SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems

XIAOYE S. LI
Lawrence Berkeley National Laboratory
and
JAMES W. DEMMEL
University of California at Berkeley

We present the main algorithmic features in the software package SuperLU_DIST, a distributed-memory sparse direct solver for large sets of linear equations. We give in detail our parallelization strategies, with a focus on scalability issues, and demonstrate the software's parallel performance and scalability on current machines. The solver is based on sparse Gaussian elimination, with an innovative static pivoting strategy proposed earlier by the authors. The main advantage of static pivoting over classical partial pivoting is that it permits *a priori* determination of data structures and communication patterns, which lets us exploit techniques used in parallel sparse Cholesky algorithms to better parallelize both *LU* decomposition and triangular solution on large-scale distributed machines.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Sparse, structured, and very large systems (direct and iterative methods)*; G.4 [**Mathematical Software**]: Mathematical Software—*Parallel and vector implementations*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Sparse direct solver, supernodal factorization, parallelism, distributed-memory computers, scalability

1. INTRODUCTION

Parallelizing sparse direct solvers has been an active research area in the past decade. Our goal is to implement a sparse direct solver for nonsymmetric matrices as scalably as possible on distributed memory machines.

This work was supported in part by the National Energy Research Scientific Computing Center (NERSC), which is supported by the Director, Office of Advanced Scientific Computing Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098, and was supported in part by the National Science Foundation Cooperative Agreement No. ACI-9619020, NSF Grant No. ACI-9813362, and Department of Energy Grant Nos. DE-FG03-94ER25219 and DE-FC03-98ER25351, and UT Subcontract No. ORA4466 from ARPA Contract No. DAAL03-91-C0047.
Authors' addresses: X. S. Li, Lawrence Berkeley National Lab, MS 50F-1650, One Cyclotron Rd., Berkeley, CA 94720; email: xsli@lbl.gov; J. W. Demmel, Computer Science Division, University of California, Berkeley, Berkeley, CA 94720; email: demmel@cs.berkeley.edu.

It is important to say exactly what we mean by scalability, because it has some reasonable sounding but unachievable interpretations. For instance, if the n-by-n matrix equation to be solved arises from a differential equation like Laplace's equation, then we cannot aspire to achieve the $O(n)$ complexity of methods like multigrid. We also do not claim linear speedups for fixed problem sizes, since this depends so much on the particular sparse matrix structure. However, we do come close to linear speedups for constant-work-per-processor scaling on reasonable model problems (see Section 4.4).

More precisely, for us scalability will mean "as scalable as solving a symmetric positive definite (spd) linear system by a sparse direct method," or more briefly "as scalable as sparse Cholesky." The reason for this is that the nonsymmetric problem is strictly more difficult than the spd case, so that we cannot hope to do better in general. Our claim of scalability is based on our ability to use all the techniques exploited to parallelize sparse Cholesky (see below). The price we pay is a very small probability of numerical instability. We note that this numerical instability never occurred on our extensive test set for the default parameter settings of our code, and in any event is always detected and reported by the code.

The advantage of sparse Cholesky over the nonsymmetric case is that pivots can be chosen in any order from the main diagonal while guaranteeing stability. This lets us perform pivot choice before numerical factorization begins, in order to minimize fill-in, maximize parallelism, precompute the nonzero structure of the Cholesky factor, and optimize the two-dimensional(2D) distributed data structures and communication pattern. Researchers have been quite successful in achieving "scalable" performance for sparse Cholesky factorization; available codes include CAPSS [Heath and Raghavan 1997], MUMPS-SYM [Amestoy et al. 2001a], PaStix [Henon et al. 1999], PSLDLT [Rothberg 1996], and PSPACES [Gupta et al. 1997].

In contrast, for nonsymmetric or indefinite systems, few distributed-memory codes exist. They are more complicated than Choleksy for at least two reasons. First and foremost, some kind of numerical pivoting is necessary for stability. Classical partial pivoting [Golub and Van Loan 1996] or the sparse variant of threshold pivoting [Duff et al. 1986] typically cause the fill-ins and workload to be generated dynamically during factorization. Therefore, we must either design dynamic data structures and algorithms to accommodate these fill-ins [Amestoy et al. 2001a], or else use static data structures which can grossly overestimate the true fill-in [Fu et al. 1998; Gupta 2001]. The second complication is the need to handle two factored matrices L and U, which are structurally different yet closely related to each other in the filled pattern. Unlike the Cholesky factor whose minimum graph representation is a tree (called the *elimination tree*, or *etree* for short) [Liu 1990], the minimum graph representations of the L and U factors are directed acyclic graphs (called *elimination DAGs*, or *edags* for short) [Gilbert 1994; Gilbert and Liu 1993].

Despite these difficulties, researchers have been addressing these issues successfully for sequential and shared memory machines; available codes include MA41 [Amestoy and Duff 1993; Amestoy and Puglisi 2002], PARDISO [Schenk et al. 2000], SPOOLES [Ashcraft and Grimes 1999], SuperLU [Demmel et al.

1999a], SuperLU_MT [Demmel et al. 1999b], UMFPACK/MA38 [Davis and Duff 1999], and WSMP [Gupta 2000].

In our earlier codes SuperLU (serial) and SuperLU_MT (shared-memory), we devised efficient "symbolic" factorization algorithms to accommodate the dynamically generated fill-ins due to partial pivoting. The symbolic algorithm could not be decoupled from the numerical factorization; instead, it was interleaved with the numerical algorithm as the numerical factorization proceeded. These symbolic factorization algorithms are not suitable for distributed-memory machines, because they involve fine-grained memory access and synchronization to manage the data structures and identify task and data dependencies. This would generate large numbers of small messages.

Therefore, for SuperLU_DIST, which is targeted for large-scale distributed-memory machines, we use a static pivoting approach, called GESP (Gaussian elimination with static pivoting), proposed earlier by the authors [Li and Demmel 1998]. We parallelized the GESP algorithm using Message Passing Interface (MPI) [MPI n.d]. Our parallelization strategies center around the scalability concern. We use a 2D block-cyclic mapping of a sparse matrix to the processors, and designed an efficient pipelined algorithm to perform parallel factorization. With GESP, the parallel algorithm and code are much simpler than if we had to pivot dynamically. The main algorithmic features of SuperLU_DIST solver are summarized as follows:

—supernodal fan-out (right-looking) based on elimination directed asyclic graphs (DAGs),
—static pivoting with possible half-precision perturbations on the diagonal,
—use of an iterative algorithm using the LU factors as a preconditioner, in order to guarantee stability,
—static 2D irregular block-cyclic mapping using supernodal structure, and
—loosely synchronous scheduling with pipelining.

In particular, static pivoting can be performed before numerical factorization, allowing us to use all the techniques in good parallel sparse Cholesky codes: choice of a (symmetric) permutation to minimize fill-in and maximize parallelism, precomputation of the fill pattern, and optimization of 2D distributed data structures and communication patterns.

The rest of the paper is organized as follows. In Section 2 we demonstrate the numerical stability, the sequential runtime efficiency and the ordering schemes of the GESP algorithm. In Section 3, we present an MPI implementation of the distributed algorithms for LU factorization and triangular solutions. In Section 4, we present and analyze the parallel performance and scalability results. Section 5 describes the related work and compares SuperLU_DIST with some other solvers. Section 6 offers our concluding remarks and presents future work.

2. THE GESP ALGORITHM

Recall that the role of numerical pivoting is to avoid small pivots and control pivot growth in the factors. Dynamic pivoting is not the only means to achieve

(1) Perform 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) Perform symbolic analysis to determine the nonzero structures of L and U.
(4) Factorize $A = L \cdot U$ with control of diagonal magnitude:
    if ($|a_{ii}| < \sqrt{\varepsilon} \cdot \|A\|_1$) **then**
        set a_{ii} to $\sqrt{\varepsilon} \cdot \|A\|_1$
    endif
(5) Perform triangular solutions using L and U.
(6) If needed, use an iterative solver like GMRES or iterative refinement (shown below):
    iterate:
        $r = b - A \cdot x$ ... sparse matrix-vector multiply
        Solve $A \cdot dx = r$ ... triangular solution
        $berr = \max_i \frac{|r|_i}{(|A| \cdot |x| + |b|)_i}$ ... componentwise backward error
        if ($berr > \varepsilon$ and $berr \leq \frac{1}{2} \cdot lastberr$) **then**
            $x = x + dx$
            $lastberr = berr$
            goto iterate
        endif
(7) If desired, estimate the condition number of A.

Fig. 1. The outline of the GESP algorithm.

this goal. We can use other algorithms to prepermute large elements on the diagonal, thereby partially fulfilling the role of dynamic pivoting. Furthermore, when large pivot growth still occurs, there are inexpensive methods to tolerate and compensate for the growth, such as iterative methods preconditioned by the computed LU factors, of which GMRES [Saad and Schultz 1986] and iterative refinement are two examples. This observation led us to design a static pivoting factorization algorithm, called GESP [Li and Demmel 1998]. We demonstrated that GESP works well for practical matrices.

In our GESP algorithm, since pivots are chosen from the main diagonal, the fill-in positions can be determined before the numerical factorization, and so the symbolic factorization can be decoupled from numerical factorization. This enables static data structure optimization, graph manipulation, and load balancing in a similar way as parallel sparse Cholesky implementations.

Figure 1 sketches our GESP algorithm. To motivate step (1), recall that a *diagonally dominant matrix* is one where each diagonal entry a_{ii} is larger in magnitude than the sum of magnitudes of the off-diagonal entries in its row ($\sum_{j \neq i} |a_{ij}|$) or column ($\sum_{j \neq i} |a_{ji}|$). It is known that choosing diagonal pivots ensures stability for such matrices [Demmel 1997; Golub and Van Loan 1996]. We therefore expect that if each diagonal entry can somehow be made larger relative to the off-diagonals in its row or column, then diagonal pivoting will be more stable. The purpose of step (1) is to choose the diagonal scaling matrices D_r and D_c, and the permutation P_r to make each a_{ii} larger in this sense. We have experimented with a number of heuristic algorithms implemented in the routine `MC64` (available from HSL [HSL 2000]) [Duff and Koster 1999]. All depend on the following graph representation of an $n \times n$ sparse matrix A: it is represented as an undirected weighted bipartite graph with one vertex for each row, one vertex for each column, and an edge with appropriate weight connecting row vertex i to column vertex j for each nonzero entry a_{ij}. Finding a

permutation P_r that puts large entries on the diagonal can thus be transformed into a weighted bipartite matching problem on this graph. In MC64, there are algorithms that choose P_r to maximize different properties of the diagonal of $P_r A$, such as the smallest magnitude of any diagonal entry, or the sum or product of magnitudes. But the best algorithm in practice is the following (option 5 of MC64): it chooses P_r to maximize the product of the diagonal entries, and chooses D_r and D_c simultaneously so that each diagonal entry of $P_r D_r A D_c$ is ± 1, and each off-diagonal entry is bounded by 1 in magnitude. The implementation is based on the algorithm by Olshowka and Neumaier [1996]. We report results for this algorithm only. The worst-case serial complexity of this algorithm is $O(n \cdot nnz(A) \cdot \log n)$, where $nnz(A)$ is the number of nonzeros in A. In practice it is much faster; the actual timings appear later in Figure 7. In Section 5, we describe the work of others who experimented with this idea in the sparse direct and iterative solvers.

We note that the diagonal scalings D_r and D_c are needed in the algorithm so that (1) the value of $\|A\|_1$ in step (4) makes sense (see below) and (2) the estimated condition number from step (7) is not overly pessimistic when the rows and columns are badly scaled (i.e., D_r and D_c are far from multiplies of the identity). Indeed, in the absence of over/underflow, as long as the diagonal entries of D_r and D_c are chosen to be multiples of the radix (typically 2), and no small pivots are encountered in step (4) (see below), then identical rounding errors will be made in parts (4) through (6) of the algorithm whether or not D_r and D_c are applied to A in step (1).

Step (2) is standard in sparse direct solvers. The column permutation P_c can be obtained from any fill-reducing heuristic. In our code, we provide the minimum degree ordering algorithm [Liu 1985] on the structure of $A^T + A$. The code can also take as input an ordering based on some other algorithm, such as the nested dissection on $A^T + A$ [George 1973; Hendrickson and Leland 1993; Karypis and Kumar 1998]. Note that we also apply P_c to the rows of A to ensure that the large diagonal entries obtained from step (1) remain on the diagonal.

In step (4), we perform factorization using diagonal pivots. The tiny pivots encountered during elimination can be set to $\sqrt{\varepsilon} \cdot \|A\|_1$, 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 (6), we perform a few steps of an iterative method like iterative refinement (shown) or GMRES [Saad and Schultz 1986] if the solution from step (5) is not accurate enough. The termination criterion is based on the componentwise backward error *berr* [Arioli et al. 1989; Demmel 1997]. The condition $berr \leq \varepsilon$ means that the computed solution is the exact solution of a slightly different sparse linear system $(A + \delta A)x = b + \delta b$ where δA changes only each *nonzero* entry a_{ij} by at most one unit in its last place, and the zero entries are left unchanged; thus one can say that the answer is as accurate as the data deserves. We terminate the iteration when the backward error *berr* is smaller than machine epsilon, or when it does not decrease by at least a factor of 2 compared with the previous iteration. The latter test is to avoid possible stagnation. (Figure 5 shows that *berr* is always small.) Note that demanding

Table I. Test Matrices and Their Disciplines

Discipline	Matrices
Fluid flow, CFD	AF23560, BBMAT, BRAMLEY1, BRAMLEY2, EX11, EX19, FIDAP011, FIDAP019, FIDAPM11, FIDAPM29, GARON2, GOODWIN, GRAHAM1, INACCURA, INV-EXTRUSION-1, LNSP3937, LNS_3937, MIXING-TANK, RAEFSKY3, RAEFSKY4, RMA10, VENKAT01, WU
Circuit simulation	ADD32, GRE_1107, GRE_115, JPWH_991, MEMPLUS, ONETONE1, ONETONE2, TWOTONE
Device simulation	ECL32, WANG3, WANG4
Chemical engineering	EXTR1, HYDR1, LHR01, LHR71C, RADFR1, RDIST1, RDIST2, RDIST3A, WEST2021
Chemical process	BAYER01, BAYER02, BAYER04
Petroleum engineering	ORSREG_1, SAYLR4, SHERMAN3, SHERMAN4, SHERMAN5
Finite element PDE	AV4408, AV11924
MagnetoHydroDynamics	MHD500
Stiff ODE	FS_541_2
Olmstead flow model	OLM5000
Aeroelasticity	TOLS4000
Reservoir modelling	PORES_2
Crystal growth simulation	CRY10000
Power flow modelling	GEMAT11
Dielectric waveguide	DW8192 (eigenproblem)
Astrophysics	MCFE
Plasma physics	UTM5940
Demography	PSMIGR_1, PSMIGR_2, PSMIGR_3
Economics	MAHINDAS, ORANI678

Note: CFD = Computational fluid dynamics; PDE = partial differential equation; ODE = ordinary differential equation.

$berr \leq \varepsilon$ is very stringent and, in practice, the refinement can be terminated earlier.

When a small diagonal is encountered and set to $\sqrt{\varepsilon} \cdot \|A\|_1$, this may cause a large backward error in A, but this error is only large in norm, not in rank. In other words, the difference between A and the product of the computed factors $L \cdot U$ is small in rank. This makes the LU factorization an excellent preconditioner of A for a method like GMRES, which (in the absence of roundoff) takes no more steps to converge than the difference in rank between $L \cdot U$ and A. This will be borne out in the experiments below.

2.1 Numerical Stability

In this subsection, we illustrate the numerical stability and runtime of our GESP algorithm on 68 unsymmetric matrices drawn from a wide variety of applications. The application domains of the matrices are given in Table I. Most of them, except for wu, can be obtained from the Harwell-Boeing Collection [Duff et al. 1992] and the collection of Davis [Davis n.d.]. Matrix wu was provided by Yushu Wu from the Earth Sciences Division of Lawrence Berkeley National Laboratory. Figure 2 plots the dimension, $nnz(A)$, and $nnz(L + U)$ (i.e., the *fill-ins*, after the minimum degree ordering on $A^T + A$). The matrices are sorted in increasing order of the LU factorization time of the sequential GESP algorithm.

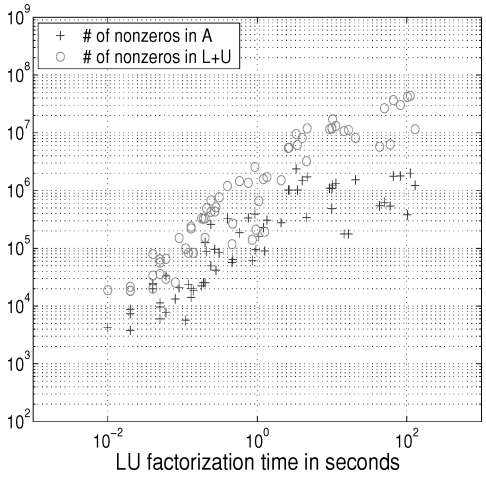

Fig. 2. Characteristics of the matrices.

The matrices of most interest for parallelization are the ones that take the most time, that is, the ones toward the right of the graph in Figure 2. It is clear that the matrices with larger numbers of nonzeros require more time to factorize. The timing results reported in this subsection are obtained on a single IBM 375-MHz POWER3 processor, running the AIX operating system. The processor has a 64-kB L1 data cache and an 8-MB L2 cache.

Detailed performance results from this section in tabular format are available at `http://www.nersc.gov/~xiaoye/SuperLU/GESP`.

As shown in Figure 1, our algorithm can be used in many "configurations":

—We may or may not perform step (1).
—One of many ordering schemes (nested dissection, minimum degree, etc.) may be used in step (2).
—We may or may not replace tiny pivots with $\sqrt{\varepsilon} \cdot \|A\|_1$ in step (4).
—We may apply several kinds of iteration (or none at all) in step (6).

In this section we report on several configurations of the algorithm. First, Figures 3 through 7 show data for the algorithm as shown in Figure 1, including the iterative refinement in step (6), which is often the fastest configuration. However, for a few matrices (see below) to get a stable solution it was important *not* to replace tiny pivots in step (4) (for other matrices it was important to replace tiny pivots as in step (4), and for most matrices it did not matter). So the data in Figures 3 through 7 actually reflects two possible configurations, depending on the matrix (replacing tiny pivots in step (4) or not).

Second, we ran all the matrices with the same configuration of the algorithm, in which restarted GMRES was used in step (6). All matrices were solved stably in this configuration, though it was sometimes slower than iterative refinement.

For the data reported in this section, we use minimum degree ordering on the structure of $A^T + A$.

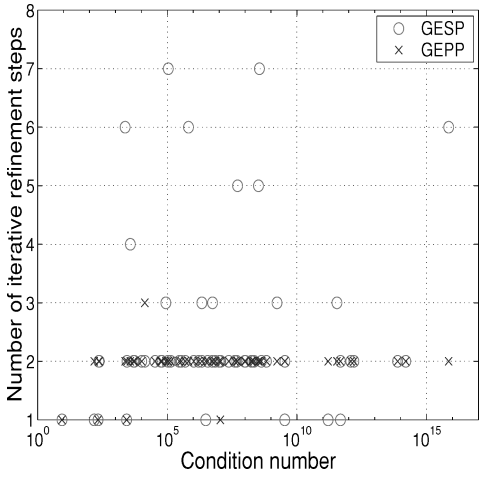

Fig. 3. Iterative refinement steps.

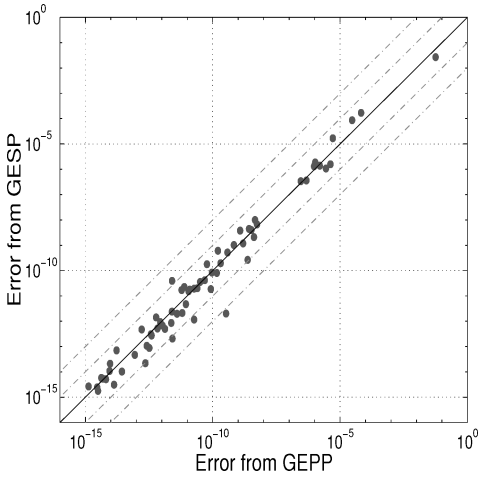

Fig. 4. Error $\frac{||x_{true}-x||_\infty}{||x||_\infty}$.

Now we consider the first configuration, when iterative refinement was used. Among the 68 matrices, many would get wrong answers or fail completely (via division by a zero pivot) without any pivoting or other precautions. In 26 of these matrices, some of the zeros present in the initial diagonal would continue to remain zero during elimination, and in another group of two matrices (bbmat and orsreg_1), new zeros would be created on the diagonal during elimination. Therefore, not pivoting at all would fail completely on these 29 matrices. For our experiment, the right-hand-side vector is generated so that the true solution x_{true} is a vector of all 1s. IEEE double precision is used as the working precision, with $\varepsilon \approx 10^{-16}$. All the test matrices have condition numbers bounded by $\frac{1}{\varepsilon}$. Figure 3 shows the number of iterations taken in the iterative refinement step. The termination criteria is that the backward error $berr = \max_i \frac{|r|_i}{(|A|\cdot|x|+|b|)_i} \leq \varepsilon$

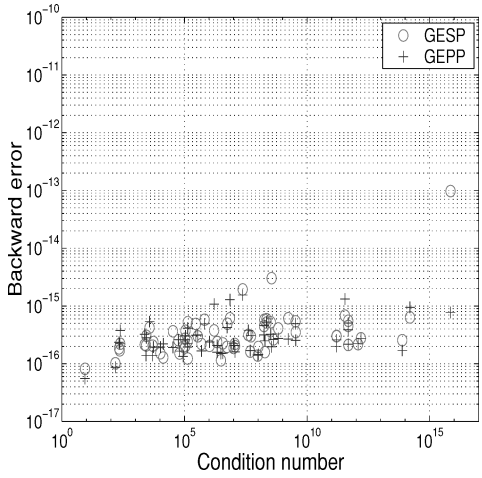

Fig. 5. Backward error $\max_i \frac{|A \cdot x - b|_i}{(|A| \cdot |x| + |b|)_i}$.

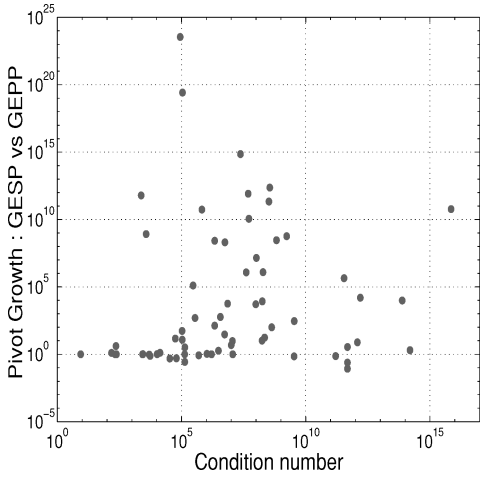

Fig. 6. The ratio of pivot growth of GESP versus GEPP.

or *berr* does not decrease by one-half of the previous step. For most matrices, the iteration terminates with no more than three steps: 9 matrices require one step, 46 matrices require two steps, 5 matrices require three steps, and 8 matrices require more than three steps. In the case of conventional Gaussian elimination with partial pivoting (GEPP) (as in sequential `SuperLU`), 4 matrices require one step, 63 matrices require two steps, and 1 matrix requires three steps.

For each matrix, we present two error metrics, in Figures 4 and 5, respectively, to assess the accuracy and stability of GESP. Figure 4 plots the error from GESP versus the error from GEPP for each matrix: a dot on the diagonal means the two errors were the same, a dot below the diagonal means GESP is more accurate, and a dot above the diagonal means GEPP is more accurate. Figure 4 shows that the error of GESP is at most a little larger, and can be

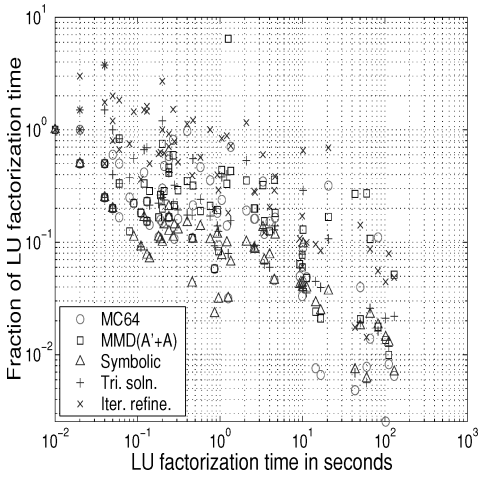

Fig. 7. The times for the other steps of GESP, as fraction of the factorization time.

smaller (36 out of 68 matrices), than the error from GEPP. Figure 5 shows that the componentwise backward error [Demmel 1997] is also small, usually near ε and never larger than 10^{-13}.

Figure 6 compares the pivot growth of GESP versus that of GEPP. Here, the pivot growth is defined as $\frac{||U||_\infty}{||A||_\infty}$. For 31 matrices, GESP and GEPP have comparable pivot growth. For 10 matrices, GESP has more than 10 orders of magnitude larger pivot growth than GEPP, up to 10^{24}. Even in the presence of such large pivot growth, the iterative refinement can effectively recover any loss of accuracy during the factorization.

Note that Figure 1 shows all the techniques that are implemented in the code. Some may not be needed for some problems. Our experiment shows that the half-precision perturbation introduced in step (4) is not needed for most matrices. It is necessary for five matrices (FIDAPM11, GOODWIN, GRAHAM1, INV-EXTRUSION-1, and MIXING-TANK), but is bad for four others (EX11, FIDAP011, INACCURA, and RAEFSKY4). The rest of the matrices are insensitive to this option, because either no tiny pivots occur or it does not matter what you do. Therefore, in our code, we provide a flexible interface so the user is able to turn on or off any of these options (steps (1), (2), (6), and the diagonal perturbation in step (4)).

Now we turn to the second configuration of our algorithm, in which restarted GMRES [Saad and Schultz 1986] was used in step (6) (we used the version from SPARSKIT [Saad n.d.]). The restart value is 50. Here, our LU factorization is used in preconditioning for GMRES. The convergence test is based on residual norm $||r_i||_2 \leq rtol * ||r_0||_2 + atol$, where the relative tolerance $rtol$ and absolute tolerance $atol$ are 10^{-6} and 10^{-10}, respectively. For the four "bad" matrices above (EX11, FIDAP011, INACCURA, and RAEFSKY4), GMRES takes 497, 530, 5, and 41 iterations to converge. The number of tiny pivots replaced in step (4) for these four matrices was 8666, 8602, 3, and 51, respectively. For most of the other matrices, GMRES terminates within two iterations. This shows that with one

parameter setting, we can solve all the test problems accurately. In the software, we plan to provide an interface to the user with the options of using various iterative schemes.

We now evaluate the runtime of each step of GESP in Figure 1, in our first configuration with iterative refinement in step (6). This is done with respect to the sequential runtime. For large enough matrices, the LU factorization in step (4) dominates all the other steps, so we will measure the time of each step with respect to step (4).

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 $O(nnz(A))$ [Davis et al. 2000; Duff and Koster 1999], as opposed to the superlinear memory requirement for L and U factors, so in the meantime we can run the ordering algorithms on a single processor.

Figure 7 shows the times spent in the other steps of GESP as the fraction of the sequential time for the factorization step. The times are significant for the small problems, but drop to smaller fractions as the problems become larger. Only the large matrices are of interest for parallel machines and are also the ones which `SuperLU_DIST` was designed for.

We note that we have developed a theoretical algorithm that provides a guarantee of stability while using static pivoting, by using variable precision. The purpose of this is to show that even if some "dynamic" (i.e., matrix-dependent) algorithm were required to absolutely guarantee stability, then the dynamic part could involve variable precision as opposed to communication patterns and layout. We note that even conventional partial pivoting does not absolutely guarantee stability, because exponential pivot growth is still possible, albeit very unlikely. It is a risk most users can live with, as we suggest is also the case for static pivoting. The details of this theoretical algorithm may be found in Appendix A of Li and Demmel [2002].

2.2 Opportunities for Better Fill-Reducing Orderings

For the unsymmetric factorizations, the preordering for sparsity is less well understood than that for the Cholesky factorization. Most unsymmetric ordering methods use the symmetric ordering techniques on a symmetrized matrix (e.g., A^TA). Now we examine the structural relationships of several matrices, and describe the rationale behind the above ordering methods. Consider the LU factorization with partial pivoting $P_rA = LU$, where P_r is a permutation matrix describing row interchanges. Also consider the Cholesky factorization $A^TA = R^TR$, and the QR factorization $A = QR$ computed by the Householder transformation.[1] Q is represented by the "Householder matrix" H whose columns are the Householder vectors. The nonzero structure for L and U cannot be predicted immediately from the nonzero structure of A, because the row interchanges during the factorization depend on the numerical values. However, for any row interchanges, the structures of L and U are *subsets* of the structures of H (or R^T) and R, respectively [George et al. 1988; George and

[1]The R factor in the Cholesky factorization and the R factor in the QR factorization are identical.

Table II. Impact of Different Ordering Methods on the Size of the Factors; the GESP Algorithm is Used

	Nonzeros in $L+U$ (10^6)			
	(A^TA)–based		(A^T+A)–based	
Matrix	MMD	COLAM	MMD	AMD
BBMAT	49.1	49.8	41.1	40.2
ECL32	73.5	72.6	42.4	42.7
FIDAPM11	26.4	24.3	24.8	24.8
INV-EXTRUSION-1	53.7	62.7	29.1	28.4
MIXING-TANK	86.9	81.4	40.7	41.2
RMA	14.7	16.3	9.3	9.3
TWOTONE	22.6	18.3	11.4	11.9
WANG4	27.7	25.5	10.5	10.7

Ng 1987]. Therefore, a good symmetric ordering P_c on A^TA (either based on minimum degree or nested dissection) that preserves the sparsity of R can be applied to the columns of A, forming AP_c^T, so that the LU factorization of the column-permuted matrix AP_c^T is sparser than that of the original matrix A. This is due to the relation $P_c(A^TA)P_c^T = (AP_c^T)^T(AP_c^T)$. A drawback with the above approach is that computing the structure of A^TA can be expensive both in time and space since A^TA may be much denser than A. Davis et al. [2000] developed an algorithm, called COLAMD (for "Column approximate minimum degree"), to compute P_c directly from the sparsity structure of A. It is based on the same strategy, that is, to make the "upper bound" matrices H and R sparser, but uses better heuristics. Both serial `SuperLU` and `SuperLU_MT` have incorporated both column ordering methods; that is, the user can choose to obtain a column ordering by calling multiple minimum defgree (MMD) [Liu 1985] on A^TA, or by calling COLAMD.

Since the "A^TA-based ordering" methods attempt to account for all possible row interchanges, such ordering may be too generous when only a limited amount of pivoting is needed. This is especially true for our GESP algorithm, in which the row interchanges are performed prior to the factorization. During the factorization, the pivots are chosen solely on the main diagonal. A better fill-reducing ordering would be based on the symmetric matrix A^T+A, instead of A^TA, because the symbolic Cholesky factor of A^T+A is a much tighter upper bound on the structures of L and U than that of A^TA. Note that in this case, we perform a symmetric permutation PAP^T so that the entries of the main diagonal of the permuted matrix remain the same as those in the original matrix A. Table II lists the amount of fill in the LU factorization using different ordering methods. It is clear that the orderings based on A^T+A is much better than those based on A^TA. Sometimes the improvement can be more than a factor of 2; see matrices INV-EXTRUSION-1, MIXING-TANK, and WANG4. The only exception is FIDAPM11, for which the three ordering methods are comparable.

Although the (A^T+A)–based orderings improve the ordering quality, they still may not be the most effective fill-reducing methods, since symmetrization A^T+A may destroy the sparsity of matrix A, particularly when A is highly unsymmetric. Recently, motivated by the GESP algorithm and an unsymmetrized

multifrontal method [Amestoy and Puglisi 2002], Amestoy, Li and Ng [Amestoy et al. 2003] proposed a new symmetric ordering scheme that does not require any symmetrization of the underlying matrix, that is, it works directly on matrix A itself. The scheme is similar to the Markowitz [1957] scheme, but limits the pivot search to the entries on the main diagonal. The efficient implementation is similar to that of approximate minimum degree (AMD) [Amestoy et al. 1996], but it generalizes the (symmetric) quotient graph to the bipartite quotient graph to model the unsymmetric node elimination. The preliminary results show that the new ordering method reduces the amount of fill by 10% on average for very unsymmetric matrices, when compared with applying AMD to $A^T + A$. In the future, we will incorporate this new ordering algorithm into `SuperLU_DIST`.

The better choice of sparsity ordering algorithm is indeed an added benefit of the GESP algorithm over GEPP. Throughout the paper, we only report the results using the ordering algorithms based on $A^T + A$.

3. PARALLEL ALGORITHMS

In this section, we describe our design, implementation, and the performance of the distributed algorithms for two major steps of the GESP method: sparse LU factorization (step (4)) and sparse triangular solve (step (5)). Our implementation uses the Message Passage Interface (MPI) [MPI n.d.] to communicate data. We have tested the code on a number of platforms, such as Cray T3E, IBM SP, and Berkeley NOW.

3.1 Matrix to Processor Mapping and Distributed Data Structure

We distribute the matrix in a two-dimensional block-cyclic fashion. In this distribution, the P processes are arranged as a 2D process grid of shape $nprow \times npcol$. The matrix is partitioned into blocks of submatrices. The block definition is based on the notion of *unsymmetric supernode* first introduced in Demmel et al. [1999a]; it is defined over the matrix factor L. A supernode is a range $(r : s)$ of columns of L with the triangular block just below the diagonal being full, and the same nonzero structure elsewhere (either full or zero). This supernode partition is used as the block partition in both row and column dimensions; that is, the diagonal blocks are square. 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. The off-diagonal blocks may be rectangular and need not be full. Furthermore, the columns in a block of U do not necessarily have the same row structure. We call a dense subcolumn in a block of U a *segment*. By block-cyclic layout, we mean block (I, J) is mapped onto the process at coordinate $((I - 1) \bmod nprow, (J - 1) \bmod npcol)$ of the process grid. During factorization, block $L(I, J)$ is only needed by the processes on the process row $((I - 1) \bmod nprow)$, thus restricting the communication. Similarly, block $U(I, J)$ is only needed by the processes on the process column $((J - 1) \bmod npcol)$. Figure 8 illustrates such a 2D block-cyclic layout.

Although a 1D partition is more natural to sparse matrices and is much easier to implement, a 2D layout strikes a good balance among locality

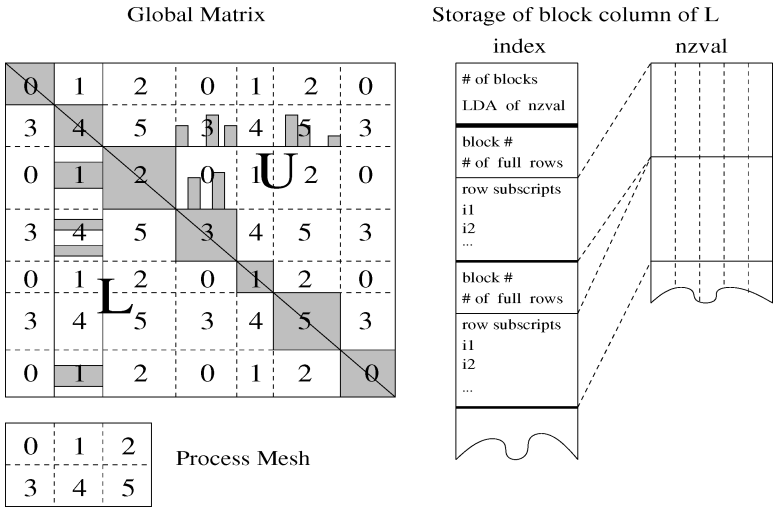

Fig. 8. The 2D block-cyclic layout and the data structure to store a local block column of L.

(by blocking), load balance (by cyclic mapping), and lower communication volume (by 2D mapping). 2D layouts have been demonstrated to be more scalable in the implementations for dense matrices [Blackford et al. 1997] and sparse Cholesky factorization [Gupta and Kumar 1995; Rothberg 1996].

We now describe the distributed data structures to store local submatrices. In the 2D blocking, each block column of L resides on more than one process, namely, a column of processes. For example, in Figure 8, the second block column of L resides on the column processes {1, 4}. Process 1 only owns two nonzero blocks, which are not contiguous in the global matrix. The schema on the right of Figure 8 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 indices for the whole block column and for each individual block in it. The zero blocks are not stored; neither do we store the zeros in a nonzero block. Both lower and upper triangles of the diagonal block are stored in the L data structure. A process owns $\lceil N/npcol \rceil$ block columns of L, so it needs $\lceil N/npcol \rceil$ pairs of `index/nzval` arrays.

For matrix U, we use a row-oriented storage for the block rows owned by a process, although for the numerical values within each block we still use column-major order. Similarly to L, we also use a pair of `index/nzval` arrays to store a block row of U. Due to asymmetry, each nonzero block in U has the skyline structure as shown in Figure 8 (see Demmel et al. [1999a] for details on the skyline structure). Therefore, the organization of the `index[]` array is different from that of L, which we omit showing in the figure.

The user can control the partitioning and mapping. First, the user can set the *maximum block size* parameter. The symbolic factorization algorithm identifies supernodes, and chops the large supernodes into smaller ones if their sizes

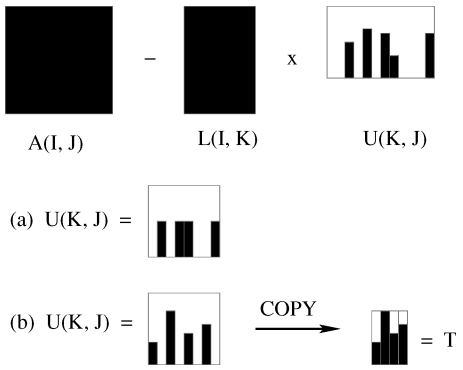

Fig. 9. Illustration of the numerical kernels used in SuperLU_DIST.

exceed this parameter. The supernodes may be smaller than this parameter due to sparsity, and the blocks are then defined by the supernode boundaries. (That is, supernodes can be smaller than the maximum block size but never larger.) Our experience has shown that a good value for this parameter on the IBM SP2 is around 40, while on the Cray T3E it is around 24, because T3E has smaller caches on each processor. Second, the user can set the shape of the process grid, such as 2×3 or 3×2. Better performance is obtained when the process row dimension is kept slightly smaller than the process column dimension. Since we do no dynamic pivoting, block partitioning and the setup of the data structure can all be performed in the symbolic algorithm. This is much cheaper to execute as opposed to partial pivoting, where the size of the data structure cannot be forecast and must be determined on the fly as factorization proceeds.

3.2 Numerical Kernel Based on Level 3 BLAS

The main numerical kernel during the factorization is a block update corresponding to the rank-k update to the Schur complement:

$$A(I, J) \leftarrow A(I, J) - L(I, K) \times U(K, J);$$

see Figure 9. In earlier versions of SuperLU, this computation was based on Level 2.5 basic linear algebra subroutines (BLAS). That is, we call the Level 2 BLAS routine GEMV (matrix-vector product) but with multiple vectors (segments), and the matrix $L(I, K)$ is kept in cache across these multiple calls. This to some extent mimics the Level 3 BLAS GEMM (matrix-matrix product) performance. However, the difference between Level 2.5 and Level 3 is still quite large on many machines, for example, the IBM SP2. This motivated us to modify the kernel in the following way in order to use Level 3 BLAS. For best performance, we distinguish two cases corresponding to the two shapes of a $U(K, J)$ block.

—*The segments in $U(K, J)$ are of same height, as shown in Figure 9 (a).* Since the nonzero segments are stored contiguously in memory, we can call GEMM directly, without performing operations on any zeros.

—*The segments in $U(K, J)$ are of different heights, as shown in Figure 9 (b).* In this case, we first copy the segments into a temporary working array T, with

some leading zeros padded if necessary. We then call GEMM using $L(I, K)$ and T (instead of $U(K, J)$). We perform some extra floating-point operations for those padding zeros. The copying itself does not incur a runtime cost, because the data must be loaded in the cache anyway. The working storage T is bounded by the maximum block size, which is a tunable parameter. For example, we usually use 40×40 on the IBM SP2 and 24×24 on the Cray T3E.

Compared with the Level 2.5 BLAS kernel, this Level 3 BLAS kernel improved the uniprocessor factorization time by about 20% to 40% on the IBM SP2. A performance gain was also observed on the Cray T3E. It is clear that the extra operations are well offset by the benefit of the more efficient Level 3 BLAS routines.

3.3 Parallel Factorization with Pipelining

In this subsection, we first describe in detail how the parallel factorization algorithm utilizes the pipeline effect. Then we discuss how to improve the performance robustness by introducing immediate sends and receives. The following notation will be used in Figure 11 and throughout the discussion. MATLAB notation is used for integer ranges and submatrices.

—Process IDs
  —$PROC_c(K)$: the set of column processes that own block column K.
    For example, in Figure 8, $PROC_c(3) = PROC_c(6) = \{2, 5\}$.
  —$PROC_r(K)$: the set of row processes that own block row K.
    For example, in Figure 8, $PROC_r(1) = PROC_r(3) = \{0, 1, 2\}$.
  —$P_K = PROC_c(K) \cap PROC_r(K)$.
  —*me*: the process rank as illustrated in Figure 8.
—Tasks labelled in Figure 11
  —F (...): Factorize a block column or a block row[2]
  —S (...): Send a block column or a block row
  —R (...): Receive a block column or a block row
  —$U^{(k)}$(...): Update the trailing submatrix using $L(:, K)$ and $U(K, :)$

The parallel sparse LU factorization algorithm is right-looking and loosely synchronous, as shown in Figure 10. It loops over the number of supernodes. The Kth iteration of the loop consists of three steps: (1) the process set $PROC_c(K)$ factors the block column $L(K : N, K)$; (2) the process set $PROC_r(K)$ factors the block row $U(K, K + 1 : N)$; and (3) all the processes perform the Schur complement update by $L(K + 1 : N, K)$ and $U(K, K + 1 : N)$. The last step represents most of the work and also exhibits more parallelism than the other two steps.

In the actual implementation we use a *pipelined* organization so that processes $PROC_c(K + 1)$ will start step (1) of iteration $K + 1$ as soon as the rank-k 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 $PROC_c(K+1)$.

[2]There is also communication involved in this task, but it is negligible, and so is omitted in the discussion.

```
for block K = 1 to N do
    (1) if [ me ∈ PROC_C(K) ] then
            Factorize block column 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 one process in PROC_C(K)
        endif
    (2) if [ me ∈ PROC_R(K) ] then
            Factorize block row U(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 one process in PROC_R(K) if I need it
        endif
    (3) for J = K + 1 to N do
            for I = K + 1 to N do
                if [ me ∈ PROC_R(I) and me ∈ PROC_C(J)
                    and L(I, K) ≠ 0 and U(K, J) ≠ 0 ] then
                    Update trailing submatrix A(I, J) ← A(I, J) − L(I, K) · U(K, J)
                endif
end for
```

Fig. 10. The parallel right-looking LU factorization.

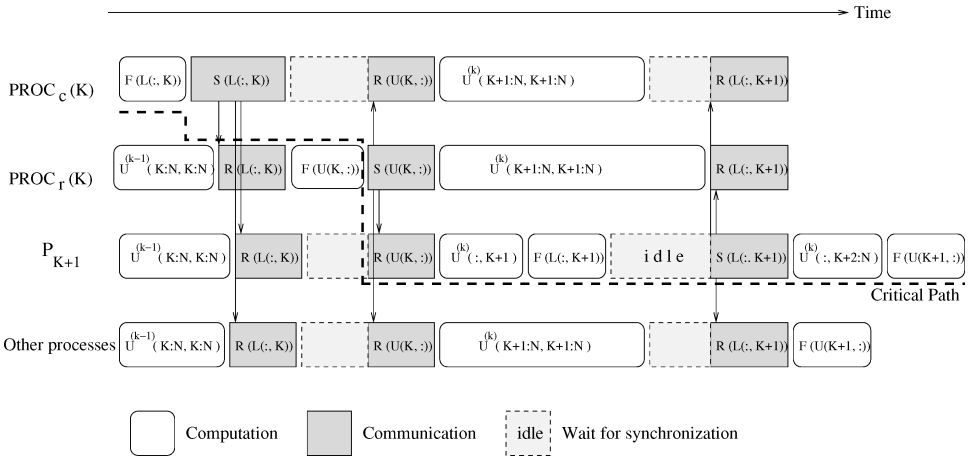

Fig. 11. Illustration of the pipeline at steps K and $K + 1$ during the `SuperLU` factorization.

Figure 11 illustrates this idea using Steps K and $K + 1$ of the algorithm. In the figure, we show the activities of the four process groups along the time line. The path marked with the dashed line represents the critical path, that is, the parallel runtime can be reduced only if the critical path is shortened. The block factorization tasks "F (...)" are usually on the critical path, whereas the update tasks "U (...)" are often overlapped with the other tasks. There is lack of parallelism for the "F (...)" tasks in Steps (1) and (2), because only one set of column processes or row processes participate in these tasks. This pipelining mechanism alleviates this problem. For instance, on 64 processors of the Cray T3E, we observed speedups of between 10% and 40% over the nonpipelined implementation as in Figure 10.

In an earlier version of the code, we used mpi's standard send and receive operations `mpi_send` and `mpi_recv` for the message transfer tasks "S (. . .)" and "R (. . .)." In Figure 11, we see idle time (longer send) during the sending of "S (L(:, $K+1$))" for process P_{K+1} on the critical path. This can happen if the sender and receiver are required to handshake before proceeding, as is the case with large messages that exceed the MPI internal buffer size [Amestoy et al. 2001b]. That is, process P_{K+1} posts `mpi_send` long before processes $PROC_r(K)$ post the matching `mpi_recv`, and the sender must be blocked to wait for `mpi_recv`. To avoid this synchronization cost, we introduced the nonblocking send and receive primitives, `mpi_isend` and `mpi_irecv` as follows:

—For the sender, we simply replace `mpi_send` by `mpi_isend`. This could eliminate the idle time during the send "S (L(:, $K+1$))" shown in Figure 11.

—For the receiver, we will post `mpi_irecv` much earlier than we actually need the data. For example, for processes $PROC_r(K)$ in Figure 11, we could post "R (L(:, $K+1$))" before "U ($A(K+1:N, K+1:N)$)." That is, as soon as we have received a message using `mpi_wait`, we will post the `mpi_irecv` for the next message, before performing the local computation with the just-arrived message.

To implement this idea, we need to provide user-level buffer space to accommodate the messages in transit. Since for each process, there is only one outstanding message to be received, we only need one extra buffer. Figure 12 sketches the pipelining algorithm using `mpi_isend` and `mpi_irecv`. The main difference from Figure 10 is in step (3). In the new algorithm, the original step (3) is split into two substeps (3.1) and (3.2). Step (3.1) implements a look-ahead scheme. Here, we only update the $(K+1)$st block column, then immediately factorize this column and post send and receive of the factorized column for the $(K+1)$st iteration of the loop. This message transfer will overlap with the rest of the trailing submatrix update appearing in step (3.2). In step (1), the processes wait for the posted send and receive to complete. In particular, `mpi_wait` in line 9 is matched with the posted `mpi_isend` in line 23 (and 3); `mpi_wait` in line 11 is matched with the posted `mpi_irecv` in line 25 (and 5).

We observed a big performance difference between the blocking and nonblocking versions of the codes on the Cray T3E. With an increasing number of processors, the message size is usually decreasing. We show this in Table III, because the smaller message size implies that there is less handshaking between the sender and receiver in the blocking code. Thus, the performance gain of the nonblocking code on a large number of processors is less dramatic than that on a smaller number of processors. The largest performance gain occurs at four processors, where the nonblocking code is almost twice as fast as the blocking code.

3.4 Parallel Triangular Solution

The sparse triangular solves are also designed around the same distributed data structure (i.e., there is no data redistribution). bThe forward substitution proceeds from the bottom of the elimination tree (etree of $A^T + A$) to the root,

```
   /* —- Set up the initial stage for the pipeline —- */
1. if [ me ∈ PROC_c(1) ] then
2.       Factorize block column L(1 : N, 1)
3.       Post send L(1 : N, 1) to the processes in my row who need it (– mpi_isend – )
4. else
5.       Post receive L(1 : N, 1) from one process in PROC_c(1) if I need it (– mpi_irecv – )
6. endif

   /* —- Main pipeline loop —- */
7. for block K = 1 to N do
8.   (1) if [ me ∈ PROC_c(K) ] then
9.           Wait for the posted send of L(K : N, K) to complete (– mpi_wait – )
10.      else
11.          Wait for the posted receive of L(K : N, K) to complete (– mpi_wait – )
12.      endif
13. (2) if [ me ∈ PROC_r(K) ] then
14.          Factorize block row U(K, K + 1 : N)
15.          Send U(K, K + 1 : N) to processes in my column who need it
16.      else
17.          Receive U(K, K + 1 : N) from one process in PROC_r(K) (if I need it)
18.      endif
19. (3.1) if [ K + 1 ≤ N ] then
             /* —- Factor-ahead scheme —- */
20.          if [ me ∈ PROC_c(K + 1) ] then
21.              Update (K + 1)-st column A(:, K + 1) ← A(:, K + 1) − L(:, K) · U(K, K + 1)
22.              Factorize block column L(:, K + 1)
23.              Post send L(:, K + 1) to the processes in my row who need it (– mpi_isend – )
24.          else
25.              Post receive L(:, K + 1) from one process in PROC_c(K + 1) (– mpi_irecv – )
26.          endif
27.      endif
28. (3.2) for J = K + 2 to N do
29.          for I = K + 1 to N do
30.              if [ me ∈ PROC_r(I) and me ∈ PROC_c(J)
31.                  and L(I, K) ≠ 0 and U(K, J) ≠ 0 ] then
32.                  Update trailing submatrix A(I, J) ← A(I, J) − L(I, K) · U(K, J)
33.              endif
34.          end for
35.      end for
36. end for
```

Fig. 12. Parallel LU factorization with nonblocking send and receive.

Table III. Maximum Size of the Message (in Mbytes) During the Factorization

Matrix	Ordering	Number of processors				
		4	8	16	32	64
BBMAT	AMD	0.19	0.18	0.09	0.09	0.05
ECL32	AMD	0.32	0.32	0.16	0.16	0.09
INV-EXTRUSION-1	AMD	0.24	0.24	0.12	0.12	0.07
MIXING-TANK	AMD	0.32	0.33	0.17	0.16	0.09

whereas the back substitution proceeds from the root to the bottom. Figure 13 outlines the algorithm for sparse lower triangular solve. The algorithm is based on a sequential "inner-product" formulation. In this formulation, before we solve for the Kth subvector $x(K)$, the update from the inner-product of $L(K, 1 : K-1)$ and $x(1 : K-1)$ must be accumulated and then subtracted from $b(K)$. The diagonal process, at the coordinate (K mod $nprow$, K mod $npcol$) of the process

```
1. Let mycol (myrow) be my process column (row) coordinate in the process grid
2. x = b;  lsum = 0
   /* —- Compute leaf nodes —- */
3. for block K = 1 to N
4.       if ( myrow = (K mod nprow) and mycol = (K mod npcol) and frecv[K] = 0 )
5.           x(K) = L(K, K)^{-1} · x(K)
6.           Send x(K) to the column processes PROC_C(K)
7.       endif
8. end for

   /* —- Compute internal nodes —- */
9. while ( I have more work ) do
10.      Receive a message
11.      if ( message is x(K) )
12.          for each of my L(I, K) ≠ 0, I > K
13.              lsum(I) = lsum(I) + L(I, K) · x(K)
14.              fmod(I) = fmod(I) − 1
15.              if ( fmod(I) = 0 )
16.                  Send lsum(I) to the diagonal process that holds x(I)
17.              endif
18.          end for
19.      else if ( message is lsum(K) )
20.          x(K) = x(K) − lsum(K);
21.          frecv(K) = frecv(K) − 1
22.          if ( frecv(K) = 0 )
23.              x(K) = L(K, K)^{-1} · x(K)
24.              Send x(K) to the column processes PROC_C(K)
25.          endif
26.      endif
27. end while
```

Fig. 13. Parallel lower triangular solve $L \cdot x = b$.

grid, is responsible for solving for $x(K)$. Since each block row $L(K, 1 : K-1)$ is distributed among the row process set $PROC_R(K)$, the inner-product is formed in a distributed way. Each process stores the partial sum in $lsum(K)$ locally. After it accumulates all the product contributions from various blocks, it sends the partial sum to the diagonal process that holds $x(K)$. This is like a reduction operation among a row process set, except that some processes may not participate in this reduction if they do not have any nonzero block in this block row. Two counters, *frecv* and *fmod*, are used to facilitate the asynchronous execution of different operations. $fmod(K)$ counts the number of local block products to be summed into $lsum(K)$. When $fmod(K)$ becomes zero, the partial sum $lsum(K)$ is sent to the owner of $x(K)$. $frecv[K]$ counts the number of process updates to $x(K)$ to be received by the owner of $x(K)$. This is needed because, due to sparsity, not all processes in $PROC_R(K)$ contribute to the update. When $frecv(K)$ becomes zero, all the needed inner-product updates to $x(K)$ are complete and $x(K)$ can then be solved.

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.

Table IV. Characteristics of the Large Matrices (N-Sym is the fraction of nonzeros matched by equal values in symmetric locations. S-Sym is the fraction of nonzeros matched by nonzeros in symmetric locations)

	Order	nnz(A)	N-Sym	S-Sym	MC64 S-Sym	nnz(L+U) (10^6)	Flops (10^9)
BBMAT	38744	1771722	0.02	0.54	0.50	41.1	34.0
ECL32	51993	380415	0.66	0.93	0.93	42.4	68.3
INV-EXTRUSION-1	30412	1793881	0.73	0.97	0.86	28.4	28.0
MIXING-TANK	29957	1995041	0.98	1.00	0.91	41.2	64.6
TWOTONE	120750	1224224	0.14	0.28	0.43	11.9	8.0
WANG4	26068	177196	0.19	1.00	1.00	10.7	9.1

Table V. Factorization Time (in Seconds) on the Cray T3E ("—" indicates not enough memory. The best time is indicated in boldface. Note: MC64 is needed only by TWOTONE, and the time is 1.6 seconds)

		Symb	Number of processors							
Matrix	Ord.	Time	1	2x2	4x4	4x8	8x8	8x16	8x32	16x32
BBMAT	AMD	4.6	—	64.7	21.3	12.8	9.2	7.2	**6.7**	6.8
	ND	6.3	—	132.9	39.8	23.5	15.6	11.1	9.9	**9.6**
ECL32	AMD	6.0	—	106.8	31.2	18.3	12.3	8.2	6.8	**6.5**
	ND	3.9	—	48.5	15.7	9.6	7.6	**5.6**	5.7	6.1
INV-EXTRUSION-1	ND	2.4	68.2	21.3	8.2	5.6	4.9	3.7	**3.5**	3.8
MIXING-TANK	ND	2.5	88.1	25.2	8.6	5.6	4.6	**3.1**	3.1	3.1
TWOTONE	MC64+ AMD	3.2	—	103.8	32.8	19.5	13.3	9.7	**7.6**	9.0
WANG4	AMD	1.3	57.0	17.8	6.8	4.8	4.3	3.4	**3.1**	3.7

The algorithm for the upper triangular solve is similar, However, because of the row-oriented storage scheme used for matrix U, there is a slight complication in the actual implementation. Namely, we have to build two vertical linked lists to enable rapid access of the matrix entries in a block column of U.

4. PARALLEL PERFORMANCE AND SCALABILITY

In this section, we restrict our attention to several large matrices selected from our testbed in Table I, because only large problems need to use parallel machines. These matrices are representative of different application domains. The characteristics of these matrices are given in Table IV. The configuration of the GESP algorithm includes steps (2) to (5) in Figure 1, and iterative refinement in step (6). Only TWOTONE requires step (1). The timing results have been obtained on the Cray T3E-900 (512 450-MHz EV-5 processors, 256 Mbytes of memory per processor, 900 peak Megaflop rate per processor) installed at NERSC.

4.1 Factorization

We show in Table V the factorization time of SuperLU_DIST. The symbolic analysis is not yet parallel. Although it takes very little time, its parallelization would enhance memory scalability, and will be our future work. There is an on-going work by Riedy [2003] on parallel bipartite matching algorithm. We will use it in place of MC64 in the future. For now, we start with a copy of the

Table VI. Maximum Size of the Messages (Max in Mbytes), Average Volume of Communication (Vol in Mbytes), and Number of Messages per Processor (Mes)

		Number of processors								
		2x2			4x4			8x8		
Matrix	Ord.	Max	Vol	Mes	Max	Vol	Mes	Max	Vol	Mes
BBMAT	AMD	0.18	81	23412	0.09	61	34176	0.05	35	35035
	ND	0.17	82	30698	0.09	62	45598	0.04	36	50925
ECL32	AMD	0.32	90	27437	0.16	67	37486	0.09	39	34981
	ND	0.25	56	28966	0.13	42	41172	0.07	24	41271
INV-EXTRUSION-1	ND	0.15	31	17774	0.08	23	25824	0.05	13	27123
MIXING-TANK	ND	0.19	40	13667	0.11	30	19635	0.05	18	19064
TWOTONE	MC64+									
	AMD	0.26	27	120006	0.15	20	153995	0.05	11	104906
WANG4	AMD	0.19	24	27728	0.10	18	34495	0.05	10	27561

entire matrix on each processor, and run steps (1) through (3) independently on each processor. The third column of Table V reports the time spent in the 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. (In the skeleton graph of U, only the first nonzero in a segment of U is stored.) The subsequent columns in the table show the numerical factorization time with a varying number of processors. For all these matrices, the algorithm can efficiently use 128 processors. Beyond 128 processors, not all matrices can benefit from the additional processor power. Only BBMAT with ND ordering [Karypis and Kumar 1998] and ECL32 with AMD [Amestoy et al. 1996] can benefit from using 512 processors. Our lack of other large unsymmetric systems gives us few data points in this regime. To further analyze the scalability of our solvers, we consider three-dimensional regular grid problems in Section 4.4.

We also observe that the algorithm does not always fully benefit from the reduction in the number of operations potentially available from the use of a nested dissection ordering (see BBMAT). There are several reasons, and the improvement remains as future work. First, the algorithm does not fully exploit the parallelism of the elimination dags. Second, the pipelining mechanism does not fully benefit from the sparsity of the factors (a blocked column factorization should be implemented). This also explains why it does not fully benefit from the better balanced tree generated by a nested dissection ordering.

To better understand the performance, we show in Table VI the average communication volume. The speed of communication can depend very much on the number and the size of the messages, and we also indicate the maximum size of the messages and the average number of messages per processor. With an increasing number of processors, the communication volume and the size of the messages usually decrease, whereas the total number of messages usually increase. This implies that on larger numbers of processors, it is important to be able to overlap the computation with communication of many small messages. Our use of nonblocking sends and receives in the loosely synchronous pipelining algorithm facilitates this.

Table VII. Solve Time (in Seconds) on the Cray T3E ("+IR" shows the time spent improving the initial solution using iterative refinement. "—" indicates not enough memory. The best time is indicated in boldface)

Matrix	Ord.	IR (Steps)	Number of processors							
			1	2x2	4x4	4x8	8x8	8x16	8x32	16x32
BBMAT	AMD	no	—	1.39	0.78	0.75	0.49	0.50	0.40	**0.38**
		IR (3)	—	5.00	2.84	2.69	1.88	1.74	1.44	1.38
	ND	no	—	2.01	1.03	0.89	0.65	0.60	0.57	**0.43**
		IR (3)	—	6.86	3.66	3.19	2.44	2.11	1.97	**1.58**
ECL32	AMD	no	—	1.87	1.09	1.09	0.68	0.73	**0.50**	0.51
		IR (2)	—	4.17	2.66	2.54	1.66	1.68	**1.19**	1.22
	ND	no	—	1.49	0.95	0.95	0.64	0.64	0.47	**0.43**
		IR (2)	—	3.37	2.47	2.25	1.63	1.51	1.13	**1.08**
INV-EXTRUSION-1	ND	no	1.50	0.73	0.43	0.39	0.29	0.27	0.22	**0.19**
		IR (3)	6.19	2.77	1.65	1.51	1.16	1.00	0.85	**0.75**
MIXING-TANK	ND	no	1.54	0.64	0.35	0.31	0.21	0.22	0.17	**0.15**
		IR (3)	6.46	2.56	1.42	1.25	0.92	0.85	0.69	**0.64**
TWOTONE	MC64+	no	—	2.63	1.93	1.84	1.28	1.24	0.93	**0.85**
	AMD	IR (3)	—	9.00	6.95	6.68	4.97	4.50	3.43	**3.18**
WANG4	AMD	no	1.04	0.63	0.42	0.43	0.28	0.27	0.22	**0.19**
		IR (2)	2.34	1.43	0.99	1.00	0.69	0.64	0.52	**0.46**

4.2 Triangular Solution

In this section, we focus on the time spent to obtain the solution. We apply enough steps of iterative refinement to ensure that the componentwise relative backward error (*berr*) is less than $\varepsilon \approx 10^{-16}$. Each step of iterative refinement involves not only a forward and a backward solve but also a matrix-vector product with the original matrix. In Table VII, we report both the time to perform one solution step (using the factorized matrix to solve $\mathbf{A}x = b$) and, the time to improve the solution using iterative refinement (lines with "IR").

On a small number of processors (fewer than eight), the solve phase is almost two orders of magnitude less costly than the factorization. On a large number of processors, because the solve phase is relatively less scalable than the factorization phases, the difference drops to one order of magnitude. On applications for which a large number of solves might be required per factorization, this could become critical for the performance and will be addressed in the future. The cost of iterative refinement can significantly increase the cost of obtaining a solution. The use of MC64 to preprocess the matrix can reduce the number of steps of iterative refinement, Although both the solve times and iterative refinement times decrease very slowly with an increasing number of processors, they still keep decreasing up to 512 processors.

4.3 Memory Usage

In Table VIII, we report the amount of memory actually used during the *LU* factorization phase. This includes both reals and integers for the matrices, the working arrays, and the communication buffers. We notice a significant reduction in the required memory per processor when increasing the number of processors, showing good memory scalability. We also observe that there is little

Table VIII. Memory Used During Factorization (in Megabytes, per Processor)

Matrix	Ordering	Number of processors					
		2x2		4x4		8x8	
		Avg	Max	Avg	Max	Avg	Max
BBMAT	AMD	113	114	50	51	33	34
	ND	124	128	60	61	43	44
ECL32	AMD	113	115	42	44	24	25
	ND	79	81	33	34	21	22
INV-EXTRUSION-1	ND	47	48	22	22	15	16
MIXING-TANK	ND	55	56	23	23	14	15
TWOTONE	MC64+AMD	66	80	35	41	24	24
WANG4	AMD	33	34	14	14	8	9

difference between the average and maximum memory usage, showing that the algorithm is well balanced.

Note that memory scalability can be critical on globally addressable platforms where parallelism increases the total memory used. On purely distributed machines such as the T3E, the main factor remains the memory used per processor, which should allow large problems to be solved when enough processors are available.

4.4 Scalability

As stated in Introduction, our goal is to make sparse LU factorization as scalable as sparse Cholesky. In this section we present the efficiency of our factorization algorithm on model problems, both analytically and experimentaly, and show that the algorithm and the implementation indeed meet our goal.

Consider the matrix from discretizing the Laplacian operator on a 3D cubic grid with N unknowns. Using the standard nested dissection ordering, the fill in the factored matrix is $O(N^{4/3})$ and the number of floating-point operations to factorize the matrix is $O(N^2)$ [George and Liu 1981]. Let the P processors be arranged as a square process grid. In our parallel algorithm (Figure 12), each nonzero element is sent to at most $\sqrt{P}$ processors. Therefore, the total communication overhead is $O(N^{4/3}\sqrt{P})$. When the total amount of work N^2 increases proportionally with the overhead $N^{4/3}\sqrt{P}$, the parallel efficiency can be kept fixed, that is, $N^2 = c \cdot N^{4/3}\sqrt{P}$, for some constant c. Rewriting this, we have the work-processor relation: $N^2 = c^3 \cdot P^{3/2}$. This function shows the growth rate of the amount of work required to keep fixed efficiency as P increases, and is called the *isoefficiency function* [Kumar et al. 1994]. A small isoefficiency function means a small increase in the amount of work is sufficient to keep constant efficiency with increasing number of processors, and hence the parallel algorithm is scalable. Rewriting it in another way, we get the memory-processor relation: $N^{4/3} = c^2 \cdot P$. That is, like the dense LU algorithm in ScaLAPACK [Blackford et al. 1997] and the sparse Cholesky algorithm in PSPACES [Gupta et al. 1997], the parallel efficiency can be maintained constant if the memory per processor is constant.

We now report the measured performance for the model problems with 11-point discretization. Both 3D cubic (NX = NY = NZ) and rectangular (NX, NX/4,

Table IX. Factorization Time (in Seconds), the Megaflop Rate, and Parallel Efficiency (Eff.) on the Cray T3E

	Cubic grids					Rectangular grids						
	Grid	flops			Eff.	Grid size			flops			Eff.
P	size	(10^9)	time	Mflops	(%)	NX	NY	NZ	(10^9)	time	Mflops	(%)
1	29	7.2	56.3	127.2	100	96	24	12	4.5	33.3	133.4	100
2	33	15.9	61.8	257.1	101	110	28	13	9.6	37.6	250.9	94
4	36	26.8	52.0	514.9	101	120	30	15	17.9	36.3	491.5	92
8	41	60.0	60.2	996.5	98	136	34	17	36.6	36.3	923.0	86
16	46	117.9	59.8	1971.5	97	152	38	19	72.7	42.2	1719.6	81
32	51	224.9	64.7	3476.7	85	168	42	21	135.3	43.8	3084.6	72
64	57	444.7	67.3	6612.6	81	184	46	23	236.0	46.6	5059.3	59
128	64	886.4	71.1	12462.9	77	208	52	26	485.6	56.1	8652.2	51

NX/8) grids are used. When increasing the number of processors, we tried to maintain a constant number of operations per processor while keeping as much as possible the same shape of grids. The size of the grids used, the number of operations, the timings, the Megaflop rates, and the parallel efficiency are reported in Table IX.

If the algorithm were perfectly scalable, the parallel runtime would be constant. Because of various overheads, this is not usually true. But from the timing results we see that the time increase is not very much even up to 128 processors. The results on parallel efficiency show that the algorithm is more scalable for cubic grids than for rectangular grids, since the cubic grids represent the best possible regular and balanced problems. Here, the efficiency on p processors is computed as the ratio of the Megaflop rate per processor on p processors over its Megaflop rate on one processor. For cubic grids, the algorithm maintains greater than 95% efficiency up to 16 processors, and greater than 75% efficiency even up to 128 processors. But for rectangular grids, the respective efficiency figures are 80% and 50%.

4.5 Load Balance and Communication/Synchronization Overhead

The efficiency of a parallel algorithm depends mainly on how the workload is distributed and how much time is spent in communication. One way to measure load balance is as follows. Let f_i denote the number of floating-point operations performed on process i. We compute the load balance factor $B = \frac{\sum_i (f_i)}{P \max_i (f_i)}$. In other words, B is the average workload divided by the maximum workload. It is clear that $0 < B \leq 1$, and higher B indicates better load balance. The parallel runtime is at least the runtime of the slowest process, whose workload is highest. In Table X we present the load balance factor B for both factorization and solve phases. As can be seen from the table, the distribution of workload is good for most matrices, except for TWOTONE.

In the same table, we also show the fraction of the runtime spent in communication or synchronization, that is, the parallel overhead. This includes the time for MPI calls and the idle time waiting for a message to be sent or to arrive. The amount of overhead is quite excessive; on 64 processors, more than 50% of the total factorization time is in overhead. For triangular solve, which

Table X. Load Balance and Communication Overhead on 64 Processors Cray T3E

	BBMAT	ECL32	INV-EXTRUSION-1	MIXING-TANK	TWOTONE	WANG4
Load balance measure						
B_{fact}	.78	.83	.87	.92	.47	.84
B_{sol}	.86	.89	.93	.94	.52	.78
Fraction of the time spent in communication and synchronization						
fact	.64	.67	.64	.55	.76	.78
sol	.85	.83	.86	.85	.84	.84

requires a relatively smaller amount of computation, communication and synchronization take more than 85% of the total time. We expect the percentage of overhead will be even higher with more processors, because the total amount of computation is more or less constant.

Although TWOTONE is a relatively large matrix, its factorization does not scale as well as for the other large matrices. One reason is that the present submatrix to process mapping results in very poor load distribution. Another reason is due to poor task scheduling that results in large overhead. When we look further into the overhead, we find that most overhead comes from the idle processors either waiting to receive a column block of L sent from a process column on the left (step (1) in Figure 12), or waiting to receive a row block of U sent from a process row from above (step (2) in Figure 12). Clearly, the critical path of the algorithm is in step (1), which must preserve certain precedence relation between loop iteration steps. Our pipelining method shortens the critical path to some extent, but we expect the length of the critical path can be further reduced by a more sophisticated DAG (task graph) scheduling. For the solve, we find that most overhead comes from the idle processors waiting to receive a message (line 10 in Figure 13). So on each process there is not much work to do but a large amount of idle time. These synchronization overheads also occur in the other matrices, but the problems are not so pronounced as for TWOTONE.

Another problem with TWOTONE is that supernode size (or block size) is very small, only two columns on average. This results in poor uniprocessor performance and low Megaflop rate.

4.6 Large Applications

In this section, we describe two application areas in which SuperLU_DIST has played a critical role. The first application is in the solution of a long-standing problem of scattering in a quantum system of three charged particles. This requires solving the complex, nonsymmetric, and very ill-conditioned linear systems. The largest system solved is of order 8 million. SuperLU_DIST is used in building the block diagonal preconditioners for the CGS iterative solver. The number of CGS iterations ranges between 12 to 35. Since each CGS iteration requires two preconditioning steps, 24 to 70 solutions of the diagonal blocks are required. For a block of size 1 million, SuperLU_DIST takes 1209 seconds to factorize using 64 processors of the IBM SP at NERSC (this is done only once), and it takes 26 seconds to perform triangular solutions (this needs to be done repeatedly in each preconditioning step). The total execution time is about 1 hour.

See Baertschy and Li [2001] for more details. The scientific breakthrough result was reported in a cover article of *Science* [Rescigno et al. 1999].

More recently, we have been collaborating with researchers at the Stanford Linear Accelerator Center to develop alternative eigensolvers for Omega3P, a widely used electromagnetics code in accelerator design. In this application the interior eigenvalues and eigenvectors of a large sparse generalized eigenvalue problem are needed. We integrated SuperLU_DIST with PARPACK [Lehoucq et al. n.d.], a parallel Lanczos code, to construct a shift-and-invert eigensolver. For a system of order 1.3 million, PARPACK needs about 4.5 solves for each eigenpair. For each solve, SuperLU_DIST takes 39 s using 32 processors of the IBM SP at NERSC. The factorization is done once, and takes 553 secs. The total time for finding 10 interior eigenpairs is 42 min.

5. RELATED WORK

Duff and Koster [1999] studied the benefits of using MC64 to permute large entries onto the diagonal in both direct and iterative solvers, and in preconditioning. For the multifrontal direct solver, they showed that using the large-diagonal permutation, the number of delayed pivots were vastly reduced in factorization. In the iterative methods such as GMRES, BiCGSTAB, and QMR using ILU preconditioners, they showed that convergence rate is substantially improved in many cases when the large-diagonal permutation is employed. Benzi et al. [2000] conducted more extensive experiments on the effect of MC64 on preconditioning strategies. Chen [2001] also considered using MC64 to avoid pivoting as much as possible in the ILU methods.

Amestoy et al. [2001a] developed a distributed-memory multifrontal solver, called MUMPS. It is based on the symmetric pattern of $A^T + A$, and performs partial threshold pivoting. It uses partial static mapping based on the elimination tree of $A^T + A$ (1D for the frontal matrices and 2D for the root). The distributed scheduling algorithm for LU factorization is dynamic and asynchronous. We performed a comprehensive comparison between SuperLU_DIST and MUMPS [Amestoy et al. 2001b]. The general observations are: SuperLU_DIST may need one more step of iterative refinement than MUMPS to achieve the same level of accuracy; SuperLU_DIST preserves the sparsity and the asymmetry of the factors better, and usually requires less memory; MUMPS is faster on smaller number of processors (e.g., fewer than 64), but SuperLU_DIST is faster on larger number of processors and shows better scalability.

A few other distributed-memory unsymmetric sparse direct solvers have been developed. Comparing SuperLU_DIST with those solvers remains future work. SPOOLES is a supernodal, left-up-looking solver [Ashcraft and Grimes 1999]. The fill reducing ordering is a hybrid approach called *multisection* [Ashcraft and Liu 1998], which is applied to the structure of $A^T + A$. It performs threshold rook pivoting with both row and column interchanges. The task dependency graph is the elimination tree of $A^T + A$. S+ is a supernodal, right-looking solver [Fu et al. 1998]. The algorithm is based on the following static information. The sparsity pattern of the Householder QR factorization of A contains the union of all sparsity patterns of L and U for all possible row

interchanges [George et al. 1988; George and Ng 1987]. This has been used to do both memory allocation and computation conservatively (on possibly zero entries), but the structural upper bound can be arbitrarily loose, particularly for matrices arising from circuit and device simulations.

6. CONCLUDING REMARKS AND FUTURE WORK

In this paper, we presented the details of the algorithms used in SuperLU_DIST solver. We demonstrated numerical stability of the GESP algorithm, and showed that a scalable implementation is feasible for this algorithm because of the static data structure and scheduling optimizations. Another added benefit of GESP is that it opens new possibilities to study better fill reducing ordering algorithms for unsymmetric LU factorization. Our goal is to have sparse LU factorization as scalable as sparse Cholesky. This is inherently a harder problem than sparse Cholesky, because two different factors L and U are involved. Our future work remains in several areas:

—*Parallel preordering and symbolic analysis.* Steps (1) and (3) of the GESP algorithm (see Figure 1) are still sequential. Although they usually do not take much time, we need to parallelize this step in order to improve memory scalability, if not timewise. The parallel algorithm may be different from the sequential algorithm used in MC64, because MC64 is inherently serial.

—*Improve parallel efficiency of factorization and triangular solves.* Although the solver exhibits good scalability now, the parallel overhead is still large for large numbers of processors (see Section 4.5). Several improvements could be made. For better load balance, we can use more general functions than 2D block cyclic to map submatrices to processors. To reduce the synchronization overhead, we can relax some task scheduling constraints imposed by the current pipelining algorithm. For example, the blocks in a block column can be factorized by the column processes independently if sparsity permits doing so. A more sophisticated scheduling algorithm can be implemented to exploit the parallelism from the elimination DAGs, which could simultaneously schedule independent tasks from multiple steps of the factorization (see Figure 12). We expect these improvements will have a large impact for very sparse and/or very unsymmetric matrices, such as TWOTONE, and for the orderings that give wide and bushy elimination DAGs, such as nested dissection.

  To speed up the triangular solve, we may apply some graph coloring heuristic to reduce the number of parallel steps [Jones and Plassmann 1994]. There are also alternative algorithms other than substitutions, such as those based on partitioned inversion [Alvarado et al. 1993] or selective inversion [Raghavan 1995]. However, these algorithms usually require preprocessing or different matrix distributions than the one used in our factorization. Whether the preprocessing and redistribution will offset the benefit offered by these algorithms will probably depend on the number of right-hand sides.

—*Improve numerical robustness.* More techniques can be used; these include performing iterative refinement with extra precise residuals [Li et al. 2002]

and using dynamic precision during the factorization; see Appendix A of Li and Demmel [2002].

ACKNOWLEDGMENTS

We thank the anonymous referees for the detailed comments which very much improved the presentation.

REFERENCES

ALVARADO, F. L., POTHEN, A., AND SCHREIBER, R. 1993. Highly parallel sparse triangular solution. In *Graph Theory and Sparse Matrix Computation*, A. George, J. R. Gilbert, and J. W. Liu, Eds. Springer-Verlag, New York, NY, 159–190.

AMESTOY, P. R., DAVIS, T. A., AND DUFF, I. S. 1996. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl. 17*, 4, 886–905. Also published as Tech. Rep. TR-94-039, University of Florida, Gainesville, FL.

AMESTOY, P. R. AND DUFF, I. S. 1993. Memory management issues in sparse multifrontal methods on multiprocessors. *Internat. J. Supercomput. Appl. 7*, 1 (Spring), 64–82.

AMESTOY, P. R., DUFF, I. S., L'EXCELLENT, J.-Y., AND KOSTER, J. 2001a. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl. 23*, 1, 15–41.

AMESTOY, P. R., DUFF, I. S., L'EXCELLENT, J.-Y., AND LI, X. S. 2001b. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Trans. Math. Softw. 27*, 4 (Dec.), 388–421.

AMESTOY, P. R., LI, X. S., AND NG, E. G. 2003. Diagonal markowitz scheme with local symmetrization. Tech. rep., Lawrence Berkeley National Laboratory. In preparatioon.

AMESTOY, P. R. AND PUGLISI, C. 2002. An unsymmetrized multifrontal LU factorization. *SIAM J. Matrix Anal. Appl. 24*, 2, 553–569.

ARIOLI, M., DEMMEL, J. W., AND DUFF, I. S. 1989. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl. 10*, 2 (April), 165–190.

ASHCRAFT, C. AND GRIMES, R. G. 1999. SPOOLES: An object oriented sparse matrix library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. (San Antonio, TX. Available online at http://www.netlib.org/linalg/spooles.)

ASHCRAFT, C. AND LIU, J. 1998. Robust ordering of sparse matrices using multisection. *SIAM J. Matrix Anal. Appl. 19*, 816–832.

BAERTSCHY, M. AND LI, X. S. 2001. Solution of a three-body poblem in quantum mechanics. In *Proceedings of SC2001: High Performance Networking and Computing Conference* (Denver, CO).

BENZI, M., HAWS, J. C., AND TUMA, M. 2000. Preconditioning highly indefinite and nonsymmetric matrices. *SIAM J. Sci. Comput. 22*, 1333–1353.

BLACKFORD, L. S., CHOI, J., D'AZEVEDO, E., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. C. 1997. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA.

CHEN, T.-Y. 2001. Preconditioning sparse matrices for computing eigenvalues and computing linear systems of equations. Ph.D. dissertation, Computer Science Division, University of California, Berkeley, Berkeley, CA.

DAVIS, T. A. n.d. University of Florida sparse matrix collection. Available online at http://www.cise.ufl.edu/~davis/sparse.

DAVIS, T. A. AND DUFF, I. S. 1999. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Trans. Math. Softw. 25*, 1, 1–19.

DAVIS, T. A., GILBERT, J. R., LARIMORE, S. I., AND NG, E. 2000. A column approximate minimum degree ordering algorithm. Tech. Rep. TR-00-005, Computer and Information Sciences Department, University of Florida, Gainesville, FL. Submitted to *ACM Trans. Math. Software*.

DEMMEL, J. W. 1997. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA.

DEMMEL, J. W., EISENSTAT, S. C., GILBERT, J. R., LI, X. S., AND LIU, J. W. H. 1999a. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl. 20*, 3, 720–755.

DEMMEL, J. W., GILBERT, J. R., AND LI, X. S. 1999b. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM J. Matrix Anal. Appl. 20*, 4, 915–952.

DUFF, I., ERISMAN, I., AND REID, J. 1986. *Direct Methods for Sparse Matrices*. Oxford University Press, London, England.

DUFF, I., GRIMES, R., AND LEWIS, J. 1992. Users' guide for the Harwell-Boeing sparse matrix collection (release 1). Tech. Rep. RAL-92-086, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, U.K.

DUFF, I. S. AND KOSTER, J. 1999. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl. 20*, 4, 889–901.

FU, C., JIAO, X., AND YANG, T. 1998. Efficient sparse LU factorization with partial pivoting on distributed memory architectures. *IEEE Trans. Parallel Distrib. Syst. 9*, 2, 109–125.

GEORGE, A. 1973. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal. 10*, 345–363.

GEORGE, A., LIU, J., AND NG, E. 1988. A data structure for sparse QR and LU factorizations. *SIAM J. Sci. Stat. Comput. 9*, 100–121.

GEORGE, A. AND LIU, J. W. H. 1981. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, Englewood Cliffs, NJ.

GEORGE, A. AND NG, E. 1987. Symbolic factorization for sparse Gaussian elimination with partial pivoting. *SIAM J. Sci. Stat. Comput. 8*, 6, 877–898.

GILBERT, J. R. 1994. Predicting structures in sparse matrix computations. *SIAM J. Matrix Anal. Appl. 15*, 1 (Jan.), 62–79.

GILBERT, J. R. AND LIU, J. W. 1993. Elimination structures for unsymmetric sparse LU factors. *SIAM J. Matrix Anal. Appl. 14*, 2 (April), 334–352.

GOLUB, G. AND VAN LOAN, C. 1996. *Matrix Computations*, 3rd ed. Johns Hopkins University Press, Baltimore, MD.

GUPTA, A. 2000. WSMP: Watson Sparse Matrix Package. Tech. Rep., IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY. Available online at http://www.cs.umn.edu/~agupta/wsmp.html.

GUPTA, A. 2001. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. Tech. Rep. RC 22137 (99131), IBM Research Center, Yorktown Heights, NY.

GUPTA, A., KARYPIS, G., AND KUMAR, V. 1997. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Trans. Parallel Distrib. Syst. 8*, 502–520.

GUPTA, A. AND KUMAR, V. 1995. Optimally scalable parallel sparse cholesky factorization. In *The 7th SIAM Conference on Parallel Processing for Sci. Comput.* 442–447.

HEATH, M. T. AND RAGHAVAN, P. 1997. Performance of a fully parallel sparse solver. *Internat. J. Supercomput. Appl. 11*, 1, 49–64.

HENDRICKSON, B. AND LELAND, R. 1993. The CHACO User's Guide. Version 1.0. Tech. Rep. SAND93-2339 • UC-405, Sandia National Laboratories, Albuquerque, NM.

HENON, P., RAMET, P., AND ROMAN, J. 1999. A mapping and scheduling algorithm for parallel sparse fan-in numerical factorization. In *EuroPar'99 Parallel Processing*. Lecture Notes in Computer Science, vol. 1685. Springer-Verlag, Berlin, Heidelberg, New York, 1059–1067.

HSL. 2000. A collection of Fortran codes for large scale scientific computation. Available online at http://www.cse.clrc.ac.uk/Activity/HSL.

JONES, M. T. AND PLASSMANN, P. E. 1994. Scalable iterative solution of sparse linear systems. *Parallel Comput. 20*, 753–773.

KARYPIS, G. AND KUMAR, V. 1998. MeTiS—*A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices—Version 4.0*. University of Minnesota, Minneapolis, MN.

KUMAR, V., GRAMA, A., GUPTA, A., AND KARYPIS, G. 1994. *Introduction to Parallel Computing*. Benjamin/Cummings, Redwood City, CA.

LEHOUCQ, R., MASCHHOFF, K., SORENSEN, D., AND YANG, C. n.d. Parallel ARPACK. Available online at http://www.caam.rice.edu/~kristyn/parpack_home.html.

LI, X. S. AND DEMMEL, J. W. 1998. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of SC98: High Performance Networking and Computing Conference* (Orlando, FL).

LI, X. S. AND DEMMEL, J. W. 2002. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. Tech. Rep. LBNL-49388, Lawrence Berkeley National Laboratory, Berkeley, CA.

LI, X. S., DEMMEL, J. W., BAILEY, D. H., HENRY, G., HIDA, Y., ISKANDAR, J., KAHAN, W., KANG, S. Y., KAPUR, A., MARTIN, M. C., THOMPSON, B. J., TUNG, T., AND YOO, D. J. 2002. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw. 28*, 2, 152–205.

LIU, J. W. 1985. Modification of the minimum degree algorithm by multiple elimination. *ACM Trans. Math. Softw. 11*, 141–153.

LIU, J. W. 1990. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl. 11*, 1 (Jan.), 134–172.

MARKOWITZ, H. M. 1957. The elimination form of the inverse and its application to linear programming. *Management Sci. 3*, 255–269.

MPI. n.d. Message Passing Interface (MPI) forum. Available online at http://www.mpi-forum.org/.

OLSHOWKA, M. AND NEUMAIER, A. 1996. A new pivoting strategy for Gaussian elimination. *Linear Algebra Appl. 240*, 131–151.

RAGHAVAN, P. 1995. Efficient parallel sparse triangular solution with selective inversion. Tech. Rep. CS-95-314, Department of Computer Science, University of Tennessee, Knoxville, TN.

RESCIGNO, T. N., BAERTSCHY, M., ISAACS, W. A., AND MCCURDY, C. W. 1999. Collisional breakup in a quantum system of three charged particles. *Science 286*, 5449, 2474–2479.

RIEDY, J. 2003. Parallel bipartite matching for sparse matrix computation. In preparation.

ROTHBERG, E. 1996. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. *SIAM J. Sci. Comput. 17*, 3 (May), 699–713.

SAAD, Y. n.d. SPARSKIT: A basic tool-kit for sparse matrix computations (Version 2). University of Minnesota, Minneapolis, MN. Available online at http://www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html.

SAAD, Y. AND SCHULTZ, M. H. 1986. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput. 7*, 856–869.

SCHENK, O., GÄRTNER, K., AND FICHTNER, W. 2000. Efficient sparse LU factorization with left–right looking strategy on shared memory multiprocessors. *BIT 40*, 1, 158–176.

Received April 2002; revised November 2002; accepted January 2003