Newly Released Capabilities in Distributed-memory SuperLU Sparse Direct Solver

XIAOYE S. LI, PAUL LIN, and YANG LIU, Lawrence Berkeley National Laboratory, USA
PIYUSH SAO, Oak Ridge National Laboratory, USA

We present the new features available in the recent release of SuperLU_DIST, Version 8.1.1. SuperLU_DIST is a distributed-memory parallel sparse direct solver. The new features include (1) a 3D communication-avoiding algorithm framework which trades off inter-process communication for selective memory duplication, (2) multi-GPU support for both NVIDIA GPUs and AMD GPUs, and (3) mixed precision routines that perform single precision LU factorization and double precision iterative refinement. Apart from the algorithm improvements, we also modernized the software build system to use CMake and Spack package installation tools to simplify the installation procedure. Throughout the paper, we describe in detail the pertinent performance-sensitive parameters associated with each new algorithmic feature, show how they are exposed to the users, and give general guidance of how to set these parameters. We illustrate that the solvers performance both in time and memory can be greatly improved after systematic tuning of the parameters, depending on the input sparse matrix and underlying hardware.

CCS Concepts: • **Mathematics of computing → Solvers**.

Additional Key Words and Phrases: Sparse direct solver, communication-avoiding, GPU, mixed-precision

1 OVERVIEW OF SUPERLU_DIST

SuperLU_DIST [1,2] is a distributed-memory parallel sparse direct solver library for solving large sets of linear equations $AX = B$ [6]. Here A is a square, nonsingular, $n \times n$ sparse matrix, and X and B are dense $n \times nrhs$ matrices, where $nrhs$ is the number of right-hand sides and solution vectors. The matrix A need not be symmetric or definite; indeed, SuperLU_DIST is particularly appropriate for unsymmetric matrices, and it respects both the unsymmetric values as well as the unsymmetric sparsity pattern. The library uses variations of Gaussian elimination (LU factorization) optimized to take advantage of the sparsity of the matrix and modern high performance computer architectures (specifically memory hierarchy and parallelism). It is implemented in ANSI C, using MPI for communication, OpenMP for multithreading, and CUDA (or HIP) for NVIDIA (or AMD) GPUs. The library includes routines to handle both real and complex matrices in single and double precisions, and some functions with mixed precisions. The distributed-memory parallel algorithm consists of the following major steps.

(1) Preprocessing
(2) Sparse LU factorization (SpLU)

[1]https://github.com/xiaoyeli/superlu_dist
[2]https://portal.nersc.gov/project/sparse/superlu/superlu_dist_code_html/index.html

Authors' addresses: Xiaoye S. Li, xsli@lbl.gov; Paul Lin, paullin@lbl.gov; Yang Liu, liuyangzhuan@lbl.gov, Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA, USA, 94720; Piyush Sao, saopk@ornl.gov, Oak Ridge National Laboratory, Oak Ridge, TN, USA, 37830.

0098-3500/2022/12-ART $15.00
https://doi.org/10.1145/3577197

(3) Sparse triangular solutions (SpTRSV)
(4) Iterative refinement (IR) (optional)

The preprocessing in Step (1) transforms the original linear system $Ax = b$ into $\bar{A}x = \bar{b}$, so that the latter one has more favorable numerical properties and sparsity structures. In `SuperLU_DIST`, typically A is first transformed into $\bar{A} = P_c P_r D_r A D_c P_c^T$. Here D_r and D_c are diagonal scaling matrices to equilibrate the system, which tends to reduce condition number of the matrix and avoid over/underflow during Gaussian elimination. P_r and P_c are *permutation matrices*. The role of P_r is to permute rows of the matrix to make diagonal elements large relative to the off-diagonal elements (numerical pivoting). The role of P_c is to permute rows and columns of the matrix to minimize the fill-in in the L and U factors (sparsity reordering). Note that we apply P_c symmetrically so that the large diagonal entries remain on the diagonal. With these transformations, the linear system to be solved is: $(P_c P_r D_r A D_c P_c^T)(P_c D_c^{-1})x = P_c P_r D_r b$. In the software configuration, each transformation can be turned off, or can be achieved with different algorithms. Further algorithm details and user interfaces can be found in [6, 8]. After these transformations, the last preprocessing step is symbolic factorization which computes the distributed nonzero structures of the L and U factors, and distributes the nonzeros of $\bar{A}$ into L and U.

In Step (2), before the new Version-7 release (2021), the distributed memory code had been largely built upon the design in the first `SuperLU_DIST` paper [7]. The main ingredients of the parallel SpLU algorithm are:

- supernodal fan-out (right-looking) based on elimination DAGs,
- static pivoting with possible half-precision perturbations on the diagonal [7],
- 2D logical process arrangement for non-uniform block-cyclic mapping, based on the supernodal block partition, and
- loosely synchronous scheduling with lookahead pipelining [14].

In Step (3), The parallel SpTRSV uses a block-cyclic layout for the L and U matrices as in the results of SpLU. It also uses a message-driven asynchronous and dynamically scheduled algorithm—designed to reduce the communication and latency costs.

In Step (4), the user can optionally invoke a few steps of iterative refinement to improve the solution accuracy. The computational kernels in the IR are SpTRSV and sparse matrix-vector multiplication (SpMV). Before this release, the iterative refinement is performed in the same precision as that of SpLU and SpTRSV.

This release paper focuses on the new capabilities in Steps (2)-(4). These includes the new 3D communication-avoiding algorithm framework (Section 2) , multi-GPU support (Section 3), mixed precision routines (Section 4), and support for new build tools (Appendix C). Throughout the paper, we discuss all the parameters that may influence the code performance. These parameters can be set in a compile-time "options" structure, or by environment variables (with capitalized names), the latter of which take precedence. Section 5 gives a summary of the parameters and some tuning results. In particular, we illustrate that the performance can be greatly improved by using an autotuner GPTune [9] for an optimal parameters setting. Finally, Section 6 summarizes our contributions and gives perspectives of future developments.

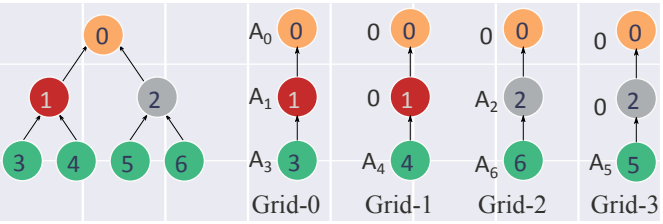
2 3D COMMUNICATION-AVOIDING ROUTINES

We developed a novel 3D algorithm framework for sparse factorization and triangular solutions. This new approach is motivated by the strong scaling requirement from exascale applications. Our novel 3D algorithm framework for sparse factorization and triangular solutions alleviates communication costs by taking advantage of the three-dimensional MPI process grid, the elimination tree parallelism, and the communication-memory tradeoff—inspired from communication-avoiding algorithms for dense linear algebra in the last decade.

The 3D processes grid, configured as $P = P_x \times P_y \times P_z$ (see Fig. 3a), can be considered as P_z sets of 2D processes layers. The distribution of the sparse matrices is governed by the supernodal elimination tree-forest (etree-forest): the standard etree is transformed into an etree-forest which is binary at the top $\log_2(P_z)$ levels and

has P_z subtree-forests at the leaf level (see Fig. 1a). The description of the tree partition and mapping algorithm is described in [12, Section 3.3]. The matrices A, L, and U corresponding to each subtree-forest are assigned to one 2D process layer. The 2D layers are referred to as Grid-0, Grid-1, . . ., up to $(P_z - 1)$ grids. Fig. 1b shows the submatrix mapping to the four 2D process grids.

(a) 2-level etree partition



(b) Matrix view on 4 process grids

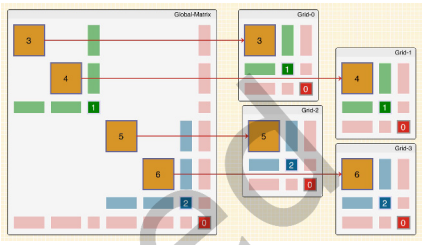


Fig. 1. Illustration of the 3D parallel SpLU algorithm with 4 process grids. Note that, here A_i refers to $A[i{:}, i{:}]$

```
typedef struct {
    MPI_Comm comm;        /* MPI communicator */
    superlu_scope_t rscp; /* row scope */
    superlu_scope_t cscp; /* column scope */
    superlu_scope_t zscp; /* scope in third dimension */
    gridinfo_t grid2d;    /* for using 2D functions */
    int iam;              /* my process number in this grid */
    int nprow;            /* number of process rows */
    int npcol;            /* number of process columns */
    int npdep;            /* number of replication factor in Z-dimension  */
    int rankorder;        /* = 0: Z-major ( default )
                           * = 1: XY-major (need set environment variable: SUPERLU_RANKORDER=XY)
                           */
} gridinfo3d_t;
```

Fig. 2. 3D process grid definition.

An example for calling the 3D algorithm to solve a sparse linear system is provided by the sample program `EXAMPLE/pddrive3d.c`, see https://github.com/xiaoyeli/superlu_dist/blob/master/EXAMPLE/pddrive3d.c. Fig. 2 shows the C structure defining the 3D process grid.

2.1 The 3D Process layout and its performance impact

A 3D process grid can be arranged in two formats: XY-major or Z-major, see Fig. 3. In XY-major format, processes with the same XY-coordinate and different Z-coordinate have consecutive global ranks. Consequently, when spawning multiple processes on a node, the spawned processes will have the same XY coordinate (except for cases where P_z is not a multiple of the number of processes spawned on the node). Alternatively, we can arrange the 3D process grid in Z-major format where processes with the same Z coordinate have consecutive global ranks. This is the default ordering in `SuperLU_DIST`.

The Z-major format can be better for performance as it keeps processes in a 2D grid closer. Hence it may provide higher bandwidth for 2D communication, typically the bottleneck in communication. On the other hand, the XY-major format can be helpful when using GPU acceleration. This can happen since the XY-major ordering will keep more GPUs active during ancestor factorizations. In some cases, e.g., sparse matrices from non-planar graphs, ancestor factorization can become compute dominant, and XY-major ordering helps by keeping more GPUs active. For example, on 16 Haswell nodes of the NERSC Cori Cray XC40, the Z-major ordering was 0.85-1.3× faster than the XY major ordering. Haswell compute nodes have dual-socket 16-core 2.3 GHz Intel Xeon E5-2698v3 CPUs. Note that this performance difference is system-dependent, depending on the hardware topology as well as the job scheduler policy of the parallel machine.

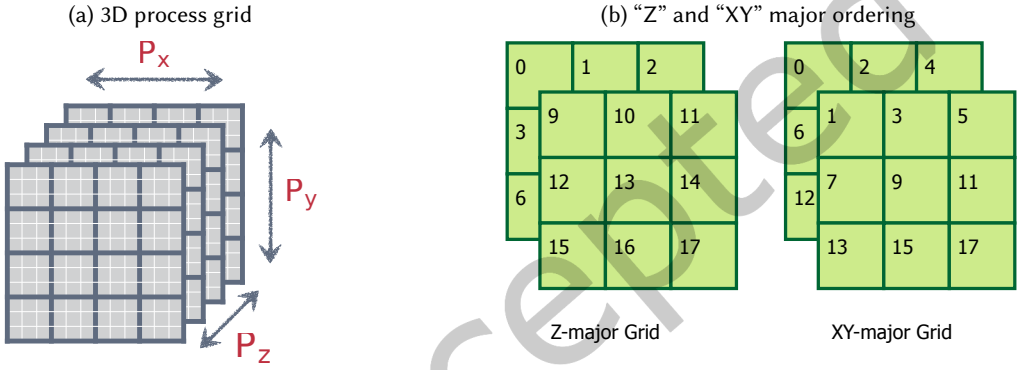


Fig. 3. A logical 3D process grid and process configuration for two types of process arrangements.

The linear solver driver routine is pdgssvx3d, with the calling API explained here: https://portal.nersc.gov/project/sparse/superlu/superlu_dist_code_html/pdgssvx3d_8c.html

The SpLU factorization progresses from leaf level $l = \log_2 P_z$ to the root level 0. The two main phases are local factorization and Ancestor-Reduction.

(1) *Local factorization.* In parallel and independently, every 2D process grid performs the 2D factorization of its locally owned submatrix of A. This is the same algorithm as the one before Version-7 [14]. The only difference is that each process grid will generate a partial Schur complement update, which will be summed up with the partial updates from the other process grids in the next phase.

(2) *Ancestor-Reduction.* After the factorization of level-i, we reduce the partial Schur complement of the ancestor nodes before factorizing the next level. In the i-th level's reduction, the receiver is the $k2^{l-i+1}$-th process grid and the sender is the $(2k+1)2^{l-i}$-th process grid, for some integer k. The process in the 2D grid which owns a block $A_{i,j}$ has the same (x,y) coordinate in both sender and receiver grids. So communication in the ancestor-reduction step is point-to-point pair-wise and takes places along the z-axis in the 3D process grid.

We analyzed the asymptotic improvements for planar graphs (e.g., those arising from 2D grid or mesh discretizations) and certain non-planar graphs (specifically for 3D grids and meshes). For a planar graph with n vertices, our algorithm reduces communication volume asymptotically in n by a factor of $O\left(\sqrt{\log n}\right)$ and latency by a factor of $O(\log n)$. For non-planar cases, our algorithm can reduce the per-process communication volume by 3× and latency by $O\left(n^{\frac{1}{3}}\right)$ times. In all cases, the extra memory needed to achieve these gains is a small constant factor of the L and U memory. We implemented our algorithm by extending the 2D data structure

used in SuperLU. Our new 3D code achieves empirical speedups up to 27× for planar graphs and up to 3.3× for non-planar graphs over the baseline 2D SuperLU when run on 24,000 cores of a Cray XC30 (Edison at NERSC). Please see [12] for comprehensive performance tests with a variety of real-world sparse matrices.

Remark. The algorithm structure requires that the z-dimension of the 3D process grid P_z must be a power-of-two integer. There is no restriction on the shape of the 2D grid P_x and P_y. The rule of thumb is to define it as square as possible. When square grid is not possible, it is better to set the row dimension P_x slightly smaller than the column dimension P_y. For example, the following are good options for the 2D grid: 2x3, 2x4, 4x4, 4x8.

Inter-grid Load-balancing in the 3D SpLU Algorithm. The 3D algorithm provides two strategies for partitioning the elimination tree to balance the load between different 2D grids. The SUPERLU_LBS environment variable specifies which one to use.

- **Nested Dissection (ND)** strategy uses the partitioning provided by a nested dissection ordering. It works well for regular grids. The ND strategy can only be used when the elimination tree is binary, i.e., when the column order is also ND, and it cannot handle cases where the separator tree has nodes with more than two children.
- **Greedy Heuristic (GD)** strategy uses a greedy algorithm to divide one level of the elimination tree. It seeks to minimize the maximum load imbalance among the children of that node; if the imbalance in children is higher than 20%, it further subdivides the largest child until the imbalance falls below 20%. The GD strategy works well for arbitrary column ordering and can handle irregular graphs; however, if it is used on heavily imbalanced trees, it leads to bigger ancestor sizes and, therefore, more memory than ND. GD strategy is the default strategy unless SUPERLU_LBS=ND is specified.

In summary, two parameters are specific to the 3D SpLU algorithm:

- superlu_rankorder (SUPERLU_RANKORDER) defines the arrangement of the 3D process grid (default is Z-major);
- superlu_lbs (SUPERLU_LBS) defines the inter-grid load-balancing strategy (default is GD).

3 GPU-ENABLED ROUTINES

In the current release, the SpLU factorization routines can offload certain computations to GPUs, which is mostly in each Schur complement update step. We support both NVIDIA and AMD GPUs. We are actively developing code for the Intel GPUs. To enable GPU offloading, first a compile-time CMake variable needs to be defined: -DTPL_ENABLE_CUDALIB=TRUE (for NVIDIA GPU with CUDA programming) or -DTPL_ENABLE_HIPLIB=TRUE (for AMD GPU with HIP programming). Then, a runtime environment variable SUPERLU_ACC_OFFLOAD is used to control whether to use GPU or not. By default, SUPERLU_ACC_OFFLOAD=1 is set. ('ACC' denotes ACCelerator.)

3.1 2D SpLU GPU algorithm and tuning parameters

The first sparse LU factorization algorithm capable of offloading the matrix-matrix multiplication to the GPU was published in [11]. The panel factorization and the Gather/Scatter operations are performed on the CPU. This algorithm has been available since SuperLU_DIST version 4.0 of the code (October 2014); however, many users are uncertain about using it correctly due to limited documentation. This section provides a gentle introduction to GPU acceleration in SuperLU_DIST and its performance tuning.

Performing the Schur complement update requires some temporary storage to hold dense blocks. In an earlier algorithm, at each elimination step, the Schur complement update is performed block by block. After performing updates on a block, the temporary storage can be reused for the next block. A conspicuous advantage of this approach is its memory efficiency, since the temporary storage required is bounded by maximum block size. The maximum block size is a tunable parameter that trades off local performance of matrix-matrix multiplication

(GEMM) with inter-process parallelism. A typical setting for the maximum block size is 512 (or smaller). However, a noticeable disadvantage of this approach is that it fails to fully utilize the abundance of local fine-grained parallelism provided by GPUs because each GEMM is too small.

In [11], we modified the algorithm in the Schur complement update step. At each step k, we first copy the individual blocks (in skyline storage) in the kth block row of U into a consecutive buffer $\mathtt{U}(k,:)$. The $L(:,k)$ is already in consecutive storage thanks to the supernodal structure. We then perform a single GEMM call to compute $V \leftarrow L(:,k) \times U(k,:)$. The matrix V is preallocated and the size of V needs to be sufficiently large to achieve close to peak GEMM performance. If the size of $L(:,k) \times U(k,:)$ is larger than V, then we partition the product into several large chunks such that each chunk requires temporary storage smaller than V. Given that modern GPUs have considerably more memory than earlier generations, this extra memory can enable a much faster runtime.

Now, each step of the Schur complement update consists of the following substeps:

(1) Gather sparse blocks $U(k,:)$ into a dense BLAS compliant buffer $\mathtt{U}(k,:)$;
(2) Call dense GEMM $V \leftarrow L(:,k) \times \mathtt{U}(k,:)$ (leading part on CPU, trailing part on GPU); and
(3) Scatter $V[\,]$ into the remaining $(k{+}1:N, k{+}1:N)$ sparse L and U blocks.

It should be noted that the Scatter operation can require indirect memory access, and therefore, it can be as expensive as the GEMM cost. The Gather operation, however, has a relatively low overhead compared to other steps involved. The GEMM offload algorithm tries to hide the overhead of Scatter and data transfer between the CPU and GPU via software pipelining. Here, we discuss the key algorithmic aspects of the GEMM offload algorithm:

- To keep both the CPU and GPU busy, we divide the $\mathtt{U}(k,:)$ into a CPU part and GPU part, so that the GEMM call is split into [cpu : gpu] parts: $L(:,k) \times \mathtt{U}(k,[cpu])$ and $L(:,k) \times \mathtt{U}(k,[gpu])$. To hide the data transfer cost, the algorithm further divides the GEMM into multiple streams. Each stream performs its own sequence of operations: CPU-to-GPU transfer, GEMM, and GPU-to-CPU transfer. Between these streams, these operations are asynchronous. The GPU matrix multiplication is also pipelined with the Scatter operation performed on the CPU.
- To offset the memory limitation on the GPU, we devised an algorithm to divide the Schur complement update into smaller chunks as $\{[cpu:gpu]_1 \mid [cpu:gpu]_2 \mid \dots\}$. These chunks depend on the available memory on the GPU and can be sized by the user. A smaller chunk size will result in many iterations of the loop.

There are three environment variables that can be used to control the memory usage and performance in the GEMM offload algorithm:

- `superlu_n_gemm` (`SUPERLU_N_GEMM`) is the minimum value of the product mnk for a GEMM call to be worth offloading to the GPU (default is 5000);
- `superlu_num_gpu_streams` (`SUPERLU_NUM_GPU_STREAMS`) defines the number of GPU streams to use (default is 8); and
- `superlu_max_buffer_size` (`SUPERLU_MAX_BUFFER_SIZE`) defines the maximum buffer size on GPU that can hold the GEMM output matrix V (default is 256M in floating-point words).

This simple GEMM offload algorithm has limited performance gains. We observed a roughly 2-3× speedup over the CPU-only code for a range of sparse matrices.

3.2 3D SpLU GPU algorithm and tuning parameters

We extend the 3D algorithm for heterogeneous architectures by adding the Highly Asynchronous Lazy Offload (Halo) algorithm for co-processor offload [10]. Compared to the GPU algorithm in the 2D code (Section 3.1),

this algorithm also offloads the Scatter operations of each Schur complement update step to the GPU (in addition to the GEMM call).

On 4096 nodes of a Cray XK7 (Titan at ORNL) with 32,768 CPU cores and 4096 Nvidia K20x GPUs, the 3D algorithm achieves empirical speedups up to 24× for planar graphs and 3.5× for non-planar graphs over the baseline 2D SuperLU with co-processor acceleration.

The performance related parameters are:

- `superlu_num_lookaheads` (`SUPERLU_NUM_LOOKAHEADS`), number of lookahead levels in the Schur-complement update (default is 10).
  In order to reduce the critical path of the sequence of panel factorizations, we devised a *software pipelining* method to overlap the panel factorization of the processes at step $k + 1$ with the Schur complement update of the other processes at step k. When there are multiple remaining supernodes in the Schur complement, the lookahead window (i.e., pipeline depth) can be greater than 1 [14]. This environment variable defines the width of the lookahead window.
- `superlu_mpi_process_per_gpu` (`MPI_PROCESS_PER_GPU`) (default is 1).
  The Halo algorithm uses GPU memory based on its availability. To do this correctly, the algorithm needs to know how many MPI processes are running on a GPU, which can be difficult to determine on some systems. This environment variable can be set to inform SuperLU_DIST that there are N ranks on each GPU so that it can limit its memory usage of each GPU to 90% of available memory shared among all MPI processes, which will, in turn, limit the amount of memory used by each rank.

3.3 2D SpTRSV GPU algorithm

When the 2D grid has one MPI rank, SpTRSV in SuperLU_DIST is parallelized using OpenMP for shared-memory processors and CUDA or HIP for GPUs. Both versions of the implementations are based on an asynchronous level-set traversal algorithm that distributes the computation workload across CPU threads and GPU threads/blocks [4]. The CPU implementation uses OpenMP taskloops and tasks for dynamic scheduling, while the GPU implementation relies on static scheduling. Fig. 4a shows the performance of SpTRSV (L and U solves) on 1 Cori Haswell node with 1 and 8 OpenMP threads with a number of matrices. The speedup ranges between 1.4 and 4.3.

Fig. 4b shows the performance of L-solve using SuperLU_DIST (8 ORNL Summit IBM POWER9 CPU cores or 1 Summit V100 GPU) and cuSPARSE (1 Summit V100 GPU). The GPU SpTRSV in SuperLU_DIST consistently outperforms cuSPARSE and is comparable to the 8-core CPU results. Here we choose 8 CPU cores as there are on average 7 CPU cores per GPU on Summit, and 8 is the closest power of 2 number. Note that GPU performance of the U-solve requires major improvements and is not available in the current release. That said, we compare the performance of SpTRSV (both L and U solves) on one Summit node using three configurations: 1. (baseline) 1-core L solve and 1-core U solve, 2. (GPU) 1-GPU L solve and 1-core U solve, and 3. (GPU+OpenMP) 1-GPU L solve and 8-core U solve. The speedups comparing to the baseline configuration are shown in Table 1.

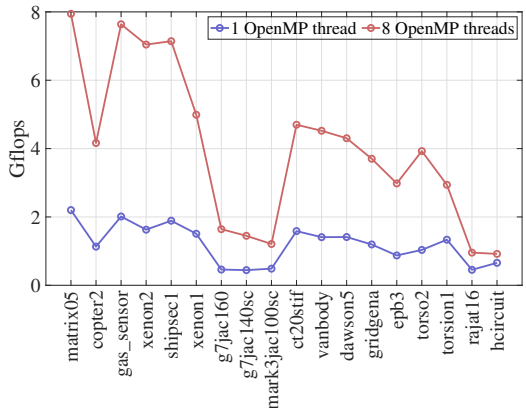
When the 2D grid has more than 1 MPI rank, SpTRSV also supports OpenMP parallelism with less speedups. In addition, the multi-GPU SpTRSV in SuperLU_DIST is under active development and will be available in future releases.

The number of OpenMP threads can be controlled by the environment variable OMP_NUM_THREADS, and the GPU SpTRSV can be turned on with the -DGPU_SOLVE compiler flag. *The user needs to make sure that only 1 MPI rank is used for the 2D grid when GPU SpTRSV is employed.*

4 MIXED-PRECISION ROUTINES

SuperLU_DIST has long supported four distinct floating-point types: IEEE FP32 real and complex, IEEE FP64 real and complex. Furthermore, the library allows all four datatypes to be used together in the same application.

(a) L and U solve (in Mflops) with 1 and 8 OpenMP threads on Cori Haswell



(b) L solve of SuperLU and cuSparse on CPU and GPU

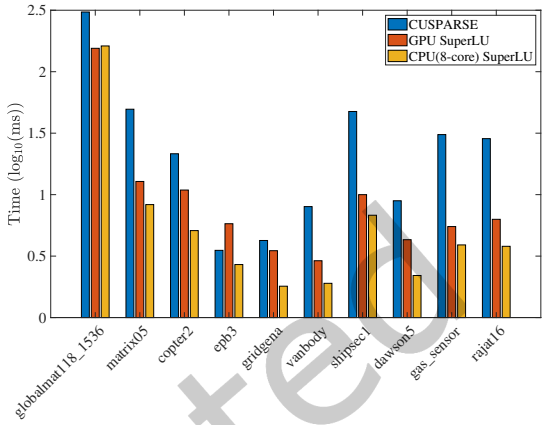


Fig. 4. Performance of SpTRSV with 1 MPI rank for a variety of sparse matrices.

	copter2	epb3	gridgena	vanbody	shipsec1	dawson5
GPU vs. Baseline	1.6	1.7	1.6	1.6	1.54	1.6
GPU+OpenMP vs. Baseline	5.3	5.7	5.3	4.4	4.1	5.2

Table 1. Speedup of GPU SpTRSV compared with sequential CPU SpTRSV.

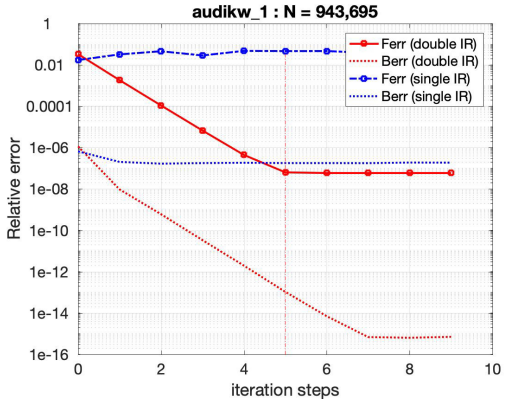
Recent hardware trends have motivated increased development of *mixed-precision* numerical libraries, mainly because hardware vendors have started designing special-purpose units for low precision arithmetic with higher speed. For direct linear solvers, a well understood method is to use lower precision to perform factorization (expensive) and higher precision to perform iterative refinement (IR) to recover accuracy (cheap). For a typical sparse matrix resulting from the 3D finite difference discretization of a regular mesh, the SpLU needs $O\left(n^2\right)$ flops while each IR step needs only $O\left(n^{4/3}\right)$ flops (including SpTRSV and SpMV).

For dense LU and QR factorizations, the benefit of lower precision format comes mainly from accelerated GEMM speed. But in the sparse case, the dimensions of the GEMM are generally smaller and of non-uniform size throughout factorization. Therefore, the speed gain from GEMM alone is limited. In addition to GEMM, a nontrivial cost is the Scatter operation. From our tests, we found that the fraction of time in GEMM usually is less than 50%. Because of this, the TensorCore version of GEMM calls led to a less than 5% speedup for the whole SpLU. When comparing FP32 with the FP64 versions of SpLU, we observed about 50% speedup with the FP32 version.

We recall the IR algorithm using three precisions in Algorithm 1 [2, 3]. This algorithm is already available as **x**GERFSX functions in LAPACK, where the input matrix is dense and so is LU. The following three precisions may be used:

- ε_w is the working precision; it is used for the input data A and b, and output x.
- ε_x is the precision for the computed solution $x^{(i)}$. We require $\varepsilon_x \leq \varepsilon_w$, possibly $\varepsilon_x \leq \varepsilon_w^2$ if necessary for componentwise convergence.
- ε_r is the precision for the residuals $r^{(i)}$. We usually have $\varepsilon_r \leq \varepsilon_w^2$, i.e., at least twice the working precision.

(a) audikw_1 convergence history



(b) Ga19As19H2 convergence history

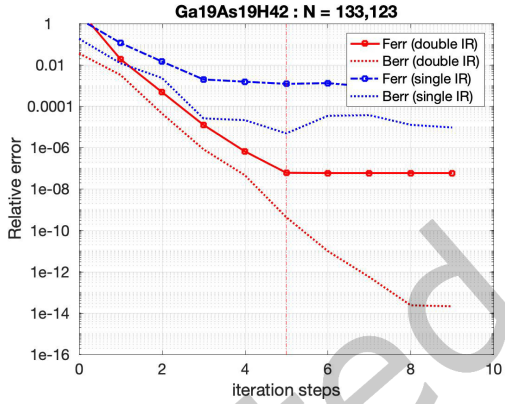


Fig. 5. Convergence history of Algorithm 1 when applied to two sparse linear systems. The vertical line in each plot corresponds to the IR steps taken when our stopping criteria are satisfied. Here, the working precision is $\varepsilon_w = 2^{-24}$. The blue lines are all single. The red lines correspond to $\varepsilon_x = \varepsilon_r = 2^{-53}$, but factorization is in single.

Algorithm 1 Three-precisions Iterative Refinement (IR) for Direct Linear Solvers

1: Solve $Ax^{(1)} = b$ using the basic solution method (e.g., LU or QR) ▷ (ε_w)
2: $i = 1$
3: **repeat**
4: $\quad r^{(i)} \leftarrow b - Ax^{(i)}$ ▷ (ε_r)
5: $\quad$ Solve $A\,dx^{(i+1)} = r^{(i)}$ using the basic solution method ▷ (ε_w)
6: $\quad$ Update $x^{(i+1)} \leftarrow x^{(i)} + dx^{(i+1)}$ ▷ (ε_x)
7: $\quad i \leftarrow i + 1$
8: **until** $x^{(i)}$ is "accurate enough"
9: **return** $x^{(i)}$ and error bounds

Algorithm 1 converges with small normwise (or componentwise) error and error bound if the normwise (or componentwise) condition number of A does not exceed $1/(\gamma\varepsilon_w)$, where $\gamma \stackrel{\text{def}}{=} \sqrt{max_i(nnz(A(i,:)))}$. Moreover, this IR procedure can return to the user both normwise and componentwise reliable error bounds. The error analysis in [2] should carry through to the sparse cases.

We implemented Algorithm 1 in SuperLU_DIST, using two precisions in IR:

- $\varepsilon_w = 2^{-24}$ (IEEE-754 single precision), $\varepsilon_x = \varepsilon_r = 2^{-53}$ (IEEE-754 double precision)

In Figure 5, the left two plots show the convergence history of two systems, in both normwise forward and backward errors, F_{err} and B_{err}, respectively (defined below). We perform two experiments: one using single precision IR, the other using double precision IR. As can be seen, single precision IR does not reduce much F_{err}, while double precision *IR* delivers F_{err} close to ε_w. The IR time is usually under 10% of the factorization time. Overall, the mixed-precision speed is still faster than using pure FP64, see Table 2.

The 2D driver routine for this mixed-precision approach is psgssvx_d2, where the suffix "d2" denotes that the intermediate x vector and r vector internal to the IR routine are carried in double precision.

Table 2. Parallel solution time (seconds) (including SpLU and IR): purely double precision, purely single precision, and mixed precision (FP32 SpLU + FP64 IR). ORNL Summit using up to 8 nodes, each node uses 6 CPU Cores (C) and 6 GPUs (G).

Matrix	Precision	6 C+G	24 C+G	48 C+G	Matrix	Precision	6 C+G	24 C+G	48 C+G
audikw_1	Double	65.9	21.1	18.9	Ga19As19H42	Double	310.9	62.4	34.3
	Single	45.8	13.8	10.5		Single	258.1	48.2	25.8
	Mixed	49.2	13.9	11.4		Mixed	262.8	48.8	26.1

The only difference from the one-precision routine `psgssvx` is the output array `err_bounds[]` (error bounds). For each right-hand side, we return the following three quantities:

- `err_bounds[0]`: normwise forward error bound: $B_{norm} = \max\left(\frac{\|dx^{(i+1)}\|_\infty/\|x^{(i)}\|_\infty}{1-\rho_{max}}, \gamma\varepsilon_w\right) \approx \frac{\|x^{(i)}-x\|_\infty}{\|x\|_\infty}$
  where, $\rho_{\max} \stackrel{\text{def}}{=} \max_{j\le i} \frac{\|dx^{(j+1)}\|_\infty}{\|dx^{(j)}\|_\infty}$ is the estimate of the convergence rate of $x^{(i)}$.
- `err_bounds[1]`: componentwise forward error bound: $\max\left(\frac{\|C^{-1}dx^{(i)}\|_\infty}{1-\hat{\rho}_{max}}, \gamma\varepsilon_w\right) \approx \max_k \left|\frac{x_k^{(i)}-x_k}{x_k}\right|$
  where, $C = diag(x)$, $\hat{\rho}_{\max} = \max_{j\le i} \frac{\|Cdx^{(j+1)}\|_\infty}{\|Cdx^{(j)}\|_\infty}$ is the estimate of the convergence rate of $C^{-1}x^{(i)}$
- `err_bounds[2]`: componentwise backward error: $\max_k \left(\frac{|b-Ax^{(i)}|_k}{(|A||x^{(i)}|+|b|)_k}\right)$

5 SUMMARY OF TUNING PARAMETERS AND TUNING RESULTS

Throughout all phases of the solution process, a number of algorithm parameters can influence the solver's performance. These parameters can be modified by the user. For each user-callable routine, the first argument is usually an input "`options`" argument, which points to the structure containing a number of algorithm choices. These choices are determined at compile time. The second column in Table 3 lists the named fields in the `options` argument. The fourth column lists all the possible values and their corresponding C's enumerated constant names. The user should call the following routine to set up the default values.

```
superlu_dist_options_t options;
set_default_options_dist(&options);
```

After setting the defaults, the user can modify each default, for example:

```
options.RowPerm = LargeDiag_HWPM;
```

For a subset of these parameters, the user can change them at runtime via environment variables. These parameters are listed in the third column in Table 3. At various places of the code, an environment inquiry function SRC/sp_ienv.c is called to retrieve the values of the environment variables.

Two algorithm blocking parameters can be changed at runtime: `SUPERLU_MAXSUP` and `SUPERLU_RELAX`. `SUPERLU_MAXSUP` sets the maximum size of a supernode. That is, if the number of columns in a supernode exceeds this value, we will split this supernode into two supernodes. Setting this parameter to a large value results in larger blocks and generally better performance for threaded and GPU GEMM. Increasing it limits the number of available parallel tasks across MPI processes. Figure 6a illustrates how performance, as measured in Gflops, varies with `SUPERLU_MAXSUP` on a single node of Cori Haswell when using 32 OpenMP threads. For smaller matrices, such as this one (torso3), performance is near its peak when `SUPERLU_MAXSUP` equals 128, which is over 50× faster than when this value is set to 4. However, above this value, the performance starts to taper off.

`SUPERLU_RELAX` is a relaxation parameter: if the number of nodes (columns) in a subtree of the elimination tree is less than this value, this subtree is treated as one supernode, regardless of the row structures. That means, we pad explicit zeros to enforce that all the columns within this relaxed supernode have the same row structure. The advantage of this padding is to mitigate many small supernodes at the bottom of the elimination tree. On

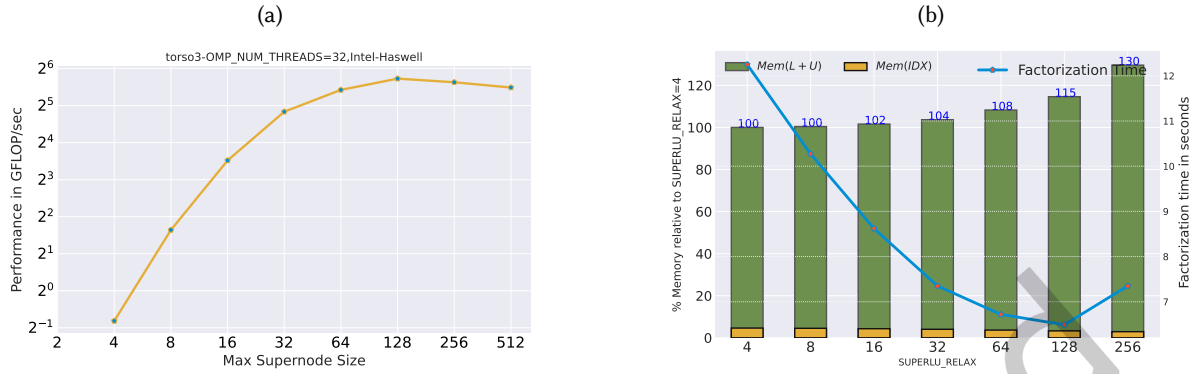


Fig. 6. Impact of maximum supernode size (SUPERLU_MAXSUP) and supernodal relaxation (SUPERLU_RELAX) on performance and memory. The machine is NERSC Cori Haswell node. The matrix is torso3 from SuiteSparse.

the other hand, a large value of SUPERLU_RELAX may introduce too many zeros which in turn propagate to the ancestors of the elimination tree, resulting in a large number of fill-ins in the L and U factors. Figure 6b shows the impact of this parameter on the memory use (left axis) and factorization time. A value of 32 or 64 represents a good tradeoff between memory and time.

The optimal settings of these parameters are matrix-dependent and hardware-dependent. Additionally, several other parameters and environment variables listed in Table 3 are performance critical for the 2D and 3D, CPU and GPU algorithms described in Sections 2, 3.1 and 3.2. It is a daunting task for manual tuning to find the optimal setting of these parameters. Now in Sections 5.1 - 5.3 we show how an autotuner can significantly simplify this task. Here we leverage an autotuner called GPTune [9] to tune the performance (time and memory) of SpLU. We consider two example matrices from the Suitesparse matrix collection, G3_circuit from circuit simulation and H2O from quantum chemistry simulation. For all the experiments, we consider a two-objective tuning scenario and generate a Pareto front from the samples demonstrating the tradeoff between memory and CPU requirement of SpLU.

5.1 3D CPU SpLU parameter tuning

For the 3D CPU SpLU algorithm (2), we use 16 NERSC Cori Haswell nodes and the G3_circuit matrix. The number of OpenMP threads is set to 1, so there are a total of $P_x P_y P_z = 512$ MPI ranks. We consider the following tuning parameters [SUPERLU_MAXSUP, SUPERLU_RELAX, num_lookaheads, P_x, P_z]. We set up GPTune to generate 100 samples. All samples and the Pareto front are plotted in Fig. 7a. The samples on the Pareto front and the default one are shown in Table 4, one can clearly see that by reducing the computation granularity (SUPERLU_MAXSUP, SUPERLU_RELAX) and increasing P_z, one can significantly improve the SpLU time while using slightly more memory.

5.2 2D GPU SpLU parameter tuning

For the 2D GPU SpLU algorithm (3.1), we use 2 NERSC Cray EX Perlmutter GPU compute nodes with 4 MPI ranks per node and the H2O matrix. Perlmutter GPU compute nodes consist of a single 64-core 2.45 GHz AMD EPYC 7763 CPU and four NVIDIA A100 (40GB HBM2) GPUs. The number of OpenMP threads is set to 16. We consider the following tuning parameters [ColPerm, SUPERLU_MAXSUP, SUPERLU_RELAX, SUPERLU_N_GEMM , SUPERLU_MAX_BUFFER_SIZE, P_x]. We set up GPTune to generate 100 samples. All samples and the Pareto front

Table 3. List of algorithm parameters used in various phases of the linear solver. The third column lists the environment variables that can be reset at runtime. parameters must be set in the `options{}` structure input to a driver routine.

phase	`options` **(compile-time)**	env variables **(runtime)**	values (enum constants)	in 2D or 3D algo.
Pre-process	Equil		NO, YES (default)	2d, 3d
	RowPerm		0: NOROWPERM	2d, 3d
			1: `LargeDiag_MC64` (default)	2d, 3d
			2: `LargeDiag_HWPM`	2d, 3d
			3: `MY_PERMR`	2d, 3d
	ColPerm		0: NATURAL	2d, 3d
			1: `MMD_ATA`	2d, 3d
			2: `MMD_AT_PLUS_A`	2d, 3d
			3: COLAMD	2d, 3d
			4: `METIS_AT_PLUS_A` (default)	2d, 3d
			5: PARMETIS	2d, 3d
			6: ZOLTAN	2d, 3d
			7: `MY_PERMC`	2d, 3d
	ParSymbFact		YES, NO (default)	2d, 3d
SpLU	ReplaceTinyPivot		YES, NO (default)	2d, 3d
	Algo3d		YES, NO (default)	3d
	DiagInv		YES, NO (default)	2d
	num_lookaheads	`SUPERLU_NUM_LOOKAHEADS`	default 10	2d, 3d (Section 3.2)
	superlu_maxsup	`SUPERLU_MAXSUP`	default 256	2d, 3d (Section 5)
	superlu_relax	`SUPERLU_RELAX`	default 60	2d, 3d
	superlu_rankorder	`SUPERLU_RANKORDER`	default Z-major	3d (Section 2.1)
	superlu_lbs	`SUPERLU_LBS`	default GD	3d (Section 2.1)
	superlu_acc_offload	`SUPERLU_ACC_OFFLOAD`	0, 1 (default)	2d, 3d (Section 3)
	superlu_n_gemm	`SUPERLU_N_GEMM`	default 5000	2d (Section 3.1)
	superlu_max_buffer_size	`SUPERLU_MAX_BUFFER_SIZE`	default 250M words	2d, 3d (Section 3.1)
	superlu_num_gpu_streams	`SUPERLU_NUM_GPU_STREAMS`	default 8	2d (Section 3.1)
	superlu_mpi_process_per_gpu	`SUPERLU_MPI_PROCESS_PER_GPU`	default 1	3d (Section 3.2)
		`OMP_NUM_THREADS`	default system dependent	2d, 3d (Section 5.4)
		`OMP_PLACES`	default system dependent	2d, 3d
		`OMP_PROC_BIND`	default system dependent	2d, 3d
		`OMP_NESTED`	default system dependent	2d, 3d
		`OMP_DYNAMIC`	default system dependent	2d, 3d
SpTRSV	IterRefine (Section 4)		0: NOREFINE (default)	2d, 3d
			1: `SLU_SINGLE`	
			2: `SLU_DOUBLE`	
Others	PrintStat		NO, YES (default)	2d, 3d

are plotted in Fig. 7b. The samples on the Pareto front and the default one are shown in Table 5. Compared to the default configuration, both the time and memory can be significantly improved by increasing the computation granularity (larger `SUPERLU_MAXSUP`, `SUPERLU_RELAX`). Also, less GPU offload (larger `SUPERLU_N_GEMM`) leads to better performance.

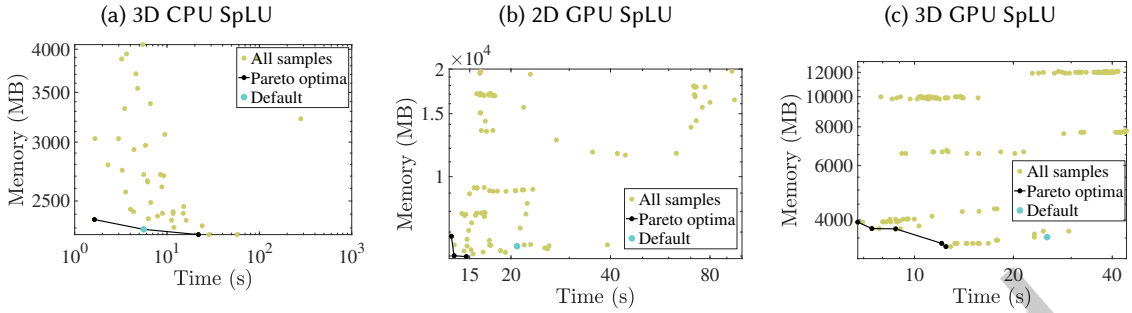


Fig. 7. Samples generated by GPTune for the three tuning experiments. Only valid samples are plotted.

5.3 3D GPU SpLU parameter tuning

For the 3D GPU SpLU algorithm in Section 3.2, we use 2 NERSC Perlmutter GPU nodes with 4 MPI ranks per node and the H2O matrix. The number of OpenMP threads is set to 16, and $P_x P_y P_z = 8$. We consider the following tuning parameters [ColPerm, SUPERLU_MAXSUP, SUPERLU_RELAX, SUPERLU_MAX_BUFFER_SIZE, P_x, P_z]. We set up GPTune to generate 200 samples. All samples and the Pareto front are plotted in Fig. 7c. The samples on the Pareto front and the default one are shown in Table 6. Compared to the default configuration, both the time and memory utilization can be significantly improved by increasing the computation granularity and decreasing GPU buffer sizes. ColPerm='4' (METIS_AT_PLUS_A) is always preferable in terms of memory usage. The effects of P_x and P_z are insignificant as only 8 MPI ranks are used.

	SUPERLU_MAXSUP	SUPERLU_RELAX	num_lookaheads	P_x	P_z	Time (s)	Memory (MB)
Default	256	60	10	16	1	5.6	2290
Tuned	31	25	17	16	1	21.9	2253
Tuned	53	35	7	4	4	1.64	2360

Table 4. Default and optimal samples returned by GPTune for the 3D CPU SpLU algorithm. Note that P_y is derived by $P_y = 512/(P_x P_z)$, as the total MPI count is fixed at 512.

	ColPerm	SUPERLU_MAXSUP	SUPERLU_RELAX	SUPERLU_N_GEMM	SUPERLU_MAX_BUFFER_SIZE	P_x	Time (s)	Memory (MB)
Default	'4'	256	60	1000	2.5E8	4	20.8	6393
Tuned	'4'	154	154	2048	2.68E8	2	13.5	6011
Tuned	'4'	345	198	262144	6.7E7	2	13.2	6813
Tuned	'4'	124	110	8192	1.3E8	2	14.6	5976

Table 5. Default and optimal samples returned by GPTune for the 2D GPU SpLU algorithm. Note that P_y is derived by $P_y = 8/P_x$, as the total MPI count is fixed at 8.

	ColPerm	SUPERLU_MAXSUP	SUPERLU_RELAX	SUPERLU_MAX_BUFFER_SIZE	P_x	P_z	Time (s)	Memory (MB)
Default	'4'	256	60	2.5E8	4	1	25.3	3520
Tuned	'4'	176	143	1.34E8	2	1	12.1	3360
Tuned	'4'	327	182	1.34E8	4	2	7.4	3752
Tuned	'4'	610	200	3.34E7	8	1	12.5	3280
Tuned	'4'	404	187	3.34E7	1	2	8.76	3744
Tuned	'4'	232	199	3.34E7	4	2	6.7	3936

Table 6. Default and optimal samples returned by GPTune for the 3D GPU SpLU algorithm. Note that P_y is calculated from P_x and P_z as the total MPI count is fixed at 8.

5.4 Tuning of OpenMP Intra-node Parallelism

SuperLU_DIST can use shared-memory parallelism on CPUs in two ways. The first is by using the multithreaded BLAS library for linear-algebraic operations. This is independent of the implementation of SuperLU_DIST itself. The second is through SuperLU_DIST's direct use of OpenMP pragmas for explicit parallelization of some computations.

OpenMP is portable across a wide variety of CPU architectures and operating systems. OpenMP offers a shared-memory programming model, which can be easier to use than a message-passing programming model. In this section, we discuss the advantages and limitations of using OpenMP, and offer some performance considerations.

Advantage of OpenMP Parallelism: We have empirically observed that hybrid programming with MPI+OpenMP often requires less memory than pure MPI. This is because OpenMP does not require additional memory for message passing buffers. In most cases, correctly tuned hybrid programming with MPI+OpenMP provides better performance than pure MPI.

Limitations of OpenMP Parallelism: Benefits of OpenMP parallelism is often less predictable than pure MPI parallelism because of non-determinism in the threading layer, CPU hardware, and thread affinities. Finding the right configuration for OpenMP may take some trials and errors because performance depends on many factors: CPU architecture, number of cores and threads, the threading library being used, and the operating system. OpenMP threading may cause a significant slowdown if parameters are chosen incorrectly. Slowdown can be due to false-sharing, NUMA effects, hyperthreading, incorrect or suboptimal thread affinities, or underlying threading libraries.

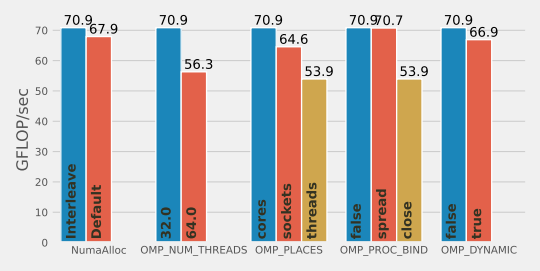
OpenMP performance tuning: To get the best performance, we recommend tuning the following OpenMP variables environment variables. OMP_NUM_THREADS governs the number of OpenMP threads that SuperLU_DIST can use. To avoid resource over-subscription, the product of MPI processes per node and OpenMP threads should be less than or equal to available physical cores. OMP_PLACES defines where the threads may run. Possible values are cores, threads, or socket. OMP_PROC_BIND controls the thread migration. When the OMP_PROC_BIND directive is set to TRUE, OpenMP threads should not be moved; when FALSE they may move between hardware cores and sockets. In general, when OMP_PLACES is set, the setting of OMP_PROC_BIND should be set to TRUE.

The other two less commonly used OpenMP environment variables are OMP_NESTED—controls levels of nested parallelism, and OMP_DYNAMIC—determines whether to change the number of threads or thread groups dynamically. Both variables are set to False by default, which works well in most systems. For performance debugging purposes, however, we can explicitly set the two variables.

In Fig. 8, we show the impact of different OpenMP variables and hybrid MPI-OpenMP configurations on the SpLU speed on Cori Haswell nodes at NERSC. Figure 8a shows the best performance achieved for different OpenMP and NUMA settings variables for purely threaded configurations. Figure 8b shows the performance for different MPI×OpenMP threads on four Haswell nodes of Cori. It should be noted that, hybrid configurations, i.e.

configurations with more than one OpenMP threads per MPI process, tends to require far less memory for MPI's internal buffers [11]. In general, using 2-8 OpenMP threads per MPI process gives good performance across a wide range of matrices.

(a) Best performance for different OMP variables



(b) Performance w.r.t MPI× OMP threads on 4 nodes

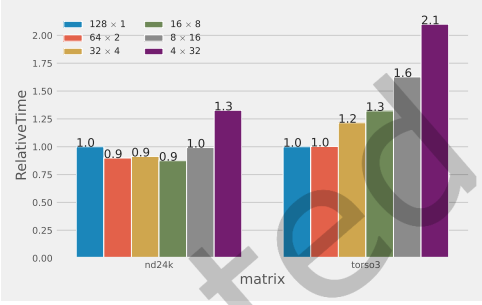


Fig. 8. OpenMP performance tuning for SpLU on Cori Haswell nodes at NERSC. In Fig. 8a, each bar shows the best performance obtained for a variable–value pair by an exhaustive parametric search for the other four variables; the test matrix is torso3 and number of threads is 32.

The OpenMP API allows control of these variables programmatically. This becomes useful when the application and SuperLU require different OpenMP configurations. For best performance, the user can use our autotuner GPTune to tune these variables automatically, see Section 5.

6 CONCLUSION

In this paper, we presented the recently added features in the distributed-memory sparse direct solver SuperLU_DIST. They represent significant algorithmic advances, including (1) communication-avoiding 3D sparse LU factorization, (2) support of multi-GPU offloading, and (3) mixed-precision computations for higher speed and lower memory consumption. Incorporating these algorithmic changes required solving challenging software engineering problems while bringing the research prototype code to the production code that is usable by the users. Furthermore, we documented the parameters that may impact the solver's performance. The parameters we discussed include those in the SuperLU_DIST library as well as some system's parameters related to OpenMP. Since the sparse solvers performance are sensitive to both sparse matrix structures and the underlying computer architectures, we show that an autotuner framework, such as GPTune, provides a powerful tool to help choose the best parameters setting.

We have plans to incorporate several recent fruitful research results into the future releases of the software. These include (1) communication-avoiding 3D SpTRSV [13] and (2) communication-hiding SpTRSV via one-sided MPI and NVSHMEM direct GPU-to-GPU communication [4, 5].

ACKNOWLEDGMENTS

We are grateful to Barry Smith for building the PETSc interface for the new 3D code, for the suggestions to improve the interface of how the parameters should be exposed to the users, and for the detailed feedback of the initial manuscript.

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and in part

by the Scientific Discovery through Advanced Computing (SciDAC) Program under the Office of Science at the U.S. Department of Energy.

REFERENCES

[1] A. Azad, A. Buluc, X.S. Li, X. Wang, and J. Langguth. 2020. A Distributed-Memory Algorithm for Computing a Heavy-Weight Perfect Matching on Bipartite Graphs. *SIAM J. Scientific Computing* 42, 4 (2020), C143–C168.

[2] J. Demmel, Y. Hida, W. Kahan, X.S. Li, S. Mukherjee, and E.J. Riedy. 2006. Error Bounds from Extra-Precise Iterative Refinement. *ACM Trans. Math. Softw.* 32, 2 (June 2006), 325–351.

[3] J. Demmel, Y. Hida, E.J. Riedy, and X.S. Li. 2009. Extra-precise iterative refinement for overdetermined least squares problems. *ACM Transactions on Mathematical Software (TOMS)* 35, 4 (2009), 28.

[4] Nan Ding, Yang Liu, Samuel Williams, and Xiaoye S. Li. [n. d.]. *A Message-Driven, Multi-GPU Parallel Sparse Triangular Solver.* 147–159. https://doi.org/10.1137/1.9781611976830.14 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611976830.14

[5] Nan Ding, Samuel Williams, Yang Liu, and Xiaoye S. Li. [n. d.]. *Leveraging One-Sided Communication for Sparse Triangular Solvers.* 93–105. https://doi.org/10.1137/1.9781611976137.9 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611976137.9

[6] X.S. Li, J.W. Demmel, J.R. Gilbert, L. Grigori, P. Sao, M. Shao, and I. Yamazaki. 1999. *SuperLU Users' Guide.* Technical Report LBNL-44289. Lawrence Berkeley National Laboratory. https://portal.nersc.gov/project/sparse/superlu/ug.pdf, Last update: June 2018.

[7] X. S. Li and J. W. Demmel. 1998. Making Sparse Gaussian Elimination Scalable by Static Pivoting. In *Proceedings of SC98: High Performance Networking and Computing Conference.* Orlando, Florida.

[8] X. S. Li and J. W. Demmel. 2003. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Mathematical Software* 29, 2 (June 2003), 110–140.

[9] Yang Liu, Wissam M. Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W. Demmel, and Xiaoye S. Li. 2021. GPTune: Multitask Learning for Autotuning Exascale Applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21).* Association for Computing Machinery, New York, NY, USA, 234–246. https://doi.org/10.1145/3437801.3441621

[10] P. Sao, X. Liu, R. Vuduc, and X.S. Li. 2015. A Sparse Direct Solver for Distributed Memory Xeon Phi-accelerated Systems. In *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS).* Hyderabad, India.

[11] P. Sao, R. Vuduc, and X. Li. 2014. A Distributed CPU-GPU Sparse Direct Solver. In *Proc. of Euro-Par 2014, LNCS Vol. 8632, pp. 487-498.* Porto, Portugal.

[12] P. Sao, R. Vuduc, and X. Li. 2019. A communication-avoiding 3D algorithm for sparse LU factorization on heterogeneous systems. *J. Parallel and Distributed Computing* (September 2019). https://doi.org/10.1016/j.jpdc.2019.03.004 https://www.sciencedirect.com/science/article/abs/pii/S0743731518305197.

[13] P. Sao, R. Vuduc, and X.S. Li. 2019. A Communication-Avoiding 3D Sparse Triangular Solver. In *ICS 2019: International Conference on Supercomputing.* Phoenix, AZ.

[14] I. Yamazaki and X.S. Li. 2012. New Scheduling Strategies and Hybrid Programming for a Parallel Right-looking Sparse LU Factorization on Multicore Cluster Systems. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS 2012).* Shanghai, China.

APPENDIX

A NAMING CONVENTION AND CODE DOCUMENTATION

The routines in SuperLU_DIST are divided into *driver routines* and *computational routines*. The routine names are inspired by the LAPACK and ScaLAPACK naming convention. For example, the 2D linear solver driver is pdgssvx, where 'p' means parallel, 'd' means double precision,[3] 'gs' means general sparse matrix format, and 'svx' means solving a linear system. Below is a list of double precision user-callable routines.

- Driver routines: pdgssvx (driver for the old 2D algorithms), pdgssvx3d (driver for the new 3D algorithms in Section 2).
- Computational routines: pdgstrf and pdgstrs are respectively triangular factorization SpLU and triangular solve in the 2D process grid. pdgstrf3d is triangular factorization SpLU in the 3D process grid. These routines take a *preprocessed* linear system as an input. An experienced user can use them directly in the application code as they can provide greater flexibility. For a new user, however, using them can be cumbersome and error-prone. We recommend using driver routines, which are easier to use.
- The pddrive and pddrive3d examples in the EXAMPLE/ directory call the respective drivers pdgssvx and pdgssvx3d to solve linear systems. Other examples in the same directory, such as pddrive1, pddrive2, etc., illustrate how to reuse the preprocessing results for a sequence of linear systems with similar structures.

The Doxygen generated documentation for all the routines is available at https://portal.nersc.gov/project/sparse/superlu/superlu_dist_code_html/. Each routine begins with a comment that breaks down input/output arguments and explains the functions of the routine. Although the original User's Guide contains comprehensive description of various internal data structures and algorithms [6], it does not contain the new features presented here.

B FORTRAN 90 INTERFACE

In the FORTRAN/ directory, there are Fortran 90 module files that implement the wrappers for Fortran programs to access the full functionality of the C functions in SuperLU_DIST. The design is based on object-oriented programming concept: define *opaque objects* in the C space, which are accessed via *handles* from the Fortran space. All SuperLU_DIST objects (e.g., process grid, LU structure) are opaque from the Fortran side. They are allocated, deallocated and operated at the C side. For each C object, we define a Fortran handle in Fortran's user space, which points to the C object and implements the access methods to manipulate the object. All handles are 64-bit integer type. For example, consider creating a 3D process grid. The following code snippet shows what are involved from the Fortran and C sides.

- Fortran side

```
/* Declare handle: */
integer(64)::f_grid3d
/* Call C wrapper routine to create 3D grid pointed to by "f_grid3d": */
call f_superlu_gridinit3d(MPI_COMM_WORLD, nprow, npcol, npdep, f_grid3d)
```

- C side

```
 /* Fortran-to-C interface routine: */
void f_superlu_gridinit3d(int *MPIcomm, int *nprow, int *npcol,int *npdep, int64_t *f_grid3d)
 {
     /* Actual call to C routine to create grid3d structture in *grid3d{} */
  superlu_gridinit3d(f2c_comm(MPIcomm),*nprow, *npcol, *npdep, (gridinfo3d_t *) *f_grid3d);
```

[3]We support four datatypes: 's' (FP32 real), 'd' (FP64 double), 'c' (FP32 complex) and 'z' (FP64 complex). Throughout the paper, we use the 'd' version of the routine names.

```
}
```

Here, the Fortran handle `f_grid3d` essentially acts as a 64-bit pointer pointing to the internal 3D grid structure, which is created by the C routine `superlu_gridinit3d()`. This structure (see Fig. 2) sits in the C space and is invisible from the Fortran side.

For all the user-callable C functions, we provide the corresponding Fortran-to-C interface functions, so that the Fortran program can access all the C functionality. These interface routines are implemented in the files `superlu_c2f_wrap.c` (precision-independent) and `superlu_c2f_dwrap.c` (double precision). The Fortran-to-C name mangling is handled by CMake through the header file `SRC/superlu_FCnames.h`. The module file `superlupara.f90` defines all the constants matching the enum constants defined in the C side (see Table 3). The module file `superlu_mod.f90` implements all the access methods (set/get) for the Fortran side to access the objects created in the C user space.

C INSTALLATION WITH CMAKE OR SPACK

C.1 Dependent external libraries

One can have a bare minimum installation of `SuperLU_DIST` without any external dependencies, although the following external libraries are useful for high performance: BLAS, (Par)METIS (sparsity-preserving ordering), CombBLAS (parallel numerical pivoting) and LAPACK (for inversion of dense diagonal block).

C.2 CMake installation

The user will need to create a build tree from which to invoke CMake. The following describes how to define the external libraries.

BLAS (highly recommended)

If the user has a fast BLAS library on their machine, the user can link it using the following CMake definition:

```
-DTPL_BLAS_LIBRARIES="<BLAS library name>"
```

Otherwise, the `CBLAS/` subdirectory contains the part of the C BLAS (single threaded) needed by `SuperLU_DIST`, but it is not optimized for performance. The user can compile and use this internal BLAS with the following CMake definition:

```
-DTPL_ENABLE_INTERNAL_BLASLIB=ON
```

ParMETIS (highly recommended)

http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/parmetis-4.0.3.tar.gz

The user can install ParMETIS and define the two environment variables as follows:

```
export PARMETIS_ROOT=<Prefix directory of the ParMETIS installation>
export PARMETIS_BUILD_DIR=${PARMETIS_ROOT}/build/Linux-x86_64
```

Note that by default, we use serial METIS as the sparsity-preserving ordering, which is available in the ParMETIS package. The user can disable ParMETIS during installation with the following CMake definition: `-DTPL_ENABLE_PARMETISLIB=OFF`. In this case, the default ordering is set to be `MMD_AT_PLUS_A`.

See Table 3 for all the possible ColPerm options.

In order to use parallel symbolic factorization function, the user needs to use ParMETIS ordering.

LAPACK (highly recommended)

In the triangular solve routine, we may use LAPACK to explicitly invert the dense diagonal block to improve the performance. The user can use it with the following CMake option:

```
-DTPL_ENABLE_LAPACKLIB=ON
```

CombBLAS (optional)

https://people.eecs.berkeley.edu/~aydin/CombBLAS/html/index.html

In order to use parallel weighted matching HWPM (Heavy Weight Perfect Matching) for numerical pre-pivoting [1], the user needs to install CombBLAS and define the environment variables:

```
export COMBBLAS_ROOT=<Prefix directory of the CombBLAS installation>
export COMBBLAS_BUILD_DIR=${COMBBLAS_ROOT}/_build
```

Then, install with the CMake option:

```
-DTPL_ENABLE_COMBBLASLIB=ON
```

Use GPU

The user can enable (NVIDIA) GPU with CUDA with the following CMake option:

```
-DTPL_ENABLE_CUDALIB=TRUE
```

The user can enable (AMD) GPU with HIP with the following CMake option:

```
-DTPL_ENABLE_HIPLIB=TRUE
```

For a simple installation with default settings:

```
mkdir build ; cd build;
cmake .. \
  -DTPL_PARMETIS_INCLUDE_DIRS="${PARMETIS_ROOT}/include;\
       ${PARMETIS_ROOT}/metis/include" \
  -DTPL_PARMETIS_LIBRARIES="${PARMETIS_BUILD_DIR}/libparmetis/libparmetis.a;\
       ${PARMETIS_BUILD_DIR}/libmetis/libmetis.a" \
```

There are a number of example build scripts in the `example_script/` directory, with filenames `run_cmake_build_*.sh` that target various machines.

To actually build (compile), type: 'make'.

To install the libraries, type: 'make install'.

To run the installation tests, type: 'test'. (The outputs are in file: 'build/Testing/Temporary/LastTest.log') or, 'ctest -D Experimental', or, 'ctest -D Nightly'.

Note that the parallel execution in `ctest` is invoked by the "mpiexec" command, which is from the MPICH environment. If the MPI is not MPICH/mpiexec based, the test execution may fail. The user can pass the definition option `-DMPIEXEC_EXECUTABLE` to CMake. For example on Cori at NERSC, the user will need the following: `cmake .. -DMPIEXEC_EXECUTABLE=/usr/bin/srun`.

Or, the user can always go to `TEST/` directory to perform testing manually.

The following list summarizes the commonly used CMake definitions. In each case, the first choice is the default setting. After running a 'cmake' installation, a configuration header file is generated in `SRC/superlu_dist_config.h`, which contains the key CPP definitions used throughout the code.

```
-DTPL_ENABLE_INTERNAL_BLASLIB=OFF | ON
-DTPL_ENABLE_PARMETISLIB=ON | OFF
-DTPL_ENABLE_LAPACKLIB=OFF | ON
-DTPL_ENABLE_COMBBLASLIB=OFF | ON
-DTPL_ENABLE_CUDALIB=OFF | ON
-DCMAKE_CUDA_FLAGS=<...>
-DTPL_ENABLE_HIPLIB=OFF | ON
-DHIP_HIPCC_FLAGS=<...>
```

```
-Denable_complex16=OFF | ON        (double-complex datatype)
-Denable_single=OFF | ON           (single precision real datatype)
-DXSDK_INDEX_SIZE=32 | 64          (integer size for indexing)
-DBUILD_SHARED_LIBS= OFF | ON
-DCMAKE_INSTALL_PREFIX=<...>
-DCMAKE_C_COMPILER=<MPI C compiler>
-DCMAKE_C_FLAGS=<...>
-DCMAKE_CXX_COMPILER=<MPI C++ compiler>
-DCMAKE_CXX_FLAGS=<...>
-DXSDK_ENABLE_Fortran=OFF | ON
-DCMAKE_Fortran_COMPILER=<MPI F90 compiler>
```

C.3 Spack installation

Spack installation of `SuperLU_DIST` is a fully automated process. Assume that the develop branch of Spack (https://github.com/spack/spack) is used. The user can find available compilers via: `spack compilers`. In the following, let's assume the available compiler is gcc@9.1.0. The installation supports the following variants:

Use 64-bit integer

The user can enable 64-bit integer with:

```
spack install superlu-dist@master+int64%gcc@9.1.0
```

Use GPU

The user can enable (NVIDIA or AMD) GPUs with:

```
spack install superlu-dist@master+cuda%gcc@9.1.0
spack install superlu-dist@master+rocm%gcc@9.1.0
```

Test installation

The user can run a few smoke tests of the spack installation via

```
spack test run superlu-dist@master (pick the appropriate installation if multiple variants available)
```