Optimization of a Statically Partitioned Hypermatrix Sparse Cholesky Factorization [⋆]

José R. Herrero[1] and Juan J. Navarro[1]

Computer Architecture Department, Universitat Politècnica de Catalunya,
Jordi Girona 1-3, Mòdul D6, E-08034 Barcelona, (Spain)
{josepr,juanjo}@ac.upc.es

Abstract. The sparse Cholesky factorization of some large matrices can require a two dimensional partitioning of the matrix. The sparse hypermatrix storage scheme produces a recursive 2D partitioning of a sparse matrix. The subblocks are stored as dense matrices so BLAS3 routines can be used. However, since we are dealing with sparse matrices some zeros may be stored in those dense blocks. The overhead introduced by the operations on zeros can become large and considerably degrade performance. 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 compare its performance with several codes and analyze the results.

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 [8, 20, 22]. 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 [22]. Operations are thus performed on blocks (submatrices). In this paper we address the optimization of the sparse Cholesky factorization of large matrices. For this purpose, we use a *Hypermatrix* [10] block data structure with static block sizes.

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* [17] is used. Columns having similar structure are taken as a group. These column groups are called *supernodes* [18]. Some supernodes may be too large to fit in cache and

[⋆] This work was supported by the Ministerio de Ciencia y Tecnología of Spain and the EU FEDER funds (TIC2001-0995-C02-01)

it is advisable to split them into *panels* [20, 22]. 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 [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 [8]. Our approach uses a data structure based on a hypermatrix (HM) scheme [10, 21]. 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 non-zero elements and is therefore unnecessary. Figure 1 shows a sparse matrix and a simple example of corresponding hypermatrix with 2 levels of pointers.

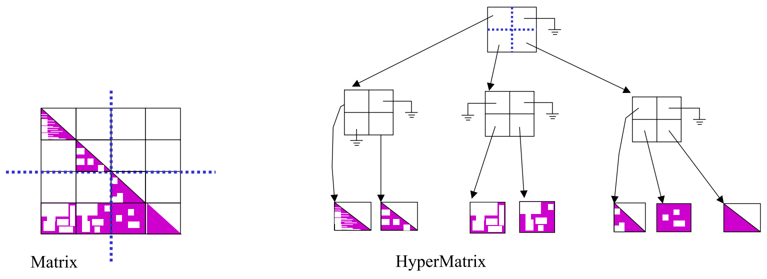

Fig. 1. A sparse matrix and a corresponding hypermatrix.

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 [19] while others come from a variety of linear multicommodity network flow generators: A Patient Distribution System (PDS) [5], with instances taken from [9]; RMFGEN [4]; GRIDGEN [16]; TRIPARTITE [11]. Table 1 shows the characteristics of several matrices obtained from such linear programming problems. Matrices were ordered with METIS [14] and renumbered by an elimination tree postorder.

Table 1. Matrix characteristics

Matrix	Dimension	NZs	NZs in L	Density	Flops to factor
GRIDGEN1	330430	3162757	130586943	0.002	278891960665
QAP8	912	14864	193228	0.463	63764878
QAP12	3192	77784	2091706	0.410	2228094940
QAP15	6330	192405	8755465	0.436	20454297601
RMFGEN1	28077	151557	6469394	0.016	6323333044
TRIPART1	4238	80846	1147857	0.127	511884159
TRIPART2	19781	400229	5917820	0.030	2926231696
TRIPART3	38881	973881	17806642	0.023	14058214618
TRIPART4	56869	2407504	76805463	0.047	187168204525
pds1	1561	12165	37339	0.030	1850179
pds10	18612	148038	3384640	0.019	2519913926
pds20	38726	319041	10739539	0.014	13128744819
pds30	57193	463732	18216426	0.011	26262856180
pds40	76771	629851	27672127	0.009	43807548523
pds50	95936	791087	36321636	0.007	61180807800
pds60	115312	956906	46377926	0.006	81447389930
pds70	133326	1100254	54795729	0.006	100023696013
pds80	149558	1216223	64148298	0.005	125002360050
pds90	164944	1320298	70140993	0.005	138765323993

2 Performance Issues

In this section we present two aspects of the work we have done to improve the performance of our sparse Hypermatrix Cholesky factorization. Both of them 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 of arbitrary size. We call M_{br_i,bc_j} the data submatrix in block-row br_i and block-column bc_j. Figure 2 shows 3 submatrices within a matrix. The highest cost within the Cholesky factorization process comes from the multiplication of data submatrices. In order to ease the explanation we will refer to the three matrices 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.

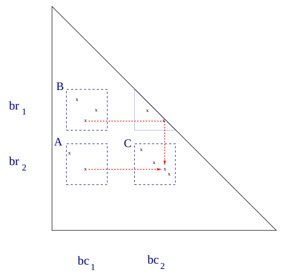

Fig. 2. Blocks within a matrix.

2.1 Efficient operation on small matrices

Choosing a block size for data submatrices is rather difficult. When operating on dense matrices, it is better to use large block sizes. On the other hand, the larger the block is, the more likely it is to contain zeros. Since computation with zeros is non productive, performance can be degraded. Thus, a trade-off between performance on dense matrices and operations on non-zeros must be reached. In a previous paper [12], we explained how we could reduce the block size while we improved performance. This was achieved by the use of a specialized set of routines which operate on small matrices of fixed size. By small matrices we mean matrices which fit in the first level cache. The basic idea used in producing this set of routines, which we call the Small Matrix Library (SML), is that of having dimensions and loop limits fixed at compilation time. For example, our matrix multiplication routines *mxmts_fix* clearly outperform the vendor's BLAS dgemm routine (*dgemm_nts*) for small matrices (figure 3a) on an R10000 processor. We achieved similar results to outperform the ATLAS [23] dgemm routine.

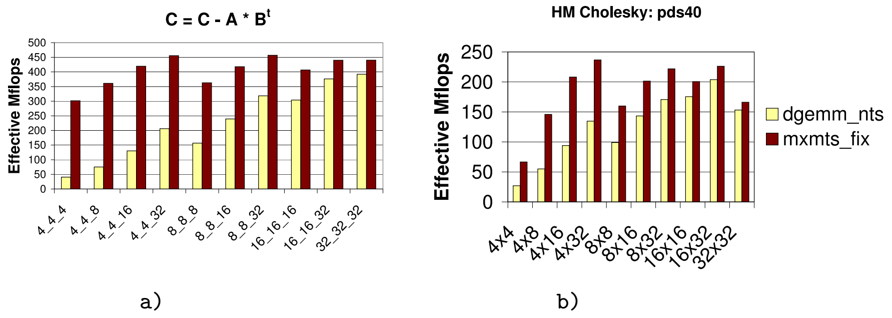

Fig. 3. a) Performance of different MxM^t routines for several matrix sizes. **b)** Factorization of matrix pds40: Mflops obtained by different MxM^t codes within HM Cholesky. Effective Mflops are reckoned excluding any operations on zeros.

The matrix multiplication routine used affects the performance of hypermatrix Cholesky factorization. This operation takes up most of the factorization time. We found that when using *mxmts_fix*, a block size of 4×32 usually produced the best performance. In order to illustrate this, figure 3b shows results of the HM Cholesky factorization on an R10000 for matrix pds40 [5]. The use of a fixed dimension matrix multiplication routine speeded up our Cholesky factorization an average of 12% for our test matrix set (table 1).

2.2 Reducing overhead: windows within data submatrices

Let A and B be two off-diagonal submatrices in the same block-column of a matrix. At first glance, these matrices should always be multiplied since they belong to the same block-column. However, there are cases where it is not necessary. We are storing data submatrices as dense while the actual contents of the submatrix are not necessarily dense. Thus, the result of the product $A \times B^t$ can be zero. Such an operation will produce an update into some matrix C whenever there exists at least one column for which both matrices A and B have any non-zero element. Otherwise, if there are no such columns, the result will be zero. Consequently, that multiplication can be bypassed.

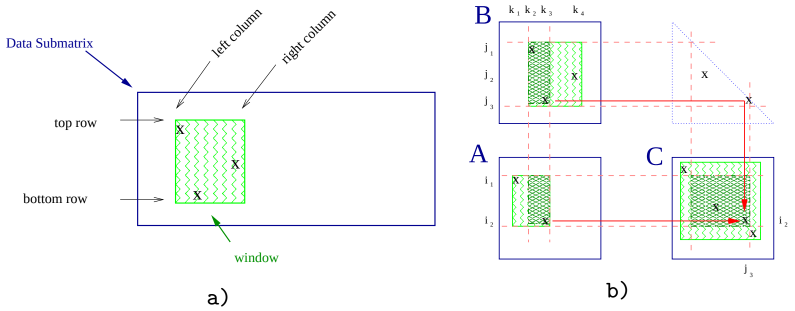

Fig. 4. a) A rectangular data submatrix and a window within it. **b)** Windows can reduce the number of operations.

In order to reduce the storage and computation of zero values, we define *windows* of non-zeros within data submatrices. Figure 4a 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 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 for 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 hypermatrix 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, it is necessary to compute only a subset of the total block (dark areas within figure 4b). When the column-wise intersection of windows in matrices A and B is null, we can avoid the multiplication of these two matrices.

3 Results

Figure 5 shows the results of using windows on our sparse Hypermatrix Cholesky for a 4×32 data matrix size. We conclude that the usage of such windows clearly improves the performance of our sparse hypermatrix Cholesky algorithm.

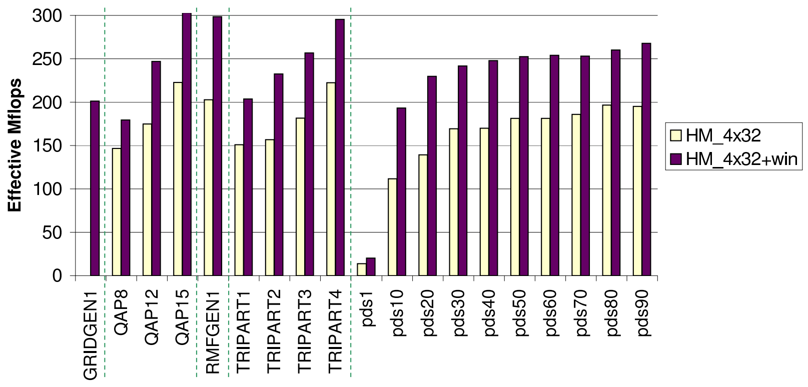

Fig. 5. HM performance with and without windows for several input matrices.

Next, we present results obtained by four different sparse Cholesky factorization codes. Figure 6 shows the best result obtained with each of them for the set of matrices introduced above. Matrix families are separated by dashed lines.

The first bar corresponds to a supernodal left-looking block Cholesky factorization (SN-LL (Ng-Peyton)) [20]. 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 [24], PCx [7],

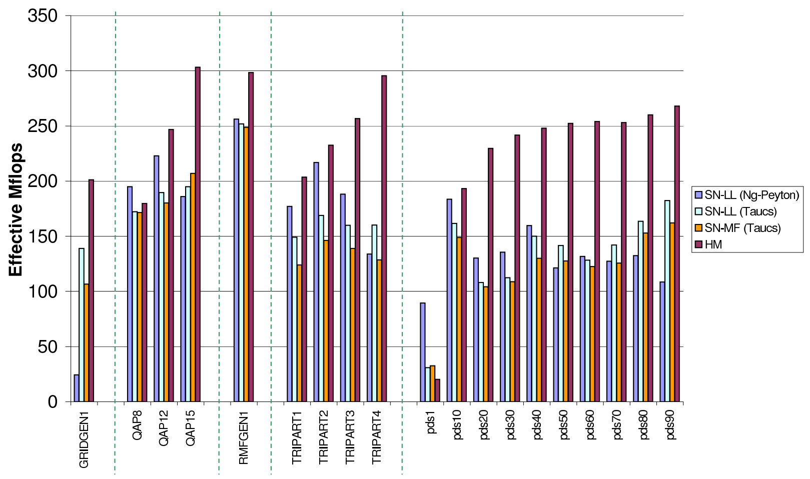

Fig. 6. Performance of several sparse Cholesky factorization codes.

IPM [6] or SparseM [15]. 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. We noticed that, in some cases, we could improve results by specifying cache sizes multiple of the actual first level cache. However, performance degrades in all cases for large matrices.

The second and third 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) [13]. In these codes the matrix is represented by an array of pointers to blocks (submatrices). Consequently, matrices are stored by block, not by column. Every block is stored contiguously in memory. The ordering of blocks in memory is based on a recursive partitioning of the matrix. The algorithms use a recursive schedule, so the schedule and the data layout match each other. Thus, the memory hierarchy is automatically exploited. The performance obtained by these codes is quite uniform.

Finally, the fourth bar shows the performance obtained by our sparse hypermatrix Cholesky code (HM). We have used SML [12] routines to improve our sparse matrix application based on hypermatrices. 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. We used windows within data submatrices.

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 many zeros. Figure 7 shows the percentage increase in number of flops in sparse HM Cholesky 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. Performance of HM Cholesky is then better than that obtained by the other algorithms tested. Note that the hypermatrix Cholesky factorization usually improves its performance as the problem size gets larger. There are two reasons for this. One is 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. The other is that since blocks tend to be larger, more operations are performed by efficient $M \times M^t$ routines.

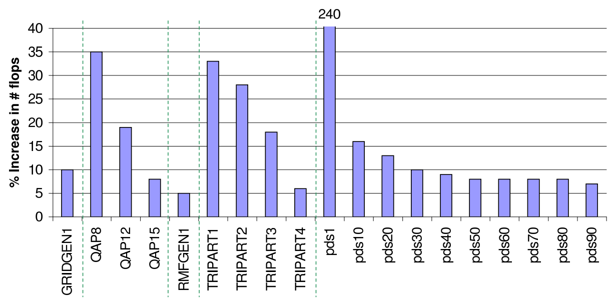

Fig. 7. Increase in number of operations in sparse HM Cholesky (4x32 + windows).

Figure 8 shows the percentage of multiplication operations performed by each of the four $M \times M^t$ routines we use. We focus on matrix multiplication because it is, by far, the most expensive operation within the Cholesky factorization.

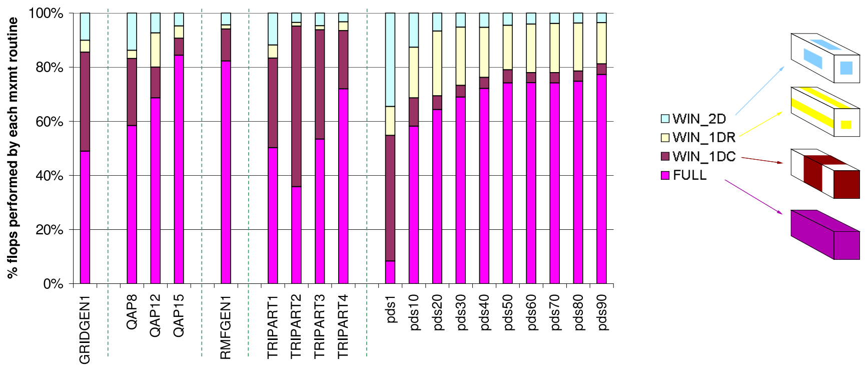

Fig. 8. HM flops per MxMt subroutine type.

FULL refers to the routine where all matrix dimensions are fixed. This is the most efficient of the four routines. 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. 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 because of the constant leading dimension used for all three matrices. 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 largest matrix 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 $M \times M^t$ routine less than 50% of the time. In addition, the least efficient routine WIN_2D is used to compute about 10% of the operations.

4 Future work

The results presented here have been obtained by splitting the matrix into blocks of fixed size for each hypermatrix level. However, we can also create the hypermatrix structure using the information from the supernodal elimination tree. The results for low amalgamation values are very poor and have not been presented. However, we expect an improvement in the performance of our sparse hypermatrix Cholesky code with different (larger) amalgamation values. In the near future we will experiment with different amalgamation values and algorithms.

We would also like to improve our code by avoiding the operation on very small matrices. We are currently studying the way to extend our code so that it can work with supernodes in addition to dense matrices in the last level of the hypermatrix (data submatrix level).

Since our approach seems promising for large matrices we want to extend it to work out-of-core. We think that the hypermatrix scheme also provides a good platform for exploiting the higher levels of the memory hierarchy.

Finally, we plan to use our code as the basis for a parallel implementation of the sparse Cholesky factorization.

5 Conclusions

A two dimensional partitioning of the matrix is necessary for large sparse matrices. 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, the performance of our sparse hypermatrix Cholesky can be up to an order of magnitude better than that of a 1D supernodal block Cholesky (which tries to use the cache memory properly by splitting supernodes into panels). It also outperforms sequential supernodal left-looking and supernodal multifrontal routines based on a 2D storage of the matrix being factored. Using windows and SML routines our HM Cholesky often gets over half of the processor's peak performance for medium and large sparse matrices factored sequentially in-core.

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/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. Jordi Castro. A specialized interior-point algorithm for multicommodity network flows. *SIAM Journal on Optimization*, 10(3):852–877, September 2000.
7. J. Czyzyk, S. Mehrotra, M. Wagner, and S. J. Wright. PCx User's Guide (Version 1.1). Technical Report OTC 96/01, Evanston, IL 60208–3119, USA, 1997.
8. 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.
9. A. Frangioni. Multicommodity Min Cost Flow problems. Data available from http://www.di.unipi.it/di/groups/optimize/Data/.
10. 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.
11. A. Goldberg, J. Oldham, S. Plotkin, and C. Stein. An implementation of a combinatorial approximation algorithm for minimum-cost multicommodity flow. In *IPCO*, 1998.
12. José R. Herrero and Juan J. Navarro. Improving Performance of Hypermatrix Cholesky Factorization. In *Euro-Par 2003,LNCS2790*, pages 461–469. Springer-Verlag, August 2003.
13. D. Irony, G. Shklarski, and S. Toledo. Parallel and fully recursive multifrontal sparse cholesky. In *ICCS 2002,LNCS2330*, pages 335–344. Springer, April 2002.
14. 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.
15. Roger Koenker and Pin Ng. SparseM: A Sparse Matrix Package for R, 2003. http://cran.r-project.org/src/contrib/PACKAGES.html#SparseM.
16. Y. Lee and J. Orlin. GRIDGEN generator., 1991. Code available from ftp://dimacs.rutgers.edu/pub/netflow/generators/network/gridgen.
17. J. H. W. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1990.
18. 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.
19. NetLib. Linear programming problems. http://www.netlib.org/lp/.
20. Esmond G. Ng and Barry W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.*, 14(5):1034–1056, 1993.
21. A. Noor and S. Voigt. Hypermatrix scheme for the STAR–100 computer. *cas*, 5:287–296, 1975.
22. 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.
23. R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98*, pages 211–217. IEEE Computer Society, Nov 1998.
24. Y. Zhang. Solving large–scale linear programs by interior–point methods under the MATLAB environment. Technical Report 96–01, Baltimore, MD 21228–5398, USA, 1996.