Toward an Efficient Column Minimum Degree Code for Symmetric Multiprocessors

Tzu-Yi Chen[*] John R. Gilbert[†] Sivan Toledo[‡]

Abstract

Ordering the columns of a nonsymmetric sparse matrix can reduce the fill created in its factorization. Minimum-degree is a popular heuristic for ordering symmetric matrices; a variant that can be used to order nonsymmetric matrices is called column minimum degree. In this paper we describe the design of a multithreaded approximate column minimum degree code. We present a framework for the algorithm, study potential sources of parallel inefficiency, and describe our attempts to circumvent the obstacles. We present experimental results that support our analysis.

1 Motivation

The factorization $A = LU$, where L is lower triangular and U is upper triangular, is frequently used in solving the system of equations $Ax = b$ via direct methods. Ordering the rows and columns of A prior to the factorization is common if A is large, sparse, and symmetric; ordering the columns is common if A is large, sparse, and nonsymmetric. A good ordering reduces the fill in the factors L and U, hence reducing both the time needed to factor A and the space needed to store the factors. A common heuristic for finding good orderings is called minimum degree. We describe this algorithm and some variants in Section 2.

Recently, a parallel sparse LU-factorization algorithm has been designed and implemented [10]. The algorithm is designed for shared-memory multiprocessors, but uses a sequential ordering algorithm which can be a bottleneck when solving large systems. To remove this bottleneck we designed and implemented a parallel minimum-degree ordering code.

This paper describes the design, implementation, and performance of a parallel minimum-degree algorithm. Section 2 describes sequential minimum-degree algorithms; Section 3 describes our parallel algorithm. Because we found that our algorithm does not always speed up significantly with more processors, Section 4 explores possible causes

[*]Computer Science Division, University of California, Berkeley, CA 94720 (tzuyi@cs.berkeley.edu). This work was supported in part by DARPA under contract No. DABT63-95-C-0087, the Department of Energy under DOE Grant Nos. DE-FG03-94ER25219, and DE-FC03-98ER25351, the National Science Foundation under NSF Infrastructure Grant No. CDA-9401156, and Grant No. ASC-9313958, and a Gift from IBM under the Shared University Research Program. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

[†]Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304 (gilbert@parc.xerox.com). This work was supported in part by DARPA under contract No. DABT63-95-C-0087.

[‡]Tel Aviv University (sivan@math.tau.ac.il). Part of this work was performed while Sivan Toledo was visiting Xerox PARC. This work was supported in part by DARPA under contract No. DABT63-95-C-0087.

of parallel inefficiency. Section 5 summarizes the results of our experiments on large matrices from Davis's matrix collection [4]. These experiments were performed on an SGI Power Challenge machine (we also ran experiments on SGI Origin 2000 and Origin 200 machines, as well as on a Sun Enterprise 5000). We present our conclusions in Section 6.

2 Background

The basic minimum degree algorithm for symmetric matrices, first proposed by Tinney and Walker [16], works on the graph $G = (V, E)$ corresponding to the matrix A. The vertices (nodes) $v_1, v_2, \ldots v_n$ of G correspond to the n rows and columns of A, and the edge set consists of edges e_{v_i,v_j} for each nonzero A_{ij} in A. At each step a vertex with the smallest degree is chosen and ordered next. The chosen vertex is removed from the graph and edges are added so that its neighbors form a clique (a fully connected subgraph). The added edges represent matrix elements that fill when we eliminate the corresponding row and column from A. The algorithm updates the degrees of the neighbors of the eliminated vertex, chooses another vertex, eliminates it, and so on until all vertices are ordered.

Since the development of the original symmetric minimum degree algorithm over 30 years ago, numerous changes have been suggested, each change improving the speed of the algorithm or the quality of the ordering found (see [6] for a survey). Important ideas incorporated in our code include the clique-cover representation, supernodes, multiple eliminations, and approximate degree updates. The clique cover representation of the graph uses a set of cliques to cover all the edges in the graph. An elimination step in this representation is implemented by merging all the cliques that the eliminated node belongs to. This representation is compact and eliminations can be implemented efficiently, but computing the degree of a node is nontrivial. Supernodes, or groups of nodes with the same neighbors, are represented by one representative node to reduce work done in the algorithm. Multiple elimination means selecting an independent set of nodes, all with minimum degree, in each iteration and eliminating all the nodes in the set before updating any node degrees. Though not an exact implementation of the minimum degree algorithm, this approach usually produces orderings that are just as good. Using approximate degrees reduces the amount of work required to update the degree of each node in a clique-cover representation. Many degree approximations are possible. One approximation is used in the `symamd` and `colamd` codes [1], another is used in the Matlab routines `symmmd` and `colmmd` [7].

The routines `colamd` and `colmmd` can order nonsymmetric matrices. For nonsymmetric matrices pivoting may be required for stability in the subsequent factorization, hence predicting and minimizing fill is more difficult. Both `colamd` and `colmmd` compute a symmetric ordering of A^TA; the permutation P is then applied only to the columns of A. The fill in an LL^T factorization of PA^TAP^T is an upper bound on the fill in an LU factorization of PA [8], and reducing fill in PA^TAP^T tends to reduce fill in the LU factors of PA. We refer to the permutation P as a *column ordering.*

State-of-the-art column minimum degree algorithms do not compute A^TA. Instead, they use the nonzero structure of A to construct an initial clique cover of A^TA directly. Essentially, the nonzero structure of each row of A is interpreted as a bitmap that specifies membership of columns in one clique.

Minimum degree algorithms are not the only heuristics for ordering matrices. Others include minimum fill [14, 15], nested dissection, and hybrid methods combining nested dissection and minimum degree [3, 9]. Although other algorithms are easier to parallelize, none is currently as widely used for column ordering nonsymmetric matrices as variants on minimum degree.

3 The Algorithm

Our main strategy for parallelizing the algorithm is to use several processors to perform multiple eliminations in a single iteration. Each node elimination is performed sequentially by one processor, but different processors can eliminate different nodes in parallel. In every iteration, one processor finds an independent set of nodes with small degrees to eliminate and creates a work queue of eliminations for the iteration; the other processors wait during this phase. Next, each processor repeatedly takes a node from the work queue and eliminates it until the queue is empty. The following pseudocode summarizes the algorithm:

```
1   preprocessing
2   calculate approximate initial degrees
3   repeat {
4       proc 0: find a set of nodes $S$ to eliminate
5       do in parallel {
6           repeat {
7               choose an uneliminated node $v$ in $S$
8               eliminate $v$ by merging cliques and updating clique
                    lists of adjacent nodes
9               calculate degrees of nodes adjacent to $v$
10          } until all nodes in $S$ are eliminated
11      }
12  } until all nodes ordered
```

The preprocessing step (line 1) identifies identical cliques (i.e., identical rows of A) and identical nodes (i.e., identical columns in A). We use a hash table to quickly identify identical rows and columns; identical rows are removed and identical columns are grouped as supernodes. This step can be performed in parallel. The code assigns a set of rows and columns to each processor for insertion into the hash table, then assign a set of hash-table buckets to each processor for further processing.

The calculation of approximate initial degrees is also done in parallel using a similar load-balancing mechanism.

The selection of an independent set S of nodes with small degrees (line 4) is controlled by three parameters which determine the number of nodes in S, their degrees relative to the smallest degree, and the amount of work permitted in this phase. These parameters can be set at compile time or run time. The code works as follows. Initially all nodes are eligible. The code repeatedly selects an eligible node with the smallest degree, marks its neighbors as ineligible, and continues by selecting another eligible node. This process terminates when one of the following conditions occurs: (a) enough nodes have been found, (b) the degree of the selected node is too high, or (c) too many nodes have been touched (selected or marked).

Setting the independent-set parameters requires considering various trade-offs. The number of nodes to be selected, k, is chosen to provide enough work in the queue to ensure reasonable load balancing. Since the elimination of two nodes can differ significantly in cost, we usually attempt to place in S more nodes than there are processors, so that if one processor is busy with a single expensive elimination, another processor which finishes an elimination quickly can eliminate another node. The trade-off is that attempting to find a large set S may require more work and may produce an inferior ordering (especially if nodes with high degrees have to be chosen). We set the maximum allowed deviation *tol* from the minimum degree so that the quality of the ordering remains good. Allowing nodes with high degrees to be selected is inconsistent with the basic idea of the minimum degree algorithm and might degrade the ordering in terms of the fill in the LU factorization. Of course, a small value for *tol* might prevent the algorithm from finding enough nodes to be eliminated in every iteration, and hence lead to poor load balancing. The limit *numsearch* on the number of nodes that are touched is used to prevent the algorithm from performing too much work in this serial phase of the algorithm.

The three main data structures used by the algorithm are a node array, where each node points to an array of cliques containing that node; a clique array, where each clique points to a linked list of the nodes it contains; and a degree array which contains linked lists of nodes which share the same approximate degree. Our code serializes accesses to these data structures by using mutual exclusion variables (mutexes). Each node, clique, and list of nodes with the same degree is protected by a mutex. The mutexes allow nodes to be eliminated in parallel without corrupting these data structures.

4 Analysis

We now analyze the performance of this parallel code. Generally speaking, we found that our algorithm rarely speeds up significantly as the number of processors increases. In designing and tuning the algorithm we made several changes to try to eliminate sources of inefficiency. Still, the algorithm often does not speed up. Therefore, our analysis in this section is meant to explain the sources of inefficiency.

Analyzing the algorithm suggests several possible explanations:

1. Ahmdahl's law: there is too much serial work.
2. Lack of parallelism: we are unable to find enough low-degree nodes to eliminate in parallel in each iteration.
3. Load imbalance: Some nodes require more nodes to eliminate than others, which might lead to load imbalance.
4. Contention: processors block waiting to acquire mutual exclusion variables.
5. Expensive synchronization primitives: barriers and mutexes are slow.
6. Cache misses: the code does not attempt to reuse data in a processor's cache. A processor may update a node which is eliminated by another processor in the following iteration. The data structure of the node is first brought to the first processor's cache, only to be used once and transported to the second processor's cache.

In the remainder of this section we analyze each possibility, suggesting causes and solutions. But first we discuss an issue that complicates the analysis, namely, the sensitivity of the algorithm to its scheduling.

4.1 Scheduling Sensitivity

To measure the parallel efficiency of our code on any given matrix, the algorithm must perform the same amount of work regardless of the number of processors. Unfortunately this is not always the case because of the code's sensitivity to thread scheduling. Not only do runs on the same input with different numbers of processors do differing amount of work, but multiple runs with the same number of processors can perform differing amounts of work. In contrast, parallel algorithms such as matrix multiplication and Cholesky factorization are insensitive to scheduling.

Sensitivity to the scheduling is due to the fact that a node's approximate degree depends on the order in which adjacent nodes are eliminated and their degree updates applied. For example, if a node v is adjacent to several nodes being eliminated in the same iteration, the degree of v at the end of the iteration depends on the order of the degree updates from the eliminated nodes. This further determines whether or not v is chosen for elimination in the next iteration and hence affects the overall ordering computed.

We assess the efficiency of our algorithm mostly by analyzing speedups, even though speedups do not accurately measure efficiency when runs with different numbers of processors perform different computations. We limit the analysis, however, to matrices in which different runs produce roughly the same computation and perform roughly the same amount of work. We estimate the similarity of two computations by comparing the number of iterations the algorithm uses. On most matrices, runs with different numbers of processors differ in the number of iterations by only 1 to 2%, which implies that these runs perform roughly the same amount of work. This allows us to use speedup as an accurate overall measure of parallel efficiency.

On some matrices, however, runs with different numbers of processors and even multiple runs with the same number of processors perform totally different computations. For example, ordering the matrix bbmat with one processor required 1037 iterations, but three separate runs with two processors required 512, 744, and 496 iterations. For such matrices, we cannot evaluate parallel efficiency simply by comparing the running time on multiple processors to the running time on one processor. Different runs with different numbers of processors may perform different amounts of work.

Therefore we limit our analysis to matrices whose ordering is relatively insensitive to the scheduling of the algorithm, as determined by the number of iterations.

4.2 Ahmdahl's Law

The independent set S is selected sequentially by one processor. If this phase of the code performs a significant fraction of the total work, the code will not speed up.

The current code spends about 1-5% of the total time on one processor in finding the set S if $k = 1$, and about 7-15% of the time on one processor in finding S if $k = 128$. While this percentage will not enable the code to scale well, it does not prevent the code from speeding up on 2 or 4 processors. We are not aware of efficient parallel algorithms to find such an independent set. While there are parallel algorithms for finding maximal independent sets, we require an algorithm for selecting a relatively small independent set consisting only of nodes with small degrees.

During the design of the algorithm, we also experimented with wider separation between selected nodes. Our current independent set S may contain nodes that are within distance 2 in G, but not nodes within distance 1 (i.e., neighbors). We also tried restricting S to nodes that are at least distance 3 or 4 apart. The advantage of such schemes is that they allow processors to perform eliminations with less locking of data structures. If nodes in S are at least distance 4 apart, no locking of cliques or nodes is necessary. If nodes are distance 3 apart, some locking is required but less than in the current distance-2 algorithm. Regardless of the distance, locking elements of the degree list is required.

Our experiments showed that finding a p-node set S, where p is the number of

Table 1

This table shows the number of iterations taken by the algorithm where p, the number of processors, is varied from 1 to 8. The maximum number of nodes found in each iteration is p, the number of nodes searched is set to 4p, and the tolerance is set to 4 and 1500 on the left and right sides, respectively.

	# of iter with tol = 4				# of iter with tol = 1500			
name	p=1	p=2	p=4	p=8	p=1	p=2	p=4	p=8
goodwin	1303	1012	923	827	1303	923	663	551
memplus	9113	5095	3063	1899	9113	4878	2775	1690
orani678	924	630	513	462	924	610	472	378
shyy161	37778	19458	10026	5201	37778	19079	9521	4803
wang4	6998	3832	2227	1351	6998	3528	1799	936

processors, whose elements are at least distance 4 apart took 60% to 90% of the total time (90% on the memplus matrix). Finding nodes of distance at least 3 apart took up to 75% of the time (also on the memplus matrix). Selecting S takes much longer when the nodes must be distance 3 or 4 apart because a large number of nodes are marked as ineligible for every selected node. Therefore, we decided to abandon this approach and only require that the nodes in S not be neighbors. As explained above, this reduces the running time of the sequential phase that selects S to a small fraction of the total running time.

For the remainder of the paper we assume the nodes in S are at least distance 2 apart.

4.3 Lack of Parallelism

By a lack of parallelism we mean the algorithm cannot find enough nodes to eliminate in each iteration, hence in some iterations some processors remain idle.

Can the algorithm find enough nodes to eliminate in parallel? We found that when we allow many nodes to be touched (i.e., *numsearch* is large) and for nodes with high degrees to be selected (i.e., *tol* is large), the algorithm can find large independent sets S in almost all the iterations. With no limits on these parameters, using a target size of $k = 1$ for S on a matrix av41092 requires 14416 iterations, but using a target size of $k = 128$ requires only 124 iterations. This indicates that the algorithm indeed found a 128-node independent set in most iterations. The results are similar on several other matrices. On some matrices, however, the reduction in the number of iterations is not so dramatic, implying that on many iterations the algorithm was only able to find much smaller independent sets. For example, on a matrix raefsky4, the number of iterations only dropped from 2997 to 50, a factor of 60, when we increased the target size k from 1 to 128.

When we set tight tolerances on the maximum degree and the number of nodes searched, the algorithm is often unable to find enough nodes to eliminate. Table 1 shows the number of iterations when we only allow nodes with degree at most 4 and at most 1500 larger than the minimum and only touch $4p$ nodes, where p is the number of processors. The table clearly shows that a tight tolerance increases the number of iterations, which implies that during many iterations some processors had no nodes to eliminate.

To summarize, it is usually possible to find large sets S, but at a computational cost and a potential degradation of the output. Selecting large independent sets increases the work required during this phase of the algorithm; the increased work is particularly significant because this phase is sequential. The ordering computed by the algorithm may degrade when we select large sets S; by allowing nodes with high degrees to be selected we significantly alter the ordering from the ordering returned by serial column minimum degree codes.

4.4 Load Imbalance

We identified two possible sources of load imbalance in the algorithm. First, after all the nodes are removed from the work queue S and until all these nodes are eliminated, the algorithm cannot proceed to the next iteration. As processors complete their last elimination for the iteration during this period and find the work queue idle, they stop performing useful work and block until the end of the iteration. If S contains many nodes, say 100 times the number of processors, we do not expect these idle times at the end of the iteration to slow down the algorithm considerably. But if S contains few nodes, especially if S contains fewer nodes than processors, then the idle times are expected to be significant.

Another source of imbalance is related to the cost of each elimination. Specifically, some nodes require more work to eliminate than others. If each processor eliminates only one or a few nodes in each iteration, or if the differences in the cost of elimination between nodes are large, then this source of imbalance may have a significant impact on the overall performance of the algorithm.

As explained above, we address this issue by selecting large sets of nodes to eliminate in every iteration. For many matrices, this leads to good load balancing. For example, on one matrix (av41092, $k = 128$), processors spend only about 2% of the running time waiting at the end of iterations when two processors are used, and about 15% of the time when 8 processors are used. These numbers are highly matrix dependent; on another matrix (memplus, $k = 128$), when 8 processors are used they spend up to 37% of the running time waiting at the end of iterations. We have found one case (the pre2 matrix) in which a single elimination took about 30% of the running time on one processor (about 300 out of 1000 seconds).

To summarize, the relative cost of different eliminations is highly matrix dependent. On some matrices the differences do not cause significant load imbalance, but on other matrices this source of imbalance essentially prevents the code from running efficiently in parallel.

4.5 Contention

As noted in the discussion regarding Ahmdahl's law, choosing nodes that are only distance 2 apart requires more synchronization. Our code normally blocks when it tries to acquire a lock and the lock is held by another processor. We measure contention for locks using a nonblocking call to try to acquire the lock. If the lock is not available, the code increments a counter that counts contention and then blocks waiting for the lock. We selectively measured contention in different phases of the algorithm and for different data structures in order to better assess the cost incurred by contention for locks.

We initially found the amount of contention for both the degree list and node array surprisingly high. For the memplus and shyy161 matrices, 3–4% of the attempts to acquire degree-list locks block due to contention with 4 processors, and only slightly less with 2 and 8 processors.

To overcome this significant contention, we separated the updating of the data structures into a local and global phase. In the local phase all changes are marked in a local private array with no locking; in the global phase all local arrays are used to update the global data structures.

This improvement typically reduces contention to between 0.1-0.5% of all attempts to acquire locks.

4.6 High Overheads for Parallel Programming Primitives

The code uses two barriers in each iteration. One guarantees that the node degrees are updated prior to selecting a new set of nodes to eliminate, the other guarantees the independent set is found before processors attempt to eliminate nodes. The code also locks data structure to maintain them correctly. Clearly barriers, locks, and unlocks need to be efficient.

We first coded the algorithm using Sun threads and POSIX threads primitives, including mutex variables and condition variables (we used condition variables only to implement global barriers). We found the cost of these primitives high on both Sun machines running Solaris and SGI machines running Irix. We suspect that the implementors of these primitives used system calls to avoid busy waiting. Our first barrier code was a code published by Sun [13], which uses both mutex variables and condition variables to implement a barrier.

The locking in our code is fairly fine grained. The code acquires and releases locks frequently in some phases of the algorithm and spends little time inside critical sections. Contention is infrequent. Hence, spinning on locks is a reasonable strategy, since busy waiting is normally absent or short, and making numerous system calls slows down our code and is unlikely improve the utilization of the system as a whole.

Therefore, we replaced the implementation of the mutex primitives and the barriers with one that uses atomic memory operations that are provided by the SGI C compiler (the operations we use are fetch-and-subtract, lock-test-and-set, and lock-release). This version of the code only works on SGI machines. Our measurements show this implementation of the primitives is fast. On a single processor, a code that calls the synchronization primitives and one that does not call them run at the same speed. This implies the primitives are essentially free when there is no waiting, at least at the rate that our code uses these primitives. The barriers that use the atomic memory operations are about 20 times faster than the barriers that use the POSIX primitives.

We also remark that the number of barriers is reduced when the algorithm finds larger independent sets since the number of iterations is reduced. This, however, has other implications that were explained above, so fast barriers are still useful. Also, the number of mutex lock operations cannot be reduced in the same way.

4.7 Cache Misses

On SMPs it is significantly faster for a processor to fetch data from its own cache rather than getting the data from main memory or from the caches of other processors. In other words, exploiting locality is important.

Our algorithm does not attempt to have processors reuse data already in their caches. Ideally, after a processor touches data for various cliques and nodes in eliminating a node, the same processor would next eliminate a nearby node and possibly reuse some of the data already in the cache from the previous elimination. However, we do not know how to achieve this goal within the framework of our algorithm, nor do we know how to otherwise enhance data reuse.

We suspect that cache misses are a significant source of inefficiency. On some matrices, despite not detecting significant contention, load imbalance, sequential bottlenecks, or other parallel overheads, the algorithm still does not speed up with more processors. We conclude that the cost of memory accesses increase with more processors due to cache misses.

In particular, we have observed that the running time of the preprocessing step in which identical rows and columns are identified actually increases with the number of processors. During this phase, the amount of work is independent of the number of processors, there is no locking (except for a few barriers), so in theory this portion of the code should benefit from additional processors. In practice this portion of the code slows down, leading us to believe that cache misses are a significant cost in the code.

We note that cache misses are not only caused by data used first on one processor and then on another, or by data that is evicted due to cache conflicts. On a multiprocessor, maintaining the coherency of caches also causes cache misses. In machines with weak coherency, such as the machines we use, frequent synchronization can lead to cache misses on data that is not actually communicated between processors.

5 Summary of Experimental Results

We now summarize our experimental results. The results reported in Table 2 are for runs with no limit on the number of nodes touched and no limit on the degrees of nodes included in the independent sets. These parameters maximize parallelism at the possible expense of the quality of the ordering. Furthermore, because the parallel preprocessing step was found to slow down as processors were added, for these results we did the preprocessing step serially.

For each matrix, we report two sequential running times: one with a $k = 1$, and the other with $k = 128$. The first case implements a conventional sequential approximate minimum degree algorithm (without multiple eliminations). The second case performs a computation similar to that performed in parallel runs, hence allowing us to compare running times. We also report runs with 2, 4, and 8 processors, all using the same target size k of 128 for S. (We thus expect the load balance to degrade with the number of processors; but increasing k to maintain load balance would prevent us from comparing running times because the computations would be different.)

The runs that we report were performed on a 12-processor SGI Power Challenge with 196Mhz R10000/R10010 processors, 32Kbytes primary data caches, 1MBytes secondary unified caches, and 1024Mbytes of 4-way interleaved main memory. The machine was lightly loaded but not idle when we ran the experiments; p CPU's were always available during runs with p threads.

6 Conclusions

Our main conclusion is that column minimum degree algorithms are difficult to parallelize. When we started this project we were aware that previous unpublished attempts to parallelize the minimum degree algorithm on distributed-memory parallel computers were unsuccessful. Therefore, we decided to concentrate on shared memory machines, hoping that faster interprocessor communication would enable us to obtain significant speedups. The problem was more difficult than we expected.

Still, the code sometimes speeds up significantly with more processors, and it rarely slows down. For example, on one matrix (av41092) the code speeds up by almost a factor of 3 with 8 processors. Therefore, our current suggestion is to use the parallel algorithm

Table 2

This table shows the number of iterations, the total time T, the preprocessing time T_{pre}, and the time to select the independent set T_{is} as a function of the number p of processors and the target size k of the independent sets.

matrix	p	k	# *its*	T	$T_{\rm pre}$	$T_{\rm is}$
av41092	1	1	14416	30.10	2.72	0.22
	1	128	124	29.67	2.73	0.37
	2	128	123	18.81	2.79	0.37
	4	128	124	12.96	2.75	0.40
	8	128	123	10.44	2.75	0.41
memplus	1	1	8186	9.55	0.61	0.12
	1	128	259	12.06	0.55	1.61
	2	128	260	9.62	0.56	1.23
	4	128	258	8.58	0.56	1.44
	8	128	232	7.58	0.56	1.29
rim	1	1	3947	10.19	1.10	0.12
	1	128	623	13.08	1.11	2.75
	2	128	404	9.38	1.12	2.31
	4	128	395	9.41	1.12	2.64
	8	128	528	11.51	1.12	3.53
raefsky4	1	1	2997	3.96	0.88	0.03
	1	128	50	2.70	0.93	0.24
	2	128	51	2.42	0.88	0.30
	4	128	51	2.31	0.88	0.33
	8	128	51	2.50	0.88	0.37
shyy161	1	1	37207	7.01	1.89	0.22
	1	128	285	7.24	1.89	0.43
	2	128	284	7.81	1.85	0.48
	4	128	285	7.18	1.91	0.50
	8	128	283	7.37	1.91	0.58
onetone1	1	1	21356	6.40	1.31	0.09
	1	128	178	6.51	1.33	0.25
	2	128	181	6.29	1.34	0.31
twotone	1	1	53658	60.91	5.06	1.51
	1	128	431	63.61	4.95	2.94
	2	128	432	62.97	5.05	2.94

when several processors are available and the user is seeking to reduce the solution time. On the other hand, on a heavily loaded system in which the overall utilization is important (and not only speed), running a sequential code makes more sense.

There are still unresolved issues concerning the current code. We have not made substantial tests comparing the quality of the orderings, especially with relaxed independent set parameters, with the quality of the orderings produced by sequential codes. We have made only a few tests that indicated that the quality of the orderings is reasonable and not particularly sensitive to these parameters, but this must be checked thoroughly before the algorithm is put into use.

While it is possible that this algorithm can be parallelized in a way that speeds up significantly with more processors, our experience leads us to believe exploiting the fine-grained parallelism inherent in this algorithm is difficult.

References

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17(4) (Oct. 1996), pp. 886-905.

[2] C. Ashcraft, *Compressed graphs and the minimum degree algorithm*, SIAM J. Sci. Comput., 16(6) (Nov. 1995), pp. 1404-1411.

[3] C. Ashcraft and J. W. H. Liu, *Robust ordering of sparse matrices using multisection*, SIAM J. Matrix Anal. Appl., 19(3) (Jul. 1998), pp. 816-832.

[4] T. Davis, *University of Florida sparse matrix collection*, NA Digest, v.92, n.42, Oct. 16, 1994 and NA Digest, v.96, n.28, Jul. 23, 1996, and NA Digest, v.97, n.23, Jun. 7, 1997. available at: http://www.cise.ufl.edu/~davis/sparse/.

[5] A. George and J. W. H. Liu, *A fast implementation of the minimum degree algorithm using quotient graphs*, ACM Trans. Math. Softw., 6(3) (Sep. 1980), pp 337-358.

[6] A. George and J. W. H. Liu, *The evolution of the minimum degree ordering algorithm*, SIAM Review, 31(1) (Mar. 1989), pp 1-19.

[7] J. R. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in Matlab: design and implementation*, SIAM J. Matrix Anal. Appl., 13(1) (Jan. 1992), pp 333-356.

[8] J. R. Gilbert and E. G. Ng, *Predicting structure in nonsymmetric sparse matrix factorizations*, in Graph Theory and Sparse Matrix Computation, Springer-Verlag, 1993.

[9] B. Hendrickson and E. Rothberg, *Effective sparse matrix ordering: just around the BEND*, in Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[10] Xiaoye S. Li, J. Demmel and J. Gilbert, *An asynchronous parallel supernodal algorithm for sparse Gaussian elimination*, SIAM Journal on Matrix Analysis and Applications, to appear. Previously published as UC Berkeley Tech Report CSD-97-943, February 1997, revised September, 1997.

[11] J. W. H. Liu, *Modification of the minimum-degree algorithm by multiple elimination*, ACM Trans. Math. Softw., 11(2) (Jun. 1985), pp. 141-153.

[12] J. W. H. Liu, *The minimum degree ordering with constraints*, SIAM J. Sci. Stat. Comput., 10(5) (Nov. 1989), pp. 1136-1145.

[13] R. Marejka, *A barrier for Threads*, SunOpsis, 4(1) (Jan.-Mar. 1995).

[14] E. Rothberg, *Ordering sparse matrices using approximate minimum local fill*, Silicon Graphics manuscript (Apr. 1996).

[15] E. Rothberg and S. C. Eisenstat, *Node selection strategies for bottom-up sparse matrix ordering*, SIAM J. Mat. Anal. Appl., 19(3) (1998), pp. 682-695.

[16] W. F. Tinney and J. W. Walker, *Direct solutions of sparse network equations by optimally ordered triangular factorization*, J. of the Proc. of the IEEE, 55 (1967), pp. 1801-1809.