Newly Released Capabilities in Distributed-memory SuperLU Sparse Direct Solver

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

A crisp summary.

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

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

ACM Reference Format:
Xiaoye S. Li, Yang Liu, Paul Lin, and Piyush Sao. 2022. Newly Released Capabilities in Distributed-memory SuperLU Sparse Direct Solver. In . ACM, New York, NY, USA, 20 pages. https://doi.org/XXXXXXX.XXXXXXX

Contents

Abstract 1
Contents 1
1 Overview of SuperLU and SuperLU_DIST 2
2 3D Communication-Avoiding Routines 3
2.1 The 3D Process layout and its performance impact 4
3 OpenMP Intra-node Parallelism 7
3.1 OpenMP Performance tuning 7
4 GPU-enabled Routines 8
4.1 2D SpLU algorithm and tuning parameters 8
4.2 3D SpLU algorithm and tuning parameters 10
4.3 2D SpTRSV algorithm 10
5 Mixed-precision Routines 10
6 Summary of Parameters, Environment Variables and Performance Influence 13
6.1 3D CPU SpLU 14
6.2 2D GPU SpLU 15
6.3 3D GPU SpLU 15
7 Fortran 90 Interface 15
8 Installation with CMake or Spack 17
8.1 Dependent external libraries 17
8.2 CMake installation 17

*All the authors contributed equally to this article.

Manuscript submitted to ACM

8.3 Spack installation 19
9 PETSc Interface with GPU Capability 19
Acknowledgments 19
References 19

1 OVERVIEW OF SUPERLU AND SUPERLU_DIST

SuperLU contains a set of sparse direct solvers for solving large sets of linear equations $AX = B$ [4]. Here A is a square, nonsingular, $n \times n$ sparse matrix, and X and B are dense $n \times nrhs$ matrices, where $nrhs$ is the number of right-hand sides and solution vectors. The matrix A need not be symmetric or definite; indeed, SuperLU is particularly appropriate for unsymmetric matrices, and it respects both the unsymmetric values as well as the unsymmetric sparsity pattern. The routines appear in three different libraries: sequential (`SuperLU`), multithreaded (`SuperLU_MT`) and distributed-memory parallel (`SuperLU_DIST`). They can be linked together in a single application. All three libraries use variations of Gaussian elimination (LU factorization) optimized to take advantage both of sparsity and of computer architecture, in particular memory hierarchy (caches) and parallelism.

The `SuperLU_DIST` library 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. 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`). The parallel algorithm consists of the following major steps.

(1) Preprocessing
(2) Sparse LU factorization (SpLU)
(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 stuctures. 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 and avoid over/underflow. P_r and P_c are *permutation matrices*. The role of P_r is to permute rows of the matrix to make diagonal element large relative to the off-diagonal elements (numerical pivoting). The role of P_c is to to permute row 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 [4, 6]. 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.

This release paper focuses on the new capabilities in Steps 2-4 in `SuperLU_DIST`. Throughout the paper, when there is no ambiguity, we simply refer to the library `SuperLU_DIST` as SuperLU.

Before the new Version-7 release (2021), the distributed memory code had been largely built upon the design in the first paper [5]. 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 (GESP),
- 2D logical process arrangement for non-uniform block-cyclic mapping, based on supernodal block partition, and
- loosely synchronous scheduling with look-ahead pipelining [10].

The parallel SpTRSV uses the same 2D block-cyclic layout of the L and U matrices as the results of SpLU. An entirely message-driven asynchronous scheduling algorithm is designed to mitigate communication dominance [5]. A working precision iterative refinement can be optionally invoked to improve solution accuracy.

The routines in SuperLU 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,[1] 'gs' means general sparse matrix format, and 'svx' means solving a linear system. Below is a list of double prercision 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 2D process grid. `pdgstrf3d` is triangular factorization SpLU in 3D process grid. These routines take as input the linear system that is already preprocessed. This may be cumbersome and error prone for a novice user. Therefore, we recommend the users to use the driver routines as much as possible.
- Example routines in EXAMPLE/ directory: `pddrive` and `pddrive3d` that call the respective drivers `pdgssvx` and `pdgssvx3d` to solve a linear systems. In EXAMPLE/ directory, there are a number of other examples, `pddrive1`, `pddrrive2`, ... etc., which illustrate how to call the drivers to reuse the preprocessing results for a sequennce of linear systems with similar structures.

The Doxygen generated documentation for all the routines are available at https://portal.nersc.gov/project/sparse/superlu/superlu_dist_code_html/. The leading comment in the source code of each routine describes in detail the input/output arguments, and the routine's functionality.

In the following sections, we will describe in detail the new features available since Version-7 release, including the 3D communication-avoiding algorithm framework, the multi-GPU acceleration, the mixed-precision algorithms, the new installation mechanisms and the user interfaces.

2 3D COMMUNICATION-AVOIDING ROUTINES

Motivated by the strong scaling requirement from the exascale applications and the communication-avoiding algorithm development for dense linear algebra in the last decade, we developed a novel 3D algorithm framework for sparse factorization and triangular solves to mitigate communication cost. We use a three-dimensional MPI process grid, exploits elimination tree parallelism, and trades off increased memory for reduced per-process communication. The 3D processes grid is configured as $P = P_x$ x P_y x P_z (see Fig. 2a). The role of each process in the 3D algorithm is not identical. From algorithm viewpoint, we should think of P processses consisting of 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 a 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 [9, Section 3.3]. The

[1] 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.

matrices A, L and U corresponding to each subtree-forest is 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 pocess grids.

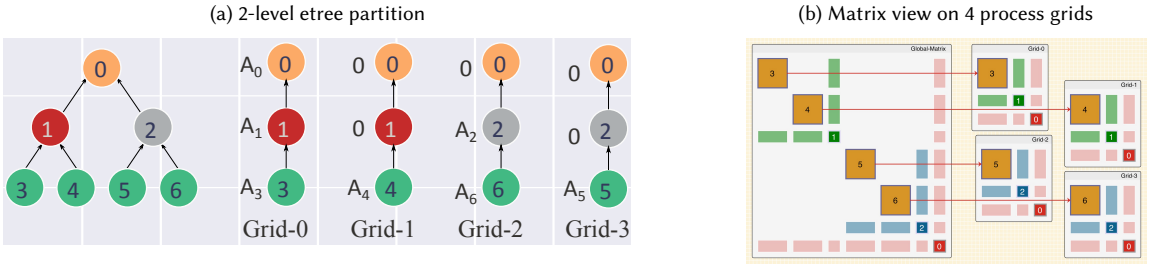


Fig. 1. Illustration of the 3D parallel SpLU algorithm with 4 process grids.**[do we need to define A_i? – YANG]**

The example program `EXAMPLE/pddrive3d.c` shows how the user can use the 3D algorithm to solve a sparse linear system. As an initialization step, the user needs to call

```
superlu_gridinit3d (MPI_COMM_WORLD, nprow, npcol, npdep, &grid);
```

Here, nprow, npcol and npdep are user input, corresponding to the P_x, P_y and P_z respectively. In this example, a new process group for SuperLU is built upon the MPI default communicator `MPI_COMM_WORLD`. In general, it can be built upon any MPI communicator. All the subsequent parallel routines in SuperLU will carry the `grid` variable and use this process group. In this way, the MPI messages within SuperLU do not mix up with the meessages from the other codes.

2.1 The 3D Process layout and its performance impact

In SuperLU, a 3D process grid can be arranged in two formats: XY-major or Z-major, see Fig. 3. In XY-major, processes with the same XY-coordinate (**[The phrase is confusing, maybe processes within the same 2D grid? – YANG]**) 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

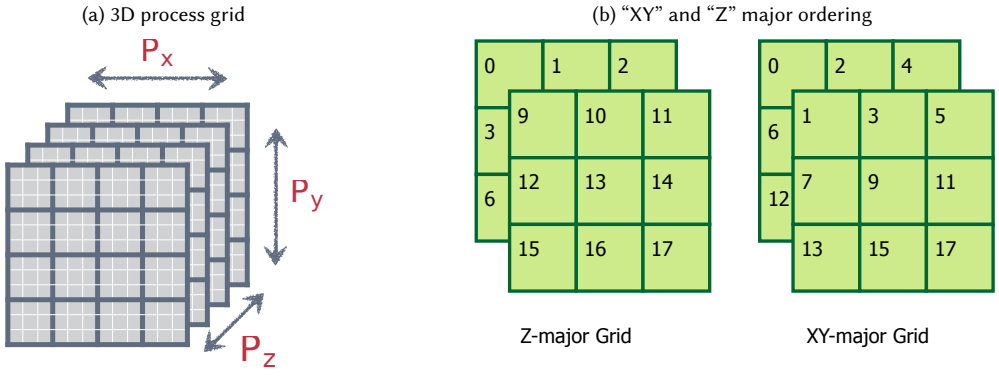


Fig. 2. A logical 3D process grid and process configuration for RANKORDER=$\{XY, Z\}$

```
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 )
                           *    e.g. 1x3x4 grid: layer0 layer1 layer2 layer3
                           *                     0      3      6      9
                           *                     1      4      7      10
                           *                     2      5      8      11
                           * = 1: XY-major (need set environment variable: RANKORDER=XY)
                           *    e.g. 1x3x4 grid: layer0 layer1 layer2 layer3
                           *                     0      1      2      4
                           *                     5      6      7      8
                           *                     9      10     11     12
   */
} gridinfo3d_t;
```

Fig. 3. 3D process grid definition. **[the ranks in X-Y-major example should be 0-11 – YANG]**

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.

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 with GPU acceleration cases.**[Why?? – SHERRY]**

Add xy-z-major figure in Fig. 1

Add perforrmance data to show RANKORDER influence.

The driver routine is `pdgssvx3d`, with the following calling API:

```
void pdgssvx3d (superlu_dist_options_t *options, SuperMatrix *A,
                dScalePermstruct_t *ScalePermstruct,
                double B[], int ldb, int nrhs, gridinfo3d_t *grid,
                /* following are output */
                dLUstruct_t *LUstruct, dSOLVEstruct_t *SOLVEstruct,
        double *berr, SuperLUStat_t *stat, int *info);
```

The first argument is input, making the algorithm choices in the `options` structure. Section 6 describes all possible options and how to change each option. Table 2 tabulates the default values. The second argument is the input matrix A stored in the `SuperMatrix` metadata structure. The third argument is an input/output structure storing all the transformation vectors obtained from the preprocessing steps. The input right-hand sides are given by the `{B, ldb, nrhs}` tuple. The `grid` structure defines the 3D process grid, including the MPI communicator for this grid.

All the precision-independent structures are defined in `superlu_defs.h`, and the precision-dependent structures are defined in `superlu_ddef.h` (for double precision). The sparse LU factors and the triangular solve structures are output. In addition the `berr` argument returns an array of componentwise relative backward error of each solution vector.

The sparse LU 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 2*D* factorization of locally owned submatrix of A. This is the same algorithm as the one before Version-7 []. 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 memory needed to achieve these gains is a constant factor **[memory is a factor, this sounds strange – YANG]**. 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 NERCS). Please see [9] 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 3D SpLU Algorithm. The 3D algorithm provides two strategies for partitioning the elimination tree to balance the load between different 2D grids. The **SUPERLU3DLBS [typesetting?? – SHERRY]** environment variable specifies which one to use.

- **Nested Dissection (ND)** based: It uses the partitioning provided by a nested dissection ordering. It works well for regular grids or user-provided column ordering **[?? – SHERRY]**. 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.
- **The greedy Heuristic (GD)** strategy uses a greedy algorithm to divide one level of the separator tree **[elimination tree – SHERRY]**. 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,

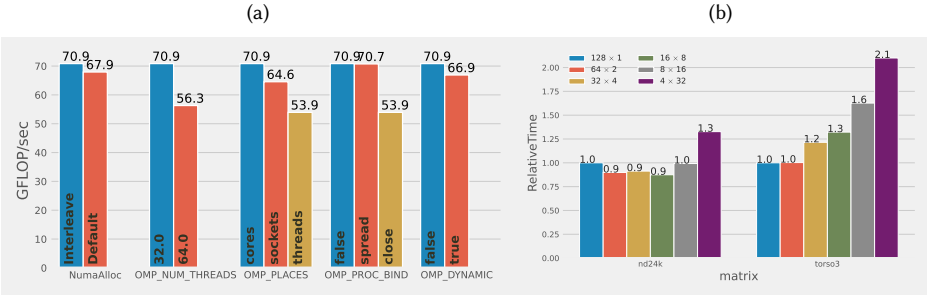


Fig. 4. Best performance achieved for different OpenMP variables.**[this figure is not cited in the text. Which machine? (Cori Haswell). Explain the plots – SHERRY]**

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 SUPERLU3DLBS=ND is specified.

3 OPENMP INTRA-NODE PARALLELISM

SuperLU can use shared-memory parallelism in two ways. First, is by using the multithreaded BLAS library for linear-algebraic operations. This is independent of the implementation of SuperLU itself. Second, SuperLU can use OpenMP pragmas for explicitly parallelizing some of the 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.*

- The performance of OpenMP parallelism is often less predictable than pure MPI parallelism. This is due to non-determinism in the threading layer, the CPU hardware, and thread affinities.
- OpenMP threading may cause a significant slowdown if parameters are chosen incorrectly. Performance slow-down often is not entirely transparent. Slow-down can be due to false-sharing, NUMA effects, hyperthreading, incorrect or suboptimal thread affinities, or underlying threading libraries.
- Performance variation can be observed between compilers and threading libraries.
- Performance can be difficult to model or predict. Performance tuning may require some trial and error. Performance tuning is also dependent on the CPU architecture, the number of cores, and the underlying operating system.

3.1 OpenMP Performance tuning

[need to tie this with Section 9. These env variables need to be put in Table 1. – SHERRY]

Performance tuning of OpenMP applications is critical to get the desired performance. In this section, we list some of the most important environment variables that impact the performance of SuperLU and indicate how they should be set to achieve maximum performance.

- OMP_NUM_THREADS: controls the number of OpenMP threads. To avoid resource over subscription, the product of MPI processes per node and OpenMP threads should be less than available physical cores.
- OMP_PLACES: Defines where OpenMP threads may run. Possible values are cores, threads, Socket. The default value is "threads," and it's generally a good choice. You might want to test both "cores" and "threads" values on older processor models.
- OMP_PROC_BIND: Defines how threads map onto the OpenMP places. Possible values are false, master, close, spread. The default value is "spread," and it's generally a good choice. You might want to test both "close" and "spread" values on an older processor model.
- OMP_NESTED: How many levels of openMP parallelism do you want to use; Typically, setting it to false gives the best performance—in fact, setting it may degrade performance due to oversubscription to threads.
- OMP_DYNAMIC: decides whether to dynamically change any of the numbers of thread/ threads groups for better performance. Typically, false gives the best performance. Setting it to true can lead to degraded performance.

The OpenMP API lets you control these variables programmatically. This becomes useful when the application and SuperLU require different OpenMP configurations.

4 GPU-ENABLED ROUTINES

In the current release, the SpLU factorization routines can offload certain amount of computation to GPU, which is mostly in each Schur complement update (SCU) 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 CPP 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' means ACCelerator.)

4.1 2D SpLU algorithm and tuning parameters

The first sparse LU factorization algorithm capable of offloading the matrix-matrix multiplication to the GPU was published in [8]. The panel factorization and the Gather/Scatter operations are 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 paper provides a gentle introduction to GPU acceleration in `SuperLU_DIST`and its performance tuning.

Performing SCU requires some temporary storage to hold dense blocks. In an earlier algorithm, at each elimination step, the SCU is performed one 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 GEMM performance for 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.

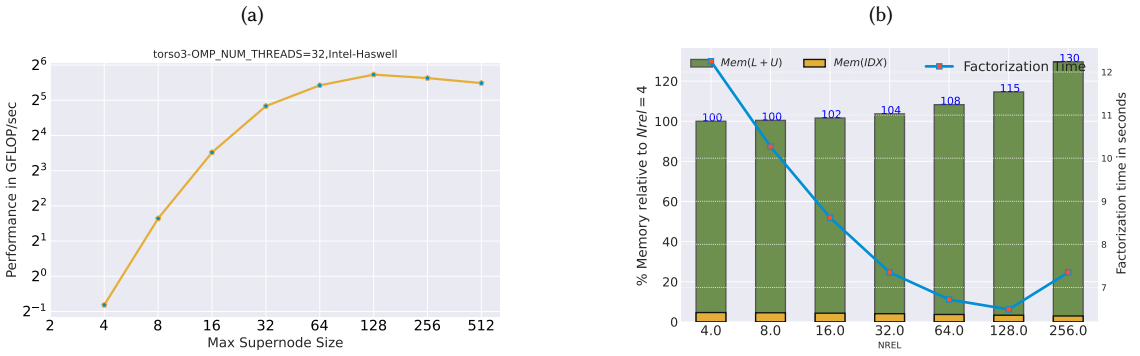


Fig. 5. Impact of maximum supernode size (NSUP) and supernodal relaxation (NREL) on performance and memory. **[This figure is not referenced? Should move this figure to Section 9. – SHERRY]**

In [8], we modified the algorithm in the SCU 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 the modern GPUs have a lot more memory than that of the earlier generations, this extra memory is far more affordable to enable a much faster runtime.

Now, each step of SCU consists of the following substeps:

(1) Gather sparse blocks $U(k,:)$ into a dense BLAS compilant 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 Cpu $\Leftrightarrow$ Gpu data transfer 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 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 GEMM into multiple streams. Each stream performs its own sequence of operations: CPU-to-GPU transfer, GEMM, and CPU-to-GPU transfer. Between these streams, these operations are asynchronous. The GPU matrix multiplication is also pipelined with the Scatter operation performed on CPU.
- To offset the memory limitation on the GPU, we devised an algorithm to divide the SCU into smaller chunks as $\{[cpu:gpu]_1 \mid [cpu:gpu]_2 \mid ...\}$. 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 two environment variables that can be used to control the memory and performance in the GEMM offload algorithm:

- number of GPU streams n_s (NUM_GPU_STREAMS, default is 8); and

- maximum buffer size (in words) on GPU that can hold the GEMM output matrix V (`MAX_BUFFER_SIZE`, default is 256000000, i.e., 256M).

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

4.2 3D SpLU 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 [**?**]. Compared to the GPU algorithm in the 2D code Section 4.1, this algorithm also offloads the Scatter operations of each SCU step to 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 environment variables are:

explain this, guidance?

- `NUM_LOOKAHEADS`, number of lookahead levels (default is 10)

New performance numbers on larger nodes on a new GPUs, A100 (Perlmutter), MI100 (Spock)?? (Piyush)

4.3 2D SpTRSV algorithm

When the 2D grid has one MPI, SpTRSV in `SuperLU` is parallelized using OpenMP for shared-memory parallelism and CUDA/HIP for the GPU. Both versions of the implementations are based a asynchronous level-set traversal algorithm that distributes the computation workloads across CPU threads and GPU threads/blocks. The CPU implementation supports OpenMP taskloops and tasks for dynamic scheduling, while the GPU implementation relies on static scheduling. Fig. 6a shows the performance of L-solve on 1 Cori Haswell node with 1 and 32 OpenMP threads. Fig. 6b shows the performance of L-solve using `SuperLU` (8 Summit CPU cores or 1 Summit V100 GPU) and CUSPARSE (1 Summit V100 GPU). The GPU SpTRSV in `SuperLU` constantly overperforms CUSPARSE and is comparable to 8-Core CPU results.

When the 2D grid has more than 1 MPIs, SpTRSV also supports OpenMP parallelism with degraded performance. In addition, the multi-GPU SpTRSV in `SuperLU` is under actively development and will be avai

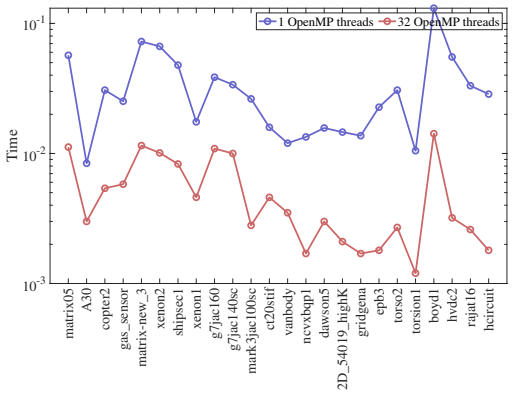
5 MIXED-PRECISION ROUTINES

SuperLU 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. This is usually not supported by many other libraries.

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. In 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 accurracy (cheap). In 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 SpMV and SpTRSV).

In the dense LU and QR factorizations, the benefit of lower precision format mainly comes 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 Scatter

(a) L-solve time with 1 and 32 OpenMP threads on Cori Haswell



(b) L-solve time of SuperLU and CuSparse on CPU and GPU

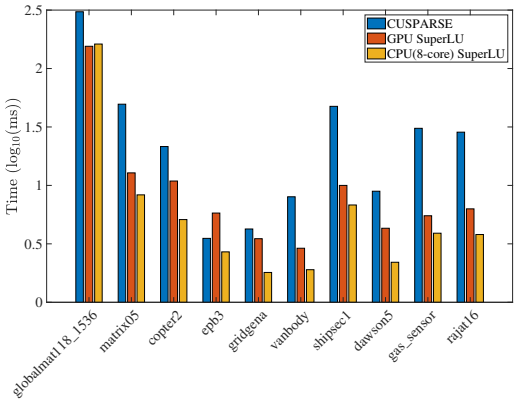
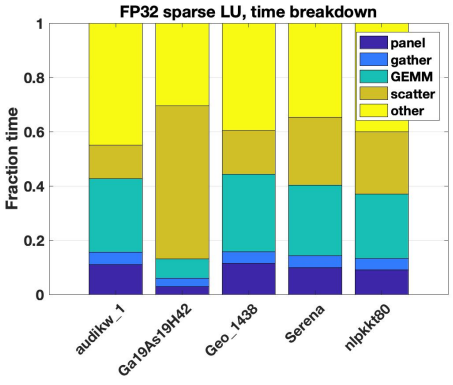


Fig. 6. Performance of SpTRSV with 1 MPI.

(a) Time breakdown of various steps of FP32 SpLU, "Other" mostly consists of MPI communication



(b) Comparision of SpLU time between the FP32 and FP64 versions

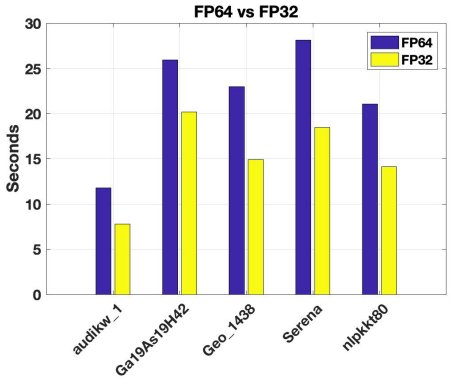


Fig. 7. Times of FP32 and FP64 SpLU for 5 matrices. All are measured on 10 nodes of ORNL Summit with 6 MPI tasks and 6 GPUs per node.

operation. In Figure 7 we tally the time of various steps in SpLU and the time comparision of using FP32 vs. FP64. These are measured times for five real matrices of dimension on the order of 1 million or so. As can be seen, depending on the matrix sparsity structure, the fraction of time in GEMM varies, and usually is less than 50% (left plot). Because of this, the Tensor Core version of GEMM calls led to only less than 5% speedup for the whole SpLU. When comparing FP32 with the FP64 versions, we observed about 50% speedup with the FP32 version (right plot).

The simplest mixed precision sparse direct solver is to use lower precision for the expensive LU and QR factorizations, and higher precision in the cheap residual and solution update in IR. We recall the IR algorithm using three precisions

(a) audikw_1 convergence history (b) Ga19As19H2 convergence history

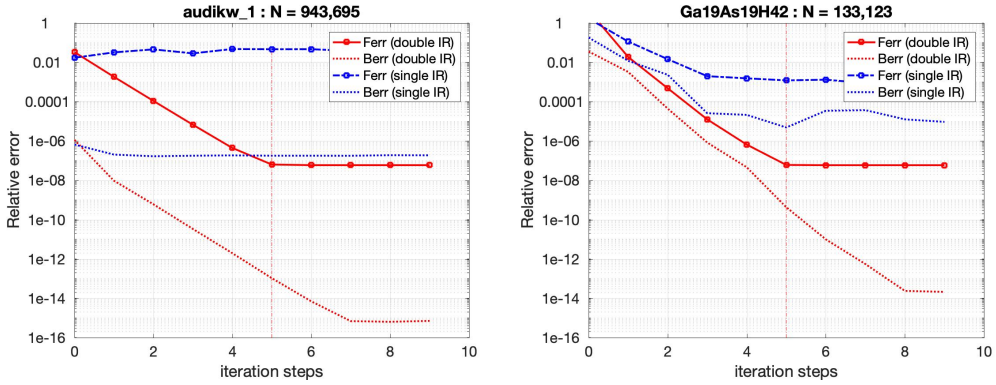


Fig. 8. Convergence history of Algorithm 1 when applied to two sparse linear systems. The vertical line in each plot corresponds to the IR steps taken when our stopping criteria are satisfied.

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. Potentially, the following three precisions may be used:

- ε_w is the working precision; it is used as 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.

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: $r^{(i)} \leftarrow b - Ax^{(i)}$ ▷ (ε_r)
5: Solve $A\,dx^{(i+1)} = r^{(i)}$ using the basic solution method ▷ (ε_w)
6: Update $x^{(i+1)} \leftarrow x^{(i)} + dx^{(i+1)}$ ▷ (ε_x)
7: $i \leftarrow i + 1$
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 $1/(\gamma(n)\varepsilon_w)$. Moreover, this IR procedure can return to the user the reliable error bounds both normwise and componentwise. The error analysis in [2] should all carry through to the sparse cases.

We implemented Algorithm 1 in SuperLU, 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 8, the left two plots show the convergence history of two systems, in both normwise foward 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 all around, see Table 1.

Table 1. Parallel solution time (seconds) (including SpLU and IR): purely double precision, purly 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 driver routines for this mixed-precision setup are `psgsssvx` and `psgssvx3d`, with the following API:

```
void psgssvx(superlu_dist_options_t *options, SuperMatrix *A,
             sScalePermstruct_t *ScalePermstruct,
             float B[], int ldb, int nrhs, gridinfo_t *grid,
             sLUstruct_t *LUstruct, sSOLVEstruct_t *SOLVEstruct,
             float *berr, SuperLUStat_t *stat, int *info)
```

To use double precision IR, we need to set: `options->IterRefine = SLU_DOUBLE`.

6 SUMMARY OF PARAMETERS, ENVIRONMENT VARIABLES AND PERFORMANCE INFLUENCE

Throughout all phases of the solution process, there are a number of algorithm parameters that can influence solver's performance and that 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 2 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 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 some other parameters, the user can change them at run time via environment variables. These parameters are listed in the third column in Table 2.

Many of the parameters and environment variables listed in Table 2 are performance critical for the 2D and 3D, CPU and GPU algorithms described in Sections 2, 4.1 and 4.2. Here we leverage an autotuner called GPTune [7] 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.

Table 2. List of algorithm parameters used in various phases of the linear solver. The '(env)' notion marks the environment variables that can be reset at runtime. The other parameters must be set in the `options{}` structure input to a driver routine. **[how about other command line options, such as -r, -c, -d? – YANG]->[those are user's responsibility. They will have their own "pddrive" – SHERRY]**

phase	`options` (compile time)	env variables (runtime)	values / enum constants	in 2D or 3D algo. ?
Preprocessing	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	`NUM_LOOKAHEADS`	default 10	2d, 3d
		`NUM_OMP_THREADS`	default ????	2d, 3d
		`OMP_PLACES`	default ????	2d, 3d
		`OMP_PROC_BIND`	default ????	2d, 3d
		`OMP_NESTED`	default ????	2d, 3d
		`OMP_DYNAMIC`	default ????	2d, 3d
		`NREL`	default 60, set via `sp_ienv(2)`	2d, 3d
		`NSUP`	default 256, set via `sp_ienv(3)`	2d, 3d
		`RANKORDER`	default Z-major	3d
		`SUPERLULBS`	default GD	3d
		`SUPERLU_ACC_OFFLOAD`	0, 1 (default)	2d, 3d
		`N_GEMM`	default 1000	2d
		`MAX_BUFFER_SIZE`	250 million	2d, 3d
		`NUM_GPU_STREAMS`	default 8	2d, 3d
		`MPI_PROCESS_PER_GPU`	default 1	2d, 3d
SpTRSV	IterRefine		0: NOREFINE (default ???)	2d, 3d
			1: `SLU_SINGLE`	
			2: `SLU_DOUBLE`	
Others	PrintStat		NO, YES (default)	2d, 3d

6.1 3D CPU SpLU

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 512 MPIs. We consider the following tuning parameters [`NSUP`, `NREL`, num_lookaheads, P_x, P_z]. We set up GPTune to generate 100 samples. All samples and the Pareto front are plotted in Fig. 9a. The samples on the Pareto front and the default one are shown in Table 3, one can clearly see that by reducing the computation granularity (`NSUP`, `NREL`) and increasing P_z, one can significantly improve the SpLU time yet uses slightly higher memory.

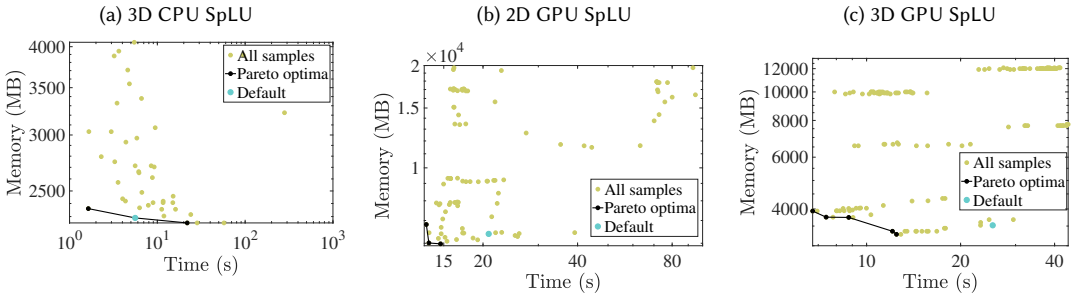


Fig. 9. Samples generated by GPTune for the three tuning experiments. Only valid samples are plotted.

6.2 2D GPU SpLU

For the 2D GPU SpLU algorithm (4.1), we use 2 NERSC Perlmutter GPU nodes with 4 MPIs per node and the H2O matrix. The number of OpenMP threads is set to 16. We consider the following tuning parameters [ColPerm, NSUP, NREL, N_GEMM , MAX_BUFFER_SIZE, P_x]. We set up GPTune to generate 100 samples. All samples and the Pareto front are plotted in Fig. 9b. The samples on the Pareto front and the default one are shown in Table 4. Compared to the default configuration, both the time and memory can be significantly improved by Increasing the computation granularity (larger NSUP, NREL). Also, less GPU offload (larger N_GEMM) leads to better performance.

6.3 3D GPU SpLU

For the 3D GPU SpLU algorithm (4.2), we use 2 NERSC Perlmutter GPU nodes with 4 MPIs per node and the H2O matrix. The number of OpenMP threads is set to 16. We consider the following tuning parameters [ColPerm, NSUP, NREL, 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. 9c. 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 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 there are only 8 MPIs used.

	NSUP	NREL	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 3. Default and optimal samples returned by GPTune for the 3D CPU SpLU algorithm.

7 FORTRAN 90 INTERFACE

In FORTRAN/ directory, there are Fortran 90 module files that implement the wrappers for the Fortran 90 programs to access the full functionality of the C functions in SuperLU. The design is based on object-oriented programming concept: define *opaque objects in the C space, which are accessed via* handles from the Fortran 90 space. All SuperLU objects (e.g., process grid, LU structure) are opaque from Fortran 90 side. They are allocated, deallocated and operated

	ColPerm	NSUP	NREL	N_GEMM	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 4. Default and optimal samples returned by GPTune for the 2D GPU SpLU algorithm.

	ColPerm	NSUP	NREL	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 5. Default and optimal samples returned by GPTune for the 3D GPU SpLU algorithm.

at the C side. For each C object, we define a Fortran 90 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 90 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);
 }
```

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. 3) 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 2). 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.

8 INSTALLATION WITH CMAKE OR SPACK

8.1 Dependent external libraries

You can have a bare minimum installation of SuperLU 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).

8.2 CMake installation

You 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 you have a fast BLAS library on your machine, you 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, but they are not optimized for speed. You 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

You 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. You can disable ParMETIS during installation with the following CMake definition: `-DHAVE_PARMEETIS=FALSE`. In this case, the default ordering is set to be `MMD_AT_PLUS_A`.

See Table 2 for all the possible ColPerm options.

In order to use parallel symbolic factorization function, you need to use ParMETIS ordering.

LAPACK (optional) [YANG: is this required for GPU? – SHERRY]

In triangular solve routine, we may use LAPACK to explicitly invert the dense diagonal block to improve speed. You 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], you need 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 cmake option:

```
-DTPL_ENABLE_COMBBLASLIB=ON
```

Use GPU

You can enable (NVIDIA) GPU with CUDA with the following cmake option:

```
-DTPL_ENABLE_CUDALIB=TRUE
```

You can enable (AMD) GPU with HIP with the following cmake option:

```
-DTPL_ENABLE_HIPLIB=TRUE
```

For a simple installation with default setting, do:

```
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 scripts in `example_script/` directory, with filenames `run_cmake_build_*.sh` that are used on various machines.

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

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

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

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

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

The following list summarize the commonly used CMake definitions. In each case, the first choice is the default setting. After running '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 precisiion 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>
```

8.3 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. You 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.

You can enable 64bit-integer with

```
spack install superlu-dist@7.2.0+int64%gcc@9.1.0
```

Use GPU.

You can enable (NVIDIA or AMD) GPUs with:

```
spack install superlu-dist@7.2.0+cuda%gcc@9.1.0
spack install superlu-dist@7.2.0+rocm%gcc@9.1.0
```

Test installation.

You can run a few smoke tests of the spack installation via

```
spack test run superlu-dist@7.2.0    (pick the appropriate installation if multiple variants available)
```

9 PETSC INTERFACE WITH GPU CAPABILITY

https://petsc.org/main/docs/manualpages/Mat/MATSOLVERSUPERLU_DIST.html

Sherry

ACKNOWLEDGMENTS

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 Matchi ng 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] 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/. Last update: June 2018.

[5] 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.

[6] 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.

[7] 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

[8] 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.

[9] 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.

[10] 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.