SuperLU Users' Guide

James W. Demmel[1]   John R. Gilbert[2]   Xiaoye S. Li[3]

September, 1999
Last update: October, 2003

[1]Computer Science Division, University of California, Berkeley, CA 94720. (demmel@cs.berkeley.edu). The research of Demmel and Li was supported in part by NSF grant ASC–9313958, DOE grant DE–FG03–94ER25219, UT Subcontract No. ORA4466 from ARPA Contract No. DAAL03–91–C0047, DOE grant DE–FG03–94ER25206, and NSF Infrastructure grants CDA–8722788 and CDA–9401156.

[2]Department of Computer Science, University of California, Santa Barbara, CA 93106. (gilbert@cs.ucsb.edu). The research of this author was supported in part by the Institute for Mathematics and Its Applications at the University of Minnesota and in part by DARPA Contract No. DABT63-95-C0087.

[3]Lawrence Berkeley National Lab, MS 50F1650, 1 Cyclotron Rd, Berkeley, CA 94720. (xiaoye@nersc.gov). 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.

Contents

1 Introduction **3**
- 1.1 Purpose of SuperLU . . . 3
- 1.2 Overall Algorithm . . . 4
- 1.3 What the three libraries have in common . . . 5
  - 1.3.1 Input and Output Data Formats . . . 5
  - 1.3.2 Tuning Parameters for BLAS . . . 6
  - 1.3.3 Performance Statistics . . . 6
  - 1.3.4 Error Handling . . . 6
  - 1.3.5 Ordering the Columns of A for Sparse Factors . . . 7
  - 1.3.6 Iterative Refinement . . . 7
  - 1.3.7 Error Bounds . . . 8
  - 1.3.8 Solving a Sequence of Related Linear Systems . . . 9
  - 1.3.9 Interfacing to other languages . . . 9
- 1.4 How the three libraries differ . . . 9
  - 1.4.1 Input and Output Data Formats . . . 9
  - 1.4.2 Parallelism . . . 10
  - 1.4.3 Pivoting Strategies for Stability . . . 10
  - 1.4.4 Memory Management . . . 11
  - 1.4.5 Interfacing to other languages . . . 11
- 1.5 Performance . . . 11
- 1.6 Software Status and Availability . . . 11
- 1.7 Acknowledgement . . . 13

2 Sequential SuperLU (Version 3.0) **14**
- 2.1 About `SuperLU` . . . 14
- 2.2 How to call a `SuperLU` routine . . . 14
- 2.3 Matrix data structures . . . 17
- 2.4 `Options` argument . . . 20
- 2.5 Permutations . . . 22
  - 2.5.1 Ordering for sparsity . . . 22
  - 2.5.2 Partial pivoting with threshold . . . 23
- 2.6 Symmetric Mode . . . 23
- 2.7 Memory management for L and U . . . 23
- 2.8 User-callable routines . . . 24
  - 2.8.1 Driver routines . . . 24
  - 2.8.2 Computational routines . . . 25
  - 2.8.3 Utility routines . . . 25

2.9 Matlab interface . . . 27
2.10 Installation . . . 28
2.10.1 File structure . . . 28
2.10.2 Testing . . . 30
2.10.3 Performance-tuning parameters . . . 31
2.11 Example programs . . . 32
2.12 Calling from Fortran . . . 32

3 Multithreaded SuperLU (Version 1.1) 38
3.1 About SuperLU_MT . . . 38
3.2 Storage types for L and U . . . 38
3.3 User-callable routines . . . 39
3.3.1 Driver routines . . . 40
3.3.2 Computational routines . . . 40
3.4 Installation . . . 41
3.4.1 File structure . . . 41
3.4.2 Performance issues . . . 41
3.5 Example programs . . . 44
3.6 Porting to other platforms . . . 44
3.6.1 Creating multiple threads . . . 44
3.6.2 Use of mutexes . . . 45

4 Distributed SuperLU with MPI (Version 2.0) 46
4.1 About SuperLU_DIST . . . 46
4.2 Formats of the input matrices A and B . . . 46
4.2.1 Global input . . . 46
4.2.2 Distributed input . . . 46
4.3 Distributed data structures for L and U . . . 47
4.4 Process grid and MPI communicator . . . 48
4.4.1 2D process grid . . . 48
4.4.2 Arbitrary grouping of processes . . . 49
4.5 Basic steps to solve a linear system . . . 50
4.6 Algorithmic background . . . 54
4.7 User-callable routines . . . 55
4.7.1 Driver routine . . . 55
4.7.2 Computational routines . . . 57
4.7.3 Utility routines . . . 57
4.8 Installation . . . 58
4.8.1 File structure . . . 58
4.8.2 Performance-tuning parameters . . . 59
4.9 Example programs . . . 59
4.10 Fortran 90 Interface . . . 60
4.10.1 Callable functions in the Fortran 90 module file spuerlu_mod.f90 . . . 64
4.10.2 C wrapper functions callable by Fortran in file spuerlu_c2f_wrap.c . . . 65

Chapter 1

Introduction

1.1 Purpose of SuperLU

This document describes a collection of three related ANSI C subroutine libraries for solving sparse linear systems of equations $AX = B$. 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. Matrix A need not be symmetric or definite; indeed, SuperLU is particularly appropriate for matrices with very unsymmetric structure. All three libraries use variations of Gaussian elimination optimized to take advantage both of sparsity and the computer architecture, in particular memory hierarchies (caches) and parallelism.

In this introduction we refer to all three libraries collectively as SuperLU. The three libraries within SuperLU are as follows. Detailed references are also given (see also [21]).

- **Sequential SuperLU** is designed for sequential processors with one or more layers of memory hierarchy (caches) [5].
- **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 [6].
- **Distributed SuperLU (SuperLU_DIST)** is designed for distributed memory parallel processors, using MPI [26] for interprocess communication. It can effectively use hundreds of parallel processors on sufficiently large matrices [23, 24].

Table 1.1 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 interface for all three libraries.

	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

Table 1.1: SuperLU software status.

The rest of the Introduction is organized as follows. Section 1.2 describes the high-level algorithm used by all three libraries, pointing out some common features and differences. Section 1.3 describes the detailed algorithms, data structures, and interface issues common to all three routines. Section 1.4 describes how the three routines differ, emphasizing the differences that most affect the user. Section 1.6 describes the software status, including planned developments, bug reporting, and licensing.

1.2 Overall Algorithm

A simple description of sparse Gaussian elimination is 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_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.

The simplest implementation, used by the "simple driver routines" within SuperLU and SuperLU_MT, is as follows:

Simple Driver Algorithm

1. *Choose P_c to order the columns of A* to increase the sparsity of the computed L and U factors, and hopefully increase parallelism (for SuperLU_MT).

2. *Compute the LU factorization of AP_c.* SuperLU and SuperLU_MT can perform dynamic pivoting of the rows during factorization for numerical stability, computing P_r, L and U at the same time.

3. *Solve the system* using P_r, P_c, L and U as described above. ($D_r = D_c = I$)

The simple driver subroutines for double precision real data are called `dgssv` and `pdgssv` for SuperLU and SuperLU_MT, respectively. The letter `d` in the subroutine names means double precision real; other options are `s` for single precision real, `c` for single precision complex, and `z` for double precision complex. The subroutine naming scheme is analogous to the one used in LAPACK [1].

SuperLU_DIST does not include this simple driver.

There is also an "expert driver subroutine" that can provide more accurate solutions, compute error bounds, and solve a sequence of related linear systems more economically. It is available in all three libraries.

Expert Driver Algorithm

1. *Equilibrate* the matrix A, i.e. compute diagonal matrices D_r and D_c so that $\hat{A} = D_r A D_c$ is "better conditioned" than A, i.e. $\hat{A}^{-1}$ is less sensitive to perturbations in $\hat{A}$ than A^{-1} is to perturbations in A.

2. *Preorder the rows of $\hat{A}$ (SuperLU_DIST only)*, i.e. replace $\hat{A}$ by $P_r\hat{A}$ where P_r is a permutation matrix. We call this step "static pivoting", and it is only done in the distributed memory algorithm.

3. *Order the columns of $\hat{A}$* to increase the sparsity of the computed L and U factors, and hopefully increase parallelism (for SuperLU_MT and SuperLU_DIST). In other words, replace $\hat{A}$ by $\hat{A}P_c^T$ in SuperLU and SuperLU_MT, or replace $\hat{A}$ by $P_c\hat{A}P_c^T$ in SuperLU_DIST, where P_c is a permutation matrix.

4. *Compute the LU factorization of $\hat{A}$.* SuperLU and SuperLU_MT can perform dynamic pivoting of the rows during factorization for numerical stability. In contrast, SuperLU_DIST uses the order computed by the preordering step but replaces tiny pivots by larger numbers for stability.

5. *Solve the system* using the computed triangular factors.

6. *Iteratively refine the solution*, again using the computed triangular factors. This is equivalent to Newton's method.

7. *Compute error bounds.* Both forward and backward error bounds are computed, as described below.

The expert driver subroutines for double precision real data are called `dgssvx`, `pdgssvx` and `pdgssvx` for SuperLU, SuperLU_MT and SuperLU_DIST, respectively. Sequential SuperLU also provides single precision real (`s`), single precision complex (`c`), and double precision complex (`z`) versions. SuperLU_MT only provides double precision real (`d`). SuperLU_DIST provides both double precision real (`d`) and complex (`z`).

The driver routines are composed of several lower level computational routines for computing permutations, computing LU factorization, solving triangular systems, and so on. The LU factorization routine for all three libraries also handles nonsquare matrices. For large matrices, the LU factorization steps takes most of the time, although choosing P_c to order the columns can also be time-consuming.

1.3 What the three libraries have in common

1.3.1 Input and Output Data Formats

All three libraries accept A and B as double precision real. (Sequential SuperLU additionally accepts single precision real and both single and double precision complex. SuperLU_DIST also accepts double precision complex.)

A is stored in a sparse data structure according to the struct `SuperMatrix`, which is described in section 3.2. In particular, A may be supplied in either column-compressed format ("Harwell-Boeing format"), or row-compressed format (i.e. A^T stored in column-compressed format). B, which is overwritten by the solution X, is stored as a dense matrix in column-major order. In SuperLU_DIST, A and B can be either replicated or distributed across all processes.

(The storage of L and U differs among the three libraries, as discussed in section 1.4.)

1.3.2 Tuning Parameters for BLAS

All three libraries depend on having high performance BLAS (Basic Linear Algebra Subroutine) libraries [20, 8, 7] in order to get high performance. In particular, they depend on matrix-vector multiplication or matrix-matrix multiplication of relatively small dense matrices. The sizes of these small dense matrices can be tuned to match the "sweet spot" of the BLAS by setting certain tuning parameters described in section 2.10.3 for SuperLU, in section 3.4.2 for SuperLU_MT, and in section 4.8.2 for SuperLU_DIST.

(In addition, SuperLU_MT and SuperLU_DIST let one control the number of parallel processes to be used, as described in section 1.4.)

1.3.3 Performance Statistics

Most of the computational routines use a struct to record certain kinds of performance data, namely the time and number of floating point operations in each phase of the computation, and data about the sizes of the matrices L and U. These statistics are collected during the computation. A statistic variable is declared with the following type:

```
typedef struct {
    int     *panel_histo; /* histogram of panel size distribution */
    double  *utime;       /* time spent in various phases */
    float   *ops;         /* floating-point operations at various phases */
    int     TinyPivots;   /* number of tiny pivots */
    int     RefineSteps;  /* number of iterative refinement steps */
} SuperLUStat_t;
```

For both SuperLU and SuperLU_MT, there is only one copy of these statistics variable. But for SuperLU_DIST, each process keeps a local copy of this variable, and records its local statistics. We need to use MPI reduction routines to find any global information, such as the sum of the floating-point operation count on all processes.

Before the computation, routine `StatInit()` should be called to malloc storage and perform initialization for the fields `panel_histo`, `utime`, and `ops`. The algorithmic phases are defined by the enumeration type `PhaseType` in `SRC/util.h`. In the end, routine `StatFree()` should be called to free storage of the above statistics fields. After deallocation, the statistics are no longer accessible. Therefore, users should extract the information they need before calling `StatFree()`, which can be accomplished by calling `(P)StatPrint()`.

An inquiry function `dQuerySpace()` is provided to compute memory usage statistics. This routine should be called after the LU factorization. It calculates the storage requirement based on the size of the L and U data structures and working arrays.

1.3.4 Error Handling

Invalid arguments and (P)XERBLA

Similar to LAPACK, for all the SuperLU routines, we check the validity of the input arguments to each routine. If an illegal value is supplied to one of the input arguments, the error handler XERBLA is called, and a message is written to the standard output, indicating which argument has an illegal value. The program returns immediately from the routine, with a negative value of INFO.

Computational failures with INFO > 0

A positive value of INFO on return from a routine indicates a failure in the course of the computation, such as a matrix being singular, or the amount of memory (in bytes) already allocated when malloc fails.

ABORT on unrecoverable errors

A macro `ABORT` is defined in `SRC/util.h` to handle unrecoverable errors that occur in the middle of the computation, such as `malloc` failure. The default action of `ABORT` is to call

```
superlu_abort_and_exit(char *msg)
```

which prints an error message, the line number and the file name at which the error occurs, and calls the `exit` function to terminate the program.

If this type of termination is not appropriate in some environment, users can alter the behavior of the abort function. When compiling the SuperLU library, users may choose the C preprocessor definition

```
-DUSER_ABORT = my_abort
```

At the same time, users would supply the following `my_abort` function

```
my_abort(char *msg)
```

which overrides the behavior of `superlu_abort_and_exit`.

1.3.5 Ordering the Columns of A for Sparse Factors

There is a choice of orderings for the columns of A both in the simple or expert driver, in section 1.2:

- Natural ordering,
- Multiple Minimum Degree (MMD) [25] applied to the structure of $A^T A$,
- Multiple Minimum Degree (MMD) [25] applied to the structure of $A^T + A$,
- Column Approximate Minimum Degree (COLAMD) [4], and
- Use a P_c supplied by the user as input.

COLAMD is designed particularly for unsymmetric matrices when partial pivoting is needed, and does not require explicit formation of $A^T A$. It usually gives comparable orderings as MMD on $A^T A$, and is faster.

The orderings based on graph partitioning heuristics are also popular, as exemplified in the METIS package [19]. The user can simply input this ordering in the permutation vector for P_c. Note that many graph partitioning algorithms are designed for symmetric matrices. The user may still apply them to the structures of $A^T A$ or $A^T + A$. Our routines `getata()` and `at_plus_a` in the file `get_perm_c.c` can be used to form $A^T A$ or $A^T + A$.

1.3.6 Iterative Refinement

Step 6 of the expert driver algorithm, iterative refinement, serves to increase accuracy of the computed solution. Given the initial approximate solution x from step 5, the algorithm for step 6 is as follows (where x and b are single columns of X and B, respectively):

Compute residual $r = Ax - b$
While residual too large
    Solve $Ad = r$ for correction d
    Update solution $x = x - d$
    Update residual $r = Ax - b$
end while

If r and then d were computed exactly, the updated solution $x - d$ would be the exact solution. Roundoff prevents immediate convergence.

The criterion "residual too large" in the iterative refinement algorithm above is essentially that

$$BERR \equiv \max_i |r_i|/s_i \tag{1.1}$$

exceeds the machine roundoff level, or is continuing to decrease quickly enough. Here s_i is the scale factor

$$s_i = (|A| \cdot |x| + |b|)_i = \sum_j |A_{ij}| \cdot |x_j| + |b_i|$$

In this expression $|A|$ is the n-by-n matrix with entries $|A|_{ij} = |A_{ij}|$, $|b|$ and $|x|$ are similarly column vectors of absolute entries of b and x, respectively, and $|A| \cdot |x|$ is conventional matrix-vector multiplication.

The purpose of this stopping criterion is explained in the next section.

1.3.7 Error Bounds

Step 7 of the expert driver algorithm computes error bounds.

It is shown in [2, 27] that $BERR$ defined in Equation 1.1 measures the *componentwise relative backward error* of the computed solution. This means that the computed x satisfies a slightly perturbed linear system of equations $(A + E)x = b + f$, where $|E_{ij}| \leq BERR \cdot |A_{ij}|$ and $|f_i| \leq BERR \cdot |b_i|$ for all i and j. It is shown in [2, 30] that one step of iterative refinement usually reduces $BERR$ to near machine epsilon. For example, if $BERR$ is 4 times machine epsilon, then the computed solution x is identical to the solution one would get by changing each nonzero entry of A and b by at most 4 units in their last places, and then solving this perturbed system *exactly*. If the nonzero entries of A and b are uncertain in their bottom 2 bits, then one should generally not expect a more accurate solution. Thus $BERR$ is a measure of backward error specifically suited to solving sparse linear systems of equations. Despite roundoff, $BERR$ itself is always computed to within about $\pm n$ times machine epsilon (and usually much more accurately) and so $BERR$ is quite accurate.

In addition to backward error, the expert driver computes a *forward error bound*

$$FERR \geq \|x_{\text{true}} - x\|_\infty / \|x\|_\infty$$

Here $\|x\|_\infty \equiv \max_i |x_i|$. Thus, if $FERR = 10^{-6}$ then each component of x has an error bounded by about 10^{-6} times the largest component of x. The algorithm used to compute $FERR$ is an approximation; see [2, 17] for a discussion. Generally $FERR$ is accurate to within a factor of 10 or better, which is adequate to say how many digits of the large entries of x are correct.

(SuperLU_DIST's algorithm for $FERR$ is slightly less reliable [24].)

1.3.8 Solving a Sequence of Related Linear Systems

It is very common to solve a sequence of related linear systems $A^{(1)}X^{(1)} = B^{(1)}$, $A^{(2)}X^{(2)} = B^{(2)}$, ... rather than just one. When $A^{(1)}$ and $A^{(2)}$ are similar enough in sparsity pattern and/or numerical entries, it is possible to save some of the work done when solving with $A^{(1)}$ to solve with $A^{(2)}$. This can result in significant savings. Here are the options, in increasing order of "reuse of prior information":

1. *Factor from scratch.* No previous information is used. If one were solving just one linear system, or a sequence of unrelated linear systems, this is the option to use.

2. *Reuse P_c, the column permutation.* The user may save the column permutation and reuse it. This is most useful when $A^{(2)}$ has the same sparsity structure as $A^{(1)}$, but not necessarily the same (or similar) numerical entries. Reusing P_c saves the sometimes quite expensive operation of computing it.

3. *Reuse P_c, P_r and data structures allocated for L and U.* If P_r and P_c do not change, then the work of building the data structures associated with L and U (including the elimination tree [14]) can be avoided. This is most useful when $A^{(2)}$ has the same sparsity structure and similar numerical entries as $A^{(1)}$. When the numerical entries are not similar, one can still use this option, but at a higher risk of numerical instability ($BERR$ will always report whether or not the solution was computed stably, so one cannot get an unstable answer without warning).

4. *Reuse P_c, P_r, L and U.* In other words, we reuse essentially everything. This is most commonly used when $A^{(2)} = A^{(1)}$, but $B^{(2)} \neq B^{(1)}$, i.e. when only the right-hand sides differ. It could also be used when $A^{(2)}$ and $A^{(1)}$ differed just slightly in numerical values, in the hopes that iterative refinement converges (using $A^{(2)}$ to compute residuals but the triangular factorization of $A^{(1)}$ to solve).

Because of the different ways L and U are computed and stored in the three libraries, these 4 options are specified slightly differently; see Chapters 2 through 4 for details.

1.3.9 Interfacing to other languages

It is possible to call all the drivers and the computational routines from Fortran. However, currently the Fortran wrapper functions are not complete. The users are expected to look at the Fortran example programs in the FORTRAN/ directory, together with the C "bridge" routine, and learn how to call SuperLU from a Fortran program. The users can modify the C bridge routine to fit their needs.

1.4 How the three libraries differ

1.4.1 Input and Output Data Formats

All Sequential SuperLU routines are available in single and double precision (real or complex), but SuperLU_MT routines are only available in double precision real, and SuperLU_DIST routines are available in double precision (real or complex).

L and U are stored in different formats in the three libraries:

- *L and U in Sequential SuperLU.* L is a "column-supernodal" matrix, in storage type `SCformat`. This means it is stored sparsely, with supernodes (consecutive columns with identical structures) stored as dense blocks. U is stored in column-compressed format `NCformat`. See section 2.3 for details.

- *L and U in SuperLU_MT.* Because of parallelism, the columns of L and U may not be computed in consecutive order, so they may be allocated and stored out of order. This means we use the "column-supernodal-permuted" format `SCPformat` for L and "column-permuted" format `NCPformat` for U. See section 3.2 for details.

- *L and U in SuperLU_DIST.* Now L and U are distributed across multiple processors. As described in detail in Sections 4.3 and 4.4, we use a 2D block-cyclic format, which has been used for dense matrices in libraries like ScaLAPACK [3]. But for sparse matrices, the blocks are no longer identical in size, and vary depending on the sparsity structure of L and U. The detailed storage format is discussed in section 4.3 and illustrated in Figure 4.1.

1.4.2 Parallelism

Sequential SuperLU has no explicit parallelism. Some parallelism may still be exploited on an SMP by using a multithreaded BLAS library if available. But it is likely to be more effective to use SuperLU_MT on an SMP, described next.

SuperLU_MT lets the user choose the number of parallel threads to use. The mechanism varies from platform to platform and is described in section 3.6.

SuperLU_DIST not only lets the user specify the number of processors, but how they are arranged into a 2D grid. Furthermore, MPI permits any subset of the processors allocated to the user may be used for SuperLU_DIST, not just consecutively numbered processors (say 0 through P-1). See section 4.4 for details.

1.4.3 Pivoting Strategies for Stability

Sequential SuperLU and SuperLU_MT use the same pivoting strategy, called *threshold pivoting*, to determine the row permutation P_r. Suppose we have factored the first $i-1$ columns of A, and are seeking the pivot for column i. Let a_{mi} be a largest entry in magnitude on or below the diagonal of the partially factored A: $|a_{mi}| = \max_{j \geq i} |a_{ji}|$. Depending on a threshold $0 < u \leq 1$ input by the user, the code will use the diagonal entry a_{ii} as the pivot in column i as long as $|a_{ii}| \geq u \cdot |a_{mi}|$, and otherwise use a_{mi}. So if the user sets $u = 1$, a_{mi} (or an equally large entry) will be selected as the pivot; this corresponds to the classical *partial pivoting strategy.* If the user has ordered the matrix so that choosing diagonal pivots is particularly good for sparsity or parallelism, then smaller values of u will tend to choose those diagonal pivots, at the risk of less numerical stability. Using $u = 0$ guarantees that the pivots on the diagonal will be chosen, unless they are zero. The error bound $BERR$ measure how much stability is actually lost.

Threshold pivoting turns out to be hard to parallelize on distributed memory machines, because of the fine-grain communication and dynamic data structures required. So SuperLU_DIST uses a new scheme called *static pivoting* instead. In static pivoting the pivot order (P_r) is chosen before numerical factorization, using a weighted perfect matching algorithm [9], and kept fixed during factorization. Since both row and column orders (P_r and P_c) are fixed before numerical factorization, we can extensively optimize the data layout, load balance, and communication schedule. The price is a higher risk of numeric instability, which is mitigated by diagonal scaling, setting very tiny pivots to larger values, and iterative refinement [24]. Again, error bound $BERR$ measure how much stability is actually lost.

1.4.4 Memory Management

Because of fill-in of entries during Gaussian elimination, L and U typically have many more nonzero entries than A. If P_r and P_c are not already known, we cannot determine the number and locations of these nonzeros before performing the numerical factorization. This means that some kind of dynamic memory allocation is needed.

Sequential SuperLU lets the user either supply a preallocated space `work[]` of length `lwork`, or depend on malloc/free. The variable `FILL` can be used to help the code predict the amount of fill, which can reduce both fragmentation and the number of calls to malloc/free. If the initial estimate of the size of L and U from `FILL` is too small, the routine allocates more space and copies the current L and U factors to the new space and frees the old space. If the routine cannot allocate enough space, it calls a user-specifiable routine ABORT. See sections 1.3.4 for details.

SuperLU_MT is similar, except that the current alpha version cannot reallocate more space for L and U if the initial size estimate from `FILL` is too small. Instead, the program calls ABORT and the user must start over with a larger value of `FILL`. See section 3.4.2.

SuperLU_DIST actually has a simpler memory management chore, because once P_r and P_c are determined, the structures of L and U can be determined efficiently and just the right amount of memory allocated using malloc and later free. So it will call ABORT only if there is really not enough memory available to solve the problem.

1.4.5 Interfacing to other languages

Sequential SuperLU has a Matlab interface to the driver via a MEX file. See section 2.9 for details.

1.5 Performance

SuperLU library incorporates a number of novel algorithmic ideas developed recently. These algorithms also exploit the features of modern computer architectures, in particular, the multi-level cache organization and parallelism. We have conducted extensive experiments on various platforms, with a large collection of test matrices. The Sequential SuperLU achieved up to 40% of the theoretical floating-point rate on a number of processors, see [5, 21]. The megaflop rate usually increases with increasing ratio of floating-point operations count over the number of nonzeros in the L and U factors. The parallel LU factorization in SuperLU_MT demonstrated 5–10 fold speedups on a range of commercially popular SMPs, and up to 2.5 Gigaflops factorization rate, see [6, 21]. The parallel LU factorization in SuperLU_DIST achieved up to 100 fold speedup on a 512-processor Cray T3E, and 10.2 Gigaflops factorization rate, see [23].

1.6 Software Status and Availability

All three libraries are freely available for all uses, commercial or noncommercial, subject to the following caveats. No warranty is expressed or implied by the authors, although we will gladly answer questions and try to fix all reported bugs. We ask that proper credit be given to the authors and that a notice be included if any modifications are made.

The following Copyright applies to the whole SuperLU software.

Some routines carry the additional notices as follows.

1. Some subroutines carry the following notice:

2. The MC64 routine (**only used in SuperLU_DIST**) carries the following notice:

The user can modify this code but, at no time shall the right or title to all or any part of this package pass to the user. The user shall make available free of charge to the authors for any purpose all information relating to any alteration or addition made to this package for the purposes of extending the capabilities or enhancing the performance of this package.

The user shall not pass this code directly to a third party without the express prior consent of the authors. Users wanting to licence their own copy of these routines should send email to hsl@aeat.co.uk

None of the comments from the Copyright notice up to and including this one shall be removed or altered in any way.

All three libraries can be obtained from the following URLs:

```
http://crd.lbl.gov/~xiaoye/SuperLU/
http://www.netlib.org/scalapack/prototype/
```

In the future, we will add more functionality in the software, such as sequential and parallel incomplete LU factorizations, as well as parallel symbolic and ordering algorithms for SuperLU_DIST; these latter routines would replace MC64 and have no restrictions on external use.

All bugs reports and queries can be e-mailed to `xsli@lbl.gov` and `demmel@cs.berkeley.edu`.

1.7 Acknowledgement

With great gratitude, we acknowledge Stan Eisenstat and Joesph Liu for their significant contributions to the development of Sequential SuperLU.

We would like to thank Jinqchong Teo for helping generate the code in Sequential SuperLU to work with four floating-point data types. We thank Tim Davis for his contribution of some subroutines related to column ordering and suggestions on improving the routines' interfaces. We thank Ed Rothberg of Silicon Graphics for discussions and providing us access to the SGI Power Challenge. We Thank Yu Wang and William F. Mitchell for developing the Fortran 90 interface for `SuperLU_DIST`.

We acknowledge the following organizations that provided the computer resources during our code development: NERSC at Lawrence Berkeley National Laboratory, Livermore Computing at Lawrence Livermore National Laboratory, NCSA at University of Illinois at Urbana-Champaign, Silicon Graphics, and Xerox Palo Alto Research Center. We thank UC Berkeley and NSF Infrastructure grant CDA-9401156 for providing Berkeley NOW.

Chapter 2

Sequential SuperLU (Version 3.0)

2.1 About `SuperLU`

In this chapter, SuperLU will always mean Sequential SuperLU. `SuperLU` package contains a set of subroutines to solve sparse linear systems $AX = B$. 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. Matrix A need not be symmetric or definite; indeed, `SuperLU` is particularly appropriate for matrices with very unsymmetric structure.

The package uses LU decomposition with partial (or threshold) pivoting, and forward/back substitutions. The columns of A may be preordered before factorization (either by the user or by `SuperLU`); this preordering for sparsity is completely separate from the factorization. To improve backward stability, we provide working precision iterative refinement subroutines [2]. Routines are also available to equilibrate the system, estimate the condition number, calculate the relative backward error, and estimate error bounds for the refined solutions. We also include a Matlab MEX-file interface, so that our factor and solve routines can be called as alternatives to those built into Matlab. The LU factorization routines can handle non-square matrices, but the triangular solves are performed only for square matrices.

The factorization algorithm uses a graph reduction technique to reduce graph traversal time in the symbolic analysis. We exploit dense submatrices in the numerical kernel, and organize computational loops in a way that reduces data movement between levels of the memory hierarchy. The resulting algorithm is highly efficient on modern architectures. The performance gains are particularly evident for large problems. There are "tuning parameters" to optimize the peak performance as a function of cache size. For a detailed description of the algorithm, see reference [5].

`SuperLU` is implemented in ANSI C, and must be compiled with a standard ANSI C compiler. It includes versions for both real and complex matrices, in both single and double precision. The file names for the single-precision real version start with letter "s" (such as `sgstrf.c`); the file names for the double-precision real version start with letter "d" (such as `dgstrf.c`); the file names for the single-precision complex version start with letter "c" (such as `cgstrf.c`); the file names for the double-precision complex version start with letter "z" (such as `zgstrf.c`).

2.2 How to call a `SuperLU` routine

As a simple example, let us consider how to solve a 5×5 sparse linear system $AX = B$, by calling a driver routine `dgssv()`. Figure 2.1 shows matrix A, and its L and U factors. This sample program

$$
\begin{pmatrix} s & & u & u & \\ l & u & & & \\ & l & p & & \\ & & & e & u \\ l & l & & & r \end{pmatrix}
\qquad
\begin{pmatrix} 19.00 & & 21.00 & 21.00 & \\ 0.63 & 21.00 & -13.26 & -13.26 & \\ & 0.57 & 23.58 & 7.58 & \\ & & & 5.00 & 21.00 \\ 0.63 & 0.57 & -0.24 & -0.77 & 34.20 \end{pmatrix}
$$

Original matrix A — Factors $F = L + U - I$

$s = 19, u = 21, p = 16, e = 5, r = 18, l = 12$

Figure 2.1: A 5×5 matrix and its L and U factors.

is located in `SuperLU/EXAMPLE/superlu.c`.

The program first initializes the three arrays, `a[]`, `asub[]` and `xa[]`, which store the nonzero coefficients of matrix A, their row indices, and the indices indicating the beginning of each column in the coefficient and row index arrays. This storage format is called compressed column format, also known as Harwell-Boeing format [10]. Next, the two utility routines `dCreate_CompCol_Matrix()` and `dCreate_Dense_Matrix()` are called to set up the matrix structures for A and B, respectively. The routine `set_default_options()` sets the default values to the input `options` argument. This controls how the matrix will be factorized and how the system will be solved. After calling the `SuperLU` routine `dgssv()`, the B matrix is overwritten by the solution matrix X. In the end, all the dynamically allocated data structures are de-allocated by calling various utility routines.

`SuperLU` can perform more general tasks, which will be explained later.

```
#include "dsp_defs.h"

main(int argc, char *argv[])
{
/*
 * Purpose
 * =======
 *
 * This is the small 5x5 example used in the Sections 1 and 2 of the
 * User's Guide to illustrate how to call a SuperLU routine, and the
 * matrix data structures used by SuperLU.
 *
 */
    SuperMatrix A, L, U, B;
    double   *a, *rhs;
    double   s, u, p, e, r, l;
    int      *asub, *xa;
    int      *perm_r; /* row permutations from partial pivoting */
    int      *perm_c; /* column permutation vector */
    int      nrhs, info, i, m, n, nnz, permc_spec;
    superlu_options_t options;
    SuperLUStat_t stat;
```

```
/* Initialize matrix A. */
m = n = 5;
nnz = 12;
if ( !(a = doubleMalloc(nnz)) ) ABORT("Malloc fails for a[].");
if ( !(asub = intMalloc(nnz)) ) ABORT("Malloc fails for asub[].");
if ( !(xa = intMalloc(n+1)) ) ABORT("Malloc fails for xa[].");
s = 19.0; u = 21.0; p = 16.0; e = 5.0; r = 18.0; l = 12.0;
a[0] = s; a[1] = l; a[2] = l; a[3] = u; a[4] = l; a[5] = l;
a[6] = u; a[7] = p; a[8] = u; a[9] = e; a[10]= u; a[11]= r;
asub[0] = 0; asub[1] = 1; asub[2] = 4; asub[3] = 1;
asub[4] = 2; asub[5] = 4; asub[6] = 0; asub[7] = 2;
asub[8] = 0; asub[9] = 3; asub[10]= 3; asub[11]= 4;
xa[0] = 0; xa[1] = 3; xa[2] = 6; xa[3] = 8; xa[4] = 10; xa[5] = 12;

/* Create matrix A in the format expected by SuperLU. */
dCreate_CompCol_Matrix(&A, m, n, nnz, a, asub, xa, SLU_NC, SLU_D, SLU_GE);

/* Create right-hand side matrix B. */
nrhs = 1;
if ( !(rhs = doubleMalloc(m * nrhs)) ) ABORT("Malloc fails for rhs[].");
for (i = 0; i < m; ++i) rhs[i] = 1.0;
dCreate_Dense_Matrix(&B, m, nrhs, rhs, m, SLU_DN, SLU_D, SLU_GE);

if ( !(perm_r = intMalloc(m)) ) ABORT("Malloc fails for perm_r[].");
if ( !(perm_c = intMalloc(n)) ) ABORT("Malloc fails for perm_c[].");

/* Set the default input options. */
set_default_options(&options);
options.ColPerm = NATURAL;

/* Initialize the statistics variables. */
StatInit(&stat);

dgssv(&options, &A, perm_c, perm_r, &L, &U, &B, &stat, &info);

dPrint_CompCol_Matrix("A", &A);
dPrint_CompCol_Matrix("U", &U);
dPrint_SuperNode_Matrix("L", &L);
print_int_vec("\nperm_r", m, perm_r);

/* De-allocate storage */
SUPERLU_FREE (rhs);
SUPERLU_FREE (perm_r);
SUPERLU_FREE (perm_c);
Destroy_CompCol_Matrix(&A);
Destroy_SuperMatrix_Store(&B);
Destroy_SuperNode_Matrix(&L);
```

```
    Destroy_CompCol_Matrix(&U);
    StatFree(&stat);
}
```

2.3 Matrix data structures

SuperLU uses a principal data structure SuperMatrix (defined in SRC/supermatrix.h) to represent a general matrix, sparse or dense. Figure 2.2 gives the specification of the SuperMatrix structure. The SuperMatrix structure contains two levels of fields. The first level defines all the properties of a matrix which are independent of how it is stored in memory. In particular, it specifies the following three orthogonal properties: storage type (Stype) indicates the type of the 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 XX a storage format called XXformat, such as NCformat, SCformat, etc.

The SuperMatrix type so defined can accommodate various types of matrix structures and appropriate operations to be applied on them, although currently SuperLU implements only a subset of this collection. Specifically, matrices A, L, U, B, and X can have the following types:

	A	L	U	B	X
Stype	SLU_NC or SLU_NR	SLU_SC	SLU_NC	SLU_DN	SLU_DN
Dtype[1]	any	any	any	any	any
Mtype	SLU_GE	SLU_TRLU	SLU_TRU	SLU_GE	SLU_GE

In what follows, we illustrate the storage schemes defined by Stype. Following C's convention, all array indices and locations below are zero-based.

- A may have storage type SLU_NC or SLU_NR. The SLU_NC format is the same as the Harwell-Boeing sparse matrix format [10], that is, the compressed column storage.

```
typedef struct {
    int  nnz;     /* number of nonzeros in the matrix */
    void *nzval;  /* array of nonzero values packed by column */
    int  *rowind; /* array of row indices of the nonzeros */
    int  *colptr; /* colptr[j] stores the location in nzval[] and rowind[]
                     which starts column j. It has ncol+1 entries,
                     and colptr[ncol] = nnz. */
} NCformat;
```

  The SLU_NR format is the compressed row storage defined below.

```
typedef struct {
    int  nnz;     /* number of nonzeros in the matrix */
    void *nzval;  /* array of nonzero values packed by row */
    int  *colind; /* array of column indices of the nonzeros */
    int  *rowptr; /* rowptr[j] stores the location in nzval[] and colind[]
```

[1]Dtype can be one of SLU_S, SLU_D, SLU_C or SLU_Z.

```
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 */
    SLU_NR,        /* row-wise, not supernodal */
    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;
```

Figure 2.2: `SuperMatrix` data structure.

```
                         which starts row j. It has nrow+1 entries,
                         and rowptr[nrow] = nnz. */
} NRformat;
```

The factorization and solve routines in SuperLU are designed to handle column-wise storage only. If the input matrix A is in row-oriented storage, i.e., in SLU_NR format, then the driver routines (dgssv() and dgssvx()) actually perform the LU decomposition on A^T, which is column-wise, and solve the system using the L^T and U^T factors. The data structures holding L and U on output are different (swapped) from the data structures you get from column-wise input. For more detailed descriptions about this process, please refer to the leading comments of the routines dgssv() and dgssvx().

Alternatively, the users may call a utility routine dCompRow_to_CompCol() to convert the input matrix in SLU_NR format to another matrix in SLU_NC format, before calling SuperLU. The definition of this routine is

```
void dCompRow_to_CompCol(int m, int n, int nnz,
                         double *a, int *colind, int *rowptr,
                         double **at, int **rowind, int **colptr);
```

This conversion takes time proportional to the number of nonzeros in A. However, it requires storage for a separate copy of matrix A.

- L is a supernodal matrix with the storage type SLU_SC. Due to the supernodal structure, L is in fact stored as a sparse block lower triangular matrix [5].

```
typedef struct {
    int  nnz;           /* number of nonzeros in the matrix */
    int  nsuper;        /* index of the last supernode */
    void *nzval;        /* array of nonzero values packed by column */
    int  *nzval_colptr; /* nzval_colptr[j] stores the location in
                           nzval[] which starts column j */
    int  *rowind;       /* array of compressed row indices of
                           rectangular supernodes */
    int  *rowind_colptr;/* rowind_colptr[j] stores the location in
                           rowind[] which starts column j */
    int  *col_to_sup;   /* col_to_sup[j] is the supernode number to
                           which column j belongs */
    int  *sup_to_col;   /* sup_to_col[s] points to the starting column
                           of the s-th supernode */
} SCformat;
```

- Both B and X are stored as conventional two-dimensional arrays in column-major order, with the storage type SLU_DN.

```
typedef struct {
    int lda;     /* leading dimension */
    void *nzval; /* array of size lda-by-ncol to represent
                    a dense matrix */
} DNformat;
```

Figure 2.3 shows the data structures for the example matrices in Figure 2.1.
For a description of `NCPformat`, see section 2.5.1.

2.4 `Options` argument

`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:

- `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 base 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.

- `SymmetricMode`
  Specifies whether to use the symmetric mode.

- `DiagPivotThresh`
  Specifies the threshold used for a diagonal entry to be an acceptable pivot.

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

The routine `set_default_options()` sets the following default values:

```
Fact              = DOFACT         /* factor from scratch */
Trans             = NOTRANS
Equil             = YES
ColPerm           = COLAMD
SymmetricMode     = NO
```

- ```
  A = { Stype = SLU_NC; Dtype = SLU_D; Mtype = SLU_GE; nrow = 5; ncol = 5;
        *Store = { nnz = 12;
                   nzval = [ 19.00, 12.00, 12.00, 21.00, 12.00, 12.00, 21.00,
                             16.00, 21.00, 5.00, 21.00, 18.00 ];
                   rowind = [ 0, 1, 4, 1, 2, 4, 0, 2, 0, 3, 3, 4 ];
                   colptr = [ 0, 3, 6, 8, 10, 12 ];
                 }
  }
  ```

- ```
  U = { Stype = SLU_NC; Dtype = SLU_D; Mtype = SLU_TRU; nrow = 5; ncol = 5;
        *Store = { nnz = 11;
                   nzval = [ 21.00, -13.26, 7.58, 21.00 ];
                   rowind = [ 0, 1, 2, 0 ];
                   colptr = [ 0, 0, 0, 1, 4, 4 ];
                 }
  }
  ```

- ```
  L = { Stype = SLU_SC; Dtype = SLU_D; Mtype = SLU_TRLU; nrow = 5; ncol = 5;
        *Store = { nnz = 11;
                   nsuper = 2;
                   nzval = [ 19.00, 0.63, 0.63, 21.00, 0.57, 0.57, -13.26,
                             23.58, -0.24, 5.00, -0.77, 21.00, 34.20 ];
                   nzval_colptr = [ 0 3, 6, 9, 11, 13 ];
                   rowind = [ 0, 1, 4, 1, 2, 4, 3, 4 ];
                   rowind_colptr = [ 0, 3, 6, 6, 8, 8 ];
                   col_to_sup = [ 0, 1, 1, 2, 2 ];
                   sup_to_col = [ 0, 1, 3, 5 ];
                 }
  }
  ```

Figure 2.3: The data structures for a 5×5 matrix and its LU factors, as represented in the `SuperMatrix` data structure. Zero-based indexing is used.
  ```

```
DiagPivotThresh   = 1.0              /* partial pivoting */
IterRefine        = NOREFINE
PrintStat         = YES
```

The other possible values for each field are documented in the source code `SRC/util.h`. The users can reset each default value according to their needs.

2.5 Permutations

Two permutation matrices are involved in the solution process. In fact, the actual factorization we perform is $P_r A P_c^T = LU$, where P_r is determined from partial pivoting (with a threshold pivoting option), and P_c is a column permutation chosen either by the user or `SuperLU`, usually to make the L and U factors as sparse as possible. P_r and P_c are represented by two integer vectors `perm_r[]` and `perm_c[]`, which are the permutations of the integers $(0 : m-1)$ and $(0 : n-1)$, respectively.

2.5.1 Ordering for sparsity

Column reordering for sparsity is completely separate from the LU factorization. The column permutation P_c should be applied before calling the factorization routine `dgstrf()`. In principle, any ordering heuristic used for symmetric matrices can be applied to $A^T A$ (or $A + A^T$ if the matrix is nearly structurally symmetric) to obtain P_c. Currently, we provide the following ordering options through `options` argument. The `options.ColPerm` field can take the following values:

- `NATURAL`: use natural ordefring (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 $A^T A$.
- `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 `perm_c[]` as an input, which may be obtained from any other ordering algorithm. For example, the nested-dissection type of ordering codes include Metis [18], Chaco [15] and Scotch [28].

Alternatively, the users can provide their own column permutation vector. For example, it may be an ordering suitable for the underlying physical problem. Both driver routines `dgssv` and `dgssvx` take `perm_c[]` as an input argument.

After permutation P_c is applied to A, we use `SLU_NCP` format to represent the permuted matrix AP_c^T, in which the consecutive columns of nonzeros may not be stored contiguously in memory. Therefore, we need two separate arrays of pointers, `colbeg[]` and `colend[]`, to indicate the beginning and end of each column in `nzval[]` and `rowind[]`.

```
typedef struct {
    int  nnz;     /* number of nonzeros in the matrix */
    void *nzval;  /* array of nonzero values, packed by column */
    int  *rowind; /* array of row indices of the nonzeros */
    int  *colbeg; /* colbeg[j] points to the location in nzval[] and rowind[]
```

```
                         which starts column j */
      int  *colend; /* colend[j] points to one past the location in nzval[]
                         and rowind[] which ends column j */
  } NCPformat;
```

2.5.2 Partial pivoting with threshold

We have included a threshold pivoting parameter $u \in [0, 1]$ to control numerical stability. The user can choose to use a row permutation obtained from a previous factorization. (The argument `options.Fact = SamePattern_SameRowPerm` should be passed to the factorization routine `dgstrf()`.) The pivoting subroutine `dpivotL()` checks whether this choice of pivot satisfies the threshold; if not, it will try the diagonal element. If neither of the above satisfies the threshold, the maximum magnitude element in the column will be used as the pivot. The pseudo-code of the pivoting policy for column j is given below.

(1) compute $thresh = u\ |a_{mj}|$, 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;

Two special values of u result in the following two strategies:

- $u = 0.0$: either use user-specified pivot order if available, or else use diagonal pivot;
- $u = 1.0$: classical partial pivoting.

2.6 Symmetric Mode

In many applications, 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*. In this case, the `options.SymmetricMode = YES` must be set.

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 may not be Identity. Please refer to [22] for more discussion on the symmetric mode.

2.7 Memory management for L and U

In the sparse LU algorithm, the amount of space needed to hold the data structures of L and U cannot be accurately predicted prior to the factorization. The dynamically growing arrays include those for the nonzero values (`nzval[]`) and the compressed row indices (`rowind[]`) of L, and for the nonzero values (`nzval[]`) and the row indices (`rowind[]`) of U.

Two alternative memory models are presented to the user:

- system-level – based on C's dynamic allocation capability (`malloc/free`);
- user-level – based on a user-supplied `work[]` array of size `lwork` (in bytes). This is similar to Fortran-style handling of work space. `Work[]` is organized as a two-ended stack, one end holding the L and U data structures, the other end holding the auxiliary arrays of known size.

Except for the different ways to allocate/deallocate space, the logical view of the memory organization is the same for both schemes. Now we describe the policies in the memory module.

At the outset of the factorization, we guess there will be `FILL*nnz(A)` fills in the factors and allocate corresponding storage for the above four arrays, where `nnz(A)` is the number of nonzeros in original matrix A, and `FILL` is an integer, say 20. (The value of `FILL` can be set in an inquiry function `sp_ienv()`, see section 2.10.3.) If this initial request exceeds the physical memory constraint, the `FILL` factor is repeatedly reduced, and attempts are made to allocate smaller arrays, until the initial allocation succeeds.

During the factorization, if any array size exceeds the allocated bound, we expand it as follows. We first allocate a chunk of new memory of size `EXPAND` times the old size, then copy the existing data into the new memory, and then free the old storage. The extra copying is necessary, because the factorization algorithm requires that each of the aforementioned four data structures be *contiguous* in memory. The values of `FILL` and `EXPAND` are normally set to 20 and 1.5, respectively. See `xmemory.c` for details.

After factorization, we do not garbage-collect the extra space that may have been allocated. Thus, there will be external fragmentation in the L and U data structures. The settings of `FILL` and `EXPAND` should take into account the trade-off between the number of expansions and the amount of fragmentation.

Arrays of known size, such as various column pointers and working arrays, are allocated just once. All dynamically-allocated working arrays are freed after factorization.

2.8 User-callable routines

The naming conventions, calling sequences and functionality of these routines mimic the corresponding LAPACK software [1]. In the routine names, such as `dgstrf`, we use the two letters `GS` to denote *general sparse* matrices. The leading letter`x` stands for `S, D, C`, or `Z`, specifying the data type.

2.8.1 Driver routines

We provide two types of driver routines for solving systems of linear equations. 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 (some of them optionally):
  - solve $A^T X = B$;
  - equilibrate the system (scale A's rows and columns to have unit norm) 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.

These driver routines cover all the functionality of the computational routines. We expect that most users can simply use these driver routines to fulfill their tasks with no need to bother with the computational routines.

2.8.2 Computational routines

The users can invoke the following computational routines, instead of the driver routines, to directly control the behavior of `SuperLU`. The computational routines can only handle column-oriented storage.

- `dgstrf()`: Factorize.

  This implements the first-time factorization, or later re-factorization with the same nonzero pattern. In re-factorizations, the code has the ability to use the same column permutation P_c and row permutation P_r obtained from a previous factorization. Several scalar arguments control how the LU decomposition and the numerical pivoting should be performed. `dgstrf()` can handle non-square matrices.

- `dgstrs()`: Triangular solve.

  This takes the L and U triangular factors, the row and column permutation vectors, and the right-hand side to compute a solution matrix X of $AX = B$ or $A^T X = B$.

- `dgscon()`: Estimate condition number.

  Given the matrix A and its factors L and U, this estimates the condition number in the one-norm or infinity-norm. The algorithm is due to Hager and Higham [16], and is the same as `CONDEST` in sparse Matlab.

- `dgsequ()/dlaqgs()`: Equilibrate.

  `dgsequ` first computes the row and column scalings D_r and D_c which would make each row and each column of the scaled matrix $D_r A D_c$ have equal norm. `dlaqgs` then applies them to the original matrix A if it is indeed badly scaled. The equilibrated A overwrites the original A.

- `dgsrfs()`: Refine solution.

  Given A, its factors L and U, and an initial solution X, this does iterative refinement, using the same precision as the input data. It also computes forward and backward error bounds for the refined solution.

2.8.3 Utility routines

The utility routines can help users create and destroy the `SuperLU` matrices easily. These routines reside in two places: `SRC/util.c` contains the routines that are precision-independent; `SRC/{s,d,c,z}util.c` contains the routines dependent on precision. Here, we list the prototypes of these routines.

```
/* Create a supermatrix in compressed column format. A is the output. */
dCreate_CompCol_Matrix(SuperMatrix *A, int m, int n, int nnz,
                       double *nzval, int *rowind, int *colptr,
                       Stype_t stype, Dtype_t dtype, Mtype_t mtype);

/* Create a supermatrix in compressed row format. A is the output. */
dCreate_CompRow_Matrix(SuperMatrix *A, int m, int n, int nnz,
                       double *nzval, int *colind, int *rowptr,
                       Stype_t stype, Dtype_t dtype, Mtype_t mtype);

/* Copy matrix A into matrix B, both in compressed column format. */
dCopy_CompCol_Matrix(SuperMatrix *A, SuperMatrix *B);

/* Create a supermatrix in dense format. X is the output.*/
dCreate_Dense_Matrix(SuperMatrix *X, int m, int n, double *x, int ldx,
                     Stype_t stype, Dtype_t dtype, Mtype_t mtype);

/* Create a supermatrix in supernodal format. L is the output. */
dCreate_SuperNode_Matrix(SuperMatrix *L, int m, int n, int nnz,
                         double *nzval, int *nzval_colptr, int *rowind,
                         int *rowind_colptr, int *col_to_sup, int *sup_to_col,
                         Stype_t stype, Dtype_t dtype, Mtype_t mtype);

/* Convert the compressed row fromat to the compressed column format. */
dCompRow_to_CompCol(int m, int n, int nnz,
                    double *a, int *colind, int *rowptr,
                    double **at, int **rowind, int **colptr);

/* Print a supermatrix in compressed column format. */
dPrint_CompCol_Matrix(char *what, SuperMatrix *A);

/* Print a supermatrix in supernodal format. */
dPrint_SuperNode_Matrix(char *what, SuperMatrix *A);

/* Print a supermatrix in dense format. */
dPrint_Dense_Matrix(char *what, SuperMatrix *A);

/* Deallocate the storage structure *Store. */
Destroy_SuperMatrix_Store(SuperMatrix *A);

/* Deallocate the supermatrix structure in compressed column format. */
Destroy_CompCol_Matrix(SuperMatrix *A)

/* Deallocate the supermatrix structure in supernodal format. */
Destroy_SuperNode_Matrix(SuperMatrix *A)

/* Deallocate the supermatrix structure in permuted compressed column format. */
```

```
Destroy_CompCol_Permuted(SuperMatrix *A)

/* Deallocate the supermatrix structure in dense format. */
Destroy_Dense_Matrix(SuperMatrix *A)
```

2.9 Matlab interface

In the SuperLU/MATLAB subdirectory, we have developed a set of MEX-files interface to Matlab. Typing make in this directory produces executables to be invoked in Matlab. The current Makefile is set up so that the MEX-files are compatible with Matlab Version 5. The user should edit Makefile for Matlab Version 4 compatibility. Right now, only the factor routine dgstrf() and the simple driver routine dgssv() are callable by invoking superlu and lusolve in Matlab, respectively. Superlu and lusolve correspond to the two Matlab built-in functions lu and \ . In Matlab, when you type

```
help superlu
```

you will find the following description about superlu's functionality and how to use it.

```
SUPERLU : Supernodal LU factorization

Executive summary:

[L,U,p] = superlu(A)          is like [L,U,P] = lu(A), but faster.
[L,U,prow,pcol] = superlu(A)  preorders the columns of A by min degree,
                                  yielding A(prow,pcol) = L*U.

Details and options:

With one input and two or three outputs, SUPERLU has the same effect as LU,
except that the pivoting permutation is returned as a vector, not a matrix:

[L,U,p] = superlu(A) returns unit lower triangular L, upper triangular U,
          and permutation vector p with A(p,:) = L*U.
[L,U] = superlu(A) returns permuted triangular L and upper triangular U
          with A = L*U.

With a second input, the columns of A are permuted before factoring:

[L,U,prow] = superlu(A,psparse) returns triangular L and U and permutation
          prow with A(prow,psparse) = L*U.
[L,U] = superlu(A,psparse) returns permuted triangular L and triangular U
          with A(:,psparse) = L*U.
Here psparse will normally be a user-supplied permutation matrix or vector
to be applied to the columns of A for sparsity.  COLMMD is one way to get
such a permutation; see below to make SUPERLU compute it automatically.
(If psparse is a permutation matrix, the matrix factored is A*psparse'.)

With a fourth output, a column permutation is computed and applied:
```

```
[L,U,prow,pcol] = superlu(A,psparse)  returns triangular L and U and
          permutations prow and pcol with A(prow,pcol) = L*U.
          Here psparse is a user-supplied column permutation for sparsity,
          and the matrix factored is A(:,psparse) (or A*psparse' if the
          input is a permutation matrix).  Output pcol is a permutation
          that first performs psparse, then postorders the etree of the
          column intersection graph of A.  The postorder does not affect
          sparsity, but makes supernodes in L consecutive.
[L,U,prow,pcol] = superlu(A,0) is the same as ... = superlu(A,I); it does
          not permute for sparsity but it does postorder the etree.
[L,U,prow,pcol] = superlu(A) is the same as ... = superlu(A,colmmd(A));
          it uses column minimum degree to permute columns for sparsity,
          then postorders the etree and factors.
```

For a description about `lusolve`'s functionality and how to use it, you can type

```
help lusolve
```

```
LUSOLVE : Solve linear systems by supernodal LU factorization.

x = lusolve(A, b) returns the solution to the linear system A*x = b,
    using a supernodal LU factorization that is faster than Matlab's
    builtin LU.  This m-file just calls a mex routine to do the work.

By default, A is preordered by column minimum degree before factorization.
Optionally, the user can supply a desired column ordering:

x = lusolve(A, b, pcol) uses pcol as a column permutation.
    It still returns x = A\b, but it factors A(:,pcol) (if pcol is a
    permutation vector) or A*Pcol (if Pcol is a permutation matrix).

x = lusolve(A, b, 0) suppresses the default minimum degree ordering;
    that is, it forces the identity permutation on columns.
```

Two M-files `trysuperlu.m` and `trylusolve.m` are written to test the correctness of `superlu` and `lusolve`. In addition to testing the residual norms, they also test the function invocations with various number of input/output arguments.

2.10 Installation

2.10.1 File structure

The top level SuperLU/ directory is structured as follows:

```
SuperLU/README    instructions on installation
SuperLU/CBLAS/    needed BLAS routines in C, not necessarily fast
SuperLU/EXAMPLE/  example programs
SuperLU/INSTALL/  test machine dependent parameters; this Users' Guide
```

```
SuperLU/MAKE_INC/ sample machine-specific make.inc files
SuperLU/MATLAB/   Matlab mex-file interface
SuperLU/SRC/      C source code, to be compiled into the superlu.a library
SuperLU/TESTING/  driver routines to test correctness
SuperLU/Makefile  top level Makefile that does installation and testing
SuperLU/make.inc  compiler, compile flags, library definitions and C
                  preprocessor definitions, included in all Makefiles
```

Before installing the package, you may need to edit `SuperLU/make.inc` for your system. This make include file is referenced inside each of the `Makefiles` in the various subdirectories. As a result, there is no need to edit the `Makefiles` in the subdirectories. All information that is machine specific has been defined in `make.inc`.

Sample machine-specific `make.inc` are provided in the `MAKE_INC/` subdirectory for several systems, including IBM RS/6000, DEC Alpha, SunOS 4.x, SunOS 5.x (Solaris), HP-PA and SGI Iris 4.x. When you have selected the machine on which you wish to install SuperLU, you may copy the appropriate sample include file (if one is present) into `make.inc`. For example, if you wish to run SuperLU on an IBM RS/6000, you can do:

```
cp MAKE_INC/make.rs6k make.inc
```

For systems other than those listed above, slight modifications to the `make.inc` file will need to be made. In particular, the following three items should be examined:

1. The BLAS library.
   If there is a BLAS library available on your machine, you may define the following in `make.inc`:

   ```
   BLASDEF = -DUSE_VENDOR_BLAS
   BLASLIB = <BLAS library you wish to link with>
   ```

   The `CBLAS/` subdirectory contains the part of the C BLAS needed by the `SuperLU` package. However, these codes are intended for use only if there is no faster implementation of the BLAS already available on your machine. In this case, you should do the following:

   1) In `make.inc`, undefine (comment out) BLASDEF, define:
      ```
      BLASLIB = ../blas$(PLAT).a
      ```
   2) In the `SuperLU/` directory, type:
      ```
      make blaslib
      ```
      to make the BLAS library from the routines in the `CBLAS/` subdirectory.

2. C preprocessor definition `CDEFS`.
   In the header file `SRC/Cnames.h`, we use macros to determine how C routines should be named so that they are callable by Fortran.[2] The possible options for `CDEFS` are:

   - `-DAdd_`: Fortran expects a C routine to have an underscore postfixed to the name;
   - `-DNoChange`: Fortran expects a C routine name to be identical to that compiled by C;
   - `-DUpCase`: Fortran expects a C routine name to be all uppercase.

3. The Matlab MEX-file interface.
   The `MATLAB/` subdirectory includes Matlab C MEX-files, so that our factor and solve routines

[2] Some vendor-supplied BLAS libraries do not have C interfaces. So the re-naming is needed in order for the `SuperLU` BLAS calls (in C) to interface with the Fortran-style BLAS.

can be called as alternatives to those built into Matlab. In the file `SuperLU/make.inc`, define MATLAB to be the directory in which Matlab is installed on your system, for example:

```
MATLAB = /usr/local/matlab
```

At the SuperLU/ directory, type:

```
make matlabmex
```

to build the MEX-file interface. After you have built the interface, you may go to the `MATLAB/` subdirectory to test the correctness by typing (in Matlab):

```
trysuperlu
trylusolve
```

A `Makefile` is provided in each subdirectory. The installation can be done completely automatically by simply typing `make` at the top level.

2.10.2 Testing

The test programs in `SuperLU/INSTALL` subdirectory test two routines:

- `slamch()/dlamch()` determines properties of the floating-point arithmetic at run-time (both single and double precision), such as the machine epsilon, underflow threshold, overflow threshold, and related parameters;
- `SuperLU_timer_()` returns the time in seconds used by the process. This function may need to be modified to run on your machine.

The test programs in the `SuperLU/TESTING` subdirectory are designed to test all the functions of the driver routines, especially the expert drivers. The Unix shell script files `xtest.csh` are used to invoke tests with varying parameter settings. The input matrices include an actual sparse matrix `SuperLU/EXAMPLE/g10` of dimension 100×100,[3] and numerous matrices with special properties from the LAPACK test suite. Table 2.1 describes the properties of the test matrices.

For each command line option specified in `dtest.csh`, the test program `ddrive` reads in or generates an appropriate matrix, calls the driver routines, and computes a number of test ratios to verify that each operation has performed correctly. If the test ratio is smaller than a preset threshold, the operation is considered to be correct. Each test matrix is subject to the tests listed in Table 2.2.

Let r be the residual $r = b - Ax$, and let m_i be the number of nonzeros in row i of A. Then the componentwise backward error $BERR$ and forward error $FERR$ [1] are calculated by:

$$BERR = \max_i \frac{|r|_i}{(|A|\ |x| + |b|)_i}\ .$$

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

Here, f is a nonnegative vector whose components are computed as $f_i = |r|_i + m_i\ \varepsilon\ (|A|\ |x| + |b|)_i$, and the norm in the numerator is estimated using the same subroutine used for estimating the condition number. $BERR$ measures the smallest relative perturbation one can make to each entry

[3] Matrix `g10` is first generated with the structure of the 10-by-10 five-point grid, and random numerical values. The columns are then permuted by COLMMD ordering from Matlab.

Matrix type	Description
0	sparse matrix `g10`
1	diagonal
2	upper triangular
3	lower triangular
4	random, $\kappa = 2$
5	first column zero
6	last column zero
7	last $n/2$ columns zero
8	random, $\kappa = \sqrt{0.1/\varepsilon}$
9	random, $\kappa = 0.1/\varepsilon$
10	scaled near underflow
11	scaled near overflow

Table 2.1: Properties of the test matrices. ε is the machine epsilon and κ is the condition number of matrix A. Matrix types with one or more columns set to zero are used to test the error return codes.

Test Type	Test ratio	Routines
0	$\|\|LU - A\|\|/(n\|\|A\|\|\varepsilon)$	`dgstrf`
1	$\|\|b - Ax\|\|/(\|\|A\|\|\ \|\|x\|\|\varepsilon)$	`dgssv`, `dgssvx`
2	$\|\|x - x^*\|\|/(\|\|x^*\|\|\kappa\varepsilon)$	`dgssvx`
3	$\|\|x - x^*\|\|/(\|\|x^*\|\|\ FERR)$	`dgssvx`
4	$BERR/\varepsilon$	`dgssvx`

Table 2.2: Types of tests. x^* is the true solution, $FERR$ is the error bound, and $BERR$ is the backward error.

of A and of b so that the computed solution is an exact solution of the perturbed problem. $FERR$ is an estimated bound on the error $\|x^* - x\|_\infty / \|x\|_\infty$, where x^* is the true solution. For further details on error analysis and error bounds estimation, see [1, Chapter 4] and [2].

2.10.3 Performance-tuning parameters

`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 reference [5] for their definitions.)

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

Users are encouraged to modify this subroutine to set the tuning parameters for their own local environment. 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. In our technical paper [5], we described a detailed methodology for setting these parameters for high performance.

The $relax$ parameter is usually set between 4 and 8. The other parameter values which give good performance on several machines are listed in Table 2.3. In a supernode-panel update, if the

Machine	On-chip Cache	External Cache	w	*maxsup*	*rowblk*	*colblk*
RS/6000-590	256 KB	–	8	100	200	40
MIPS R8000	16 KB	4 MB	20	100	800	100
Alpha 21064	8 KB	512 KB	8	100	400	40
Alpha 21164	8 KB-L1 96 KB-L2	4 MB	16	50	100	40
Sparc 20	16 KB	1 MB	8	100	400	50
UltraSparc-I	16 KB	512 KB	8	100	400	40
Cray J90	–	–	1	100	1000	100

Table 2.3: Typical blocking parameter values for several machines.

updating supernode is too large to fit in cache, then a 2D block partitioning of the supernode is used, in which *rowblk* and *colblk* determine that a block of size $rowblk \times colblk$ is used to update current panel.

If *colblk* is set greater than *maxsup*, then the program will never use 2D blocking. For example, for the Cray J90 (which does not have cache), $w = 1$ and 1D blocking give good performance; more levels of blocking only increase overhead.

2.11 Example programs

In the `SuperLU/EXAMPLE/` subdirectory, we present a few sample programs to illustrate how to use various functions provded in `SuperLU`. The users can modify these examples to suit their applications. Here are the brief descriptions of the double precision version of the examples:

- `dlinsol`: use simple driver `dgssv()` to solve a linear system one time.
- `dlinsol1`: use simple driver `dgssv()` in the symmetric mode.
- `dlinsolx`: use `dgssvx()` with the full (default) set of options to solve a linear system.
- `dlinsolx1`: use `dgssvx()` to factorize A first, then solve the system later.
- `dlinsolx2`: use `dgssvx()` to solve systems repeatedly with the same sparsity pattern of matrix A.
- `superlu`: the small 5x5 sample program in Section 2.2.

In this directory, a `Makefile` is provided to generate the executables, and a `README` file describes how to run these examples.

2.12 Calling from Fortran

The `SuperLU/FORTRAN/` subdirectory contains an example of using `SuperLU` from a Fortran program. The General rules for mixing Fortran and C programs are as follows.

- Arguments in C are passed by value, while in Fortran are passed by reference. So we always pass the address (as a pointer) in the C calling routine. (You cannot make a call with numbers directly in the parameters.)

- Fortran uses 1-based array addressing, while C uses 0-based. Therefore, the row indices (`rowind[]`) and the integer pointers to arrays (`colptr[]`) should be adjusted before they are passed into a C routine.

Because of the above language differences, in order to embed `SuperLU` in a Fortran environment, users are required to use "wrapper" routines (in C) for all the `SuperLU` routines that will be called from Fortran programs. The example `c_fortran_dgssv.c` in the `FORTRAN/` directory shows how a wrapper program should be written. This program is listed below.

```
#include "dsp_defs.h"

#define HANDLE_SIZE  8

typedef struct {
    SuperMatrix *L;
    SuperMatrix *U;
    int *perm_c;
    int *perm_r;
} factors_t;

int
c_fortran_dgssv_(int *iopt, int *n, int *nnz, int *nrhs, double *values,
                 int *rowind, int *colptr, double *b, int *ldb,
                 int factors[HANDLE_SIZE], /* a handle containing the pointer
                                              to the factored matrices */
                 int *info)

{
/*
 * This routine can be called from Fortran.
 *
 * iopt (input) int
 *      Specifies the operation:
 *      = 1, performs LU decomposition for the first time
 *      = 2, performs triangular solve
 *      = 3, free all the storage in the end
 *
 * factors (input/output) integer array of size 8
 *      If iopt == 1, it is an output and contains the pointer pointing to
 *                    the structure of the factored matrices.
 *      Otherwise, it it an input.
 *
 */
    SuperMatrix A, AC, B;
    SuperMatrix *L, *U;
    int *perm_r; /* row permutations from partial pivoting */
    int *perm_c; /* column permutation vector */
    int *etree;  /* column elimination tree */
```

```
SCformat *Lstore;
NCformat *Ustore;
int      i, panel_size, permc_spec, relax;
trans_t  trans;
double   drop_tol = 0.0;
mem_usage_t   mem_usage;
superlu_options_t options;
SuperLUStat_t stat;
factors_t *LUfactors;

trans = NOTRANS;

if ( *iopt == 1 ) { /* LU decomposition */

    /* Set the default input options. */
    set_default_options(&options);

    /* Initialize the statistics variables. */
    StatInit(&stat);

    /* Adjust to 0-based indexing */
    for (i = 0; i < *nnz; ++i) --rowind[i];
    for (i = 0; i <= *n; ++i) --colptr[i];

    dCreate_CompCol_Matrix(&A, *n, *n, *nnz, values, rowind, colptr,
                           SLU_NC, SLU_D, SLU_GE);
    L = (SuperMatrix *) SUPERLU_MALLOC( sizeof(SuperMatrix) );
    U = (SuperMatrix *) SUPERLU_MALLOC( sizeof(SuperMatrix) );
    if ( !(perm_r = intMalloc(*n)) ) ABORT("Malloc fails for perm_r[].");
    if ( !(perm_c = intMalloc(*n)) ) ABORT("Malloc fails for perm_c[].");
    if ( !(etree = intMalloc(*n)) ) ABORT("Malloc fails for etree[].");

    /*
     * Get column permutation vector perm_c[], according to permc_spec:
     *   permc_spec = 0: natural ordering
     *   permc_spec = 1: minimum degree on structure of A'*A
     *   permc_spec = 2: minimum degree on structure of A'+A
     *   permc_spec = 3: approximate minimum degree for unsymmetric matrices
     */
    permc_spec = 3;
    get_perm_c(permc_spec, &A, perm_c);

    sp_preorder(&options, &A, perm_c, etree, &AC);

    panel_size = sp_ienv(1);
    relax = sp_ienv(2);
```

```
    dgstrf(&options, &AC, drop_tol, relax, panel_size,
           etree, NULL, 0, perm_c, perm_r, L, U, &stat, info);

    if ( *info == 0 ) {
        Lstore = (SCformat *) L->Store;
        Ustore = (NCformat *) U->Store;
        printf("No of nonzeros in factor L = %d\n", Lstore->nnz);
        printf("No of nonzeros in factor U = %d\n", Ustore->nnz);
        printf("No of nonzeros in L+U = %d\n", Lstore->nnz + Ustore->nnz);
        dQuerySpace(L, U, &mem_usage);
        printf("L\\U MB %.3f\ttotal MB needed %.3f\texpansions %d\n",
               mem_usage.for_lu/1e6, mem_usage.total_needed/1e6,
               mem_usage.expansions);
    } else {
        printf("dgstrf() error returns INFO= %d\n", *info);
        if ( *info <= *n ) { /* factorization completes */
           dQuerySpace(L, U, &mem_usage);
           printf("L\\U MB %.3f\ttotal MB needed %.3f\texpansions %d\n",
                  mem_usage.for_lu/1e6, mem_usage.total_needed/1e6,
                  mem_usage.expansions);
        }
    }

    /* Restore to 1-based indexing */
    for (i = 0; i < *nnz; ++i) ++rowind[i];
    for (i = 0; i <= *n; ++i) ++colptr[i];

    /* Save the LU factors in the factors handle */
    LUfactors = (factors_t*) SUPERLU_MALLOC(sizeof(factors_t));
    LUfactors->L = L;
    LUfactors->U = U;
    LUfactors->perm_c = perm_c;
    LUfactors->perm_r = perm_r;
    factors[0] = (int) LUfactors;

    /* Free un-wanted storage */
    SUPERLU_FREE(etree);
    Destroy_SuperMatrix_Store(&A);
    Destroy_CompCol_Permuted(&AC);
    StatFree(&stat);

} else if ( *iopt == 2 ) { /* Triangular solve */
    /* Initialize the statistics variables. */
    StatInit(&stat);

    /* Extract the LU factors in the factors handle */
    LUfactors = (factors_t*) factors[0];
```

```
        L = LUfactors->L;
        U = LUfactors->U;
        perm_c = LUfactors->perm_c;
        perm_r = LUfactors->perm_r;

        dCreate_Dense_Matrix(&B, *n, *nrhs, b, *ldb, SLU_DN, SLU_D, SLU_GE);

        /* Solve the system A*X=B, overwriting B with X. */
        dgstrs (trans, L, U, perm_c, perm_r, &B, &stat, info);

        Destroy_SuperMatrix_Store(&B);
        StatFree(&stat);

    } else if ( *iopt == 3 ) { /* Free storage */
        /* Free the LU factors in the factors handle */
        LUfactors = (factors_t*) factors[0];
        SUPERLU_FREE (LUfactors->perm_r);
        SUPERLU_FREE (LUfactors->perm_c);
        Destroy_SuperNode_Matrix(LUfactors->L);
        Destroy_CompCol_Matrix(LUfactors->U);
        SUPERLU_FREE (LUfactors->L);
        SUPERLU_FREE (LUfactors->U);
        SUPERLU_FREE (LUfactors);
    } else {
        fprintf(stderr, "Invalid iopt=%d passed to c_fortran_dgssv()\n");
        exit(-1);
    }
}
```

Since the matrix structures in C cannot be directly returned to Fortran, we use a handle named `factors` to access those structures. The handle is essentially an integer pointer pointing to the factored matrices obtained from `SuperLU`. So the factored matrices are opaque objects to the Fortran program, but can only be manipulated from the C wrapper program.

The Fortran program `FORTRAN/f77_main.f` shows how a Fortran program may call `c_fortran_dgssv()`, and is listed below. A `README` file in this directory describes how to compile and run this program.

```
      program f77_main
      integer maxn, maxnz
      parameter ( maxn = 10000, maxnz = 100000 )
      integer rowind(maxnz), colptr(maxn)
      real*8  values(maxnz), b(maxn)
      integer n, nnz, nrhs, ldb, info
      integer factors(8), iopt
*
*     Read the matrix file in Harwell-Boeing format
      call hbcode1(n, n, nnz, values, rowind, colptr)
*
```

```
      nrhs = 1
      ldb = n
      do i = 1, n
         b(i) = 1
      enddo
*
* First, factorize the matrix. The factors are stored in factor() handle.
      iopt = 1
      call c_fortran_dgssv( iopt, n, nnz, nrhs, values, rowind, colptr,
     $                      b, ldb, factors, info )
*
      if (info .eq. 0) then
         write (*,*) 'Factorization succeeded'
      else
         write(*,*) 'INFO from factorization = ', info
      endif
*
* Second, solve the system using the existing factors.
      iopt = 2
      call c_fortran_dgssv( iopt, n, nnz, nrhs, values, rowind, colptr,
     $                      b, ldb, factors, info )
*
      if (info .eq. 0) then
         write (*,*) 'Solve succeeded'
         write (*,*) (b(i), i=1, 10)
      else
         write(*,*) 'INFO from triangular solve = ', info
      endif

* Last, free the storage allocated inside SuperLU
      iopt = 3
      call c_fortran_dgssv( iopt, n, nnz, nrhs, values, rowind, colptr,
     $                      b, ldb, factors, info )
*
      stop
      end
```

Chapter 3

Multithreaded SuperLU (Version 1.1)

3.1 About SuperLU_MT

Among the various steps of the solution process in the sequential SuperLU, the LU factorization dominates the computation; it usually takes more than 95% of the sequential runtime for large sparse linear systems. We have designed and implemented an algorithm to perform the factorization in parallel on machines with a shared address space and multithreading. The parallel algorithm is based on the efficient sequential algorithm implemented in SuperLU. Although we attempted to minimize the amount of changes to the sequential code, there are still a number of non-trivial modifications to the serial SuperLU, mostly related to the matrix data structures and memory organization. All these changes are summarized in Table 3.1 and their impacts on performance are studied thoroughly in [6, 21]. In this part of the Users' Guide, we describe only the changes that the user should be aware of. Other than these differences, most of the material in chapter 2 is still applicable.

Construct	Parallel algorithm
panel	restricted so it does not contain branchings in the elimination tree
supernode	restricted to be a fundamental supernode in the elimination tree
supernode storage	use either static or dynamic upper bound (section 3.4.2)
pruning & DFS	use both $G(L^T)$ and pruned $G(L^T)$ to avoid locking

Table 3.1: The differences between the parallel and the sequential algorithms.

3.2 Storage types for L and U

As in the sequential code, the type for the factored matrices L and U is `SuperMatrix` (Figure 2.2), however, their storage formats (stored in `*Store`) are changed. In the parallel algorithm, the adjacent panels of the columns may be assigned to different processes, and they may be finished and put in global memory out of order. That is, the consecutive columns or supernodes may not be stored contiguously in memory. Thus, in addition to the pointers to the beginning of each column or supernode, we need pointers to the end of the column or supernode. In particular, the storage type for L is `SCP` (Supernode, Column-wise and Permuted), defined as:

```
typedef struct {
```

```
    int  nnz;          /* number of nonzeros in the matrix */
    int  nsuper;       /* number of supernodes */
    void *nzval;       /* pointer to array of nonzero values,
                          packed by column */
    int *nzval_colbeg; /* nzval_colbeg[j] points to beginning of column j
                          in nzval[] */
    int *nzval_colend; /* nzval_colend[j] points to one past the last
                          element of column j in nzval[] */
    int *rowind;       /* pointer to array of compressed row indices of
                          the supernodes */
    int *rowind_colbeg;/* rowind_colbeg[j] points to beginning of column j
                          in rowind[] */
    int *rowind_colend;/* rowind_colend[j] points to one past the last
                          element of column j in rowind[] */
    int *col_to_sup;   /* col_to_sup[j] is the supernode number to which
                          column j belongs */
    int *sup_to_colbeg;/* sup_to_colbeg[s] points to the first column
                          of the s-th supernode /
    int *sup_to_colend;/* sup_to_colend[s] points to one past the last
                          column of the s-th supernode */
} SCPformat;
```

The storage type for U is NCP, defined as:

```
typedef struct {
    int  nnz;     /* number of nonzeros in the matrix */
    void *nzval;  /* pointer to array of nonzero values, packed by column */
    int  *rowind; /* pointer to array of row indices of the nonzeros */
    int  *colbeg; /* colbeg[j] points to the location in nzval[] and rowind[]
                     which starts column j */
    int  *colend; /* colend[j] points to one past the location in nzval[]
                     and rowind[] which ends column j */
} NCPformat;
```

The table below summarizes the data and storage types of all the matrices involved in the parallel routines:

	A	L	U	B	X
Stype	SLU_NC or SLU_NR	SLU_SCP	SLU_NCP	SLU_DN	SLU_DN
Dtype	SLU_D	SLU_D	SLU_D	SLU_D	SLU_D
Mtype	SLU_GE	SLU_TRLU	SLU_TRU	SLU_GE	SLU_GE

3.3 User-callable routines

As in the sequential SuperLU, we provide both computational routines and driver routines. To name those routines that involve parallelization in the call-graph, we prepend a letter `p` to the names of their sequential counterparts, for example `pdgstrf`. For the purely sequential routines, we use the same names as before. Here, we only list the routines that are different from the sequential ones.

3.3.1 Driver routines

We provide two types of driver routines for solving systems of linear equations. The driver routines can handle both column- and row-oriented storage schemes.

- A simple driver `pdgssv`, which solves the system $AX = B$ by factorizing A and overwriting B with the solution X.

- An expert driver `pdgssvx`, which, in addition to the above, also performs the following functions (some of them optionally):

  - solve $A^T X = B$;
  - equilibrate the system (scale A's rows and columns to have unit norm) 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.

3.3.2 Computational routines

The user can invoke the following computational routines to directly control the behavior of SuperLU. The computational routines can only handle column-oriented storage. Except for the parallel factorization routine `pdgstrf`, all the other routines are identical to those appeared in the sequential superlu.

- `pdgstrf`: Factorize (in parallel).

  This implements the first-time factorization, or later re-factorization with the same nonzero pattern. In re-factorizations, the code has the ability to use the same column permutation P_c and row permutation P_r obtained from a previous factorization. Several scalar arguments control how the LU decomposition and the numerical pivoting should be performed. `pdgstrf` can handle non-square matrices.

- `dgstrs`: Triangular solve.

  This takes the L and U triangular factors, the row and column permutation vectors, and the right-hand side to compute a solution matrix X of $AX = B$ or $A^T X = B$.

- `dgscon`: Estimate condition number.

  Given the matrix A and its factors L and U, this estimates the condition number in the one-norm or infinity-norm. The algorithm is due to Hager and Higham [16], and is the same as `condest` in sparse Matlab.

- `dgsequ/dlaqgs`: Equilibrate.

  `dgsequ` first computes the row and column scalings D_r and D_c which would make each row and each column of the scaled matrix $D_r A D_c$ have equal norm. `dlaqgs` then applies them to the original matrix A if it is indeed badly scaled. The equilibrated A overwrites the original A.

- `dgsrfs`: Refine solution.

  Given A, its factors L and U, and an initial solution X, this does iterative refinement, using the same precision as the input data. It also computes forward and backward error bounds for the refined solution.

3.4 Installation

3.4.1 File structure

The top level SuperLU_MT/ directory is structured as follows:

```
SuperLU_MT/README    instructions on installation
SuperLU_MT/CBLAS/    needed BLAS routines in C, not necessarily fast
SuperLU_MT/EXAMPLE/  example programs
SuperLU_MT/INSTALL/  test machine dependent parameters; the Users' Guide
SuperLU_MT/SRC/      C source code, to be compiled into superlu_mt.a library
SuperLU_MT/TESTING/  driver routines to test correctness
SuperLU_MT/Makefile  top level Makefile that does installation and testing
SuperLU_MT/make.inc  compiler, compile flags, library definitions and C
                     preprocessor definitions, included in all Makefiles.
```

We have ported the parallel programs to a number of platforms, which are reflected in the make include files provided in the top level directory, for example, `make.sun, make.sgi, make.cray and make.pthreads`. If you are using one of these machines, such as a Sun, you can simply copy `make.sun` into `make.inc` before compiling. If you are not using any of the machines to which we have ported, you will need to read section 3.6 about the porting instructions.

The rest of the installation and testing procedure is similar to that described in section 2.10 for the serial SuperLU. Then, you can type `make` at the top level directory to finish installation. In the `SuperLU_MT/TESTING` subdirectory, you can type `pdtest.csh` to perform testings.

3.4.2 Performance issues

Memory management for L and U

In the sequential SuperLU, four data arrays associated with the L and U factors can be expanded dynamically, as described in section 2.7. In the parallel code, the expansion is hard and costly to implement, because when a process detects that an array bound is exceeded, it has to send a signal to and suspend the execution of the other processes. Then the detecting process can proceed with the array expansion. After the expansion, this process must wake up all the suspended processes.

In this release of the parallel code, we have not yet implemented the above expansion mechanism. For now, the user must pre-determine an estimated size for each of the four arrays through the inquiry function `sp_ienv()`. There are two interpretations for each integer value `FILL` returned by calling this function with `ispec = 6, 7, or 8`. A negative number is interpreted as the fills growth factor, that is, the program will allocate `(-FILL)*nnz(A)` elements for the corresponding array. A positive number is interpreted as the true amount the user wants to allocate, that is, the program will allocate `FILL` elements for the corresponding array. In both cases, if the initial request exceeds the physical memory constraint, the sizes of the arrays are repeatedly reduced until the initial allocation succeeds.

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

`Ispec` specifies the parameter to be returned:

ispec = ...
= 6: size of the array to store the values of the L supernodes (`nzval`)
= 7: size of the array to store the columns in U (`nzval/rowind`)
= 8: size of the array to store the subscripts of the L supernodes (`rowind`);

If the actual fill exceeds any array size, the program will abort with a message showing the current column when failure occurs, and indicating how many elements are needed up to the current column. The user may reset a larger fill parameter for this array and then restart the program.

To make the storage allocation more efficient for the supernodes in L, we devised a special storage scheme. The need for this special treatment and how we implement it are fully explained and studied in [6, 21]. Here, we only sketch the main idea. Recall that the parallel algorithm assigns one panel of columns to one process. Two consecutive panels may be assigned to two different processes, even though they may belong to the same supernode discovered later. Moreover, a third panel may be finished by a third process and put in memory between these two panels, resulting in the columns of a supernode being noncontiguous in memory. This is undesirable, because then we cannot directly call BLAS routines using this supernode unless we pay the cost of copying the columns into contiguous memory first. To overcome this problem, we exploited the observation that the nonzero structure for L is contained in that of the Householder matrix H from the Householder sparse QR transformation [11, 12]. Furthermore, it can be shown that a fundamental supernode of L is always contained in a fundamental supernode of H. This containment property is true for for any row permutation P_r in $P_rA = LU$. Therefore, we can pre-allocate storage for the L supernodes based on the size of H supernodes. Fortunately, there exists a fast algorithm (almost linear in the number of nonzeros of A) to compute the size of H and the supernodes partition in H [13].

In practice, the above static prediction is fairly tight for most problems. However, for some others, the number of nonzeros in H greatly exceeds the number of nonzeros in L. To handle this situation, we implemented an algorithm that still uses the supernodes partition in H, but dynamically searches the supernodal graph of L to obtain a much tighter bound for the storage. Table 6 in [6] demonstrates the storage efficiency achieved by both static and dynamic approach.

In summary, our program tries to use the static prediction first for the L supernodes. In this case, we ignore the integer value given in the function `sp_ienv(6)`, and simply use the nonzero count of H. If the user finds that the size of H is too large, he can invoke the dynamic algorithm at runtime by setting the following UNIX shell environment variable:

```
setenv SuperLU_DYNAMIC_SNODE_STORE 1
```

The dynamic algorithm incurs runtime overhead. For example, this overhead is usually between 2% and 15% on a single processor RS/6000-590 for a range of test matrices.

Symmetric structure pruning

In both serial and parallel algorithms, we have implemented Eisenstat and Liu's symmetric pruning idea of representing the graph $G(L^T)$ by a reduced graph G', and thereby reducing the DFS traversal time. A subtle difficulty arises in the parallel implementation.

When the owning process of a panel starts DFS (depth-first search) on G' built so far, it only sees the partial graph, because the part of G' corresponding to the busy panels down the elimination tree is not yet complete. So the structural prediction at this stage can miss some nonzeros. After performing the updates from the finished supernodes, the process will wait for all the busy descendant panels to finish and perform more updates from them. Now, we make a conservative assumption that all these busy panels will update the current panel so that their nonzero structures are included in the current panel.

This approximate scheme works fine for most problems. However, we found that this conservatism may sometimes cause a large number of structural zeros (they are related to the supernode amalgamation performed at the bottom of the elimination tree) to be included and they in turn are propagated through the rest of the factorization.

We have implemented an exact structural prediction scheme to overcome this problem. In this scheme, when each numerical nonzero is scattered into the sparse accumulator array, we set the occupied flag as well. Later when we accumulate the updates from the busy descendant panels, we check the occupied flags to determine the exact nonzero structure. This scheme avoids unnecessary zero propagation at the expense of runtime overhead, because setting the occupied flags must be done in the inner loop of the numeric updates.

We recommend that the user use the approximate scheme (by default) first. If the user finds that the amount of fill from the parallel factorization is substantially greater than that from the sequential factorization, he can then use the accurate scheme. To invoke the second scheme, the user should recompile the code by defining the macro:

```
-D SCATTER_FOUND
```

for the C preprocessor.

The inquiry function `sp_ienv()`

For some user controllable constants, such as the blocking parameters and the size of the global storage for L and U, SuperLU_MT calls the inquiry function `sp_ienv()` to retrieve their values. The declaration of this function is

```
int sp_ienv(int ispec).
```

The full meanings of the returned values are as follows:

ispec = 1: the panel size w
= 2: the relaxation parameter to control supernode amalgamation ($relax$)
= 3: the maximum allowable size for a supernode ($maxsup$)
= 4: the minimum row dimension for 2D blocking to be used ($rowblk$)
= 5: the minimum column dimension for 2D blocking to be used ($colblk$)
= 6: size of the array to store the values of the L supernodes ($nzval$)
= 7: size of the array to store the columns in U ($nzval/rowind$)
= 8: size of the array to store the subscripts of the L supernodes ($rowind$)

We should take into account the trade-off between cache reuse and amount of parallelism in order to set the appropriate w and $maxsup$. Since the parallel algorithm assigns one panel factorization to one process, large values may constrain concurrency, even though they may be good for uniprocessor performance. We recommend that w and $maxsup$ be set a bit smaller than the best values used in the sequential code.

The settings for parameters 2, 4 and 5 are the same as those described in section 2.10.3. The settings for parameters 6, 7 and 8 are discussed in section 3.4.2.

In the file `SRC/sp_ienv.c`, we provide sample settings of these parameters for several machines.

make.inc	Platforms	Programming Model	Environment Variable
make.pthreads	Machines with POSIX threads	pthreads	
make.sun	Sun Ultra Enterprise	Solaris threads	
make.alpha	DEC Alpha Servers	DECthreads	
make.sgi	SGI Power Challenge	parallel C	`MPC_NUM_THREADS`
make.origin	SGI/Cray Origin2000	parallel C	`MP_SET_NUMTHREADS`
make.cray	Cray C90/J90	microtasking	`NCPUS`

Table 3.2: Platforms on which SuperLU_MT was tested.

3.5 Example programs

In the `SuperLU_MT/EXAMPLE/` subdirectory, we present a few sample programs to illustrate the complete calling sequences to use the simple and expert drivers to solve systems of equations. Examples are also given to illustrate how to perform a sequence of factorizations for the matrices with the same sparsity pattern, and how SuperLU_MT can be integrated into the other multithreaded application such that threads are created only once. A `Makefile` is provided to generate the executables. A `README` file in this directory shows how to run these examples. The leading comment in each routine describes the functionality of the example.

3.6 Porting to other platforms

We have provided the parallel interfaces for a number of shared memory machines. Table 3.2 lists the platforms on which we have tested the library, and the respective `make.inc` files. The most portable interface for shared memory programming is POSIX threads [29], since nowadays many commercial UNIX operating systems have support for it. We call our POSIX threads interface the `Pthreads` interface. To use this interface, you can copy `make.pthreads` into `make.inc` and then compile the library. In the last column of Table 3.2, we list the runtime environment variable to be set in order to use multiple CPUs. For example, to use 4 CPUs on the Origin2000, you need to set the following before running the program:

```
setenv MP_SET_NUMTHREADS 4
```

In the source code, all the platform specific constructs are enclosed in the C `#ifdef` preprocessor statement. If your platform is different from any one listed in Table 3.2, you need to go to these places and create the parallel constructs suitable for your machine. The two constructs, concurrency and synchronization, are explained in the following two subsections, respectively.

3.6.1 Creating multiple threads

Right now, only the factorization routine `pdgstrf` is parallelized, since this is the most time-consuming part in the whole solution process. There is one single thread of control on entering and exiting `pdgstrf`. Inside this routine, more than one thread may be created. All the newly created threads begin by calling the thread function `pdgstrf_thread` and they are concurrently executed on multiple processors. The thread function `pdgstrf_thread` expects a single argument of type `void*`, which is a pointer to the structure containing all the shared data objects.

Mutex	Critical region
`ULOCK`	allocate storage for a column of matrix U
`LLOCK`	allocate storage for row subscripts of matrix L
`LULOCK`	allocate storage for the values of the supernodes
`NSUPER_LOCK`	increment supernode number `nsuper`
`SCHED_LOCK`	invoke `Scheduler()` which may update global task queue

Table 3.3: Five mutex variables.

3.6.2 Use of mutexes

Although the threads `pdgstrf_thread` execute independently of each other, they share the same address space and can communicate efficiently through shared variables. Problems may arise if two threads try to access (at least one is to modify) the shared data at the same time. Therefore, we must ensure that all memory accesses to the same data are mutually exclusive. There are five critical regions in the program that must be protected by mutual exclusion. Since we want to allow different processors to enter different critical regions simultaneously, we use five mutex variables as listed in Table 3.3. The user should properly initialize them in routine `ParallelInit`, and destroy them in routine `ParallelFinalize`. Both these routines are in file `pxgstrf_synch.c`.

Chapter 4

Distributed SuperLU with MPI (Version 2.0)

4.1 About SuperLU_DIST

In this part, we describe the SuperLU_DIST library designed for distributed memory parallel computers. The parallel programming model is SPMD. The library is implemented in ANSI C, using MPI [26] for communication, and so is highly portable. We have tested the code on a number of platforms, including Cray T3E, IBM SP, and Berkeley NOW. The library includes routines to handle both real and complex matrices in double precision. The parallel routine names for the double-precision real version start with letters "pd" (such as pdgstrf); the parallel routine names for double-precision complex version start with letters "pz" (such as pzgstrf).

4.2 Formats of the input matrices A and B

We provide two input interfaces for matrices A and B. One is the global interface, another is an entirely distributed interface.

4.2.1 Global input

The input matrices A and B are globally available (replicated) on all the processes. The storage type for A is SLU_NC, as in sequential case (see Section 2.3). The user-callable routines with this interface all have the names "xxxxxxx_ABglobal". If there is sufficient memory, this interface is faster than the distributed input interface described in the next section, because the latter requires more data re-distribution at different stages of the algorithm.

4.2.2 Distributed input

Both input matrices 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 sparse matrix A is stored in a compressed row format, called SLU_NR_loc storage type, which is defined below.

```
typedef struct {
    int nnz_loc;  /* number of nonzeros in the local submatrix */
```

```
    int m_loc;     /* number of rows local to this process */
    int fst_row;   /* row number of the first row in the local submatrix */
    void *nzval;   /* pointer to array of nonzero values, packed by row */
    int *rowptr;   /* pointer to array of beginning of rows in nzval[]
                      and colind[]  */
    int *colind;   /* pointer to array of column indices of the nonzeros */
} NRformat_loc;
```

Let m_i be the number of rows owned by the ith process. Then the global row dimension for A is $nrow = \sum_{i=0}^{P-1} m_i$. The global column dimension is $ncol$. Both $nrow$ and $ncol$ are recorded in the higher level `SuperMatrix` data structure, see Figure 2.2. The utility routine `dCreate_CompRowLoc_Matrix_dist` can help the user to create the structure for A. The definition of this routine is

```
void dCreate_CompRowLoc_Matrix_dist(SuperMatrix *A, int m, int n,
                                    int nnz_loc, int m_loc, int fst_row,
                                    double *nzval, int *colind, int *rowptr,
                                    Stype_t stype, Dtype_t dtype, Mtype_t mtype);
```

where, the first argument is output and the rest are inputs.

The local full matrix B is stored in the standard Fortran style column major format, with dimension $m\_loc \times nrhs$, and ldb refers to the local leading dimension in the local storage.

4.3 Distributed data structures for L and U

We distribute both L and U matrices in a two-dimensional block-cyclic fashion. We first identify the supernode boundary based on the nonzero structure of L. This supernode partition is then used as the block partition in both row and column dimensions for both L and U. The size of each block is matrix dependent. It should be clear that all the diagonal blocks are square and full (we store zeros from U in the upper triangle of the diagonal block), whereas the off-diagonal blocks may be rectangular and may not be full. The matrix in Figure 4.1 illustrates such a partition. By block-cyclic mapping we mean block (I, J) $(0 \le I, J \le N-1)$ is mapped into the process at coordinate $\{I \; mod \; \texttt{nprow}, J \; mod \; \texttt{npcol}\}$ of the $\texttt{nprow} \times \texttt{npcol}$ 2D process grid. Using this mapping, a block $L(I, J)$ in the factorization is only needed by the row of processes that own blocks in row I. Similarly, a block $U(I, J)$ is only needed by the column of processes that own blocks in column J.

In this 2D mapping, each block column of L resides on more than one process, namely, a column of processes. For example in Figure 4.1, the second block column of L resides on the column processes {1, 4}. Process 4 only owns two nonzero blocks, which are not contiguous in the global matrix. The schema on the right of Figure 4.1 depicts the data structure to store the nonzero blocks on a process. Besides the numerical values stored in a Fortran-style array `nzval[]` in column major order, we need the information to interpret the location and row subscript of each nonzero. This is stored in an integer array `index[]`, which includes the information for the whole block column and for each individual block in it. Note that many off-diagonal blocks are zero and hence 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 $\lceil N/\texttt{npcol} \rceil$ block columns of L, so it needs $\lceil N/\texttt{nprow} \rceil$ pairs of `index`/`nzval` arrays.

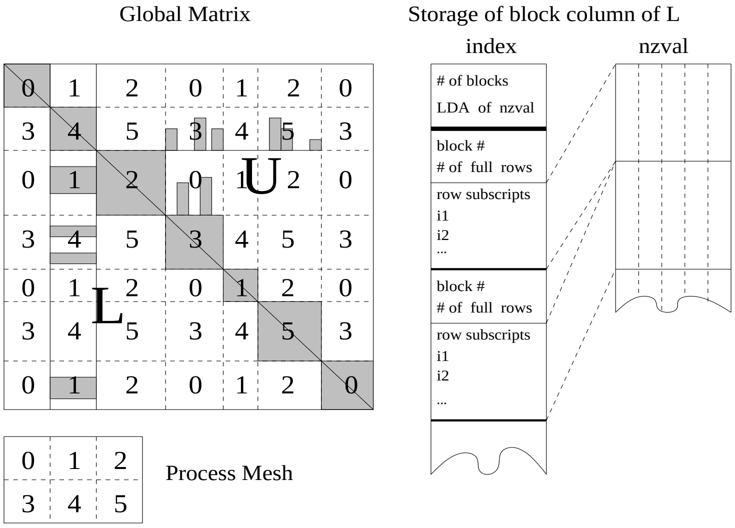

Figure 4.1: The 2 block-cyclic layout and the data structure to store a local block column of L.

For 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. Similar 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 4.1 (see [5] for details on the skyline structure). Therefore, the organization of the `index[]` array is different from that for L, which we omit showing in the figure.

4.4 Process grid and MPI communicator

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 $0 \ldots N_p - 1$.

4.4.1 2D process grid

For `SuperLU_DIST` library, we create a new process group derived from an existing group using N_g processes. There is a good reason to use a new group rather than `MPI_COMM_WORLD`, that is, the message passing calls of the SuperLU library will be isolated from those in other libraries or in the user's code. For better scalability of the LU factorization, we map the 1D array of N_g processes into a logical 2D process grid. This grid will have `nprow` process rows and `npcol` process columns, such that $\texttt{nprow} * \texttt{npcol} = N_g$. A process can be referenced either by its rank in the new group or by its coordinates within the grid. The routine `superlu_gridinit` maps already-existing 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 $\texttt{nprow} * \texttt{npcol}$ processes from the base MPI communicator `Bcomm`, and assign them to the grid in a row-major ordering. The input argument `Bcomm` is an MPI communicator representing the existing base group upon which the new group will be 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.

The macros MYROW(iam, grid) and MYCOL(iam, grid) give the row and column coordinates in the 2D grid of the process who has rank iam.

NOTE: All processes in the base group, including those not in the new group, must call this grid creation routine. This is required by the MPI routine MPI_Comm_create *to create a new communicator.*

4.4.2 Arbitrary grouping of processes

It is sometimes desirable to divide up the processes into several subgroups, each of which performs independent work of a single application. In this situation, we cannot simply use the first nprow * npcol processes to define the grid. A more sophisticated process-to-grid mapping routine superlu_gridmap is designed 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 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} position of the 2D process grid. After grid creation, this subset of processes is logically numbered in a consistent manner with the initial set of processes; that is, they have the ranks in the range $0 \ldots$ nprow * npcol $- 1$ in the new grid. For example, if we want to map 6 processes with ranks $11 \ldots 16$ into a 2×3 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

NOTE: All processes in the base group, including those not in the new group, must call this routine.

Superlu_gridinit simply calls superlu_gridmap with usermap[] holding the first nprow * npcol process ranks.

4.5 Basic steps to solve a linear system

In this section, we use a complete sample program to illustrate the basic steps required to use SuperLU_DIST. This program is listed below, and is also available as EXAMPLE/pddrive.c in the source code distribution. All the routines must include the header file superlu_ddefs.h (or superlu_zdefs.h, the complex counterpart) which contains the definitions of the data types, the macros and the function prototypes.

```
#include <math.h>
#include "superlu_ddefs.h"

main(int argc, char *argv[])
/*
 * Purpose
 * =======
 *
 * The driver program PDDRIVE.
 *
 * This example illustrates how to use PDGSSVX with the full
 * (default) options to solve a linear system.
 *
 * Five basic steps are required:
 *   1. Initialize the MPI environment and the SuperLU process grid
 *   2. Set up the input matrix and the right-hand side
 *   3. Set the options argument
 *   4. Call pdgssvx
 *   5. Release the process grid and terminate the MPI environment
 *
 * On the Cray T3E, the program may be run by typing
 *    mpprun -n <procs> pddrive -r <proc rows> -c <proc columns> <input_file>
 *
 */
{
    superlu_options_t options;
    SuperLUStat_t stat;
    SuperMatrix A;
    ScalePermstruct_t ScalePermstruct;
    LUstruct_t LUstruct;
    SOLVEstruct_t SOLVEstruct;
    gridinfo_t grid;
    double   *berr;
    double   *b, *xtrue;
    int_t    m, n, nnz;
    int_t    nprow, npcol;
    int      iam, info, ldb, ldx, nrhs;
    char     trans[1];
    char     **cpp, c;
    FILE *fp, *fopen();
```

```
nprow = 1;  /* Default process rows.      */
npcol = 1;  /* Default process columns.   */
nrhs = 1;   /* Number of right-hand side. */

/* ------------------------------------------------------------
   INITIALIZE MPI ENVIRONMENT.
   ------------------------------------------------------------*/
MPI_Init( &argc, &argv );

/* Parse command line argv[]. */
for (cpp = argv+1; *cpp; ++cpp) {
    if ( **cpp == '-' ) {
        c = *(*cpp+1);
        ++cpp;
        switch (c) {
          case 'h':
              printf("Options:\n");
              printf("\t-r <int>: process rows    (default %d)\n", nprow);
              printf("\t-c <int>: process columns (default %d)\n", npcol);
              exit(0);
              break;
          case 'r': nprow = atoi(*cpp);
              break;
          case 'c': npcol = atoi(*cpp);
              break;
        }
    } else { /* Last arg is considered a filename */
        if ( !(fp = fopen(*cpp, "r")) ) {
            ABORT("File does not exist");
        }
        break;
    }
}

/* ------------------------------------------------------------
   INITIALIZE THE SUPERLU PROCESS GRID.
   ------------------------------------------------------------*/
superlu_gridinit(MPI_COMM_WORLD, nprow, npcol, &grid);

/* Bail out if I do not belong in the grid. */
iam = grid.iam;
if ( iam >= nprow * npcol ) goto out;

/* ------------------------------------------------------------
   GET THE MATRIX FROM FILE AND SETUP THE RIGHT HAND SIDE.
   ------------------------------------------------------------*/
```

```
dcreate_matrix(&A, nrhs, &b, &ldb, &xtrue, &ldx, fp, &grid);

if ( !(berr = doubleMalloc_dist(nrhs)) )
    ABORT("Malloc fails for berr[].");

/* ------------------------------------------------------------
   NOW WE SOLVE THE LINEAR SYSTEM.
   ------------------------------------------------------------*/

/* Set the default input options. */
set_default_options_dist(&options);

m = A.nrow;
n = A.ncol;

/* Initialize ScalePermstruct and LUstruct. */
ScalePermstructInit(m, n, &ScalePermstruct);
LUstructInit(m, n, &LUstruct);

/* Initialize the statistics variables. */
PStatInit(&stat);

/* Call the linear equation solver. */
pdgssvx(&options, &A, &ScalePermstruct, b, ldb, nrhs, &grid,
        &LUstruct, &SOLVEstruct, berr, &stat, &info);


/* Check the accuracy of the solution. */
pdinf_norm_error(iam, ((NRformat_loc *)A.Store)->m_loc,
                 nrhs, b, ldb, xtrue, ldx, &grid);

PStatPrint(&options, &stat, &grid);        /* Print the statistics. */

/* ------------------------------------------------------------
   DEALLOCATE STORAGE.
   ------------------------------------------------------------*/

PStatFree(&stat);
Destroy_CompRowLoc_Matrix_dist(&A);
ScalePermstructFree(&ScalePermstruct);
Destroy_LU(n, &grid, &LUstruct);
LUstructFree(&LUstruct);
if ( options.SolveInitialized ) {
    dSolveFinalize(&options, &SOLVEstruct);
}
SUPERLU_FREE(b);
SUPERLU_FREE(xtrue);
```

```
    SUPERLU_FREE(berr);

    /* ------------------------------------------------------------
       RELEASE THE SUPERLU PROCESS GRID.
       ------------------------------------------------------------*/
out:
    superlu_gridexit(&grid);

    /* ------------------------------------------------------------
       TERMINATES THE MPI EXECUTION ENVIRONMENT.
       ------------------------------------------------------------*/
    MPI_Finalize();

}
```

Five basic steps are required to call a SuperLU routine:

1. Initialize the MPI environment and the SuperLU process grid.
   This is achieved by the calls to the MPI routine `MPI_Init()` and the SuperLU routine `superlu_gridinit()`. In this example, the communication domain for SuperLU is built upon the MPI default communicator `MPI_COMM_WORLD`. In general, it can be built upon any MPI communicator. Section 4.4 contains the details about this step.

2. Set up the input matrix and the right-hand side.
   This example uses the interface with the distributed input matrices, see Section 4.2.2. In most practical applications, the matrices can be generated on each process without the need to have a centralized place to hold them. But for this example, we let process 0 read the input matrix stored on disk in Harwell-Boeing format [10] (a.k.a. compressed column storage), and distribute it to all the other processes, so that each process only owns a block of rows of matrix. The right-hand side matrix is generated so that the exact solution matrix consists of all ones. The subroutine `dcreate_matrix()` accomplishes this task.

3. Initialize the input arguments: `options, ScalePermstruct, LUstruct, stat`.
   The input argument `options` controls how the linear system would be solved—use equilibration or not, how to order the rows and the columns of the matrix, use iterative refinement or not. The subroutine `set_default_options_dist()` sets the `options` argument so that the solver performs all the functionality. You can also set it up according to your own needs, see section 4.7.1 for the fields of this structure. `ScalePermstruct` is the data structure that stores the several vectors describing the transformations done to A. `LUstruct` is the data structure in which the distributed L and U factors are stored. `Stat` is a structure collecting the statistics about runtime and flop count, etc.

4. Call the SuperLU routine `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 a call to the SuperLU routine `superlu_gridexit()`. When all computations have been completed, the MPI routine `MPI_Finalize()` should be called.

4.6 Algorithmic background

Although partial pivoting is used in both sequential and shared-memory parallel factorization algorithms, it is not used in the distributed-memory parallel algorithm, because it requires dynamic adaptation of data structure and load balancing, and so is hard to make it scalable. We use alternative techniques to stabilize the algorithm, suas as statically pivot large elements to the diagonal, single-precision diagonal adjustment to avoid small pivots, and iterative refinement. Figure 4.2 sketches our GESP algorithm (Gaussian elimination with Static Pivoting). Numerical experiments show that for a wide range of problems, GESP is as stable as GEPP [24].

Figure 4.2: The outline of the GESP algorithm.

(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 $berr \le \frac{1}{2} \cdot lastberr$) **then**
        $x = x + dx$
        $lastberr = berr$
        goto iterate
      endif
(7) If desired, estimate the condition number of A

We have parallelized the most time-consuming steps (4) to (7). The preprocessing and analysis steps (1) to (3) are mostly performed sequentially at present. If the distributed input interface is used (Section 4.2.2), we first gather the distributed graph of A (and the value of A if step (1) is needed) onto one processor. Work is underway to remove this sequential bottleneck.

Step (1) is accomplished by a weighted bipartite matching algorithm due to Duff and Koster [9]. Currently, process 0 computes P_r and then broadcasts it to all the other processes. For step (2), we provide several ordering options, such as multiple minimum degree ordering [25] on the graphs of $A + A^T$. The user can use any other ordering in place of these, such as an ordering based on graph partitioning. (Note, since we will pivot on the diagonal in step (4), an ordering based on the structure of $A + A^T$ almost always yields sparser factors than that based on the structure of $A^T A$. This is different from SuperLU and SuperLU_MT, where we can pivot off-diagonal.) In this step, every process runs the same algorithm independently.

4.7 User-callable routines

4.7.1 Driver routine

There are two driver routines to solve systems of linear equations, which are named `pdgssvx_ABglobal` for the global input interface, and `pdgssvx` for the distributed interface. We recommend that the general users, especially the beginners, use a driver routine rather than the computational routines, because correctly using the driver routine does not require thorough understanding of the underlying data structures. Although the interface of these routines are simple, we expect their rich functionality can meet the requirements of most applications. `Pdgssvx_ABglobal/pdgssvx` perform the following functions:

- Equilibrate the system (scale A's rows and columns to have unit norm) if A is poorly scaled;
- Find a row permutation that makes diagonal of A large relative to the off-diagonal;
- Find a column permutation that preserves the sparsity of the L and U factors;
- Solve the system $AX = B$ for X by factoring A followed by forward and back substitutions;
- Refine the solution X.

Options argument

One important input argument to `pdgssvx_ABglobal/pdgssvx` is `options`, which controls how the linear system will be solved. Although the algorithm presented in Figure 4.2 consists of seven steps, for some matrices not all steps are needed to get accurate solution. For example, for diagonally dominant matrices, choosing the diagonal pivots ensures the stability; there is no need for row pivoting in step (1). 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. (P_r and P_c are implemented as permutation vectors `perm_r` and `perm_c`.) For the above examples, performing all seven steps does more work than necessary. `Options` is used to accommodate the various requirements of applications; it contains the following fields:

- `Fact`
  This option specifies whether or not the factored form of the matrix A is supplied on entry, and if not, how the matrix A will be factored base on some assumptions of the previous history. `fact` can be one of:
  - `DOFACT`: the matrix A will be factorized from scratch.
  - `SamePattern`: the matrix A will be factorized assuming that a factorization of a matrix with the same sparsity pattern was performed prior to this one. Therefore, this factorization will reuse column permutation vector `perm_c`.
  - `SampPattern_SameRowPerm`: the matrix A will be factorized assuming that a factorization of a matrix with the same sparsity pattern and similar numerical values was performed prior to this one. Therefore, this factorization will reuse both row and column permutation vectors `perm_r` and `perm_c`, both row and column scaling factors D_r and D_c, and the distributed data structure set up from the previous symbolic factorization.
  - `FACTORED`: the factored form of A is input.

- Equil
  This option specifies whether to equilibrate the system.
- ColPerm
  This option specifies the column ordering method for fill reduction.
  - NATURAL: natural ordering.
  - MMD_AT_PLUS_A: minimum degree ordering on the structure of $A^T + A$.
  - MMD_ATA: minimum degree ordering on the structure of $A^T A$.
  - COLAMD: approximate minimum degree column ordering.
  - MY_PERMC: use the ordering given in **perm_c** input by the user.
- RowPerm
  This option specifies how to permute rows of the original matrix.
  - NATURAL: use the natural ordering.
  - LargeDiag: use a weighted bipartite matching algorithm to permute the rows to make the diagonal large relative to the off-diagonal.
  - MY_PERMR: use the ordering given in **perm_r** input by the user.
- ReplaceTinyPivot
  This option specifies whether to replace the tiny diagonals by $\sqrt{\varepsilon} \cdot ||A||$ during LU factorization.
- IterRefine
  This option specifies how to perform iterative refinement.
  - NO: no iterative refinement.
  - DOUBLE: accumulate residual in double precision.
  - EXTRA: accumulate residual in extra precision. (*not yet implemented.*)
- SolveInitialized (Used only by the distributed input interface)
  This option specifies whether the initialization has been performed to the triangular solve.
- RefineInitialized (Used only by the distributed input interface)
  This option specifies whether the initialization has been performed to the sparse matrix-vector multiplication routine needed in the iterative refinement.
- PrintStat
  Specifies whether to print the solver's statistics.

There is a routine named **set_default_options_dist()** that sets the default values of these options, which are:

```
fact              = DOFACT          /* factor from scratch */
equil             = YES
colperm           = MMD_AT_PLUS_A
rowperm           = LargeDiag       /* use MC64 */
ReplaceTinyPivot  = YES
IterRefine        = DOUBLE
SolveInitialized  = NO
RefineInitialized = NO
PrintStat         = YES
```

4.7.2 Computational routines

The experienced users can invoke the following computational routines to directly control the behavior of SuperLU_DIST in order to meet their requirements.

- pdgstrf(): Factorize in parallel.
  This routine factorizes the input matrix A (or the scaled and permuted A). It assumes that the distributed data structures for L and U factors are already set up, and the initial values of A are loaded into the data structures. If not, the routine symbfact() should be called to determine the nonzero patterns of the factors, and the routine pddistribute() should be called to distribute the matrix. Pdgstrf() can factor non-square matrices.

- pdgstrs()/pdgstrs_Bglobal(): Triangular solve in parallel.
  This routine solves the system by forward and back substitutions using the the L and U factors computed by pdgstrf(). Pdgstrs() takes distributed B. For pdgstrs_Bglobal(), B must be globally available on all processes.

- pdgsrfs()/pdgsrfs_ABXglobal(): Refine solution in parallel.
  Given A, its factors L and U, and an initial solution X, this routine performs iterative refinement. Pdgsrfs() takes distributed A, B and X. For pdgsrfs_ABXglobal(), A, B and X must be globally available on all processes.

4.7.3 Utility routines

The following utility routines can help users create and destroy the SuperLU_DIST matrices. These routines reside in three places: SRC/util.c, SRC/{d,z}util.c, and SRC/pd,zutil.c. Most of the utility routines in sequential SuperLU can also be used in SuperLU_DIST for the local data, see Section 2.8.3. Here, we only list those new routines specific to SuperLU_DIST. Note that in order to avoid name clash between SuperLU and SuperLU_DIST, we append "_dist" to each routine name in SuperLU_DIST.

```
/* Create a supermatrix in distributed compressed row format. A is output. */
dCreate_CompRowLoc_Matrix_dist(SuperMatrix *A, int_t m, int_t n,
                               int_t nnz_loc, int_t m_loc, int_t fst_row,
                               double *nzval, int_t *colind, int_t *rowptr,
                               Stype_t stype, Dtype_t dtype, Mtype_t mtype);

/* Deallocate the supermatrix in distributed compressed row format. */
Destroy_CompRowLoc_Matrix_dist(SuperMatrix *A);

/* Allocate storage in *ScalePermstruct. */
ScalePermstructInit(const int_t m, const int_t n,
                    ScalePermstruct_t *ScalePermstruct);

/* Deallocate ScalePermstruct */
ScalePermstructFree(ScalePermstruct_t *ScalePermstruct);

/* Allocate storage in *LUstruct. */
LUstructInit(const int_t m, const int_t n, LUstruct_t *LUstruct);
```

```
/* Deallocate the distributed L & U factors in *LUstruct. */
Destroy_LU(int_t n, gridinfo_t *grid, LUstruct_t *LUstruct);

/* Deallocate *LUstruct. */
LUstructFree(LUstruct_t *LUstruct);

/* Initialize the statistics variable. */
PStatInit(SuperLUStat_t *stat);

/* Print the statistics. */
PStatPrint(superlu_options_t *options, SuperLUStat_t *stat,
           gridinfo_t *grid);

/* Deallocate the statistics variable. */
PStatFree(SuperLUStat_t *stat);
```

4.8 Installation

4.8.1 File structure

The top level SuperLU_DIST/ directory is structured as follows:

```
SuperLU_DIST/README     instructions on installation
SuperLU_DIST/CBLAS/     needed BLAS routines in C, not necessarily fast
SuperLU_DIST/EXAMPLE/   example programs
SuperLU_DIST/INSTALL/   test machine dependent parameters; the Users' Guide.
SuperLU_DIST/SRC/       C source code, to be compiled into a library
SuperLU_DIST/Makefile   top level Makefile that does installation and testing
SuperLU_DIST/MAKE_INC   sample machine-specific make.inc files
SuperLU_DIST/make.inc   compiler, compile flags, library definitions and C
                        preprocessor definitions, included in all Makefiles.
                        (You may need to edit it to be suitable for your
                         system before compiling the whole package.)
```

Before installing the package, you may need to edit `SuperLU_DIST/make.inc` for your system. This make include file is referenced inside each of the Makefiles in the various subdirectories. As a result, there is no need to edit the Makefiles in the subdirectories. All information that is machine specific has been defined in this include file.

Sample machine-specific `make.inc` are provided in the `MAKE_INC/` directory for several platforms, such as Cray T3E and IBM SP. When you have selected the machine to which you wish to install `superlud`, you may copy the appropriate sample include file (if one is present) into `make.inc`. For example, if you wish to run on a Cray T3E, you can do:

```
cp MAKE_INC/make.t3e make.inc
```

For the systems other than those listed above, slight modifications to the `make.inc` file will need to be made. In particular, the following items should be examined:

1. The BLAS library.
   If there is a BLAS library available on your machine, you may define the following in `make.inc`:

```
BLASDEF = -DUSE_VENDOR_BLAS
BLASLIB = <BLAS library you wish to link with>
```

The CBLAS/ subdirectory contains the part of the BLAS (in C) needed by SuperLU_DIST package. However, these routines are intended for use only if there is no faster implementation of the BLAS already available on your machine. In this case, you should do the following:

1) In make.inc, undefine (comment out) BLASDEF, define:
```
BLASLIB = ../blas$(PLAT).a
```
2) At the top level SuperLU_DIST directory, type:
```
make blaslib
```
to create the BLAS library from the routines in CBLAS/ subdirectory.

2. C preprocessor definition CDEFS.
In the header file SRC/Cnames.h, we use macros to determine how C routines should be named so that they are callable by Fortran.[1] The possible options for CDEFS are:

- -DAdd_: Fortran expects a C routine to have an underscore postfixed to the name;
- -DNoChange: Fortran expects a C routine name to be identical to that compiled by C;
- -DUpCase: Fortran expects a C routine name to be all uppercase.

A Makefile is provided in each subdirectory. The installation can be done completely automatically by simply typing make at the top level.

4.8.2 Performance-tuning parameters

Similar to sequential SuperLU, several performance related parameters are set in the inquiry function sp_ienv(). The declaration of this function is

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

Ispec specifies the parameter to be returned[2]:

ispec = 2: the relaxation parameter to control supernode amalgamation
= 3: the maximum allowable size for a block
= 6: the estimated fills factor for the adjacency structures of L and U

The values to be returned may be set differently on different machines. The setting of maximum block size (parameter 3) should take into account the local Level 3 BLAS speed, the load balance and the degree of parallelism. Small block size may result in better load balance and more parallelism, but poor individual node performance, and vice versa for large block size.

4.9 Example programs

In the SuperLU_DIST/EXAMPLE/ directory, we present a few sample programs to illustrate the complete calling sequences to use the expert driver to solve systems of equations. These include how to set up the process grid and the input matrix, how to obtain a fill-reducing ordering. A Makefile

[1] Some vendor-supplied BLAS libraries do not have C interfaces. So the re-naming is needed in order for the SuperLU BLAS calls (in C) to interface with the Fortran-style BLAS.
[2] The numbering of 2, 3 and 6 is consistent with that used in SuperLU and SuperLU_MT.

is provided to generate the executables. A README file in this directory shows how to run these examples. The leading comment in each routine describes the functionality of the example. The two basic examples are pddrive_ABglobal() and pddrive(). The first shows how to use the global input interface, and the second shows how to use the distributed input interface.

4.10 Fortran 90 Interface

We developed a complete Fortran 90 interface for SuperLU_DIST. All the interface files and an example driver program are located in the SuperLU_DIST/FORTRAN/ directory. Table 4.1 lists all the files.

f_pddrive.f90	An example Fortran driver routine.
superlu_mod.f90	Fortran 90 module that defines the interface functions to access SuperLU_DIST's data structures.
superlupara.f90	It contains parameters that correspond to SuperLU_DIST's enums.
hbcode1.f90	Fortran function for reading a sparse Harwell-Boeing matrix from the file.
superlu_c2f_wrap.c	C wrapper functions, callable from Fortran. The functions fall into three classes: 1) Those that allocate a structure and return a handle, or deallocate the memory of a structure. 2) Those that get or set the value of a component of a struct. 3) Those that are wrappers for SuperLU_DIST functions.
dcreate_dist_matrix.c	C function for distributing the matrix in a distributed compressed row format.

Table 4.1: The Fortran 90 interface files and an example driver routine.

Note that in this interface, all objects (such as grid, options, etc.) in SuperLU_DIST are *opaque*, meaning their size and structure are not visible to the Fortran user. These opaque objects are allocated, deallocated and operated in the C side and not directly accessible from Fortran side. They can only be accessed via *handles* that exist in Fortran's user space. In Fortran, all handles have type INTEGER. Specifically, in our interface, the size of Fortran handle is defined by superlu_ptr in superlupara.f90. For different systems, the size might need to be changed. Then using these handles, Fortran user can call C wrapper routines to manipulate the opaque objects. For example, you can call f_create_gridinfo(grid_handle) to allocate memory for structure grid, and return a handle grid_handle.

The sample program illustrates the basic steps required to use SuperLU_DIST in Fortran to solve systems of equations. These include how to set up the processor grid and the input matrix, how to call the linear equation solver. This program is listed below, and is also available as f_pddrive.f90 in the subdirectory. Note that the routine must include the moudle superlu_mod which contains the definitions of all parameters and the Fortran wrapper functions. A Makefile is provided to generate the executable. A README file in this directory shows how to run the example.

```
      program f_pddrive
!
```

```
! Purpose
! =======
!
! The driver program F_PDDRIVE.
!
! This example illustrates how to use F_PDGSSVX with the full
! (default) options to solve a linear system.
!
! Seven basic steps are required:
!   1. Create C structures used in SuperLU
!   2. Initialize the MPI environment and the SuperLU process grid
!   3. Set up the input matrix and the right-hand side
!   4. Set the options argument
!   5. Call f_pdgssvx
!   6. Release the process grid and terminate the MPI environment
!   7. Release all structures
!
      use superlu_mod
      include 'mpif.h'
      implicit none
      integer maxn, maxnz, maxnrhs
      parameter ( maxn = 10000, maxnz = 100000, maxnrhs = 10 )
      integer rowind(maxnz), colptr(maxn)
      real*8  values(maxnz), b(maxn), berr(maxnrhs)
      integer n, m, nnz, nrhs, ldb, i, ierr, info, iam
      integer nprow, npcol
      integer init

      integer(superlu_ptr) :: grid
      integer(superlu_ptr) :: options
      integer(superlu_ptr) :: ScalePermstruct
      integer(superlu_ptr) :: LUstruct
      integer(superlu_ptr) :: SOLVEstruct
      integer(superlu_ptr) :: A
      integer(superlu_ptr) :: stat


! Create Fortran handles for the C structures used in SuperLU_DIST
      call f_create_gridinfo(grid)
      call f_create_options(options)
      call f_create_ScalePermstruct(ScalePermstruct)
      call f_create_LUstruct(LUstruct)
      call f_create_SOLVEstruct(SOLVEstruct)
      call f_create_SuperMatrix(A)
      call f_create_SuperLUStat(stat)

! Initialize MPI environment
```

```
      call mpi_init(ierr)

! Initialize the SuperLU_DIST process grid
      nprow = 2
      npcol = 2
      call f_superlu_gridinit(MPI_COMM_WORLD, nprow, npcol, grid)

! Bail out if I do not belong in the grid.
      call get_GridInfo(grid, iam=iam)
      if ( iam >= nprow * npcol ) then
         go to 100
      endif
      if ( iam == 0 ) then
         write(*,*) ' Process grid ', nprow, ' X ', npcol
      endif

! Read Harwell-Boeing matrix, and adjust the pointers and indices
! to 0-based indexing, as required by C routines.
      if ( iam == 0 ) then
         open(file = "g20.rua", status = "old", unit = 5)
         call hbcode1(m, n, nnz, values, rowind, colptr)
         close(unit = 5)
!
         do i = 1, n+1
            colptr(i) = colptr(i) - 1
         enddo
         do i = 1, nnz
            rowind(i) = rowind(i) - 1
         enddo
      endif

! Distribute the matrix to the gird
      call  f_dcreate_matrix_dist(A, m, n, nnz, values, rowind, colptr, grid)

! Setup the right hand side
      nrhs = 1
      call  get_CompRowLoc_Matrix(A, nrow_loc=ldb)
      do i = 1, ldb
         b(i) = 1.0
      enddo

! Set the default input options
      call f_set_default_options(options)

! Change one or more options
!      call set_superlu_options(options,Fact=FACTORED)
```

```
! Initialize ScalePermstruct and LUstruct
      call get_SuperMatrix(A,nrow=m,ncol=n)
      call f_ScalePermstructInit(m, n, ScalePermstruct)
      call f_LUstructInit(m, n, LUstruct)

! Initialize the statistics variables
      call f_PStatInit(stat)

! Call the linear equation solver
      call f_pdgssvx(options, A, ScalePermstruct, b, ldb, nrhs, &
                     grid, LUstruct, SOLVEstruct, berr, stat, info)

      if (info == 0) then
         write (*,*) 'Backward error: ', (berr(i), i = 1, nrhs)
      else
         write(*,*) 'INFO from f_pdgssvx = ', info
      endif

! Deallocate SuperLU allocated storage
      call f_PStatFree(stat)
      call f_Destroy_CompRowLoc_Matrix_dist(A)
      call f_ScalePermstructFree(ScalePermstruct)
      call f_Destroy_LU(n, grid, LUstruct)
      call f_LUstructFree(LUstruct)
      call get_superlu_options(options, SolveInitialized=init)
      if (init == YES) then
         call f_dSolveFinalize(options, SOLVEstruct)
      endif

! Release the SuperLU process grid
100   call f_superlu_gridexit(grid)

! Terminate the MPI execution environment
      call mpi_finalize(ierr)

! Destroy all C structures
      call f_destroy_gridinfo(grid)
      call f_destroy_options(options)
      call f_destroy_ScalePermstruct(ScalePermstruct)
      call f_destroy_LUstruct(LUstruct)
      call f_destroy_SOLVEstruct(SOLVEstruct)
      call f_destroy_SuperMatrix(A)
      call f_destroy_SuperLUStat(stat)

      stop
      end
```

Similar to the driver routine pddrive.c in C, seven basic steps are required to call a SuperLU_DIST routine in Fortran:

1. Create C structures used in SuperLU: grid, options, ScalePermstruct, LUstruct, SOLVEstruct, A and stat. This is achieved by the calls to the C wrapper *"create"* routines f_create_XXX(), where XXX is the name of the corresponding structure.

2. Initialize the MPI environment and the SuperLU process grid. This is achieved by the calls to mpi_init() and the C wrapper routine f_superlu_gridinit(). Note that f_superlu_gridinit() requires the numbers of row and column of the process grid. In this example, we set them to be 2, respectively.

3. Set up the input matrix and the right-hand side. This example uses the distributed input interface, so we need to convert the input matrix to the distributed compressed row format. Process 0 first reads the input matrix stored on disk in Harwell-Boeing format by calling Fortran routine hbcode1(). The file name in this example is `g20.rua`. Then all processes call a C wrapper routine f_dcreate_dist_matrix() to distribute the matrix to all the processes distributed by block rows. The right-hand side matrix in this example is a column vector of all ones. Note that, before setting the right-hand side, we use get_CompRowLoc_Matrix() to get the number of local rows in the distributed matrix A.

   One important note is that all the C routines use 0-based indexing scheme. Therefore, after process 0 reads the matrix in compressed column format, we decrement its column pointers (`colptr`*) and row indices (*`rowind`*) by 1 so they become 0-based indexing.*

4. Set the input arguments: options, ScalePermstruct, LUstruct, and stat. The input argument options controls how the linear system would be sloved. The routine f_set_default_options_dist() sets the options argument so that the slover performs all the functionalities. You can also set it according to your own needs, using a call to the Fortran routine set_superlu_options(). LUstruct is the data struture in which the distributed L and U factors are stored. ScalePermstruct is the data struture in which several vectors describing the transformations done to matrix A are stored. stat is a structure collecting the statistcs about runtime and flop count. These three structures can be set by calling the C wrapper *"init"* routines f_XXXInit.

5. Call the C wrapper routine f_pdgssvx() to solve the equation.

6. 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 a call to f_spuerlu_gridexit(). When all computations have been completed, the C wrapper routine mpi_finalize() should be called.

7. Deallocate all the structures. First we need to deallocate the storage allocated by `SuperLU_DIST` by a set of *"free"* calls. Note that this should be called before f_spuerlu_gridexit(), since some of the *"free"* calls use the grid. Then we call the C wrapper *"destroy"* routines f_destroy_XXX() to destroy all the Fortran handles. Note that f_destroy_gridinfo() should be called after f_spuerlu_gridexit().

4.10.1 Callable functions in the Fortran 90 module file spuerlu_mod.f90

The Fortran 90 module `superlu_mod` contains the interface routines that can manipulate a `SuperLU_DIST` object from Fortran. The object is pointed to by the corresponding handle input to these routines. The routines are divided into two sets. One set is to get the properties of an object, with the routine names "get_XXX()". Another set is to set some properties for an object, with the routine names "set_XXX()". These functions have optional arguments, so the users do not have to provide the full set of parameters. Superlu_mod module uses superluparam_mod module that defines all the integer constants corresponding to the enumeration constants in SuperLU_DIST. Below are the calling sequences of all the routines.

```
subroutine get_GridInfo(grid, iam, nprow, npcol)
  integer(superlu_ptr) :: grid
  integer, optional :: iam, nprow, npcol

subroutine get_SuperMatrix(A, nrow, ncol)
  integer(superlu_ptr) :: A
  integer, optional :: nrow, ncol

subroutine set_SuperMatrix(A, nrow, ncol)
  integer(superlu_ptr) :: A
  integer, optional :: nrow, ncol

subroutine get_CompRowLoc_Matrix(A, nrow, ncol, nnz_loc, nrow_loc, fst_row)
  integer(superlu_ptr) :: A
  integer, optional :: nrow, ncol, nnz_loc, nrow_loc, fst_row

subroutine set_CompRowLoc_Matrix(A, nrow, ncol, nnz_loc, nrow_loc, fst_row)
  integer(superlu_ptr) :: A
  integer, optional :: nrow, ncol, nnz_loc, nrow_loc, fst_row

subroutine get_superlu_options(opt, Fact, Trans, Equil, RowPerm, &
                               ColPerm, ReplaceTinyPivot, IterRefine, &
                               SolveInitialized, RefineInitialized)
integer(superlu_ptr) :: opt
  integer, optional :: Fact, Trans, Equil, RowPerm, ColPerm, &
                       ReplaceTinyPivot, IterRefine, SolveInitialized, &
                       RefineInitialized

subroutine set_superlu_options(opt, Fact, Trans, Equil, RowPerm, &
                               ColPerm, ReplaceTinyPivot, IterRefine, &
                               SolveInitialized, RefineInitialized)
  integer(superlu_ptr) :: opt
  integer, optional :: Fact, Trans, Equil, RowPerm, ColPerm, &
                       ReplaceTinyPivot, IterRefine, SolveInitialized, &
                       RefineInitialized
```

4.10.2 C wrapper functions callable by Fortran in file spuerlu_c2f_wrap.c

This file contains the Fortran-callable C functions which wraps around the user-callable C routines in SuperLU_DIST. The functions are divided into three classes: 1) allocate a C structure and return a handle to Fortran, or deallocate the memory of of a C structure given its Fortran handle; 2) get or set the value of certain fields of a C structure given its Fortran handle; 3) wrapper functions for the SuperLU_DIST C functions. Below are the calling sequences of these routines.

```
/* functions that allocate memory for a structure and return a handle */
void f_create_gridinfo(fptr *handle)
void f_create_options(fptr *handle)
void f_create_ScalePermstruct(fptr *handle)
void f_create_LUstruct(fptr *handle)
void f_create_SOLVEstruct(fptr *handle)
void f_create_SuperMatrix(fptr *handle)
void f_create_SuperLUStat(fptr *handle)

/* functions that free the memory allocated by the above functions */
void f_destroy_gridinfo(fptr *handle)
void f_destroy_options(fptr *handle)
void f_destroy_ScalePermstruct(fptr *handle)
void f_destroy_LUstruct(fptr *handle)
void f_destroy_SOLVEstruct(fptr *handle)
void f_destroy_SuperMatrix(fptr *handle)
void f_destroy_SuperLUStat(fptr *handle)

/* functions that get or set certain fields in a C structure. */
void f_get_gridinfo(fptr *grid, int *iam, int *nprow, int *npcol)
void f_get_SuperMatrix(fptr *A, int *nrow, int *ncol)
void f_set_SuperMatrix(fptr *A, int *nrow, int *ncol)
void f_get_CompRowLoc_Matrix(fptr *A, int *m, int *n, int *nnz_loc,
                                      int *m_loc, int *fst_row)
void f_set_CompRowLoc_Matrix(fptr *A, int *m, int *n, int *nnz_loc,
                                      int *m_loc, int *fst_row)
void f_get_superlu_options(fptr *opt, int *Fact, int *Trans, int *Equil,
                           int *RowPerm, int *ColPerm, int *ReplaceTinyPivot,
                           int *IterRefine, int *SolveInitialized,
                           int *RefineInitialized)
void f_set_superlu_options(fptr *opt, int *Fact, int *Trans, int *Equil,
                           int *RowPerm, int *ColPerm, int *ReplaceTinyPivot,
                           int *IterRefine, int *SolveInitialized,
                           int *RefineInitialized)

/* wrappers for SuperLU_DIST routines */
void f_dCreate_CompRowLoc_Matrix_dist(fptr *A, int *m, int *n, int *nnz_loc,
                                      int *m_loc, int *fst_row, double *nzval,
                                      int *colind, int *rowptr, int *stype,
                                      int *dtype, int *mtype)
void f_set_default_options(fptr *options)
void f_superlu_gridinit(int *Bcomm, int *nprow, int *npcol, fptr *grid)
void f_superlu_gridexit(fptr *grid)
void f_ScalePermstructInit(int *m, int *n, fptr *ScalePermstruct)
void f_ScalePermstructFree(fptr *ScalePermstruct)
```

```
void f_PStatInit(fptr *stat)
void f_PStatFree(fptr *stat)
void f_LUstructInit(int *m, int *n, fptr *LUstruct)
void f_LUstructFree(fptr *LUstruct)
void f_Destroy_LU(int *n, fptr *grid, fptr *LUstruct)
void f_Destroy_CompRowLoc_Matrix_dist(fptr *A)
void f_dSolveFinalize(fptr *options, fptr *SOLVEstruct)
void f_pdgssvx(fptr *options, fptr *A, fptr *ScalePermstruct, double *B,
               int *ldb, int *nrhs, fptr *grid, fptr *LUstruct,
               fptr *SOLVEstruct, double *berr, fptr *stat, int *info)
void f_check_malloc(int *iam)
```

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Release 2.0*. SIAM, Philadelphia, 1995. 324 pages.

[2] M. Arioli, J. W. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl.*, 10(2):165–190, April 1989.

[3] L. S. Blackford, J. Choi, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997. 325 pages.

[4] Timothy A. Davis, John R. Gilbert, Stefan I. Larimore, and Esmond Ng. A column approximate minimum degree ordering algorithm. Technical Report TR-00-005, Computer and Information Sciences Department, University of Florida, 2000. submitted to *ACM Trans. Math. Software*.

[5] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

[6] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[7] J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.

[8] J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[9] Iain S. Duff and Jacko Koster. 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, 1999.

[10] I.S. Duff, R.G. Grimes, and J.G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (release 1). Technical Report RAL-92-086, Rutherford Appleton Laboratory, December 1992.

[11] Alan George, Joseph Liu, and Esmond Ng. A data structure for sparse QR and LU factorizations. *SIAM J. Sci. Stat. Comput.*, 9:100–121, 1988.

[12] Alan George and Esmond Ng. Symbolic factorization for sparse Gaussian elimination with partial pivoting. *SIAM J. Sci. Stat. Comput.*, 8(6):877–898, 1987.

[13] J. R. Gilbert, X. S. Li, E. G. Ng, and B. W. Peyton. Computing row and column counts for sparse QR and LU factorization. *BIT*, 41(4):693–710, 2001.

[14] J. R. Gilbert and E. G. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, pages 107–139. Springer-Verlag, 1993.

[15] B. Hendrickson and R. Leland. The CHACO User's Guide. Version 1.0. Technical Report SAND93-2339 • UC-405, Sandia National Laboratories, Albuquerque, 1993. http://www.cs.sandia.gov/~bahendr/chaco.html.

[16] N. J. Higham. Algorithm 674: FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation. *ACM Trans. Math. Soft.*, 14:381–396, 1988.

[17] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, 1996.

[18] G. Karypis and V. Kumar. METIS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0.* University of Minnesota, September 1998. http://www-users.cs.umn.edu/~karypis/metis.

[19] George Karypis and Vipin Kumar. Serial and parallel software packages for partitioning unstructured graphs and for computing fill-reducing orderings of sparse matrices. AHPCRC, University of Minnesota. http://www.arc.umn.edu/software/.

[20] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[21] Xiaoye S. Li. Sparse Gaussian elimination on high performance computers. Technical Report UCB//CSD-96-919, Computer Science Division, U.C. Berkeley, September 1996. Ph.D dissertation.

[22] Xiaoye S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. Technical Report LBNL-53848, Lawrence Berkeley National Laboratory, September 2003. http://crd.lbl.gov/~xiaoye/LBNL-53848.ps.gz.

[23] Xiaoye S. Li and James W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of SC98: High Performance Networking and Computing Conference*, Orlando, Florida, November 7–13 1998.

[24] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

[25] Joseph W.H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11:141–153, 1985.

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

[27] W. Oettli and W. Prager. Compatibility of approximate solution of linear equations with given error bounds for coefficients and right hand sides. *Num. Math.*, 6:405–409, 1964.

[28] Francois Pellegrini. SCOTCH 3.4 User's Guide. Tech report 1264-01, LaBRI, URM CNRS 5800, University Bordeaux I, France, November 2001. http://www.labri.fr/~pelegrin/scotch.

[29] POSIX System Application Arogram Interface: Threads extension [C Language], POSIX 1003.1c draft 4. IEEE Standards Department.

[30] R.D. Skeel. Iterative refinement implies numerical stability for Gaussian elimination. *Mathematics of Computation*, 35(151):817–832, 1980.