

Sparse Matrix Methods on High Performance Computers

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

CS267/EngC233: Applications of Parallel Computers
March 16, 2010

- ❖ Solving a system of linear equations $Ax = b$
- ❖ Iterative methods
 - A is not changed (read-only)
 - Key kernel: sparse matrix-vector multiply
 - Easier to optimize and parallelize
 - Low algorithmic complexity, but may not converge for hard problems
- ❖ 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
- ❖ Increasingly more interest on hybrid methods

❖ Direct methods . . . Sparse factorization

- Sparse compressed formats
- Deal with many graph algorithms: directed/undirected graphs, paths, elimination trees, depth-first search, heuristics for NP-hard problems, cliques, graph partitioning, cache-friendly dense matrix kernels, and more . . .

❖ Preconditioners . . . Incomplete factorization

❖ Hybrid method . . . Domain decomposition

❖ Survey of different types of factorization codes

<http://crd.lbl.gov/~xiaoye/SuperLU/SparseDirectSurvey.pdf>

- LL^T (s.p.d.)
- LDL^T (symmetric indefinite)
- LU (nonsymmetric)
- QR (least squares)
- Sequential, shared-memory (multicore), distributed-memory, out-of-core

❖ Distributed-memory codes: usually MPI-based

- **SuperLU_DIST** [Li/Demmel/Grigori]
 - accessible from PETSc, Trilinos
- MUMPS, PaSTiX, WSMP, ...

- ◆ First step of GE:

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

- ◆ Repeats GE on C
- ◆ Results in LU factorization ($A = LU$)
 - L lower triangular with unit diagonal, U upper triangular
- ◆ Then, x is obtained by solving two triangular systems with L and U

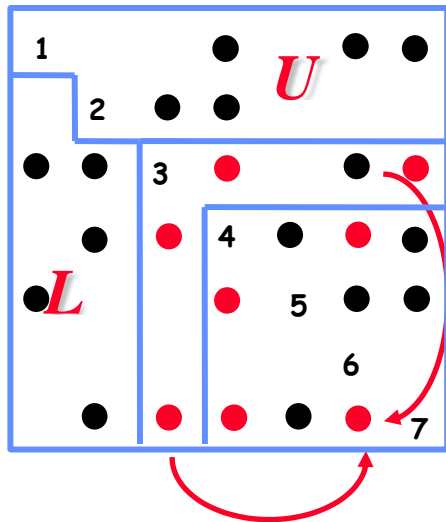
- ## ❖ Sparse matrices are ubiquitous

➤ Example: A of dimension 10^6 , 10~100 nonzeros per row

❖ Nonzero costs flops and memory

❖ Scalar algorithm: 3 nested loops

- Can re-arrange loops to get different variants: left-looking, right-looking, ...



```

for i = 1 to n
    column_scale ( A(:,i) )
    for k = i+1 to n s.t. A(i,k) != 0
        for j = i+1 to n s.t. A(j,i) != 0
            A(j,k) = A(j,k) - A(j,i) * A(i,k)
        end for
    end for
end for

```

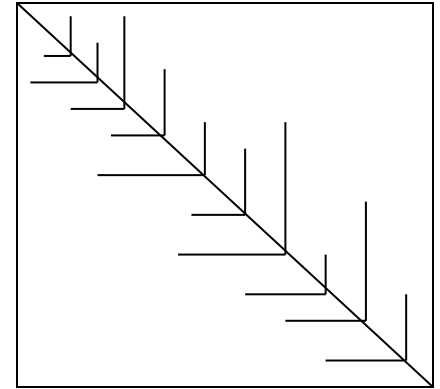
Typical fill-ratio: 10x for 2D problems, 30-50x for 3D problems

❖ Define bandwidth for each row or column

- A little more sophisticated than band solver

❖ Use Skyline storage (SKS)

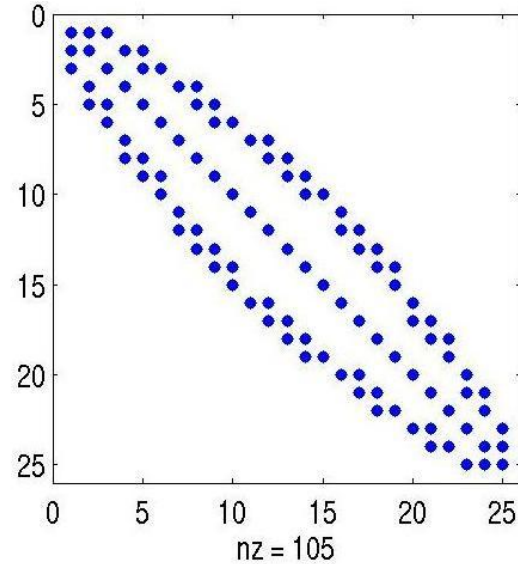
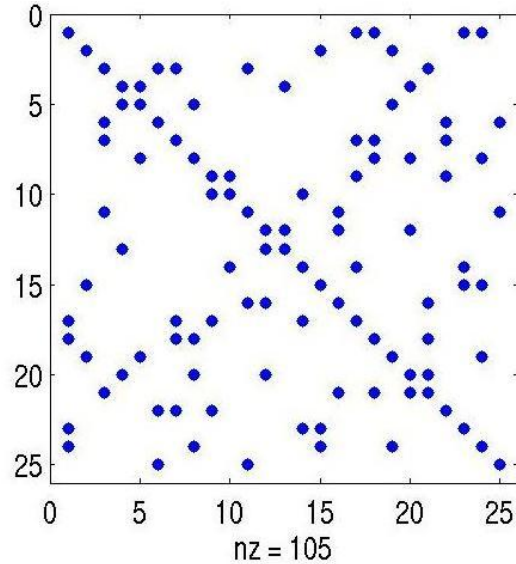
- Lower triangle stored row by row
Upper triangle stored column by column
- In each row (column), first nonzero defines a profile
- All entries within the profile (some may be zeros) are stored
- All fill-ins are confined in the profile



❖ A good ordering would be based on bandwidth reduction

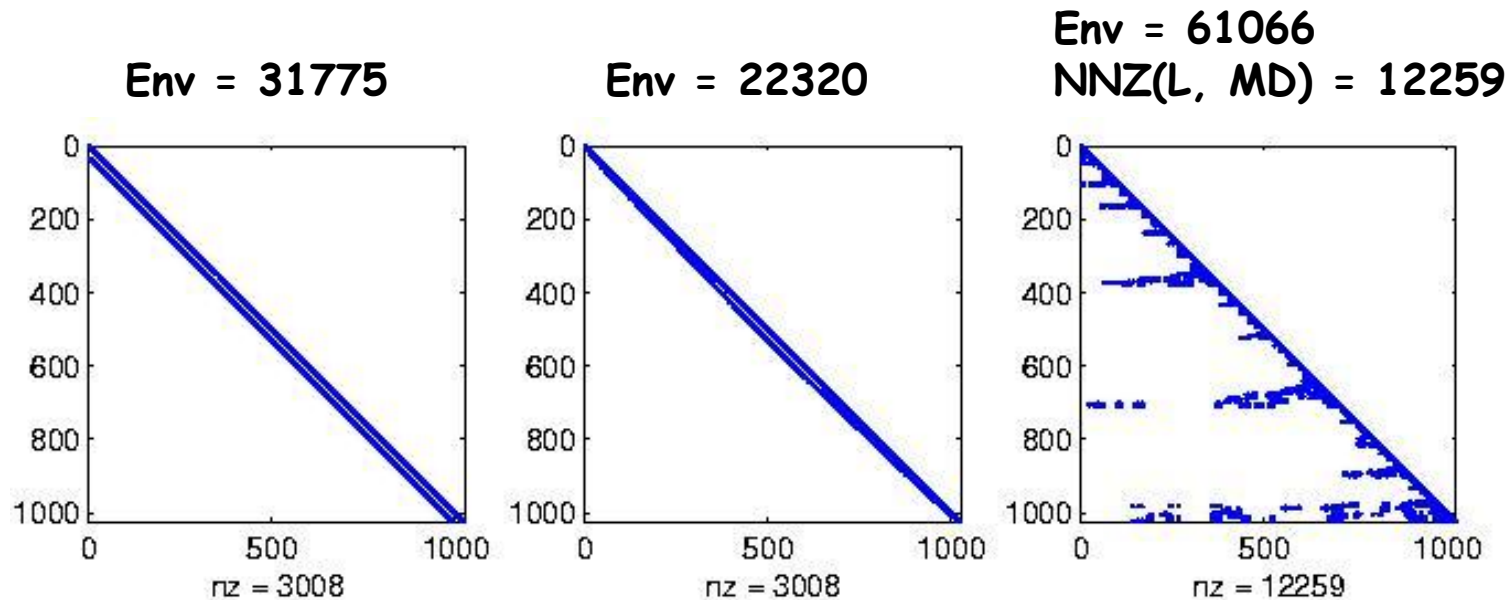
- E.g., (reverse) Cuthill-McKee

❖ Breadth-first search, numbering by levels, then reverse



Is Profile Solver Good Enough?

- ♦ Example: 3 orderings (natural, RCM, Minimum-degree)
- ♦ Envelop size = sum of bandwidths
- ♦ After LU, envelop would be entirely filled



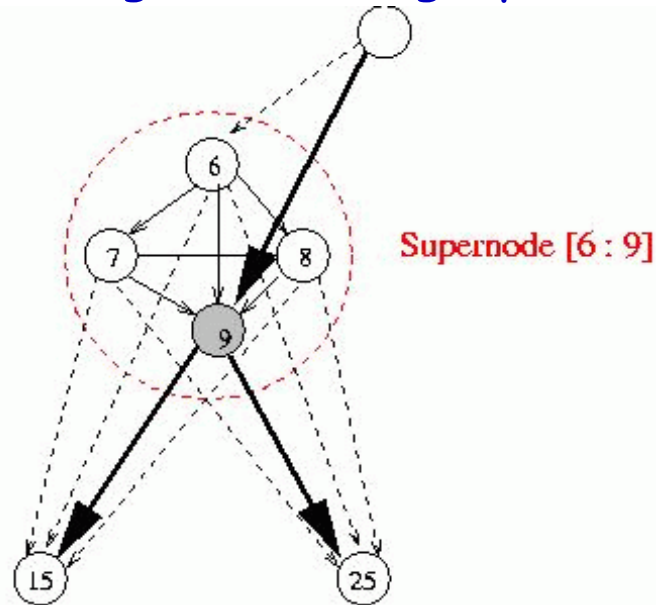
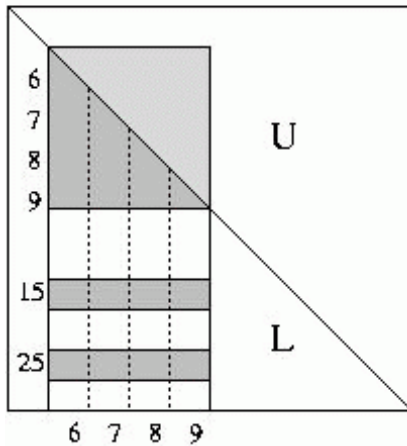
- ◆ Also known as **Harwell-Boeing** format
- ◆ Store nonzeros columnwise contiguously
- ◆ 3 arrays:
 - Storage: NNZ reals, NNZ+N+1 integers
- ◆ Efficient for columnwise algorithms

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

nzval	1 c	2 d e	3 k	a 4 h	b f 5 i l	6	g j 7
rowind	1 3	2 3 4	3 7	1 4 6	2 4 5 6 7	6	5 6 7
colptr	1 3 6 8 11 16 17 20						

- ◆ *"Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods", R. Barrett et al.*

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



- ◆ One step of GE:

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

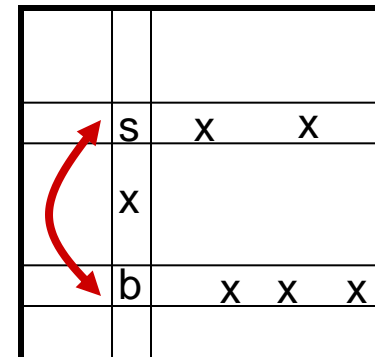
- ◆ $C = B - \frac{v \cdot w^T}{\alpha}$
 - If α is small, some entries in B may be lost from addition
- ◆ **Pivoting**: swap the current diagonal entry with a larger entry from the other part of the matrix
- ◆ Goal: prevent C from getting too large

❖ Dense GE: $P_r A P_c = LU$

- P_r and P_c are permutations chosen to maintain stability
- Partial pivoting suffices in most cases : $P_r A = LU$

❖ Sparse GE: $P_r A P_c = LU$

- P_r and P_c are chosen to maintain stability **and preserve sparsity, and increase parallelism**
- Dynamic pivoting causes dynamic structural change
 - **Alternatives: threshold pivoting, static pivoting, . . .**



	s	x	x	
	x			
	b	x	x	x

❖ Minimize number of fill-ins, maximize parallelism

- 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)

❖ Predict the fill-in positions in L & U

- **Symbolic factorization** (combinatorial algorithms)

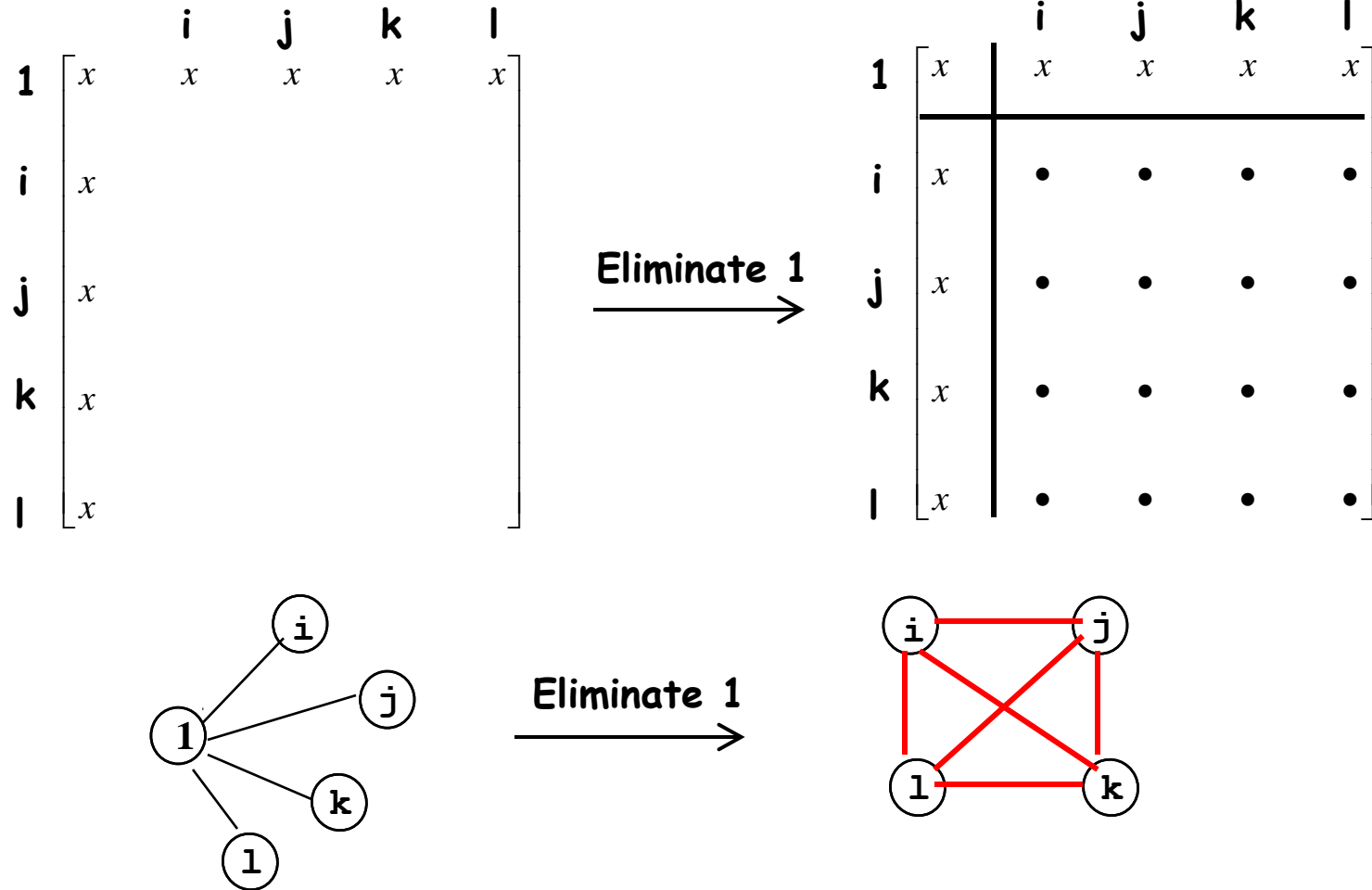
❖ Perform factorization and triangular solutions

- **Numerical algorithms** (F.P. operations only on nonzeros)
 - **How and when to pivot ?**
- Usually dominate the total runtime

- ❖ RCM is good for profile solver
- ❖ General unstructured methods:
 - Minimum degree (locally greedy)
 - Nested dissection (divided-conquer, suitable for parallelism)

Ordering : Minimum Degree (1/3)

Local greedy: minimize upper bound on fill-in



❖ At each step

- Eliminate the vertex with the smallest degree
- Update degrees of the neighbors

❖ Greedy principle: do the best locally

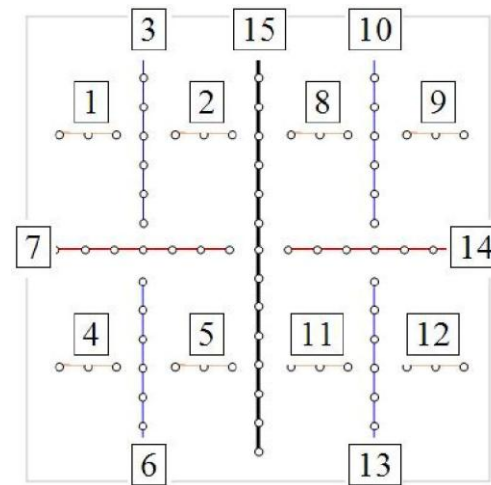
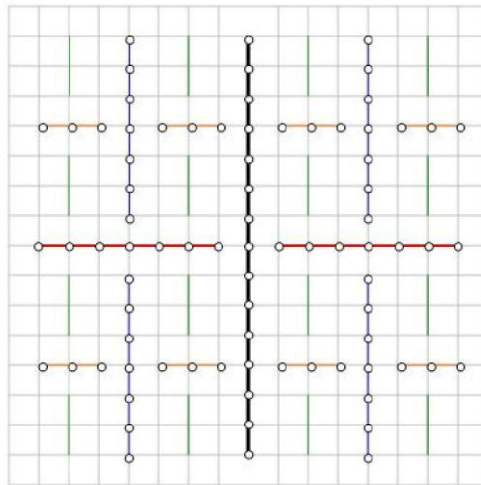
- Best for modest size problems
- Hard to parallelize

❖ Straightforward implementation is slow and requires too much memory

- Newly added edges are more than eliminated vertices

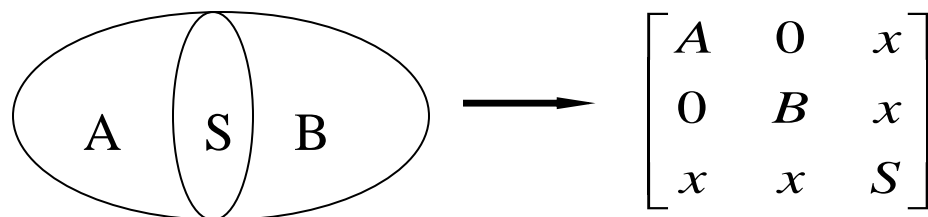
- ❖ Use **quotient graph** as a compact representation [George/Liu '78]
- ❖ Collection of cliques resulting from the eliminated vertices affects the degree of an uneliminated vertex
- ❖ Represent each connected component in the eliminated subgraph by a single "supervertex"
- ❖ **Storage required to implement QG model is bounded by size of A**
- ❖ Large body of literature on implementation variants
 - Tinney/Walker '67, George/Liu '79, Liu '85, Amestoy/Davis/Duff '94, Ashcraft '95, Duff/Reid '95, et al., . .

- ❖ Model problem: discretized system $Ax = b$ from certain PDEs, e.g., 5-point stencil on $k \times k$ grid, $N = k^2$
- ❖ Theorem: ND ordering gave optimal complexity in exact arithmetic [George '73, Hoffman/Martin/Rose, Eisenstat, Schultz and Sherman]
 - 2D ($k \times k = N$ grids): $O(N \log N)$ memory, $O(N^{3/2})$ operations
 - 3D ($k \times k \times k = N$ grids): $O(N^{4/3})$ memory, $O(N^2)$ operations



◆ Generalized nested dissection [Lipton/Rose/Tarjan '79]

- Global graph partitioning: top-down, divide-and-conquer
- Best for largest problems
- Parallel codes available: e.g., ParMetis, Scotch
- First level

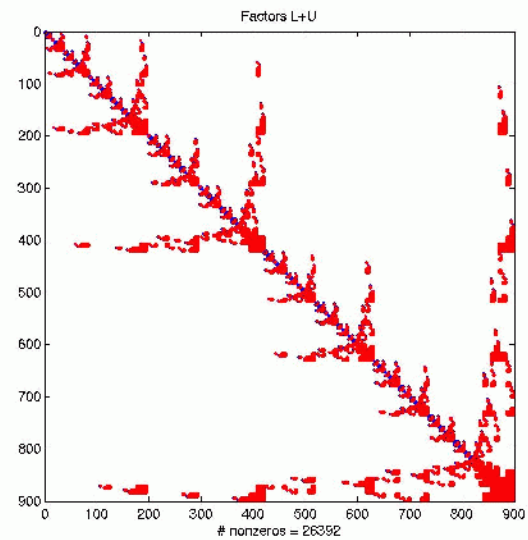
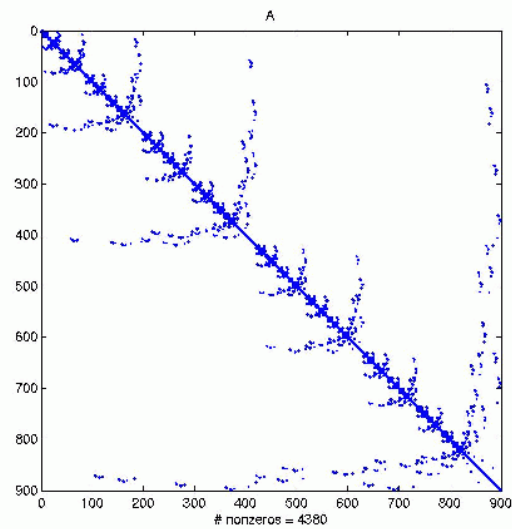
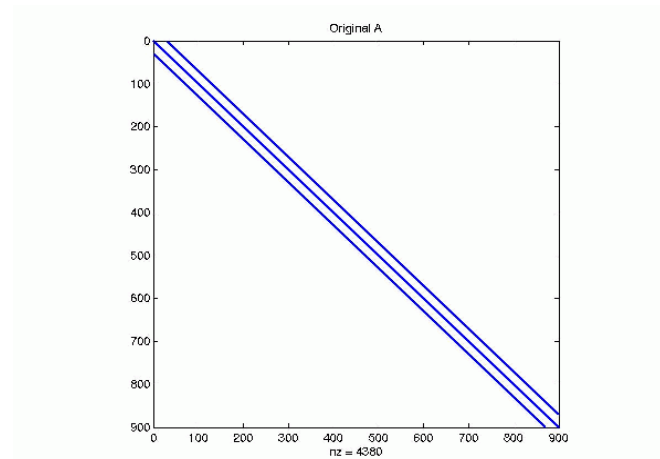


- Recurse on A and B

◆ 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 (3/3)



- ❖ Can use a symmetric ordering on a symmetrized matrix
 - Case of partial pivoting (sequential SuperLU):
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
 - Diagonal Markowitz [Amestoy/Li/Ng '06]
 - Similar to minimum degree, but without symmetrization
 - Hypergraph partition [Boman, Grigori, et al., '09]
 - Similar to ND on $A^T A$, but no need to compute $A^T A$

High Performance Issues: *Reduce Cost of Memory Access & Communication*

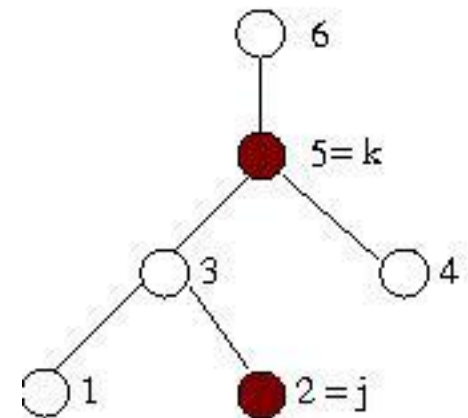


- ❖ Blocking to increase flops-to-bytes ratio
- ❖ Aggregate small messages into one larger message
 - Reduce cost due to latency
- ❖ Well done in LAPACK, ScaLAPACK
 - Dense and banded matrices
- ❖ Adopted in the new generation sparse software
 - Performance much more sensitive to latency in sparse case

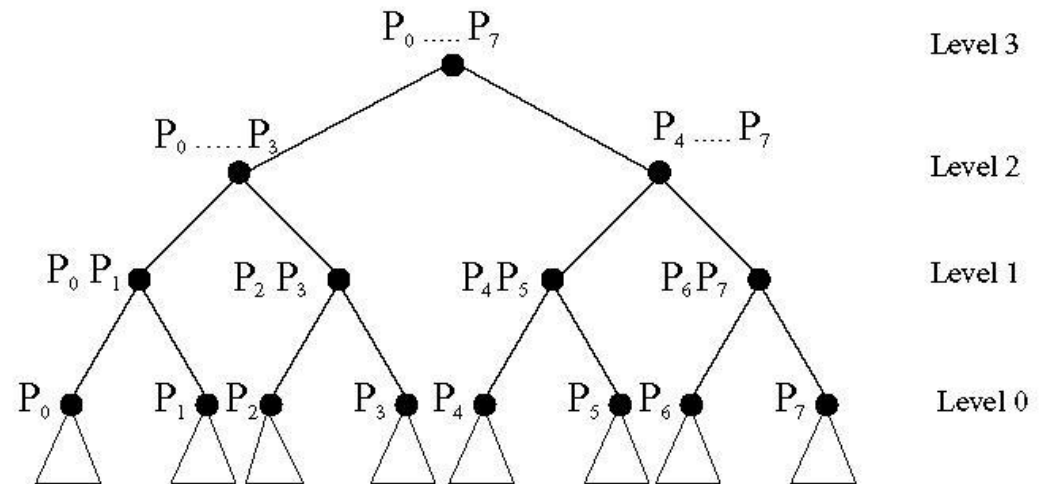
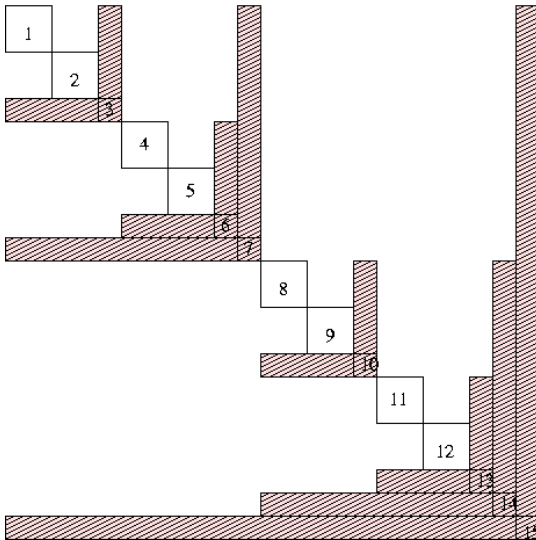
Source of parallelism (1): Elimination Tree



- ♦ For any ordering . . .
- ♦ A column \longleftrightarrow a vertex in the tree
- ♦ Exhibits column dependencies during elimination
 - If column j updates column k , then vertex j is a descendant of vertex k
 - Disjoint subtrees can be eliminated in parallel
- ♦ Almost linear algorithm to compute the tree

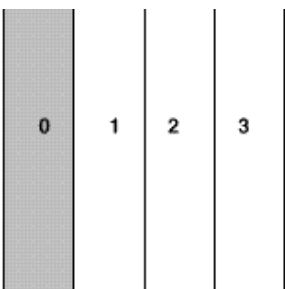


◆ Ordering by graph partitioning

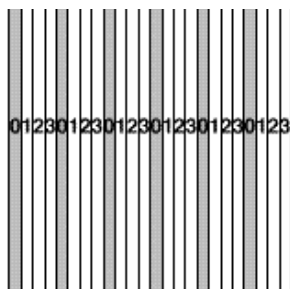




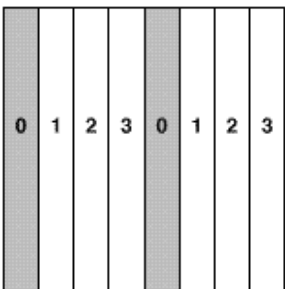
1D blocked



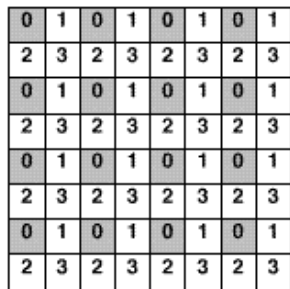
1D cyclic



1D block cyclic



2D block cyclic



- ❖ 2D block cyclic recommended for many linear algebra algorithms
 - Better load balance, less communication, and BLAS-3

Major stages of sparse LU

1. Ordering
2. Symbolic factorization
3. Numerical factorization - usually dominates total time
 - How to pivot?
4. Triangular solutions

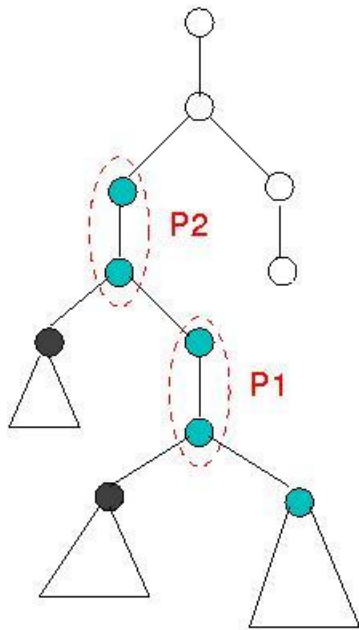
SuperLU_MT

1. Sparsity ordering
2. Factorization (steps interleave)
 - Partial pivoting
 - Symb. fact.
 - Num. fact. (BLAS 2.5)
3. Solve

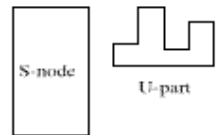
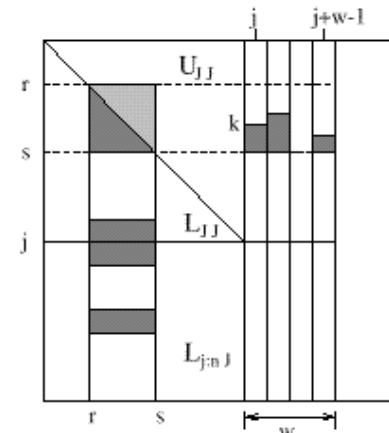
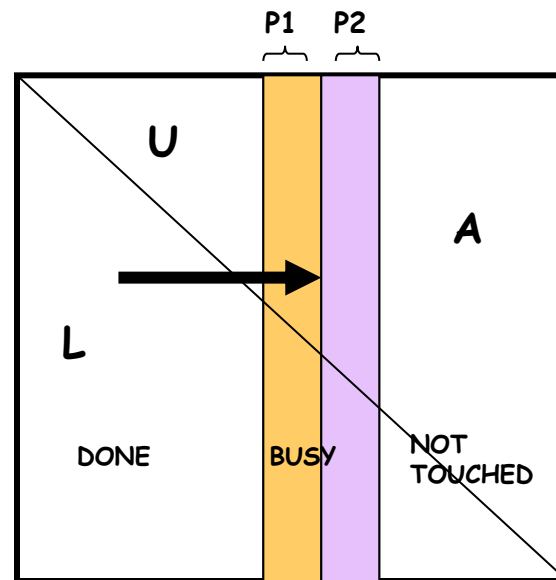
SuperLU_DIST

1. Static pivoting
2. Sparsity ordering
3. Symbolic fact.
4. Numerical fact. (BLAS 3)
5. Solve

- ❖ Pthreads or OpenMP
- ❖ Left looking -- many more reads than writes
- ❖ Use shared task queue to schedule ready columns in the elimination tree (bottom up)

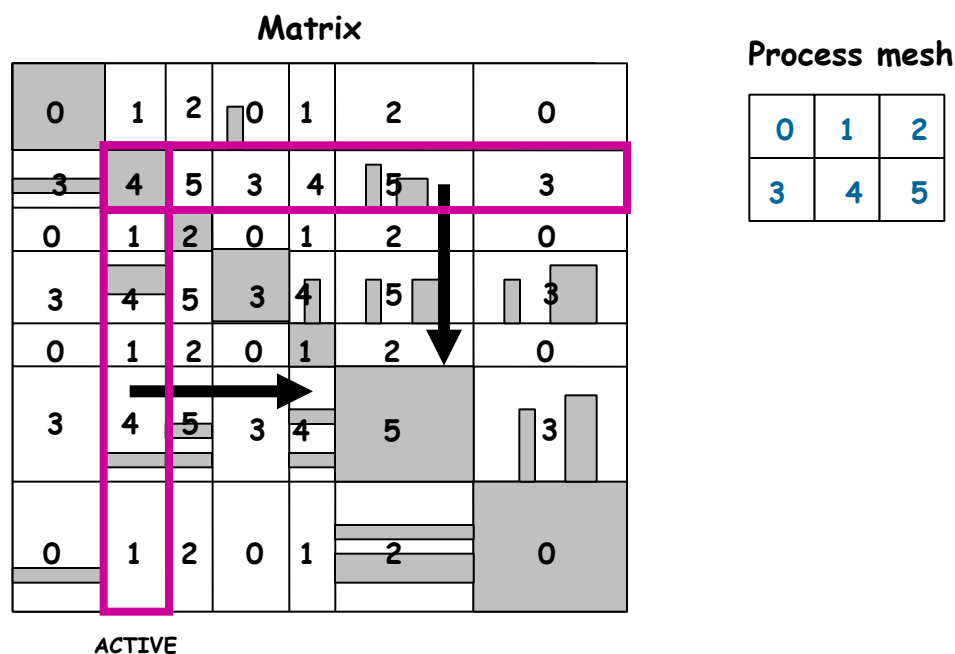


● DONE
● BUSY



❖ MPI

- ❖ Right looking -- many more writes than reads
- ❖ Global 2D block cyclic layout, compressed blocks
- ❖ One step look-ahead to overlap comm. & comp.



❖ Intel Clovertown:

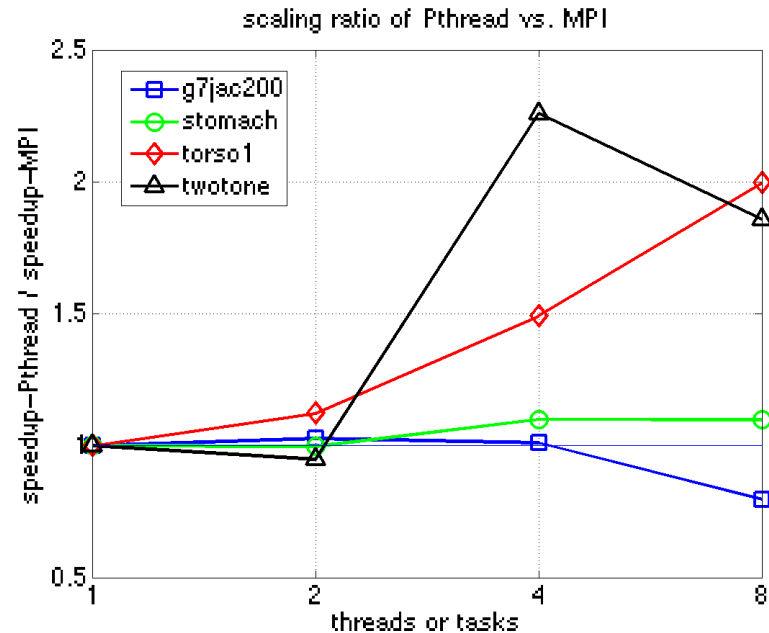
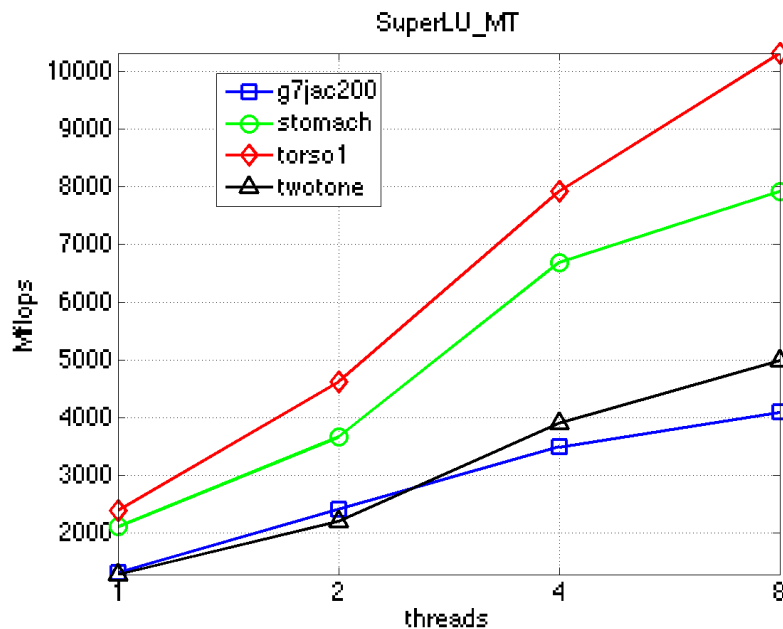
- 2.33 GHz Xeon, 9.3 Gflops/core
- 2 sockets X 4 cores/socket
- L2 cache: 4 MB/2 cores

❖ Sun VictoriaFalls:

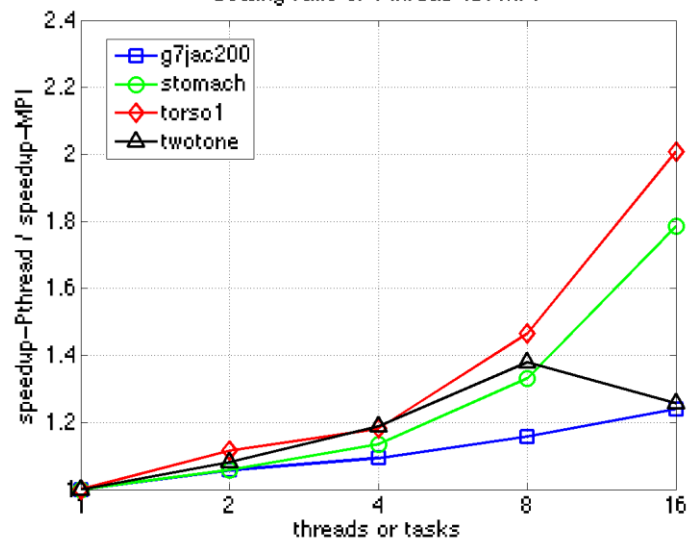
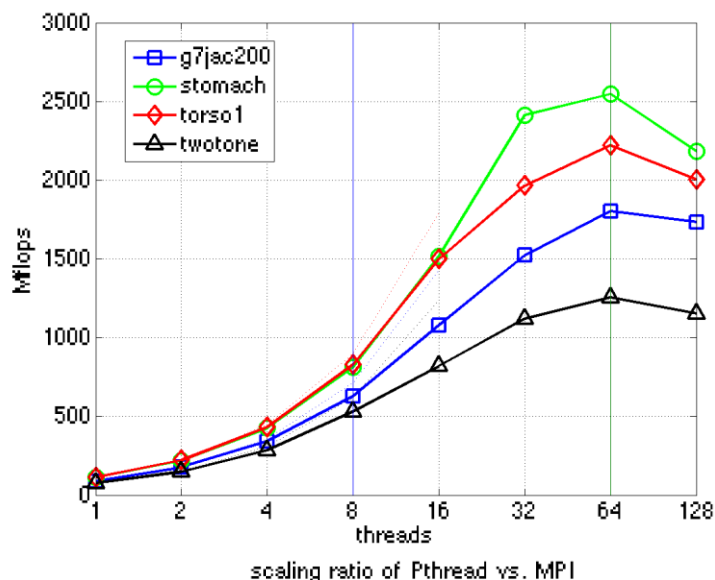
- 1.4 GHz UltraSparc T2, 1.4 Gflops/core
- 2 sockets X 8 cores/socket X 8 hardware threads/core
- L2 cache shared: 4 MB

	apps	dim	nnz(A)	SLU_MT Fill	SLU_DIST Fill	Avg. S-node
g7jac200	Economic model	59,310	0.7 M	33.7 M	33.7 M	1.9
stomach	3D finite diff.	213,360	3.0 M	136.8 M	137.4 M	4.0
torso3	3D finite diff.	259,156	4.4 M	784.7 M	785.0 M	3.1
twotone	Nonlinear analog circuit	120,750	1.2 M	11.4 M	11.4 M	2.3

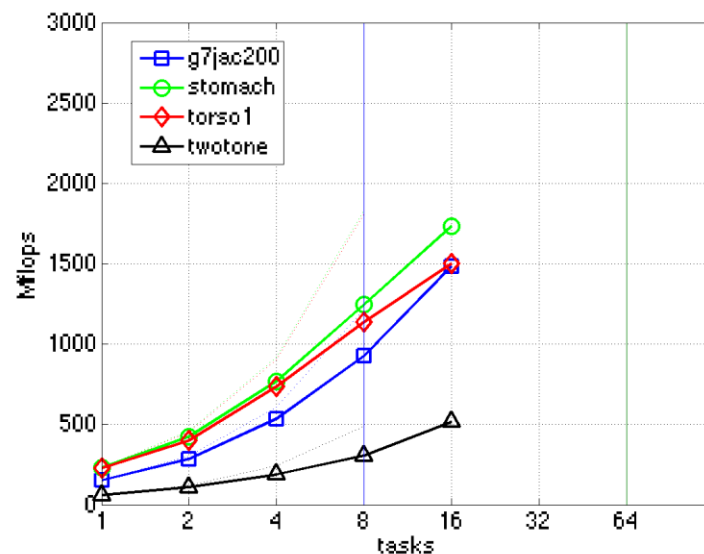
- ❖ Maximum speedup 4.3, smaller than conventional SMP
- ❖ Pthreads scale better
- ❖ Question: tools to analyze resource contention



SuperLU_MT



SuperLU_DIST



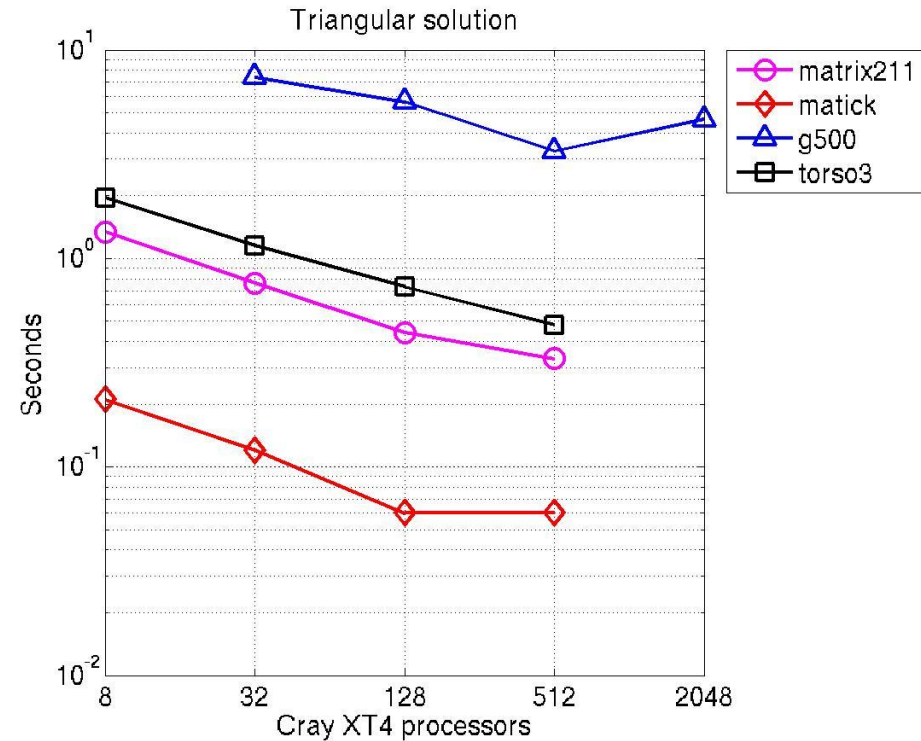
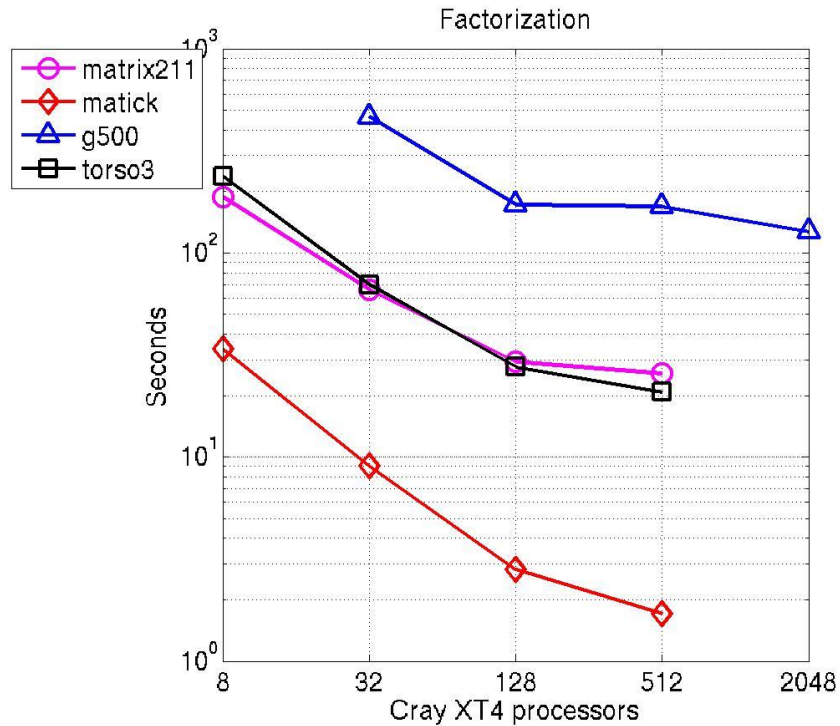
- Maximum speedup 20
- Pthreads more robust, scale better
- MPICH crashes with large #tasks, mismatch between coarse and fine grain models

Larger matrices

Name	Application	Data type	N	A / N Sparsity	L\U (10 ⁶)	Fill-ratio
g500	Quantum Mechanics (LBL)	Complex	4,235,364	13	3092.6	56.2
matrix181	Fusion, MHD eqns (PPPL)	Real	589,698	161	888.1	9.3
dds15	Accelerator, Shape optimization (SLAC)	Real	834,575	16	526.6	40.2
matck	Circuit sim. MNA method (IBM)	Complex	16,019	4005	64.3	1.0

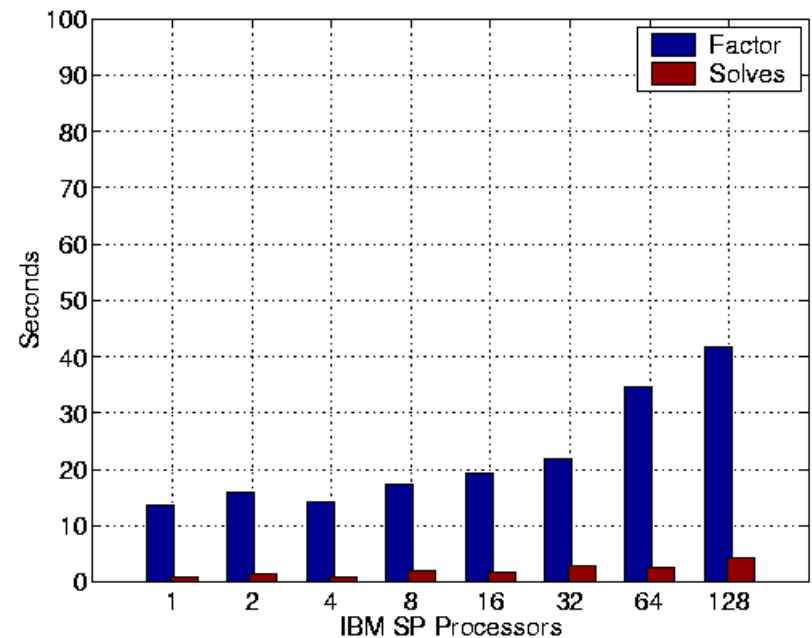
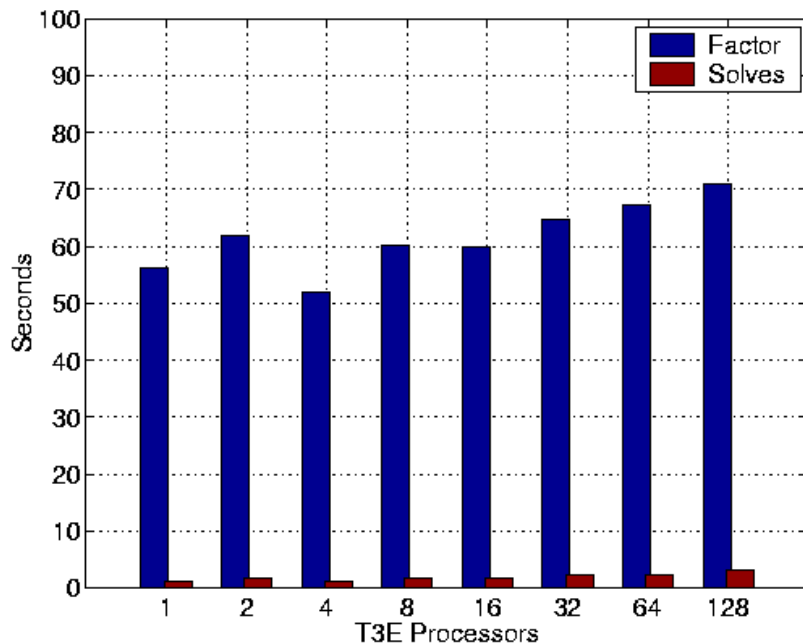
❖ Sparsity ordering: MeTis applied to structure of $A'+A$

Strong scaling: Cray XT4 (2.3 GHz)



❖ Up to 794 Gflops factorization rate

- ♦ 3D KxKxK cubic grids, scale $N^2 = K^6$ with P for constant-work-per-processor
- ♦ Performance sensitive to communication latency
 - Cray T3E latency: 3 microseconds (~ 2700 flops, 450 MHz, 900 Mflops)
 - IBM SP latency: 8 microseconds (~ 11940 flops, 1.9 GHz, 7.6 Gflops)



- ♦ Model problem: matrix from 11 pt Laplacian on $k \times k \times k$ (3D) mesh;
Nested dissection ordering
 - $N = k^3$
 - Factor nonzeros (**Memory**) : $O(N^{4/3})$
 - Number of flops (**Work**) : $O(N^2)$
 - Total communication overhead : $O(N^{4/3} \sqrt{P})$
(assuming P processors arranged as $\sqrt{P} \times \sqrt{P}$ grid)

- ♦ Isoefficiency function: Maintain constant efficiency if "**Work**" increases proportionally with "**Overhead**": $N^2 = c \cdot N^{4/3} \sqrt{P}$, for some const. c
This is equivalent to:
 - Memory-processor relation: $N^{4/3} = c^2 \cdot P$
 - Parallel efficiency can be kept constant if the memory-per-processor is constant, same as dense LU in ScaLAPACK
 - Work-processor relation: $N^2 = c^3 \cdot P^{3/2}$
 - Work needs to grow faster than processors

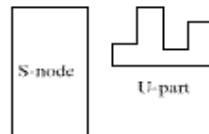
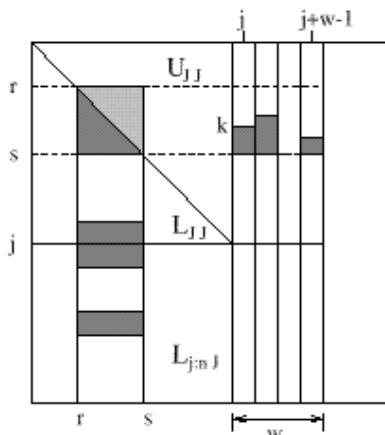
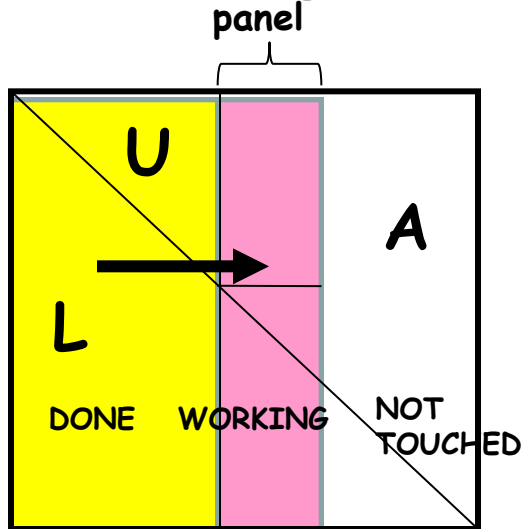
- A very simplified view:

$A = \tilde{L}\tilde{U} + E$, if $\|E\|$ small, $(\tilde{L}\tilde{U})^{-1}A$ may be well conditioned

Then, solve $(\tilde{L}\tilde{U})^{-1}Ax = (\tilde{L}\tilde{U})^{-1}b$ iteratively

- Structure-based dropping: level of fill
 - ILU(0), ILU(k)
 - Rationale: the higher the level, the smaller the entries
 - Separate symbolic factorization step to determine fill-in pattern
- Value-based dropping: drop truly small entries
 - Fill-in pattern must be determined on-the-fly
- ILUTP [Saad]: among the most sophisticated, and (arguably) robust
 - "T" = threshold, "P" = pivoting
 - Implementation similar to direct solver
- We use SuperLU code base to perform ILUTP

• Left-looking, supernode



1. Sparsity ordering of columns
use graph of A^*A

2. Factorization

For each panel ...

- Partial pivoting
- Symbolic fact.
- Num. fact. (BLAS 2.5)

3. Triangular solve

❖ Similar to ILUTP, adapted to supernode

1. U-part:

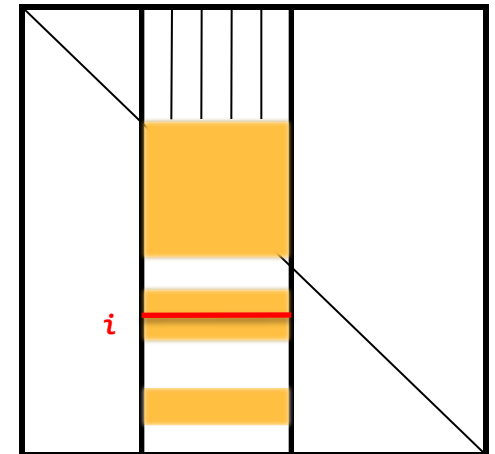
If $|u_{ij}| < \tau \cdot \|A(:, j)\|_{\infty}$, then set $u_{ij} = 0$

2. L-part: retain supernode

Supernode $L(:, s:t)$, if $\|L(i, s:t)\|_{\infty} < \tau$, then set the entire i -th row to zero

❖ Compare with scalar ILU(tau)

- For 54 matrices, S-ILU+GMRES converged with 47 cases, versus 43 with scalar ILU+GMRES
- S-ILU +GMRES is 2.3x faster than scalar ILU+GMRES



❖ Control fill ratio with a user-desired upper bound γ

❖ Earlier work, column-based

- [Saad]: $\text{ILU}(\tau, p)$, at most p largest nonzeros allowed in each row
- [Gupta/George]: p adaptive for each column $p(j) = \gamma \cdot \text{nnz}(A(:, j))$
May use interpolation to compute a threshold function, no sorting

❖ Our new scheme is "area-based"

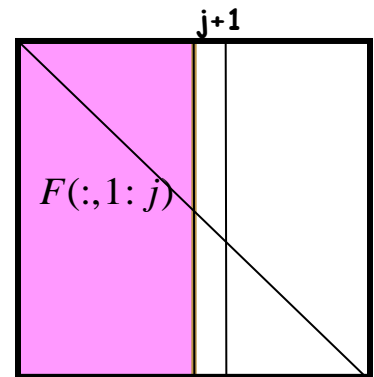
- Look at fill ratio from column 1 up to j :

$$fr(j) = \text{nnz}(F(:, 1:j)) / \text{nnz}(A(:, 1:j))$$

- Define adaptive upper bound function $f(j) \in [1, \gamma]$

If $fr(j)$ exceeds $f(j)$, retain only p largest, such that $fr(j) \leq f(j)$

- More flexible, allow some columns to fill more, but limit overall



- ❖ Use restarted GMRES with our ILU as a right preconditioner

$$\text{Solve } PA(\tilde{L}\tilde{U})^{-1}y = Pb$$

- ❖ Size of Krylov subspace set to 50

- ❖ Stopping criteria: $\|b - Ax_k\|_2 \leq 10^{-8}\|b\|_2$ and ≤ 1000 iterations

S-ILU for extended MHD (plasma fusion engery)



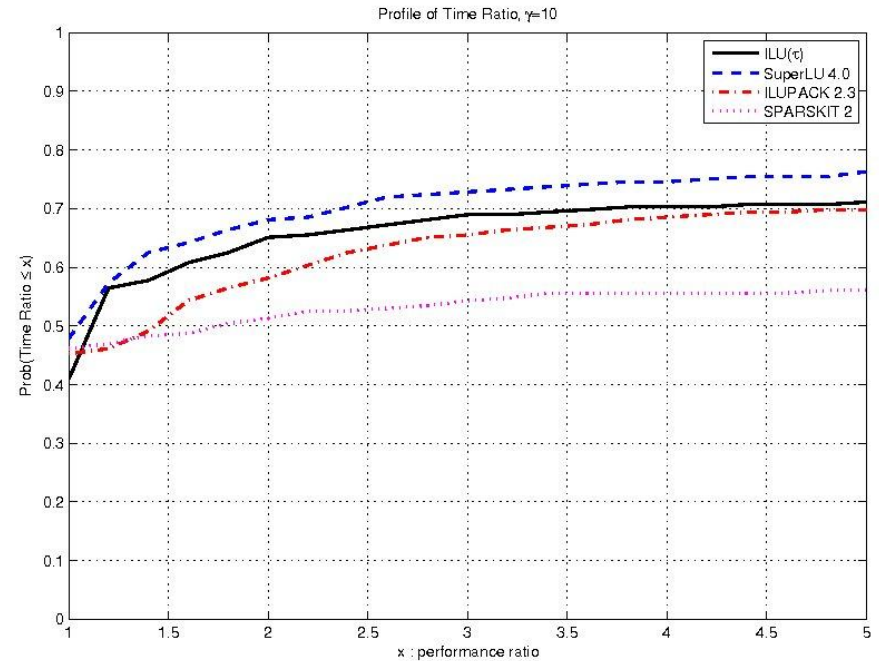
- ❖ Opteron 2.2 GHz (jacquard at NERSC), one processor
- ❖ ILU parameters: $\text{drop_tol} = 1e-4$, $\text{gamma} = 10$
- ❖ Up to 9x smaller fill ratio, and 10x faster

Problems	order	Nonzeros (millions)	ILU time ratio	fill- ratio	GMRES time iters		SuperLU time ratio	
matrix31	17,298	2.7 m	8.2	2.7	0.6	9	33.3	13.1
matrix41	30,258	4.7 m	18.6	2.9	1.4	11	111.1	17.5
matrix61	66,978	10.6 m	54.3	3.0	7.3	20	612.5	26.3
matrix121	263,538	42.5 m	145.2	1.7	47.8	45	fail	-
matrix181	589,698	95.2 m	415.0	1.7	716.0	289	fail	-

Compare with other ILU codes

- ❖ SPARSKIT 2 : scalar version of ILUTP [Saad]
- ❖ ILUPACK 2.3 : inverse-based multilevel method [Bolhoefer et al.]
- ❖ 232 test matrices : dimension 5K-1M
- ❖ **Performance profile of runtime** - fraction of the problems a solver could solve within a multiple of X of the best solution time among all the solvers

- ❖ S-ILU succeeded with 141
- ILUPACK succeeded with 130
- Both succeeded with 99



❖ Schur complement method

- a.k.a. iterative substructuring method
- a.k.a. non-overlapping domain decomposition

❖ Partition into many subdomains

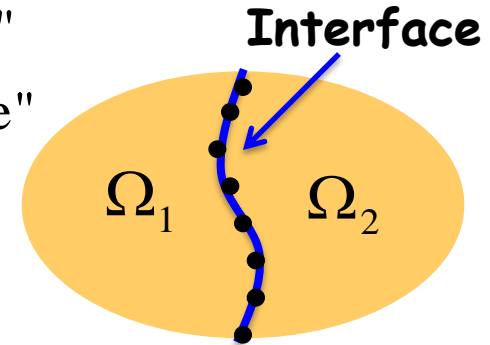
- Direct method for each subdomain, perform partial elimination independently, in parallel
- Preconditioned iterative method for the Schur complement system, which is often better conditioned, smaller but denser

❖ Case with two subdomains

Substructure contribution:

$$A^{(i)} = \begin{pmatrix} A_{ii}^{(i)} & A_{iI}^{(i)} \\ A_{Ii}^{(i)} & A_{II}^{(i)} \end{pmatrix} \quad \begin{array}{l} i = \text{"interior"} \\ I = \text{"Interface"} \end{array}$$

1. Assembled block matrix $A = \begin{pmatrix} A_{ii}^{(1)} & & A_{iI}^{(1)} \\ & A_{ii}^{(2)} & A_{iI}^{(2)} \\ A_{Ii}^{(1)} & A_{Ii}^{(2)} & A_{II}^{(1)} + A_{II}^{(2)} \end{pmatrix}$



2. Perform direct elimination of $A^{(1)}$ and $A^{(2)}$ independently,

$$\text{Local Schur complements: } S^{(i)} = A_{II}^{(i)} - A_{Ii}^{(i)} A_{ii}^{(i)-1} A_{iI}^{(i)}$$

$$\text{Assembled Schur complement } S = S^{(1)} + S^{(2)}$$

❖ Nested dissection, graph partitioning

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right) = \left(\begin{array}{cccc|c} A_{11}^{(1)} & & & & A_{12}^{(1)} \\ & A_{11}^{(2)} & & & A_{12}^{(2)} \\ & & \ddots & & \\ & & & A_{11}^{(k)} & A_{12}^{(k)} \\ \hline A_{12}^{(1)} & A_{12}^{(2)} & & A_{12}^{(k)} & A_{22} \end{array} \right)$$

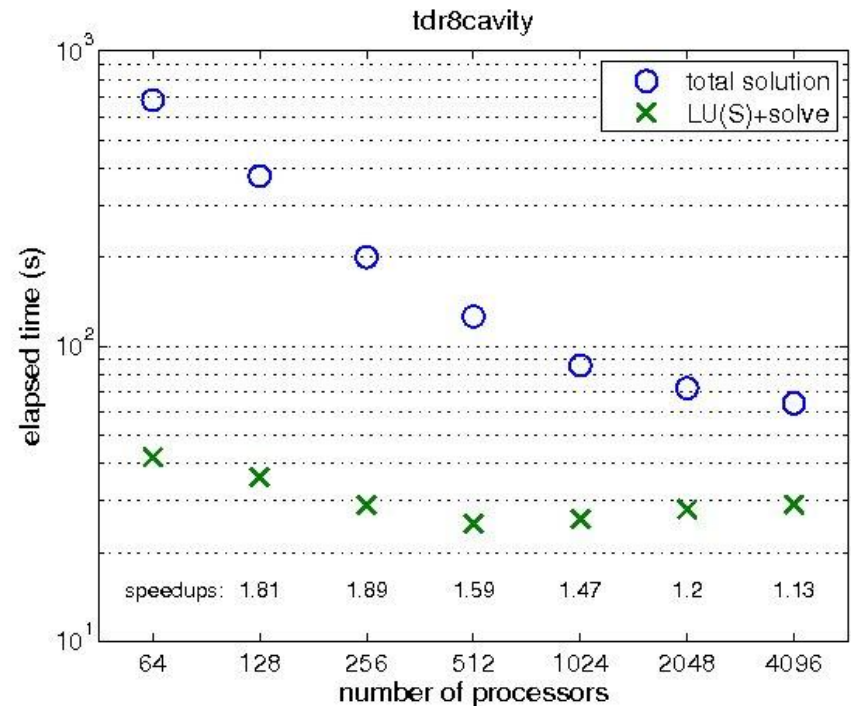
❖ Memory requirement: fill is restricted within

- “small” diagonal blocks of A_{11} , and
- ILU(S), sparsity can be enforced

❖ Two levels of parallelism: can use lots of processors

- multiple processors for each subdomain direct solution
 - only need modest level of parallelism from direct solver
- multiple processors for interface iterative solution

- ♦ Omega3P to design ILC accelerator cavity (Rich Lee, SLAC)
- ♦ Dimension: 17.8 M, real symmetric, highly indefinite
- ♦ PT-SCOTCH to extract 64 subdomains of size $\sim 277K$. The Schur complement size is $\sim 57K$
- ♦ SuperLU_DIST to factorize each subdomain
- ♦ BiCGStab of PETSc to solve the Schur system, with LU(S1) preconditioner
 - Converged in ~ 10 iterations, with relative residual $< 1e-12$



- ❖ Sparse LU, ILU are important kernels for science and engineering applications, used in practice on a regular basis
- ❖ Good implementation on high-performance machines requires a large set of tools from CS and NLA
- ❖ Performance more sensitive to latency than dense case

- ❖ Much room for optimizing performance
 - Automatic tuning of blocking parameters
 - Use of modern programming language to hide latency (e.g., UPC)
- ❖ Scalability of sparse triangular solve
 - Switch-to-dense, partitioned inverse
- ❖ Parallel ILU
- ❖ Optimal complexity sparse factorization
 - In the spirit of fast multipole method, but for matrix inversion
 - J. Xia's dissertation (May 2006)
- ❖ Latency-avoiding sparse factorizations