REDUCING OVERHEAD IN SPARSE HYPERMATRIX CHOLESKY FACTORIZATION

Jose R. Herrero and Juan J. Navarro
Computer Architecture Department, Universitat Politecnica de Catalunya [*]
Jordi Girona 1-3, Modul D6, E-08034 Barcelona, Spain
{josepr,juanjo}@ac.upc.es

Abstract

The sparse hypermatrix storage scheme produces a recursive 2D partitioning of a sparse matrix. Data subblocks are stored as dense matrices. Since we are dealing with sparse matrices some zeros can be stored in those dense blocks. The overhead introduced by the operations on zeros can become really large and considerably degrade performance. In this paper, we present several techniques for reducing the operations on zeros in a sparse hypermatrix Cholesky factorization. By associating a bit to each column within a data submatrix we create a bit vector. We can avoid computations when the bitwise AND of their bit vectors is null. By keeping information about the actual space within a data submatrix which stores non-zeros (dense window) we can reduce both storage and computation.

Keywords: Sparse Hypermatrix Cholesky, bit vector, dense window

1. Introduction

Sparse Cholesky factorization is heavily used in several application domains, including finite-element and linear programming methods. It often takes a large part of the overall computation time incurred by those methods. Consequently, there has been great interest in improving its performance (Duff, 1982; Ng and Peyton, 1993; Rothberg, 1996). 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 (Rothberg, 1996). Operations are thus performed on blocks (submatrices). A matrix M is divided into submatrices of arbitrary size. We call

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

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

M_{br_i,bc_j} the data submatrix in block-row br_i and block-column bc_j. Figure 1 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.

Block size 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* (Liu, 1990) is used. Columns having similar structure are taken as a group. These column groups are called *supernodes* (Liu et al., 1993). Some supernodes may be too large to fit in cache and it is advisable to split them into *panels* (Ng and Peyton, 1993; Rothberg and Gupta, 1991). In other cases, supernodes can be too small to yield good performance. This is the case of supernodes with only one or a few columns. Level 1 BLAS routines are used in this case and the performance obtained is therefore poor. This problem can be lightened by *amalgamating* several supernodes into a single larger one (Ashcraft and Grimes, 1989). Then, some null elements are both stored and used for computation. However, the use of level 3 BLAS routines often results in some performance improvement.

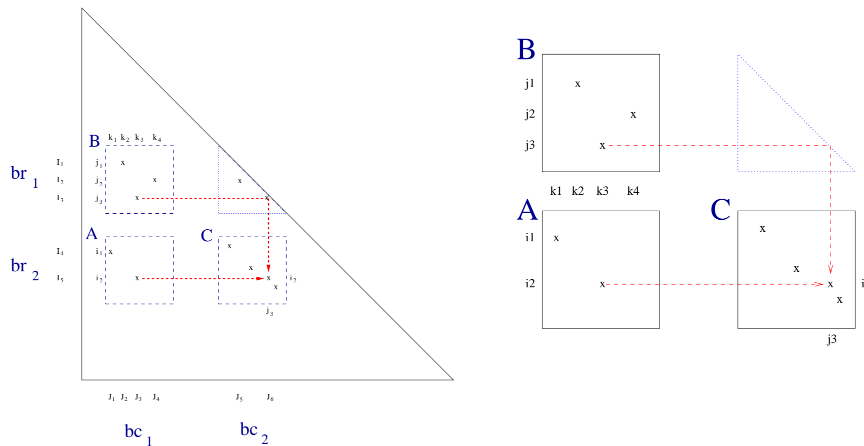

Figure 1. Blocks within a matrix: definition and example of use.

In this paper we address the problem of dealing with zeros within submatrices. This problem can arise either when a fixed partitioning is used or when supernodes are amalgamated. We present several techniques we have used to

reduce the overhead introduced by the fact that data submatrices are stored and operated on as dense. This is interesting in the context where we are focused: the optimization of the *Hypermatrix* (Fuchs et al., 1972) data structure.

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 (Duff, 1982). Our application uses a data structure based on a hypermatrix (HM) scheme (Fuchs et al., 1972; Noor and Voigt, 1975). 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 2 shows a sparse matrix and a simple example of corresponding hypermatrix with 2 levels of pointers.

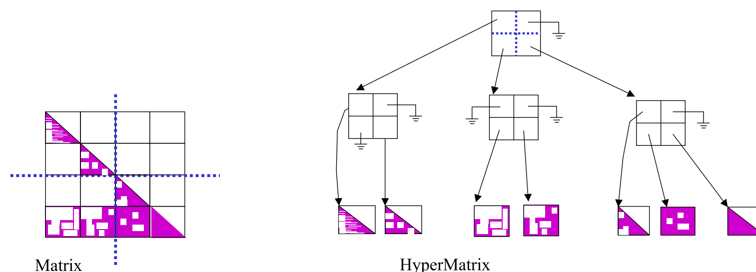

Figure 2. A sparse matrix and a corresponding hypermatrix.

The main potential advantages of a HM structure w.r.t. 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 (Ast et al., 1997). It can solve very large systems out-of-core and can work in parallel. However, the disadvantages of the hypermatrix structure, namely the storage and computation on zeros, introduce a large overhead. Recently a variable size blocking was introduced to save storage and to speed the parallel execution (Ast et al., 2000). In this way the HM was adapted to the sparse matrix being factored.

Previous work

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 operation on non-zeros must be reached. In a previous paper (Herrero and Navarro, 2003), 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 routine *dgemm_nts* for small matrices (figure 3a) on an R10000 processor.

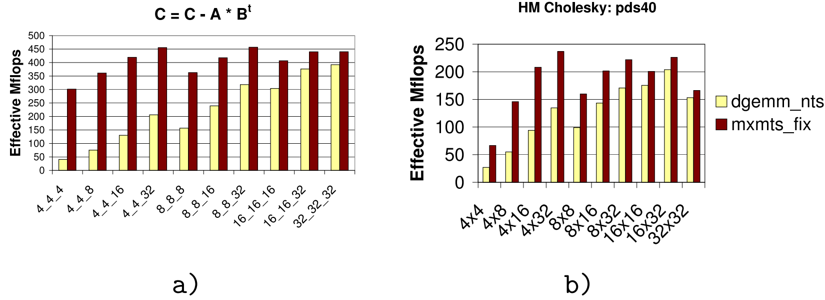

Figure 3. **a)** Performance of different MxM^t routines for several matrix sizes. **b)** Factorization of matrix pds40: Mflops obtained by different MxM^t codes within HM Cholesky. Effective Mflops are reckoned excluding any operations on zeros.

The matrix multiplication routine used affects the performance of hypermatrix Cholesky factorization. This operation takes up most of the factorization time. We found that 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 (Carolan et al., 1990). 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).

Goals

The work presented in this paper focuses on the reduction of the overhead introduced by operations on zeros kept in data submatrices. In addition to the techniques mentioned above on block size reduction and the use of specialized routines, we want to reduce the amount of operations on zeros within blocks.

2. Reducing overhead

Let A and B be two off-diagonal submatrices in the same block-column. 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 do not necessarily have to be 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 (e.g. column k_3 in figure 1). Otherwise, if there are no such columns, the result will be zero. Consequently, that multiplication can be skipped. In the following subsections we present several techniques which can reduce the number of non productive operations.

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*. Each bit in the vector is used to point to the existence of any non-zero in the corresponding column. For instance, consider matrix B in figure 4a. 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. If all bits are zero, then we can skip it. 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.

Dense windows within data submatrices

In order to reduce the storage and computation of zero values, we define *windows* of non-zeros within data submatrices. Figure 5a 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

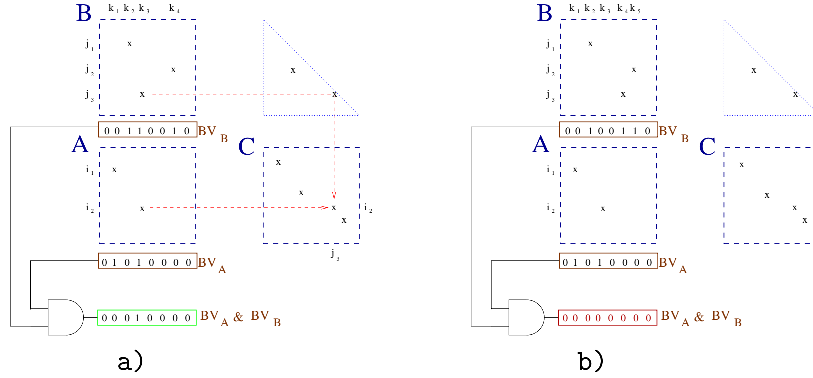

Figure 4. **a)** $BV_A \& BV_B \neq 0$: operation must be performed. **b)** $BV_A \& BV_B = 0$: operation can be avoided.

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 leading dimensions of matrices fixed, while loop limits can be given as parameters. Some of them 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 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 a subset of the total block is computed (dark areas within figure 5b).

When the column-wise intersection of windows in matrices A and B is null, we can avoid the multiplication of these two matrices (figure 6a). There are cases where the window definition we have used is not enough to avoid unnecessary operations. Consider figure 6b: 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]

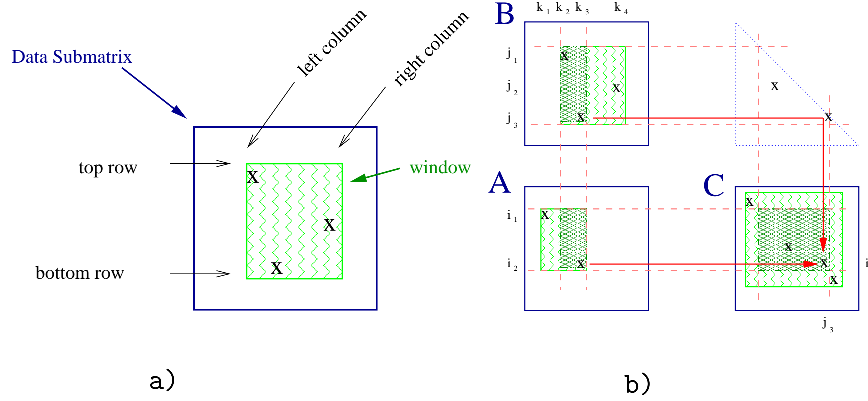

Figure 5. **a)** A data submatrix and a window within it. **b)** Windows can reduce the number of operations.

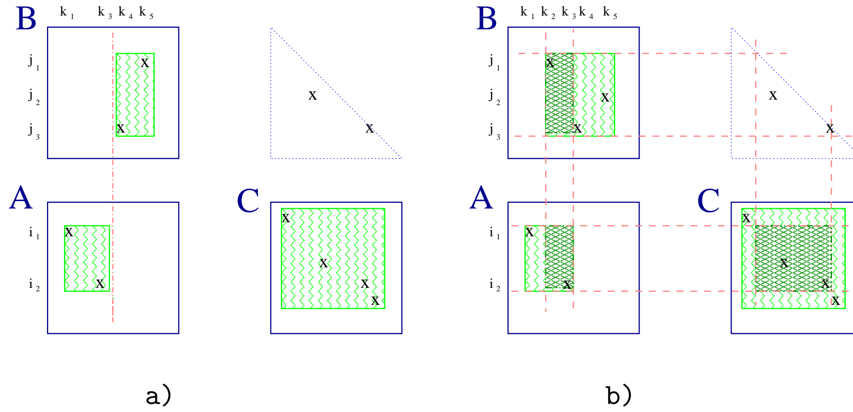

Figure 6. **a)** Disjoint windows can avoid matrix products. **b)** Windows can be ineffective to detect false intersections.

3. Results

We have used several test matrices. All of them are sparse matrices corresponding to linear programming problems. QAP matrices come from Netlib (NetLib,) while others come from a variety of linear multicommodity network flow generators: A Patient Distribution System (PDS) (Carolan et al., 1990), with instances taken from (Frangioni,); RMFGEN (Badics, 1991); GRIDGEN (Lee and Orlin, 1991); TRIPARTITE (Goldberg et al., 1998). Table 1 shows the characteristics of several matrices obtained from such linear programming problems. Matrices were ordered with METIS (Karypis and Kumar, 1995) and renumbered by an elimination tree postorder. 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.

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).

The left half of table 2 presents results obtained by a supernodal (SN) block Cholesky factorization (Ng and Peyton, 1993). 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 (Zhang, 1996), PCx (Czyzyk et al., 1997), IPM (Castro, 2000) or SparseM (Koenker and Ng, 2003). 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 we could improve its results by specifying cache sizes larger than the actual first level cache. However, performance degrades in all cases for large matrices.

The right half of table 2 shows results obtained by several variants of our sparse hypermatrix Cholesky code. We have used SML (Herrero and Navarro,

Table 2. Supernodal vs Hypermatrix Cholesky: Mflops

	Supernodal Cholesky (Ng-Peyton)								Hypermatrix Cholesky					
Upper levels									32 x 512					
Block size									8 x 8		4 x 32			
Windows									No		No		Yes	
Bit Vectors									No	Yes	No	Yes	No	Yes
Cache	32K		512K		1M		2M							
Unrolling	4	8	4	8	4	8	4	8						
GRIDGEN1						23.8		24.4	—	—	—	—	201.2	199.5
QAP8	186.6	194.2	186.7	194.9		175.1		177.3	139.0	142.5	146.6	151.5	179.6	180.2
QAP12	118.8	102.2	181.0	223.0		215.7		166.3	160.5	176.0	174.7	197.5	246.8	247.3
QAP15	49.0	54.8	152.4	186.0		165.6		149.1	213.1	214.4	222.7	248.4	303.1	300.2
RMFGEN1	55.9	61.9	169.8	189.0		256.2		154.9	221.0	220.8	202.8	210.7	298.4	300.9
TRIPART1	118.7	176.4	175.6	177.1		160.5		164.5	151.2	170.7	151.0	152.8	203.6	207.1
TRIPART2	205.2	182.5	208.4	213.1		216.9		171.8	175.5	202.5	156.8	178.3	232.5	235.3
TRIPART3		116.9		142.7		188.2		151.3	199.0	213.1	181.7	185.3	256.6	261.1
TRIPART4		46.4		119.3	121.1	133.8		118.7	222.8	231.5	222.5	241.4	295.5	295.2
pds1	87.5	89.6		75.7		73.8			19.0	20.2	13.7	14.3	20.2	20.2
pds10	121.5	125.5		183.5		132.2			102.4	106.5	111.6	121.3	193.3	192.3
pds20	106.8	82.7		130.3		104.2			126.1	127.1	139.3	149.9	229.7	227.6
pds30	104.0	78.2		133.5		135.5			142.9	141.5	169.3	178.0	241.7	241.1
pds40	99.9	97.3		126.4		159.8			144.5	144.6	169.8	176.8	247.9	242.1
pds50	84.8	92.0		121.1		121.4			140.2	147.5	181.3	191.4	252.4	252.2
pds60	85.0	66.0		131.8		112.0			152.8	153.4	181.4	188.1	253.9	254.5
pds70		91.4		127.4		111.3			154.6	154.8	186.0	194.4	253.0	252.4
pds80		62.3		132.5		107.2			154.1	164.4	196.6	198.0	260.1	259.5
pds90				108.5		105.1			166.3	169.4	195.2	195.4	267.9	265.7

2003) routines to improve our sparse matrix application based on hypermatrices. A fixed partitioning of the matrix has been done to be able to test the impact of each overhead reduction technique used. We present results obtained with and without bit vectors for two data submatrix sizes: 8×8 and 4×32. For the latter we also introduce the usage of windows.

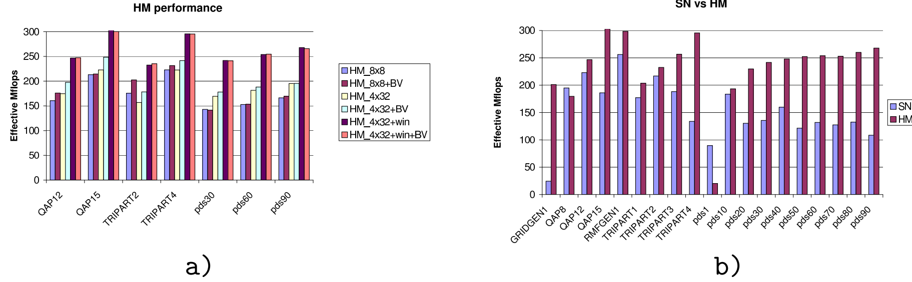

Figure 7. **a)** HM performance for several input matrices. **b)** SN vs HM performance.

Figure 7a summarizes these results. 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. Figure 7b 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.

Finally, figure 8 shows performance of each algorithm on several matrix families. Note that, contrary to the supernodal algorithm behavior, the hypermatrix Cholesky factorization improves its performance as the problem size gets larger.

4. Conclusions

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 prop-

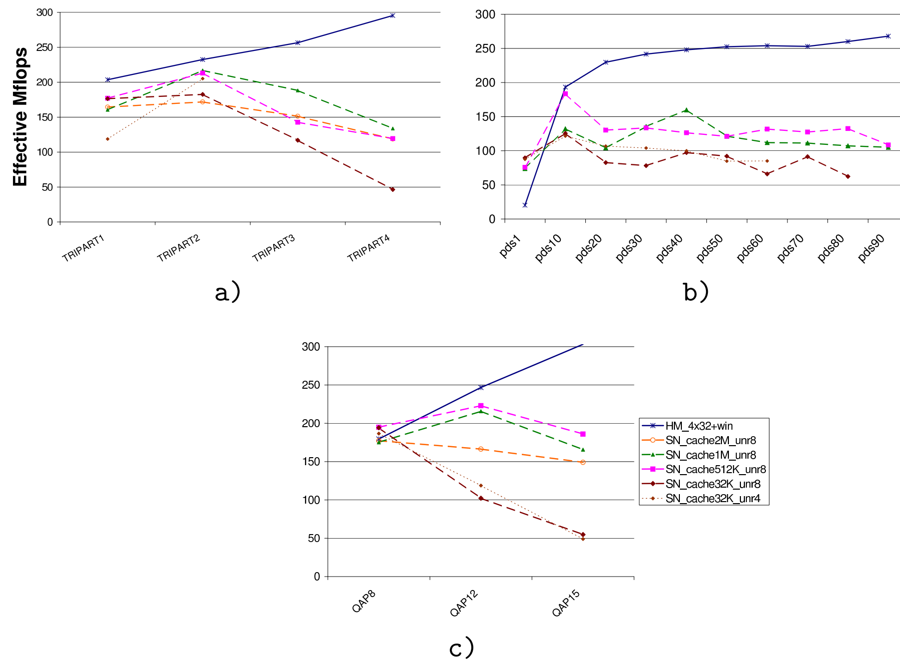

Figure 8. SN vs HM Cholesky for 3 matrix families: **a)** Tripart; **b)** PDS; **c)** QAP.

erly 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.

Notes

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.

References

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

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

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

Badics, Tamas (1991). RMFGEN generator. Code available from ftp://dimacs.rutgers.edu/pub/netflow/generators/network/genrmf.

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

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

Czyzyk, J., Mehrotra, S., Wagner, M., and Wright, S. J. (1997). PCx User's Guide (Version 1.1). Technical Report OTC 96/01, Evanston, IL 60208–3119, USA.

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

Frangioni, A. Multicommodity Min Cost Flow problems. Data available from http://www.di.unipi.it/di/groups/optimize/Data/.

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

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

Herrero, José R. and Navarro, Juan J. (2003). Improving Performance of Hypermatrix Cholesky Factorization. In *Euro-Par 2003,LNCS2790*, pages 461–469. Springer-Verlag.

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

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

Lee, Y. and Orlin, J. (1991). GRIDGEN generator. Code available from ftp://dimacs.rutgers.edu/pub/netflow/generators/network/gridgen.

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

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

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

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

Noor, A. and Voigt, S. (1975). Hypermatrix scheme for the STAR–100 computer. *cas*, 5:287–296.

Rothberg, Edward (1996). 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.

Rothberg, Edward and Gupta, Anoop (1991). Efficient sparse matrix factorization on high-performance workstations: Exploiting the memory hierarchy. *ACM Trans. Math. Soft.*, 17(3):313–334.

Zhang, Y. (1996). Solving large–scale linear programs by interior–point methods under the MATLAB environment. Technical Report 96–01, Baltimore, MD 21228–5398, USA.