Sparse Gaussian Elimination on High Performance Machines

Xiaoye Sherry Li

CS 267 Lecture, April 25 1996

Work with

Jim Demmel (UC Berkeley)

Stan Eisenstat (Yale Univ)

John Gilbert (Xerox PARC)

Joseph Liu (York Univ)

Outline

- Motivation and background.
- New sequential algorithm and software.
- New parallel algorithm and preliminary results.
- Summary.

Motivation

- **Problem:** Solve $Ax = b$.
  - Core computation in many branches of scientific applications.

- Many large problems have sparse structure, and problems are getting larger.

- **Algorithm:** Gaussian elimination (LU factorization) $PA = LU$, followed by lower/upper triangular solves.

- Existing codes
  - Okay for small problems, slow for large ones.
  - Do not work well on new architectures:
    * Hierarchical memory system.
    * Superscalar processor.
    * Multiprocessors.

Test Matrices

Matrix	Name	N	Nnz	Nnz/N	$\lvert L+U\rvert/\lvert A\rvert$	#flops (10^6)
1	MEMPLUS	17758	99147	5.6	1.4	1.8
2	GEMAT11	4929	33185	6.7	2.8	1.6
3	RDIST1	4134	9408	2.3	36.0	12.9
4	MCFE	765	24382	31.8	4.8	11.7
5	SHERMAN5	3312	20793	6.3	13.6	30.4
6	LNSP3937	3937	25407	6.5	21.0	59.6
7	LNS3937	3937	25407	6.5	22.0	66.4
8	SHERMAN3	5005	20033	4.0	21.6	61.7
9	JPWH991	991	6027	6.1	26.7	23.5
10	ORANI678	2529	90158	35.6	6.9	103.9
11	ORSREG1	2205	14133	6.4	32.1	76.6
12	SAYLR4	3564	22316	6.3	34.7	144.7
13	SHYY161	76480	329762	4.3	23.2	1578.5
14	VENKAT01	62424	1717792	27.5	7.4	2730.9
15	GOODWIN	7320	324772	44.4	9.6	665.6
16	INACCURA	16146	1015156	62.9	9.9	4126.1
17	DENSE1000	1000	1000000	1000	1.0	666.7
18	RAEFSKY3	21200	1488768	70.2	11.8	12128.7
19	WANG3	26064	177168	6.8	74.9	14559.4
20	RAEFSKY4	19779	1316789	66.6	20.3	31307.1
21	VAVASIS3	41029	1683902	41.0	29.5	89865.1

- N = dimension
- Nnz = number of nonzeros

One Approach to Achieving Good Performance

Advances in dense matrix algorithms:

- Formulate algorithms as **block** operations.
- Express algorithms in BLAS for wider portability.
- Parallelize.

BLAS as building blocks:

- Deliver high performance while hiding machine details.
- $Reuse\_ratio = \dfrac{\#\text{ flops}}{\#\text{ memory access}}$ bounds performance.

BLAS-1	vector-vector $y := y + \alpha \cdot x$	$Reuse\_ratio = 2/3$
BLAS-2	matrix-vector $y := A \cdot x$	$Reuse\_ratio = 2$
BLAS-3	matrix-matrix $C := A \cdot B$	$Reuse\_ratio = n/2$

- Higher level BLAS achieves higher performance.

Successful examples in public domain software:

- LAPACK – workstations and shared memory multiprocessors.
- ScaLAPACK – distributed memory multiprocessors.

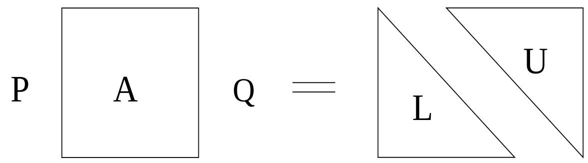
GOAL

- Extend these techniques to sparse matrix algorithms!

Gaussian Elimination

- **Problem:** Solve $Ax = b$.

- **Approach:** Gaussian elimination (LU factorization), followed by lower/upper triangular solves.



  P and Q are permutation matrices.

- **Sparse algorithms:**
  - Combinatorial algorithms to manipulate nonzero structure.
  - New nonzeros in $L + U$: "fills".

1. **Cholesky factorization** $A = LL^T$, where A is symmetric and positive definite, and $U = L^T$.
   - No numerical pivoting.
   - **Static** determination of nonzero structure of L.

2. **LU factorization with partial pivoting. (MY WORK)**
   - **Need numerical pivoting to get accurate solution.**
   - **Sparsity structure changes dynamically.**
   - **Need symbolic steps to find exact structure during the factorization.**

Choosing Permutations

- **Algorithm:** $PA\,Q = LU$, A is sparse and nonsymmetric, P, Q are permutations.

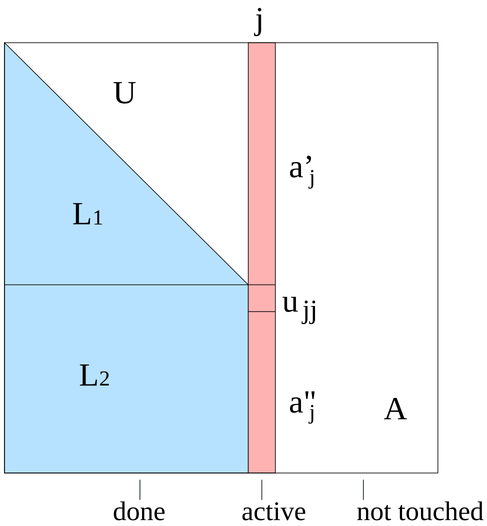
- **Column permutation** Q to control sparsity **prior to** the factorization.
  - Fills occur during elimination process ...
  - **Theorem (George & Ng 1987)**
    If $R^T R = A^T A$ and $PA = LU$, then for any P, the nonzero structure of $L + U$ is contained in the nonzero structure of $R^T + R$.
  - Making R sparse tends to make $L + U$ sparse.
  - Fill-reducing strategy:
    1. Find good symmetric preordering Q from $A^T A$.
       (Davis, Gilbert, Ng & Peyton)
    2. Apply Q to columns of A

$$Q^T (A^T A) Q = (AQ)^T (AQ)\ .$$

- **Row permutation** P to maintain numerical stability; this must be done **during** the factorization.
  - Partial pivoting, with other options

High Level Algorithm for Sparse LU

- Each step computes one column of U and L by using the previous columns of L.
- Set up the sparsity structure of the new column before computing it – **symbolic factorization**.



for $j = 1$ to n

symbolic factor: determine which columns will update column j;

solve $L_1 u_j = a'_j$ for u_j; {sparse triangular solve}

$l_j := a''_j - L_2 u_j$; {sparse matrix-vector multiply}

swap u_{jj} with largest-magnitude entry of l_j; {**pivoting**}

$l_j := l_j / u_{jj}$;

endfor

Previous Efforts

- **GP** (Gilbert/Peierls '88, used in Matlab)

  + symbolic cost $\propto$ # flops
  − only BLAS-1 speed, poor cache reuse
  − symbolic part still expensive

  $\Longrightarrow$ prune symbolic representation

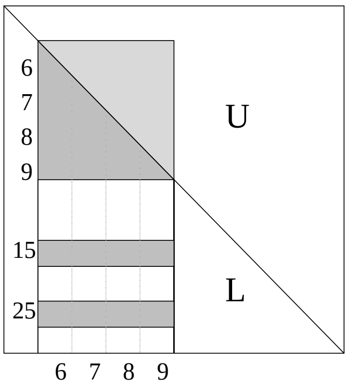
- **GP-mod** (Eisenstat/Liu '92)

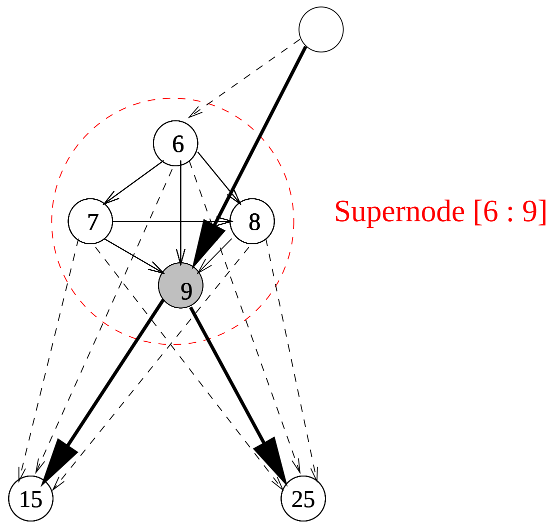
  + much cheaper symbolic factorization (4x)
  − still only BLAS-1
  − speedup still possible on symbolic part

  $\Longrightarrow$ introduce supernodes (s-nodes)

Nonsymmetric Supernodes

- Exploit dense submatrices in the L & U factors of $PA = LU$.





- Why are supernodes good?
  - Permit use of higher level BLAS.
  - Reduce inefficient indirect addressing (scatter/gather).
  - Reduce symbolic time by traversing supernodal graph.

Example Matrix with Supernodes

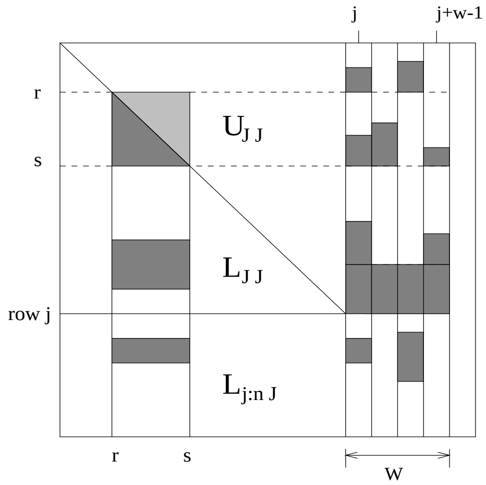
$$
\begin{pmatrix}
1 & \bullet & \bullet & & & \bullet & & & & \\
\bullet & 2 & & \bullet & & & & \bullet & & \\
& & 3 & & & & \bullet & \bullet & & \\
& & & 4 & \bullet & \bullet & & & \bullet & \\
& & & \bullet & & & \bullet & & & \\
\bullet & & & & & 6 & & & \bullet & \\
& & & & & & 7 & & \bullet & \bullet \\
\bullet & & \bullet & & & & & & & \\
& & & \bullet & & \bullet & & & 9 & \\
& & \bullet & & & & \bullet & & \bullet & 10
\end{pmatrix}
\qquad
\begin{pmatrix}
1 & \bullet & \bullet & & & \bullet & & & & \\
\bullet & 2 & \bullet & \bullet & & \bullet & & \bullet & & \\
& & 3 & & & & \bullet & \bullet & & \\
& & & 4 & \bullet & \bullet & & & \bullet & \\
& & & \bullet & 5 & \bullet & \bullet & & \bullet & \\
\bullet & \bullet & \bullet & \bullet & \bullet & 6 & \bullet & \bullet & \bullet & \\
& & & & & & 7 & & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 8 & \bullet & \bullet \\
& & & \bullet & \bullet & \bullet & \bullet & \bullet & 9 & \bullet \\
& & \bullet & & & & \bullet & \bullet & \bullet & 10
\end{pmatrix}
$$

Original matrix A $\qquad$ Factors $F = L + U - I$

$$
\begin{array}{c}
1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10
\end{array}
\begin{pmatrix}
\circ & \circ & u_3 & & & u_6 & & & & \\
\circ & \circ & u_3 & u_4 & & u_6 & & u_8 & & \\
& & \circ & & & & u_7 & u_8 & & \\
& & & \circ & \circ & \circ & & & u_9 & \\
& & & \circ & \circ & \circ & u_7 & & u_9 & \\
\circ & \circ & \circ & \circ & \circ & \circ & u_7 & u_8 & u_9 & \\
& & & & & & \circ & & \circ & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\
& & & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\
& & \circ & & & & \circ & \circ & \circ & \circ
\end{pmatrix}
$$

4 Supernodes: (1,2), (3), (4,5,6), (7,8,9,10)

Supernodal Factorization



- **Sup-Col** (Eisenstat/Gilbert/Liu '93)

  + BLAS-1 replaced by faster BLAS-2 : s-node $\times$ vector

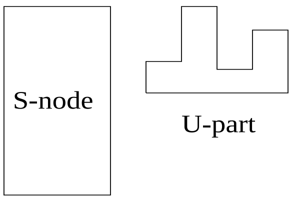
  + symbolic part yet cheaper ($\propto$ s-node size)

  − cache still not reused well

  $\Longrightarrow$ update next panel, not just column

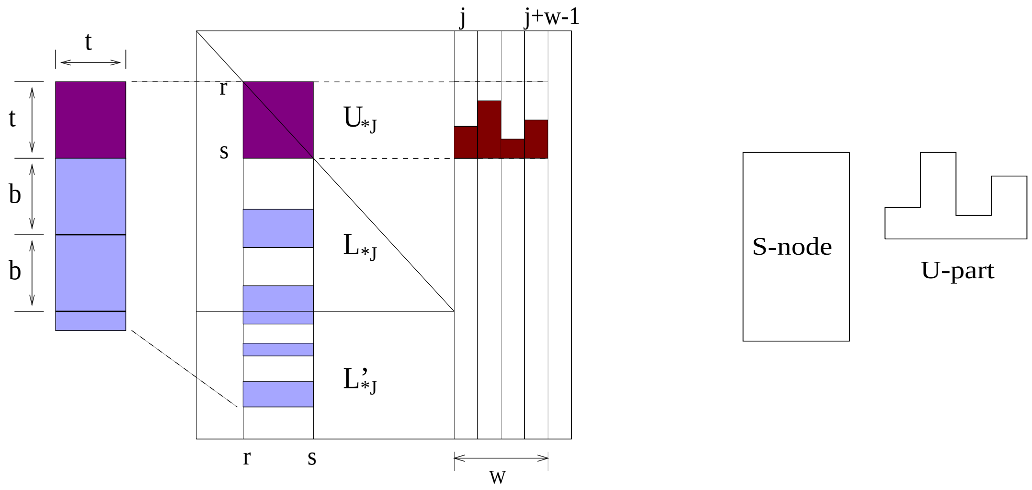
- **SuperLU** (Demmel/Gilbert/Li '95)

  + BLAS-2 replaced by "BLAS $2\frac{1}{2}$" – nearly BLAS-3 speed



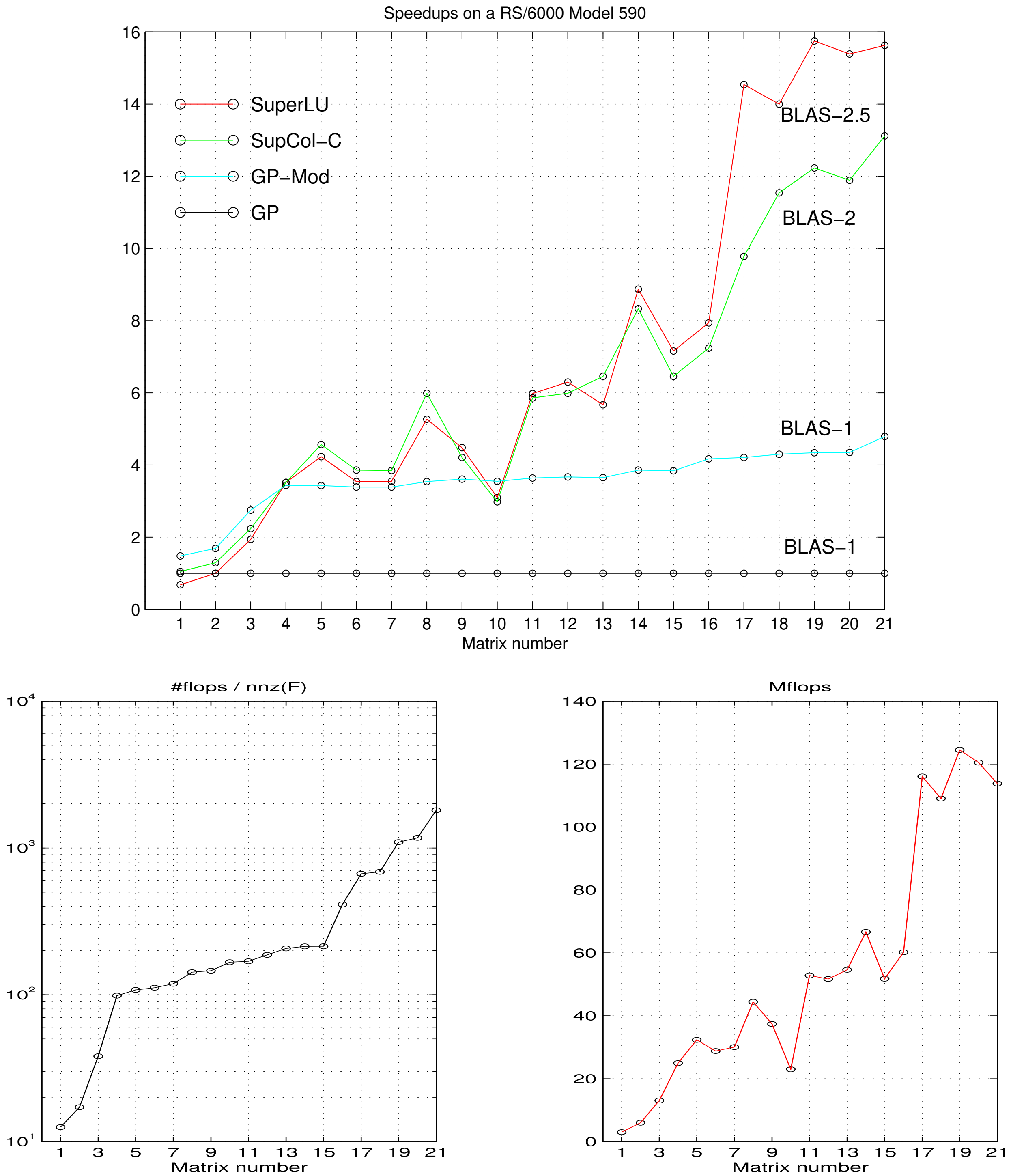
$\pm$ a limited set of tuning parameters for different cache sizes

Supernode-panel Factorization – SuperLU



- "BLAS $2\frac{1}{2}$" kernel – nearly BLAS-3 speed.
- Cache awareness boosts performance for larger problems.
  - 2D blocking in order to do well for large matrix, or for small cache $\Longrightarrow$ guaranteed in-cache BLAS-2.
  - Detailed methodology as to how to choose blocking parameters (t, b, w) based on cache size.
  - Ultimate code chooses 1D or 2D blocking based on S-node size.

SuperLU Uniprocessor Performance



- Sorted in increasing order of $Reuse\_ratio = \#Flops/nnz(F)$

Some Statistics

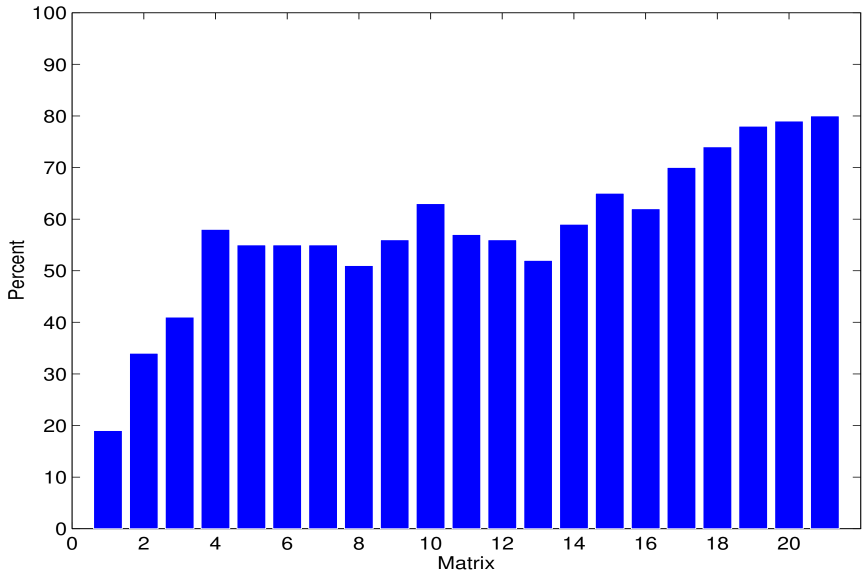


Figure 1: Percentage of the runtime spent in numeric factorization.

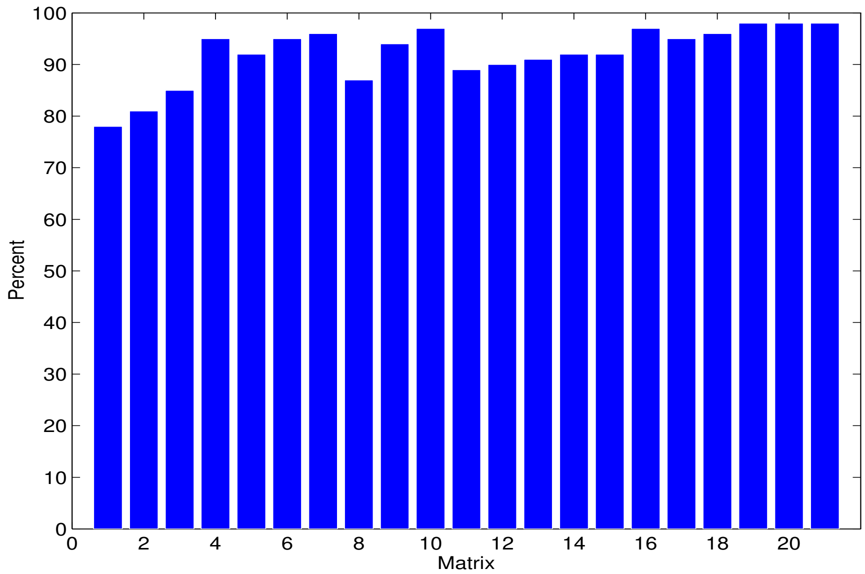


Figure 2: Percentage of the flops spent in DGEMVroutine.

SuperLU Public Domain Software

- Performance highlights
  - Up to 15x speedup over Matlab sparse LU algorithm
  - Observed Mflops rate

	RS/6000-590	**SGI Power Challenge**
Peak	266	360
large sparse	up to 124	up to 147
dense1000 (LAPACK)	116 (168)	173 (241)

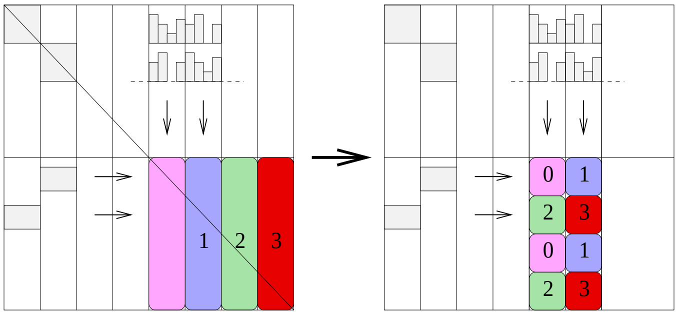
- LAPACK-style interface

- Functions
  - User-input to control pivoting
    - pre-assigned row and/or column permutations
    - partial pivoting with threshold
  - Transposed system
  - Equilibration
  - Condition number estimation
  - Iterative refinement to improve accuracy
  - Error bounds
    backward error, forward error

- Available at
  http://www.cs.berkeley.edu/~xiaoye/superlu.html
  ftp.cs.berkeley.edu : pub/src/lapack/SuperLU/superlu.tar.gz

Parallel SuperLU

- 1D vs. 2D matrix layout.



1D layout

+ simple data structure
– not scalable

2D layout

+ scalable
– complicated data structure
– more synchronizations

- Shared memory vs. distributed memory

Shared memory

+ easy programming model
+ easy to support on different platforms
+ easy to perform dynamic scheduling and load balancing
– not scalable (yet)

Distributed memory

+ scalable
– explicit message passing

How Parallelism Arises

- Column elimination tree (etree of A^TA) – each column of the matrix has one vertex in the tree.

- Exhibits column dependencies during the elimination:

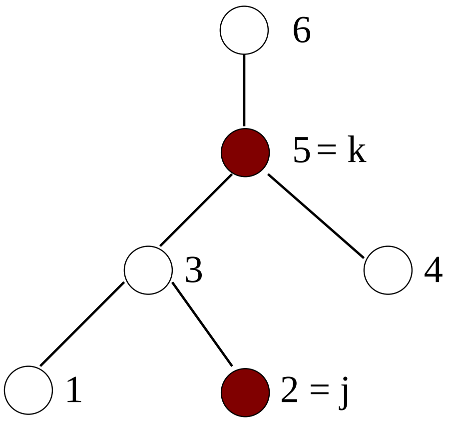
  1. *If column j updates column k, then j is a descendant of k;*

  2. *Conversely, if j is a descendant of k, column j* **may** *or* **may not** *update column k (depending on pivoting).*

- May be pessimistic about dependencies, since it captures all possible row permutations!

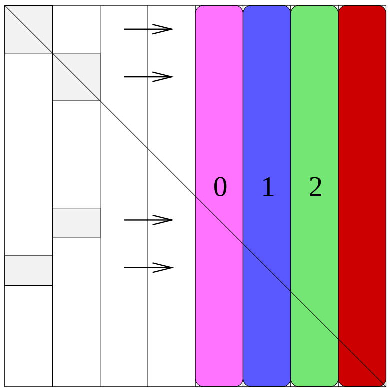
- Ancestor nodes can start some types of processing as soon as descendants finish.

$$A = \begin{pmatrix} 1 & & \bullet & & \bullet & \bullet \\ & 2 & \bullet & & \bullet & \\ & \bullet & 3 & & & \\ \bullet & & & 4 & \bullet & \\ & & & & 5 & \bullet \\ \bullet & & & \bullet & & 6 \end{pmatrix}$$



Shared Memory Working Code

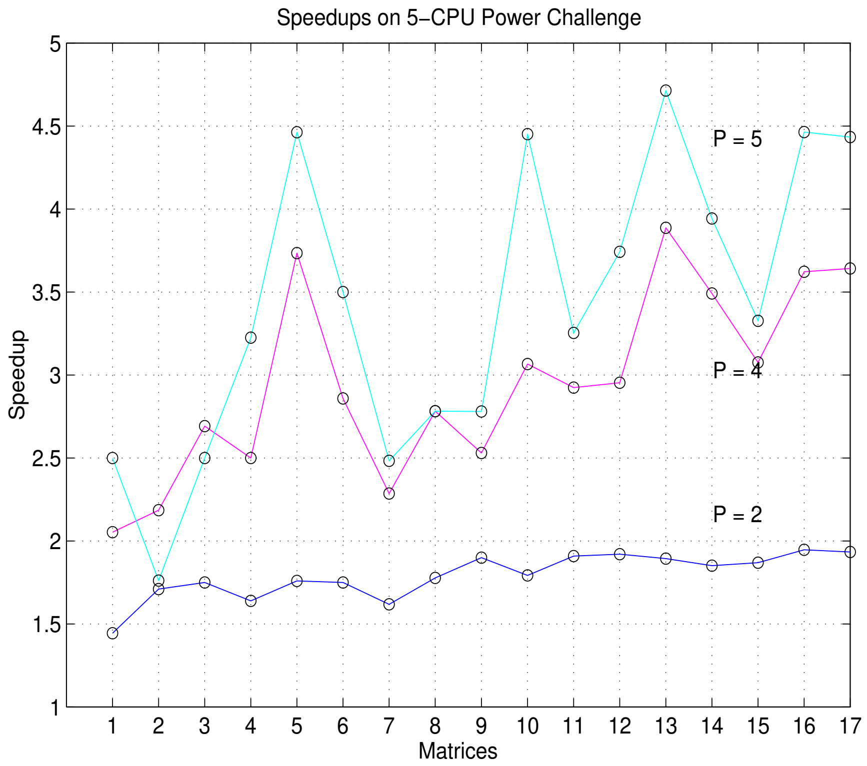
- 1D data layout.



- Central scheduler dynamically assigns panel tasks to processors.
- Retain "BLAS $2\frac{1}{2}$" kernel on single processor.
- Almost identical code is running on both SGI Power Challenge and Sun SPARCcenter 2000.

Performance on SGI Power Challenge

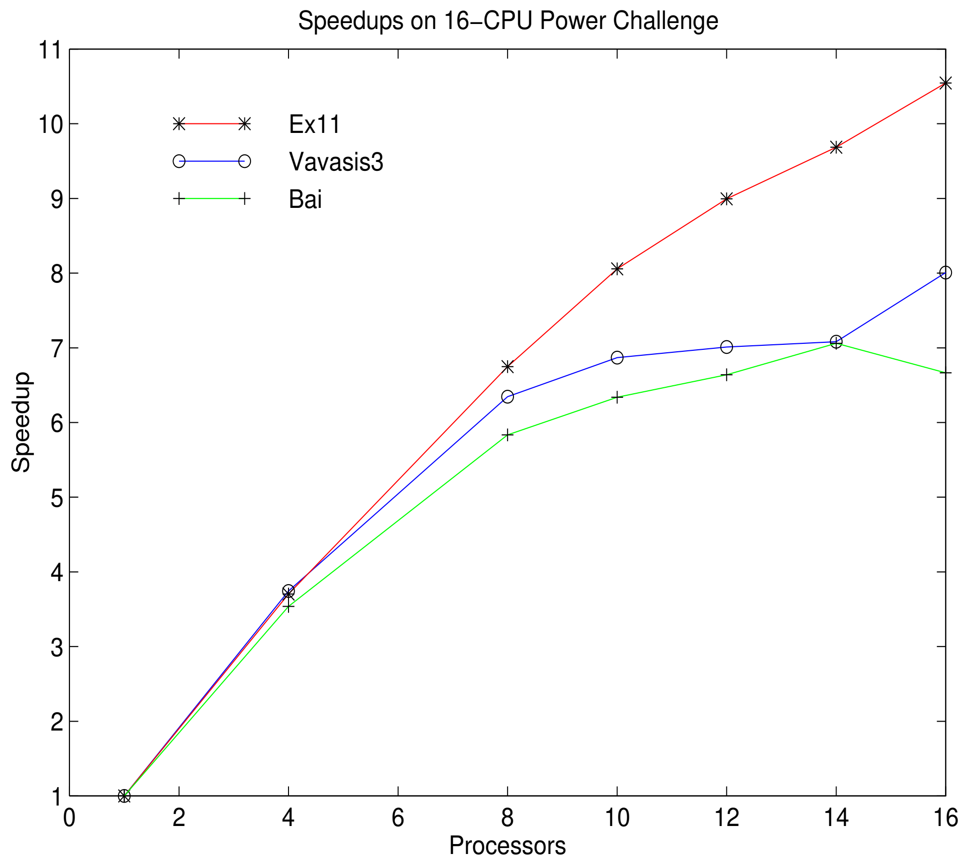
- With help of Ed Rothberg from SGI.



- Observed ~ 0.5 Gigaflops factorization rate.

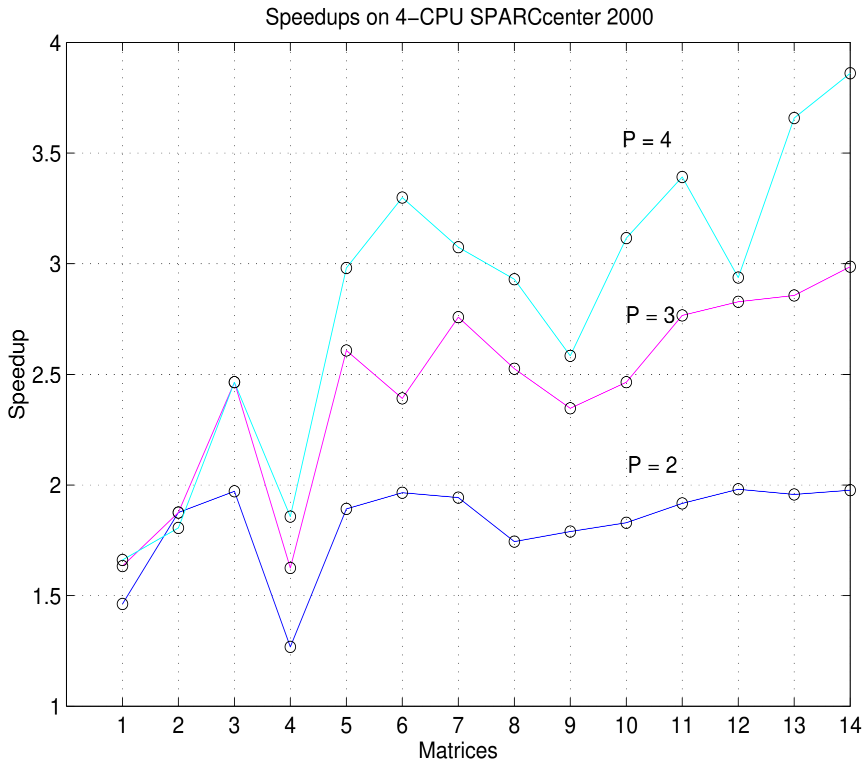
Performance on SGI Power Challenge

Matrix	N	Nnz	Nnz/N	#Flops
Ex11	16614	1096948	66	2.7×10^{10}
Vavasis3	41029	1683902	41	9.0×10^{10}
Bai	23560	460598	20	6.4×10^{9}



- Observed 1.2 Gigaflops factorization rate.

Performance on Sun SPARCcenter 2000



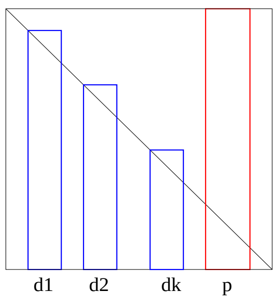
Performance Bound on the 1D Algorithm
(*a posteriori* estimate)

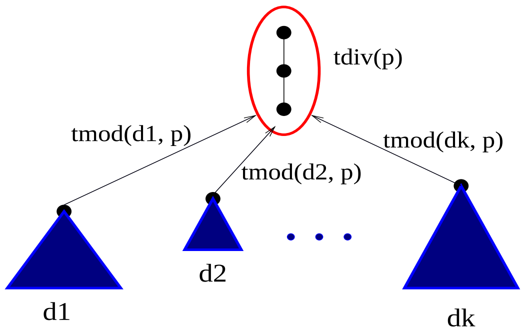
Goal

- Derive the minimum parallel runtime by studying the floating-point operations that must be done *serially*.

Model

- Abstract the factorization as a task graph – DAG.
  - nodes $\longleftrightarrow$ computations
  - edges $\longleftrightarrow$ precedence constraint
- Simplifications: ignore number of processors, scheduling details, synchronization overhead.





Notation

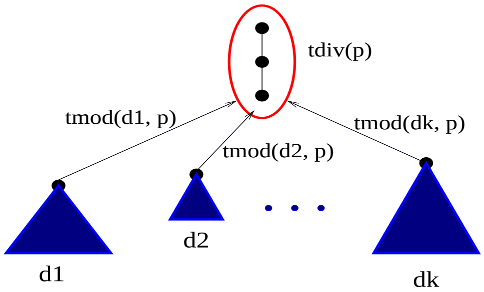
- $tmod(p, d)$:= time for modification from a descendant supernode d to panel p.
- $tdiv(p)$:= time for panel inner factorization.
- $EST(p)$:= earliest possible starting time of panel p.
- $EFT(p)$:= earliest possible finishing time of panel p.

All times are expressed in units of floating-point operations.

Performance Bound on the 1D Algorithm

Assumptions in the current 1D algorithm:

- T_{mod} and T_{div} are indivisible tasks.
- T_{div} cannot start until all descendant T_{mod} are finished.



Find $EFT(p)$**:**

1. *Sort the EFTs of the descendants in increasing order*

$$EFT(1) \leq EFT(2) \leq \ldots \leq EFT(k) \ .$$

2. *Run the following recurrence:*

   $t = 0$;
   for $i \ = \ 1, \ k$
   $\quad t = \max\{t, \ EFT(i)\} + tmod(i)$;
   end for;

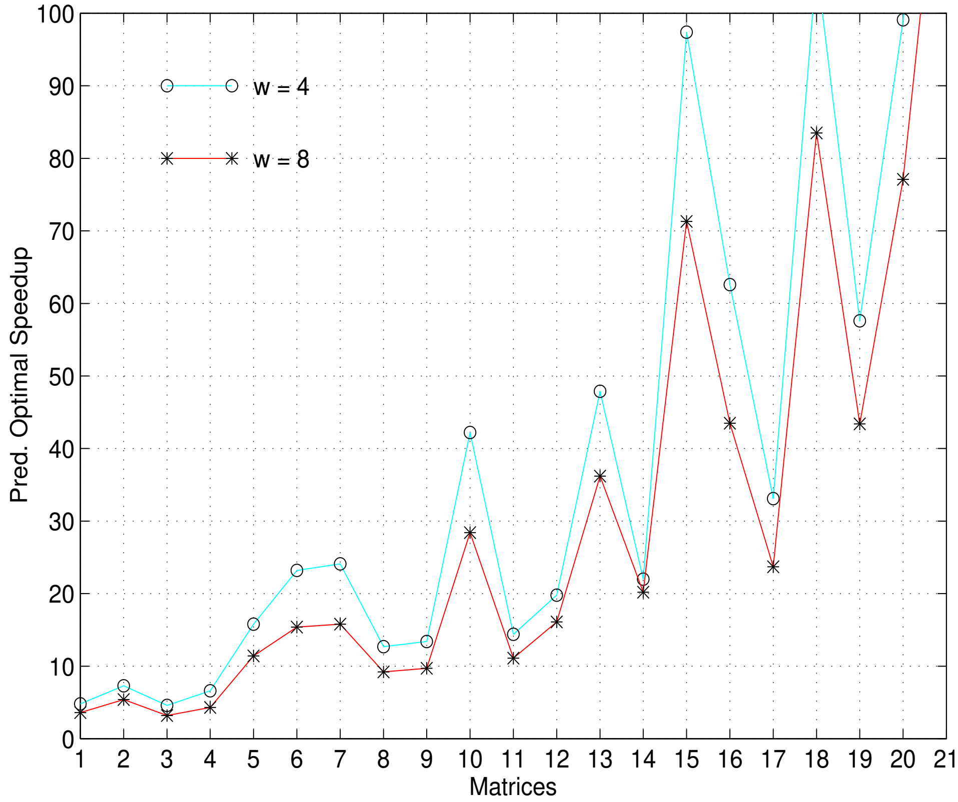
3. *Set* $EST(p) = t$, *and* $EFT(p) = EST(p) + tdiv(p)$.

Claim: The computed $EFT(root)$ gives the shortest parallel completion time.

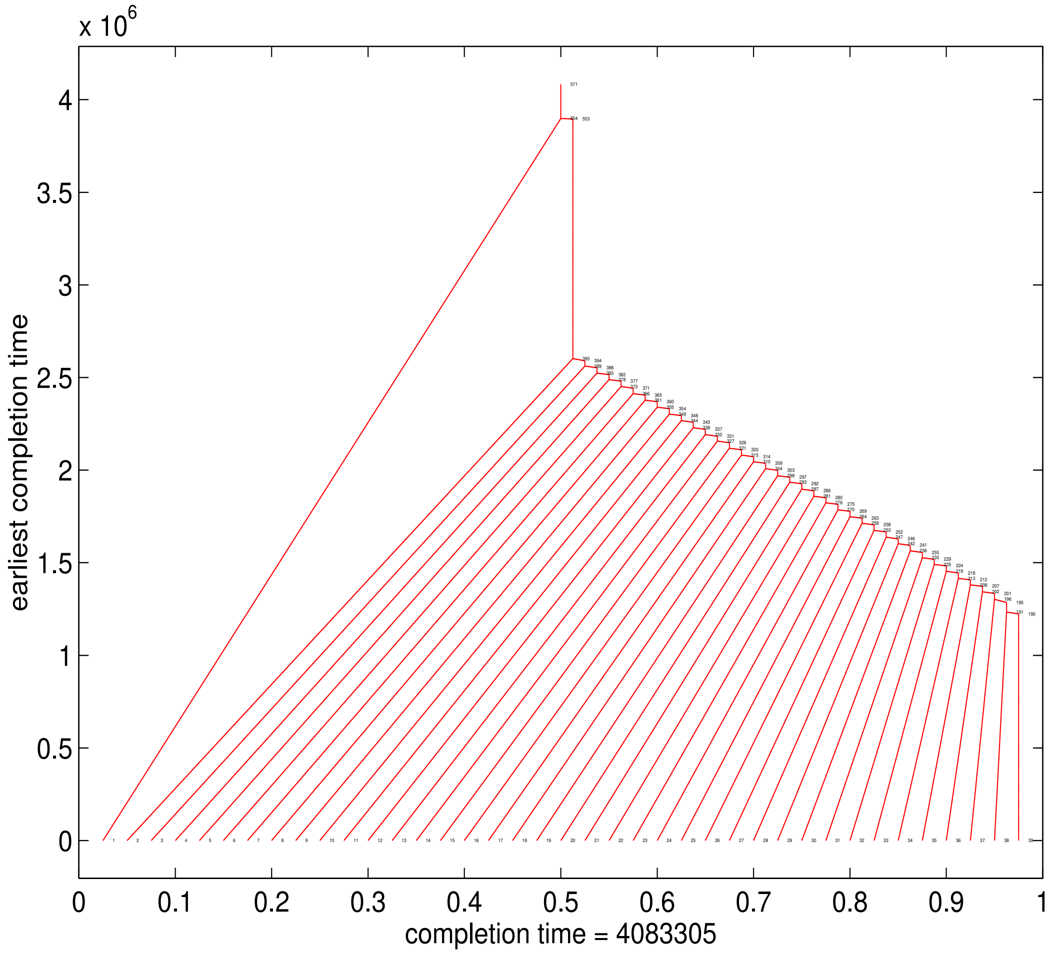
Optimal speedup: $\frac{\#\ \text{Total flops}}{EFT(root)}$.

Optimal Speedup of the 1D Algorithm

- w = number of columns in a panel being worked on.



Poor Example

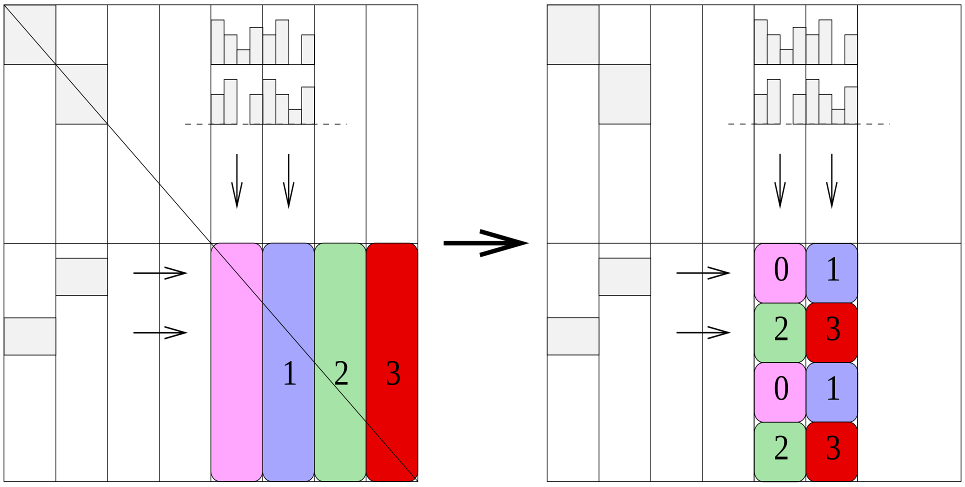


Where to Improve?

- Better column ordering.
- 2D parallel algorithm.

Towards a More Scalable Algorithm

- Parallelize along both row and column dimensions.



- Target at larger scale parallel machines.

- Demonstrated scalability in dense LU and sparse Cholesky.

- Research issues:
  - more synchronizations
  - 2D matrix layout on distributed memory machines
  - which communication protocol is better?

2-sided	send/receive	MPI
1-sided	put/get	Active Message, Split-C

- We are now simulating the algorithm and evaluating various strategies . . .

Summary

Contributions

- Highly efficient algorithms on workstations and modest size SMPs.
  - Public domain software – SuperLU
  - Shared memory 1D code achieves good speedup;
    Some “alpha” users.

Current work

- 2D parallel algorithm under development.