SuperLU: Sparse Direct Solver and Preconditioner

X. Sherry Li
xsli@lbl.gov
http://crd.lbl.gov/~xiaoye/SuperLU

Argonne Training Program on Extreme-Scale Computing (ATPESC)
August 8, 2014
Acknowledgements

- Supports from DOE, NSF, DARPA
  - FASTMath (Frameworks, Algorithms and Scalable Technologies for Mathematics)
  - TOPS (Towards Optimal Petascale Simulations)
  - CEMM (Center for Extended MHD Modeling)

- Developers and contributors
  - Sherry Li, LBNL
  - James Demmel, UC Berkeley
  - John Gilbert, UC Santa Barbara
  - Laura Grigori, INRIA, France
  - Meiyue Shao, Umeå University, Sweden
  - Pietro Cicotti, UC San Diego
  - Piyush Sao, Georgia Tech
  - Daniel Schreiber, UIUC
  - Yu Wang, U. North Carolina, Charlotte
  - Ichitaro Yamazaki, LBNL
  - Eric Zhang, Albany High School
Quick installation

  - Users’ Guide, HTML code documentation

- Gunzip, untar

- Follow README at top level directory
  - Edit make.inc for your platform (compilers, optimizations, libraries, ...)
    (may move to autoconf in the future)
  - Link with a fast BLAS library
    - The one under CBLAS/ is functional, but not optimized
    - Vendor, GotoBLAS, ATLAS, ...
Outline of Tutorial

- Functionality
- Sparse matrix data structure, distribution, and user interface
- Background of the algorithms
  - Differences between sequential and parallel solvers
- Examples, Fortran 90 interface

- Hands on exercises
Solve sparse $Ax=b$ : lots of zeros in matrix

- fluid dynamics, structural mechanics, chemical process simulation, circuit simulation, electromagnetic fields, magneto-hydrodynamics, seismic-imaging, economic modeling, optimization, data analysis, statistics, . . .
- Example: $A$ of dimension $10^6$, 10~100 nonzeros per row
- Matlab: > spy(A)

Boeing/msc00726 (structural eng.)
Mallya/lhr01 (chemical eng.)
Solving a system of linear equations $Ax = b$

- Sparse: many zeros in $A$; worth special treatment

Iterative methods: (e.g., Krylov, multigrid, …)

- $A$ is not changed (read-only)
- Key kernel: sparse matrix-vector multiply
- Easier to optimize and parallelize
- Low algorithmic complexity, but may not converge

Direct methods

- $A$ is modified (factorized)
- Harder to optimize and parallelize
- Numerically robust, but higher algorithmic complexity

Often use direct method to precondition iterative method

- Solve an easy system: $M^{-1}Ax = M^{-1}b$
Available direct solvers

- Survey of different types of factorization codes
  
  
  - $L L^T$ (s.p.d.)
  - $L D L^T$ (symmetric indefinite)
  - $L U$ (nonsymmetric)
  - $Q R$ (least squares)
  - Sequential, shared-memory (multicore), distributed-memory, out-of-core
    - GPU, FPGA become active.

- Distributed-memory codes: usually MPI-based
  
  - SuperLU_DIST [Li/Demmel/Grigori/Yamazaki]
    - accessible from PETSc, Trilinos, . . .
  - MUMPS, PasTiX, WSMP, . . .
SuperLU Functionality

- LU decomposition, triangular solution
- Incomplete LU (ILU) preconditioner (serial SuperLU 4.0 up)
- Transposed system, multiple RHS
- Sparsity-preserving ordering
  - Minimum degree ordering applied to $A^T A$ or $A^T + A$ [MMD, Liu `85]
  - ‘Nested-dissection’ applied to $A^T A$ or $A^T + A$ [(Par)Metis, (PT)-Scotch]
- User-controllable pivoting
  - Pre-assigned row and/or column permutations
  - Partial pivoting with threshold
- Equilibration: $D_r A D_c$
- Condition number estimation
- Iterative refinement
- Componentwise error bounds [Skeel `79, Arioli/Demmel/Duff `89]
### Software Status

<table>
<thead>
<tr>
<th></th>
<th>SuperLU</th>
<th>SuperLU_MT</th>
<th>SuperLU_DIST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Platform</strong></td>
<td>Serial</td>
<td>SMP, multicore</td>
<td>Distributed memory</td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td>C</td>
<td>C + Pthreads or OpenMP</td>
<td>C + MPI + OpenMP + CUDA</td>
</tr>
<tr>
<td><strong>Data type</strong></td>
<td>Real/complex, Single/double</td>
<td>Real/complex, Single/double</td>
<td>Real/complex, Double</td>
</tr>
<tr>
<td><strong>Data structure</strong></td>
<td>CCS / CRS</td>
<td>CCS / CRS</td>
<td>Distributed CRS</td>
</tr>
</tbody>
</table>

- **Fortran interfaces**
- **SuperLU_MT similar to SuperLU both numerically and in usage**
Usage of SuperLU

- **Industry**
  - Cray Scientific Libraries
  - FEMLAB
  - HP Mathematical Library
  - IMSL Numerical Library
  - NAG
  - Sun Performance Library
  - Python (NumPy, SciPy)

- **Research**
  - In FASTMath Tools: Hypre, PETSc, Trilinos, …
  - M3D-C¹, NIMROD (burning plasmas for fusion energies)
  - Omega3P (accelerator design)
  - …
Data structure: Compressed Row Storage (CRS)

- Store nonzeros row by row contiguously
- Example: $N = 7$, $NNZ = 19$
- 3 arrays:
  - Storage: $NNZ$ reals, $NNZ+N+1$ integers

\[
\begin{pmatrix}
1 & a \\
2 & b \\
c & d & 3 \\
e & 4 & f \\
5 & g \\
h & i & 6 & j \\
k & l & 7
\end{pmatrix}
\]

\[
\begin{array}{cccccccccccc}
nzval & 1 & a & 2 & b & c & d & 3 & e & 4 & f & 5 & g & h & i & 6 & j & k & l & 7 \\
\hline
1 & 3 & 5 & 8 & 11 & 13 & 17 & 20
\end{array}
\]

User interface – distribute input matrices

- Matrices involved:
  - A, B (turned into X) – input, users manipulate them
  - L, U – output, users do not need to see them

- A (sparse) and B (dense) are distributed by block rows

\[
\begin{bmatrix}
P0 & A \\
P1 \\
P2 & B \\
\end{bmatrix}
\]

- Local A stored in Compressed Row Format

- Natural for users, and consistent with other popular packages: e.g. PETSc
**Distributed input interface**

- Each process has a structure to store local part of A

Distributed Compressed Row Storage

```c
typedef struct {
    int_t  nnz_loc;  // number of nonzeros in the local submatrix
    int_t  m_loc;    // number of rows local to this processor
    int_t  fst_row;  // global index of the first row
    void  *nzval;   // pointer to array of nonzero values, packed by row
    int_t  *colind; // pointer to array of column indices of the nonzeros
    int_t  *rowptr; // pointer to array of beginning of rows in nzval[]and colind[]
} NRformat_loc;
```
Distributed Compressed Row Storage

A is distributed on 2 processors:

- Processor P0 data structure:
  - nnz_loc = 5
  - m_loc = 2
  - fst_row = 0 // 0-based indexing
  - nzval = { s, u, u, l, u }
  - colind = { 0, 2, 4, 0, 1 }
  - rowptr = { 0, 3, 5 }

- Processor P1 data structure:
  - nnz_loc = 7
  - m_loc = 3
  - fst_row = 2 // 0-based indexing
  - nzval = { l, p, e, u, l, l, r }
  - colind = { 1, 2, 3, 4, 0, 1, 4 }
  - rowptr = { 0, 2, 4, 7 }

P0

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>u</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>u</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P1

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>l</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>e</td>
<td>u</td>
<td></td>
</tr>
</tbody>
</table>

|   | l | l | r |

P0 is distributed on 2 processors.
### 2D block cyclic layout

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Process mesh</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2  0 1 2  0</td>
<td>0 1 2</td>
</tr>
<tr>
<td>3 4 5  3 4 5  3</td>
<td>3 4 5</td>
</tr>
<tr>
<td>0 1 2  0 1 2  0</td>
<td></td>
</tr>
<tr>
<td>0 1 2  0 1 2  0</td>
<td></td>
</tr>
</tbody>
</table>

- **Active**
### Process grid and MPI communicator

- **Example:** Solving a preconditioned linear system
  
  \[ M^{-1}A \mathbf{x} = M^{-1} \mathbf{b} \]

  \[ M = \text{diag}(A_{11}, A_{22}, A_{33}) \]

  → use **SuperLU_DIST** for each diagonal block

- Create 3 process grids, same logical ranks (0:3), but different physical ranks
- Each grid has its own MPI communicator
Two ways to create a process grid

- `superlu_gridinit( MPI_Comm Bcomm, int nprow, int npcol, gridinfo_t *grid );`
  - Maps the first \{nprow, npcol\} processes in the MPI communicator Bcomm to SuperLU 2D grid

- `superlu_gridmap( MPI_Comm Bcomm, int nprow, int npcol, int usermap[], int ldumap, gridinfo_t *grid );`
  - Maps an *arbitrary* set of \{nprow, npcol\} processes in the MPI communicator Bcomm to SuperLU 2D grid. The ranks of the selected MPI processes are given in usermap[] array.

For example:

- \[
\begin{array}{ccc}
0 & 1 & 2 \\
0 & 11 & 12 & 13 \\
1 & 14 & 15 & 16 \\
\end{array}
\]
Review of Gaussian Elimination (GE)

- Solving a system of linear equations $Ax = b$

- First step of GE: (make sure $\alpha$ not too small . . . Otherwise do pivoting)

$$A = \begin{bmatrix}
\alpha & w^T \\
v & B
\end{bmatrix}
= \begin{bmatrix} 1 & 0 \\ v/\alpha & I \end{bmatrix}
\begin{bmatrix}
\alpha & \ w^T \\
0 & C
\end{bmatrix}
$$

$$C = B - \frac{v \cdot w^T}{\alpha}$$

- Repeats GE on $C$

- Results in $\{L\backslash U\}$ decomposition ($A = LU$)
  - $L$ lower triangular with unit diagonal, $U$ upper triangular

- Then, $x$ is obtained by solving two triangular systems with $L$ and $U$
Sparse factorization

- Store A explicitly ... many sparse compressed formats
- “Fill-in” . . . new nonzeros in L & U
  - Typical fill-ratio: 10x for 2D problems, 30-50x for 3D problems
- Graph algorithms: directed/undirected graphs, bipartite graphs, paths, elimination trees, depth-first search, heuristics for NP-hard problems, cliques, graph partitioning, . . .
- Unfriendly to high performance, parallel computing
  - Irregular memory access, indirect addressing, strong task/data dependency

Diagrams showing L and U matrices, and a graph with directed edges illustrating the structure of the graphs.
Graph tool: reachable set, fill-path

Edge (x,y) exists in filled graph $G^+$ due to the path: $x \rightarrow 7 \rightarrow 3 \rightarrow 9 \rightarrow y$

Finding fill-ins $\leftrightarrow$ finding transitive closure of $G(A)$
Algorithmic phases in sparse GE

1. Minimize number of fill-ins, maximize parallelism (~10% time)
   - Sparsity structure of L & U depends on that of A, which can be changed by row/column permutations (vertex re-labeling of the underlying graph)
   - Ordering (combinatorial algorithms; “NP-complete” to find optimum [Yannakis ’83]; use heuristics)

2. Predict the fill-in positions in L & U (~10% time)
   - Symbolic factorization (combinatorial algorithms)

3. Design efficient data structure for storage and quick retrieval of the nonzeros
   - Compressed storage schemes

4. Perform factorization and triangular solutions (~80% time)
   - Numerical algorithms (F.P. operations only on nonzeros)
   - Usually dominate the total runtime

   For sparse Cholesky and QR, the steps can be separate; for sparse LU with pivoting, steps 2 and 4 may be interleaved.
General Sparse Solver

- Use (blocked) CRS or CCS, and any ordering method
  - Leave room for fill-ins! (symbolic factorization)
- Exploit “supernode” (dense) structures in the factors
  - Can use Level 3 BLAS
  - Reduce inefficient indirect addressing (scatter/gather)
  - Reduce graph traversal time using a coarser graph
Numerical Pivoting

- Goal of pivoting is to control element growth in L & U for stability
  - For sparse factorizations, often relax the pivoting rule to trade with better sparsity and parallelism (e.g., threshold pivoting, static pivoting, . . .)

- **Partial pivoting** used in sequential SuperLU and SuperLU_MT (GEPP)
  - Can force diagonal pivoting (controlled by diagonal threshold)
  - Hard to implement scalably for sparse factorization

- **Static pivoting** used in SuperLU_DIST (GESP)
  - Before factor, scale and permute A to maximize diagonal: \( P_r D_r A D_c = A' \)
  - During factor \( A' = LU \), replace tiny pivots by \( \sqrt{\epsilon \|A\|} \), without changing data structures for L & U
  - If needed, use a few steps of iterative refinement after the first solution

\[ \begin{array}{c|c|c|c|c}
  s & x & x & \\
  x & & & \\
  b & x & x & x \\
\end{array} \]
**Ordering: Minimum Degree**

**Local greedy: minimize upper bound on fill-in**

\[
\begin{bmatrix}
1 & x & x & x & x & x \\
1 & x \\
1 & x \\
1 & x \\
\end{bmatrix}
\]

Eliminate 1

\[
\begin{bmatrix}
1 & x & x & x & x & x \\
1 & x \\
1 & x \\
1 & x \\
\end{bmatrix}
\]

Eliminate 1

\[
\begin{bmatrix}
i & j & k & l \\
i & x & x & x & x & x \\
j & x & x & x & x & x \\
k & x & x & x & x & x \\
l & x & x & x & x & x \\
\end{bmatrix}
\]
Ordering: Nested Dissection

- Model problem: discretized system $Ax = b$ from certain PDEs, e.g., 5-point stencil on $n \times n$ grid, $N = n^2$
  - Factorization flops: $O(n^3) = O(N^{3/2})$
- Theorem: ND ordering gives optimal complexity in exact arithmetic [George 73, Hoffman/Martin/Rose]
ND Ordering

- Generalized nested dissection [Lipton/Rose/Tarjan ’79]
  - Global graph partitioning: top-down, divide-and-conquer
  - Best for largest problems
  - Parallel codes available: ParMetis, PT-Scotch
- First level
  - Recurse on A and B
- Goal: find the smallest possible separator S at each level
  - Multilevel schemes:
    - Chaco [Hendrickson/Leland ‘94], Metis [Karypis/Kumar ‘95]
  - Spectral bisection [Simon et al. ‘90-‘95]
  - Geometric and spectral bisection [Chan/Gilbert/Teng ‘94]
ND Ordering

2D mesh

A, with row-wise ordering

A, with ND ordering

L & U factors
Ordering for LU (unsymmetric)

- Can use a symmetric ordering on a symmetrized matrix
  - Case of partial pivoting (serial SuperLU, SuperLU_MT):
    Use ordering based on $A^T A$
  - Case of static pivoting (SuperLU_DIST):
    Use ordering based on $A^T + A$

- Can find better ordering based solely on $A$, without symmetrization
  - Diagonal Markowitz  [Amestoy/Li/Ng `06]
    - Similar to minimum degree, but without symmetrization
  - Hypergraph partition  [Boman, Grigori, et al. `08]
    - Similar to ND on $A^T A$, but no need to compute $A^T A$
Ordering Interface in SuperLU

- Library contains the following routines:
  - Ordering algorithms: MMD [J. Liu], COLAMD [T. Davis]
  - Utility routines: form $A^T + A$, $A^T A$

- Users may input any other permutation vector (e.g., using Metis, Chaco, etc.)

```c
... set_default_options_dist ( &options );
options.ColPerm = MY_PERMC; // modify default option
ScalePermstructInit ( m, n, &ScalePermstruct );
METIS ( . . . , &ScalePermstruct.perm_c );
...
pdgssvx ( &options, . . . , &ScalePermstruct, . . . );
...```
Symbolic Factorization

- **Cholesky** [George/Liu `81 book]
  - Use elimination graph of \( L \) and its transitive reduction (elimination tree)
  - Complexity linear in output: \( O(\text{nnz}(L)) \)

- **LU**
  - Use elimination graphs of \( L \) & \( U \) and their transitive reductions (elimination DAGs) [Tarjan/Rose `78, Gilbert/Liu `93, Gilbert `94]
  - Improved by symmetric structure pruning [Eisenstat/Liu `92]
  - Improved by supernodes
  - Complexity greater than \( \text{nnz}(L+U) \), but much smaller than \( \text{flops}(LU) \)
Numerical Factorization

- Sequential SuperLU
  - Enhance data reuse in memory hierarchy by calling Level 3 BLAS on the supernodes

- SuperLU_MT
  - Exploit both coarse and fine grain parallelism
  - Employ dynamic scheduling to minimize parallel runtime

- SuperLU_DIST
  - Enhance scalability by static pivoting and 2D matrix distribution
**SuperLU_MT** [Li/Demmel/Gilbert]

- Pthread or OpenMP
- **Left-looking** – relatively more READs than WRITEs
- Use shared task queue to schedule ready columns in the elimination tree (bottom up)
- **Over 12x speedup on conventional 16-CPU SMPs (1999)**

![Diagram of SuperLU_MT workflow](image)
SuperLU DIST [Li/Demmel/Grigori/Yamazaki]

- **MPI**
- **Right-looking** – relatively more WRITEs than READs
- **2D block cyclic layout**
- **Look-ahead to overlap comm. & comp.**
- **Scales to 1000s processors**

![Matrix and Process mesh diagram](image)
Multicore platforms

- Intel Clovertown:
  - 2.33 GHz Xeon, 9.3 Gflops/core
  - 2 sockets x 4 cores/socket
  - L2 cache: 4 MB/2 cores

- Sun VictoriaFalls:
  - 1.4 GHz UltraSparc T2, 1.4 Gflops/core
  - 2 sockets x 8 cores/socket x 8 hardware threads/core
  - L2 cache shared: 4 MB
## Benchmark matrices

<table>
<thead>
<tr>
<th>apps</th>
<th>dim</th>
<th>nnz(A)</th>
<th>SLU_MT Fill</th>
<th>SLU_DIST Fill</th>
<th>Avg. S-node</th>
</tr>
</thead>
<tbody>
<tr>
<td>g7jac200</td>
<td>59,310</td>
<td>0.7 M</td>
<td>33.7 M</td>
<td>33.7 M</td>
<td>1.9</td>
</tr>
<tr>
<td>Economic model</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stomach</td>
<td>213,360</td>
<td>3.0 M</td>
<td>136.8 M</td>
<td>137.4 M</td>
<td>4.0</td>
</tr>
<tr>
<td>3D finite diff.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>torso3</td>
<td>259,156</td>
<td>4.4 M</td>
<td>784.7 M</td>
<td>785.0 M</td>
<td>3.1</td>
</tr>
<tr>
<td>3D finite diff.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>twotone</td>
<td>120,750</td>
<td>1.2 M</td>
<td>11.4 M</td>
<td>11.4 M</td>
<td>2.3</td>
</tr>
<tr>
<td>Nonlinear analog circuit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Maximun speedup 4.3, smaller than conventional SMP
Pthreads scale better
Question: tools to analyze resource contention?
SuperLU tutorial

SunVictoriaFalls – multicore + multithread

- Maximum speedup 20
- Pthreads more robust, scale better
- MPICH crashes with large #tasks, mismatch between coarse and fine grain models
### Performance of larger matrices

| Name       | Application                      | Data type | N        | \(|A| / N\) Sparsity | \(|L\backslash U|\) (10^6) | Fill-ratio |
|------------|----------------------------------|-----------|----------|---------------------|-----------------|-------------|
| matrix211  | Fusion, MHD eqns (M3D-C1)        | Real      | 801,378  | 161                 | 1276.0          | 9.9         |
| cc_linear2 | Fusion, MHD eqns (NIMROD)       | Complex   | 259,203  | 109                 | 199.7           | 7.1         |
| matick     | Circuit sim. MNA method (IBM)    | Complex   | 16,019   | 4005                | 64.3            | 1.0         |
| cage13     | DNA electrophoresis              | Real      | 445,315  | 17                  | 4550.9          | 608.5       |

- **Sparsity ordering:** MeTis applied to structure of A’ + A
Strong scaling (fixed size): Cray XE6 (hopper@nersc)

- 2 x 12-core AMD 'MagnyCours' per node, 2.1 GHz processor

Up to 1.4 Tflops factorization rate
SuperLU_DIST 3.0: better DAG scheduling

- Implemented new static scheduling and flexible look-ahead algorithms that shortened the length of the critical path.
- Idle time was significantly reduced (speedup up to 2.6x)
- To further improve performance:
  - more sophisticated scheduling schemes
  - hybrid programming paradigms

Accelerator, n=2.7M, fill-ratio=12
DNA, n = 445K, fill-ratio=609
Multicore / GPU-Aware

New hybrid programming code: MPI+OpenMP+CUDA, able to use all the CPUs and GPUs on manycore computers.

Algorithmic changes:
- Aggregate small BLAS operations into larger ones.
- CPU multithreading Scatter/Gather operations.
- Hide long-latency operations.

Results: using 100 nodes GPU clusters, up to 2.7x faster, 2x-5x memory saving.

New SuperLU_DIST 4.0 release, August 2014.
CPU + GPU algorithm

① Aggregate small blocks
② GEMM of large blocks
③ Scatter

GPU acceleration:
Software pipelining to overlap GPU execution with CPU Scatter, data transfer.
ILU Interface

- Available in serial SuperLU 4.0, June 2009
- Similar to ILUTP [Saad]: “T” = threshold, “P” = pivoting
  - among the most sophisticated, more robust than structure-based dropping (e.g., level-of-fill)
- ILU driver: SRC/dgsisx.c
- ILU factorization routine: SRC/dgsitrf.c
- GMRES driver: EXAMPLE/ditersol.c
- Parameters:
  - ilu_set_default_options ( &options )
    - options.ILU_DropTol – numerical threshold (τ)
    - options.ILU_FillFactor – bound on the fill-ratio (γ)
Result of Supernodal ILU (S-ILU)

- New dropping rules S-ILU(τ, γ)
  - supernode-based thresholding (τ)
  - adaptive strategy to meet user-desired fill-ratio upper bound (γ)

- Performance of S-ILU
  - For 232 test matrices, S-ILU + GMRES converges with 138 cases (~60% success rate)
  - S-ILU + GMRES is 1.6x faster than scalar ILU + GMRES
### S-ILU for extended MHD (fusion energy sim.)

- **AMD Opteron 2.4 GHz (Cray XT5)**
- **ILU parameters:** $\tau = 10^{-4}$, $\Upsilon = 10$
- **Up to 9x smaller fill ratio, and 10x faster**

<table>
<thead>
<tr>
<th>Problems</th>
<th>order</th>
<th>Nonzeros (millions)</th>
<th>SuperLU Time</th>
<th>SuperLU fill-ratio</th>
<th>S-ILU time</th>
<th>S-ILU fill-ratio</th>
<th>GMRES Time</th>
<th>GMRES Iters</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix31</td>
<td>17,298</td>
<td>2.7 m</td>
<td>33.3</td>
<td>13.1</td>
<td>8.2</td>
<td>2.7</td>
<td>0.6</td>
<td>9</td>
</tr>
<tr>
<td>matrix41</td>
<td>30,258</td>
<td>4.7 m</td>
<td>111.1</td>
<td>17.5</td>
<td>18.6</td>
<td>2.9</td>
<td>1.4</td>
<td>11</td>
</tr>
<tr>
<td>matrix61</td>
<td>66,978</td>
<td>10.6 m</td>
<td>612.5</td>
<td>26.3</td>
<td>54.3</td>
<td>3.0</td>
<td>7.3</td>
<td>20</td>
</tr>
<tr>
<td>matrix121</td>
<td>263,538</td>
<td>42.5 m</td>
<td>x</td>
<td>x</td>
<td>145.2</td>
<td>1.7</td>
<td>47.8</td>
<td>45</td>
</tr>
<tr>
<td>matrix181</td>
<td>589,698</td>
<td>95.2 m</td>
<td>x</td>
<td>x</td>
<td>415.0</td>
<td>1.7</td>
<td>716.0</td>
<td>289</td>
</tr>
</tbody>
</table>
**Tips for Debugging Performance**

- Check sparsity ordering
- Diagonal pivoting is preferable
  - E.g., matrix is diagonally dominant, . . .

- Need good BLAS library (vendor, ATLAS, GOTO, . . .)
  - May need adjust block size for each architecture
    (Parameters modifiable in routine `sp_ienv()`)
    - Larger blocks better for uniprocessor
    - Smaller blocks better for parallelism and load balance

- Open problem: automatic tuning for block size?
Summary

- Sparse LU, ILU are important kernels for science and engineering applications, used in practice on a regular basis
- Performance more sensitive to latency than dense case
- Continuing developments funded by DOE SciDAC projects
  - Integrate into more applications
  - Hybrid model of parallelism for multicore/vector nodes, differentiate intra-node and inter-node parallelism
    - Hybrid programming models, hybrid algorithms
  - Parallel HSS preconditioners
  - Parallel hybrid direct-iterative solver based on domain decomposition
Exercises of SuperLU_DIST


- On vesta:
  /gpfs/vesta-fs0/projects/FASTMath/ATPESC-2014/examples/superlu
  /gpfs/vesta-fs0/projects/FASTMath/ATPESC-2014/install/superlu

Examples in EXAMPLE/

- **pddrive.c**: Solve one linear system
- **pddrive1.c**: Solve the systems with same A but different right-hand side at different times
  - Reuse the factored form of A
- **pddrive2.c**: Solve the systems with the same pattern as A
  - Reuse the sparsity ordering
- **pddrive3.c**: Solve the systems with the same sparsity pattern and similar values
  - Reuse the sparsity ordering and symbolic factorization
- **pddrive4.c**: Divide the processes into two subgroups (two grids) such that each subgroup solves a linear system independently from the other.
**SuperLU_DIST Example Program**

- EXAMPLE/pddrive.c

- Five basic steps
  1. Initialize the MPI environment and SuperLU process grid
  2. Set up the input matrices A and B
  3. Set the options argument (can modify the default)
  4. Call SuperLU routine PDGSSVX
  5. Release the process grid, deallocate memory, and terminate the MPI environment
**Fortran 90 Interface in FORTRAN/**

- All SuperLU objects (e.g., LU structure) are **opaque** for F90
  - They are allocated, deallocated and operated in the C side and not directly accessible from Fortran side.
- C objects are accessed via **handles** that exist in Fortran’s user space
- In Fortran, all handles are of type INTEGER
- **Example:** FORTRAN/f_5x5.f90

\[
A = \begin{bmatrix}
  s & u & u \\
  l & u & \\
  l & p & \\
  l & e & u \\
  l & l & r
\end{bmatrix},\quad s = 19.0,\ u = 21.0,\ p = 16.0,\ e = 5.0,\ r = 18.0,\ l = 12.0
\]
Exercises of SuperLU_DIST

- [Link](https://redmine.scorec.rpi.edu/anonsvn/fastmath/docs/ATPESC_2014/Exercises/superlu/README.html)

- On vesta:
  /gpfs/vesta-fs0/projects/FASTMath/ATPESC-2014/examples/superlu
  /gpfs/vesta-fs0/projects/FASTMath/ATPESC-2014/install/superlu

- [Link](http://crd.lbl.gov/~xiaoye/SuperLU/slu_hands_on.html)
Examples in EXAMPLE/

- `pddrive.c`: Solve one linear system
- `pddrive1.c`: Solve the systems with same A but different right-hand side at different times
  - Reuse the factored form of A
- `pddrive2.c`: Solve the systems with the same pattern as A
  - Reuse the sparsity ordering
- `pddrive3.c`: Solve the systems with the same sparsity pattern and similar values
  - Reuse the sparsity ordering and symbolic factorization
- `pddrive4.c`: Divide the processes into two subgroups (two grids) such that each subgroup solves a linear system independently from the other.
SuperLU_DIST Example Program

- EXAMPLE/pddrive.c

- Five basic steps
  1. Initialize the MPI environment and SuperLU process grid
  2. Set up the input matrices A and B
  3. Set the options argument (can modify the default)
  4. Call SuperLU routine PDGSSVX
  5. Release the process grid, deallocate memory, and terminate the MPI environment
Fortran 90 Interface in FORTRAN/

- All SuperLU objects (e.g., LU structure) are **opaque** for F90
  - They are allocated, deallocated and operated in the C side and not directly accessible from Fortran side.

- C objects are accessed via **handles** that exist in Fortran’s user space

- In Fortran, all handles are of type INTEGER

- Example: FORTRAN/f_5x5.f90

\[
A = \begin{bmatrix}
s & u & u \\
l & u \\
l & p \\
e & u \\
l & l & r \\
\end{bmatrix}, \quad s = 19.0, \quad u = 21.0, \quad p = 16.0, \quad e = 5.0, \quad r = 18.0, \quad l = 12.0
\]