A Scalable Sparse Direct Solver Using Static Pivoting

Xiaoye S. Li *　　James W. Demmel †

Abstract

We propose several techniques as alternatives to partial pivoting to stabilize sparse Gaussian elimination. From numerical experiments we demonstrate that for a wide range of problems the new method is as stable as partial pivoting. The main advantage of the new method over partial pivoting is that it permits *a priori* determination of data structures and communication pattern, which makes it more scalable. We demonstrate the scalability of our algorithms on large-scale distributed memory computers.

1 Introduction

Traditionally, partial pivoting is used as an effective mechanism to control the element growth during Gaussian elimination on general matrices, thereby stabilizing the underlying algorithm. In our earlier work we developed efficient algorithms and software to perform Gaussian elimination with partial pivoting (GEPP) [3, 4, 5, 9]. Since the computational graph does not unfold until runtime due to partial pivoting, our shared memory parallel GEPP algorithm uses a centralized task queue for dynamic scheduling and load balancing. However, this is too expensive on distributed memory machines. Instead, for distributed memory machines, we propose to *not pivot dynamically*, and so enable static data structure optimization, graph manipulation and load balancing (as with Cholesky) and yet *remain numerically stable*.

2 GESP algorithm and stability

We considered a variety of techniques as alternatives to partial pivoting to maintain stability, such as pre-pivoting large elements to the diagonal, iterative refinement, using extra precision when needed, and allowing low rank modifications with corrections at the end. In this paper we show by experiments that even a subset of them can effectively stabilize the algorithm, and the costs associated with them are usually small. Figure 1 sketches our GESP algorithm (Gaussian elimination with static pivoting) that incorporates some of these techniques. In step (1), the diagonal scale matrices D_r and D_r are chosen independently, to make each row and each column of $D_r A D_c$ have largest entries equal to 1 in magnitude. Finding a permutation P_r that puts large entries on the diagonal can be transformed into a weighted bipartite matching problem. There is a large body of

*NERSC, Lawrence Berkeley National Lab 1 Cyclotron Rd, MS 50F, Berkeley, CA 94720 (xiaoye@nersc.gov). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Energy Research of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

†Computer Science Division, University of California, Berkeley, CA 94720 (demmel@cs.berkeley.edu). This research was supported in part by NSF grant ASC-9313958, DOE grant DE-FG03-94ER25219, UT Subcontract No. ORA4466 from ARPA Contract No. DAAL03-91-C0047, DOE grant DE-FG03-94ER25206, NSF Infrastructure grants CDA-8722788 and CDA-9401156 and DOE grant DE-FC03-98ER25351.

(1) Row/column equilibration and row permutation: $A \leftarrow P_r \cdot D_r \cdot A \cdot D_c$,
where D_r and D_c are diagonal matrices and P_r is a row permutation
chosen to make the diagonal large compared to the off-diagonal

(2) Find a column permutation P_c to preserve sparsity: $A \leftarrow P_c \cdot A \cdot P_c^T$

(3) Factorize $A = L \cdot U$ with control of diagonal magnitude

  if ($|a_{ii}| < \sqrt{\varepsilon} \cdot ||A||$) **then**

    set a_{ii} to $\sqrt{\varepsilon} \cdot ||A||$

  endif

(4) Solve $A \cdot x = b$ using the L and U factors, with the following iterative refinement

  iterate:

    $r = b - A \cdot x$ ... sparse matrix-vector multiply

    Solve $A \cdot dx = r$ ... triangular solve

    $berr = \max_i \frac{|r|_i}{(|A| \cdot |x| + |b|)_i}$ ... componentwise backward error

    if ($berr > \varepsilon$ and $berr \leq \frac{1}{2} \cdot lastberr$) **then**

      $x = x + dx$

      $lastberr = berr$

      goto iterate

    endif

FIG. 1. *The outline of the new GESP algorithm.*

literature on efficient algorithms to solve this problem. We experimented the algorithms by Duff and Koster [6] that choose P_r to maximize different properties of the diagonal of $P_r D_r A D_c$, such as the smallest magnitude of any diagonal entry, or the sum or product of magnitudes. But the best algorithm in practice seems to be the one that picks P_r, D_r and D_c simultaneously so that each diagonal entry of $P_r D_r A D_c$ is ± 1, each off-diagonal entry is bounded by 1 in magnitude, and the product of the diagonal entries is maximized. The column permutation P_c in step (2) can be obtained from any fill-reducing heuristic, such as minimum degree or nested dissection. In step (3), we simply set any tiny pivots encountered during elimination to $\sqrt{\varepsilon} \cdot \|A\|$, where ε is machine precision. This is equivalent to a small (half precision) perturbation to the original problem, and trades off some numerical stability for the ability to keep pivots from getting too small. In step (4), we perform a few steps of iterative refinement if the solution is not accurate enough, which also corrects for the $\sqrt{\varepsilon} \cdot \|A\|$ perturbations in step (3). The termination criterion is based on the componentwise backward error $berr$ [1].

We tested the GESP algorithm on 53 unsymmetric matrices from a wide range of applications. Among them, 22 matrices contain zeros on the diagonal to begin with which remain zero during elimination, and 5 more create zeros on the diagonal during elimination. Therefore, not pivoting at all would fail completely on these 27 matrices. Most of the other 26 matrices would get unacceptably large errors due to pivot growth. For most matrices, the iterative refinement terminates with no more than 3 steps; 5 matrices require 1 step, 31 matrices require 2 steps, 9 matrices require 3 steps, and 8 matrices require more than 3 steps. In Figure 2, for each matrix, we plot the error of the computed solution from GESP versus the error from GEPP (as implemented in SuperLU, also with iterative refinement). The error of GESP is at most a little larger, and can be smaller (21 out of 53) than the error from GEPP. Figure 3 shows that the componentwise backward errors are also small, usually near machine epsilon ($\approx 10^{-16}$ in IEEE double precision), and never larger than

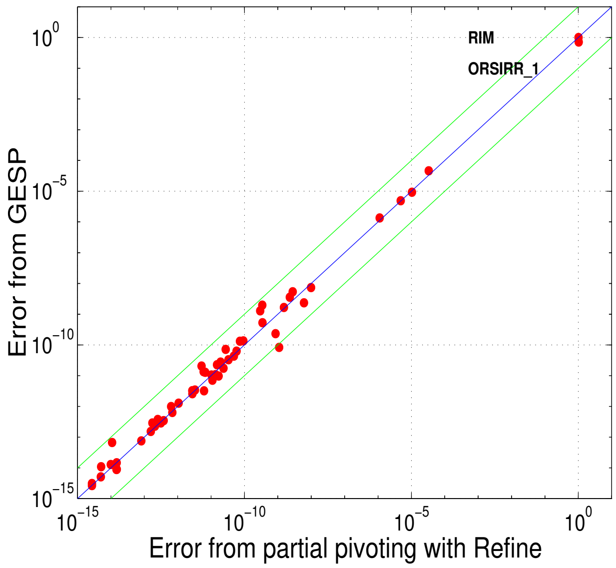

FIG. 2. *The error* $\frac{||x_{true}-x||_\infty}{||x||_\infty}$.

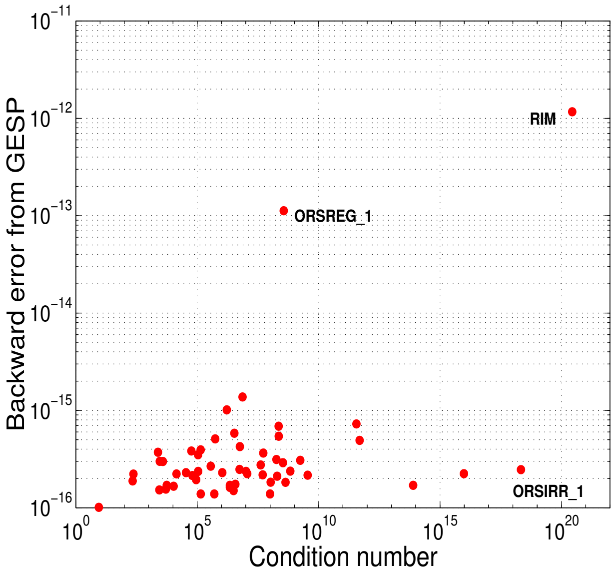

FIG. 3. *The backward error* $\max_i \frac{|A \cdot x - b|_i}{(|A| \cdot |x| + |b|)_i}$.

10^{-12}.

We now evaluate the cost of each step in GESP. This is done with respect to the serial implementation, since we have only parallelized the numerical phases of the algorithm (steps (3) and (4)), which are the most time-consuming. In particular, for large enough matrices, the LU factorization in step (3) dominates all the other steps, so we will measure the times of each step with respect to step (3). Simple equilibration in step (1) (computing D_r and D_c using the algorithm in DGEEQU from LAPACK) is usually negligible and is easy to parallelize. Both row and column permutation algorithms in steps (1) and (2) (computing P_r and P_c) are not easy to parallelize (their parallelization is future work). Fortunately, their memory requirement is just $O(nnz(A))$ [2, 6], whereas the memory requirement for L and U factors grows superlinearly in $nnz(A)$, so in the meantime we can run them on a single processor. Figure 4 shows the fraction of time spent finding P_r in step (1) using the algorithm in [6], as a fraction of the factorization time. The time is significant for small problems, but drops to 1% to 10% for large matrices requiring a long time to factor, the problems of most interest on parallel machines. The time to find a sparsity-preserving ordering P_c in step (2) is very much matrix dependent. It is usually cheaper than factorization, although there exist matrices for which the ordering is more expensive. Nevertheless, in applications where we repeatedly solve a system of equations with the same nonzero pattern but different values, the ordering algorithm needs to be run only once, and its cost can be amortized over all the factorizations. Computing the residual (sparse matrix-vector multiplication $r = b - A \cdot x$) is cheaper than a triangular solve ($A \cdot dx = r$), and both take a small fraction of the factorization time. For large matrices the solve time is often less than 5% of the factorization time. Both algorithms have been parallelized Finally, our code has the ability to estimate a forward error bound for the true error $\frac{||x_{true}-x||_\infty}{||x||_\infty}$. This is by far the most expensive step after factorization. (For small matrices, it can be more expensive than factorization, since it requires multiple triangular solves.) Therefore, we will do this only when the user asks for it.

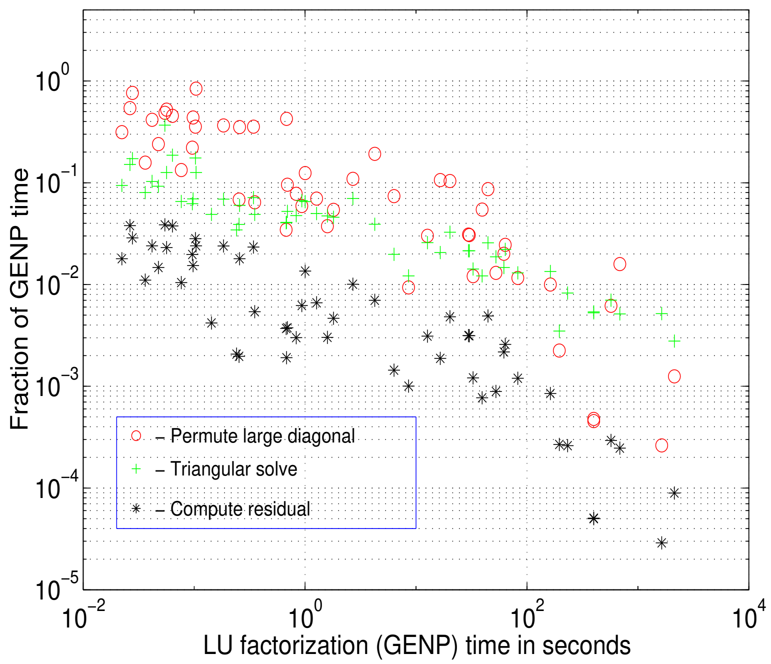

FIG. 4. *The times to factorize, solve, permute large diagonal, compute residual, on a 195 MHz MIPS R10000.*

	Order	$nnz(A)$	NumSym	StrSym	$nnz(L+U-I)$ $(\times 10^6)$	Flops $(\times 10^9)$
AF23560	23560	460598	.0512	.9465	12.8	4.9
BBMAT	38744	1771722	.0224	.5398	49.1	4.3
ECL32	51993	380415	.6572	.9325	73.5	120.4
EX11	16614	1096948	.9999	1.0000	14.1	8.4
FIDAPM11	22294	623554	.5476	.9965	23.0	17.9
RMA10	46835	2374001	.2443	.9809	14.7	1.8
TWOTONE	120750	1224224	.1418	.2738	22.6	8.7
WANG4	26068	177196	.1868	1.0000	27.7	35.3

TABLE 1

Characteristics of the test matrices. NumSym is the fraction of nonzeros matched by equal values in symmetric locations. StrSym is the fraction of nonzeros matched by nonzeros in symmetric locations.

3 A scalable and portable implementation using MPI

In this section, we describe our design, implementation and the performance of the distributed algorithms for two main steps of the GESP method, sparse LU factorization and sparse triangular solve. Our implementation uses MPI [11] to communicate data, and so is highly portable. We have tested the code on a number of platforms, such as Cray T3E, IBM SP2, and Berkeley NOW. Here, we only report the results from a 512 node Cray T3E-900 at NERSC. To illustrate scalability of the algorithms, we restrict our attention to eight relatively large matrices selected from our testbed. They are representative of different application domains. The characteristics of these matrices are given in Table 1.

3.1 Matrix distribution and distributed data structure

We distribute the matrix in a two-dimensional block-cyclic fashion. In this distribution, the P processes (not restricted to be a power of 2) are arranged as a 2-D process grid of shape $P_r \times P_c$. The matrix is decomposed into blocks of submatrices. Then, these blocks are cyclically mapped onto the process grid, in both row and column dimensions. Such a 2-D layout strikes a good balance among locality (by blocking), load balance (by cyclic mapping), and lower communication volume (by 2-D mapping). 2-D layouts were used in scalable implementations of sparse Cholesky factorization [8, 10].

The matrix partitioning is based on the notion of *unsymmetric supernode* first introduced in [3]. Let L be the lower triangular matrix in the LU factorization. A supernode is a range $(r : s)$ of columns of L with the triangular block just below the diagonal being full, and with the same row structure below this block. Because of the identical row structure of a supernode, it can be stored in a dense format in memory. This supernode partition is used as our block partition in both row and column dimensions. If there are N supernodes in an n-by-n matrix, the matrix will be partitioned into N^2 blocks of nonuniform size. The size of each block is matrix dependent. It should be clear that all the diagonal blocks are square and full (we store zeros from U in the upper triangle of the diagonal block), whereas the off-diagonal blocks may be rectangular and may not be full. The matrix in Figure 5 illustrates such a partitioning. By block-cyclic mapping we mean block (I, J) $(0 \le I, J \le N-1)$ is mapped onto the process at coordinate (I mod P_r, J mod P_c) of the process grid. Using this mapping, a block $L(I, J)$ in the factorization is only needed by the row of processes that own blocks in row I. Similarly, a block $U(I, J)$ is only needed by the column of processes that own blocks in column J.

In this 2-D mapping, each block column of L resides on more than one process, namely, a column of processes. For example in Figure 5, the k-th block column of L resides on the column processes $\{0, 3\}$. Process 3 only owns two nonzero blocks, which are not contiguous in the global matrix. The schema on the right of Figure 5 depicts the data structure to store the nonzero blocks on a process. Besides the numerical values stored in a Fortran-style array `nzval[]` in column major order, we need the information to interpret the location and row subscript of each nonzero. This is stored in an integer array `index[]`, which includes the information for the whole block column and for each individual block in it. A process owns $\lceil N/P_c \rceil$ block columns of L, so it needs $\lceil N/P_c \rceil$ pairs of `index/nzval` arrays. Similarly, each process has $\lceil N/P_r \rceil$ pairs of `index/nzval` arrays to store block rows of matrix U.

3.2 Sparse LU factorization

Figure 6 outlines the parallel sparse LU factorization algorithm. We use Matlab notation for integer ranges and submatrices. There are three steps in the K-th iteration of the loop. In step (1), only a column of processes participate in factoring the block column $L(K : N, K)$. In step (2), only a row of processes participate in the triangular solves to obtain the block row $U(K, K+1 : N)$. The rank-b update by $L(K+1 : N, K)$ and $U(K, K+1 : N)$ in step (3) represents most of the work and also exhibits more parallelism than the other two steps, where b is the block size of the K-th block column/row. For ease of understanding, the algorithm presented here is simplified. The actual implementation uses a *pipelined* organization so that processes $PROC_C(K+1)$ will start step (1) of iteration $K+1$ as soon as the rank-b update (step (3)) of iteration K to block column $K+1$ finishes, before completing the update to the trailing matrix $A(K+1 : N, K+2 : N)$ owned by $PROC_C(K+1)$. The pipelining alleviates the lack of parallelism in both steps (1) and (2).

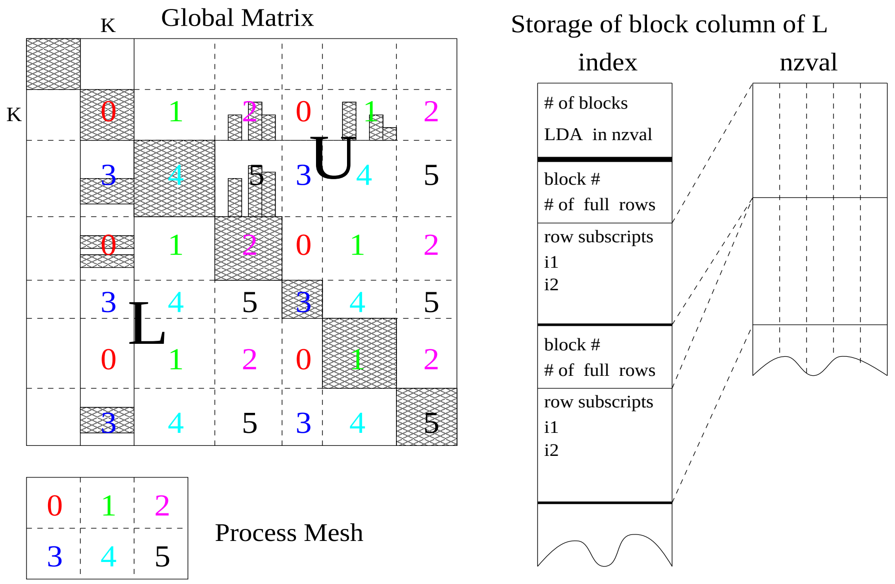

FIG. 5. *The 2-D block-cyclic layout and the data structure to store a local block column of L.*

Let $mycol$ ($myrow$) be my process column (row) number in the process grid
Let $PROC_C(K)$ ($PROC_R(K)$) be the column (row) processes that own block column (row) K
for block $K = 1$ **to** N **do**
    (1) **if** ($mycol = PROC_C(K)$)
        Obtain the block column factor $L(K:N, K)$
        Send $L(K:N, K)$ to the processes in my row who need it
    else
        Receive $L(K:N, K)$ from processes $PROC_C(K)$ if I need it
    endif
    (2) **if** ($myrow = PROC_R(K)$)
        Perform parallel triangular solves : $U(K, K+1:N) = L(K,K)^{-1} \cdot A(K, K+1:N)$
        Send $U(K, K+1:N)$ to processes in my column who need it
    else
        Receive $U(K, K+1:N)$ from processes $PROC_R(K)$ if I need it
    endif
    (3) **for** $J = K+1$ to N **do**
        for $I = K+1$ to N **do**
            if ($myrow = PROC_R(I)$ & $mycol = PROC_C(J)$ & $L(I,K) \neq 0$ & $U(K,J) \neq 0$)
                $A(I,J) = A(I,J) - L(I,K) \cdot U(K,J)$
            endif
end for

FIG. 6. *Distributed sparse LU factorization algorithm.*

On 64 processors of Cray T3E, for instance, we observed speedups between 10% to 40% over the non-pipelined implementation.

In each iteration, the major communication steps are send/receive $L(K:N,K)$ across process rows and send/receive $U(K,K+1:N)$ down process columns. Our data structure (see Figure 5) ensures that all the blocks of $L(K:N,K)$ and $U(K,K+1:N)$ on a process are contiguous in memory, thereby eliminating the need for packing and unpacking in a send-receive operation or sending many more smaller messages. In each send-receive pair, two messages are exchanged, one for `index[]` and another for `nzval[]`. To further reduce the amount of communication, we employ the notion of *elimination dags* (EDAGs) [7]. That is, we send the K-th column of L rowwise to the process owning the J-th column of L only if there exists a path between (super)nodes K and J in the elimination dags. This is done similarly for the columnwise communication of rows of U. Therefore, each block in L may be sent to fewer than P_c processes and each block in U may be sent to fewer than P_r processes. In other words, our communication takes into account the sparsity of the factors as opposed to "send-to-all" approach in a dense factorization. For example, for AF23560 on 32 (4×8) processes, the total number of messages is reduced from 351052 to 302570, or 16% fewer messages. The reduction is even more with more processes or sparser problems.

3.3 Sparse triangular solve

The sparse lower and upper triangular solves are also designed around the same distributed data structure. The forward substitution proceeds from the bottom of the elimination tree to the root, whereas the back substitution proceeds from the root to the bottom. Figure 7 outlines the algorithm for sparse lower triangular solve. The algorithm is based on a sequential variant called "inner product" formulation. In this formulation, before the K-th subvector $x(K)$ is solved, the update from the inner product of $L(K,1:K-1)$ and $x(1:K-1)$ must be accumulated and subtracted from $b(K)$. The diagonal process, at the coordinate ($K \bmod P_r$, $K \bmod P_c$) of the process grid, is responsible for solving $x(K)$. Two counters, $frecv$ and $fmod$, are used to facilitate the asynchronous execution of different operations. $frecv[K]$ counts the number of process updates to $x(K)$ to be received by the diagonal process owning $x(K)$. This is needed because $L(K,1:K-1)$ is distributed among the row processes $PROC_R(K)$, and due to sparsity, not all processes in $PROC_R(K)$ contribute to the update. When $frecv(K)$ becomes zero, all the necessary updates to $x(K)$ are complete and $x(K)$ is solved. $fmod(K)$ counts the number of block modifications to be summed into the local inner product update (stored in $lsum(K)$) to $x(K)$. When $fmod(K)$ becomes zero, the partial sum $lsum(K)$ is sent to the diagonal process that owns $x(K)$.

The execution of the program is *message-driven*. A process may receive two types of messages, one is the partial sum $lsum(K)$, another is the solution subvector $x(K)$. Appropriate action is taken according to the message type. The asynchronous communication enables large overlapping between communication and computation. This is very important because the communication to computation ratio is much higher in triangular solve than in factorization.

3.4 Parallel performance

Table 2 shows the performance of the factorization on the Cray T3E-900. The symbolic analysis is not yet parallel, so we start with a copy of the entire matrix on each processor, and run symbolic analysis independently on each processor. Thus the time is independent of the number of processors. The first column of Table 2 reports the time spent in the

```
Let mycol (myrow) be my process column (row) number in the process grid
Let PROC_C(K) be the column processes that own block column K
x = b
lsum = 0
for each block K that I own                    ... Compute leaf nodes
    if ( myrow = K mod P_r & mycol = K mod P_c & frecv[K] = 0 )
        x(K) = L(K,K)^{-1} · x(K)
        Send x(K) to the column processes PROC_C(K)
    endif
end for
while ( I have more work ) do                  ... Compute internal nodes
    Receive a message
    if ( message is lsum(K) )
        x(K) = x(K) + lsum(K);
        frecv(K) = frecv(K) - 1
        if ( frecv(K) = 0 )
            x(K) = L(K,K)^{-1} · x(K)
            Send x(K) to the column processes PROC_C(K)
        endif
    else if ( message is x(K) )
        for each I > K, L(I,K) ≠ 0 that I own
            lsum(I) = lsum(I) - L(I,K) · x(K)
            fmod(I) = fmod(I) - 1
            if ( fmod(I) = 0 )
                Send lsum(I) to the diagonal process who owns L(I,I)
            endif
        end for
    endif
end while
```

FIG. 7. *Distributed lower triangular solve $L \cdot x = b$.*

symbolic analysis. The memory requirement of the symbolic analysis is small, because we only store and manipulate the *supernodal graph* of L and the *skeleton graph* of U, which are much smaller than the graphs of L and U. The subsequent columns in the table show the factorization time with a varying number of processors. For four large matrices (BBMAT, ECL32, FIDAPM11 and WANG4), the factorization time continues decreasing up to 512 processors, demonstrating excellent scalability. The last column reports the numeric factorization rate in Mflops. More than 8 Gflops is achieved for matrix ECL32.

As a reference, we compare our distributed memory code to our shared memory SuperLU_MT code using small numbers of processors. For example, using 4 processor DEC AlphaServer 8400 (SMP) [1], the factorization times of SuperLU_MT for matrices AF23560 and EX11 are 19 and 23 seconds, respectively, comparable to the 4-processor T3E timings. This indicates that our distributed data structure and message passing algorithm do not incur much overhead.

Table 3 shows the performance of the lower and upper triangular solves altogether.

[1] Each processor is the same as one T3E processor, except there is a 4 MB tertiary cache.

	Symbolic	Numeric							
		P=4	16	32	64	128	256	512	Mflops
AF23560	1.7	32.2	11.0	7.3	5.9	**5.8**	7.0	7.1	856
BBMAT	11.8	430.8	110.5	62.3	35.6	25.8	18.4	**17.0**	2493
ECL32	14.0	325.0	92.3	52.2	30.8	21.5	15.7	**14.3**	8352
EX11	1.8	20.6	6.7	4.3	3.3	**3.2**	3.4	3.4	2628
FIDAPM11	4.1	115.1	31.9	19.0	11.7	8.5	7.6	**7.1**	2291
RMA10	1.6	13.6	7.0	5.4	6.0	**5.4**	6.5	7.2	511
TWOTONE	6.6	99.8	48.1	31.6	29.4	**28.5**	29.9	31.7	297
WANG4	4.2	72.8	21.1	13.3	8.8	6.6	6.8	**6.5**	5542

TABLE 2

LU factorization time in seconds and Megaflop rate on the 512 node T3E-900.

	P=4	8	16	32	64	Mflops
AF23560	0.94	0.90	0.69	0.67	0.64	42
BBMAT	3.69	3.42	2.27	2.23	1.83	56
ECL32	2.95	2.60	1.66	1.57	1.17	128
EX11	0.50	0.46	0.32	0.31	0.26	112
FIDAPM11	1.39	1.26	0.83	0.83	0.68	70
RMA10	0.77	0.74	0.58	0.53	0.50	60
TWOTONE	4.37	4.37	3.65	3.15	2.95	16
WANG4	1.09	0.99	0.67	0.63	0.50	112

TABLE 3

Triangular solves time in seconds and Megaflop rate on the T3E-900.

When the number of processors continues increasing beyond 64, the communication takes more than 95% of the solve time, and the solve time remains roughly the same. Although triangular solves do not achieve high Megaflop rates, the time is usually much less than that for factorization.

4 Conclusions

We propose a number of techniques in place of partial pivoting to stabilize sparse Gaussian elimination. Their effectiveness is demonstrated by numerical experiments. These techniques enable static analysis of the nonzero structure of the factors and the communication pattern. As a result, a more scalable implementation becomes feasible on large-scale distributed memory machines with hundreds of processors. Our prototype software is being used in a quantum chemistry application at Lawrence Berkeley National Laboratory, where a complex unsymmetric system of order 736,164 has been solved within 20 minutes.

References

[1] M. Arioli, J. W. Demmel, and I. S. Duff, *Solving sparse linear systems with sparse backward error*, SIAM J. Matrix Anal. Appl., 10 (1989), pp. 165–190.

[2] T. A. Davis, J. R. Gilbert, E. Ng, and B. Peyton, *Approximate minimum degree ordering for unsymmetric matrices*. Talk presented at XIII Householder Symposium on Numerical Algebra, June 1996. Journal version in preparation.

[3] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. Liu, *A supernodal approach to sparse partial pivoting*, Tech. Rep. UCB//CSD-95-883, Computer Science Division, U.C. Berkeley, 1995. To appear in *SIAM J. Matrix Anal. Appl.*

[4] J. W. Demmel, J. R. Gilbert, and X. S. Li, *An asynchronous parallel supernodal algorithm for sparse gaussian elimination*, Tech. Rep. UCB//CSD-97-943, Computer Science Division, U.C. Berkeley, 1997. To appear in *SIAM J. Matrix Anal. Appl.*

[5] ——, *SuperLU and SuperLU_MT*, November 1997. http://www.netlib.org/scalapack/prototype/.

[6] I. S. Duff and J. Koster, *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices*, Tech. Rep. RAL-TR-97-059, Rutherford Appleton Laboratory, 1997.

[7] J. R. Gilbert and J. W. Liu, *Elimination structures for unsymmetric sparse LU factors*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 334–352.

[8] A. Gupta and V. Kumar, *Optimally scalable parallel sparse cholesky factorization*, in The 7th SIAM Conference on Parallel Processing for Scientific Computing, 1995, pp. 442–447.

[9] X. S. Li, *Sparse Gaussian elimination on high performance computers*, Tech. Rep. UCB//CSD-96-919, Computer Science Division, U.C. Berkeley, September 1996. Ph.D dissertation.

[10] E. E. Rothberg and A. Gupta, *An efficient block-oriented approach to parallel sparse cholesky factorization*, in Supercomputing, November 1993, pp. 503–512.

[11] *Message Passing Interface (MPI) forum*. http://www.mpi-forum.org/.