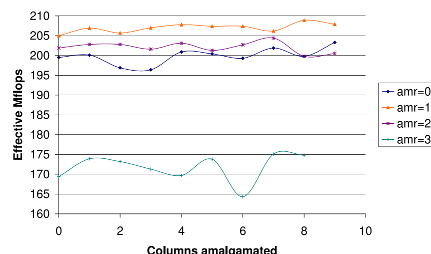

Figure 12: Intra-block amalgamation: matrix pds10.

Figure 14 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: 4×32 as the data submatrix size; 32×32 for the next pointer level, and 512×512 as the upper pointer level.

Finally, we present results obtained by five different sparse Cholesky factorization codes. Figure 15 shows the results obtained with each of them for the set of matrices

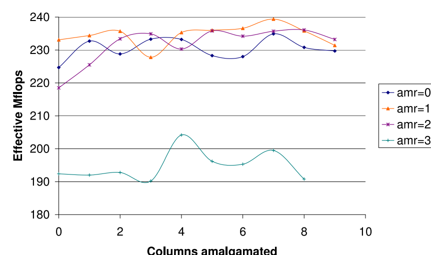

Figure 13: Intra-block amalgamation: matrix pds20.

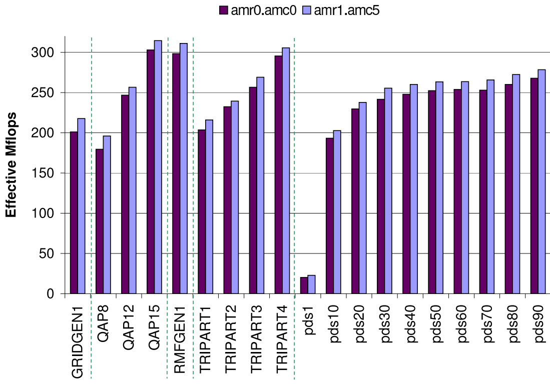

Figure 14: Performance of sparse HM Cholesky without and with intra-block amalgamation.

introduced above. Matrix families are separated by dashed lines.

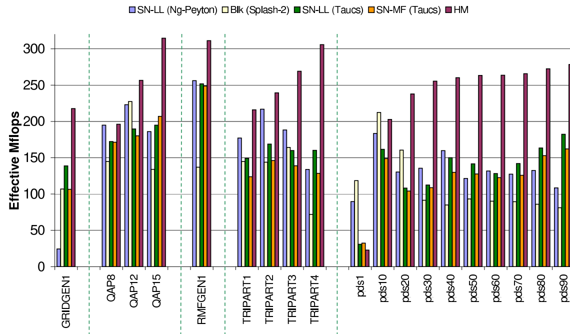

Figure 15: Performance of several sparse Cholesky factorization codes.

The first bar corresponds to a supernodal left-looking block Cholesky factorization (SN-LL (Ng-Peyton)) [18]. The second bar shows the performance obtained by the sequential version of a 2D block-oriented approach [21] as found in the SPLASH-2 [22] suite. Although submatrices are kept in a two-dimensional data layout 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) [12]. 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 two 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 [11] within data submatrices and SML [10] routines to improve our sparse matrix application based on hypermatrices. Values 1 for row-wise and 5 for column-wise amalgamation have been used. A fixed partitioning of the matrix has been used. We present results obtained for data submatrix sizes 4×32 and upper hypermatrix levels with sizes 32×32 and 512×512.

4 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, and by columns with a value of 5, 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.

References

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

[2] M. Ast, C. Barrado, J.M. Cela, R. Fischer, O. Laborda, H. Manz, and U. Schulz. Sparse matrix structure for dynamic parallelisation efficiency. In *Euro-Par 2000,LNCS1900*, pages 519–526, September 2000.

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

[4] Tamas Badics. RMFGEN generator., 1991. Code available from ftp://dimacs.rutgers.edu in directory pub/netflow/generators/network/genrmf.

[5] 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.

[6] 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.

[7] A. Frangioni. Multicommodity Min Cost Flow problems. Data available from www.di.unipi.it/di/groups/optimize/Data.

[8] 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.

[9] A. Goldberg, J. Oldham, S. Plotkin, and C. Stein. An implementation of a combinatorial approximation algorithm for minimum-cost multicommodity flow. In *IPCO*, 1998.

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

[11] J. R. Herrero and J. J. Navarro. Optimization of a statically partitioned hypermatrix sparse cholesky factorization. In *PARA'04 Workshop on state-of-the-art in scientific computing, LNCS-TBA*. Springer-Verlag, June 2004.

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

[13] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR95-035, Department of Computer Science, University of Minnesota, October 1995.

[14] Y. Lee and J. Orlin. GRIDGEN generator., 1991. Code available from ftp://dimacs.rutgers.edu in directory pub/netflow/generators/network/gridgen.

[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] 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.

[17] NetLib. Linear programming problems. http://www.netlib.org/lp/.

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

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

[20] 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.

[21] 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.

[22] 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 Int. Symp. on Computer architecture*, pages 24–36. ACM Press, 1995.