Efficient Methods for Out-of-Core Sparse Cholesky Factorization

Edward Rothberg*, Robert Schreiber
Compiler and Architecture Research

HPL-98-114
June, 1998

sparse matrix, memory hierarchy, Cholesky factorization

We consider the problems of sparse Cholesky factorization with limited main memory. The goal is to efficiently factor matrices whose Cholesky factors essentially fill the available disk storage, using very little memory (as little as 16 Mbytes). This would enable very large industrial problems to be solved with workstations of very modest cost.

We consider three candidate algorithms. Each is based on a partitioning of the matrix into panels. The first is a robust, out-of-core multifrontal method that keeps the factor, the stack, and the large frontal matrices on disk. The others are left-looking methods. We find that straightforward implementations of all of them suffer from excessive disk I/O for large problems that arise in interior-point algorithms for linear programming. We introduce several improvements to these simple out-of-core methods, and find that a left-looking method that nevertheless uses the multifrontal algorithm for portions of the matrix (subtrees of the supernodal elimination tree whose multifrontal stack fits in memory) is very effective. With 32 Mbytes of main memory, it achieves over 77 percent of its in-core performance on all but one of our twelve test matrices (67 percent in that one case), even though the size of the factor is, in all cases, hundreds of millions or even billions of bytes.

*ILOG, Inc., Mountain View, California

Efficient Methods for Out-of-Core Sparse Cholesky Factorization

Edward Rothberg* Robert Schreiber†

May 29, 1997; revised February 24, 1998

Abstract

We consider the problem of sparse Cholesky factorization with limited main memory. The goal is to efficiently factor matrices whose Cholesky factors essentially fill the available disk storage, using very little memory (as little as 16 Mbytes). This would enable very large industrial problems to be solved with workstations of very modest cost.

We consider three candidate algorithms. Each is based on a partitioning of the matrix into panels. The first is a robust, out-of-core multifrontal method that keeps the factor, the stack, and the large frontal matrices on disk. The others are left-looking methods. We find that straightforward implementations of all of them suffer from excessive disk I/O for large problems that arise in interior-point algorithms for linear programming. We introduce several improvements to these simple out-of-core methods, and find that a left-looking method that nevertheless uses the multifrontal algorithm for portions of the matrix (subtrees of the supernodal elimination tree whose multifrontal stack fits in memory) is very effective. With 32 Mbytes of main memory, it achieves over 77 percent of its in-core performance on all but one of our twelve test matrices (67 percent in that one case), even though the size of the factor is, in all cases, hundreds of millions or even billions of bytes.

1 Introduction

Due to recent trends in microprocessor design, including improved integrated-circuit fabrication techniques and the introduction of hardware for the detection and exploitation of instruction-level parallelism, inexpensive microprocessors now offer peak performance levels that were only available on vector supercomputers just a few years ago. Of course, near-peak performance can only be obtained when algorithms make effective use of the multi-level memory hierarchies (registers, on-chip caches, off-chip caches, etc.) typically found on these systems. Fortunately, sparse matrix factorization can be written to make excellent use of a memory hierarchy [15, 17]. The effect is that extremely large sparse linear systems can be solved in reasonable time on very inexpensive systems. Large sparse linear systems routinely arise in a variety of engineering and operations research disciplines, so there is significant practical interest in solving them in a robust and cost effective manner.

One important issue that affects both robustness and cost effectiveness is the sheer size of the factor matrices computed during the factorization. One can address this issue by simply purchasing an enormous amount of memory. The obvious drawbacks of this approach are:

- It substantially increases the price of the workstation. Today, adding one gigabyte of memory to a typical workstation (enough for a relatively large problem by today's standards) roughly triples the cost of the machine.
- It is inflexible. There will always be problems too big for the chosen quantity of memory.

A second approach is to rely on the virtual memory paging system, allowing the operating system to move data between memory and disk. This approach has the advantage that it requires no modification to

[*] ILOG, Inc., Mountain View, CA
[†] Hewlett Packard Company, Palo Alto, CA

in-core programs, but experience with large-scale scientific applications has shown that it is unacceptably slow.

The third approach, already widely used in the structural analysis community, is to use an out-of-core method in which the Cholesky factor is kept on disk, other data structures remain in core, and data is explicitly moved between memory and disk. Out-of-core sparse factorization was originally performed using frontal or profile methods [12, 16, 25]. More recently, people have moved to more efficient approaches such as the multifrontal method [8]. When the multifrontal method succeeds in running without exhausting available core storage, it is quite effective. Unfortunately, it imposes a significant minimum memory requirement: if the multifrontal stack does not fit in core, the method fails. While this stack is typically much smaller than the factor for two-dimensional structural analysis problems, the stack can grow quite large for three-dimensional problems, and can actually be larger than the factor for some linear programming matrices.

We seek to perform Cholesky factorizations of arbitrarily large problems, constrained only by the size of the disk. We describe and evaluate several methods for doing this. The first is a simple extension of the multifrontal method to handle the case where the multifrontal stack does not fit in core. The second and third are panel-oriented, left-looking, out-of-core methods. We find that while the multifrontal method is extremely effective when its stack fits in core, a left-looking variant is actually more effective when the multifrontal stack does not fit. We present experimental results for some very large matrices from structural analysis, computational fluid dynamics, and linear programming. Even for problems where the out-of-core multifrontal method requires 1 GByte of memory for its stack, this left-looking method achieves two-thirds of in-core performance using only 16 MBytes of in-core storage for numerical data.

We call our best method the *Bobcat* method. Like the animals found in nature and the machines found at construction sites, this method is quite versatile, packing a lot of power into a small space.

2 Sparse Matrix Factorization

Every symmetric, positive definite matrix A has a Cholesky factorization $A = LL^T$; computing L is the most costly step in solving the linear system $Ax = b$. If A is sparse, then normally L is also sparse, but less so: the nonzero structure of A, *i.e.* the set of pairs (i, j) for which $A_{ij} \neq 0$, is a subset of the nonzero structure of $L + L^T$. One ordinarily first permutes the rows and columns of A symmetrically so as to reduce the number of nonzeros in L.

2.1 The Cholesky Algorithm

Let n be the order of A. The following program computes its Cholesky factor L.

```
for j = 1 to n
    copy A(*,j) into L(*,j)
    for k = 1 to j-1
        if (L(j,k) != 0) then cmod(j,k)
    endfor
    cdiv(j)
endfor
```

The *cmod*(j,k) operation subtracts L_{jk} times the k^{th} column of L from the j^{th}; the *cdiv*(j) operation scales the j^{th} column by the square root of its j^{th} element.

We can define an n-vertex tree, known as the *elimination tree* of A [22], by examining the nonzero structure of L. The tree is rooted at vertex n, and all paths to the root traverse a monotone increasing sequence of vertices. In fact, the parent of vertex k is vertex j if the first nonzero below the main diagonal in the k^{th} column of L occurs in the j^{th} row[1]. We let T_j denote the subtree rooted at vertex j (T_n is the whole elimination tree). The significance of the elimination tree is that it expresses the dependences in the elimination algorithm compactly. The elimination tree is the transitive reduction of the digraph of L, *i.e.* the set of column-modification dependences.

[1] If A is reducible, then it has an elimination forest. We assume that A is irreducible.

We use the term *panel* to mean a set of adjacent matrix columns. If the columns of the given matrix and its factor are partitioned into panels, and the panels are numbered from 1 through N, then the factorization program above can be modified as follows:

```
for J = 1 to N
    copy A(*,J) into L(*,J)
    for K = 1 to J-1
        if (L(J,K) != 0) then pmod(J,K)
    endfor
    pdiv(J)
endfor
```

The $pmod(J,K)$ and $pdiv(J)$ operations are natural analogues of the corresponding column operations. They operate on matrices instead of individual columns. In discussing panel methods, we will make use of the panel elimination tree, which has one vertex per panel. We let T_J denote the subtree rooted at panel J.

An important concept in sparse factorization, particularly in the context of performance, is the *supernode* [3]. A supernode is a set of adjacent columns (a panel) in the factor matrix with identical nonzero structure. A supernodal method is a panel method in which the panels are supernodes (perhaps not maximal). The corresponding panel elimination tree is then a supernodal elimination tree. The fact that all columns within a supernode share the same nonzero structure allows most of the work in a panel operation ($pmod(J,K)$ or $pdiv(J)$) to be performed using dense linear algebra kernels. In general, larger supernodes lead to higher performance, since more work is done in dense kernels.

It is customary to perform *supernode amalgamation* to reduce the number of distinct supernodes in the factor matrix, thus increasing the size of the supernodes [2, 8] at the expense of treating some zeros of the factor as if they are nonzero. Amalgamation merges pairs of supernodes with similar but not identical zero structure. Amalgamation typically reduces the number of supernodes in the factor matrix substantially.

2.2 Left-Looking Factorization

The sparse factorization program introduced in the last section specified a particular order of operations. Because all modifications to the pivot column or panel (j or J) are performed together, using source columns to its left, this organization is usually called a *left-looking* method. Note that this sequence is not mandated. The *right-looking* method, which will be discussed shortly, is obtained by interchanging the two loops.

A left-looking method can be made to go out-of-core by taking advantage of the following observation: once column j of L has been completed, row j of L is never accessed again. Liu's general sparse out-of-core scheme [13] writes column j to disk once a $cdiv(j)$ operation has been performed, and occasionally purges all nonzeros in rows numbered less than the current destination column from memory. Results in Liu's paper, however, indicate that this approach requires more in-core memory than the multifrontal method.

2.3 The Multifrontal Method

The multifrontal method [8] is an approach to organizing right-looking sparse matrix factorization with these advantages: it performs most of its computation using dense matrix data structures and algorithms, and it goes out of core in a natural way.

Associated with each supernode is a *frontal matrix.* If the first column in a c column supernode has m nonzeros, then the associated frontal matrix is of order m; the leftmost c columns contain the nonzeros from the supernode, and the remaining triangular matrix of order $m - c$ is called the update matrix. It contains all the updates (by *cmod* operations), from columns in the supernode and their elimination tree descendents, to higher-numbered columns.

The multifrontal method makes a postorder traversal [5] of the supernodal elimination tree. When it visits a vertex, it performs the following steps:

MF-1 Allocate storage for the frontal matrix.

MF-2 Scatter-add the appropriate columns of A into the first c columns of the frontal matrix.

MF-3 Scatter-add the update matrices from all the child supernodes into the frontal matrix. These supernodes are located at the top of a *multifrontal update stack*, and are popped off, as they will not be used after this.

MF-4 Perform c elimination steps on the frontal matrix, to compute the c factor columns and the update matrix.

MF-5 Store the c factor columns in the data structure for the factor, and push the update matrix onto the stack.

The algorithm uses three important data structures: the factor is one, the stack is the second, and the space for the current frontal matrix is the third.

An out-of-core method is obtained by keeping the multifrontal update stack and the current supernode in core (its update portion is typically allocated at the top of the update stack). Factor columns are written to disk as they are computed (Step MF-4). Note that this method performs the minimum possible I/O for an out-of-core method: it writes the factor matrix to disk once. Unfortunately, it fails when either the stack or any of the frontal matrices is too big for main memory. In practice, many large problems cause this simple approach to fail. A simple extension to the method keeps the stack on disk as well. This approach still fails when a single frontal matrix does not fit. We will not consider either of these approaches further, although we will use the traditional multifrontal method as a subroutine for our other methods.

3 Limited Memory Factorization Methods

We now consider three methods for performing limited storage out-of-core factorization. As mentioned earlier, the first is a simple extension of the multifrontal method. The second and third are left-looking methods.

3.1 A Robust Multifrontal Method

Our simple extension of the multifrontal approach begins by identifying the largest subtrees of the supernodal elimination tree that can be factored with a traditional multifrontal method using available in-core memory. Using the terminology of [4], we call each of these subtrees a *domain*, and we refer to all elimination tree vertices not belonging to such subtrees as the *multisector*. During the postorder traversal of the elimination tree, domains are factored with the stack held in core. Once the factorization of a domain is complete, the update matrix from the supernode at the root of the domain is written to a stack file on disk.

When a supernode not belonging to one of these domains is factored, its children have stored their updates in the disk stack. Its own frontal matrix may or may not fit in core. To deal with this possibility, the frontal matrix is divided into panels. Panels are chosen so that each one fits in half of available memory. We use the other half to hold other panels that must be read from disk to fully compute this panel. For each panel, the corresponding panels of the updates from its children are read from the stack file on disk and added to the panel; then, modifications are performed from all earlier factor panels in the same frontal matrix, and the panel is written to disk. Note that no step in this process requires more than two panels to be in memory simultaneously[2].

One subtle issue in this method is that it performs left-looking factorization within the frontal matrix; a panel is modified by factor panels to its left. The (to us) more natural extension to the multifrontal method would perform a right-looking factorization, computing updates from the current panel to later panels in the current frontal matrix and storing them to disk. This extension would perform roughly twice as much I/O as the variant we employ. For each panel (with the exception of the first and last ones in the frontal matrix), the right-looking method must read updates from disk, modify them, and then write them back. The variant we use only reads completed panels. The issue of a left-looking approach performing less I/O

[2] We write the panels of the current supernode's update matrix as they are computed. This occurs before all panels of the frontal matrices of children are read. Thus a simple disk stack does not suffice. One obvious solution is to allow the stack on disk to be nonsequential, but the accompanying fragmentation issues would be difficult to address. Instead we use an odd stack and an even stack. An update matrix from a supernode whose depth in the supernodal elimination tree is an odd number is written to the odd stack, and *vice versa*. Thus, one always reads updates from one stack file and writes updates to the other.

when a frontal matrix does not fit in memory arises later, when we compare multifrontal and left-looking methods.

3.2 Block-Oriented Methods

An alternative out-of-core factorization method, which we considered in [20], partitions the matrix into rectangular blocks. In other words, we partition the columns into contiguous subsets and make the same partition to the rows, thereby blocking the matrix into rectangular sub-matrices. All nonzero values that fall in L_{IJ} are part of one logical block. The partitioning is chosen so that no block occupies more than one-third of available memory. The factorization can then be carried out by viewing the matrix as a dense matrix of sparse blocks and performing a standard, dot-product dense Cholesky factorization on these blocks. At most three blocks need to be in memory at any one time.

There are a number of interesting problems related to this method: how to choose the column partition? What ordering of the task graph minimizes I/O? What block replacement strategy? (The *optimal* strategy, which replaces the block whose next use is last, is usable here since we know *a priori* the entire computational schedule.)

Experiments with our block-oriented method have produced results inferior to those achieved by the panel-oriented methods we present here; we have not pursued this approach further.

3.3 Left-Looking Out-of-Core Methods

The two left-looking out-of-core methods we consider are panel methods. Each panel is a supernode or a part of a supernode – we split the supernodes of L into panels so that no panel is larger than half of available memory. Thus, our panels are contiguous sets of columns that have the same nonzero structure. The left-looking methods perform a panel factorization in the natural way. Let there be N panels. The algorithm is:

```
/* Left Looking, Out-of-Core Panel Factorization */
real X0(half_of_core), X1(the_other_half)

for J = 1 to N
    populate_panel_from_A( J, X0 )

    for K = 1 to J-1
        read_panel_into_core( K, X1 )
        update_panel_from_panel( J, X0, K, X1 )
    endfor

    factor_panel_in_core( J, X0 )
    write_panel_to_disk( J, X0 )
endfor
```

Rows of supernodes must either be all zero or all nonzero. For this reason, in update_panel_from_panel(), every column of the source panel K is involved, while a subset of the columns of the destination panel J are modified.

The reader may object that this approach is less efficient than the multifrontal method, since the update step is a sparse update, while all updates in the multifrontal method are dense. Previous studies [15, 17] have shown that left-looking methods actually give comparable performance to multifrontal methods.

Clearly this algorithm performs more I/O than a multifrontal method when the multifrontal stack fits in core. It reads panels from disk (several times) as well as writing them to disk (once)[3]. We therefore consider hybridizations of the multifrontal and left-looking methods to reduce overall I/O. Many hybrids are

[3] Note, however, that when read_panel_into_core() reads panel K, it need only read the portion at and below block row J. Our disk files are organized in row-major order within each panel to facilitate this optimization. We discovered that failure to do this yields prohibitively inferior results.

possible [1, 14]; we consider two. Both find subtrees of the supernodal elimination tree that can be factored using the multifrontal method with the stack held in core (domains). The remainder of the matrix (the multisector) is factored using the left-looking algorithm above.

The difference between these two hybrids is in how they handle the seam between the domains and the multisector. The first, which we call Pruned Panel, Left-Looking (PPLL), simply performs the left-looking, out-of-core algorithm on the multisector. Source panels are fetched from disk whether they belong to a domain or to the multisector. Note that this hybrid does not need to compute updates from domain nodes to multisector nodes. The multifrontal method is therefore modified to compute updates only to columns within the same domain.

In the second hybrid, the frontal update matrix from the root supernode of each domain is written to disk. When the method factors a panel in the multisector, it fetches all relevant updates from domains, plus all relevant supernodes from the multisector. We call this second hybrid Pruned Panel, Left-Looking (with Updates) (PPLL_U).

Details of the required data structures and their impact on storage requirements are discussed in Appendix A.

3.4 Differences in the Methods

Before presenting results, let us first discuss how we expect the methods to behave. First we note that all three are identical at two extremes: (i) when the multifrontal stack fits in core, and (ii) when the factor matrix is dense. The main differences occur between the two extremes.

Tall, narrow multisector supernodes reveal an important difference between the methods. These supernodes produce large update matrices, often forcing the multifrontal method to write the associated update matrix to disk, even though little work is performed on the entries of this matrix. Such supernodes are handled more effectively by a left-looking approach.

The multifrontal method has an advantage when the columns in a source panel modify only a few of the columns in a destination panel. Left-looking methods can potentially fetch a large source panel to update only a small number of destinations. In the multifrontal method, all modifications are done within dense frontal matrices, so every column fetched from disk modifies every column in the destination panel.

The PPLL method is expected to have a smaller average I/O grain than the other two methods. The reason is that it fetches supernodes from deep in the elimination tree. Supernode widths generally decrease as you move down in the tree. As we will discuss shortly, I/O grain is an important determinant of I/O speed.

4 Results

This section presents results for the methods described above. We then describe two techniques for improving the observed results.

4.1 Test Environment

All performance data presented in this paper comes from a Silicon Graphics Origin 200 system with a 180 MHz R10000 processor and 1 GByte of main memory. We use a machine with a large amount of memory so that we can compare out-of-core factorization performance against in-core performance[4].

In our experiments, in-core storage for the factorization is limited to 16, 32, or 64 MBytes. Most of the results presented here are for 32 MBytes, since the 16 MByte and 64 MByte results are qualitatively similar. Our final performance summary table includes results for all three. It is reasonable to ask whether these memory levels are too small, especially since some of the *unfactored* matrices we consider are larger than the memory set aside for the factorization. Our intent in this paper is to demonstrate that very little memory is required to obtain large fractions of in-core performance. We therefore choose particularly stringent memory constraints. Recognize too that not all of the memory in a machine is available for the factorization.

[4] For the three test matrices whose factors are larger than a gigabyte, we estimate in-core runtime by performing out-of-core factorization within the available memory and subtracting the time spent moving data to and from disk.

Table 1: Test matrices.

			Multifrontal				Ops to
	Rows	$\lvert A \rvert$	max front	max stack	L Ind	L NZ	factor
	(K)	(MBytes)	(MBytes)	(MBytes)	(MBytes)	(MBytes)	(Billion)
One 2-D Problem							
XLEMD	655	202	24	48	13	766	47
Five 3-D Problems							
TROLL	213	73	29	60	6	508	53
TH2	123	19	16	36	4	342	33
SWEDEN	84	80	76	144	3	723	179
CUBE50	125	6	62	117	3	385	83
CUBE80	512	24	387	725	13	2594	1314
Six LP Problems							
GISMONDI	12	5	151	295	1	272	138
MULTICOM	30	10	145	312	2	421	165
DEC92FD	61	10	146	408	3	571	248
FLEET	49	5	151	412	2	362	169
PRODPLAN	1205	316	262	10290	19	1035	340
STE36B	28	19	469	996	1	1217	1105

The file system we use to hold the factor and stack matrices consists of a 2-way striped disk. We measured latency for random read requests at roughly 10 milliseconds and bandwidth at nearly 20 MB/s. To put these numbers in perspective, we note that a 200 KByte read happens at an effective bandwidth of 10 MB/s. A 20 KByte read happens at less than 2 MB/s. Hence, I/O grain size can play an important role in the performance of an out-of-core algorithm.

We measure the performance of the various out-of-core algorithms using two metrics: total required I/O and average read grain size. Note that we ignore write grain size. All of the algorithms considered here write large, contiguous blocks of data. They often write these blocks in small pieces (e.g., completed panels of the factor matrix). These small writes are easily buffered in memory (by the programmer or by the file system), and therefore can be treated as having a large effective grain. Reads, on the other hand, typically involve disparate locations on the disk, making buffering ineffective. We discuss our assumptions about file systems in detail in Appendix B.

5 Test Matrices

Table 1 lists the sparse matrices considered in this paper. These matrices are chosen from a variety of application areas, including structural analysis, computational fluid dynamics, and interior point linear programming. The matrices are heuristically reordered prior to factorization using BEND, a multi-level, vertex separator, nested dissection method [11, 19]. We then perform aggressive supernode amalgamation [2, 8] to reduce the number of supernodes in the matrix, thus increasing the computational grain and consequently increasing factorization performance. The matrices are then reordered to minimize the size of the multifrontal update stack [14]. We assume that the reordered matrix A is on disk at the beginning of the factorization.

The table shows the number of rows and columns in each matrix, as well as the size of A (in MBytes). It also shows the size of the largest frontal matrix plus the maximum size of the multifrontal update stack for a standard multifrontal method (assuming the current update is held at the top of the stack). The table also shows the amount of storage required to keep track of the nonzero structure of L using compressed indices [24], and the amount required to hold the nonzero values in L. We assume an integer requires 4 bytes of storage and a floating-point value requires 8 bytes. The table also shows the number of floating-point operations required to factor the matrix. The first problem comes from the discretization of a 2-D domains. We include only a single 2-D problem because the methods described in this paper are not needed for such problems. Note that for the one 2-D problem considered, the size of the multifrontal stack is much smaller than A. The next five problems come from discretizations of 3-D problem domains (including two regular

Table 2: Extra I/O (relative to $|L|$) and read grain size (in MBytes) for out-of-core methods.

	Extra I/O			Read grain		
Matrix	PPLL	PPLL$_U$	MF	PPLL	PPLL$_U$	MF
XLEMD	0.3	0.2	0.3	0.06	3.32	2.99
TROLL	0.5	0.4	0.5	0.09	3.68	3.47
TH2	0.1	0.1	0.1	0.06	7.00	7.00
SWEDEN	2.2	2.2	2.6	0.41	3.50	3.18
CUBE50	1.9	1.7	2.1	0.07	2.59	2.31
CUBE80	7.8	7.6	6.7	0.25	3.79	3.14
GISMONDI	4.7	5.3	7.3	0.22	3.76	4.00
MULTICOM	5.3	4.5	7.2	0.14	2.35	2.66
DEC92FD	5.2	5.6	8.2	0.07	0.60	0.62
FLEET	6.0	6.2	9.8	0.05	0.39	0.43
PRODPLAN	4.0	11.4	32.3	0.06	0.96	2.27
STE36B	15.4	15.7	15.2	1.75	4.72	4.33

grids). Note that the multifrontal stack can be quite large for these problems. The final six problems are normal equations arising from interior point algorithms in linear programming. Note that the standard out-of-core multifrontal method is ineffective for most of these problems.

5.1 I/O Performance

The first four columns of Table 2 show the amount of extra I/O performed by the PPLL, PPLL$_U$, and MF methods, over and above the amount required to write L to disk once. An entry of 1.0 means that the number of matrix entries read and written during the factorization is $2|L|$. The final three columns in the table show average read grain sizes (in MBytes) for the various methods. Recall that all results presented in this paper except those in the final table use 32 MBytes of memory to hold factor data.

Looking at the I/O volume numbers, we find that the three methods perform similar amounts for the 2-D and 3-D problems. The left-looking methods usually perform significantly less for the linear programming matrices. The data is particularly striking for problem PRODPLAN, where the PPLL method performs one-eighth as much additional I/O as the MF method.

Considering I/O grain, the data in the table shows that the PPLL method produces a much smaller average read grain size than the other methods. Note that a read grain size of 100 KBytes (roughly average for this approach) achieves a transfer rate of less than 7 MB/s from a 20 MB/s disk system with a seek time of 10 milliseconds.

It is clear from the data in the table that the PPLL$_U$ method is comparable to each of the other methods in their respective areas of strength. The one exception is that it (like MF) does far too much I/O for problem PRODPLAN. The next section shows that this notable failure of PPLL$_U$ can be avoided through a better choice of domains.

5.2 Optimal Domains

Recall that the methods of the previous section choose the largest domains possible. For some matrices, this strategy is far from optimal. Consider the results for method PPLL$_U$ on matrix PRODPLAN. This matrix contains many tall, narrow supernodes near the leaves of the supernodal elimination tree. The offending supernodes generate very large frontal update matrices. The strategy of choosing the largest possible domains places these narrow supernodes within domains, while the associated frontal update matrices are written to and subsequently read from disk. A better approach would be to place these supernodes in the multisector. Clearly, a more effective general strategy is needed for choosing domains.

We now describe an algorithm that chooses the unique set of domains that minimizes I/O volume. The first step in this algorithm is to compute the amount of I/O that would be generated if there were no domains. (we again consider only I/O above and beyond that performed in writing L to disk). Recall that

Table 3: Extra I/O (relative to $|L|$) and read grain size (in MBytes) for optimal domains with algorithm PPLL$_U$.

Matrix	Extra I/O	Grain (MBytes)
XLEMD	0.2	3.32
TROLL	0.4	3.68
TH2	0.1	7.00
SWEDEN	2.1	3.40
CUBE50	1.7	2.59
CUBE80	7.6	3.76
GISMONDI	4.8	1.47
MULTICOM	4.5	2.35
DEC92FD	5.3	0.54
FLEET	5.8	0.34
PRODPLAN	4.3	0.27
STE36B	15.5	4.04

the portion of panel K below block row J must be fetched from disk for every panel J for which the block L_{JK} is not zero. Summing these quantities over all relevant panels J gives a quantity $fetch(K)$, the total amount of I/O associated with fetching panel K. We can then easily compute $fetch(T_J)$, the total volume of I/O generated by fetching panels from subtree T_J, for each T_J ($fetch(T_N)$ is the total I/O volume).

Note that all of the I/O captured in $fetch(T_J)$ can be avoided by creating a domain T_J. The only cost of doing so is a write and subsequent read of the update matrix from T_J. We can thus easily compute $saved(T_J)$, the amount of I/O that would be saved by creating a domain out of T_J. (Of course, if a subtree cannot be factored with the in-core multifrontal method, it is not allowed to be a domain.)

Given the quantity $saved(T_J)$ for each subtree, our goal of minimizing I/O is then equivalent to choosing the set $\mathcal{D}$ of disjoint, allowed subtrees that maximizes $\sum_{T_J \in \mathcal{D}}(saved(T_J))$. The key observation here is given an allowed subtree T_J, the optimal choice of disjoint subtrees from this tree is either: (i) T_J itself, or (ii) the union of the optimal disjoint subtrees from the trees rooted at the children of J. This observation leads to a simple recurrence for identifying the optimal set of subtrees. For each panel J with children $kids(J)$, if you know $optimal(T_c)$ for all $c \in kids(J)$ (the optimal savings from disjoint subtrees rooted at c), then $optimal(T_J)$ is given by:

$$\max\left(\sum\nolimits_{c \in kids(J)}(optimal(T_c)),\ saved(T_J)\right).$$

The optimal subtrees can be recovered by finding the largest allowed subtrees T_J for which $optimal(T_J) = saved(T_J)$.

Note that this algorithm can be improved somewhat. Recall that I/O grain size is often a more important consideration than I/O volume. Rather than computing $fetch(K)$, which is the volume of I/O associated with panel K, we can instead compute $time(K)$, an estimate of the runtime cost of fetching panel K. The runtime of a single fetch would then include the latency of a read plus the transfer time for reading the panel from disk. Similarly, we can compute $saved(T_J)$ as the runtime savings of writing and subsequently reading an update from T_J rather than fetching the panels in T_J.

Table 3 shows the results of applying this algorithm to our test set. The table shows extra I/O for the method and the average read grain size. Comparing this table to Table 2, we find that I/O volumes decrease significantly for several problems. Extra I/O for algorithm PPLL$_U$ on matrix PRODPLAN drops from 11.4 times $|L|$ to 4.3. Read grain sizes drop for some problems as well (from 0.96 MBytes to 0.27 MBytes for PRODPLAN), but recall that the method takes grain size into account. It has chosen the optimal tradeoff between I/O volume and I/O grain. This is the Bobcat algorithm – PPLL$_U$ with optimal domains.

5.3 Performance

We now look at the performance of the implementation of the PPLL$_U$ method we have described. Table 4 gives performance (in Mflops) for in-core factorization of the matrices in our test set, as well as the fractions

Table 4: Factorization performance of PPLL_U and in-core.

	MF stack	$\|L\|$	In-core perf	Fraction of in-core perf		
	(MBytes)	(MBytes)	(Mflops)	16 MB	32 MB	64 MB
XLEMD	48	779	201	0.94	0.98	0.98
TROLL	60	514	229	0.89	0.95	0.97
TH2	36	346	227	0.91	0.95	0.97
SWEDEN	144	726	247	0.84	0.91	0.96
CUBE50	117	388	249	0.83	0.91	0.95
CUBE80	725	2606	263	0.69	0.82	0.88
GISMONDI	295	272	255	0.80	0.87	0.94
MULTICOM	312	422	249	0.76	0.89	0.95
DEC92FD	408	573	253	0.74	0.81	0.88
FLEET	412	364	253	0.69	0.77	0.85
PRODPLAN	10290	1053	213	0.63	0.67	0.69
STE36B	996	1218	270	0.64	0.79	0.85

of in-core performance obtained with 16, 32, and 64 MBytes of in-core storage. Note that performance degradations are quite small. With 32 MBytes of memory, the method achieves 77–98% of in-core performance for all problems except PRODPLAN, even when the matrix and the multifrontal stack are significantly larger than memory. Matrix PRODPLAN gets 67% of in-core performance.

5.4 Overdecomposition

It is not strictly necessary to reserve half of in-core memory for the source panel and half for the destination. For example, one could fill all but a small piece of memory with a destination panel and fetch individual source columns from disk. This strategy reduces I/O volume significantly, since each column fetched from disk modifies many more destination columns. The drawback is significantly reduced I/O grain size and compute grain size. One can strike a balance between the two by performing *overdecomposition*, by which we mean a partitioning of the matrix into smaller panels, so that several may simultaneously occupy main memory.

To be more specific, we divide the matrix into panels so that each panel is no larger than $1/D$ of available memory. The factorization can then hold $D-1$ destination panels in core and fetch a single source panel at a time. This approach provides a clean approach to working with panels as logical units while not requiring symmetry between sources and destinations. It has the added advantage that the destination panels do not need to come from the same supernode. This reduces I/O when there are narrow supernodes. The disadvantage is that read grain sizes and compute grain sizes decrease. We experimented with PPLL_U using $D = 4$. We found that I/O volumes drop by 20-30% for the 3-D and linear programming problems, while read grain sizes usually drop by nearly a factor of two.

Given the large fraction of in-core performance obtained with $D = 2$, it is perhaps not surprising that our overdecomposition approach did not significantly improve achieved performance. If I/O costs reduce performance when $D = 2$ by 20%, for example, and overdecomposition reduces I/O volumes by 25%, then the maximum possible performance improvement is 5%. This benefit must be traded off against the drawbacks of overdecomposition; it reduces the I/O grain, thus increasing the fixed costs associated with the I/O, and it reduces the width of the panels, thus reducing the performance of the computational kernels. While overdecomposition consistently improved performance, the maximum improvement was only a few percent. We would expect to see significant benefits only when the I/O rate is significantly lower, relative to the computation rate, than it is on the machine used for our experiments.

6 Related Work

Reducing memory requirements in the multifrontal method by performing the frontal matrix computation out-of-core is not a new idea. We've heard it discussed in several contexts [9, 10, 23]. Details have not been published, however.

Salmon and Warren have recently conducted an investigation of out-of-core methods for the N-body problem. Their motivation is identical to ours, and they also achieved excellent results; a slowdown of about fifteen percent compared with in-core methods. In contrast to our approach, they built a user-level demand paging strategy, with variable page granularity and replacement policy, and employed it as the basis of an implementation. The completely dynamic nature of the interactions in N-body solvers seems to mandate this approach [21].

An interesting approach to solving linear systems in limited memory was proposed by Eisenstat, Schultz, and Sherman [9]. Their proposal does not rely on disk; rather, the columns of L are simply discarded after their last involvement in a *cmod* operation, with the exception of the lower right submatrix that corresponds to a top level separator. Retaining only this portion of the matrix and the updated right-hand side allows one to solve for the unknowns on the top-level separator. If one then removes these rows and columns from the matrix, it becomes reducible, and one recursively applies the method to the decoupled subproblems. While this method has the nice property of requiring no disk, it is not robust under very limited main memory, and it performs significant redundant work when recomputing discarded portions of the matrix.

7 Discussion

One technique not considered here that might improve read grain size, particularly for problem PRODPLAN, is sibling amalgamation. The amalgamation approaches in the literature only consider merging a child into its parent. We could identify cases where merging a child into one of its siblings introduced fewer nonzero values. Unfortunately, finding appropriate sibling merges is much more complicated than finding parent-child merges. One reason is that a supernode has only one parent, while it can have many siblings. Another is that the nonzero structure of the parent is always a superset of the structure of the child, so the number of nonzero values added to the child is easily computed. Siblings do not have this superset relationship. This issue will require further investigation.

In our view, this work changes the relative merits of direct and iterative methods for symmetric positive definite problems. While iterative methods are often faster than direct methods, perhaps the most common motivation for their use is to be able to solve very large linear systems using little memory. The results of this study show that direct methods can also use very little memory.

Another issue that will require further investigation is whether the PPLL$_U$ method explored here could be used as the basis for a limited memory *parallel* out-of-core method. One obvious approach would be to build parallel computational kernels. Recall, however, that the PPLL$_U$ method spent 10-30% of its runtime waiting for data from disk. Without also doing parallel I/O, the benefits of adding processors would fall off quickly. An alternative approach would be to use a parallel panel left-looking method [18], where processors would be responsible for updates to distinct destination panels. Each processor would then fetch relevant source panels from disk independently. This approach might lead to a potentially difficult tradeoff: wide panels reduce I/O volumes, but they also reduce parallelism. This issue will require further study.

Another possible extension is limited memory unsymmetric factorization with partial pivoting. UMFPACK [6] and SuperLU [7] bear many similarities to symmetric multifrontal and left-looking methods, respectively. It would be interesting to consider how the extensions described here might apply to these approaches.

An issue not considered here is asynchronous I/O. Most file systems allow a program to issue a file system request, continue with computation, and then retrieve the result of the request at a later time. This allows the program to hide much of the latency of the request. In a left-looking out-of-core approach, the program could overlap the fetching of a panel from disk with the computation of an update from the previous panel. Of course, doing so requires added memory to hold both the previous and current source panels. This approach therefore introduces a tradeoff: more I/O due to the reduction in available memory versus better hiding of I/O costs.

One technique we did not consider in this paper is a 2-D decomposition within the panels of the PPLL$_U$ approach. One could easily iterate through rectangular sub-matrices of the current destination panel, fetching appropriate sub-matrices from source panels when performing updates. This approach has the advantage that the width of a destination block does not need to decrease as the height of the panel increases. While the asymptotic growth rates favor a 2-D approach as the problem size goes to infinity (or the memory size goes to zero), the constant factors are such that the method would only provide significant advantages for matrices much larger than those considered here.

8 Conclusions

This paper has explored three approaches to limited memory Cholesky factorization. Each of the simple approaches we considered had a serious flaw when we looked at the behavior of the methods on a wide range of matrices arising in structural analysis, computational fluid dynamics, and interior point linear programming. We enhanced one of the approaches (a pruned panel, left-looking method) to address the observed flaws. We optimally chose portions of the matrix to factor using a multifrontal method, and we overdecomposed the matrix into smaller panels than strictly necessary to reduce I/O volumes. The resulting Bobcat method gives most of the performance of an in-core method using only a small fraction of the in-core memory.

Acknowledgements

We would like to thank Cleve Ashcraft and Roger Grimes for several very illuminating discussions, Stan Eisenstat for his helpful comments on an earlier draft, and an anonymous referee for many very perceptive suggestions for improving the paper.

References

[1] C. Ashcraft, *A common formulation of the general sparse and multifrontal factorization algorithms*, Tech. Rep. MEA-TR-207, Boeing Computer Services, december 1992.

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

[3] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon, *Recent progress in sparse matrix methods for large linear systems*, International Journal of Supercomputer Applications, 1 (1987), pp. 10–30.

[4] C. Ashcraft and J. W. H. Liu, *Robust ordering of sparse matrices using multisection*, Tech. Rep. ISSTECH-96-002, Boeing Information and Support Services, 1996.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1994.

[6] T. A. Davis and I. S. Duff, *A combined unifrontal/multifrontal method for unsymmetric sparse matrices*, Tech. Rep. TR-95-020, University of Florida, 1995.

[7] J. Demmel, S. Eisenstat, J. Gilbert, X. Li, and J. Liu, *A supernodal approach to sparse partial pivoting*, Tech. Rep. CSD-95-883, Computer Science, UC Berkeley, September 1995.

[8] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Software, 9 (1983), pp. 302–325.

[9] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman, *Software for sparse Gaussian elimination with limited core storage*, in Sparse Matrix Proceedings 1978, I. S. Duff and G. W. Stewart, eds., SIAM Press, 1979, pp. 135–153.

[10] R. Grimes. private communication.

[11] B. Hendrickson and E. Rothberg, *Improving the runtime and quality of nested dissection ordering*, Tech. Rep. SAND96-0868J, Sandia National Laboratories, 1996.

[12] B. M. Irons, *A frontal solution program for finite elements*, Internat. J. Numer. Methods, 2 (1970), pp. 5–32.

[13] J. W. H. Liu, *An adaptive general sparse out-of-core Cholesky factorization scheme*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. 585–599.

[14] ——, *On the storage requirement of the out-of-core multifrontal method of sparse factorization*, ACM Trans. Math. Software, 12 (1987).

[15] E. Ng and B. Peyton, *Block sparse Cholesky algorithms on advanced uniprocessor computers*, SIAM J. Sci. Comput., 14 (1993), pp. 1034–1056.

[16] J. R. Perry, *Secondary storage methods for solving symmetric, positive definite, banded linear systems*, Tech. Rep. Research Report 201, Department of Computer Science, Yale University, 1981.

[17] E. Rothberg and A. Gupta, *An evaluation of left-looking, right-looking and multifrontal approach to sparse Cholesky factorization on hierarchical-memory machines*, International Journal of High Speed Computing, 5 (1993), pp. 537–593.

[18] E. Rothberg, A. Gupta, E. Ng, and B. Peyton, *Parallel sparse Cholesky factorization algorithms for shared-memory multiprocessor systems*, in Proceedings of the Seventh IMACS International Conference on Computer Methods for Partial Differential Equations, 1992.

[19] E. Rothberg and B. Hendrickson, *Sparse matrix ordering methods for interior point linear programming*, Tech. Rep. SAND96-0475J, Sandia National Laboratories, 1996.

[20] E. Rothberg and R. Schreiber, *An alternative approach to sparse out-of-core factorization.* SIAM Sparse Matrix conference talk, October 1996.

[21] J. Salmon and M. S. Warren, *Parallel, out-of-core methods for n-body simulation*, in Proc. Eighth SIAM Conf. on Parallel Processing for Scientific Computing, 1997.

[22] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Trans. Math. Software, 8 (1982), pp. 256–276.

[23] S. Shamsian, L. Komzsik, and D. Petesch, *Direct sparse solver in msc/nastran.* SIAM Sparse Matrix conference talk, October 1996.

[24] A. H. Sherman, *On the efficient solution of sparse systems of linear and nonlinear equations*, PhD thesis, Yale University, 1975.

[25] E. L. Wilson, K. J. Bathe, and P. Doherty, *Direct solution of large systems of linear equations*, Comput. Structures, 4 (1974), pp. 363–372.

A Out-of-Core Sparse Matrix Data Structures

While the nonzero values in the matrix consume the majority of storage during the factorization, other data structures also consume in-core memory. The largest of these is the data structure that records the nonzero structure of the factor matrix (the compressed indices). Compressed indices typically consume a small fraction of the storage required by the nonzeros in the factor matrix for the matrices we consider (typically around 1%). However, the size of the compressed indices is not always trivial compared to the amount of in-core memory used in the out-of-core method. We would prefer not to be forced to keep these indices in memory.

Note that the multifrontal method only needs to retain in-core the nonzero structures of the frontal update matrices on the update stack, plus the nonzero structure of the current supernode. One can maintain a stack of indices, similar to the update stack, to retain these nonzero structures. The size of this stack is insignificant.

Our PPLL$_U$ method only needs to retain the nonzero structures of the supernodes in the multisector portion, plus the structures of the domain updates. Again, the aggregate size of this structure information is quite trivial.

Structural information becomes a problem in the PPLL method. This approach often reaches deep down in the supernodal elimination tree to fetch a panel. One option is to keep structure information on disk, retrieving the structure of a panel when the panel itself is retrieved. While this option would not increase I/O volume significantly, it would roughly halve the I/O grain size unless this information were somehow interspersed with the panel data (so that the structure of a panel were stored contiguously with the nonzero values in the panel). We consider this somewhat awkward solution a negative for this approach.

In practice, we believe that the compressed indices are sufficiently compact that it is reasonable to keep them in memory, letting the virtual memory system move them to and from disk as necessary. Due to the access patterns discussed above, though, we expect the PPLL approach to generate significantly more virtual memory I/O traffic for the compressed indices than the other two approaches.

B File System Characteristics

To evaluate the effectiveness of an out-of-core factorization method, it is important to understand the characteristics of the disks drives and file systems found on high-performance computers. The features we describe here are typical in UNIX and advanced PC operating systems.

The performance of a disk drive is usually described using two parameters: *latency* and *transfer rate*. The latency is the time to move a physical disk head to the appropriate portion of the disk (a *seek*) plus the *rotational delay* of the spinning disk. The transfer rate is determined by the rate at which data passes under this disk head. Typical parameters for a low-cost SCSI disk today are 10 milliseconds latency with a transfer rate of 10 MB/s.

Application software does not generally write data directly to the disk. Instead, it submits requests to the file system, and the file system determines how to satisfy those requests. File systems employ several techniques to improve their overall performance. One important technique is file caching, wherein the file system uses free memory in the system to cache disk data. A write from a user application is copied to the file system cache rather than being written straight to disk. Similarly, file system reads move data from the cache to the user if the requested data is available in the cache, and from the disk to the cache and then to the user if it is not. When the disk cache fills, the file system must discard cached data. Caching provides several benefits. One obvious benefit is that file system reads are often serviced from the cache, avoiding the cost of a disk access. The effectiveness of this caching of course depends on the data access pattern and the size of the disk cache. Another benefit is that most writes complete almost immediately, since they simply transfer data to the cache. The data must eventually be transferred to disk, but these transfers are usually performed in the background, and they can often be performed at a coarser grain than the user's original write requests. The cost of a disk cache is the system memory used for the cache.

File systems also use *striping* to increase disk throughput, where the file system creates one logical disk using multiple physical disks. Data blocks from a single file are then interleaved on these disks. When a file system read or write spans multiple data blocks (which are typically several KBytes), the file system can perform the appropriate reads or writes to different disk drives in parallel. If each physical disk can deliver 10 MB/s, an n-way interleaved file system can then deliver $10n$ MB/s (until some other resource, such as the disk controller, saturates). Note that striping does not improve latency. On the contrary, it may make it worse, since each drive must move its disk head to the appropriate location to service a request.

As noted earlier, the machine used to perform our experiments is endowed with sufficient memory to hold several of our test matrices entirely in the file system cache. To obtain realistic performance numbers for out-of-core methods on this machine, given that the data is often not actually fetched from the disk, we use a simple trick. For each file system read, we compute the amount of time the read should require (using a simple function of the disk seek time, the disk transfer rate, and the size of the request). If the read completes in less time than expected, the program sits in an idle loop until the appropriate amount of time has elapsed.