# An Overview of SuperLU: Algorithms, Implementation, and User Interface

XIAOYE S. LI

We give an overview of the algorithms, design philosophy, and implementation techniques in the software SuperLU, for solving sparse unsymmetric linear systems. In particular, we highlight the differences between the sequential SuperLU (including its multithreaded extension) and parallel SuperLU_DIST. These include the numerical pivoting strategy, the ordering strategy for preserving sparsity, the ordering in which the updating tasks are performed, the numerical kernel, and the parallelization strategy. Because of the scalability concern, the parallel code is drastically different from the sequential one. We describe the user interfaces of the libraries, and illustrate how to use the libraries most efficiently depending on some matrix characteristics. Finally, we give some examples of how the solver has been used in large-scale scientific applications, and the performance.

Categories and Subject Descriptors: G.1.3 [**Mathematics of Computing**]: Numerical Linear Algebra—*sparse, structured, and very large systems (direct and iterative methods)*; G.4 [**Mathematics of Computing**]: 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

SuperLU contains a set of sparse direct solvers for solving large sets of linear equations $AX = B$ [Demmel et al. 1999b]. Here $A$ is a square, nonsingular, $n \times n$ sparse matrix, and $X$ and $B$ are dense $n \times nrhs$ matrices, where $nrhs$ is the number of right-hand sides and solution vectors. The matrix $A$ need not be symmetric or definite; indeed, SuperLU is particularly appropriate for matrices with very unsymmetric structure. The routines appear in three different libraries: sequential, multithreaded and parallel. They can be linked together in a single application. All three libraries use variations of Gaussian elimination (LU factorization) optimized to take advantage both of sparsity and of computer architecture, in particular memory hierarchy (caches) and parallelism. Below is a brief summary of the three libraries.

This work was supported in part 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. Xiaoye S. Li, xsli@lbl.gov, Lawrence Berkeley National Lab, MS 50F-1650, One Cyclotron Rd., Berkeley, CA 94720.

—**Sequential SuperLU** is designed for uniprocessors with one or more layers of memory hierarchy [Demmel et al. 1999].

—**Multithreaded SuperLU (`SuperLU_MT`)** is designed for shared memory multiprocessors (SMPs), and can effectively use up to 16 or 32 parallel processors on sufficiently large matrices in order to speed up the computation [Demmel et al. 1999a].

—**Distributed SuperLU (`SuperLU_DIST`)** is designed for distributed memory parallel machines, using MPI [MPI ] for interprocess communication. It can effectively use hundreds of parallel processors on sufficiently large matrices [Li and Demmel 1998; 2003].

The kernel algorithm in SuperLU is sparse Gaussian elimination. The high-level algorithm can be summarized as follows:

(1) Compute a *triangular factorization* $P_r D_r A D_c P_c = LU$. Here $D_r$ and $D_c$ are diagonal matrices to equilibrate the system, $P_r$ and $P_c$ are *permutation matrices*. Premultiplying $A$ by $P_r$ reorders the rows of $A$, and postmultiplying $A$ by $P_c$ reorders the columns of $A$. $P_r$ and $P_c$ are chosen to enhance sparsity, numerical stability, and parallelism. $L$ is a unit lower triangular matrix ($L_{ii} = 1$) and $U$ is an upper triangular matrix. The factorization can also be applied to non-square matrices.

(2) Solve $AX = B$ by evaluating $X = A^{-1}B = (D_r^{-1}P_r^{-1}LUP_c^{-1}D_c^{-1})^{-1}B = D_c(P_c(U^{-1}(L^{-1}(P_r(D_rB)))))$. This is done efficiently by multiplying from right to left in the last expression: Scale the rows of $B$ by $D_r$. Multiplying $P_rD_rB$ means permuting the rows of $D_rB$. Multiplying $L^{-1}(P_rD_rB)$ means solving $nrhs$ triangular systems of equations with matrix $L$ by substitution. Similarly, multiplying $U^{-1}(L^{-1}(P_rD_rB))$ means solving triangular systems with $U$.

Table I summarizes the current status of the software. All the routines are implemented in C, with parallel extensions using Pthreads (POSIX threads for shared-memory programming) or MPI (for distributed-memory programming). We provide Fortran interfaces for all three libraries. Sequential `SuperLU` also has a MATLAB interface to the driver via MEX files. In addition to the kernel algorithms aforementioned, we provide routines for performing iterative refinement, estimating the componentwise error bounds, and estimating the condition number.

The error bounds are based on the *componentwise* error analysis [Anderson et al. 1999; Arioli et al. 1989; Demmel 1997; Higham 1996; Oettli and Prager 1964], rather than the normwise one. The componentwise error bounds respect the presence of zero or tiny entries in $A$, and hence are more appropriate for sparse systems. The componentwise relative backward error is given by

$$BERR = \max_i \frac{|b - A \cdot x|_i}{(|A| \cdot |x| + |b|)_i} \ . \tag{1}$$

This means the computed $\hat{x}$ is the exact solution of a slightly perturbed system $(A + E)\hat{x} = b + f$, where $|E_{ij}| \leq BERR \cdot |A_{ij}|$ and $|f_i| \leq BERR \cdot |b_i|$ for all $i$ and $j$. In other words, $E$ and $f$ are small relative perturbations in each entry of $A$ and $b$, respectively. Note that in Equation (1), the denominator may be tiny or exactly zero, resulting in overflow or division-by-zero. In the code, we always test

Table I. SuperLU software status.

| | Sequential SuperLU | SuperLU_MT | SuperLU_DIST |
|---|---|---|---|
| Platform | serial | shared-memory | distributed-memory |
| Language (with Fortran interface) | C | C + Pthreads (or pragmas) | C + MPI |
| Data type | real/complex single/double | real double | real/complex double |

whether the $i$th component of the denominator is smaller than a threshold. If it is, the quantity $n \cdot sfmin$ is added to the $i$th component of both the numerator and the denominator before division, where $sfmin$ is the safe minimum such that $1/sfmin$ does not overflow.

The forward error bound is the bound on the accuracy of the solution:

$$||x - \hat{x}||_\infty / ||x||_\infty \leq FERR \ .$$

This depends on the conditioning of the system as well as $BERR$. In practice, $FERR$ is calculated by the following formula:

$$FERR = \frac{|| \ |A^{-1}| \ f \ ||_\infty}{||\hat{x}||_\infty} \ . \tag{2}$$

Here, $f$ is a nonnegative vector whose components are computed as

$$f_i = |\hat{r}|_i + m_i \ \varepsilon \ (|A| \ |\hat{x}| + |b|)_i \ , \tag{3}$$

$\hat{r}$ is the computed value of the residual $b - A\hat{x}$, and $m_i$ is the number of nonzeros in row $i$ of $A$. The norm in the numerator is estimated using the same algorithm that estimates the condition number.

Note that for SuperLU_DIST where we use static pivoting instead of partial pivoting, we have yet to include the forward error bound estimation in the software. This is because the error analysis is less understood in this case and remains our future work.

An efficient sparse Gaussian elimination procedure depends on good ordering of equations and variables to minimize fill, a fast symbolic algorithm to determine the exact nonzero structure of the triangular factors $L$ and $U$, and fast numerical factorization with cheap accommodation of numerical pivoting. For unsymmetric matrices, we usually use a column ordering that is obtained from a symmetric fill-reducing ordering on a symmetrized matrix $A^TA$ or $A^T + A$. The ordering heuristics can be minimum-degree-like or nested-dissection-like. Row permutation is independently performed for numerical stability. The symbolic factorization is based on Gilbert-Peierls's depth-first search traversal of the graph, which in time is proportional to arithmetic operations [Gilbert and Peierls 1988]. We sped up this process by combining the supernodal graph with Eisenstat-Liu's symmetric pruning [Eisenstat and Liu 1992; Demmel et al. 1999], so that the resulting graph is much coarser. The numerical factorization is based on block submatrix updating, which effectively uses Level 3 BLAS. We also use 2D partitioning of supernodes (loop blocking) to avoid cache thrashing and to increase parallelism.

From both the users' and the algorithm's points of view, SuperLU_MT is very similar to sequential SuperLU, therefore, we will not give further details on SuperLU_MT. Instead, we will focus on sequential SuperLU and SuperLU_DIST. The rest of this

Table II. Major differences between sequential SuperLU and SuperLU_DIST.

| | sequential SuperLU | SuperLU_DIST |
|---|---|---|
| Numerical pivoting to choose $P_r$ | partial pivoting with threshold | static pivoting |
| Sparsity ordering to choose $P_c$ | $A^TA$–based | $(A^T+A)$–based |
| BLAS kernel | BLAS-2.5 | BLAS-3 |
| Update ordering in GE | left-looking, supernode-panel | right-looking, 2D block |

paper is organized as follows. In Section 2, we describe the main algorithmic and implementational differences between the two libraries. In Section 3, we describe the user interfaces for the two libraries. Some of the interfaces are common to both, and some are different. In Section 4, we illustrate how SuperLU can be used in solving large linear systems and eigensystems arising from scientific applications. We also highlight the performance of the solver. Finally, in Section 5, we discuss future work.

## 2. DIFFERENCES BETWEEN SEQUENTIAL AND PARALLEL SUPERLU

Although the high-level algorithms in the two libraries are the same: sparse Gaussian elimination followed by the triangular solutions, there exist significant differences in the actual implementation. Thus the performance is quite different even when they are run on one processor. Their main differences are summarized in Table II. In the following subsections, we describe in more detail each difference.

### 2.1 Numerical Pivoting

The goal of numerical pivoting is to control the growth in the size of the elements in the factors so to avoid loss of accuracy. A commonly used strategy in dense LU factorization is partial pivoting, which swaps the largest element of the row or column with the diagonal at each step of elimination. In sparse factorizations, however, it is common to relax the pivoting rule to trade for better sparsity and parallelism.

Sequential SuperLU uses *partial pivoting with diagonal threshold*. The row permutation $P_r$ is determined during factorization. Suppose we have factorized the first $j-1$ columns of $A$, and are seeking a pivot for column $j$. Let $a_{mj}$ be a largest entry in magnitude on or below the diagonal of the partially factored $A$: $|a_{mj}| = \max_{i \geq j} |a_{ij}|$. Depending on a threshold $u$ ($0.0 \leq u \leq 1.0$) selected by the user, the code may use the diagonal entry $a_{jj}$ as the pivot in column $j$ as long as $|a_{jj}| \geq u\,|a_{mj}|$ and $a_{jj} \neq 0$, or else use $a_{mj}$. If the user sets $u = 1.0$, $a_{mj}$ (or an equally large entry) will be used as the pivot; this corresponds to the classical partial pivoting. If the user has ordered the matrix so that choosing diagonal pivots is particularly good for sparsity or parallelism, then smaller values of $u$ tend to choose those diagonal pivots, at the risk of less numerical stability. Selecting $u = 0.0$ guarantees that the pivot on the diagonal will be chosen, unless it is zero. The code can also use a user-input $P_r$ to choose pivots, as long as each pivot satisfies the threshold for each column. The error bound $BERR$ measures how much stability is actually lost. Below is the pseudo-code to choose pivot for column $j$:

(1) compute $thresh = u \cdot |a_{mj}|$, where $|a_{mj}| = \max_{i \geq j} |a_{ij}|$;
(2) **if** user specifies pivot row $k$ **and** $|a_{kj}| \geq thresh$ **and** $a_{kj} \neq 0$ **then**
pivot row $= k$;

**else if** $|a_{jj}| \geq thresh$ **and** $a_{jj} \neq 0$ **then**
    pivot row $= j$;
**else**
    pivot row $= m$;
**endif**;

It is hard to get satisfactory execution speed with partial pivoting on distributed memory machines, because of the fine-grain communication and the dynamic data structures required. Some codes implement partial pivoting, including MUMPS [Amestoy et al. 2003] and SPOOLES [Ashcraft and Grimes 1999]. `SuperLU_DIST` on the other hand uses *static pivoting*. Here, $P_r$ is chosen before factorization and based solely on the values of the original $A$; it remains fixed during factorization. We use a maximum weighted matching algorithm and the code `MC64` developed by Duff and Koster [Duff and Koster 1999]. The algorithm 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 $\pm 1$ and each off-diagonal entry is bounded by 1 in magnitude. Since $P_r$ is chosen based on $A$ and not on the Schur complement at each elimination step, the method is potentially less stable than partial pivoting. On the basis of empirical evidence, when we combine this approach with diagonal scaling, setting very tiny pivots to larger values, and iterative refinement, the algorithm is as stable as partial pivoting for most matrices that have occurred in actual applications. The detailed numerical experiments can be found in [Li and Demmel 2003].

In this static pivoting approach, since both row and column orderings ($P_r$ and $P_c$) are fixed before factorization, the symbolic factorization is performed before numerical factorization. We can perform extensive off-line optimization for the data layout, load balance, and communication schedule [Grigori and Li 2002]. The price is a higher risk of numerical instability, which is mitigated by the several other numerical techniques aforementioned. In the case when static pivoting does not give good guarantee of accuracy, then at least an indication of the presence of numerical problems is given by the error bound $BERR$.

Note that static pivoting can also be beneficial to shared memory sparse solvers (see for example `PARDISO` code [Schenk and Gärtner 2004]).

## 2.2 Sparsity Ordering

For unsymmetric factorizations, preordering for sparsity is less well understood than that for Cholesky factorization. Many unsymmetric ordering methods use symmetric ordering techniques on a symmetrized matrix (e.g., $A^T A$ or $A^T + A$). This attempts to minimize certain upper bounds on the actual fills. Which symmetrized matrix to use strongly depends on how the numerical pivoting is performed.

For sequential `SuperLU` with partial pivoting, we use $A^T A$–based ordering algorithms. The reason is as follows. Consider the LU factorization with row interchanges $P_r A = LU$. Also consider the Cholesky factorization $A^T A = R^T R$, and the QR factorization $A = QR$ computed by 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

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

nonzero structure of $A$, because the row interchanges chosen 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 Ng 1987]. Therefore, a good symmetric ordering $P_c$ on $A^TA$ that preserves the sparsity of $R$ can be applied to the columns of $A$, forming $AP_c^T$, so that the LU factorization of $AP_c^T$ is likely to be sparser than that of the original $A$. This can be seen from the relation $P_c(A^TA)P_c^T = (AP_c^T)^T(AP_c^T)$. Two conditions are needed for this to work well: the structure of $A^TA$ must be sparse and it must be possible to find a good permutation $P_c$ such that the factorization of $P_c(A^TA)P_c^T$ is sparse.

In `SuperLU_DIST`, we first use an *a priori* row permutation $P_r$ computed by `MC64` [Duff and Koster 1999] to form $P_rA$. With this pre-determined row permutation, we then consider the sparsity ordering for $P_rA$. In this context, the $A^TA$–based ordering methods may be too generous, since they attempt to account for all possible row interchanges, whereas we already have a fixed row permutation. Therefore, we use $(A^T + A)$–based ordering methods.[2] The reason is as follows. The symbolic Cholesky factor of $A^T + A$ is a much tighter upper bound on the structures of $L$ and $U$ than that of $A^TA$ when the pivots are chosen on the diagonal. Note that after we find $P_c$, we actually perform a symmetric permutation $P_c(P_rA)P_c^T$ so that the diagonal entries of the permuted matrix remain the same as those in $P_rA$, and they are larger in magnitude than the off-diagonal entries.[3] Our experiments showed that the amount of fill can be reduced by more than a factor of two with $(A^T + A)$–based ordering compared to $A^TA$–based ordering [Li and Demmel 2003, Table II]. More recently, we realized that we can further improve the ordering quality for `SuperLU_DIST` if we respect the asymmetry of $A$'s structure [Amestoy et al. 2003]. This new ordering scheme does not require any symmetrization of $A$, and works directly on $A$ itself. The scheme is similar to the Markowitz scheme [Markowitz 1957] but limits the pivot search to the diagonal entries. 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 showed that the new ordering method reduces the amount of fill by 10-15% on average for very unsymmetric matrices, when compared to applying AMD to $A^T + A$. A generalization of this algorithm without restricting the pivots on the diagonal is also under investigation [Pralet et al. 2004]. We plan to incorporate these new ordering codes into `SuperLU_DIST` when they are ready.

### 2.3 BLAS Kernel

Both factorization algorithms in sequential `SuperLU` and `SuperLU_DIST` are based on unsymmetric supernodes [Demmel et al. 1999]. 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 below the triangular block. Matrix $U$ is considered rowwise partitioned by the same supernodal boundaries. But due to the lack of symmetry,

[2]For simplicity, we still refer to $A^T + A$, with the understanding that the $A$ here corresponds to $P_rA$ for original $A$.

[3]Now the final row permutation is $P_cP_r$, which corresponds to $P_r$ described in Section 1.

the partitions of $U$ need not have the nice dense structure of those of $L$. The nonzero structure of $U$ consists of dense column segments of various lengths. The sparse storage schemes for $L$ and $U$ are described in [Demmel et al. 1999] for sequential `SuperLU` and [Li and Demmel 2003] for `SuperLU_DIST`. The most time-consuming kernel in factorization is the following block update:

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

Since $L$ is partitioned by supernodes, each block $L(I,K)$ has a regular dense structure in the compressed format. But block $U(K,J)$ is not so regular; it contains dense vectors of different lengths. Because of this, it is not straightforward to call the dense matrix-matrix multiplication routine (Level 3 BLAS).

In sequential `SuperLU`, we perform multiple calls to the matrix-vector multiplication routine GEMV in Level 2 BLAS, for each vector in the $U(K,J)$ block. We designed tunable blocking parameters to ensure that the source block $L(I,K)$ is small enough to fit in the fastest cache. So we spend time to fetch $L(I,K)$ only once across the multiple calls to GEMV. We call this BLAS-2.5 kernel. The detailed analysis of the blocking parameters can be found in [Demmel et al. 1999].

In `SuperLU_DIST`, because of the new algorithmic and data structures, it is easier to use Level 3 BLAS. This is achieved by padding zeros to the beginning of some dense column segments in $U(K,J)$, to make all the column vectors the same length, and then copying them into contiguous memory. After zero-padding and copying, we can call GEMM directly. The zero-padding results in extra floating-point operations, but copying is almost free because the data must be loaded in the cache anyway. Overall, the benefit of using GEMM well offsets the cost of the extra floating-point operations. We observed 20% to 40% uniprocessor performance improvement [Li and Demmel 2003]. We believe the main advantage of GEMM over multiple calls of GEMV comes from exploiting the register blocking technique. In the future, we will implement this scheme in sequential `SuperLU`, and evaluate whether there is benefit compared with the BLAS-2.5 kernel in that context.

### 2.4 Task Ordering in Gaussian Elimination

The Gaussian elimination algorithm can be organized in different ways, such as left-looking (fan-in) or right-looking (fan-out). These variants are mathematically equivalent under the assumption that the floating-point additions and multiplications are associative. They perform the same number of floating-point operations, but have very different memory access and communication patterns. In our blocking framework, the outer loop of the algorithm involves a block row or column of the matrix. The pseudo-code for the left-looking algorithm is given in Algorithm 1.

ALGORITHM 1. *Left-looking Gaussian elimination*

**for** *block* $K = 1$ **to** $N$ **do**
  *(1) Compute* $U(1:K-1,K)$
    *(via a sequence of triangular solves)*
  *(2) Update* $A(K:N,K) \leftarrow A(K:N,K) - L(1:N,1:K-1) \cdot U(1:K-1,K)$
    *(via a sequence of calls to GEMM)*
  *(3) Factorize* $A(K:N,K) \rightarrow L(K:N,K)$
    *(may involve pivoting)*
**end for**

The pseudo-code for the right-looking algorithm is given in Algorithm 2.

ALGORITHM 2. *Right-looking Gaussian elimination*

**for** *block* $K = 1$ **to** $N$ **do**
*(1) Factorize* $A(K:N, K) \rightarrow L(K:N, K)$
*(may involve pivoting)*
*(2) Compute* $U(K, K+1:N)$
*(via a sequence of triangular solves)*
*(3) Update* $A(K+1:N, K+1:N) \leftarrow$
$A(K+1:N, K+1:N) - L(K+1:N, K) \cdot U(K, K+1:N)$
*(via a sequence of calls to GEMM)*
**end for**

For sequential `SuperLU`, we chose to use left-looking algorithm for the following reasons.

—In each step, the sparsity changes are restricted to the $K$th block column, instead of the whole trailing submatrix.
—There are more memory "read" operations than "write" operations in Algorithm 1. This is better for most modern cache-based computer architectures, because "write" tends to be more expensive in order to maintain cache coherency.

For `SuperLU_DIST`, we chose to use right-looking algorithm for the following reasons, mainly motivated by scalability.

—The sparsity structure can be determined before numerical factorization because of static pivoting.
—The right-looking algorithm fundamentally has more parallelism: at step (3) of Algorithm 2, all the GEMM updates to the trailing submatrix are independent and so can be done in parallel. On the other hand, each step of the left-looking algorithm involves operations that need to be carefully sequenced, which requires a sophisticated pipeline mechanism to exploit parallelism across multiple loop steps.
—In each step of Algorithm 2, we only need a small amount of buffer space for transferring a block column of $L$ and a block row of $U$, to facilitate the trailing submatrix update. Whereas in Algorithm 1, a block column of $L$ and $U$ will be needed by different loop steps, hence we either need to transfer them many times or need a large buffer space to hold many previously-transferred block columns.

## 3. USER INTERFACES

In this section, we present the user interfaces of the SuperLU libraries. Section 3.1 addresses the interface issues common to both sequential `SuperLU` and `SuperLU_DIST`. Sections 3.2 and 3.3 contain the interface issues specific in sequential `SuperLU` and `SuperLU_DIST`, respectively.

### 3.1 Interfaces Common to Both Sequential `SuperLU` and `SuperLU_DIST`

3.1.1 *Sparse Matrix Data Structure.* The principal data structure for a matrix is `SuperMatrix`, which is defined in `SRC/supermatrix.h`. Figure 1 shows the specification of the `SuperMatrix` structure. The `SuperMatrix` structure contains two

levels of fields. The first level defines the three orthogonal properties of a matrix which are independent of how it is stored in memory: storage type (`Stype`) indicates the type of the compressed storage scheme in `*Store`; data type (`Dtype`) encodes the four precisions; mathematical type (`Mtype`) specifies some mathematical properties. The second level (`*Store`) points to the actual storage used to store the matrix. We associate with each `Stype SLU_XX` a storage format called `XXformat`, such as `NCformat`, `SCformat`, etc. The reader may refer to the Users' Guide for the memory layout of each storage format [Demmel et al. 1999b].

```
typedef struct {
    Stype_t Stype; /* Storage type: indicates the storage format of *Store. */
    Dtype_t Dtype; /* Data type. */
    Mtype_t Mtype; /* Mathematical type */
    int  nrow;     /* number of rows */
    int  ncol;     /* number of columns */
    void *Store;   /* pointer to the actual storage of the matrix */
} SuperMatrix;

typedef enum {
    SLU_NC,        /* column-wise, not supernodal (a.k.a. CCS) */
    SLU_NR,        /* row-wise, not supernodal (a.k.a. CRS) */
    SLU_SC,        /* column-wise, supernodal */
    SLU_SR,        /* row-wise, supernodal */
    SLU_NCP,       /* column-wise, not supernodal, permuted by columns
                      (After column permutation, the consecutive columns of
                       nonzeros may not be stored contiguously. */
    SLU_DN,        /* Fortran style column-wise storage for dense matrix */
    SLU_NR_loc     /* distributed compressed row format */
} Stype_t;

typedef enum {
    SLU_S,         /* single */
    SLU_D,         /* double */
    SLU_C,         /* single-complex */
    SLU_Z          /* double-complex */
} Dtype_t;

typedef enum {
    SLU_GE,        /* general */
    SLU_TRLU,      /* lower triangular, unit diagonal */
    SLU_TRUU,      /* upper triangular, unit diagonal */
    SLU_TRL,       /* lower triangular */
    SLU_TRU,       /* upper triangular */
    SLU_SYL,       /* symmetric, store lower half */
    SLU_SYU,       /* symmetric, store upper half */
    SLU_HEL,       /* Hermitian, store lower half */
    SLU_HEU        /* Hermitian, store upper half */
} Mtype_t;
```

Fig. 1. `SuperMatrix` data structure.

The `SuperMatrix` type so defined can accommodate various types of matrix structures and the appropriate operations to be applied on them. Although currently

SuperLU implements only a subset of this collection (mostly related to general unsymmetric matrices), the structure is extensible to include, for example, symmetric capabilities in the future.

3.1.2 `Options` *Argument.* The `options` argument is the input argument to control the behaviour of the libraries. The user can tell the solvers how the linear systems should be solved based on some known characteristics of the system. For example, for diagonally dominant matrices, choosing the diagonal pivots ensures stability; there is no need for numerical pivoting (i.e., $P_r$ can be an Identity matrix). In another situation where a sequence of matrices with the same sparsity pattern need be factorized, the column permutation $P_c$ (and also the row permutation $P_r$, if the numerical values are similar) need be computed only once, and reused thereafter. In these cases, the solvers' performance can be much improved over using the default settings. `Options` is implemented as a C structure containing the following fields (some may be used only by sequential `SuperLU`, and some only by `SuperLU_DIST`):

—`Fact`
Specifies whether or not the factored form of the matrix $A$ is supplied on entry, and if not, how the matrix $A$ will be factorized based on the previous history, such as factor from scratch, reuse $P_c$ and/or $P_r$, or reuse the data structures of $L$ and $U$.

—`Trans`
Specifies whether to solve the transposed system.

—`Equil`
Specifies whether to equilibrate the system (scale $A$'s rows and columns to have unit norm).

—`ColPerm`
Specifies how to permute the columns of the matrix for sparsity preservation.

—`IterRefine`
Specifies whether to perform iterative refinement, and in what precision to compute the residual.

—`PrintStat`
Specifies whether to print the solver's statistics.

—`DiagPivotThresh` (only for sequential `SuperLU`)
Specifies the threshold used for a diagonal entry to be an acceptable pivot.

—`SymmetricMode` (only for sequential `SuperLU`)
Specifies whether to use symmetric mode.

—`RowPerm` (only for `SuperLU_DIST`)
Specifies how to permute the rows of the matrix for numerical stability.

—`ReplaceTinyPivot` (only for `SuperLU_DIST`)
Specifies whether to replace the tiny diagonals by $\sqrt{\varepsilon} \cdot ||A||$ during the LU factorization.

—`SolveInitialized` (only for `SuperLU_DIST`)
Specifies whether the initialization has been performed to the triangular solve.

—RefineInitialized (only for SuperLU_DIST)
Specifies whether the initialization has been performed to the sparse matrix-vector multiplication routine needed in the iterative refinement.

The routine set_default_options() sets the following default values for sequential SuperLU:

```
Fact              = DOFACT          /* factor from scratch */
Trans             = NO
Equil             = YES
ColPerm           = COLAMD
DiagPivotThresh   = 1.0             /* partial pivoting */
SymmetricMode     = NO
IterRefine        = NO
PrintStat         = YES
```

The routine set_default_options_dist() sets the following default values for SuperLU_DIST:

```
Fact              = DOFACT          /* factor from scratch */
Trans             = NO
Equil             = YES
ColPerm           = MMD_AT_PLUS_A
RowPerm           = LargeDiag       /* use MC64 */
ReplaceTinyPivot  = YES
IterRefine        = DOUBLE
SolveInitialized  = NO
RefineInitialized = NO
PrintStat         = YES
```

The users can reset each default value according to their needs.

3.1.3 *Ordering Option.* Finding a good ordering to preserve the sparsity of the factors has been an active research area. Many algorithms have been proposed, and high quality codes based on some of those algorithms are also available. It is impossible to incorporate all these algorithms and codes into SuperLU. Right now, SuperLU contains only two minimum degree ordering algorithms, one is due to Liu and the code is called MMD [Liu 1985], another is due to Davis et al. and the code is called COLAMD [Davis et al. 2000] (an $A^TA$–based ordering method). In addition, the library has a flexible interface so that the user can easily plug in any other ordering algorithm. Here is how it works. The options.ColPerm field can take the following values:

—NATURAL: use natural ordering (i.e., $P_c = I$).

—MMD_AT_PLUS_A: use minimum degree ordering on the structure of $A^T + A$.

—MMD_ATA: use minimum degree ordering on the structure of $A^TA$.

—COLAMD: use approximate minimum degree column ordering.

—MY_PERMC: use the ordering given in the permutation vector perm_c[], which is input by the user.

If `options.ColPerm` is set to the last value, the library will use the permutation vector obtained from any other ordering algorithm. For example, the nested-dissection type of ordering codes include `MeTiS` [Karypis and Kumar 1998], `CHACO` [Hendrickson and Leland 1993] and `SCOTCH` [Pellegrini 2001]. SuperLU also contains user-callable routines to form the structure of $A^T + A$ or $A^T A$. These routines are named `at_plus_a()` and `getata()`.

3.1.4 *User-tunable Parameters Related to Performance.* SuperLU chooses such machine-dependent parameters as block size by calling an inquiry function `sp_ienv()`, which may be set to return different values on different machines. The declaration of this function is

```
int sp_ienv(int ispec);
```

`Ispec` specifies the parameter to be returned, (See [Demmel et al. 1999] for their definitions.)

ispec = 1: the panel size ($w$), i.e., the number of consecutive columns being factorized at a time.
= 2: the relaxation parameter to control supernode amalgamation ($relax$)
= 3: the maximum allowable size for a supernode ($maxsup$)
= 4: the minimum row dimension for 2-D blocking to be used ($rowblk$)
= 5: the minimum column dimension for 2-D blocking to be used ($colblk$)
= 6: the estimated fills factor for L and U, compared with A

Sequential `SuperLU` uses all six parameters, whereas `SuperLU_DIST` uses only three of them, which are 2, 3 and 6. Users are encouraged to modify this subroutine to set the appropriate values for their own local environments.

The $relax$ parameter (2) allows several consecutive columns ($\leq relax$) at the bottom of the elimination tree to be amalgamated into a supernode, and the supernode structure is the union of the structures of the columns. That is, after padding explicit zeros, we will get supernodes of larger size. This parameter is usually set between 4 and 10, which gives better performance and not too much more fill.

The fill estimate parameter (6, call it `FILL`) is used differently in sequential `SuperLU` and `SuperLU_DIST`. In sequential `SuperLU`, the number of nonzeros in $L$ and $U$ is not known a priori. So in the beginning we allocate arrays for $L$ and $U$ of size `FILL*nnz(A)`. If this is not enough, we expand each array dynamically. If this value is too large, there will be too much wasted memory. If it is too small, there will be more memory expansions. In practice, setting it to be 20 works quite well. In `SuperLU_DIST` with static pivoting, the symbolic factorization is separate from the numerical factorization. This value is used only in the symbolic factorization, where a much coarser graph is involved. Therefore, a smaller value, say 4 or 5, is usually sufficient.

For the other three blocking parameters (3, 4 and 5), the optimal values depend mainly on the cache size and the BLAS speed. If your system has a very small cache, or if you want to efficiently utilize the closest cache in a multilevel cache organization, you should pay special attention to these parameter settings. As a general rule of thumb, you need large blocks for better BLAS performance. On

the other hand, if the blocks are larger than the cache, the BLAS 2.5 in sequential `SuperLU` will not perform well. In [Demmel et al. 1999], we described a detailed methodology for setting these parameters for sequential `SuperLU`. For `SuperLU_DIST`, in addition to the cache performance, block size also affects load balance and amount of parallelism. Relatively smaller blocks are preferable in this case.

We ackowledge that automatic tuning for block size still remains an open research question, and is especially difficult for a parallel environment.

3.1.5 *Example Programs.* In the source code distribution, the `EXAMPLE/` directory contains several examples of how to use the driver routines. The examples illustrate the following usages:

—solve a system once

—solve different systems with the same $A$, but different right-hand sides

—solve different systems with the same sparsity pattern of $A$

—solve different systems with the same sparsity pattern and similar numerical values of $A$

Except for the one-time solution case, all the other examples can reuse some of the data structures obtained from a previous factorization, hence, save some time compared with factorizing $A$ from scratch. The users can easily modify these examples to fit their needs.

## 3.2 User Interfaces of Sequential `SuperLU`

3.2.1 *Driver Routines.* For each precision, there are two types of driver routines. The driver routines can handle both column- and row-oriented storage schemes.

—A simple driver `dgssv()`, which solves the system $AX = B$ by factorizing $A$ and overwriting $B$ with the solution $X$.

—An expert driver `dgssvx()`, which, in addition to the above, also performs the following functions depending on the `options` argument:
  —solve $A^T X = B$;
  —equilibrate the system if $A$ is poorly scaled;
  —estimate the condition number of $A$, check for near-singularity, and check for pivot growth;
  —refine the solution and compute forward and backward error bounds.

We expect that most users can simply use these driver routines to fulfill their tasks without the need to directly call the computational routines.

3.2.2 *Symmetric Mode.* In many applications, the matrix $A$ may be diagonally dominant or nearly so. In this case, pivoting on the diagonal is sufficient for stability and is preferable for sparsity to off-diagonal pivoting. To do this, the user can set a small (less-than-one) diagonal pivot threshold (e.g., 0.0, 0.01) and choose an $(A^T + A)$–based column permutation algorithm. We call this setting *symmetric mode*. Note that, when a diagonal entry is smaller than the threshold, the code will still choose an off-diagonal pivot. That is, the row permutation $P_r$ need not be the Identity.

One performance inefficiency may arise in the symmetric mode. This is related to the postordering of the column elimination tree (i.e., elimination tree of $A^TA$) [Demmel et al. 1999, Sections 2.3 and 2.4]. Recall that the postordering of the column elimination tree serves two purposes:

(1) It brings together larger unsymmetric supernodes;

(2) It allows several consecutive columns at the bottom of the elimination tree to be treated as a relaxed supernodes.

It is shown that, without supernode relaxation (2), permuting the matrix columns using this postorder does not change the sparsity of the $L$ and $U$ factors [Demmel et al. 1999, Theorem 2.2]. Let $P_t$ denote the permutation matrix from this tree postordering, $P_{c_1}$ denote the permutation matrix from an $A^TA$–based ordering algorithm, and $P_{c_2}$ denote the permutation matrix from an $(A^T + A)$–based ordering algorithm. `SuperLU` actually performs the factorizations $P_r A P_{c_1} P_t = LU$ or $P_r A P_{c_2} P_t = LU$.

When (2) is introduced and $P_{c_1}$ is used, the number of structural zeros introduced is still well restrained. This is because the objective of $P_{c_1}$ ordering is to minimize the fill-ins of the Cholesky factor of $P_{c_1}^T A^T A P_{c_1}$, and the column elimination tree of $AP_{c_1}$ (i.e., the elimination tree of $(AP_{c_1})^T(AP_{c_1})$) is defined consistently to model the elimination of $P_{c_1}^T A^T A P_{c_1}$. (The column elimination tree is invariant under row permutation.) However, when (2) is used in the symmetric mode (using $P_{c_2}$ ordering), there can be many more structural zeros generated throughout the factorization. This is because the objective of $P_{c_2}$ ordering is to minimize the fill-ins of the Cholesky factor of $P_{c_2}^T(A^T + A)P_{c_2}$, and the column elimination tree of $AP_{c_2}$ (i.e., the elimination tree of $(AP_{c_2})^T(AP_{c_2})$) is not a consistent model to capture the updating relations. Then, the ordering given by $P_{c_2}P_t$ may destroy the original minimization objective, and some structural zeros may be greatly propagated, resulting in an amount of fill far bigger than what was attempted to achieve by the $P_{c_2}$ ordering algorithm. On the other hand, we cannot use the elimination tree of $P_{c_2}^T(A^T + A)P_{c_2}$, because that tree is not invariant under row permutation and the row permutation $P_r$ is not the Identity whenever a diagonal entry does not satisfy the numerical threshold.

We recently improved the relaxation algorithm for the symmetric mode: we use the original heap-ordered column elimination tree to identify the relaxed supernodes *without performing postordering* (i.e., $P_t = I$). Only when the nodes in a subtree are numbered consecutively do we consider this subtree as a relaxed supernode. We call this a *weak relaxation* scheme, or a more conservative approach. Compared to the postorder-based relaxation, the weak relaxation may give fewer relaxed supernodes, but it prevents from catastrophic propagation of the structural zeros. Table III shows an example matrix Zhao1[4] with two relaxation schemes. The weak relaxation gives less than one-third of the fill-ins, and is much more effective in preserving sparsity in the symmetric mode.

[4]Available at http://www.cise.ufl.edu/~davis/sparse/Zhao

Table III. Fill-ins for matrix Zhao1, with diagonal pivoting and MMD($A^T + A$) ordering.

| | relaxation with postorder | *weak relaxation without postorder* |
|---|---|---|
| $relax = 1$ (no relaxation) | 8051345 | 8051345 |
| $relax = 4$ | 29550911 | 8598797 |

### 3.3 User Interfaces of SuperLU_DIST

3.3.1 *Input Formats.* There are two input interfaces for matrices $A$ and $B$. One is the global interface, another is an entirely distributed interface.

In the global interface, $A$ and $B$ are globally available (replicated) on all the processes. The storage type for $A$ is SLU_NC, as in the sequential case (see Section 3.1.1). The user-callable routines with this interface all have the names "xxxxxxx_ABglobal".

In the distributed interface, both $A$ and $B$ are distributed among all the processes. They use the same distribution based on block rows. That is, each process owns a block of consecutive rows of $A$ and $B$. Each local part of the sparse matrix $A$ is stored in a compressed row format, called SLU_NR_loc storage type. It is known as *distributed compress row storage.*

For better scalability, $L$ and $U$ are represented as 2D block matrices, and are distributed in a 2D block-cyclic fashion [Li and Demmel 2003]. The users do not need to understand the $L$ and $U$ data structures. The library contains routines to re-distribute the input $A$ into the $L$ and $U$ structures. Recently, we conducted performance study for both global and distributed interfaces, and found that the distributed interface is as fast as the global interface on the IBM SP [Li and Wang 2003].

3.3.2 *SuperLU 2D Process Grid and MPI Communicator.* SuperLU_DIST uses MPI [MPI ] for interprocess communication. All MPI applications begin with a default communication domain that includes all processes, say $N_p$, of this parallel job. The default communicator MPI_COMM_WORLD represents this communication domain. The $N_p$ processes are identified as a linear array of process IDs in the range $0 \ldots N_p - 1$. SuperLU_DIST does not use MPI_COMM_WORLD for its communicator, instead, it creates a new process group derived from an existing group using $N_g$ MPI processes. This way, the message passing calls within SuperLU_DIST will be isolated from those in other libraries or in the user's code. We map the 1D array of $N_g$ processes into a logical 2D process grid. This grid has nprow process rows and npcol process columns, such that nprow $\times$ npcol $= N_g$. A process can be referred to either by its rank in the new group or by its coordinates within the grid. In the beginning of the program, te user needs to call either superlu_gridinit() or superlu_gridmap() to set up the process grid. The routine superlu_gridinit() maps the existent processes to a 2D process grid.

```
superlu_gridinit(MPI_Comm Bcomm, int nprow, int npcol,
                 gridinfo_t *grid);
```

This process grid will use the first nprow $\times$ npcol processes from the base MPI communicator Bcomm. The processes are assigned to the grid in a row-major ordering. The input argument Bcomm is an MPI communicator representing the existent base group upon which the new group is formed. For example, it can be

MPI_COMM_WORLD. The output argument grid represents the derived group to be used in SuperLU_DIST. Grid is a structure containing the following fields:

```
struct {
    MPI_Comm comm;        /* MPI communicator for this group */
    int iam;              /* my process rank in this group   */
    int nprow;            /* number of process rows          */
    int npcol;            /* number of process columns       */
    superlu_scope_t rscp; /* process row scope               */
    superlu_scope_t cscp; /* process column scope            */
} grid;
```

In the *LU* factorization, some communications occur only among the processes in a row (column), not among all processes. For this purpose, we introduce two process subgroups, namely rscp (row scope) and cscp (column scope). For rscp (cscp) subgroup, all processes in a row (column) participate in the communication.

For some applications, such as block-diagonal preconditioning (see Section 4), it is desirable to divide the processes into several subgroups, each of which solves a distinct linear system. Thus, we cannot simply use the first nprow×npcol processes to define the grid. We can use superlu_gridmap() to create a grid with processes of arbitrary ranks.

```
superlu_gridmap(MPI_Comm Bcomm, int nprow, int npcol,
                int usermap[], int ldumap, gridinfo_t *grid);
```

The array usermap[] contains the ranks of the processes to be used in the newly created grid. usermap[] is indexed like a Fortran-style 2D array with ldumap as the leading dimension. So usermap[i+j*ldumap] (i.e., usermap(i,j) in Fortran notation) holds the process rank to be placed in {i, j} coordinate of the 2D process grid. After grid creation, this subset of processes is logically ranked in the range 0 ... nprow*npcol − 1 in the new grid. For example, if we want to map 6 processes with ranks 11 ... 16 into a $2 \times 3$ grid, we define usermap = $\{11, 14, 12, 15, 13, 16\}$ and ldumap = 2. Such a mapping is shown below

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 11 | 12 | 13 |
| 1 | 14 | 15 | 16 |

In the actual implementation, superlu_gridinit() simply calls superlu_gridmap() with usermap[] holding the first nprow × npcol process ranks.

3.3.3 *Driver Routines.* There are two driver routines, one is called pdgssvx_ABglobal() for the global input interface, and another is called pdgssvx() for the distributed input interface. Their calling sequences are as follows.

```
pdgssvx_ABglobal(superlu_options_t *options, SuperMatrix *A,
                 ScalePermstruct_t *ScalePermstruct,
                 double B[], int ldb, int nrhs, gridinfo_t *grid,
                 LUstruct_t *LUstruct, double *berr,
                 SuperLUStat_t *stat, int *info);
```

```
pdgssvx(superlu_options_t *options, SuperMatrix *A,
        ScalePermstruct_t *ScalePermstruct,
        double B[], int ldb, int nrhs, gridinfo_t *grid,
        LUstruct_t *LUstruct, SOLVEstruct_t *SOLVEstruct, double *berr,
        SuperLUStat_t *stat, int *info);
```

Five basic steps are required to use the above routines:

(1) Initialize the MPI environment and the `SuperLU_DIST` process grid.
This is achieved by the calls to the MPI routine `MPI_Init()` and the `SuperLU_DIST` routine `superlu_gridinit()` or `superlu_gridmap()`. The `grid` structure is then input to the driver routine and all the underlying computational routines.

(2) Set up the input matrix and the right-hand side.
In most applications, the matrices can be generated on each process without the need to have a centralized place to hold them. In this case, using `pdgssvx()` is more convenient.

(3) Initialize the input arguments: `options, ScalePermstruct, LUstruct, stat`.
The subroutine `set_default_options_dist()` sets the default values for `options` argument. The user can modify any of its field afterwards. `ScalePermstruct` is the data structure that stores the several vectors describing the transformations done to $A$, including permutations and equilibrations. The routine `ScalePermstructInit()` initializes this structure. `LUstruct` is the data structure in which the distributed $L$ and $U$ factors are stored, and can be initialized by the routine `LUstructInit()`. `Stat` is a structure collecting the statistics about runtime and flop count, etc., and can be initialized by the routine `PStatInit()`.

(4) Call the `SuperLU_DIST` routine `pdgssvx_ABglobal()` or `pdgssvx()`.

(5) Release the process grid and terminate the MPI environment.
After the computation on a process grid has been completed, the process grid should be released by calling `superlu_gridexit()`. When all computations have been completed, the MPI routine `MPI_Finalize()` should be called.

3.3.4 *Distributed Sparse Matrix-vector Multiplication Routine.* Sparse matrix-vector multiplication is needed in the iterative refinement routine to compute the residual $r = b - Ax$. It is also of great interest by itself, because it is a key kernel in most iterative solvers. It is worth mentioning the routine designed for the distributed input interface, where $A$ is distributed by block rows. Consider $y \leftarrow Ax$. For each $i$, we need to compute $y_i = \sum_{j=1}^{n} a_{i,j}x_j$, where $a_{i,j}$ are on the same processor for all $j$. But some $x_j$ may be on some other processor, so there is a need to communicate the $x$ components. The algorithm consists of an initialization phase and an actual multiplication phase. In the initialization phase, each processor processes the local graph of $A$ (i.e., all $a_{i,j}$), and determines all the $j$'s such that $x_j$ is nonlocal. It then informs those processors who own $x_j$ so that they know they need to send $x_j$ to this processor. This phase involves an all-to-all communication so in the end every processor knowns which of my local $x_j$ needs to be sent to which other processors. Some optimization is performed to reduce communication. For example, if a processor needs to send several $x_j$'s to one other processor, these $x_j$'s are packed into one message, so that each processor sends no

more than one message to any other processor. Note that the initialization phase is time-consuming, so we run it only once and save the communication pattern. In the actual multiplication phase, each processor sends the corresponding local parts of $x$ to the processors that need them. Each processor also receives all those nonlocal parts of $x$, and together with the local part of $x$, it then performs the multiplication.

The initialization routine is called `pdgsmv_init()` and the multiplication routine is called `pdgsmv()`. These routines are quite independent from the rest of the library, and can be easily used outside `SuperLU_DIST`. From our performance study for large matrices on the IBM SP at NERSC, the initialization phase is usually only 3- to 4-fold slower than the multiplication phase [Li and Wang 2003]. So this pair of routines can be very useful for many iterative solvers.

## 4. ILLUSTRATION OF USE IN LARGE APPLICATIONS

In this section, we describe two applications in which `SuperLU_DIST` has played a critical role. The timings were obtained on the IBM SP at NERSC. A compute node of the system contains sixteen 375-MHz Power3 processors.

### 4.1 Solving a Three-Body Problem in Quantum Mechanics

The first application is in the solution of a long-standing problem of scattering in a quantum system of three charged particles. The particles' wave functions are represented by the time-independent Schrödinger equation. A team of scientists discovered a new exterior complex scaling formulism to represent the scattering states, which significantly simplified the boundary conditions, and made the problem computationally tractable [Baertschy et al. 1999]. Their finite difference scheme led to complex, nonsymmetric linear systems. The matrix is sparse but has a block structure, as shown in Figure 2. Each diagonal block $A_{ii}$ has the structure of a 2D finite difference Laplacian matrix, which is very sparse: the number of nonzeros per row is no more than 13 in these diagonal blocks. The block size is usually between 200,000 and 350,000. For some model problems, the diagonal block can go up to $2 \times 10^6$. Each off-diagonal block $d_{ij}$ is a diagonal matrix. The total dimension of the whole system can be as large as 8.4 million.

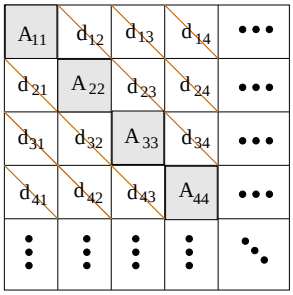


Fig. 2. The block-matrix structure of the quantum mechanics application.

These systems are very ill-conditioned. Even for the small model problems, none of the iterative algorithms with simple preconditioners converge. On the other

Table IV. **SuperLU_DIST** timings (seconds) for the quantum mechanics matrix of order $2 \times 10^6$.

| | P = 2x2 | P = 2x4 | P = 4x4 | P = 4x8 | P = 8x8 | P = 8x16 |
|---|---|---|---|---|---|---|
| LU | 5813.4 | 3160.6 | 1748.3 | 1070.5 | 742.2 | 560.6 |
| Solve | 66.5 | 85.4 | 53.9 | 53.9 | 36.1 | 27.5 |

hand, it is not feasible to use a direct solver for the whole system, because the long-range connectivity in the structure would cause tremendous fill using any ordering algorithm. What we finally used is a combination of iterative and direct algorithms. `SuperLU_DIST` is used in building the block diagonal preconditioners for the CGS iterative solver. That is, we solve a transformed linear system $M^{-1}Ax = M^{-1}b$, where $M = diag(A_{11}, A_{22}, A_{33}, \ldots)$. The overall program is organized as follows. We divide the total number of processors into processor subgroups of equal size. Each subgroup is assigned to a diagonal block $A_{ii}$. The parallel factorization of each $A_{ii} = L_{ii}U_{ii}$ is performed within each processor subgroup, and all the diagonal block factorizations are performed simultaneously by all the subgroups. The global matrix $A$ and the vectors are distributed by block rows among all the processors. During each iteration of the CGS algorithm, the sparse matrix-vector multiply $Ax$ and a few dense vector operatations are performed by all the processors. Applying the preconditioner $M^{-1}v$ at each iteration involves parallel triangular solutions using the $L_{ii}$ and $U_{ii}$ blocks, which are performed independently by all the processor subgroups. This example shows the usefulness of being able to arbitarily group processes into a `SuperLU_DIST` grid via `superlu_gridmap()`, see Section 3.3.2.

For this application, there is no need to perform numerical pivoting or iterative refinement. Therefore, we set `options.RowPerm = NO` (i.e., $P_r = I$) and `options.IterRefine = NO`. The default values are used for the rest of the parameters, see Section 3.1.2.

The time spent in preconditioning, including a one-time factorization and the triangular solutions in each CGS iteration, usually accounts for 80-90% of the total runtime. So here, we only present the scaling result of `SuperLU_DIST` for one diagonal block. Table IV shows the times of factorization and triangular solution for a diagonal block of order $2 \times 10^6$ using different numbers of processors. The factorization achieved 10-fold speedup from 4 to 128 processors, and 30 Gflops factorization rate. The solve time is usually under 5% of the factorization time, but its scaling needs to be improved.

In the typical production runs, the number of CGS iterations ranges between 12 to 35 depending on models. Since each CGS iteration requires two preconditioning steps, 24 to 70 solutions of the diagonal blocks are required. The total execution time ranges between half an hour to a few hours depending on the problem size and the processor configuration. See [Baertschy et al. 2001; Baertschy and Li 2001] for more details on the computations. This calculation was unprecedented, and the scientific breakthrough result was reported in a cover article of *Science* [Rescigno et al. 1999].

### 4.2 Modeling the Next Generation of Particle Accelerators

The second application is in the solution of Maxwell's equations in the electromagnetic field. This arises from the accelerator design where the cavity mode frequencies and the field vector are sought. The researchers at the Stanford Linear

Accelerator Center developed the widely used Omega3P simulation code for this purpose. The finite element methods lead to a large sparse generalized eigensystem $Kx = \lambda Mx$. Both matrices have the same nonzero structure and are very sparse– typically have only tens of nonzeros per row on average. The problem is challenging because of the wide distribution of the spectrum, and the need for finding the relatively small interior eigenvalues. Typically, there are a number of eigenvalues very close to zero. We want to find the first few (tens) smallest "nonzero" eigenvalues and their associated eigenvectors.

An effective method to find these small interior eigenvalues is by spectral transformation, which yields the new eigensystem: $M(K - \sigma M)^{-1}Mx = \mu Mx$, with $\mu = \frac{1}{\lambda - \sigma}$. The largest eigenvalues of the new system correspond to the original eigenvalues closest to the shift $\sigma$, and they are well separated from the others. When a Lanczos-type algorithm is employed for the new system, at each iteration, a shifted linear system involving operator $(K - \sigma M)^{-1}$ must be solved. The original algorithm used in Omega3P is called the Filtering algorithm [Sun 2003], which is a hybrid scheme combining two techniques: Inexact Shift and Invert Lanczos (ISIL) for obtaining eigenvector approximations efficiently, and Jacobi Orthogonal Component Correction (JOCC, a Newton type refinement) for refining the eigenvector approximations. The advantage of this algorithm is that the inner linear system $(K - \sigma M)x = b$ in both ISIL and JOCC can be solved inexactly using an iterative solver. The bulk of the computation is in sparse matrix-vector multiply, which is relatively easy to parallelize. The drawbacks are that it may take many Lanczos iterations for ISIL to settle with an acceptable approximation, and using one shift it can only find a few nearest eigenvalues.

We implemented an efficient Exact Shift and Invert Lanczos (ESIL) solver by coupling `SuperLU_DIST` with `PARPACK` [Lehoucq et al. ]), in which `SuperLU_DIST` is used for direct solution of the inner linear system $(K - \sigma M)x = b$. The advantages of the ESIL method are that the Lanczos iteration converges very fast, it is known to be very reliable and can find quite a few eigenvalues with one shift, corresponding to one factorization of $K - \sigma M$. The drawbacks are that it is more memory demanding due to the fill-ins in the factors, and each Lanczos iteration requires a sparse triangular solve which is slower and less scalable than sparse matrix-vector multiply.

For the ESIL solver, there is no need to perform numerical pivoting or iterative refinement. Therefore, we set `options.RowPerm = NO` (i.e., $P_r = I$) and `options.IterRefine = NO`. The default values are used for the rest of the parameters, see Section 3.1.2.

Our experience shows that when there is enough memory, our ESIL solver is usually faster than the Filtering algorithm [Husbands et al. 2003]. Figure 3 compares the parallel runtimes of the two solvers. For ESIL, we used two different reordering methods before performing factorization: one is `MeTiS` [Karypis and Kumar 1998], another is minimum degree [Liu 1985]. Linear elements are used to discretize a 47-cell DDS cavity structure, which results in the matrices $K$ and $M$ with dimension $1.3 \times 10^6$ and $20.1 \times 10^6$ nonzeros in each matrix. We need to find 16 eigenvalues/vectors near a shift specified by the user. It is clear that although each iteration of ESIL is more expensive than Filtering, many fewer iterations are needed

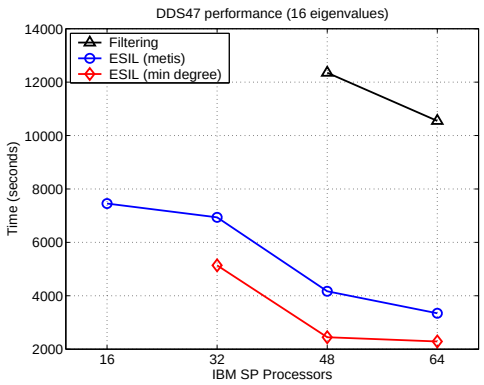


Fig. 3. Parallel runtime (seconds) of the Filtering and the ESIL solvers.

by ESIL (about 9 iterations per eigenvalue), and its total time is much less than that of Filtering. Even for the same ESIL algorithm, different ordering methods result in different sizes of the factors and hence different amounts of work and space required. In this example, minimum degree ordering is better than `MeTiS` ordering. The largest system solved so far with ESIL is of order $7.5 \times 10^6$ with $304 \times 10^6$ nonzeros in each matrix. Using one shift, we are able to find 10 eigenvalues close to that shift. `PARPACK` needs about 5.5 solves for each eigenvalue. Using 24 processors, the factorization takes 3347 seconds, one triangular solve takes 61 seconds, and the total eigensolver time is about 2.5 hours.

## 5. CONCLUSIONS AND FUTURE WORK

We have reviewed the algorithms and the implementation techniques in SuperLU. In describing the user interface, we illustrated how the solver's functionalities can be easily deployed and expanded. The Users' Guide [Demmel et al. 1999b] should serve as a complete documentation on all the user-callable routines. Looking at the real applications that used `SuperLU_DIST` to solve large-scale linear systems and eigenvalue problems, we demonstrated the solver's usability and capability of handling very large systems.

Future work is planned in the following areas: (1) We will improve the performance of the parallel triangular solution routine. This becomes important because many applications use SuperLU in a preconditioning context, such as the two mentioned in Section 4. In those cases, for one factorization, there needs many triangular solutions. The techniques we plan to investigate include switching to a dense representation for the last Schur complement block when it becomes sufficiently full, and *partitioned inverse* [Alvarado et al. 1993] that transforms the substitution into a sequence of sparse matrix-vector multiplications. (2) We will parallelize the symbolic factorization routine, which will enhance the memory scalability of `SuperLU_DIST`. We plan to apply `ParMeTiS` [Karypis et al. 2003] to the graph of $A^T + A$ to obtain both a sparsity-preserving ordering and a separator tree. We will then use a subtree to sub-processor mapping for the initial distribution of $G(A)$. The symbolic factorization algorithm will work from bottom to the top of the separator

tree, and obtain one column structure of $L$ and one row structure of $U$ at each step. (3) We will improve numerical robustness for `SuperLU_DIST`. We can enhance the current iterative refinement routine by using extra precision to compute the residuals [Li et al. 2002]. In addition to the standard iterative refinement, we can add other iterative solvers such as GMRES or QMR as an option to the refinement method.

## ACKNOWLEDGMENTS

We thank the referees for their careful reading of the manuscript and providing very constructive comments that significantly improved the presentation quality.

## 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, 159–190.

Amestoy, P. R., Davis, T. A., and Duff, I. S. 1996. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Analysis and Applications 17,* 4, 886–905. Also University of Florida TR-94-039.

Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y., and Koster, J. 2003. MUltifrontal Massively Parallel Solver (MUMPS version 4.3) Users' Guide.

Amestoy, P. R., Li, X. S., and Ng, E. G. 2003. Diagonal markowitz scheme with local symmetrization. Tech. Rep. LBNL-53854, Lawrence Berkeley National Laboratory. October. Also ENSEEIHT-IRIT RT/APO/03/05.

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. 1999. *LAPACK Users' Guide, Release 3.0.* SIAM, Philadelphia. 407 pages.

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, Texas. http://www.netlib.org/linalg/spooles.

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, Colorado.

Baertschy, M., Rescigno, T., Isaacs, W., Li, X., and McCurdy, C. 2001. Electron-impact ionization of atomic hydrogen. *Physical Review A 63 022712.*

Baertschy, M., Rescigno, T. N., and McCurdy, C. W. 1999. Accurate numerical solution to a coulomb 3-body problem. *Phys. Rev. Letters.* Submitted.

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. submitted to *ACM Trans. Math. Software.*

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

Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S., and Liu, J. W. H. 1999. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications 20,* 3, 720–755.

Demmel, J. W., Gilbert, J. R., and Li, X. S. 1999a. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications 20,* 4, 915–952.

Demmel, J. W., Gilbert, J. R., and Li, X. S. 1999b. SuperLU Users' Guide. Tech. Rep. LBNL-44289, Lawrence Berkeley National Laboratory. September. Software is available at http://crd.lbl.gov/~xiaoye/SuperLU.

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 Analysis and Applications 20,* 4, 889–901.

EISENSTAT, S. C. AND LIU, J. W. 1992. Exploiting structural symmetry in sparse unsymmetric symbolic factorization. *SIAM J. Matrix Anal. Appl.*, 13:202–211.

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

GILBERT, J. R. AND PEIERLS, T. 1988. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Scientific and Statistical Computing 9*, 862–874.

GRIGORI, L. AND LI, X. S. 2002. A new scheduling algorithm for parallel sparse LU factorization with static pivoting. In *Proceedings of SC2002.* Baltimore.

HENDRICKSON, B. AND LELAND, R. 1993. The CHACO User's Guide. Version 1.0. Tech. Rep. SAND93-2339 • UC-405, Sandia National Laboratories, Albuquerque. http://www.cs.sandia.gov/∼bahendr/chaco.html.

HIGHAM, N. J. 1996. *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadelphia, PA.

HUSBANDS, P., YANG, C., LI, X., NG, E., SUN, Y., KO, K., AND GOLUB, G. 2003. When computing interior eigenvalues, just how far can a direct solver take you? SIAM Annual Meeting, June 16-20, Montreal, Canada.

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. http://www-users.cs.umn.edu/∼karypis/metis.

KARYPIS, G., SCHLOEGEL, K., AND KUMAR, V. 2003. ParMeTiS*: Parallel Graph Partitioning and Sparse Matrix Ordering Library – Version 3.1.* University of Minnesota. http://www-users.cs.umn.edu/∼karypis/metis/parmetis/.

LEHOUCQ, R., MASCHHOFF, K., SORENSEN, D., AND YANG, C. Parallel ARPACK. 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, Florida.

LI, X. S. AND DEMMEL, J. W. 2003. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software 29,* 2 (June), 110–140.

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. Mathematical Software 28,* 2, 152–205.

LI, X. S. AND WANG, Y. 2003. Performance evaluation and enhancement of SuperLU_DIST 2.0. Tech. Rep. LBNL-53624, Lawrence Berkeley National Laboratory. August.

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

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

MPI. Message Passing Interface (MPI) forum. http://www.mpi-forum.org/.

OETTLI, W. AND PRAGER, W. 1964. Compatibility of approximate solution of linear equations with given error bounds for coefficients and right hand sides. *Num. Math. 6*, 405–409.

PELLEGRINI, F. 2001. SCOTCH 3.4 User's Guide. Tech report 1264-01, LaBRI, URM CNRS 5800, University Bordeaux I, France. November. http://www.labri.fr/∼pelegrin/scotch.

PRALET, S., AMESTOY, P. R., AND LI, X. S. 2004. Unsymmetric ordering using a constraint Markowitz scheme. Tech. rep., In preparation.

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 (December 24,), 2474–2479.

SCHENK, O. AND GÄRTNER, K. 2004. Parallel sparse direct linear solver PARDISO, User Guide Version 1.2. http://www.computational.unibas.ch/cs/scicomp/software/pardiso/.

SUN, Y. 2003. The filter algorithm for solving large-scale eigenproblems from accelerator simulations. Ph.D. thesis, Department of Computer Science, Stanford University.