LBNL-44289

ERNEST ORLANDO LAWRENCE
BERKELEY NATIONAL LABORATORY

SuperLU Users' Guide

James W. Demmel, John R. Gilbert,
and Xiaoye S. Li

National Energy Research
Scientific Computing Division

RECEIVED
MAR 24 2000
OSTI

September 1999

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor The Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or The Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, or The Regents of the University of California.

Ernest Orlando Lawrence Berkeley National Laboratory
is an equal opportunity employer.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

LBNL-44289

SuperLU Users' Guide

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

[1]Computer Science Division
University of California
Berkeley, California 94720

[2]Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

[3]National Energy Research Scientific Computing Center
Ernest Orlando Lawrence Berkeley National Laboratory
University of California
Berkeley, California 94720

September 1999

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, Mathematical, Information, and Computational Sciences Division, DOE Grants DE-FG03-94ER25219 and DE-FG03-94ER25206, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098, and by National Science Foundation Grant ASC-9313958, Infrastructure Grants CDA-8722788 and CDA-9401156, DARPA Contract No. DABT63-95-C0087 and ARPA Contract No. DAAL03-91-C0047.

Contents

1 Introduction **1**

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

2 Sequential SuperLU **11**

2.1 About SuperLU 11
2.2 How to call a SuperLU routine 11
2.3 Matrix Data Structures 15
2.4 Permutations 18
2.4.1 Ordering for sparsity 18
2.4.2 Partial pivoting with threshold 18
2.5 Memory management for L and U 19
2.6 User-callable routines 20
2.6.1 Driver routines 20
2.6.2 Computational routines 20

2.7 Matlab interface 21
2.8 Installation 23
2.8.1 File structure 23
2.8.2 Testing 24
2.8.3 Performance-tuning parameters 25
2.9 Example programs 26
2.9.1 Repeated factorizations 26
2.9.2 Calling from Fortran 28

3 Multithreaded SuperLU **30**
3.1 About SuperLU_MT 30
3.2 Storage types for L and U 30
3.3 User-callable routines 31
3.3.1 Driver routines 32
3.3.2 Computational routines 32
3.4 Installation 33
3.4.1 File structure 33
3.4.2 Performance issues 33
3.5 Example programs 36
3.6 Porting to other platforms 36
3.6.1 Creating multiple threads 36
3.6.2 Use of mutexes 37

4 Distributed SuperLU with MPI **38**
4.1 About SuperLU_DIST 38
4.2 Basic steps to solve a linear system 38
4.3 Process grid and MPI communicator 43
4.3.1 SuperLU 2-D grid 43
4.3.2 Arbitrary grounding of processes 44
4.4 Matrix distribution and distributed data structures for L and U 44
4.5 Algorithmic background 45
4.6 User-callable routines 46
4.6.1 Driver routines 46
4.6.2 Computational routines 48
4.7 Installation 49
4.7.1 File structure 49
4.7.2 Performance-tuning parameters 40
4.8 Example programs 50

Bibliography **51**

A Specifications of routines in sequential SuperLU **53**
A.1 dgsequ 53
A.2 dgscon 54
A.3 dgsrfs 55
A.4 dgssv 57

A.5 dgssvx 60
A.6 dgstrf 67
A.7 dgstrs 70
A.8 dlaqgs 71

B Specifications of routines in multithreaded SuperLU_MT **73**
B.1 pdgssv 73
B.2 pdgssvx 75
B.3 pdgstrf 83

C Specifications of routines in MPI-based SuperLU_DIST **87**
C.1 pdgssvx_ABglobal 87
C.2 pdgstrf 96
C.3 pdgstrs_Bglobal 98
C.4 pdgsrfs_ABXglobal 99

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 [19]).

- **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 in order to speed up the computation [20].

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. Section 1.7 describes the organization of the rest of the document.

1.2 Overall Algorithm

A simple description of sparse Gaussian elimination is as follows:

1. Compute a *triangular factorization* $P_r A P_c = LU$. Here P_r and P_c are *permutation matrices*. Premultiplying by P_r reorders the rows of A, and postmultiplying 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 lower triangular matrix and U is an upper triangular matrix. Typically L is a unit triangular matrix, i.e. $L_{ii} = 1$.

2. Solve $AX = B$ by evaluating $X = A^{-1}B = (P_c^{-1}LUP_r^{-1})^{-1}B = P_c(U^{-1}(L^{-1}(P_rB)))$. This is done efficiently by multiplying from right to left in the last expression: Multiplying P_rB means permuting the rows of B. Multiplying $L^{-1}(P_rB)$ means solving *nrhs* triangular systems of equations with matrix L by substitution. Similarly, multiplying $U^{-1}(L^{-1}(P_rB))$ means solving triangular systems with U. Finally, multiplying $P_c(U^{-1}(L^{-1}(P_rB)))$ is again permutation.

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.

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 $\hat{A} = D_rAD_c$ 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_ABglobal` 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 the current version of SuperLU_DIST, A and B are replicated across all processors; in a future version they will be distributed.)

(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 [18, 7, 8] 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 2.8.3 for SuperLU, in section 3.4.2 for SuperLU_MT, and in section 4.7.2 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

The expert driver in all three libraries returns a struct with 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 in the course of the computation. A variable `SuperLUStat` 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 operation count in various phases */
} 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 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 `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 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 either in the simple or expert driver, in section 1.2:

- Natural ordering,
- Multiple Minimum Degree (MMD) [22] applied to the structure of $A^T A$,
- Multiple Minimum Degree (MMD) [22] 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, 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 [17]. 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 $A + A^T$. Our routines `getata` and `a_plus_at` in the file `get_perm_c.c` can be used to form $A^T A$ or $A + A^T$.

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, 23] 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, 25] 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, 16] 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 [20].)

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 [13]) 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

All three drivers, and their computational routines, may be called by C or Fortran.

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 section 4.3, we use a 2-D 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 4.4 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 2-D 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 4.3 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 [20]. 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 2.7 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, 19]. 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, 19]. 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 [20, 21].

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.

1. Some subroutines carry the following notice:

   > Copyright (c) 1994 by Xerox Corporation. All rights reserved.
   > THIS MATERIAL IS PROVIDED AS IS, WITH ABSOLUTELY NO WARRANTY EXPRESSED OR IMPLIED. ANY USE IS AT YOUR OWN RISK.
   > Permission is hereby granted to use or copy this program for any purpose, provided the above notices are retained on all copies. Permission to modify the code and to distribute modified code is granted, provided the above notices are retained, and a notice that the code was modified is included with the above copyright notice.

2. The MC64 package carries the following notice:

   > COPYRIGHT (c) 1999 Council for the Central Laboratory of the Research Councils. All rights reserved. PACKAGE MC64A/AD AUTHORS Iain Duff (i.duff@rl.ac.uk) and Jacko Koster (jak@ii.uib.no) LAST UPDATE 20/09/99
   > *** Conditions on external use ***
   > The user shall acknowledge the contribution of this package in any publication of material dependent upon the use of the package. The user shall use reasonable endeavours to notify the authors of the package of this publication.
   > 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 Netlib through the URL address:

```
http://www.netlib.org/scalapack/prototype/
```

They are also available on the FTP server at UC Berkeley:

```
ftp ftp.cs.berkeley.edu
login: anonymous
ftp> cd /pub/src/lapack/SuperLU
ftp> binary
ftp> get superlu_2.0.tar.gz
```

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 other queries should be e-mailed to `xiaoye@nersc.gov` and `demmel@cs.berkeley.edu`.

1.7 Document organization

The rest of this document is organized as follows. Chapter 2 describes Sequential SuperLU. Chapter 3 describes SuperLU_MT. Chapter 4 describes SuperLU_DIST. Finally, the calling sequence and the leading comment of the user-callable routines for all three libraries are listed in the appentices.

1.8 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 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

2.1 About SuperLU

In this chapter, SuperLU will always mean Sequential SuperLU. The 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 matrices A and B, respectively, in the data structures internally used by SuperLU. The routine `get_perm_c` is called to generate a column permutation vector, stored in `perm_c[]`. A good column permutation should make the L and U factors as sparse as possible. The user can supply `perm_c[]` instead of using the one provided by SuperLU. 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"
#include "util.h"

main(int argc, char *argv[])
{
    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;

    /* 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, NC, _D, 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, DN, _D, GE);

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

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

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

    dPrint_CompCol_Matrix("A", &A);
    dPrint_CompCol_Matrix("U", &U);
    dPrint_SuperNode_Matrix("L", &L);
    PrintInt10("\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);
}
```

```
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 {
    NC,        /* column-wise, not supernodal */
    NR,        /* row-wise, not supernodal */
    SC,        /* column-wise, supernodal */
    SR,        /* row-wise, supernodal */
    NCP,       /* column-wise, not supernodal, permuted by columns
                  (After column permutation, the consecutive columns of
                   nonzeros may not be stored contiguously. */
    DN         /* Fortran style column-wise storage for dense matrix */
} Stype_t;

typedef enum {
    _S,         /* single */
    _D,         /* double */
    _C,         /* single-complex */
    _Z          /* double-complex */
} Dtype_t;

typedef enum {
    GE,        /* general */
    TRLU,      /* lower triangular, unit diagonal */
    TRUU,      /* upper triangular, unit diagonal */
    TRL,       /* lower triangular */
    TRU,       /* upper triangular */
    SYL,       /* symmetric, store lower half */
    SYU,       /* symmetric, store upper half */
    HEL,       /* Hermitian, store lower half */
    HEU        /* Hermitian, store upper half */
} Mtype_t;
```

Figure 2.2: SuperMatrix data structure.

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 presents 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	NC or NR	SC	NC	DN	DN
Dtype[1]	any	any	any	any	any
Mtype	GE	TRLU	TRU	GE	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 `NC` or `NR`. The `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 `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[]
                     which starts row j. It has nrow+1 entries,
                     and rowptr[nrow] = nnz. */
} NRformat;
```

[1] `Dtype` can be one of _S, _D, _C or _Z.

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 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 routines dgssv and dgssvx in Appendix A.

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

```
void sCompRow_to_CompCol(int m, int n, int nnz,
                         float *a, int *colind, int *rowptr,
                         float **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 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 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 2.4.1.

```
• A = { Stype = NC; Dtype = _D; Mtype = 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 = NC; Dtype = _D; Mtype = 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 = SC; Dtype = _D; Mtype = 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.

2.4 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.

2.4.1 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 subroutine `get_perm_c`.

```
void get_perm_c(int ispec, SuperMatrix *A, int *perm_c);
```

`Ispec` specifies the ordering to be returned in `*perm_c`, the integer vector representing the permutation matrix P_c:

ispec = 0: natural ordering (i.e., $P_c = I$)
= 1: MMD applied to the structure of $A^T A$
= 2: MMD applied to the structure of $A + A^T$
= 3: COLAMD, approximate minimum degree column ordering

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 `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;
```

2.4.2 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 `*refact = 'Y'` 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.

2.5 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 2.8.3.) 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.

2.6 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. Appendix A contains, for each individual routine, the leading comments and the complete specification of the calling sequence and arguments.

2.6.1 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.

2.6.2 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 [15], and is the same as CONDEST in sparse Matlab.

- dgsequ/xlaqgs: 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.

2.7 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.

2.8 Installation

2.8.1 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/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 top-level `SuperLU/` directory 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.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 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.

2.8.2 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.

[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.

[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.

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 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].

2.8.3 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 ($maxsup$)

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.

= 4: the minimum row dimension for 2-D blocking to be used (*rowblk*)
= 5: the minimum column dimension for 2-D blocking to be used (*colblk*)
= 6: the estimated fills factor for L and U, compared with A

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 updating supernode is too large to fit in cache, then a 2-D 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 2-D blocking. For example, for the Cray J90 (which does not have cache), $w = 1$ and 1-D blocking give good performance; more levels of blocking only increase overhead.

2.9 Example programs

In the `SuperLU/EXAMPLE/` subdirectory, we present a few sample programs, such as `xLINSOL` and `xLINSOLX`, to illustrate the complete calling sequences used to solve systems of equations. These include how to set up the matrix structures, how to obtain a fill-reducing ordering, and how to call driver routines. A `Makefile` is provided to generate the executables. A `README` file in this directory shows how to run these examples.

Based on these sample programs, we now illustrate how we may use SuperLU in some other ways.

2.9.1 Repeated factorizations

In many iterative processes, matrices with the same sparsity pattern but different numerical values must be factored repeatedly. Thus, computing a fill-reducing ordering and performing column permutation are needed only once. In addition, the memory for L and U can be allocated only once, and reused in the subsequent factorizations. If there is not enough space for L and U from the

```
main()
{
    /* Declare variables */
    SuperMatrix A;  /* original matrix */
    SuperMatrix AC; /* A postmultiplied by a permutation matrix Q */
    char   refact[1];
    ...... /* declarations of other variables */

    /* Initialization */
    {
        StatInit(panel_size, relax);
        ......
    }

    /* First-time factorization */
    *refact = 'N';

    /* Obtain and apply column permutation */
    get_perm_c(1, &A, perm_c);
    sp_preorder(refact, &A, perm_c, etree, &AC);

    /* Factorization */
    dgstrf(refact, &AC, 1.0, 0.0, relax, panel_size,
           etree, NULL, 0, perm_r, &L, &U, &info);
    ...... /* solve first system */

    /* Subsequent factorizations */
    *refact = 'Y';

    for ( i = 1; i <= niter; ++i ) {
        dgstrf(refact, &AC, 1.0, 0.0, relax, panel_size,
               etree, NULL, 0, perm_r, &L, &U, &info);
        /* Numerical values of matrix AC may change across iterations.
           The factors L and U are overwritten in each iteration. */
        {
               ...... /* solve later system */
        }
    }

    StatFree();

}
```

Figure 2.4: Code segment to perform repeated factorizations.

previous factorization (due to different pivoting), the factor routines `xGSTRF` automatically expand memory as needed. Figure 2.4 shows the code segment for this purpose.

2.9.2 Calling from Fortran

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 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 supply "bridge" routines (in C) for all the SuperLU subroutines that will be called from Fortran programs. Figure 2.5 is an example showing how a bridge program should be written. See the files `f77_main.f` and `c_bridge_dgssv.c` for complete descriptions.

In the future, we may provide complete Fortran interfaces to the user-callable routines in SuperLU.

Fortran program (f77_main.f)

```
    program f77_main
    integer maxn, maxnz
    parameter ( maxn = 10000, maxnz = 100000 )
    integer rowind(maxnz), colptr(maxn)
    real*8  values(maxnz), b(maxn)
    ......

    call c_bridge_dgssv( n, nnz, nrhs, values, rowind, colptr, b, ldb, info )
    ......

    stop
    end
```

The bridge program in C (c_bridge_dgssv.c)

```
    int c_bridge_dgssv(int *n, int *nnz, int *nrhs, double *values, int *rowind,
                       int *colptr, double *b, int *ldb, int *info)
    {
        SuperMatrix A, B, L, U;
        int *perm_c, *perm_r;
        ......

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

        /* Construct Matrix structures A and B */
        dCreate_CompCol_Matrix(&A, *n, *n, *nnz, values, rowind, colptr,
                               NC, _D, GE);
        dCreate_Dense_Matrix(&B, *n, *nrhs, b, *ldb, DN, _D, GE);
        ......

        /* B is overwritten by the solution vector */
        dgssv(&A, perm_c, perm_r, &L, &U, &B, info);
        ......
    }
```

Figure 2.5: Interface with Fortran

Chapter 3

Multithreaded SuperLU

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, 19]. 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	NC or NR	SCP	NCP	DN	DN
Dtype	_D	_D	_D	_D	_D
Mtype	GE	TRLU	TRU	GE	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. Appendix B contains, for each individual routine, the leading

comments and the complete specification of the calling sequence and arguments. 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 [15], 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 2.8 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 2.5. 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, 19]. 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 [14].

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 2-D blocking to be used (*rowblk*)
= 5: the minimum column dimension for 2-D 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 2.8.3. 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 [24], 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

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 Basic steps to solve a linear system

In this section, we use a complete sample program to illustrate the basic steps required to use the MPI version of the SuperLU library. 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_ABglobal 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_ABglobal
 *   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;
    gridinfo_t grid;
    double   *berr;
    double   *a, *b, *xtrue;
    int_t    *asub, *xa;
    int_t    i, 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. */

    /* 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")) ) {
                    fprintf(stderr, "File does not exist.");
            exit(-1);
        }
        break;
    }
}

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

/* ------------------------------------------------------------
   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;

/* ------------------------------------------------------------
   PROCESS 0 READS THE MATRIX A, AND THEN BROADCASTS IT TO ALL
   THE OTHER PROCESSES.
   ------------------------------------------------------------*/
if ( !iam ) {
    /* Read the matrix stored on disk in Harwell-Boeing format. */
    dreadhb(iam, fp, &m, &n, &nnz, &a, &asub, &xa);

    printf("\tDimension\t%dx%d\t # nonzeros %d\n", m, n, nnz);
    printf("\tProcess grid\t%d X %d\n", grid.nprow, grid.npcol);

    /* Broadcast matrix A to the other PEs. */
    MPI_Bcast( &m,   1,   mpi_int_t,  0, grid.comm );
    MPI_Bcast( &n,   1,   mpi_int_t,  0, grid.comm );
    MPI_Bcast( &nnz, 1,   mpi_int_t,  0, grid.comm );
    MPI_Bcast( a,    nnz, MPI_DOUBLE, 0, grid.comm );
    MPI_Bcast( asub, nnz, mpi_int_t,  0, grid.comm );
    MPI_Bcast( xa,   n+1, mpi_int_t,  0, grid.comm );
} else {
    /* Receive matrix A from PE 0. */
    MPI_Bcast( &m,   1,   mpi_int_t,  0, grid.comm );
    MPI_Bcast( &n,   1,   mpi_int_t,  0, grid.comm );
    MPI_Bcast( &nnz, 1,   mpi_int_t,  0, grid.comm );
```

```
    /* Allocate storage for compressed column representation. */
    dallocateA(n, nnz, &a, &asub, &xa);

    MPI_Bcast( a,    nnz, MPI_DOUBLE, 0, grid.comm );
    MPI_Bcast( asub, nnz, mpi_int_t,  0, grid.comm );
    MPI_Bcast( xa,   n+1, mpi_int_t,  0, grid.comm );
}

/* Create compressed column matrix for A. */
dCreate_CompCol_Matrix(&A, m, n, nnz, a, asub, xa, NC, _D, GE);

/* Generate the exact solution and compute the right-hand side. */
if ( !(b = doubleMalloc(m * nrhs)) ) ABORT("Malloc fails for b[]");
if ( !(xtrue = doubleMalloc(n * nrhs)) ) ABORT("Malloc fails for xtrue[]");
*trans = 'N';
ldx = n;
ldb = m;
dGenXtrue(n, nrhs, xtrue, ldx);
dFillRHS(trans, nrhs, xtrue, ldx, &A, b, ldb);

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

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

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

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

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

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

/* Check the accuracy of the solution. */
if ( !iam ) dinf_norm_error(n, nrhs, b, ldb, xtrue, ldx);

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

/* ------------------------------------------------------------
```

```
      DEALLOCATE STORAGE.
      ------------------------------------------------------------*/
   PStatFree(&stat);
   Destroy_CompCol_Matrix(&A);
   Destroy_LU(n, &grid, &LUstruct);
   ScalePermstructFree(&ScalePermstruct);
   LUstructFree(&LUstruct);
   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 4.3 contains the details about this step.

2. Set up the input matrix and the right-hand side.
   In this example, process 0 reads the input matrix stored on disk in Harwell-Boeing format [10], and broadcasts it to all the other processes. The right-hand side matrix is generated so that the exact solution matrix consists of all ones. Currently the library requires the input matrix and the right-hand side are available on every process. In the future, we will allow these two matrices being distributed on input.

3. Initialize the input arguments: `options, Astruct, 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` 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 4.6.1 for the fields of this structure. `Astruct` is the data structure in which matrix A of the linear system and several vectors describing the transformations done to A are stored. `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.

4. Call the SuperLU routine `pdgssvx_ABglobal`.

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.

4.3 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 $0 \ldots N_p - 1$.

4.3.1 SuperLU 2-D grid

For SuperLU 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 1-D array of N_g processes into a logical 2-D process grid. This grid will have nprow process rows and npcol process columns, such that nprow $*$ 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 2-D process grid.

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

This process grid will use the first nprow $*$ 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 the routines of SuperLU library. 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 2-D 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.*

4.3.2 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. So 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 2-D 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 2-D 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.

4.4 Matrix distribution and 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.paragraph The matrix in Figure 4.1 illustrates such a partition. By block-cyclic mapping we mean block (I, J) $(0 \leq I, J \leq N-1)$ is mapped into the process at coordinate $\{I \; mod$ `nprow`$, J \; mod$ `npcol`$\}$ of the `nprow` $\times$ `npcol` process grid. Using this mapping, a block $L(I, J)$ in the factorization is only needed by the row of processes that own blocks in row I. Similarly, a block $U(I, J)$ is only needed by the column of processes that own blocks in column J.

In this 2-D mapping, each block column of L resides on more than one process, namely, a column of processes. For example in Figure 4.1, the k-th block column of L resides on the column processes {0, 3}. Process 3 only owns two nonzero blocks, which are not contiguous in the global matrix. The schema on the right of Figure 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

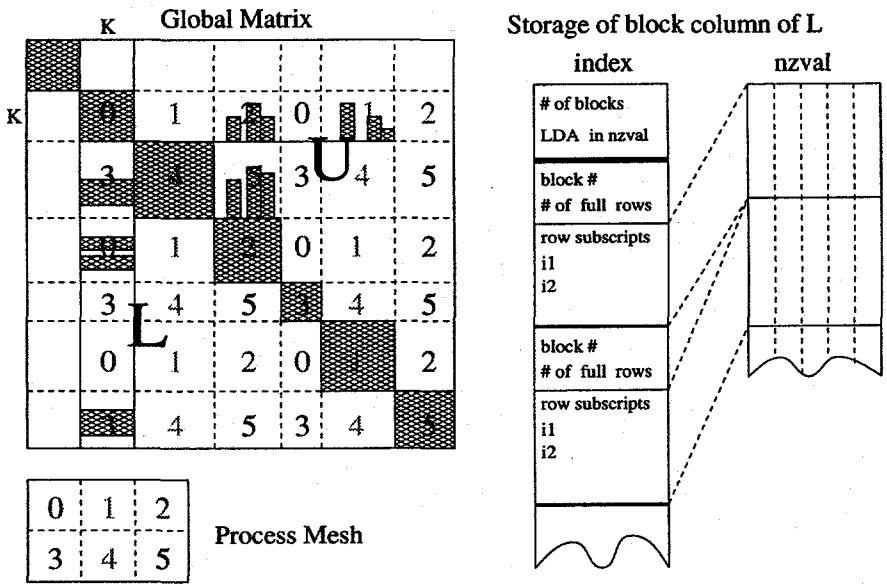

Figure 4.1: The 2-D block-cyclic layout and the data structure to store a local block column of L.

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.

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. Similarly to L, we also use a pair of `index/nzval` arrays to store a block row of U. Due to asymmetry, each nonzero block in U has the skyline structure as shown in Figure 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.

Since currently some steps of the algorithm (steps (1) to (3) in Figure 4.2) are not yet parallel, we start with a copy of the entire matrix A on each process. The routine `symbfact` determines the nonzero patterns of L and U as well as the block partition. The routine `ddistribute` uses this information to sets up the L and U data structures and load the initial values of A into L and U.

4.5 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 parallelize. We use alternative techniques to stabilize the algorithm, suas as statically pivot large elements to the diagonal, half-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 [20].

We have parallelized the two most time-consuming steps in this algorithm, which are Step (4) and Step (5). Currently, process 0 in the logical process grid computes D_r and D_c and broadcasts them to all the other processes, which in turn just apply them to A. Step (2) is accomplished by

(1) Row/column equilibration: $A \leftarrow D_r \cdot A \cdot D_c$
D_r and D_c are diagonal matrices chosen so that the largest entry of each row and column is ± 1.
(2) Row permutation: $A \leftarrow P_r \cdot A$
P_r is a row permutation chosen to make the diagonal large compared to the off-diagonal.
(3) Find a column permutation P_c to preserve sparsity: $A \leftarrow P_c \cdot A \cdot P_c^T$
(4) Factorize $A = L \cdot U$ with control of diagonal magnitude
if ($|a_{ii}| < \sqrt{\varepsilon} \cdot ||A||$) **then**
set a_{ii} to $\sqrt{\varepsilon} \cdot ||A||$
endif
(5) Solve $A \cdot x = b$ using the L and U factors, with the following iterative refinement
iterate:
$r = b - A \cdot x$ ... sparse matrix-vector multiply
Solve $A \cdot dx = r$ ... triangular solution
$berr = \max_i \frac{|r|_i}{(|A| \cdot |x| + |b|)_i}$ ... componentwise backward error
if ($berr > \varepsilon$ and $berr \leq \frac{1}{2} \cdot lastberr$) **then**
$x = x + dx$
$lastberr = berr$
goto iterate
endif

Figure 4.2: The outline of the GESP algorithm.

a weighted bipartite matching algorithm due to Duff and Koster [9]. Again, process 0 computes P_r and then broadcasts it to all the other processes. For Step (3), we provide several ordering options, such as multiple minimum degree ordering [22] on the graphs of $A + A^T$ or $A^T A$, and the approximate minimum degree column ordering [4]. 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 $A + A^T$ tends to yield 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. After the above sequential setup, we perform parallel factorization, parallel triangular solutions and parallel iterative refinement.

4.6 User-callable routines

Appendix C contains the complete specifications of the routines in SuperLU_DIST.

4.6.1 Driver routine

There is one driver routine to solve systems of linear equations, which is named `pdgssvx_ABglobal`. We recommend that the general users, especially the beginners, use this driver routine rather than the computational routines, because correctly using this routine does not require thorough understanding of the underlying data structures. Although the interface of this routine is simple, we expect its rich functionality can meet the requirements of most applications. `Pdgssvx_ABglobal` performs 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` is `options`, which controls how the linear system will be solved. Although the algorithm presented in Figure 4.2 consists of five steps, for some matrices not all five 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 (2). 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 five 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.
- `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.

- `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.
- `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.*)

There is a routine named `set_default_options` that sets the default values of these options, which are:

```
fact             = DOFACT
equil            = YES
rowperm          = LargeDiag
colperm          = MMD_AT_PLUS_A
ReplaceTinyPivot = YES
IterRefine       = DOUBLE
```

4.6.2 Computational routines

The experienced users can invoke the following computational routines to directly control the behavior of SuperLU 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 `ddistribute` should be called to distribute the matrix. `Pdgstrf` can factor non-square matrices.

  Currently, A must be globally available on all processes.
- `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`.

  Currently, B must be globally available on all processes.

- `pdgsrfs_ABXglobal`: Refine solution in parallel.
  Given A, its factors L and U, and an initial solution X, this routine performs iterative refinement.

  Currently, A, B, and X must be globally available on all processes.

4.7 Installation

4.7.1 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  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 top-level `SuperLU_DIST` directory for several systems, such as Cray T3E and IBM SP. When you have selected the machine to which you wish to install SuperLU_DIST, 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.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.

4.7.2 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.

4.8 Example programs

In the `SuperLU_DIST/EXAMPLE/` subdirectory, we present a few sample programs, such as `pddrive`, 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 the input matrix, how to obtain a fill-reducing ordering. 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.

[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.

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. manuscript, in preparation.

[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. Technical Report UCB//CSD-95-883, Computer Science Division, U.C. Berkeley, 1995. To appear in *SIAM J. Matrix Anal. Appl.*

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

[7] 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.

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

[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 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.

[14] John R. Gilbert, Xiaoye S. Li, Esmond G. Ng, and Barry W. Peyton. Computing row and column counts for sparse QR factorization. manuscript, in preparation.

[15] 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.

[16] N. J. Higham. *Accuracy and stability of numerical algorithms.* SIAM, 1996.

[17] 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/.

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

[19] 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.

[20] Xiaoye S. Li and James W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of SC98*, Orlando, Florida, November 1998.

[21] Xiaoye S. Li and James W. Demmel. A scalable sparse direct solver using static pivoting. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 22–24 1999.

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

[23] 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.

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

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

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

Appendix A

Specifications of routines in sequential SuperLU

A.1 dgsequ

```
void
dgsequ(SuperMatrix *A, double *r, double *c, double *rowcnd,
       double *colcnd, double *amax, int *info)

    Purpose
    =======

    DGSEQU computes row and column scalings intended to equilibrate an
    M-by-N sparse matrix A and reduce its condition number. R returns the row
    scale factors and C the column scale factors, chosen to try to make
    the largest element in each row and column of the matrix B with
    elements B(i,j)=R(i)*A(i,j)*C(j) have absolute value 1.

    R(i) and C(j) are restricted to be between SMLNUM = smallest safe
    number and BIGNUM = largest safe number.  Use of these scaling
    factors is not guaranteed to reduce the condition number of A but
    works well in practice.

    See supermatrix.h for the definition of 'SuperMatrix' structure.

    Arguments
    =========

    A       (input) SuperMatrix*
            The matrix of dimension (A->nrow, A->ncol) whose equilibration
            factors are to be computed. The type of A can be:
            Stype = NC; Dtype = _D; Mtype = GE.

    R       (output) double*, size A->nrow
            If INFO = 0 or INFO > M, R contains the row scale factors
```

```
        for A.

C       (output) double*, size A->ncol
        If INFO = 0,  C contains the column scale factors for A.

rowcnd  (output) double*
        If INFO = 0 or INFO > M, ROWCND contains the ratio of the
        smallest R(i) to the largest R(i).  If ROWCND >= 0.1 and
        AMAX is neither too large nor too small, it is not worth
        scaling by R.

colcnd  (output) double*
        If INFO = 0, COLCND contains the ratio of the smallest
        C(i) to the largest C(i).  If COLCND >= 0.1, it is not
        worth scaling by C.

amax    (output) double*
        Absolute value of largest matrix element.  If AMAX is very
        close to overflow or very close to underflow, the matrix
        should be scaled.

info    (output) int*
        = 0:  successful exit
        < 0:  if info = -i, the i-th argument had an illegal value
        > 0:  if info = i,  and i is
              <= A->nrow:  the i-th row of A is exactly zero
              >  A->ncol:  the (i-M)-th column of A is exactly zero
```

A.2 dgscon

```
void
dgscon(char *norm, SuperMatrix *L, SuperMatrix *U,
       double anorm, double *rcond, int *info)

    Purpose
    =======

    DGSCON estimates the reciprocal of the condition number of a general
    real matrix A, in either the 1-norm or the infinity-norm, using
    the LU factorization computed by DGETRF.

    An estimate is obtained for norm(inv(A)), and the reciprocal of the
    condition number is computed as
       RCOND = 1 / ( norm(A) * norm(inv(A)) ).

    See supermatrix.h for the definition of 'SuperMatrix' structure.
```

```
Arguments
=========

norm    (input) char*
        Specifies whether the 1-norm condition number or the
        infinity-norm condition number is required:
        = '1' or 'O':  1-norm;
        = 'I':         Infinity-norm.

L       (input) SuperMatrix*
        The factor L from the factorization Pr*A*Pc=L*U as computed by
        dgstrf(). Use compressed row subscripts storage for supernodes,
        i.e., L has types: Stype = SC, Dtype = _D, Mtype = TRLU.

U       (input) SuperMatrix*
        The factor U from the factorization Pr*A*Pc=L*U as computed by
        dgstrf(). Use column-wise storage scheme, i.e., U has types:
        Stype = NC, Dtype = _D, Mtype = TRU.

anorm   (input) double
        If NORM = '1' or 'O', the 1-norm of the original matrix A.
        If NORM = 'I', the infinity-norm of the original matrix A.

rcond   (output) double*
        The reciprocal of the condition number of the matrix A,
        computed as RCOND = 1/(norm(A) * norm(inv(A))).

info    (output) int*
        = 0:  successful exit
        < 0:  if INFO = -i, the i-th argument had an illegal value
```

A.3 dgsrfs

```
void
dgsrfs(char *trans, SuperMatrix *A, SuperMatrix *L, SuperMatrix *U,
       int *perm_r, int *perm_c, char *equed, double *R, double *C,
       SuperMatrix *B, SuperMatrix *X,
       double *ferr, double *berr, int *info)

Purpose
=======

DGSRFS improves the computed solution to a system of linear
equations and provides error bounds and backward error estimates for
the solution.
```

```
If equilibration was performed, the system becomes:
     (diag(R)*A_original*diag(C)) * X = diag(R)*B_original.

See supermatrix.h for the definition of 'SuperMatrix' structure.

Arguments
=========

trans   (input) char*
        Specifies the form of the system of equations:
        = 'N':  A * X = B     (No transpose)
        = 'T':  A**T * X = B  (Transpose)
        = 'C':  A**H * X = B  (Conjugate transpose = Transpose)

A       (input) SuperMatrix*
        The original matrix A in the system, or the scaled A if
        equilibration was done. The type of A can be:
        Stype = NC, Dtype = _D, Mtype = GE.

L       (input) SuperMatrix*
        The factor L from the factorization Pr*A*Pc=L*U. Use
        compressed row subscripts storage for supernodes,
        i.e., L has types: Stype = SC, Dtype = _D, Mtype = TRLU.

U       (input) SuperMatrix*
        The factor U from the factorization Pr*A*Pc=L*U as computed by
        dgstrf(). Use column-wise storage scheme,
        i.e., U has types: Stype = NC, Dtype = _D, Mtype = TRU.

perm_r  (input) int*, dimension (A->nrow)
        Row permutation vector, which defines the permutation matrix Pr;
        perm_r[i] = j means row i of A is in position j in Pr*A.

perm_c  (input) int*, dimension (A->ncol)
        Column permutation vector, which defines the
        permutation matrix Pc; perm_c[i] = j means column i of A is
        in position j in A*Pc.

equed   (input) Specifies the form of equilibration that was done.
        = 'N': No equilibration.
        = 'R': Row equilibration, i.e., A was premultiplied by diag(R).
        = 'C': Column equilibration, i.e., A was postmultiplied by
               diag(C).
        = 'B': Both row and column equilibration, i.e., A was replaced
               by diag(R)*A*diag(C).
```

```
R       (input) double*, dimension (A->nrow)
        The row scale factors for A.
        If equed = 'R' or 'B', A is premultiplied by diag(R).
        If equed = 'N' or 'C', R is not accessed.

C       (input) double*, dimension (A->ncol)
        The column scale factors for A.
        If equed = 'C' or 'B', A is postmultiplied by diag(C).
        If equed = 'N' or 'R', C is not accessed.

B       (input) SuperMatrix*
        B has types: Stype = DN, Dtype = _D, Mtype = GE.
        The right hand side matrix B.
        if equed = 'R' or 'B', B is premultiplied by diag(R).

X       (input/output) SuperMatrix*
        X has types: Stype = DN, Dtype = _D, Mtype = GE.
        On entry, the solution matrix X, as computed by dgstrs().
        On exit, the improved solution matrix X.
        if *equed = 'C' or 'B', X should be premultiplied by diag(C)
            in order to obtain the solution to the original system.

FERR    (output) double*, dimension (B->ncol)
        The estimated forward error bound for each solution vector
        X(j) (the j-th column of the solution matrix X).
        If XTRUE is the true solution corresponding to X(j), FERR(j)
        is an estimated upper bound for the magnitude of the largest
        element in (X(j) - XTRUE) divided by the magnitude of the
        largest element in X(j).  The estimate is as reliable as
        the estimate for RCOND, and is almost always a slight
        overestimate of the true error.

BERR    (output) double*, dimension (B->ncol)
        The componentwise relative backward error of each solution
        vector X(j) (i.e., the smallest relative change in
        any element of A or B that makes X(j) an exact solution).

info    (output) int*
         = 0:  successful exit
         < 0:  if INFO = -i, the i-th argument had an illegal value
```

A.4 dgssv

```
void
dgssv(SuperMatrix *A, int *perm_c, int *perm_r, SuperMatrix *L,
      SuperMatrix *U, SuperMatrix *B, int *info )
```

Purpose
=======

DGSSV solves the system of linear equations A*X=B, using the LU factorization from DGSTRF. It performs the following steps:

1. If A is stored column-wise (A->Stype = NC):

   1.1. Permute the columns of A, forming A*Pc, where Pc is a permutation matrix. For more details of this step, see sp_preorder.c.

   1.2. Factor A as Pr*A*Pc=L*U with the permutation Pr determined by Gaussian elimination with partial pivoting. L is unit lower triangular with offdiagonal entries bounded by 1 in magnitude, and U is upper triangular.

   1.3. Solve the system of equations A*X=B using the factored form of A.

2. If A is stored row-wise (A->Stype = NR), apply the above algorithm to the transpose of A:

   2.1. Permute columns of transpose(A) (rows of A), forming transpose(A)*Pc, where Pc is a permutation matrix. For more details of this step, see sp_preorder.c.

   2.2. Factor A as Pr*transpose(A)*Pc=L*U with the permutation Pr determined by Gaussian elimination with partial pivoting. L is unit lower triangular with offdiagonal entries bounded by 1 in magnitude, and U is upper triangular.

   2.3. Solve the system of equations A*X=B using the factored form of A.

See supermatrix.h for the definition of 'SuperMatrix' structure.

Arguments
=========

A (input) SuperMatrix*
Matrix A in A*X=B, of dimension (A->nrow, A->ncol). The number of linear equations is A->nrow. Currently, the type of A can be: Stype = NC or NR; Dtype = _D; Mtype = GE. In the future, more general A will be handled.

```
perm_c  (input/output) int*
        If A->Stype = NC, column permutation vector of size A->ncol
        which defines the permutation matrix Pc; perm_c[i] = j means
        column i of A is in position j in A*Pc.
        On exit, perm_c may be overwritten by the product of the input
        perm_c and a permutation that postorders the elimination tree
        of Pc'*A'*A*Pc; perm_c is not changed if the elimination tree
        is already in postorder.

        If A->Stype = NR, column permutation vector of size A->nrow
        which describes permutation of columns of transpose(A)
        (rows of A) as described above.

perm_r  (output) int*
        If A->Stype = NC, row permutation vector of size A->nrow,
        which defines the permutation matrix Pr, and is determined
        by partial pivoting.  perm_r[i] = j means row i of A is in
        position j in Pr*A.

        If A->Stype = NR, permutation vector of size A->ncol, which
        determines permutation of rows of transpose(A)
        (columns of A) as described above.

L       (output) SuperMatrix*
        The factor L from the factorization
            Pr*A*Pc=L*U                (if A->Stype = NC) or
            Pr*transpose(A)*Pc=L*U     (if A->Stype = NR).
        Uses compressed row subscripts storage for supernodes, i.e.,
        L has types: Stype = SC, Dtype = _D, Mtype = TRLU.

U       (output) SuperMatrix*
        The factor U from the factorization
            Pr*A*Pc=L*U                (if A->Stype = NC) or
            Pr*transpose(A)*Pc=L*U     (if A->Stype = NR).
        Uses column-wise storage scheme, i.e., U has types:
        Stype = NC, Dtype = _D, Mtype = TRU.

B       (input/output) SuperMatrix*
        B has types: Stype = DN, Dtype = _D, Mtype = GE.
        On entry, the right hand side matrix.
        On exit, the solution matrix if info = 0;

info    (output) int*
        = 0: successful exit
        > 0: if info = i, and i is
            <= A->ncol: U(i,i) is exactly zero. The factorization has
               been completed, but the factor U is exactly singular,
```

```
            so the solution could not be computed.
          > A->ncol: number of bytes allocated when memory allocation
            failure occurred, plus A->ncol.
```

A.5 dgssvx

```
void
dgssvx(char *fact, char *trans, char *refact,
       SuperMatrix *A, factor_param_t *factor_params, int *perm_c,
       int *perm_r, int *etree, char *equed, double *R, double *C,
       SuperMatrix *L, SuperMatrix *U, void *work, int lwork,
       SuperMatrix *B, SuperMatrix *X, double *recip_pivot_growth,
       double *rcond, double *ferr, double *berr,
       mem_usage_t *mem_usage, int *info )

    Purpose
    =======

    DGSSVX solves the system of linear equations A*X=B or A'*X=B, using
    the LU factorization from dgstrf(). Error bounds on the solution and
    a condition estimate are also provided. It performs the following steps:

      1. If A is stored column-wise (A->Stype = NC):

         1.1. If fact = 'E', scaling factors are computed to equilibrate the
              system:
                trans = 'N':  diag(R)*A*diag(C)     *inv(diag(C))*X = diag(R)*B
                trans = 'T': (diag(R)*A*diag(C))**T *inv(diag(R))*X = diag(C)*B
                trans = 'C': (diag(R)*A*diag(C))**H *inv(diag(R))*X = diag(C)*B
              Whether or not the system will be equilibrated depends on the
              scaling of the matrix A, but if equilibration is used, A is
              overwritten by diag(R)*A*diag(C) and B by diag(R)*B (if trans='N')
              or diag(C)*B (if trans = 'T' or 'C').

         1.2. Permute columns of A, forming A*Pc, where Pc is a permutation
              matrix that usually preserves sparsity.
              For more details of this step, see sp_preorder.c.

         1.3. If fact = 'N' or 'E', the LU decomposition is used to factor the
              matrix A (after equilibration if fact = 'E') as Pr*A*Pc = L*U,
              with Pr determined by partial pivoting.

         1.4. Compute the reciprocal pivot growth factor.

         1.5. If some U(i,i) = 0, so that U is exactly singular, then the
              routine returns with info = i. Otherwise, the factored form of
```

A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, info = A->ncol+1 is returned as a warning, but the routine still goes on to solve for X and computes error bounds as described below.

1.6. The system of equations is solved for X using the factored form of A.

1.7. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

1.8. If equilibration was used, the matrix X is premultiplied by diag(C) (if trans = 'N') or diag(R) (if trans = 'T' or 'C') so that it solves the original system before equilibration.

2. If A is stored row-wise (A->Stype = NR), apply the above algorithm to the transpose of A:

2.1. If fact = 'E', scaling factors are computed to equilibrate the system:

```
trans = 'N':  diag(R)*A'*diag(C)      *inv(diag(C))*X = diag(R)*B
trans = 'T': (diag(R)*A'*diag(C))**T *inv(diag(R))*X = diag(C)*B
trans = 'C': (diag(R)*A'*diag(C))**H *inv(diag(R))*X = diag(C)*B
```

Whether or not the system will be equilibrated depends on the scaling of the matrix A, but if equilibration is used, A' is overwritten by diag(R)*A'*diag(C) and B by diag(R)*B (if trans='N') or diag(C)*B (if trans = 'T' or 'C').

2.2. Permute columns of transpose(A) (rows of A), forming transpose(A)*Pc, where Pc is a permutation matrix that usually preserves sparsity.
For more details of this step, see sp_preorder.c.

2.3. If fact = 'N' or 'E', the LU decomposition is used to factor the transpose(A) (after equilibration if fact = 'E') as Pr*transpose(A)*Pc = L*U with the permutation Pr determined by partial pivoting.

2.4. Compute the reciprocal pivot growth factor.

2.5. If some U(i,i) = 0, so that U is exactly singular, then the routine returns with info = i. Otherwise, the factored form of transpose(A) is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, info = A->nrow+1 is returned as

a warning, but the routine still goes on to solve for X and computes error bounds as described below.

2.6. The system of equations is solved for X using the factored form of transpose(A).

2.7. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

2.8. If equilibration was used, the matrix X is premultiplied by diag(C) (if trans = 'N') or diag(R) (if trans = 'T' or 'C') so that it solves the original system before equilibration.

See supermatrix.h for the definition of 'SuperMatrix' structure.

Arguments

fact (input) char*
Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.
= 'F': On entry, L, U, perm_r and perm_c contain the factored form of A. If equed is not 'N', the matrix A has been equilibrated with scaling factors R and C.
A, L, U, perm_r are not modified.
= 'N': The matrix A will be factored, and the factors will be stored in L and U.
= 'E': The matrix A will be equilibrated if necessary, then factored into L and U.

trans (input) char*
Specifies the form of the system of equations:
= 'N': A * X = B (No transpose)
= 'T': A**T * X = B (Transpose)
= 'C': A**H * X = B (Transpose)

refact (input) char*
Specifies whether we want to re-factor the matrix.
= 'N': Factor the matrix A.
= 'Y': Matrix A was factored before, now we want to re-factor matrix A with perm_r and etree as inputs. Use the same storage for the L\U factors previously allocated, expand it if necessary. User should insure to use the same memory model.
If fact = 'F', then refact is not accessed.

```
A       (input/output) SuperMatrix*
        Matrix A in A*X=B, of dimension (A->nrow, A->ncol). The number
        of the linear equations is A->nrow. Currently, the type of A can be:
        Stype = NC or NR, Dtype = _D, Mtype = GE. In the future,
        more general A can be handled.

        On entry, If fact = 'F' and equed is not 'N', then A must have
        been equilibrated by the scaling factors in R and/or C.
        A is not modified if fact = 'F' or 'N', or if fact = 'E' and
        equed = 'N' on exit.

        On exit, if fact = 'E' and equed is not 'N', A is scaled as follows:
        If A->Stype = NC:
          equed = 'R':  A := diag(R) * A
          equed = 'C':  A := A * diag(C)
          equed = 'B':  A := diag(R) * A * diag(C).
        If A->Stype = NR:
          equed = 'R':  transpose(A) := diag(R) * transpose(A)
          equed = 'C':  transpose(A) := transpose(A) * diag(C)
          equed = 'B':  transpose(A) := diag(R) * transpose(A) * diag(C).

factor_params (input) factor_param_t*
        The structure defines the input scalar parameters, consisting of
        the following fields. If factor_params = NULL, the default
        values are used for all the fields; otherwise, the values
        are given by the user.
        - panel_size (int): Panel size. A panel consists of at most
            panel_size consecutive columns. If panel_size = -1, use
            default value 8.
        - relax (int): To control degree of relaxing supernodes. If the
            number of nodes (columns) in a subtree of the elimination
            tree is less than relax, this subtree is considered as one
            supernode, regardless of the row structures of those columns.
            If relax = -1, use default value 8.
        - diag_pivot_thresh (double): Diagonal pivoting threshold.
            At step j of the Gaussian elimination, if
                abs(A_jj) >= diag_pivot_thresh * (max_(i>=j) abs(A_ij)),
            then use A_jj as pivot. 0 <= diag_pivot_thresh <= 1.
            If diag_pivot_thresh = -1, use default value 1.0,
            which corresponds to standard partial pivoting.
        - drop_tol (double): Drop tolerance threshold. (NOT IMPLEMENTED)
            At step j of the Gaussian elimination, if
                abs(A_ij)/(max_i abs(A_ij)) < drop_tol,
            then drop entry A_ij. 0 <= drop_tol <= 1.
            If drop_tol = -1, use default value 0.0, which corresponds to
            standard Gaussian elimination.
```

```
perm_c  (input/output) int*
        If A->Stype = NC, Column permutation vector of size A->ncol,
        which defines the permutation matrix Pc; perm_c[i] = j means
        column i of A is in position j in A*Pc.
        On exit, perm_c may be overwritten by the product of the input
        perm_c and a permutation that postorders the elimination tree
        of Pc'*A'*A*Pc; perm_c is not changed if the elimination tree
        is already in postorder.

        If A->Stype = NR, column permutation vector of size A->nrow,
        which describes permutation of columns of transpose(A)
        (rows of A) as described above.

perm_r  (input/output) int*
        If A->Stype = NC, row permutation vector of size A->nrow,
        which defines the permutation matrix Pr, and is determined
        by partial pivoting.  perm_r[i] = j means row i of A is in
        position j in Pr*A.

        If A->Stype = NR, permutation vector of size A->ncol, which
        determines permutation of rows of transpose(A)
        (columns of A) as described above.

        If refact is not 'Y', perm_r is output argument;
        If refact = 'Y', the pivoting routine will try to use the input
        perm_r, unless a certain threshold criterion is violated.
        In that case, perm_r is overwritten by a new permutation
        determined by partial pivoting or diagonal threshold pivoting.

etree   (input/output) int*,  dimension (A->ncol)
        Elimination tree of Pc'*A'*A*Pc.
        If fact is not 'F' and refact = 'Y', etree is an input argument,
        otherwise it is an output argument.
        Note: etree is a vector of parent pointers for a forest whose
        vertices are the integers 0 to A->ncol-1; etree[root]==A->ncol.

equed   (input/output) char*
        Specifies the form of equilibration that was done.
        = 'N': No equilibration.
        = 'R': Row equilibration, i.e., A was premultiplied by diag(R).
        = 'C': Column equilibration, i.e., A was postmultiplied by diag(C).
        = 'B': Both row and column equilibration, i.e., A was replaced
               by diag(R)*A*diag(C).
        If fact = 'F', equed is an input argument, otherwise it is
        an output argument.
```

```
R       (input/output) double*, dimension (A->nrow)
        The row scale factors for A or transpose(A).
        If equed = 'R' or 'B', A (if A->Stype = NC) or transpose(A) (if
            A->Stype = NR) is multiplied on the left by diag(R).
        If equed = 'N' or 'C', R is not accessed.
        If fact = 'F', R is an input argument; otherwise, R is output.
        If fact = 'F' and equed = 'R' or 'B', each element of R must
           be positive.

C       (input/output) double*, dimension (A->ncol)
        The column scale factors for A or transpose(A).
        If equed = 'C' or 'B', A (if A->Stype = NC) or transpose(A) (if
            A->Stype = NR) is multiplied on the right by diag(C).
        If equed = 'N' or 'R', C is not accessed.
        If fact = 'F', C is an input argument; otherwise, C is output.
        If fact = 'F' and equed = 'C' or 'B', each element of C must
           be positive.

L       (output) SuperMatrix*
        The factor L from the factorization
            Pr*A*Pc=L*U              (if A->Stype = NC) or
            Pr*transpose(A)*Pc=L*U   (if A->Stype = NR).
        Uses compressed row subscripts storage for supernodes, i.e.,
        L has types: Stype = SC, Dtype = _D, Mtype = TRLU.

U       (output) SuperMatrix*
        The factor U from the factorization
            Pr*A*Pc=L*U              (if A->Stype = NC) or
            Pr*transpose(A)*Pc=L*U   (if A->Stype = NR).
        Uses column-wise storage scheme, i.e., U has types:
        Stype = NC, Dtype = _D, Mtype = TRU.

work    (workspace/output) void*, size (lwork) (in bytes)
        User supplied workspace, should be large enough
        to hold data structures for factors L and U.
        On exit, if fact is not 'F', L and U point to this array.

lwork   (input) int
        Specifies the size of work array in bytes.
        = 0:  allocate space internally by system malloc;
        > 0:  use user-supplied work array of length lwork in bytes,
              returns error if space runs out.
        = -1: the routine guesses the amount of space needed without
              performing the factorization, and returns it in
              mem_usage->total_needed; no other side effects.

        See argument 'mem_usage' for memory usage statistics.
```

```
B       (input/output) SuperMatrix*
        B has types: Stype = DN, Dtype = _D, Mtype = GE.
        On entry, the right hand side matrix.
        On exit,
           if equed = 'N', B is not modified; otherwise
           if A->Stype = NC:
              if trans = 'N' and equed = 'R' or 'B', B is overwritten by
                 diag(R)*B;
              if trans = 'T' or 'C' and equed = 'C' of 'B', B is
                 overwritten by diag(C)*B;
           if A->Stype = NR:
              if trans = 'N' and equed = 'C' or 'B', B is overwritten by
                 diag(C)*B;
              if trans = 'T' or 'C' and equed = 'R' of 'B', B is
                 overwritten by diag(R)*B.

X       (output) SuperMatrix*
        X has types: Stype = DN, Dtype = _D, Mtype = GE.
        If info = 0 or info = A->ncol+1, X contains the solution matrix
        to the original system of equations. Note that A and B are modified
        on exit if equed is not 'N', and the solution to the equilibrated
        system is inv(diag(C))*X if trans = 'N' and equed = 'C' or 'B',
        or inv(diag(R))*X if trans = 'T' or 'C' and equed = 'R' or 'B'.

recip_pivot_growth (output) double*
        The reciprocal pivot growth factor max_j( norm(A_j)/norm(U_j) ).
        The infinity norm is used. If recip_pivot_growth is much less
        than 1, the stability of the LU factorization could be poor.

rcond   (output) double*
        The estimate of the reciprocal condition number of the matrix A
        after equilibration (if done). If rcond is less than the machine
        precision (in particular, if rcond = 0), the matrix is singular
        to working precision. This condition is indicated by a return
        code of info > 0.

FERR    (output) double*, dimension (B->ncol)
        The estimated forward error bound for each solution vector
        X(j) (the j-th column of the solution matrix X).
        If XTRUE is the true solution corresponding to X(j), FERR(j)
        is an estimated upper bound for the magnitude of the largest
        element in (X(j) - XTRUE) divided by the magnitude of the
        largest element in X(j).  The estimate is as reliable as
        the estimate for RCOND, and is almost always a slight
        overestimate of the true error.
```

```
BERR     (output) double*, dimension (B->ncol)
         The componentwise relative backward error of each solution
         vector X(j) (i.e., the smallest relative change in
         any element of A or B that makes X(j) an exact solution).

mem_usage (output) mem_usage_t*
         Record the memory usage statistics, consisting of following fields:
         - for_lu (float)
           The amount of space used in bytes for L\U data structures.
         - total_needed (float)
           The amount of space needed in bytes to perform factorization.
         - expansions (int)
           The number of memory expansions during the LU factorization.

info     (output) int*
         = 0: successful exit
         < 0: if info = -i, the i-th argument had an illegal value
         > 0: if info = i, and i is
              <= A->ncol: U(i,i) is exactly zero. The factorization has
                    been completed, but the factor U is exactly
                    singular, so the solution and error bounds
                    could not be computed.
              = A->ncol+1: U is nonsingular, but RCOND is less than machine
                    precision, meaning that the matrix is singular to
                    working precision. Nevertheless, the solution and
                    error bounds are computed because there are a number
                    of situations where the computed solution can be more
                    accurate than the value of RCOND would suggest.
              > A->ncol+1: number of bytes allocated when memory allocation
                    failure occurred, plus A->ncol.
```

A.6 dgstrf

```
void
dgstrf(char *refact, SuperMatrix *A, double diag_pivot_thresh,
       double drop_tol, int relax, int panel_size, int *etree,
       void *work, int lwork, int *perm_r, int *perm_c,
       SuperMatrix *L, SuperMatrix *U, int *info)

    Purpose
    =======

    DGSTRF computes an LU factorization of a general sparse m-by-n
    matrix A using partial pivoting with row interchanges.
    The factorization has the form
        Pr * A = L * U
```

where Pr is a row permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if A->nrow > A->ncol), and U is upper triangular (upper trapezoidal if A->nrow < A->ncol).

See supermatrix.h for the definition of 'SuperMatrix' structure.

Arguments
=========

refact (input) char*
Specifies whether we want to use perm_r from a previous factor.
= 'Y': re-use perm_r; perm_r is input, unchanged on exit.
= 'N': perm_r is determined by partial pivoting, and output.

A (input) SuperMatrix*
Original matrix A, permuted by columns, of dimension
(A->nrow, A->ncol). The type of A can be:
Stype = NCP; Dtype = D; Mtype = GE.

diag_pivot_thresh (input) double
Diagonal pivoting threshold. At step j of the Gaussian elimination,
if abs(A_jj) >= thresh * (max_(i>=j) abs(A_ij)), use A_jj as pivot.
0 <= thresh <= 1. The default value of thresh is 1, corresponding
to partial pivoting.

drop_tol (input) double (NOT IMPLEMENTED)
Drop tolerance parameter. At step j of the Gaussian elimination,
if abs(A_ij)/(max_i abs(A_ij)) < drop_tol, drop entry A_ij.
0 <= drop_tol <= 1. The default value of drop_tol is 0.

relax (input) int
To control degree of relaxing supernodes. If the number
of nodes (columns) in a subtree of the elimination tree is less
than relax, this subtree is considered as one supernode,
regardless of the row structures of those columns.

panel_size (input) int
A panel consists of at most panel_size consecutive columns.

etree (input) int*, dimension (A->ncol)
Elimination tree of A'*A.
Note: etree is a vector of parent pointers for a forest whose
vertices are the integers 0 to A->ncol-1; etree[root]==A->ncol.
On input, the columns of A should be permuted so that the
etree is in a certain postorder.

work (input/output) void*, size (lwork) (in bytes)

```
        User-supplied work space and space for the output data structures.
        Not referenced if lwork = 0;

lwork   (input) int
        Specifies the size of work array in bytes.
        = 0:  allocate space internally by system malloc;
        > 0:  use user-supplied work array of length lwork in bytes,
              returns error if space runs out.
        = -1: the routine guesses the amount of space needed without
              performing the factorization, and returns it in
              *info; no other side effects.

perm_r  (input/output) int*, dimension (A->nrow)
        Row permutation vector which defines the permutation matrix Pr,
        perm_r[i] = j means row i of A is in position j in Pr*A.
        If refact is not 'Y', perm_r is output argument;
        If refact = 'Y', the pivoting routine will try to use the input
        perm_r, unless a certain threshold criterion is violated.
        In that case, perm_r is overwritten by a new permutation
        determined by partial pivoting or diagonal threshold pivoting.

perm_c  (input) int*, dimension (A->ncol)
        Column permutation vector, which defines the
        permutation matrix Pc; perm_c[i] = j means column i of A is
        in position j in A*Pc.
        When searching for diagonal, perm_c[*] is applied to the
        row subscripts of A, so that diagonal threshold pivoting
        can find the diagonal of A, rather than that of A*Pc.

L       (output) SuperMatrix*
        The factor L from the factorization Pr*A=L*U; use compressed row
        subscripts storage for supernodes, i.e., L has type:
        Stype = SC, Dtype = _D, Mtype = TRLU.

U       (output) SuperMatrix*
        The factor U from the factorization Pr*A*Pc=L*U. Use column-wise
        storage scheme, i.e., U has types: Stype = NC,
        Dtype = _D, Mtype = TRU.

info    (output) int*
        = 0: successful exit
        < 0: if info = -i, the i-th argument had an illegal value
        > 0: if info = i, and i is
           <= A->ncol: U(i,i) is exactly zero. The factorization has
              been completed, but the factor U is exactly singular,
              and division by zero will occur if it is used to solve a
              system of equations.
```

```
> A->ncol: number of bytes allocated when memory allocation
  failure occurred, plus A->ncol. If lwork = -1, it is
  the estimated amount of space needed, plus A->ncol.
```

A.7 dgstrs

```
void
dgstrs(char *trans, SuperMatrix *L, SuperMatrix *U,
       int *perm_r, int *perm_c, SuperMatrix *B, int *info)

    Purpose
    =======

    DGSTRS solves a system of linear equations A*X=B or A'*X=B
    with A sparse and B dense, using the LU factorization computed by
    DGSTRF.

    See supermatrix.h for the definition of 'SuperMatrix' structure.

    Arguments
    =========

    trans   (input) char*
            Specifies the form of the system of equations:
            = 'N':  A * X = B  (No transpose)
            = 'T':  A'* X = B  (Transpose)
            = 'C':  A**H * X = B  (Conjugate transpose)

    L       (input) SuperMatrix*
            The factor L from the factorization Pr*A*Pc=L*U as computed by
            dgstrf(). Use compressed row subscripts storage for supernodes,
            i.e., L has types: Stype = SC, Dtype = _D, Mtype = TRLU.

    U       (input) SuperMatrix*
            The factor U from the factorization Pr*A*Pc=L*U as computed by
            dgstrf(). Use column-wise storage scheme, i.e., U has types:
            Stype = NC, Dtype = _D, Mtype = TRU.

    perm_r  (input) int*, dimension (L->nrow)
            Row permutation vector, which defines the permutation matrix Pr;
            perm_r[i] = j means row i of A is in position j in Pr*A.

    perm_c  (input) int*, dimension (L->ncol)
            Column permutation vector, which defines the
            permutation matrix Pc; perm_c[i] = j means column i of A is
            in position j in A*Pc.
```

```
B       (input/output) SuperMatrix*
        B has types: Stype = DN, Dtype = _D, Mtype = GE.
        On entry, the right hand side matrix.
        On exit, the solution matrix if info = 0;

info    (output) int*
        = 0: successful exit
        < 0: if info = -i, the i-th argument had an illegal value
```

A.8 dlaqgs

```
void
dlaqgs(SuperMatrix *A, double *r, double *c,
       double rowcnd, double colcnd, double amax, char *equed)

    Purpose
    =======

    DLAQGS equilibrates a general sparse M by N matrix A using the row and
    scaling factors in the vectors R and C.

    See supermatrix.h for the definition of 'SuperMatrix' structure.

    Arguments
    =========

    A       (input/output) SuperMatrix*
            On exit, the equilibrated matrix.  See EQUED for the form of
            the equilibrated matrix. The type of A can be:
            Stype = NC; Dtype = _D; Mtype = GE.

    R       (input) double*, dimension (A->nrow)
            The row scale factors for A.

    C       (input) double*, dimension (A->ncol)
            The column scale factors for A.

    rowcnd  (input) double
            Ratio of the smallest R(i) to the largest R(i).

    colcnd  (input) double
            Ratio of the smallest C(i) to the largest C(i).

    amax    (input) double
            Absolute value of largest matrix entry.
```

```
equed   (output) char*
        Specifies the form of equilibration that was done.
        = 'N':  No equilibration
        = 'R':  Row equilibration, i.e., A has been premultiplied by
                diag(R).
        = 'C':  Column equilibration, i.e., A has been postmultiplied
                by diag(C).
        = 'B':  Both row and column equilibration, i.e., A has been
                replaced by diag(R) * A * diag(C).
```

Appendix B

Specifications of routines in multithreaded SuperLU_MT

B.1 pdgssv

```
void
pdgssv(int nprocs, SuperMatrix *A, int *perm_c, int *perm_r,
       SuperMatrix *L, SuperMatrix *U, SuperMatrix *B, int *info )


    Purpose
    =======

    pdgssv() solves the system of linear equations A*X=B, using the parallel
    LU factorization routine pdgstrf(). It performs the following steps:

      1. If A is stored column-wise (A->Stype = NC):

         1.1. Permute the columns of A, forming A*Pc, where Pc is a
              permutation matrix.
              For more details of this step, see sp_preorder.c.

         1.2. Factor A as Pr*A*Pc=L*U with the permutation Pr determined
              by Gaussian elimination with partial pivoting.
              L is unit lower triangular with offdiagonal entries
              bounded by 1 in magnitude, and U is upper triangular.

         1.3. Solve the system of equations A*X=B using the factored
              form of A.

      2. If A is stored row-wise (A->Stype = NR), apply the above algorithm
         to the transpose of A:

         2.1. Permute columns of transpose(A) (rows of A),
              forming transpose(A)*Pc, where Pc is a permutation matrix.
```

For more details of this step, see sp_preorder.c.

2.2. Factor A as Pr*transpose(A)*Pc=L*U with the permutation Pr determined by Gaussian elimination with partial pivoting. L is unit lower triangular with offdiagonal entries bounded by 1 in magnitude, and U is upper triangular.

2.3. Solve the system of equations A*X=B using the factored

See supermatrix.h for the definition of "SuperMatrix" structure.

Arguments
=========

nprocs (input) int
Number of processes (or threads) to be spawned and used to perform the LU factorization by pdgstrf(). There is a single thread of control to call pdgstrf(), and all threads spawned by pdgstrf() are terminated before returning from pdgstrf().

A (input) SuperMatrix*
Matrix A in A*X=B, of dimension (A->nrow, A->ncol), where A->nrow = A->ncol. Currently, the type of A can be: Stype = NC or NR; Dtype = _D; Mtype = GE. In the future, more general A will be handled.

perm_c (input/output) int*
If A->Stype=NC, column permutation vector of size A->ncol, which defines the permutation matrix Pc; perm_c[i] = j means column i of A is in position j in A*Pc.
On exit, perm_c may be overwritten by the product of the input perm_c and a permutation that postorders the elimination tree of Pc'*A'*A*Pc; perm_c is not changed if the elimination tree is already in postorder.

If A->Stype=NR, column permutation vector of size A->nrow which describes permutation of columns of transpose(A) (rows of A) as described above.

perm_r (output) int*,
If A->Stype=NR, row permutation vector of size A->nrow, which defines the permutation matrix Pr, and is determined by partial pivoting. perm_r[i] = j means row i of A is in position j in Pr*A.

If A->Stype=NR, permutation vector of size A->ncol, which

```
        determines permutation of rows of transpose(A)
        (columns of A) as described above.

L       (output) SuperMatrix*
        The factor L from the factorization
            Pr*A*Pc=L*U              (if A->Stype=NC) or
            Pr*transpose(A)*Pc=L*U   (if A->Stype=NR).
        Uses compressed row subscripts storage for supernodes, i.e.,
        L has types: Stype = SCP, Dtype = _D, Mtype = TRLU.

U       (output) SuperMatrix*
        The factor U from the factorization
            Pr*A*Pc=L*U              (if A->Stype=NC) or
            Pr*transpose(A)*Pc=L*U   (if A->Stype=NR).
        Use column-wise storage scheme, i.e., U has types:
        Stype = NCP, Dtype = _D, Mtype = TRU.

B       (input/output) SuperMatrix*
        B has types: Stype = DN, Dtype = _D, Mtype = GE.
        On entry, the right hand side matrix.
        On exit, the solution matrix if info = 0;

info    (output) int*
        = 0: successful exit
        > 0: if info = i, and i is
             <= A->ncol: U(i,i) is exactly zero. The factorization has
                been completed, but the factor U is exactly singular,
                so the solution could not be computed.
             > A->ncol: number of bytes allocated when memory allocation
                failure occurred, plus A->ncol.
```

B.2 pdgssvx

```
void
pdgssvx(int nprocs, pdgstrf_options_t *pdgstrf_options, SuperMatrix *A,
        int *perm_c, int *perm_r, equed_t *equed, double *R, double *C,
        SuperMatrix *L, SuperMatrix *U,
        SuperMatrix *B, SuperMatrix *X, double *recip_pivot_growth,
        double *rcond, double *ferr, double *berr,
        superlu_memusage_t *superlu_memusage, int *info)


    Purpose
    =======

    PDGSSVX solves the system of linear equations A*X=B or A'*X=B, using
    the LU factorization from dgstrf(). Error bounds on the solution and
```

a condition estimate are also provided. It performs the following steps:

1. If A is stored column-wise (A->Stype = NC):

   1.1. If fact = EQUILIBRATE, scaling factors are computed to equilibrate the system:
   trans = NOTRANS: diag(R)*A*diag(C)*inv(diag(C))*X = diag(R)*B
   trans = TRANS: (diag(R)*A*diag(C))**T *inv(diag(R))*X = diag(C)*B
   trans = CONJ: (diag(R)*A*diag(C))**H *inv(diag(R))*X = diag(C)*B
   Whether or not the system will be equilibrated depends on the scaling of the matrix A, but if equilibration is used, A is overwritten by diag(R)*A*diag(C) and B by diag(R)*B (if trans = NOTRANS) or diag(C)*B (if trans = TRANS or CONJ).

   1.2. Permute columns of A, forming A*Pc, where Pc is a permutation matrix that usually preserves sparsity.
   For more details of this step, see sp_colorder.c.

   1.3. If fact = DOFACT or EQUILIBRATE, the LU decomposition is used to factor the matrix A (after equilibration if fact = EQUILIBRATE) as Pr*A*Pc = L*U, with Pr determined by partial pivoting.

   1.4. Compute the reciprocal pivot growth factor.

   1.5. If some U(i,i) = 0, so that U is exactly singular, then the routine returns with info = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, info = A->ncol+1 is returned as a warning, but the routine still goes on to solve for X and computes error bounds as described below.

   1.6. The system of equations is solved for X using the factored form of A.

   1.7. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

   1.8. If equilibration was used, the matrix X is premultiplied by diag(C) (if trans = NOTRANS) or diag(R) (if trans = TRANS or CONJ) so that it solves the original system before equilibration.

2. If A is stored row-wise (A->Stype = NR), apply the above algorithm to the transpose of A:

   2.1. If fact = EQUILIBRATE, scaling factors are computed to equilibrate the system:

```
trans = NOTRANS:diag(R)*A'*diag(C)*inv(diag(C))*X = diag(R)*B
trans = TRANS: (diag(R)*A'*diag(C))**T *inv(diag(R))*X = diag(C)*B
trans = CONJ:  (diag(R)*A'*diag(C))**H *inv(diag(R))*X = diag(C)*B
```

Whether or not the system will be equilibrated depends on the scaling of the matrix A, but if equilibration is used, A' is overwritten by diag(R)*A'*diag(C) and B by diag(R)*B (if trans = NOTRANS) or diag(C)*B (if trans = TRANS or CONJ).

2.2. Permute columns of transpose(A) (rows of A), forming transpose(A)*Pc, where Pc is a permutation matrix that usually preserves sparsity.
For more details of this step, see sp_colorder.c.

2.3. If fact = DOFACT or EQUILIBRATE, the LU decomposition is used to factor the matrix A (after equilibration if fact = EQUILIBRATE) as Pr*transpose(A)*Pc = L*U, with the permutation Pr determined by partial pivoting.

2.4. Compute the reciprocal pivot growth factor.

2.5. If some U(i,i) = 0, so that U is exactly singular, then the routine returns with info = i. Otherwise, the factored form of transpose(A) is used to estimate the condition number of the matrix A.
If the reciprocal of the condition number is less than machine precision, info = A->nrow+1 is returned as a warning, but the routine still goes on to solve for X and computes error bounds as described below.

2.6. The system of equations is solved for X using the factored form of transpose(A).

2.7. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

2.8. If equilibration was used, the matrix X is premultiplied by diag(C) (if trans = NOTRANS) or diag(R) (if trans = TRANS or CONJ) so that it solves the original system before equilibration.

See supermatrix.h for the definition of 'SuperMatrix' structure.

Arguments
=========

procs (input) int
Number of processes (or threads) to be spawned and used to perform the LU factorization by pdgstrf(). There is a single thread of

control to call pdgstrf(), and all threads spawned by pdgstrf() are terminated before returning from pdgstrf().

pdgstrf_options (input) pdgstrf_options_t*

The structure defines the input parameters and data structure to control how the LU factorization will be performed. The following fields should be defined for this structure:

o fact (fact_t)

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

= FACTORED: On entry, L, U, perm_r and perm_c contain the factored form of A. If equed is not NOEQUIL, the matrix A has been equilibrated with scaling factors R and C. A, L, U, perm_r are not modified.

= DOFACT: The matrix A will be factored, and the factors will be stored in L and U.

= EQUILIBRATE: The matrix A will be equilibrated if necessary, then factored into L and U.

o trans (trans_t)

Specifies the form of the system of equations:

```
= NOTRANS: A * X = B        (No transpose)
= TRANS:   A**T * X = B     (Transpose)
= CONJ:    A**H * X = B     (Transpose)
```

o refact (yes_no_t)

Specifies whether this is first time or subsequent factorization.

= NO: this factorization is treated as the first one;

= YES: it means that a factorization was performed prior to this one. Therefore, this factorization will reuse some existing data structures, such as L and U storage, column elimination tree, and the symbolic information of the Householder matrix.

o panel_size (int)

A panel consists of at most panel_size consecutive columns.

o relax (int)

To control degree of relaxing supernodes. If the number of nodes (columns) in a subtree of the elimination tree is less than relax, this subtree is considered as one supernode, regardless of the row structures of those columns.

o diag_pivot_thresh (double)

Diagonal pivoting threshold. At step j of the Gaussian

```
  elimination, if
      abs(A_jj) >= diag_pivot_thresh * (max_(i>=j) abs(A_ij)),
  use A_jj as pivot, else use A_ij with maximum magnitude.
  0 <= diag_pivot_thresh <= 1. The default value is 1,
  corresponding to partial pivoting.

o usepr (yes_no_t)
  Whether the pivoting will use perm_r specified by the user.
  = YES: use perm_r; perm_r is input, unchanged on exit.
  = NO:  perm_r is determined by partial pivoting, and is output.

o drop_tol (double) (NOT IMPLEMENTED)
  Drop tolerance parameter. At step j of the Gaussian elimination,
  if abs(A_ij)/(max_i abs(A_ij)) < drop_tol, drop entry A_ij.
  0 <= drop_tol <= 1. The default value of drop_tol is 0,
  corresponding to not dropping any entry.

o work (void*) of size lwork
  User-supplied work space and space for the output data structures.
  Not referenced if lwork = 0;

o lwork (int)
  Specifies the length of work array.
  = 0:  allocate space internally by system malloc;
  > 0:  use user-supplied work array of length lwork in bytes,
        returns error if space runs out.
  = -1: the routine guesses the amount of space needed without
        performing the factorization, and returns it in
        superlu_memusage->total_needed; no other side effects.

A       (input/output) SuperMatrix*
        Matrix A in A*X=B, of dimension (A->nrow, A->ncol), where
        A->nrow = A->ncol. Currently, the type of A can be:
        Stype = NC or NR, Dtype = _D, Mtype = GE. In the future,
        more general A will be handled.

        On entry, If pdgstrf_options->fact = FACTORED and equed is not
        NOEQUIL, then A must have been equilibrated by the scaling factors
        in R and/or C.  On exit, A is not modified
        if pdgstrf_options->fact = FACTORED or DOFACT, or
        if pdgstrf_options->fact = EQUILIBRATE and equed = NOEQUIL.

        On exit, if pdgstrf_options->fact = EQUILIBRATE and equed is not
        NOEQUIL, A is scaled as follows:
        If A->Stype = NC:
          equed = ROW:  A := diag(R) * A
          equed = COL:  A := A * diag(C)
```

```
        equed = BOTH: A := diag(R) * A * diag(C).
      If A->Stype = NR:
        equed = ROW:  transpose(A) := diag(R) * transpose(A)
        equed = COL:  transpose(A) := transpose(A) * diag(C)
        equed = BOTH: transpose(A) := diag(R) * transpose(A) * diag(C).

perm_c  (input/output) int*
      If A->Stype = NC, Column permutation vector of size A->ncol,
      which defines the permutation matrix Pc; perm_c[i] = j means
      column i of A is in position j in A*Pc.
      On exit, perm_c may be overwritten by the product of the input
      perm_c and a permutation that postorders the elimination tree
      of Pc'*A'*A*Pc; perm_c is not changed if the elimination tree
      is already in postorder.

      If A->Stype = NR, column permutation vector of size A->nrow,
      which describes permutation of columns of transpose(A)
      (rows of A) as described above.

perm_r  (input/output) int*
      If A->Stype = NC, row permutation vector of size A->nrow,
      which defines the permutation matrix Pr, and is determined
      by partial pivoting.  perm_r[i] = j means row i of A is in
      position j in Pr*A.

      If A->Stype = NR, permutation vector of size A->ncol, which
      determines permutation of rows of transpose(A)
      (columns of A) as described above.

      If pdgstrf_options->usepr = NO, perm_r is output argument;
      If pdgstrf_options->usepr = YES, the pivoting routine will try
         to use the input perm_r, unless a certain threshold criterion
         is violated. In that case, perm_r is overwritten by a new
         permutation determined by partial pivoting or diagonal
         threshold pivoting.

equed   (input/output) equed_t*
      Specifies the form of equilibration that was done.
      = NOEQUIL: No equilibration.
      = ROW:  Row equilibration, i.e., A was premultiplied by diag(R).
      = COL:  Column equilibration, i.e., A was postmultiplied by diag(C).
      = BOTH: Both row and column equilibration, i.e., A was replaced
              by diag(R)*A*diag(C).
      If pdgstrf_options->fact = FACTORED, equed is an input argument,
      otherwise it is an output argument.

R       (input/output) double*, dimension (A->nrow)
```

```
        The row scale factors for A or transpose(A).
        If equed = ROW or BOTH, A (if A->Stype = NC) or transpose(A)
           (if A->Stype = NR) is multiplied on the left by diag(R).
        If equed = NOEQUIL or COL, R is not accessed.
        If fact = FACTORED, R is an input argument; otherwise, R is output.
        If fact = FACTORED and equed = ROW or BOTH, each element of R must
           be positive.

C       (input/output) double*, dimension (A->ncol)
        The column scale factors for A or transpose(A).
        If equed = COL or BOTH, A (if A->Stype = NC) or transpose(A)
           (if A->Stype = NR) is multiplied on the right by diag(C).
        If equed = NOEQUIL or ROW, C is not accessed.
        If fact = FACTORED, C is an input argument; otherwise, C is output.
        If fact = FACTORED and equed = COL or BOTH, each element of C must
           be positive.

L       (output) SuperMatrix*
        The factor L from the factorization
            Pr*A*Pc=L*U              (if A->Stype = NC) or
            Pr*transpose(A)*Pc=L*U   (if A->Stype = NR).
        Uses compressed row subscripts storage for supernodes, i.e.,
        L has types: Stype = SCP, Dtype = _D, Mtype = TRLU.

U       (output) SuperMatrix*
        The factor U from the factorization
            Pr*A*Pc=L*U              (if A->Stype = NC) or
            Pr*transpose(A)*Pc=L*U   (if A->Stype = NR).
        Uses column-wise storage scheme, i.e., U has types:
        Stype = NCP, Dtype = _D, Mtype = TRU.

B       (input/output) SuperMatrix*
        B has types: Stype = DN, Dtype = _D, Mtype = GE.
        On entry, the right hand side matrix.
        On exit,
           if equed = NOEQUIL, B is not modified; otherwise
           if A->Stype = NC:
              if trans = NOTRANS and equed = ROW or BOTH, B is overwritten
                 by diag(R)*B;
              if trans = TRANS or CONJ and equed = COL of BOTH, B is
                 overwritten by diag(C)*B;
           if A->Stype = NR:
              if trans = NOTRANS and equed = COL or BOTH, B is overwritten
                 by diag(C)*B;
              if trans = TRANS or CONJ and equed = ROW of BOTH, B is
                 overwritten by diag(R)*B.
```

```
X       (output) SuperMatrix*
        X has types: Stype = DN, Dtype = _D, Mtype = GE.
        If info = 0 or info = A->ncol+1, X contains the solution matrix
        to the original system of equations. Note that A and B are modified
        on exit if equed is not NOEQUIL, and the solution to the
        equilibrated system is inv(diag(C))*X if trans = NOTRANS and
        equed = COL or BOTH, or inv(diag(R))*X if trans = TRANS or CONJ
        and equed = ROW or BOTH.

recip_pivot_growth (output) double*
        The reciprocal pivot growth factor computed as
            max_j ( max_i(abs(A_ij)) / max_i(abs(U_ij)) ).
        If recip_pivot_growth is much less than 1, the stability of the
        LU factorization could be poor.

rcond   (output) double*
        The estimate of the reciprocal condition number of the matrix A
        after equilibration (if done). If rcond is less than the machine
        precision (in particular, if rcond = 0), the matrix is singular
        to working precision. This condition is indicated by a return
        code of info > 0.

ferr    (output) double*, dimension (B->ncol)
        The estimated forward error bound for each solution vector
        X(j) (the j-th column of the solution matrix X).
        If XTRUE is the true solution corresponding to X(j), FERR(j)
        is an estimated upper bound for the magnitude of the largest
        element in (X(j) - XTRUE) divided by the magnitude of the
        largest element in X(j).  The estimate is as reliable as
        the estimate for RCOND, and is almost always a slight
        overestimate of the true error.

berr    (output) double*, dimension (B->ncol)
        The componentwise relative backward error of each solution
        vector X(j) (i.e., the smallest relative change in
        any element of A or B that makes X(j) an exact solution).

superlu_memusage (output) superlu_memusage_t*
        Record the memory usage statistics, consisting of following fields:
        - for_lu (float)
          The amount of space used in bytes for L\U data structures.
        - total_needed (float)
          The amount of space needed in bytes to perform factorization.
        - expansions (int)
          The number of memory expansions during the LU factorization.

info    (output) int*
```

```
        = 0: successful exit
        < 0: if info = -i, the i-th argument had an illegal value
        > 0: if info = i, and i is
             <= A->ncol: U(i,i) is exactly zero. The factorization has
                  been completed, but the factor U is exactly
                  singular, so the solution and error bounds
                  could not be computed.
             = A->ncol+1: U is nonsingular, but RCOND is less than machine
                  precision, meaning that the matrix is singular to
                  working precision. Nevertheless, the solution and
                  error bounds are computed because there are a number
                  of situations where the computed solution can be more
                  accurate than the value of RCOND would suggest.
             > A->ncol+1: number of bytes allocated when memory allocation
                  failure occurred, plus A->ncol.
```

B.3 pdgstrf

```
void
pdgstrf(pdgstrf_options_t *pdgstrf_options, SuperMatrix *A, int *perm_r,
        SuperMatrix *L, SuperMatrix *U, Gstat_t *Gstat, int *info)

   Purpose
   =======

   PDGSTRF computes an LU factorization of a general sparse nrow-by-ncol
   matrix A using partial pivoting with row interchanges. The factorization
   has the form
       Pr * A = L * U
   where Pr is a row permutation matrix, L is lower triangular with unit
   diagonal elements (lower trapezoidal if A->nrow > A->ncol), and U is
   upper triangular (upper trapezoidal if A->nrow < A->ncol).

   Arguments
   =========

   pdgstrf_options (input) pdgstrf_options_t*
          The structure defines the parameters to control how the sparse
          LU factorization is performed. The following fields must be set
          by the user:

          o nprocs (int)
            Number of processes to be spawned and used for factorization.

          o refact (yes_no_t)
            Specifies whether this is first time or subsequent factorization.
```

= NO: this factorization is treated as the first one;
= YES: it means that a factorization was performed prior to this one. Therefore, this factorization will reuse some existing data structures, such as L and U storage, column elimination tree, and the symbolic information of the Householder matrix.

o panel_size (int)
A panel consists of at most panel_size consecutive columns.

o relax (int)
Degree of relaxing supernodes. If the number of nodes (columns) in a subtree of the elimination tree is less than relax, this subtree is considered as one supernode, regardless of the row structures of those columns.

o diag_pivot_thresh (double)
Diagonal pivoting threshold. At step j of Gaussian elimination, if abs(A_jj) >= diag_pivot_thresh * (max_(i>=j) abs(A_ij)), use A_jj as pivot. 0 <= diag_pivot_thresh <= 1. The default value is 1.0, corresponding to partial pivoting.

o usepr (yes_no_t)
Whether the pivoting will use perm_r specified by the user.
= YES: use perm_r; perm_r is input, unchanged on exit.
= NO: perm_r is determined by partial pivoting, and is output.

o drop_tol (double) (NOT IMPLEMENTED)
Drop tolerance parameter. At step j of the Gaussian elimination, if abs(A_ij)/(max_i abs(A_ij)) < drop_tol, drop entry A_ij. 0 <= drop_tol <= 1. The default value of drop_tol is 0, corresponding to not dropping any entry.

o perm_c (int*)
Column permutation vector of size A->ncol, which defines the permutation matrix Pc; perm_c[i] = j means column i of A is in position j in A*Pc.

o perm_r (int*)
Column permutation vector of size A->nrow.
If pdgstrf_options->usepr = NO, this is an output argument.

o work (void*) of size lwork
User-supplied work space and space for the output data structures. Not referenced if lwork = 0;

o lwork (int)

```
        Specifies the length of work array.
          = 0:  allocate space internally by system malloc;
          > 0:  use user-supplied work array of length lwork in bytes,
                returns error if space runs out.
          = -1: the routine guesses the amount of space needed without
                performing the factorization, and returns it in
                superlu_memusage->total_needed; no other side effects.

A       (input) SuperMatrix*
        Original matrix A, permuted by columns, of dimension
        (A->nrow, A->ncol). The type of A can be:
        Stype = NCP; Dtype = _D; Mtype = GE.

perm_r (input/output) int*, dimension A->nrow
        Row permutation vector which defines the permutation matrix Pr,
        perm_r[i] = j means row i of A is in position j in Pr*A.
        If pdgstrf_options->usepr = NO, perm_r is output argument;
        If pdgstrf_options->usepr = YES, the pivoting routine will try
           to use the input perm_r, unless a certain threshold criterion
           is violated. In that case, perm_r is overwritten by a new
           permutation determined by partial pivoting or diagonal
           threshold pivoting.

L       (output) SuperMatrix*
        The factor L from the factorization Pr*A=L*U; use compressed row
        subscripts storage for supernodes, i.e., L has type:
        Stype = SCP, Dtype = _D, Mtype = TRLU.

U       (output) SuperMatrix*
        The factor U from the factorization Pr*A*Pc=L*U. Use column-wise
        storage scheme, i.e., U has types: Stype = NCP, Dtype = _D,
        Mtype = TRU.

Gstat   (output) Gstat_t*
        Record all the statistics about the factorization;
        See Gstat_t structure defined in util.h.

info    (output) int*
        = 0: successful exit
        < 0: if info = -i, the i-th argument had an illegal value
        > 0: if info = i, and i is
             <= A->ncol: U(i,i) is exactly zero. The factorization has
                been completed, but the factor U is exactly singular,
                and division by zero will occur if it is used to solve a
                system of equations.
             > A->ncol: number of bytes allocated when memory allocation
                failure occurred, plus A->ncol.
```

Appendix C

Specifications of routines in MPI-based SuperLU_DIST

C.1 pdgssvx_ABglobal

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

  Purpose
  =======

  pdgssvx_ABglobal solves a system of linear equations A*X=B,
  by using Gaussian elimination with "static pivoting" to
  compute the LU factorization of A.

  Static pivoting is a technique that combines the numerical stability
  of partial pivoting with the scalability of Cholesky (no pivoting),
  to run accurately and efficiently on large numbers of processors.

  See our paper at http://www.nersc.gov/~xiaoye/SuperLU/ for a detailed
  description of the parallel algorithms.

  Here are the options for using this code:

    1. Independent of all the other options specified below, the
       user must supply

       - B, the matrix of right hand sides, and its dimensions ldb and nrhs
       - grid, a structure describing the 2D processor mesh
       - options->IterRefine, which determines whether or not to
           improve the accuracy of the computed solution using
```

iterative refinement

On output, B is overwritten with the solution X.

2. Depending on options->Fact, the user has several options for solving A*X=B. The standard option is for factoring A "from scratch". (The other options, described below, are used when A is sufficiently similar to a previously solved problem to save time by reusing part or all of the previous factorization.)

   - options->Fact = DOFACT: A is factored "from scratch"

   In this case the user must also supply

   - A, the input matrix

   as well as the following options, which are described in more detail below:

   - options->Equil, to specify how to scale the rows and columns of A to "equilibrate" it (to try to reduce its condition number and so improve the accuracy of the computed solution)

   - options->RowPerm, to specify how to permute the rows of A (typically to control numerical stability)

   - options->ColPerm, to specify how to permute the columns of A (typically to control fill-in and enhance parallelism during factorization)

   - options->ReplaceTinyPivot, to specify how to deal with tiny pivots encountered during factorization (to control numerical stability)

   The outputs returned include

   - ScalePermstruct, modified to describe how the input matrix A was equilibrated and permuted:
     - ScalePermstruct->DiagScale, indicates whether the rows and/or columns of A were scaled
     - ScalePermstruct->R, array of row scale factors
     - ScalePermstruct->C, array of column scale factors
     - ScalePermstruct->perm_r, row permutation vector
     - ScalePermstruct->perm_c, column permutation vector

(part of ScalePermstruct may also need to be supplied on input, depending on options->RowPerm and options->ColPerm as described later).

- A, the input matrix A overwritten by the scaled and permuted matrix
  Pc*Pr*diag(R)*A*diag(C)
  where
  Pr and Pc are row and columns permutation matrices determined by ScalePermstruct->perm_r and ScalePermstruct->perm_c, respectively, and
  diag(R) and diag(C) are diagonal scaling matrices determined by ScalePermstruct->DiagScale, ScalePermstruct->R and ScalePermstruct->C

- LUstruct, which contains the L and U factorization of A1 where

  A1 = Pc*Pr*diag(R)*A*diag(C)*Pc^T = L*U

  (Note that A1 = Aout * Pc^T, where Aout is the matrix stored in A on output.)

3. The second value of options->Fact assumes that a matrix with the same sparsity pattern as A has already been factored:

   - options->Fact = SamePattern: A is factored, assuming that it has the same nonzero pattern as a previously factored matrix. In this case the algorithm saves time by reusing the previously computed column permutation vector stored in ScalePermstruct->perm_c and the "elimination tree" of A stored in LUstruct->etree

   In this case the user must still specify the following options as before:

   - options->Equil
   - options->RowPerm
   - options->ReplaceTinyPivot

   but not options->ColPerm, whose value is ignored. This is because the previous column permutation from ScalePermstruct->perm_c is used as input. The user must also supply

   - A, the input matrix
   - ScalePermstruct->perm_c, the column permutation
   - LUstruct->etree, the elimination tree

   The outputs returned include

- A, the input matrix A overwritten by the scaled and permuted matrix as described above
- ScalePermstruct, modified to describe how the input matrix A was equilibrated and row permuted
- LUstruct, modified to contain the new L and U factors

4. The third value of options->Fact assumes that a matrix B with the same sparsity pattern as A has already been factored, and where the row permutation of B can be reused for A. This is useful when A and B have similar numerical values, so that the same row permutation will make both factorizations numerically stable. This lets us reuse all of the previously computed structure of L and U.

   - options->Fact = SamePattern_SameRowPerm: A is factored, assuming not only the same nonzero pattern as the previously factored matrix B, but reusing B's row permutation.

   In this case the user must still specify the following options as before:

   - options->Equil
   - options->ReplaceTinyPivot

   but not options->RowPerm or options->ColPerm, whose values are ignored. This is because the permutations from ScalePermstruct->perm_r and ScalePermstruct->perm_c are used as input.

   The user must also supply

   - A, the input matrix
   - ScalePermstruct->DiagScale, how the previous matrix was row and/or column scaled
   - ScalePermstruct->R, the row scalings of the previous matrix, if any
   - ScalePermstruct->C, the columns scalings of the previous matrix, if any
   - ScalePermstruct->perm_r, the row permutation of the previous matrix
   - ScalePermstruct->perm_c, the column permutation of the previous matrix
   - all of LUstruct, the previously computed information about L and U (the actual numerical values of L and U stored in LUstruct->Llu are ignored)

   The outputs returned include

   - A, the input matrix A overwritten by the scaled and permuted matrix as described above
   - ScalePermstruct, modified to describe how the input matrix A was

```
                    equilibrated
            (thus ScalePermstruct->DiagScale, R and C may be modified)
    -  LUstruct, modified to contain the new L and U factors

  5. The fourth and last value of options->Fact assumes that A is
     identical to a matrix that has already been factored on a previous
     call, and reuses its entire LU factorization

     -  options->Fact = Factored: A is identical to a previously
           factorized matrix, so the entire previous factorization
           can be reused.

     In this case all the other options mentioned above are ignored
     (options->Equil, options->RowPerm, options->ColPerm,
      options->ReplaceTinyPivot)

     The user must also supply

     -  A, the unfactored matrix, only in the case that iterative refinment
           is to be done (specifically A must be the output A from
           the previous call, so that it has been scaled and permuted)
     -  all of ScalePermstruct
     -  all of LUstruct, including the actual numerical values of L and U

     all of which are unmodified on output.


Arguments
=========

options (input) superlu_options_t*
        The structure defines the input parameters to control
        how the LU decomposition will be performed.
        The following fields should be defined for this structure:

        o Fact (fact_t)
          Specifies whether or not the factored form of the matrix
          A is supplied on entry, and if not, how the matrix A should
          be factorized based on the previous history.

          = DOFACT: The matrix A will be factorized from scratch.
                Inputs:  A
                         options->Equil, RowPerm, ColPerm, ReplaceTinyPivot
                Outputs: modified A
                            (possibly row and/or column scaled and/or
                             permuted)
                         all of ScalePermstruct
```

```
                all of LUstruct

  = 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
    ScalePermstruct->perm_c and the elimination tree
    LUstruct->etree
        Inputs:  A
                 options->Equil, RowPerm, ReplaceTinyPivot
                 ScalePermstruct->perm_c
                 LUstruct->etree
        Outputs: modified A
                    (possibly row and/or column scaled and/or
                     permuted)
                 rest of ScalePermstruct (DiagScale, R, C, perm_r)
                 rest of LUstruct (GLU_persist, Llu)

  = SamePattern_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 scaling factors R and C, and the
    both row and column permutation vectors perm_r and perm_c,
    distributed data structure set up from the previous symbolic
    factorization.
        Inputs:  A
                 options->Equil, ReplaceTinyPivot
                 all of ScalePermstruct
                 all of LUstruct
        Outputs: modified A
                    (possibly row and/or column scaled and/or
                     permuted)
                 modified LUstruct->Llu
  = FACTORED: the matrix A is already factored.
        Inputs:  all of ScalePermstruct
                 all of LUstruct

o Equil (yes_no_t)
  Specifies whether to equilibrate the system.
  = NO:  no equilibration.
  = YES: scaling factors are computed to equilibrate the system:
             diag(R)*A*diag(C)*inv(diag(C))*X = diag(R)*B.
         Whether or not the system will be equilibrated depends
         on the scaling of the matrix A, but if equilibration is
         used, A is overwritten by diag(R)*A*diag(C) and B by
         diag(R)*B.
```

```
        o RowPerm (rowperm_t)
          Specifies how to permute rows of the matrix A.
          = NATURAL:   use the natural ordering.
          = LargeDiag: use the Duff/Koster algorithm to permute rows of
                       the original matrix to make the diagonal large
                       relative to the off-diagonal.
          = MY_PERMR:  use the ordering given in ScalePermstruct->perm_r
                       input by the user.

        o ColPerm (colperm_t)
          Specifies what type of column permutation to use to reduce fill.
          = NATURAL:       use the natural ordering.
          = COLAMD:        use approximate minimum degree column ordering.
          = MMD_ATA:       use minimum degree ordering on structure of A'*A.
          = MMD_AT_PLUS_A: use minimum degree ordering on structure of A'+A.
          = MY_PERMC:      use the ordering given in ScalePermstruct->perm_c.

        o ReplaceTinyPivot (yes_no_t)
          = NO:  do not modify pivots
          = YES: replace tiny pivots by sqrt(epsilon)*norm(A) during
                 LU factorization.

        o IterRefine (IterRefine_t)
          Specifies how to perform iterative refinement.
          = NO:     no iterative refinement.
          = DOUBLE: accumulate residual in double precision.
          = EXTRA:  accumulate residual in extra precision.

        NOTE: all options must be indentical on all processes when
              calling this routine.

A (input/output) SuperMatrix*
        On entry, matrix A in A*X=B, of dimension (A->nrow, A->ncol).
        The number of linear equations is A->nrow. The type of A must be:
        Stype = NC; Dtype = _D; Mtype = GE. That is, A is stored in
        compressed column format (also known as Harwell-Boeing format).
        See supermatrix.h for the definition of 'SuperMatrix'.
        This routine only handles square A, however, the LU factorization
        routine pdgstrf_Aglobal can factorize rectangular matrices.
        On exit, A may be overwritten by Pc*Pr*diag(R)*A*diag(C),
        depending on ScalePermstruct->DiagScale, options->RowPerm and
        options->colpem:
            if ScalePermstruct->DiagScale != NOEQUIL, A is overwritten by
               diag(R)*A*diag(C).
            if options->RowPerm != NATURAL, A is further overwritten by
               Pr*diag(R)*A*diag(C).
```

```
        if options->ColPerm != NATURAL, A is further overwritten by
           Pc*Pr*diag(R)*A*diag(C).
    If all the above condition are true, the LU decomposition is
    performed on the matrix Pc*Pr*diag(R)*A*diag(C)*Pc^T.

    NOTE: Currently, A must reside in all processes when calling
          this routine.

ScalePermstruct (input/output) ScalePermstruct_t*
    The data structure to store the scaling and permutation vectors
    describing the transformations performed to the matrix A.
    It contains the following fields:

    o DiagScale (DiagScale_t)
      Specifies the form of equilibration that was done.
      = NOEQUIL: no equilibration.
      = ROW:     row equilibration, i.e., A was premultiplied by
                 diag(R).
      = COL:     Column equilibration, i.e., A was postmultiplied
                 by diag(C).
      = BOTH:    both row and column equilibration, i.e., A was
                 replaced by diag(R)*A*diag(C).
      If options->Fact = FACTORED or SamePattern_SameRowPerm,
      DiagScale is an input argument; otherwise it is an output
      argument.

    o perm_r (int*)
      Row permutation vector, which defines the permutation matrix Pr;
      perm_r[i] = j means row i of A is in position j in Pr*A.
      If options->RowPerm = MY_PERMR, or
      options->Fact = SamePattern_SameRowPerm, perm_r is an
      input argument; otherwise it is an output argument.

    o perm_c (int*)
      Column permutation vector, which defines the
      permutation matrix Pc; perm_c[i] = j means column i of A is
      in position j in A*Pc.
      If options->ColPerm = MY_PERMC or options->Fact = SamePattern
      or options->Fact = SamePattern_SameRowPerm, perm_c is an
      input argument; otherwise, it is an output argument.
      On exit, perm_c may be overwritten by the product of the input
      perm_c and a permutation that postorders the elimination tree
      of Pc*A'*A*Pc'; perm_c is not changed if the elimination tree
      is already in postorder.

    o R (double*) dimension (A->nrow)
      The row scale factors for A.
```

```
        If DiagScale = ROW or BOTH, A is multiplied on the left by
                       diag(R).
        If DiagScale = NOEQUIL or COL, R is not defined.
        If options->Fact = FACTORED or SamePattern_SameRowPerm, R is
        an input argument; otherwise, R is an output argument.

      o C (double*) dimension (A->ncol)
        The column scale factors for A.
        If DiagScale = COL or BOTH, A is multiplied on the right by
                       diag(C).
        If DiagScale = NOEQUIL or ROW, C is not defined.
        If options->Fact = FACTORED or SamePattern_SameRowPerm, C is
        an input argument; otherwise, C is an output argument.

B       (input/output) double*
        On entry, the right-hand side matrix of dimension (A->nrow, nrhs).
        On exit, the solution matrix if info = 0;

        NOTE: Currently, B must reside in all processes when calling
              this routine.

ldb     (input) int (global)
        The leading dimension of matrix B.

nrhs    (input) int (global)
        The number of right-hand sides.
        If nrhs = 0, only LU decomposition is performed, the forward
        and back substitution are skipped.

grid    (input) gridinfo_t*
        The 2D process mesh. It contains the MPI communicator, the number
        of process rows (NPROW), the number of process columns (NPCOL),
        and my process rank. It is an input argument to all the
        parallel routines.
        Grid can be initialized by subroutine SUPERLU_GRIDINIT.
        See superlu_ddefs.h for the definition of 'gridinfo_t'.

LUstruct (input/output) LUstruct_t*
        The data structures to store the distributed L and U factors.
        It contains the following fields:

      o etree (int*) dimension (A->ncol)
        Elimination tree of A'*A, dimension A->ncol.
        It is computed in sp_colorder() during the first factorization,
        and is reused in the subsequent factorizations of the matrices
        with the same nonzero pattern.
        On exit of sp_colorder(), the columns of A are permuted so that
```

```
            the etree is in a certain postorder. This postorder is reflected
            in ScalePermstruct->perm_c.

            NOTE: Etree is a vector of parent pointers for a forest whose
                  vertices are the integers 0 to A->ncol-1;
                  etree[root] = A->ncol.

          o Glu_persist (Glu_persist_t*)
            Global data structure (xsup, supno) replicated on all processes,
            describing the supernode partition in the factored matrices
            L and U:
              xsup[s] is the leading column of the s-th supernode,
              supno[i] is the supernode number to which column i belongs.

          o Llu (LocalLU_t*)
            The distributed data structures to store L and U factors.
            See superlu_ddefs.h for the definition of 'LocalLU_t'.

   berr    (output) double*, dimension (nrhs)
           The componentwise relative backward error of each solution
           vector X(j) (i.e., the smallest relative change in
           any element of A or B that makes X(j) an exact solution).

   stat   (output) SuperLUStat_t*
          Record the statistics on runtime and floating-point operation count.
          See util.h for the definition of 'SuperLUStat_t'.

   info    (output) int*
           = 0: successful exit
           > 0: if info = i, and i is
               <= A->ncol: U(i,i) is exactly zero. The factorization has
                  been completed, but the factor U is exactly singular,
                  so the solution could not be computed.
               > A->ncol: number of bytes allocated when memory allocation
                  failure occurred, plus A->ncol.
```

C.2 pdgstrf

```
void
pdgstrf(superlu_options_t *options, int m, int n, double anorm,
        LUstruct_t *LUstruct, gridinfo_t *grid, SuperLUStat_t *stat, int *info)

   Purpose
   =======

   pdgstrf performs the LU factorization in parallel.
```

```
Arguments
=========

options (input) superlu_options_t*
        The structure defines the input parameters to control
        how the LU decomposition will be performed.
        The following field should be defined:
        o ReplaceTinyPivot (yes_no_t)
          Specifies whether to replace the tiny diagonals by
          sqrt(epsilon)*norm(A) during LU factorization.

m       (input) int
        Number of rows in the matrix.

n       (input) int
        Number of columns in the matrix.

anorm   (input) double
        The norm of the original matrix A, or the scaled A if
        equilibration was done.

LUstruct (input/output) LUstruct_t*
        The data structures to store the distributed L and U factors.
        The following fields should be defined:

        o Glu_persist (input) Glu_persist_t*
          Global data structure (xsup, supno) replicated on all processes,
          describing the supernode partition in the factored matrices
          L and U:
        xsup[s] is the leading column of the s-th supernode,
             supno[i] is the supernode number to which column i belongs.

        o Llu (input/output) LocalLU_t*
          The distributed data structures to store L and U factors.
          See superlu_ddefs.h for the definition of 'LocalLU_t'.

grid    (input) gridinfo_t*
        The 2D process mesh. It contains the MPI communicator, the number
        of process rows (NPROW), the number of process columns (NPCOL),
        and my process rank. It is an input argument to all the
        parallel routines.
        Grid can be initialized by subroutine SUPERLU_GRIDINIT.
        See superlu_ddefs.h for the definition of 'gridinfo_t'.

stat    (output) SuperLUStat_t*
        Record the statistics on runtime and floating-point operation count.
```

```
        See util.h for the definition of 'SuperLUStat_t'.

   info   (output) int*
          = 0: successful exit
          < 0: if info = -i, the i-th argument had an illegal value
          > 0: if info = i, U(i,i) is exactly zero. The factorization has
               been completed, but the factor U is exactly singular,
               and division by zero will occur if it is used to solve a
               system of equations.
```

C.3 pdgstrs_Bglobal

```
void
pdgstrs_Bglobal(int n, LUstruct_t *LUstruct, gridinfo_t *grid, double *B,
int ldb, int nrhs, SuperLUStat_t *stat, int *info)

   Purpose
   =======

   pdgstrs_Bglobal solves a system of distributed linear equations
   A*X = B with a general N-by-N matrix A using the LU factorization
   computed by pdgstrf.

   Arguments
   =========

   n      (input) int (global)
          The order of the system of linear equations.

   LUstruct (input) LUstruct_t*
          The distributed data structures storing L and U factors.
          The L and U factors are obtained from pdgstrf for
          the possibly scaled and permuted matrix A.
          See superlu_ddefs.h for the definition of 'LUstruct_t'.

   grid   (input) gridinfo_t*
          The 2D process mesh. It contains the MPI communicator, the number
          of process rows (NPROW), the number of process columns (NPCOL),
          and my process rank. It is an input argument to all the
          parallel routines.
          Grid can be initialized by subroutine SUPERLU_GRIDINIT.
          See superlu_ddefs.h for the definition of 'gridinfo_t'.

   B      (input/output) double*
          On entry, the right-hand side matrix of the possibly equilibrated
          and row permuted system.
```

```
        On exit, the solution matrix of the possibly equilibrated
        and row permuted system if info = 0;

        NOTE: Currently, the N-by-NRHS  matrix B must reside on all
              processes when calling this routine.

   ldb    (input) int (global)
          Leading dimension of matrix B.

   nrhs   (input) int (global)
          Number of right-hand sides.

   stat   (output) SuperLUStat_t*
          Record the statistics about the triangular solves.
          See util.h for the definition of 'SuperLUStat_t'.

   info   (output) int*
           = 0: successful exit
           < 0: if info = -i, the i-th argument had an illegal value
```

C.4 pdgsrfs_ABXglobal

```
void
pdgsrfs_ABXglobal(int n, SuperMatrix *A, double anorm, LUstruct_t *LUstruct,
                  gridinfo_t *grid, double *B, int ldb, double *X, int ldx,
                  int nrhs, double *berr, SuperLUStat_t *stat, int *info)

   Purpose
   =======

   pdgsrfs_ABXglobal improves the computed solution to a system of linear
   equations and provides error bounds and backward error estimates
   for the solution.

   Arguments
   =========

   n      (input) int (global)
          The order of the system of linear equations.

   A      (input) SuperMatrix*
          The original matrix A, or the scaled A if equilibration was done.
          A is also permuted into the form Pc*Pr*A*Pc', where Pr and Pc
          are permutation matrices. The type of A can be:
          Stype = NCP; Dtype = _D; Mtype = GE.
```

NOTE: Currently, A must reside in all processes when calling
this routine.

anorm (input) double
The norm of the original matrix A, or the scaled A if
equilibration was done.

LUstruct (input) LUstruct_t*
The distributed data structures storing L and U factors.
The L and U factors are obtained from pdgstrf for
the possibly scaled and permuted matrix A.
See superlu_ddefs.h for the definition of 'LUstruct_t'.

grid (input) gridinfo_t*
The 2D process mesh. It contains the MPI communicator, the number
of process rows (NPROW), the number of process columns (NPCOL),
and my process rank. It is an input argument to all the
parallel routines.
Grid can be initialized by subroutine SUPERLU_GRIDINIT.
See superlu_ddefs.h for the definition of 'gridinfo_t'.

B (input) double* (global)
The N-by-NRHS right-hand side matrix of the possibly equilibrated
and row permuted system.

NOTE: Currently, B must reside on all processes when calling
this routine.

ldb (input) int (global)
Leading dimension of matrix B.

X (input/output) double* (global)
On entry, the solution matrix X, as computed by PDGSTRS.
On exit, the improved solution matrix X.
If DiagScale = COL or BOTH, X should be premultiplied by diag(C)
in order to obtain the solution to the original system.

NOTE: Currently, X must reside on all processes when calling
this routine.

ldx (input) int (global)
Leading dimension of matrix X.

nrhs (input) int
Number of right-hand sides.

berr (output) double*, dimension (nrhs)

```
        The componentwise relative backward error of each solution
        vector X(j) (i.e., the smallest relative change in
        any element of A or B that makes X(j) an exact solution).

stat    (output) SuperLUStat_t*
        Record the statistics about the refinement steps.
        See util.h for the definition of SuperLUStat_t.

info    (output) int*
        = 0: successful exit
        < 0: if info = -i, the i-th argument had an illegal value
```