4th Gene Golub SIAM Summer School, 7/22 – 8/7, 2013, Shanghai

Factorization-based Sparse Solvers and Preconditioners

Xiaoye Sherry Li
Lawrence Berkeley National Laboratory, USA
xsli@lbl.gov

crd-legacy.lbl.gov/~xiaoye/G2S3/

Acknowledgement

- Jim Demmel, UC Berkeley, course on “Applications of Parallel Computers”: http://www.cs.berkeley.edu/~demmel/cs267_Spr13/
- John Gilbert, UC Santa Barbara, course on “Sparse Matrix Algorithms”: http://cs.ucsb.edu/~gilbert/cs219/cs219Spr2013/
- Patrick Amestoy, Alfredo Buttari, ENSEEIHT, course on “Sparse Linear Algebra”
- Jean-Yves L’Excellent, Bora Uçar, ENS-Lyon, course on “High-Performance Matrix Computations”
- Artem Napov, Univ. of Brussels
- Francois-Henry Rouet, LBNL
- Meiyue Shao, EPFL
- Sam Williams, LBNL
- Jianlin Xia, Shen Wang, Purdue Univ.

Course outline

1. Fundamentals of high performance computing
2. Basics of sparse matrix computation: data structure, graphs, matrix-vector multiplication
3. Combinatorial algorithms in sparse factorization: ordering, pivoting, symbolic factorization
4. Numerical factorization & triangular solution: data-flow organization
5. Parallel factorization & triangular solution
6. Preconditioning: incomplete factorization
7. Preconditioning: low-rank data-sparse factorization
8. Hybrid methods: domain decomposition, substructuring method

Course materials online: crd-legacy.lbl.gov/~xiaoye/G2S3/

4th Gene Golub SIAM Summer School, 7/22 – 8/7, 2013, Shanghai

Lecture 1

Fundamentals: Parallel computing, Sparse matrices

Xiaoye Sherry Li
Lawrence Berkeley National Laboratory, USA
xsli@lbl.gov

Lecture outline

- Parallel machines and programming models
- Principles of parallel computing performance
- Design of parallel algorithms
  - Matrix computations: dense & sparse
  - Partial Differential Equations (PDEs)
  - Mesh methods
  - Particle methods
  - Quantum Monte-Carlo methods
  - Load balancing, synchronization techniques

Parallel machines & programming model
(hardware & software)

Idealized Uniprocessor Model

- Processor names bytes, words, etc. in its address space
  - These represent integers, floats, pointers, arrays, etc.
- Operations include
  - Read and write into very fast memory called registers
  - Arithmetic and other logical operations on registers
- Order specified by program
  - Read returns the most recently written data
  - Compiler and architecture translate high level expressions into "obvious" lower level instructions (assembly)

$$A = B + C \Rightarrow$$

Read address(B) to R1
Read address(C) to R2
R3 = R1 + R2
Write R3 to Address(A)

  - Hardware executes instructions in order specified by compiler
- *Idealized Cost*
  - Each operation has roughly the same cost

    (read, write, add, multiply, etc.)

Uniprocessors in the Real World

- Real processors have
  - registers and caches
    - small amounts of fast memory
    - store values of recently used or nearby data
    - different memory ops can have very different costs
  - parallelism
    - multiple "functional units" that can run in parallel
    - different orders, instruction mixes have different costs
  - pipelining
    - a form of parallelism, like an assembly line in a factory
- Why need to know this?
  - In theory, compilers and hardware "understand" all this and can optimize your program; in practice they don't.
  - They won't know about a different algorithm that might be a much better "match" to the processor

Parallelism within single processor – pipelining

- Like assembly line in manufacturing
- Instruction pipeline allows overlapping execution of multiple instructions with the same circuitry

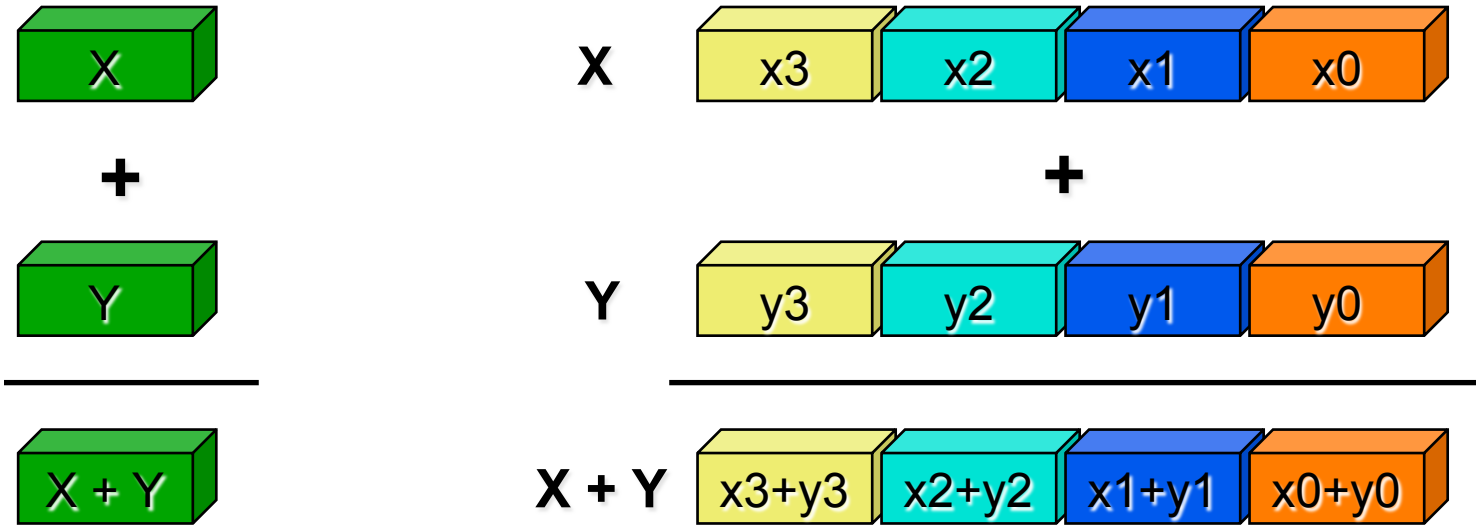
Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

IF = Instruction Fetch
ID = Instruction Decode
EX = Execute
MEM = Memory access
WB = Register write back

  - Sequential execution: 5 (cycles) * 5 (inst.) = 25 cycles
  - Pipelined execution: 5 (cycles to fill the pipe, latency) + 5 (cycles, 1 cycle/inst. throughput) = 10 cycles
- Arithmetic unit pipeline: A FP multiply may have latency 10 cycles, but throughput of 1/cycle
- Pipeline helps throughput/bandwidth, but not latency

Parallelism within single processor – SIMD

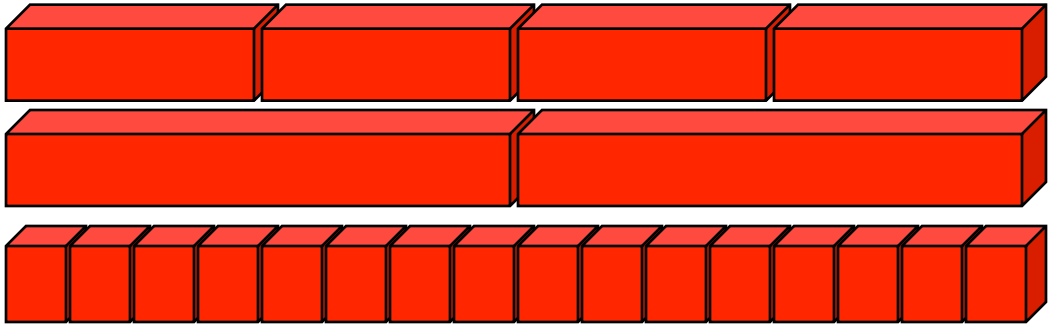
- SIMD: Single Instruction, Multiple Data
  - Scalar processing
    - traditional mode
    - one operation produces one result
  - SIMD processing
    - with SSE / SSE2
    - SSE = streaming SIMD extensions
    - one operation produces multiple results



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,



4x floats

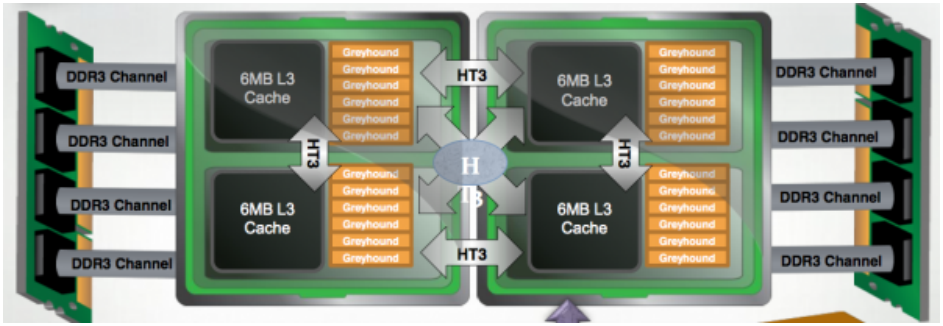
2x doubles

16x bytes

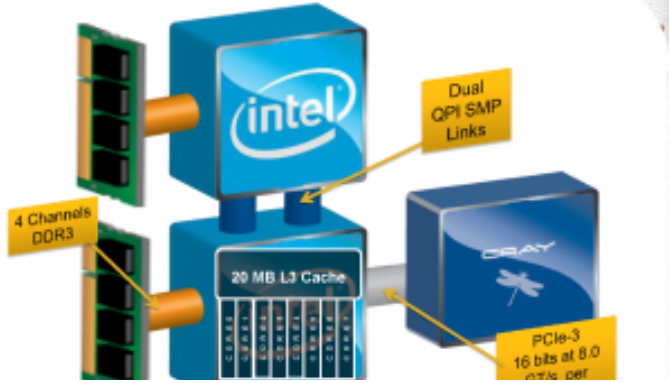
- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
  - Need to be contiguous in memory and aligned
  - Some instructions to move data around from one part of register to another
- Similar on GPUs, vector processors (but many more simultaneous operations)

Variety of node architectures

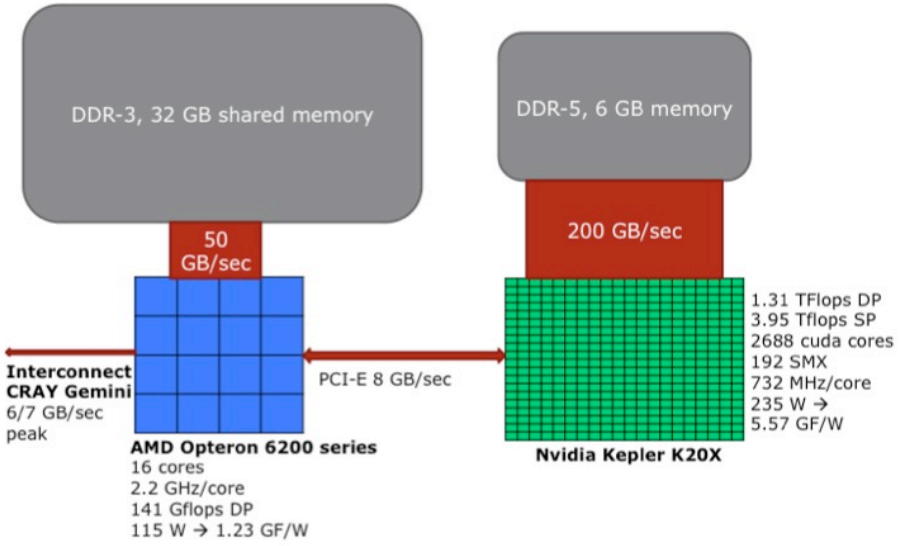
Cray XE6: dual-socket x 2-die x 6-core, 24 cores



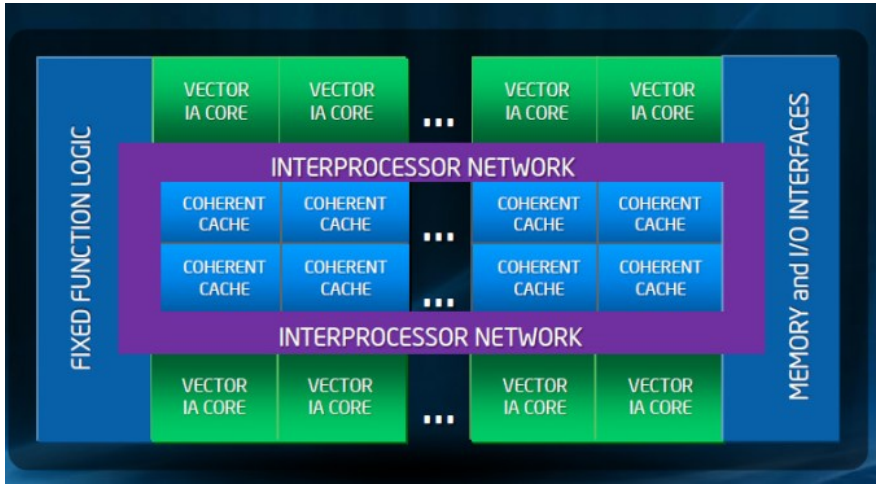
Cray XC30: dual-socket x 8-core, 16 cores



Cray XK7: 16-core AMD + K20X GPU



Intel MIC: 16-core host + 60^{+} cores co-processor



TOP500 (www.top500.org)

- Listing of 500 fastest computers
- Metric: LINPACK benchmark
  - "How fast is your computer?" =
    "How fast can you solve dense linear system Ax=b?"
- Current records (June, 2013)

Rank	Machine	Cores	Linpack (Petaflop/s)	Peak (Petaflop/s)
1	Tianhe-2 – Intel MIC (China National Univ. of Defense Technology)	3,120,000	33.8 (61%)	54.9
2	Titan – Cray XK7 (US Oak Ridge National Lab)	560, 640	17.6 (65%)	27.1
3	Sequoia – BlueGene/Q (US Lawrence Livermore National Lab)	1,572,864	17.1 (85%)	20.1

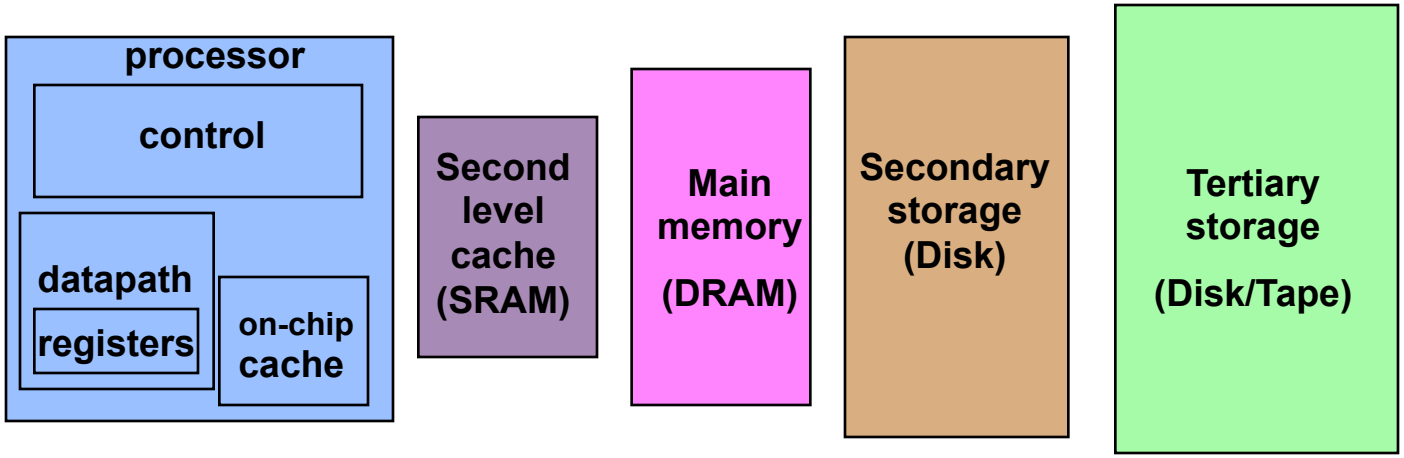
Units of measure in HPC

- High Performance Computing (HPC) units are:
  - Flop: floating point operation
  - Flops/s: floating point operations per second
  - Bytes: size of data (a double precision floating point number is 8)
- Typical sizes are millions, billions, trillions…

Mega	**Mflop/s = 10^6 flop/sec**	**Mbyte = 2^{20} = 1048576 ~ 10^6 bytes**
Giga	**Gflop/s = 10^9 flop/sec**	**Gbyte = 2^{30} ~ 10^9 bytes**
Tera	**Tflop/s = 10^{12} flop/sec**	**Tbyte = 2^{40} ~ 10^{12} bytes**
Peta	**Pflop/s = 10^{15} flop/sec**	**Pbyte = 2^{50} ~ 10^{15} bytes**
Exa	**Eflop/s = 10^{18} flop/sec**	**Ebyte = 2^{60} ~ 10^{18} bytes**
Zetta	**Zflop/s = 10^{21} flop/sec**	**Zbyte = 2^{70} ~ 10^{21} bytes**
Yotta	**Yflop/s = 10^{24} flop/sec**	**Ybyte = 2^{80} ~ 10^{24} bytes**

Memory Hierarchy … Flops is not everything

- Most programs have a high degree of locality in their accesses
  - **spatial locality:** accessing things nearby previous accesses
  - **temporal locality:** reusing an item that was previously accessed
- Memory hierarchy tries to exploit locality to improve average



Speed	1ns	10ns	100ns	10ms	10sec
Size	KB	MB	GB	TB	PB

Hopper Node Topology

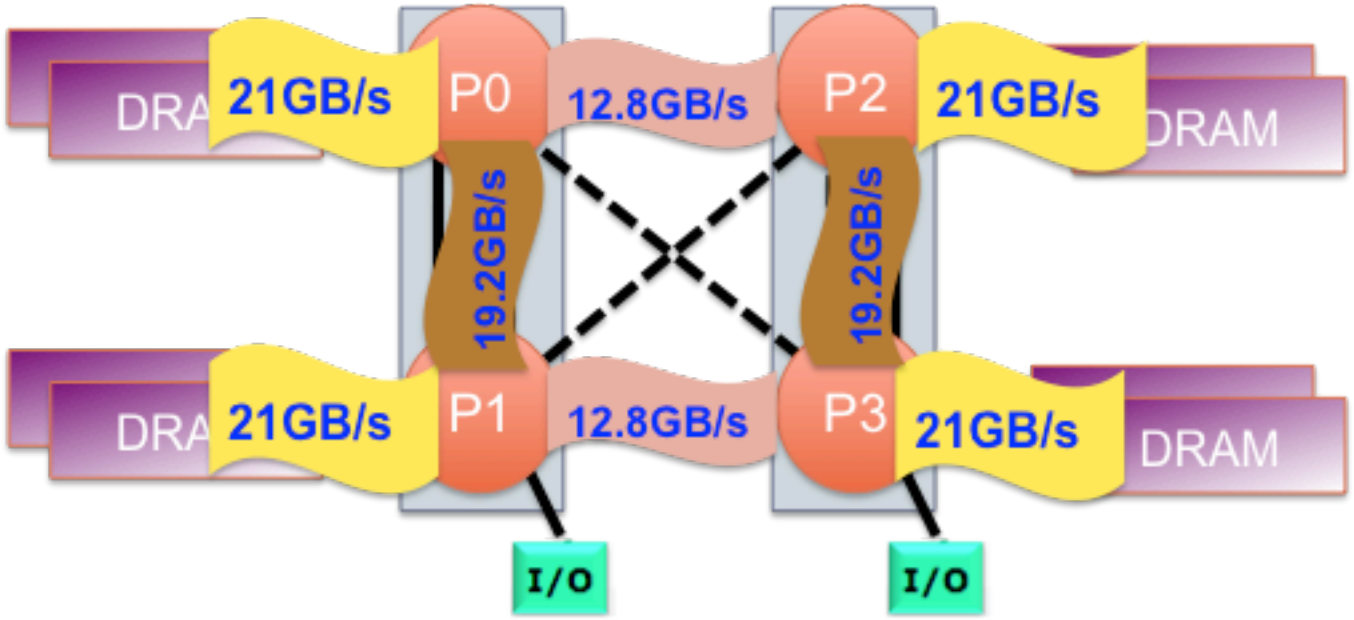
Understanding NUMA Effects **[J. Shalf]**

- **Heterogeneous Memory access between dies**
- **"First touch" assignment of pages to memory.**

2xDDR1333 channel
21.328 GB/s

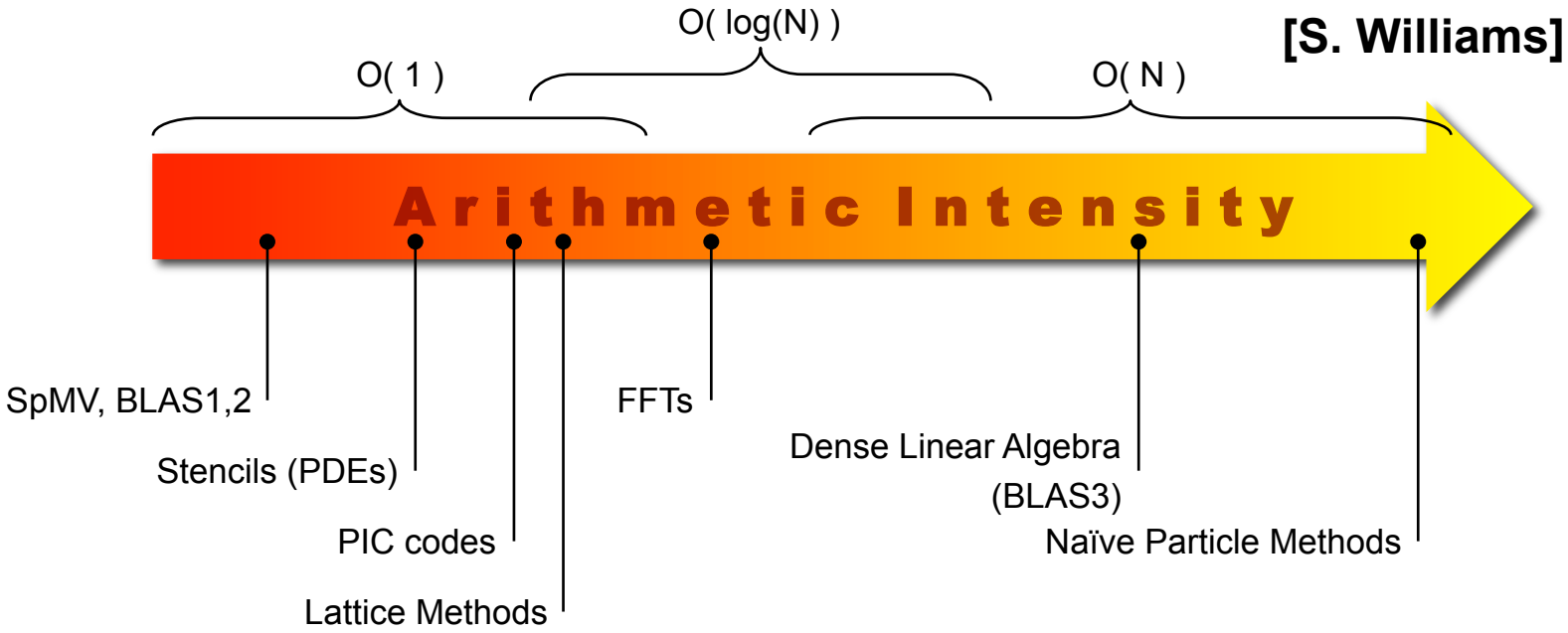
3.2GHz x8 lane HT
6.4 GB/s bidirectional

3.2GHz x16 lane HT
12.8 GB/s bidirectional



- **Locality is key** *(just as per Exascale Report)*
- **Only *indirect* locality control with OpenMP**

Arithmetic Intensity



- Arithmetic Intensity (AI) ~ Total Flops / Total DRAM Bytes
  - E.g.: dense matrix-matrix multiplication: n^3 flops / n^2 memory

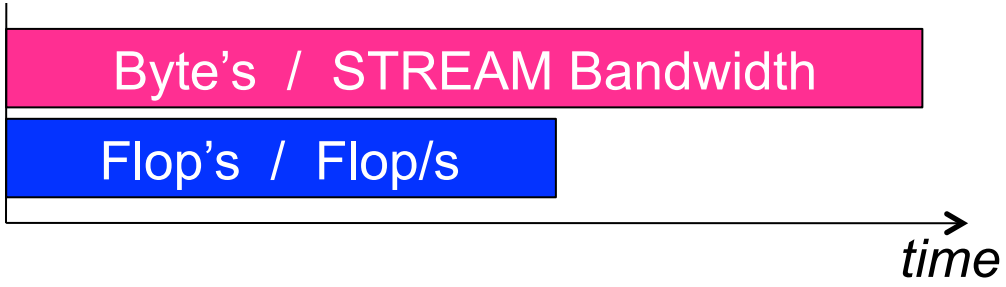
- Higher AI → better locality → amenable to many optimizations → achieve higher % machine peak

Roofline model (S. Williams)

basic concept

- Synthesize communication, computation, and locality into a single visually-intuitive performance figure using bound and bottleneck analysis.
  - Assume FP kernel maintained in DRAM, and perfectly overlap computation and communication w/ DRAM
  - Arithmetic Intensity (AI) is computed based on DRAM traffic after being filtered by cache
- Question : is the code computation-bound or memory-bound?

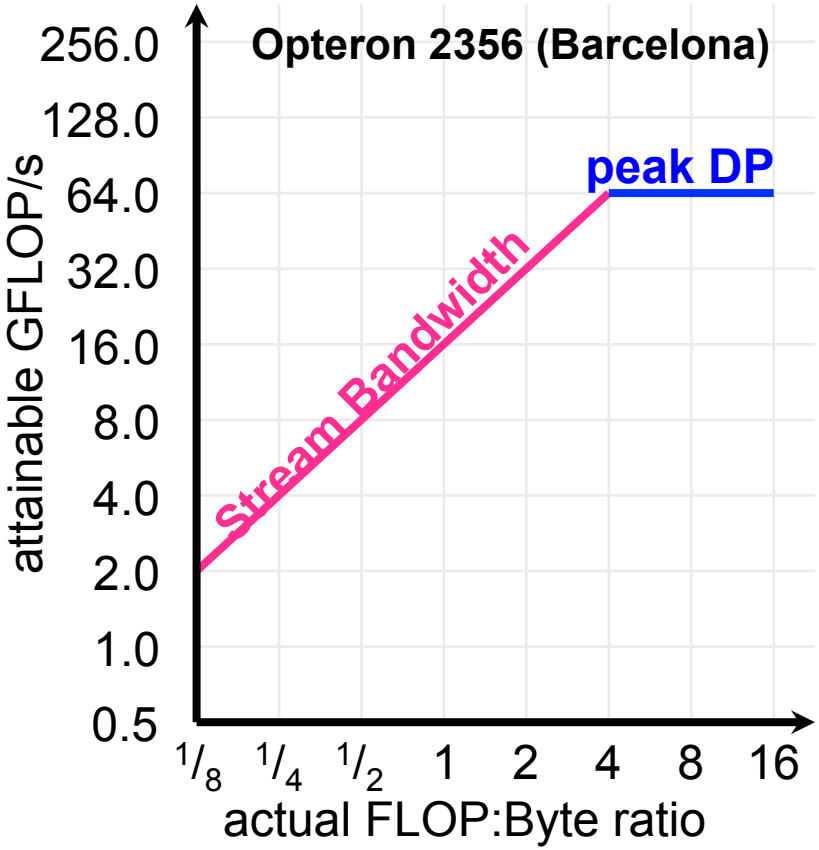
- Time is the maximum of the time required to transfer the data and the time required to perform the floating point operations.



Roofline model
simple bound

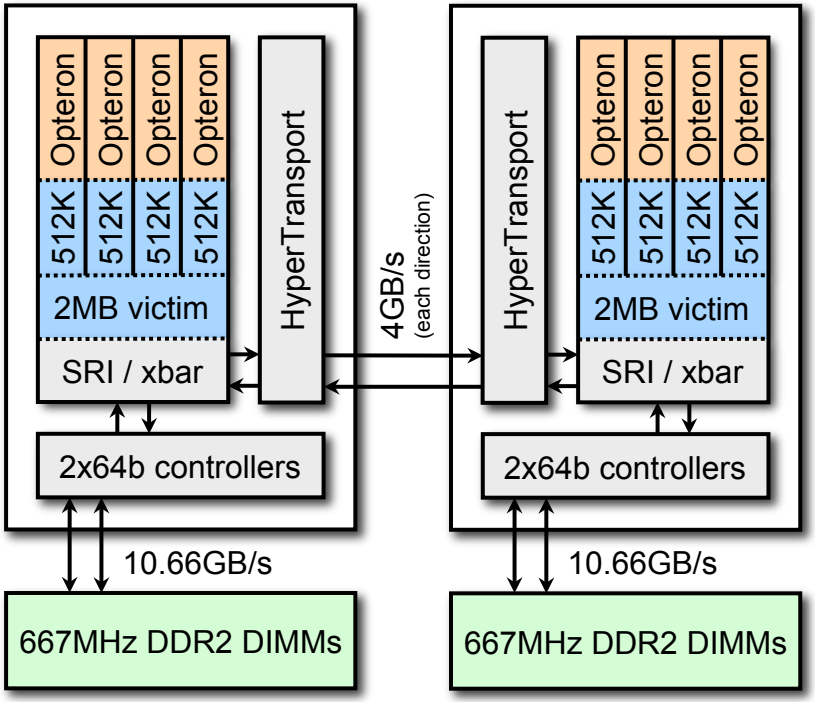
$$\text{Attainable Performance}_{ij} = \min \begin{cases} \text{FLOP/s} \ (\text{with Optimizations}_{1\text{-}i}) \\ \text{AI} * \text{Bandwidth} \ (\text{with Optimizations}_{1\text{-}j}) \end{cases}$$

- Roofline
  - Given the code AI, can inspect the figure to bound performance
  - Provides insights as to which optimizations will potentially be beneficial
- Machine-dependent, code-dependent



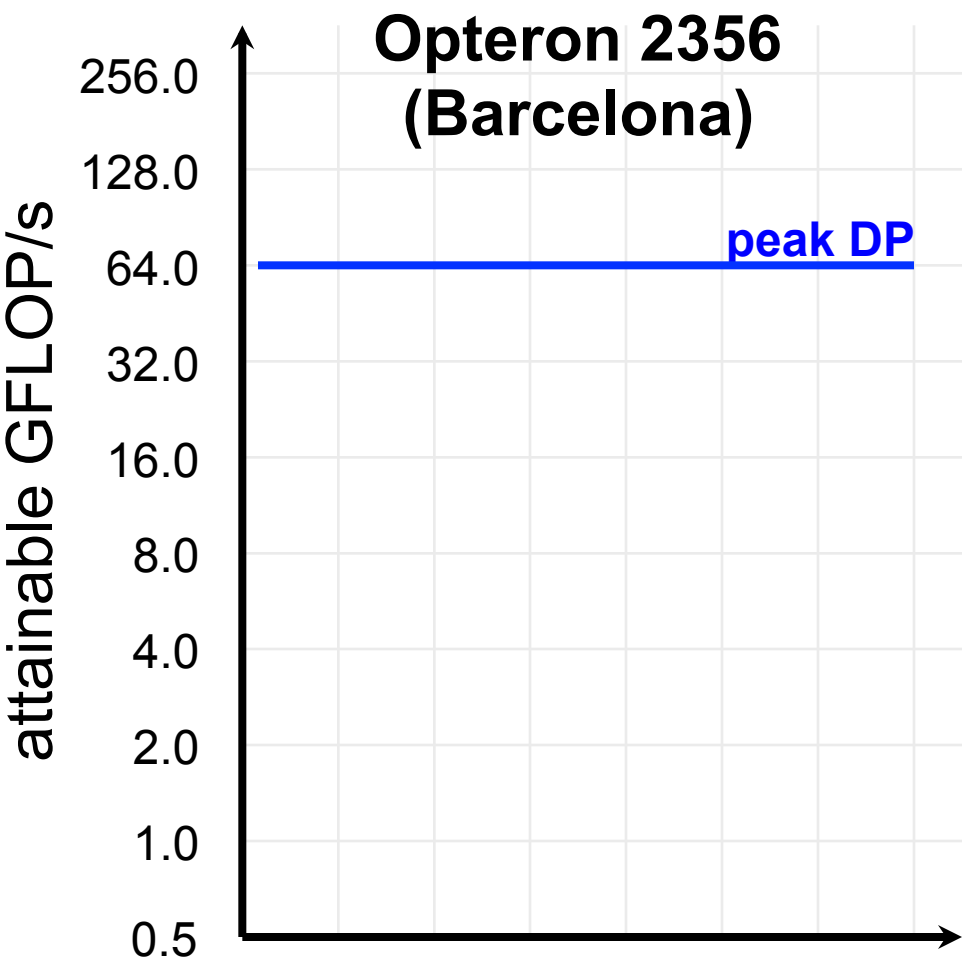
Example

- Consider the Opteron 2356:
  - Dual Socket (NUMA)
  - limited HW stream prefetchers
  - quad-core (8 total)
  - 2.3GHz
  - 2-way SIMD (DP)
  - separate FPMUL and FPADD datapaths
  - 4-cycle FP latency



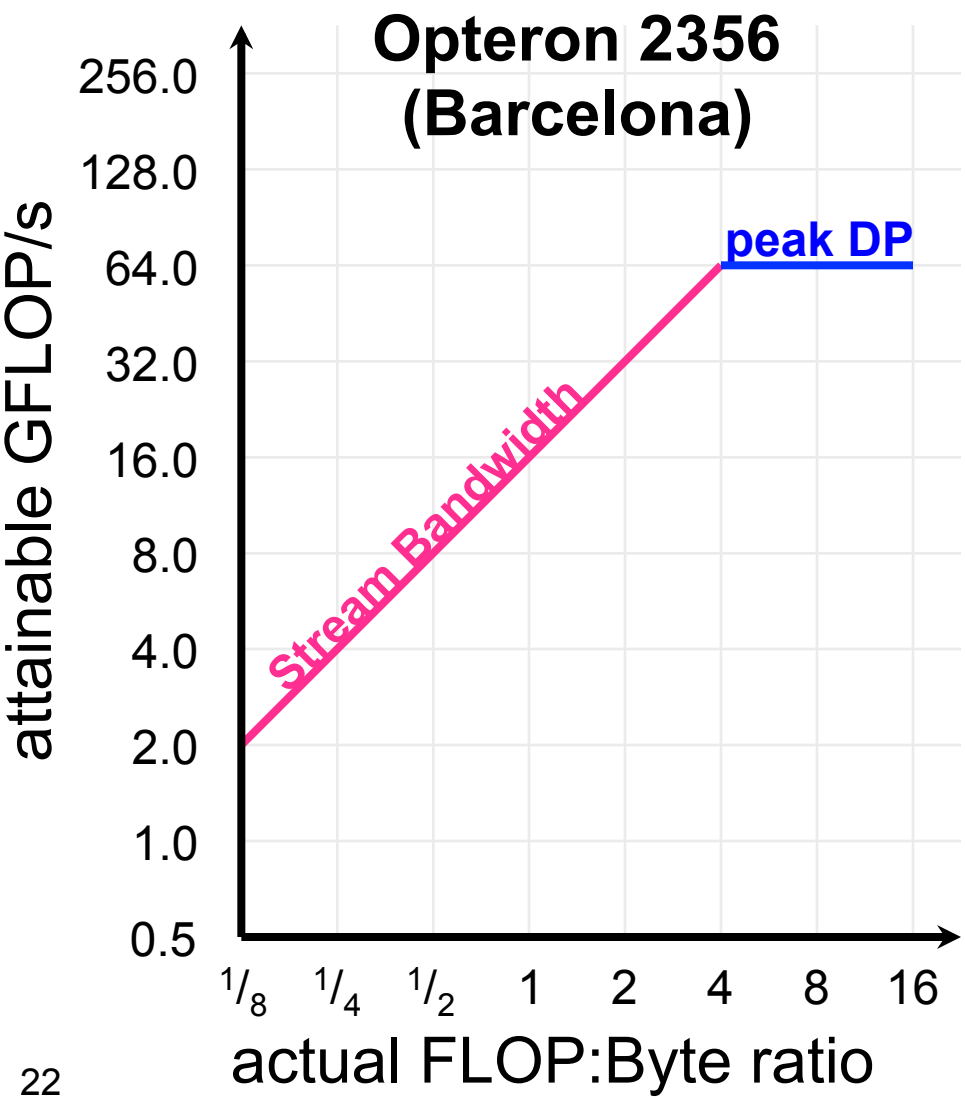
- Assuming **expression of parallelism** is the challenge on this architecture, what would the roofline model look like ?

Roofline Model
Basic Concept



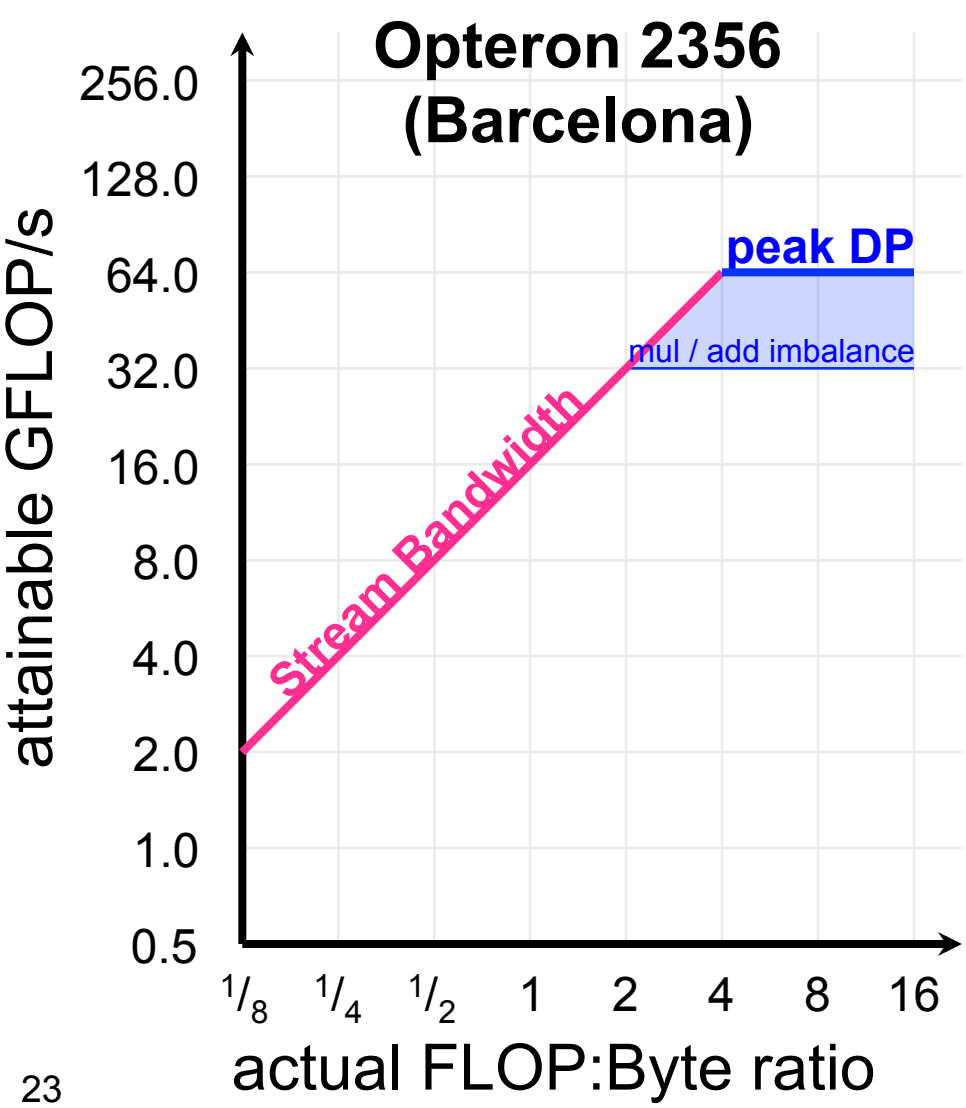
- Naively, one might assume peak performance is always attainable.

Roofline Model
Basic Concept



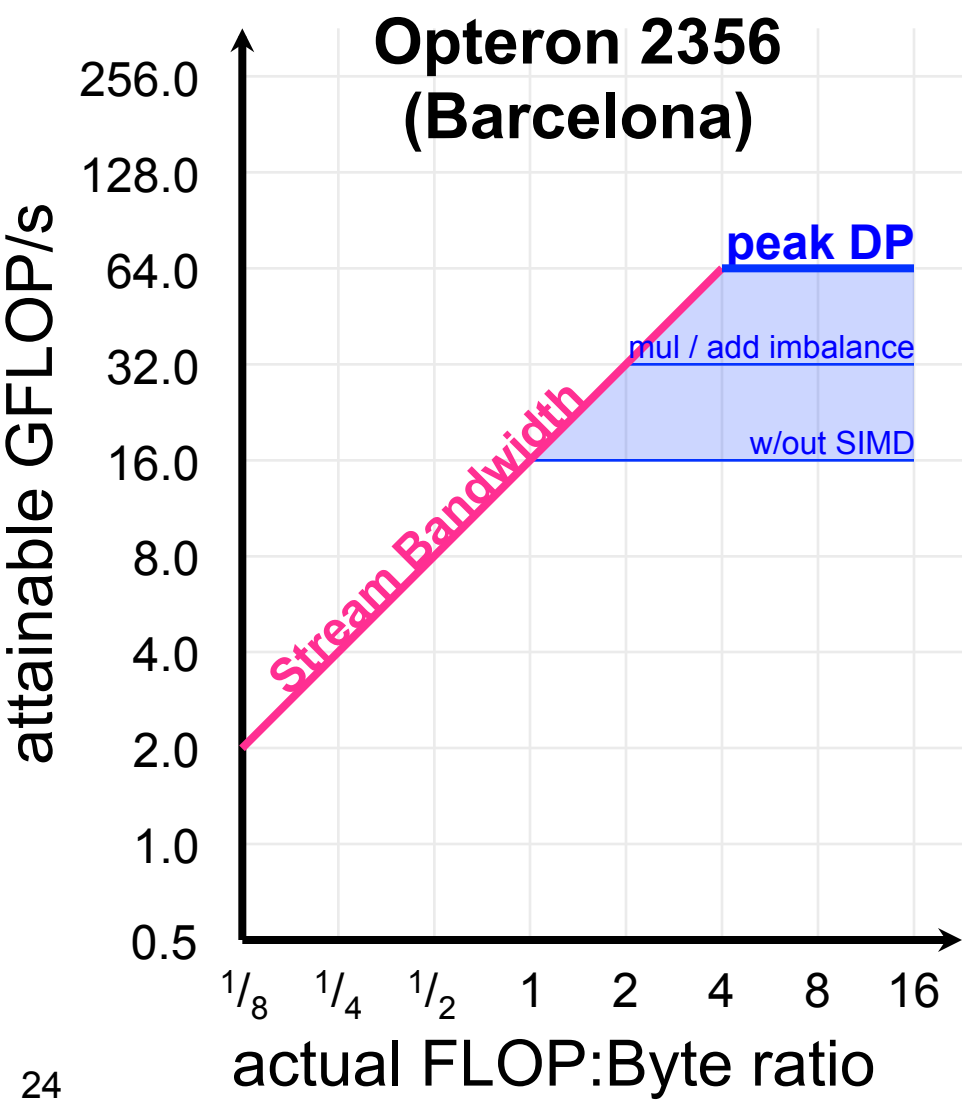
- However, with a lack of locality, **DRAM bandwidth can be a bottleneck**
- Plot on log-log scale
- Given AI, we can easily bound performance
- But architectures are much more complicated
- We will bound performance as we eliminate specific forms of in-core parallelism

Roofline Model
computational ceilings



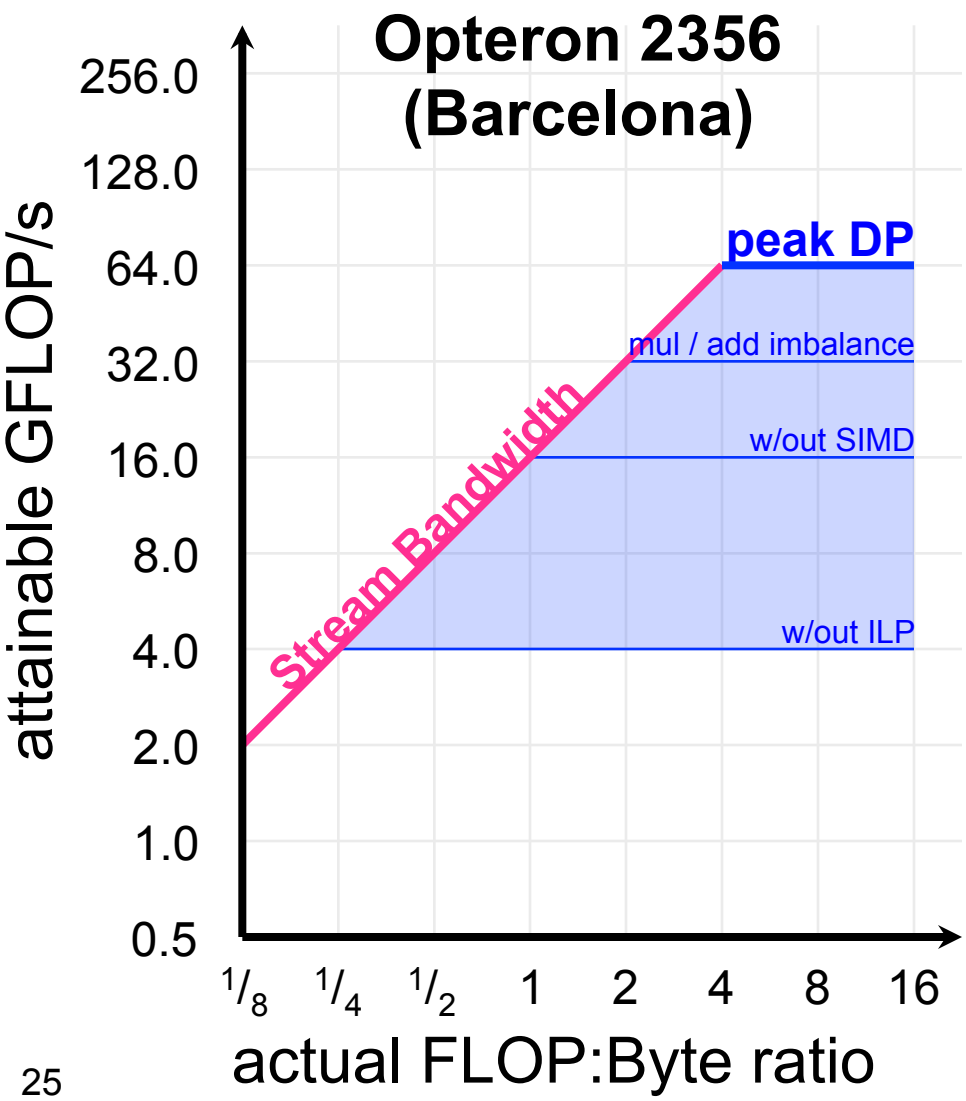
- Opterons have dedicated multipliers and adders.
- If the code is dominated by adds, then attainable performance is half of peak.
- We call these **Ceilings**
- They act like constraints on performance

Roofline Model
computational ceilings



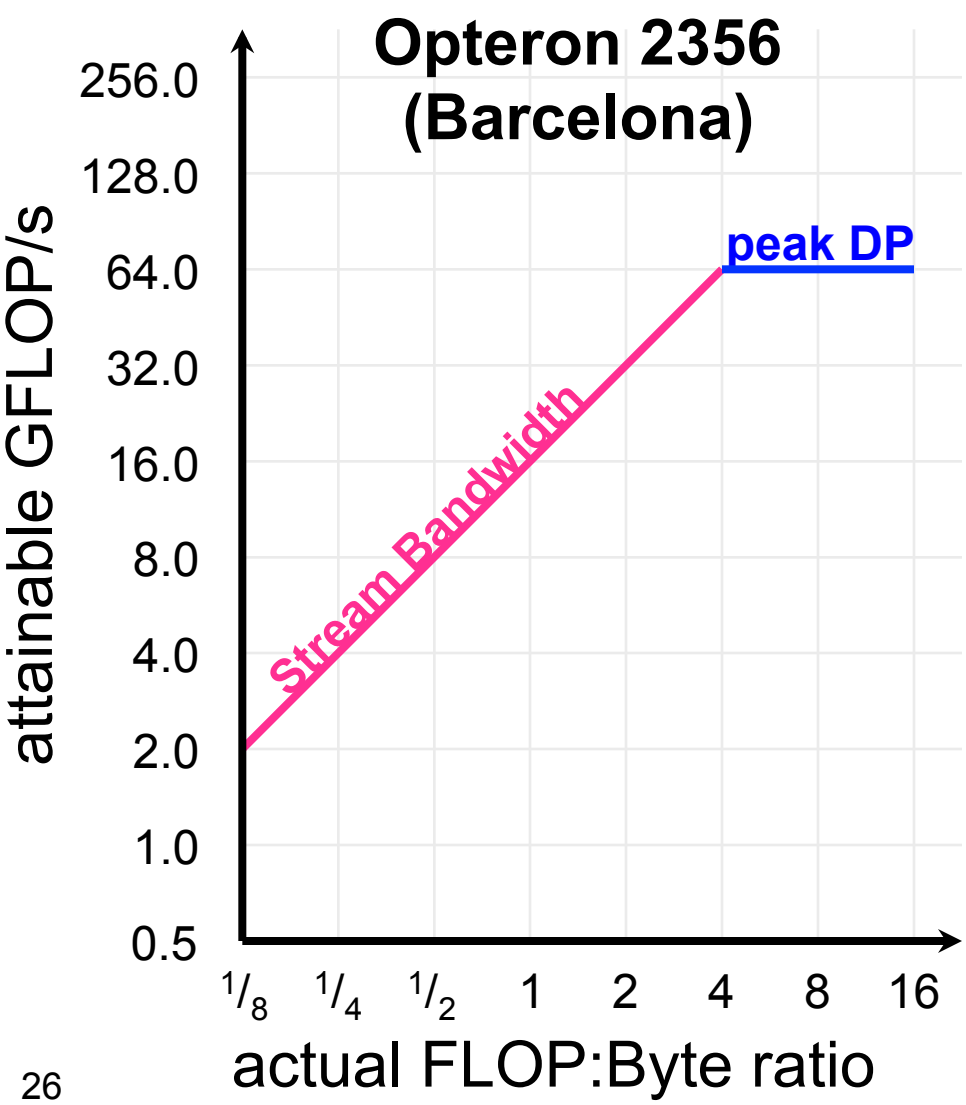
- Opterons have 128-bit datapaths.
- If instructions aren't SIMDized, attainable performance will be halved

Roofline Model
computational ceilings



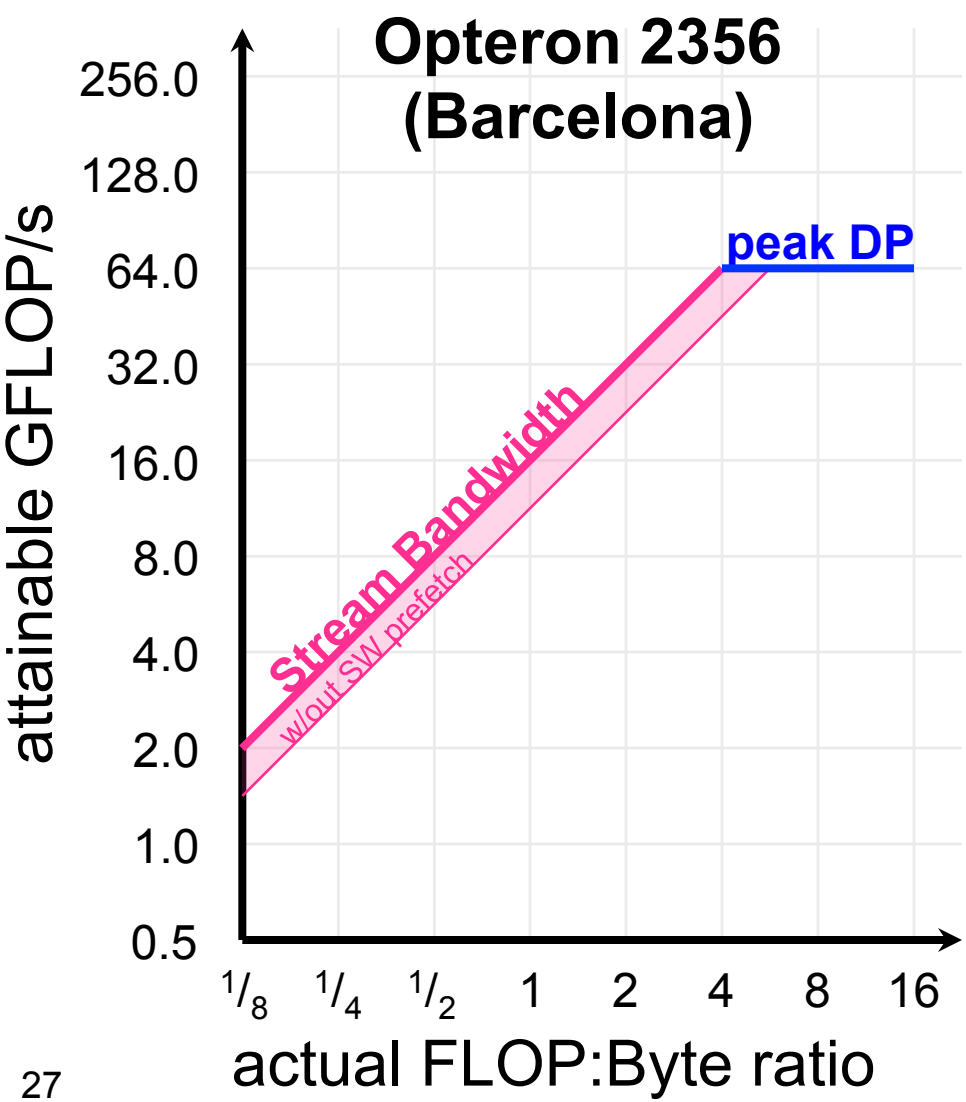
- On Opterons, floating-point instructions have a 4 cycle latency.
- If we don't express 4-way ILP, performance will drop by as much as 4x

Roofline Model
communication ceilings



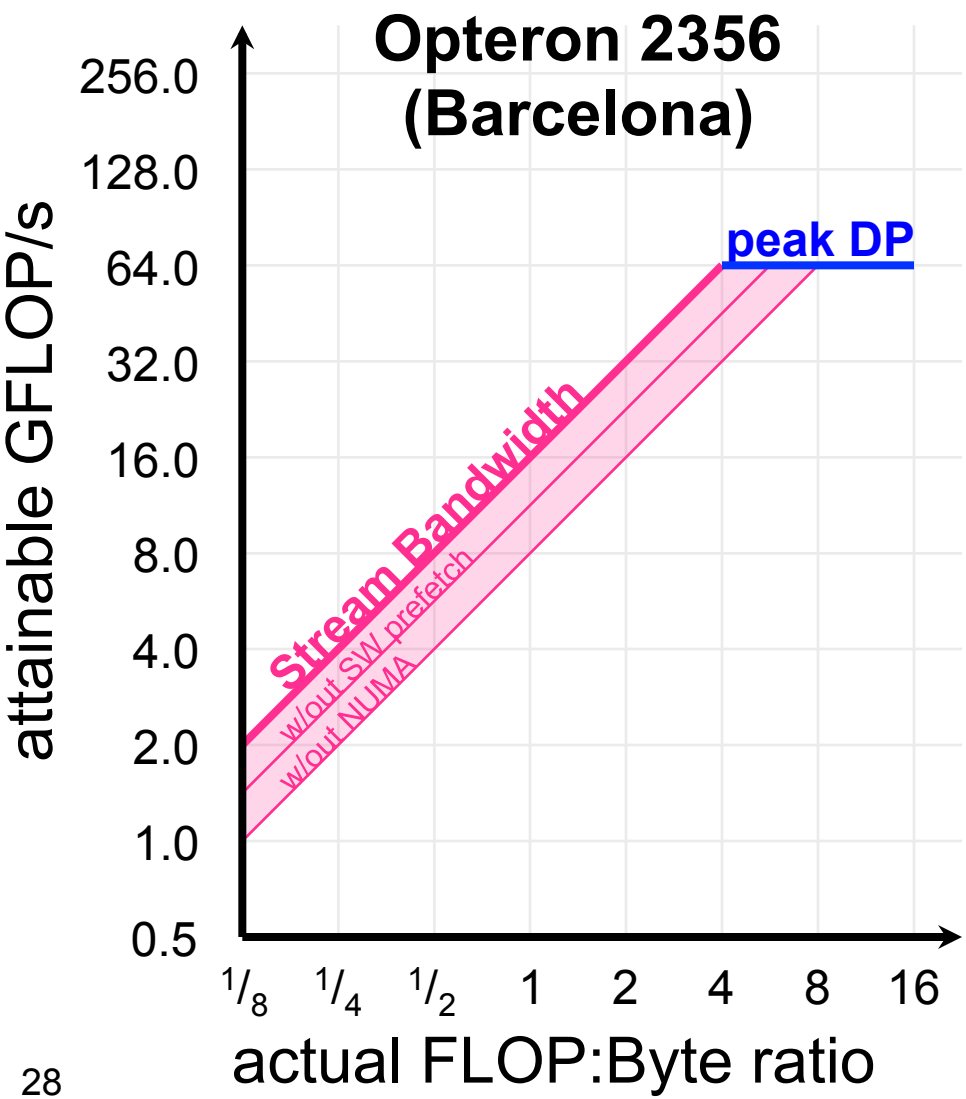
- We can perform a similar exercise taking away parallelism from the memory subsystem

Roofline Model
communication ceilings



- Explicit software prefetch instructions are required to achieve peak bandwidth

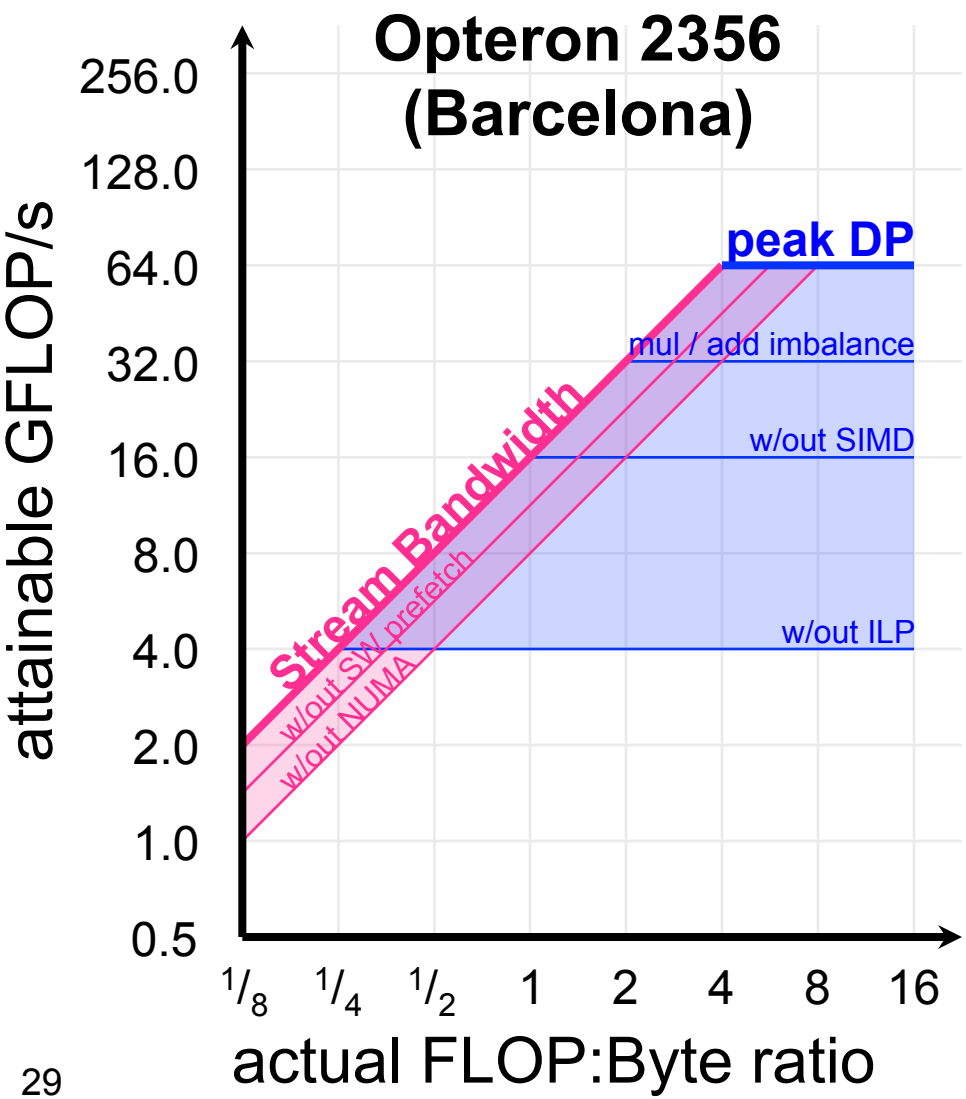
Roofline Model
communication ceilings



- Opterons are NUMA
- As such memory traffic must be correctly balanced among the two sockets to achieve good Stream bandwidth.
- We could continue this by examining strided or random memory access patterns

Roofline Model

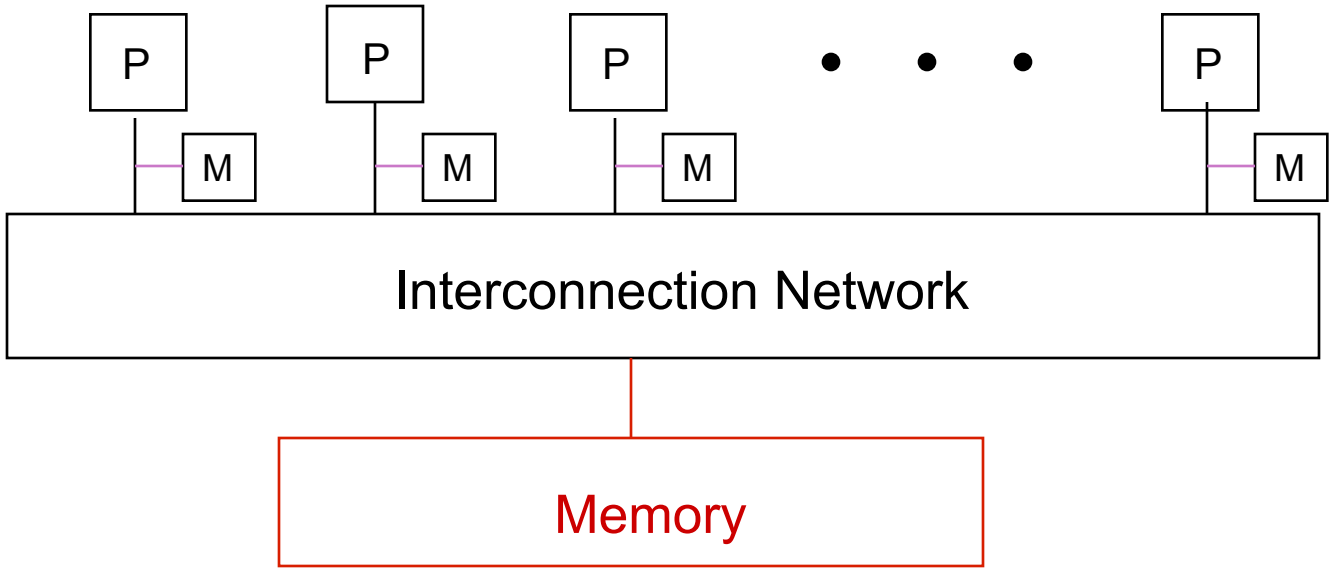
computation + communication ceilings



- We may bound performance based on the combination of expressed in-core parallelism and attained bandwidth.

- Parallel machines
  - Shared memory
  - Shared address space
  - Message passing
  - Data parallel: vector processors
  - Clusters of SMPs
  - Grid
- Programming model reflects hardware
  - Historically, tight coupling
  - Today, portability is important

A generic parallel architecture



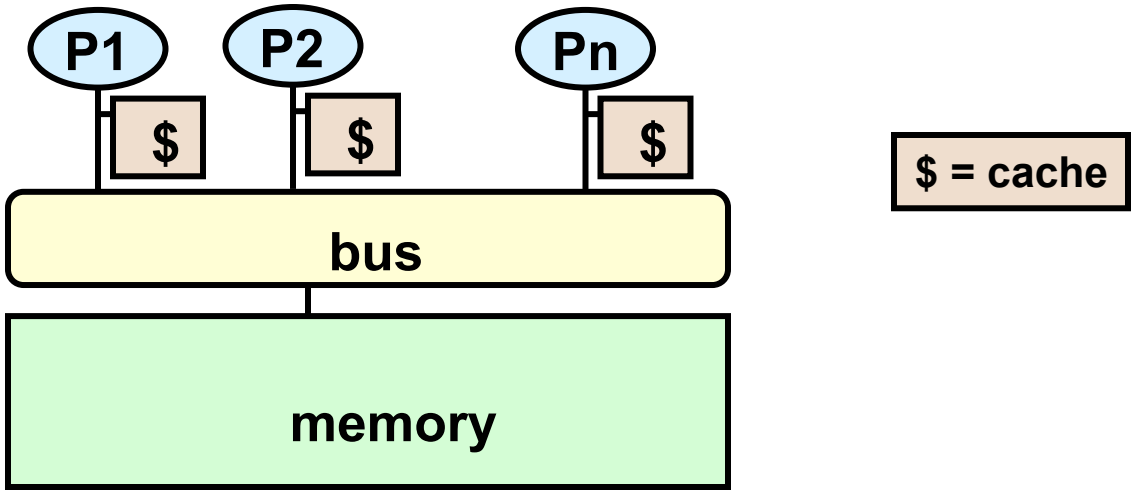
- Where is the memory physically located?

Parallel programming models

- Control
  - How is parallelism created?
  - What orderings exist between operations?
  - How do different threads of control synchronize?
- Data
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?
- Operations
  - What are the atomic (indivisible) operations?
- Cost
  - How do we account for the cost of each of the above?

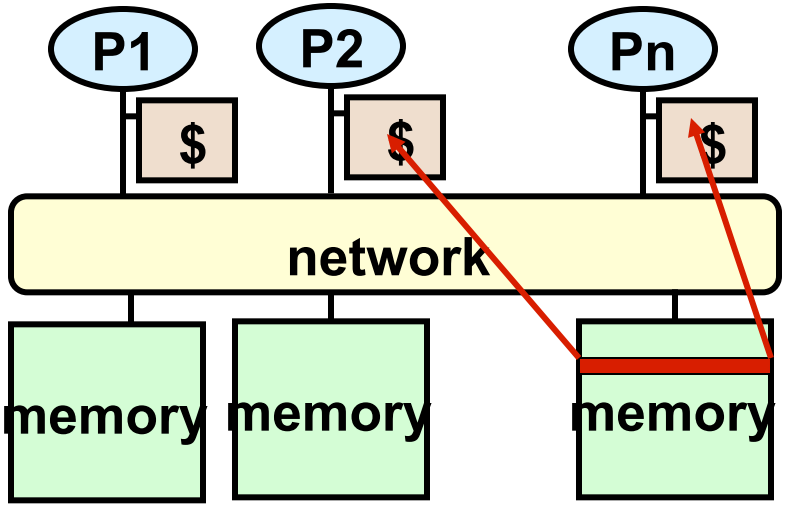
Machine model 1a: shared memory

- **Processors all connected to a** common **shared memory.**
  - **Processors → sockets → dies → cores**
  - **Intel, AMD : multicore, multithread chips**
- **Difficulty scaling to large numbers of processors**
  - **<= 32 processors typical**
- Memory access:
  - **uniform memory access (UMA)**
  - **Nonuniform memory access (NUMA, more common now)**
- **Cost: much cheaper to access data in cache than main memory.**



Machine model 1b: distributed shared memory

- Memory is logically shared, but physically distributed (e.g., SGI Altix)
  - Any processor can access any address in memory
  - Cache lines (or pages) are passed around machine
  - Limitation is *cache coherency protocols* – how to keep cached copies of the same address consistent



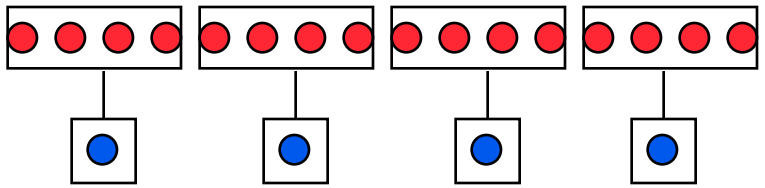
Simple programming example

Consider dot product:

- Parallel Decomposition: $\sum_{i=0}^{n-1} x(i) * y(i)$
  - Each evaluation and each partial sum is a task.
- Assign n/p numbers to each of p procs
  - Each computes independent "private" results and partial sum.
  - One (or all) collects the p partial sums and computes the global sum.

Two Classes of Data:

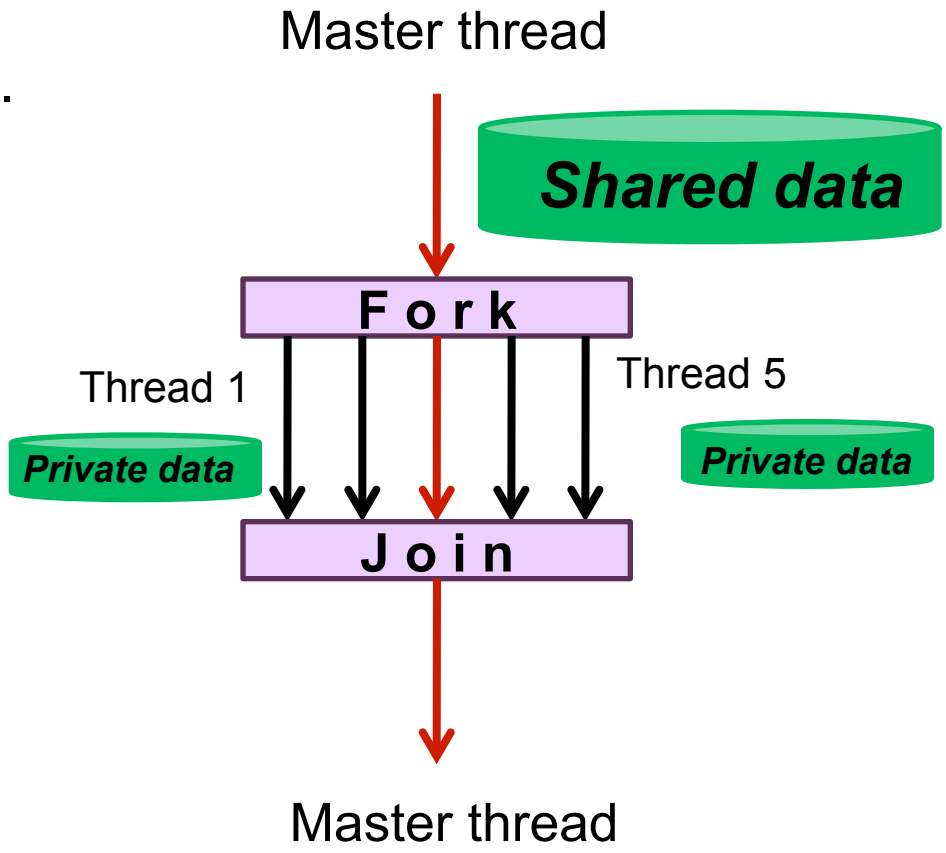
- Logically Shared
  - The original n numbers, the global sum.
- Logically Private
  - The individual partial sums.



OpenMP shared-memory programming

- Share the node address space.

- Most data shared within node.
- Threads communicate via memory read & write.
- Concurrent write to shared data needs *locking* or *atomic operation.*



Incorrect program

```
int s = 0;
```

```
Thread 1

  for i = 0, n/2-1
      s = s + x(i)*y(i)
```

```
Thread 2

  for i = n/2, n-1
      s = s + x(i)*y(i)
```

- There is a race condition on variable s in the program
- A race condition or data race occurs when:
  - two threads access the same variable, and at least one does a write.
  - the accesses are concurrent (not synchronized) so they could happen simultaneously

Correct program

```
int s = 0;
Lock lk;
```

Thread 1

```
local_s1= 0
for i = 0, n/2-1
    local_s1 = local_s1 + x(i)*y(i)
   lock(lk);
s = s + local_s1
   unlock(lk);
```

Thread 2

```
local_s2 = 0
for i = n/2, n-1
    local_s2= local_s2 + x(i)*y(i)
 lock(lk);
s = s +local_s2
 unlock(lk);
```

- Since addition is associative, it's OK to rearrange order
- Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - Race condition is fixed by adding locks to critical region (only one thread can hold a lock at a time; others wait for it)
- Shared-memory programming standards: OpenMP, PTHREADS

Dot-product using OpenMP in C

```
int n = 100;
double x[100], y[100];
double s = 0, local_s;

#pragma omp parallel shared (s) private (local_s)
{
   local_s = 0.0;
   #pragma omp for
      for (i = 0; i < n; ++i) {
           local_s = local_s + x[i] * y[i];
      }
   #pragma omp critical
   {
       s = s + local_s;
   }
}
```

Exercise: complete this program, and run it with at least 4 threads.

OpenMP tutorial:

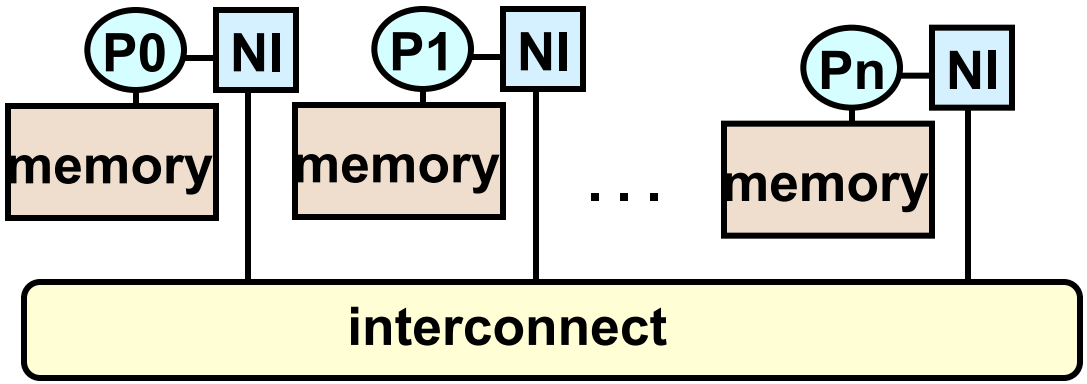
https://computing.llnl.gov/tutorials/openMP/

OpenMP sample programs:

https://computing.llnl.gov/tutorials/openMP/exercise.html

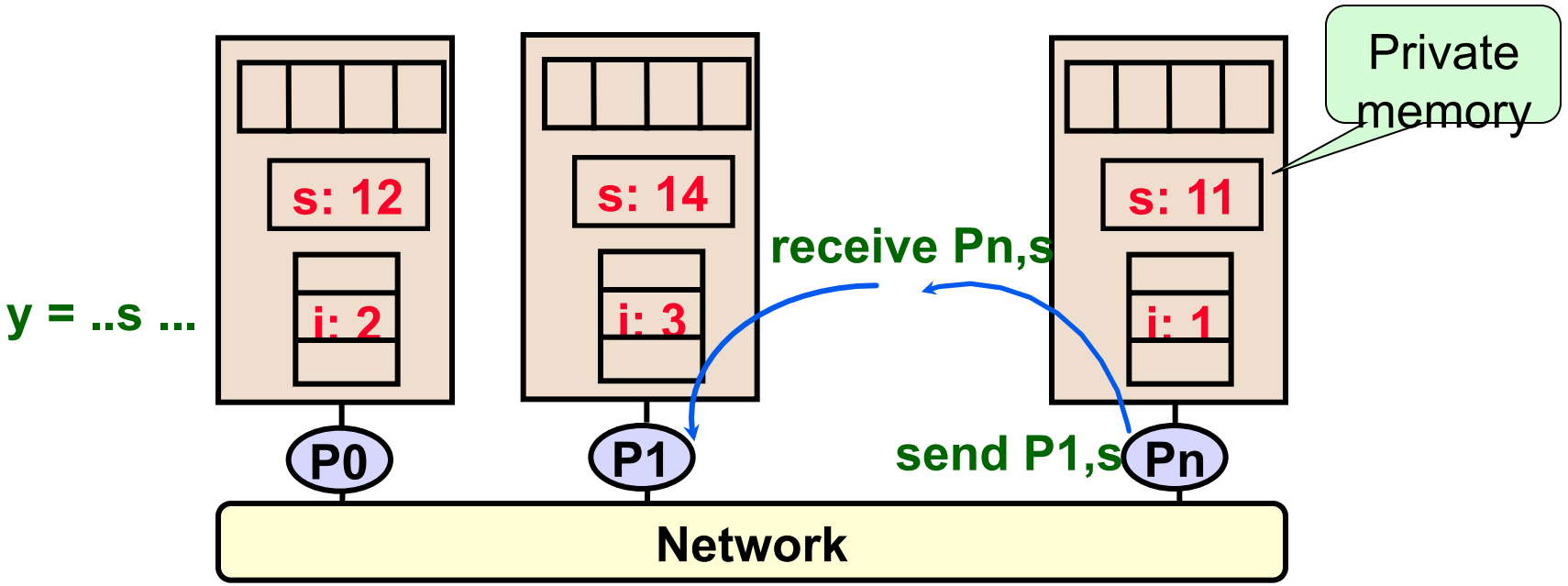
Machine model 2: distributed memory

- Cray XE6, IBM SP, PC Clusters ..., can be large
- Each processor has its own memory and cache, but cannot directly access another processor's memory.
- Each "node" has a Network Interface (NI) for all communication and synchronization.



Programming Model 2: Message Passing

- Program consists of a collection of named processes
  - **Usually fixed at program startup time**
  - **Thread of control plus local address space -- NO shared data**
- Processes communicate by explicit send/receive pairs
  - **Coordination is implicit in every communication event.**
  - **Message Passing Interface (MPI) is the most commonly used SW**



Distributed dot product

Processor 1

```
s = 0
for i = 0, n/2-1
      s = s + x(i)*y(i)
MPI_Recv(s_remote, p2,...)
MPI_Send(s, p2, ...)
s = s + s_remote
```

Processor 2

```
s = 0
for i = 0, n/2-1
      s = s + x(i)*y(i)
MPI_Send(s, p1, ...)
MPI_Recv(s_remote, p1,...)
s = s + s_remote
```

MPI – the de facto standard

- MPI has become the de facto standard for parallel computing using message passing
- Pros and Cons
  - MPI created finally a standard for applications development in the HPC community → portability
  - The MPI standard is a least common denominator building on mid-80s technology, so may discourage innovation

- MPI tutorial:
  https://computing.llnl.gov/tutorials/mpi/
  https://computing.llnl.gov/tutorials/mpi/exercise.html

Other machines & programming models

- Data parallel
  - SIMD
  - Vector machines (often has compiler support)
    - **SSE, SSE2 (Intel: Pentium/IA64)**
    - **Altivec (IBM/Motorola/Apple: PowerPC)**
    - **VIS (Sun: Sparc)**
  - GPU, at a larger scale
- Hybrid: cluster of SMP/multicore/GPU node
- MPI + X
  - X = OpenMP, CUDA/OpenCL, …
- Global Address Space programming (GAS languages)
  - UPC, Co-Array Fortran
  - Local and shared data, as in shared memory model
  - But, shared data is partitioned over local processes

Outline

- Parallel machines and programming models
- Principles of parallel computing performance
  - Models of performance bound
- Design of parallel algorithms

Principles of Parallel Computing

- Finding enough parallelism (Amdahl's Law)
- Granularity – how big should each parallel task be
- Locality – moving data costs more than arithmetic
- Load balance – don't want 1K processors to wait for one slow one
- Coordination and synchronization – sharing data safely
- Performance modeling/debugging/tuning

Finding enough parallelism

- Suppose only part of an application is parallel
- Amdahl's law
  - Let s be the fraction of work done sequentially, so (1-s) is fraction parallelizable
  - P = number of cores

$$\text{Speedup}(P) = \text{Time}(1)/\text{Time}(P)$$

$$<= 1/(s + (1-s)/P)$$

$$<= 1/s$$

(e.g., s = 1% → speedup <= 100)

- Even if the parallel part speeds up perfectly, performance is limited by the sequential part

Overhead of parallelism

- Given enough parallel work, this is the biggest barrier to getting desired speedup

- Parallelism overheads include:
  - cost of starting a thread or process
  - cost of communicating shared data
  - cost of synchronizing
  - extra (redundant) computation

- Each of these can be in the range of milliseconds (= millions of flops) on some systems

- Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (I.e. large granularity), but not so large that there is not enough parallel work

Performance properties of a network

- Latency: delay between send and receive times
  - Latency tends to vary widely across machines
  - Vendors often report hardware latencies (wire time)
  - Application programmers care about software latencies (user program to user program)
  - ➢ Latency is important for programs with many small messages (e.g., sparse matrices)
- The bandwidth of a link measures how much volume can be transferred in unit-time (e.g., MBytes/sec)
  - ➢ Bandwidth is important for applications with mostly large messages (e.g., dense matrices)

Latency and bandwidth model

- Time to send a message of length n is roughly

$$\begin{aligned} \text{Time} &= \text{latency} + n * \text{time\_per\_word} \\ &= \text{latency} + n / \text{bandwidth} \end{aligned}$$

Called "α−β model" and written:

$$\textbf{Time} = \alpha + \beta \times n$$

- Usually α >> β >> time per flop

→ One long message is cheaper than many short ones.

- Can do hundreds or thousands of flops for cost of one message

$$\alpha + n*\beta \ << \ n*(\alpha + 1*\beta)$$

Communication versus F.P. speed

Cray XE6 at NERSC, LBNL: dual-socket x 2-die x 6-core, 24 cores

- Inter-node
  - FP Peak/core: 8.5 Gflops → time_per_flop = 0.11 nanosec
  - Communication using MPI

$$\alpha = 1.5\ micro\,\text{sec}\ (\approx 13{,}636\ \text{FPs})$$

$$1/\beta = 5.8\ \text{GB/s}, \quad \beta = 0.17\ nano\,\text{sec}\ (\approx 12\ \text{FPs/double-word})$$

- Intra-node (on-node memory): 24 cores
  - 1.3 - 2.6 GB/core

→ Extremely difficult for accurate performance prediction.

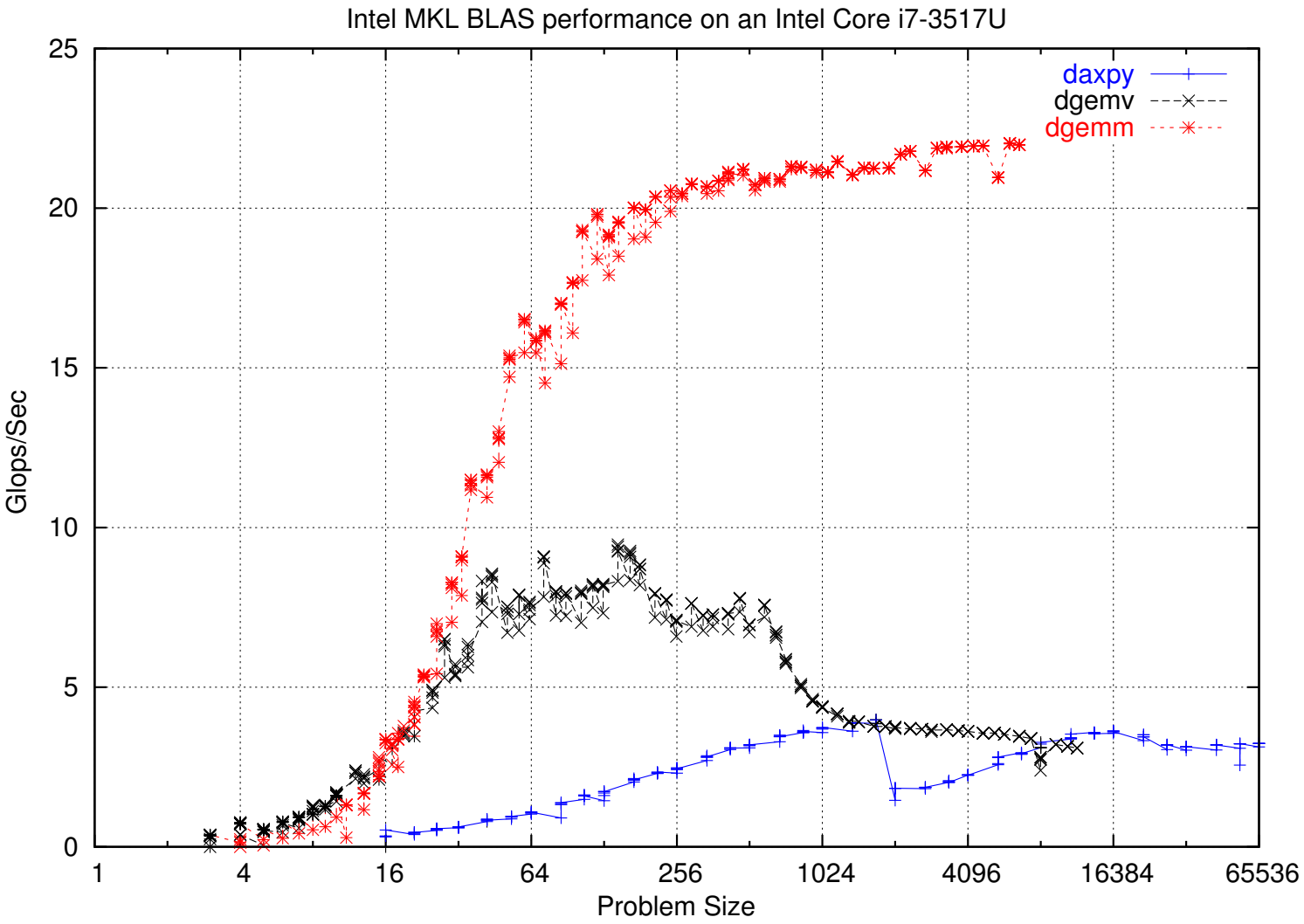
BLAS – Basic Linear Algebra Subroutines

http://www.netlib.org/blas/blast-forum/

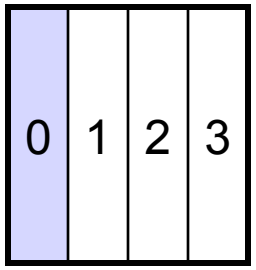
- Building blocks for all linear algebra
- Parallel versions call serial versions on each processor
  - So they must be fast!
- Reuse ratio: q = # flops / # mem references (i.e. Arithmetic Intensity)
  - The larger is q, the faster the algorithm can go in the presence of memory hierarchy

BLAS level	Ex.	# mem refs	# flops	q
1	"Axpy", Dot prod	3n	$2n^1$	2/3
2	Matrix-vector mult	n^2	$2n^2$	2
3	Matrix-matrix mult	$4n^2$	$2n^3$	n/2

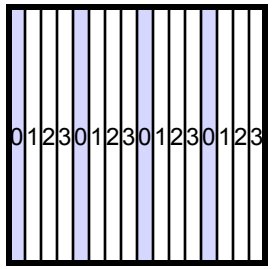
BLAS performance



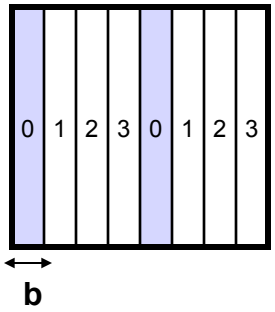
Parallel data layouts for matrices



1) 1D Column Blocked Layout

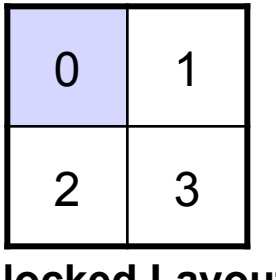


2) 1D Column Cyclic Layout

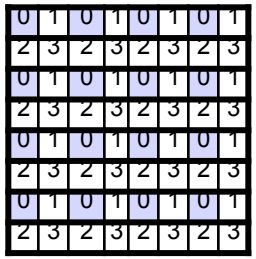


3) 1D Column Block Cyclic Layout

4) Row versions of the previous layouts



5) 2D Row and Column Blocked Layout



Generalizes others

6) 2D Row and Column Block Cyclic Layout

Summary

- Performance bounds and models
  - Roofline model: captures on-node memory speed
  - Amdahl's Law: upper bound of speedup
  - "$\alpha-\beta$ model" (latency-bandwidth): captures network speed
  - Strong/weaking scaling: algorithm scalability (Lectures 3-4)
- Hybrid programming becomes necessary
  - MPI + X
- Sparse matrix algorithms have much lower arithmetic density
  - Critical to reduce memory access and communication

References

- OpenMP tutorial: https://computing.llnl.gov/tutorials/openMP/
- MPI tutorial: https://computing.llnl.gov/tutorials/mpi/
- “The Landscape of Parallel Processing Research: The View from Berkeley”

  http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/ECS-2006-183.pdf
  - Contains many references
- Jim Demmel, Kathy Yelick, et al., UCB/CS267 lecture notes for parallel computing class

  http://www.cs.berkeley.edu/~demmel/cs267_Spr13/
- S. Williams, A. Waterman, D. Patterson, "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures", Communications of the ACM (CACM), April 2009.
- MAGMA, Matrix Algebra on GPU and Multicore Architectures,

  http://icl.cs.utk.edu/magma/index.html
- PLASMA, The Parallel Linear Algebra for Scalable Multi-core Architectures

  http://icl.cs.utk.edu/plasma/

Exercises

1. Complete and run the OpenMP code to perform dot-product.
   - Other examples: https://computing.llnl.gov/tutorials/openMP/exercise.html
2. Write an OpenMP code to perform GEMM
   - Validate correctness
   - How fast does your program run on one node of the cluster?
3. Run the following MPI codes in Hands-On-Exercises/
   - Hello-MPI
   - DOT-MPI (Is it simpler than OpenMP DOT ?)
4. Write an MPI program to find the Maximum and Minimum entries of an array.
5. Run the MPI ping-pong benchmark code in Hands-On-Exercises/LatencyBandwidth/ directory, to find {alpha, beta} on your machine.

Exercises

6. Run the MPI code to perform GEMM
   - How to distribute the matrix?
   - The parallel algorithm is called SUMMA

7. Write a hybrid MPI + OpenMP code to perform GEMM
   - Use 2 nodes of the cluster (2 x 12 cores)
   - Can have various MPI and OpenMP configurations:
     2 MPI tasks X 12 threads, 4 MPI tasks X 6 threads, …
   - Other tuning parameters:

Cannon's matrix-multiplication algorithm

- Views the processes as being arranged in a virtual two-dimensional square array. It uses this array to distribute the matrices A, B, and the result matrix C in a block fashion.
- If n x n is the size of each matrix and p is the total number of processes, then each matrix is divided into square blocks of size $n/\sqrt{p} \times n/\sqrt{p}$
- Process $P_{i,j}$ in the grid is assigned the $A_{i,j}$, $B_{i,j}$, and $C_{i,j}$ blocks of each matrix.
- The algorithm proceeds in $\sqrt{p}$ steps. In each step, every process multiplies the local blocks of matrices A and B, and then sends the block of A to the leftward process, and the block of B to the upward process.

4th Gene Golub SIAM Summer School, 7/22 – 8/7, 2013, Shanghai

Lecture 2

Sparse matrix data structures, graphs, manipulation

Xiaoye Sherry Li
Lawrence Berkeley National Laboratory, USA
xsli@lbl.gov

crd-legacy.lbl.gov/~xiaoye/G2S3/

Lecture outline

- PDE → discretization → sparse matrices
- Sparse matrix storage formats
  - Sparse matrix-vector multiplication with various formats
- Graphs associated with the sparse matrices
- Distributed sparse matrix-vector multiplication

Solving partial differential equations

- Hyperbolic problems (waves):
  - Sound wave (position, time)
  - Use explicit time-stepping: Combine nearest neighbors on grid
  - Solution at each point depends on neighbors at previous time
- Elliptic (steady state) problems:
  - Electrostatic potential (position)
  - Everything depends on everything else, use implicit method
  - This means locality is harder to find than in hyperbolic problems
  - Canonical example is the Poisson equation

$$\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 + \partial^2 u/\partial z^2 = f(x,y,z)$$

- Parabolic (time-dependent) problems:
  - Temperature (position, time)
  - Involves an elliptic solve at each time-step

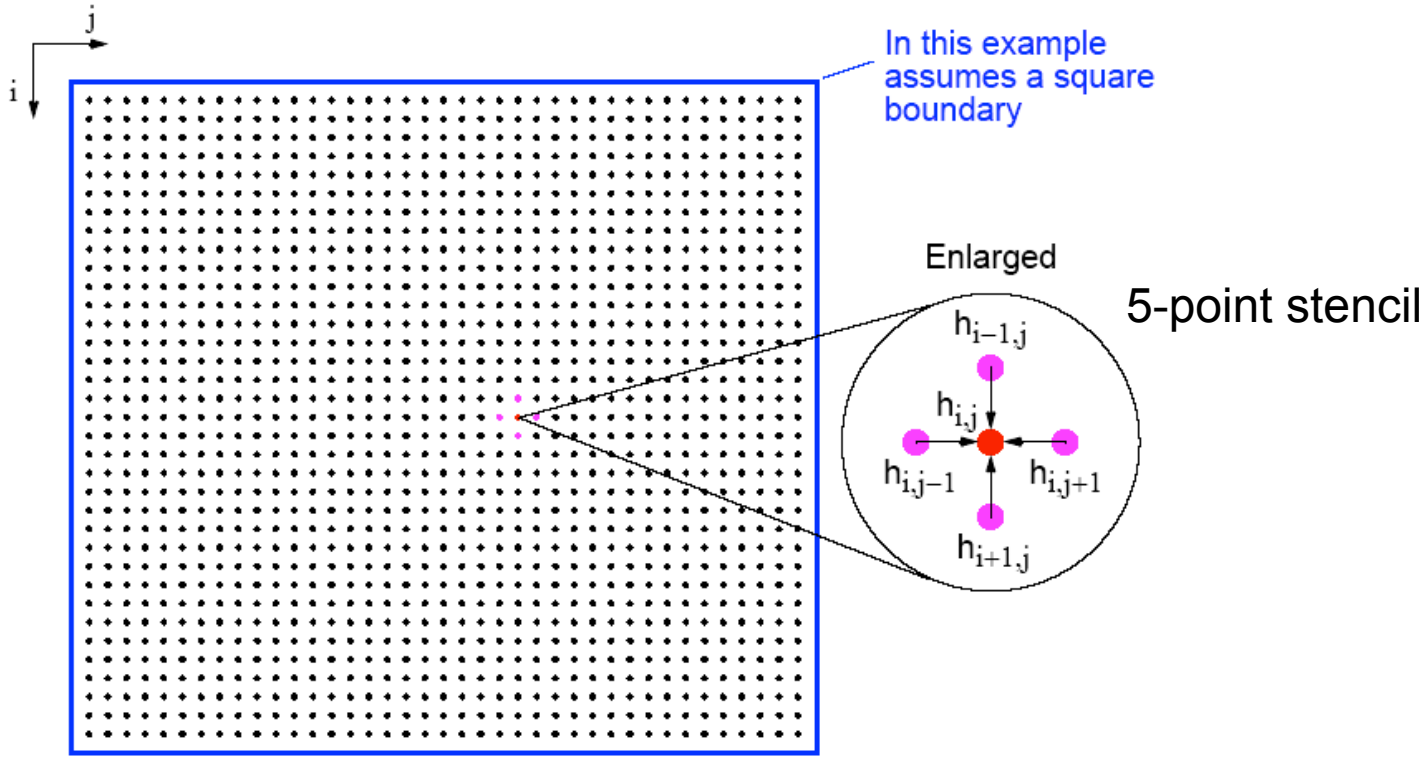
PDE discretization leads to sparse matrices

- Poisson equation in 2D:

$$\frac{\partial^2 u}{\partial x^2}(x,y) + \frac{\partial^2 u}{\partial y^2}(x,y) = f(x,y), \quad (x,y) \in R$$

$$u(x,y) = g(x,y), \quad (\mathrm{x},\mathrm{y}) \text{ on the boundary}$$

- Finite difference discretization → stencil computation

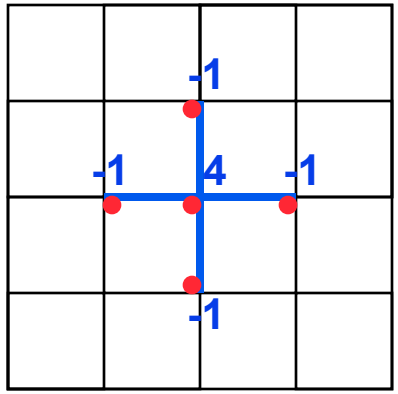


Matrix view

$$4 \cdot u(i,j) - u(i-1,j) - u(i+1,j) - u(i,j-1) - u(i,j+1) = f(i,j)$$

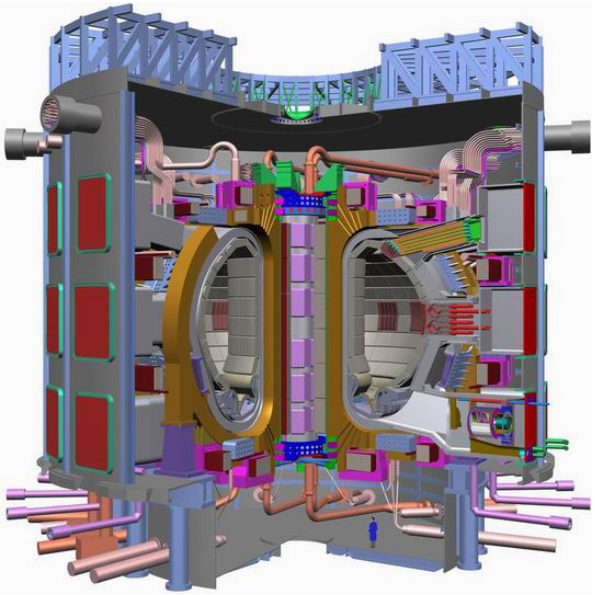
$$A = \left(\begin{array}{ccc:ccc:ccc}
4 & -1 & & -1 & & & & & \\
-1 & 4 & -1 & & -1 & & & & \\
 & -1 & 4 & & & -1 & & & \\
\hdashline
-1 & & & 4 & -1 & & -1 & & \\
 & -1 & & -1 & 4 & -1 & & -1 & \\
 & & -1 & & -1 & 4 & & & -1 \\
\hdashline
 & & & -1 & & & 4 & -1 & \\
 & & & & -1 & & -1 & 4 & -1 \\
 & & & & & -1 & & -1 & 4
\end{array}\right)$$

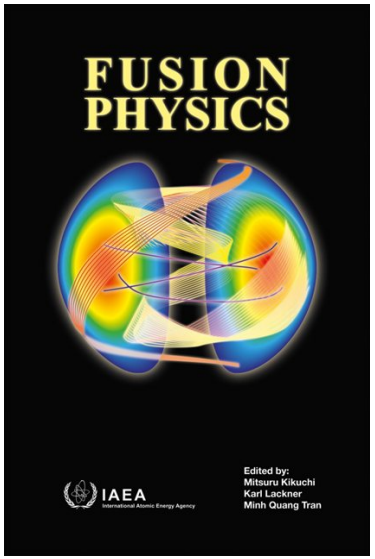
Graph and "stencil"



Application 1: Burning plasma for fusion energy

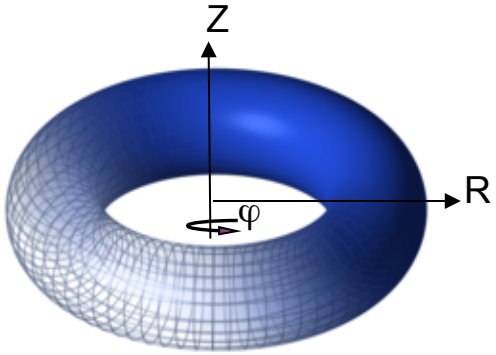
- ITER – a new fusion reactor being constructed in Cadarache, France
  - International collaboration: China, the European Union, India, Japan, Korea, Russia, and the United States
  - Study how to harness fusion, creating clean energy using nearly inexhaustible hydrogen as the fuel. ITER promises to produce 10 times as much energy than it uses — but that success hinges on accurately designing the device.
- One major simulation goal is to predict microscopic MHD instabilities of burning plasma in ITER. This involves solving extended and nonlinear Magnetohydrodynamics equations.





Application 1: ITER modeling

- US DOE SciDAC project (Scientific Discovery through Advanced Computing)
  - Center for Extended Magnetohydrodynamic Modeling (CEMM), PI: S. Jardin, PPPL
- Develop simulation codes to predict microscopic MHD instabilities of burning magnetized plasma in a confinement device (e.g., tokamak used in ITER experiments).
  - Efficiency of the fusion configuration increases with the ratio of thermal and magnetic pressures, but the MHD instabilities are more likely with higher ratio.
- Code suite includes M3D-C^1, NIMROD



(S. Jardin)

- At each φ = constant plane, scalar 2D data is represented using 18 degree of freedom quintic triangular finite elements Q_{18}
- Coupling along toroidal direction

ITER modeling: 2-Fluid 3D MHD Equations

$$\frac{\partial n}{\partial t} + \nabla \bullet (nV) = 0 \qquad \text{continuity}$$

$$\frac{\partial B}{\partial t} = -\nabla \times E, \quad \nabla \bullet B = 0, \quad \mu_0 J = \partial \times B \qquad \text{Maxwell}$$

$$nM_t \left(\frac{\partial V}{\partial t} + V \bullet \nabla V \right) + \nabla p = J \times B - \nabla \bullet \Pi_{GV} - \nabla \bullet \Pi_{\mu} \qquad \text{Momentum}$$

$$E + V \times B = \eta J + \frac{1}{ne} (J \times B - \nabla p_e - \nabla \bullet \Pi_e) \qquad \text{Ohm's law}$$

$$\frac{3}{2} \frac{\partial p_e}{\partial t} + \nabla \bullet \left(\frac{3}{2} p_e V \right) = -p_e \nabla \bullet \nabla + \eta J^2 - \nabla \bullet q_e + Q_{\Delta} \qquad \text{electron energy}$$

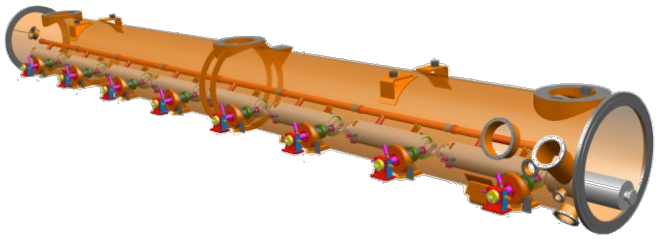
$$\frac{3}{2} \frac{\partial p_i}{\partial t} + \nabla \bullet \left(\frac{3}{2} p_i V \right) = -p_i \nabla \bullet \nabla - \Pi_{\mu} \bullet \nabla V - \nabla \bullet q_i - Q_{\Delta} \qquad \text{ion energy}$$

The objective of the M3D-C^1 project is to solve these equations as accurately as possible in 3D toroidal geometry with realistic B.C. and optimized for a low-β torus with a strong toroidal field.

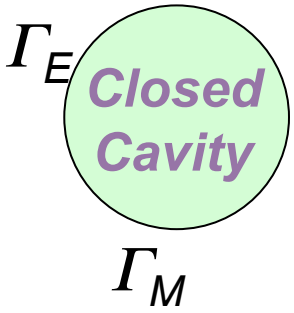
Application 2: particle accelerator cavity design

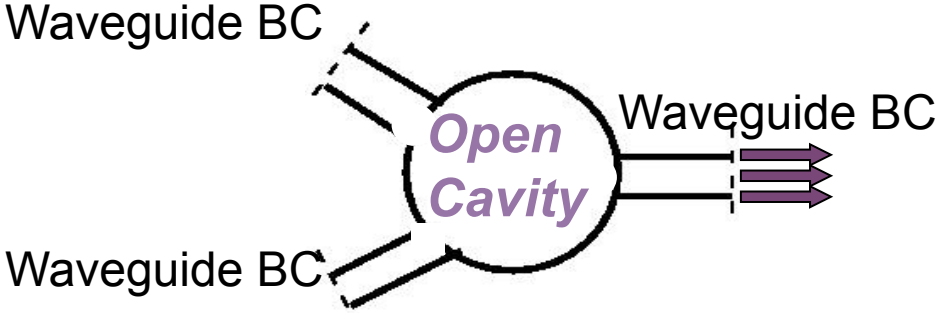
- US DOE SciDAC project
  - Community Petascale Project for Accelerator Science and Simulation (ComPASS), PI: P. Spentzouris, Fermilab
- **Development of a comprehensive computational infrastructure for accelerator modeling and optimization**
- **RF cavity: Maxwell equations in electromagnetic field**
- **FEM in frequency domain leads to large sparse eigenvalue problem; needs to solve shifted linear systems**

(L.-Q. Lee)



RF unit in ILC





linear eigenvalue problem

$$(K_0 - \sigma^2 M_0)\, x = M_0\, b$$

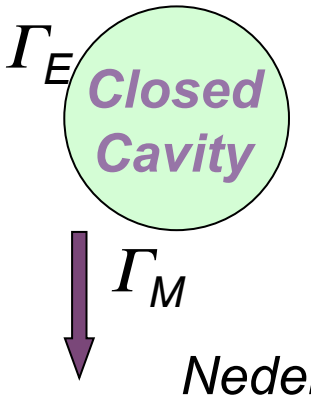
nonlinear complex eigenvalue problem

$$(K_0 + i\,\sigma W - \sigma^2 M_0)\, x = b$$

RF Cavity Eigenvalue Problem

Find frequency and field vector of normal modes:

"Maxwell's Equations"



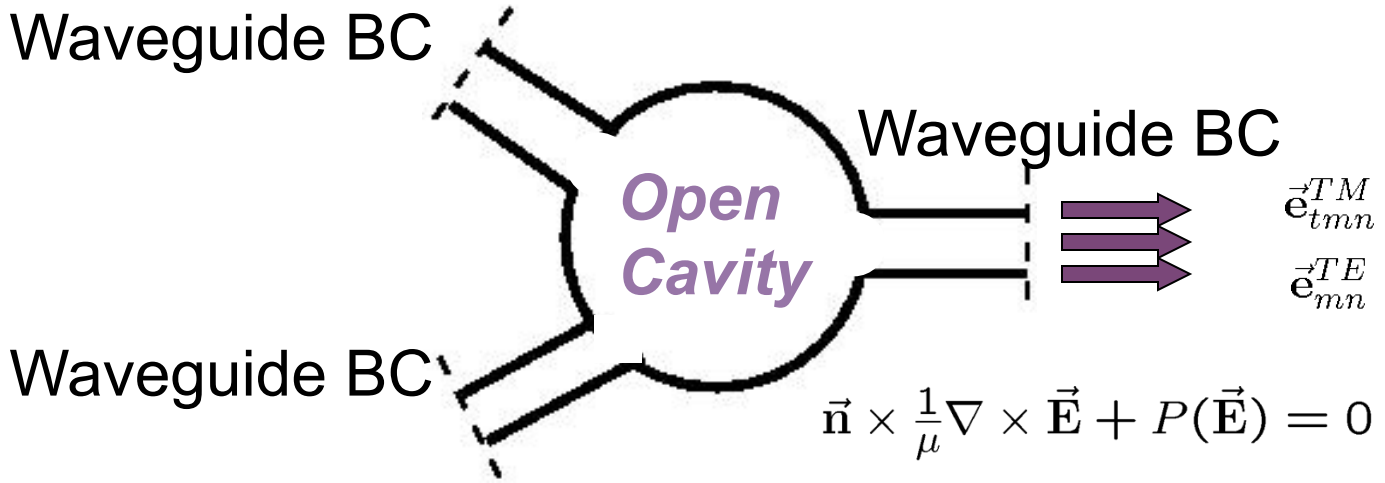
$$\nabla \times \left(\frac{1}{\mu}\nabla \times \vec{\mathbf{E}}\right) - k^2\epsilon\vec{\mathbf{E}} = 0 \ on \ \Omega$$
$$\vec{\mathbf{n}} \times \vec{\mathbf{E}} = 0 \ on \ \Gamma_E$$
$$\vec{\mathbf{n}} \times \frac{1}{\mu}\nabla \times \vec{\mathbf{E}} = 0 \ on \ \Gamma_M$$

Nedelec-type finite-element discretization $\mathbf{E} = \sum_i x_i \mathbf{N}_i$

$$\mathbf{K}\mathbf{x} = k^2\mathbf{M}\mathbf{x}$$

$$\mathbf{K}_{ij} = \int_\Omega (\nabla \times \mathbf{N}_i) \cdot \frac{1}{\mu}(\nabla \times \mathbf{N}_j)\, d\Omega$$
$$\mathbf{M}_{ij} = \int_\Omega \mathbf{N}_i \cdot \epsilon \mathbf{N}_j \, d\Omega$$

Cavity with Waveguide coupling for multiple waveguide modes



$$\vec{\mathbf{n}} \times \tfrac{1}{\mu}\nabla \times \vec{\mathbf{E}} + P(\vec{\mathbf{E}}) = 0 \text{ where}$$

$$P(\vec{\mathbf{E}}) = \sum_m^\infty \sum_n^\infty \frac{k^2}{i\sqrt{k^2-k_{mn}^2}} \vec{\mathbf{e}}_{tmn}^{TM} \int_\Gamma \vec{\mathbf{e}}_{tmn}^{TM} \cdot \vec{\mathbf{E}}\, d\Gamma - \sum_m^\infty \sum_n^\infty i\sqrt{k^2-k_{mn}^2}\, \vec{\mathbf{e}}_{mn}^{TE} \int_\Gamma \vec{\mathbf{e}}_{mn}^{TE} \cdot \vec{\mathbf{E}}\, d\Gamma$$

- Vector wave equation with waveguide boundary conditions can be modeled by a **non-linear complex eigenvalue problem**

$$\mathbf{K}x + i\sum_{m,n} \sqrt{k^2-k_{mn}^2}\, \mathbf{W}_{mn}^{TE} x + i\sum_{m,n} \frac{k^2}{\sqrt{k^2-k_{mn}^2}} \mathbf{W}_{mn}^{TM} x = k^2 \mathbf{M}x$$

where

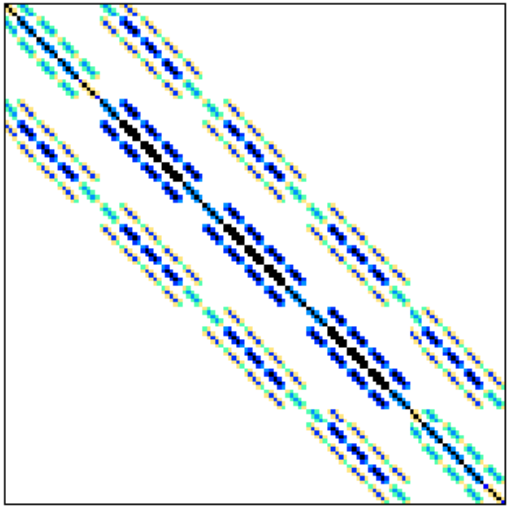
$$(\mathbf{W}_{mn}^{TE})_{ij} = \int_\Gamma \vec{\mathbf{e}}_{mn}^{TE} \cdot \mathbf{N}_i\, d\Gamma \int_\Gamma \vec{\mathbf{e}}_{mn}^{TE} \cdot \mathbf{N}_j\, d\Gamma$$

$$(\mathbf{W}_{mn}^{TM})_{ij} = \int_\Gamma \vec{\mathbf{e}}_{tmn}^{TM} \cdot \mathbf{N}_i\, d\Gamma \int_\Gamma \vec{\mathbf{e}}_{tmn}^{TM} \cdot \mathbf{N}_j\, d\Gamma$$

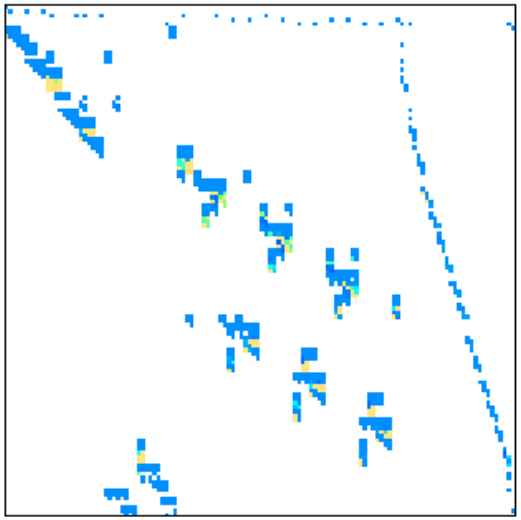
Sparse: lots of zeros in matrix

- 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.)



Sparse Storage Schemes

- Assume arbitrary sparsity pattern …
- Notation
  - N – dimension
  - NNZ – number of nonzeros

- Obvious:
  - "triplets" format ({i, j, val}) is not sufficient . . .
    - **Storage: 2*NNZ integers, NNZ reals**
    - **Not easy to randomly access one row or column**
  - Linked list format provides flexibility, but not friendly on modern architectures . . .
    - **Cannot call BLAS directly**

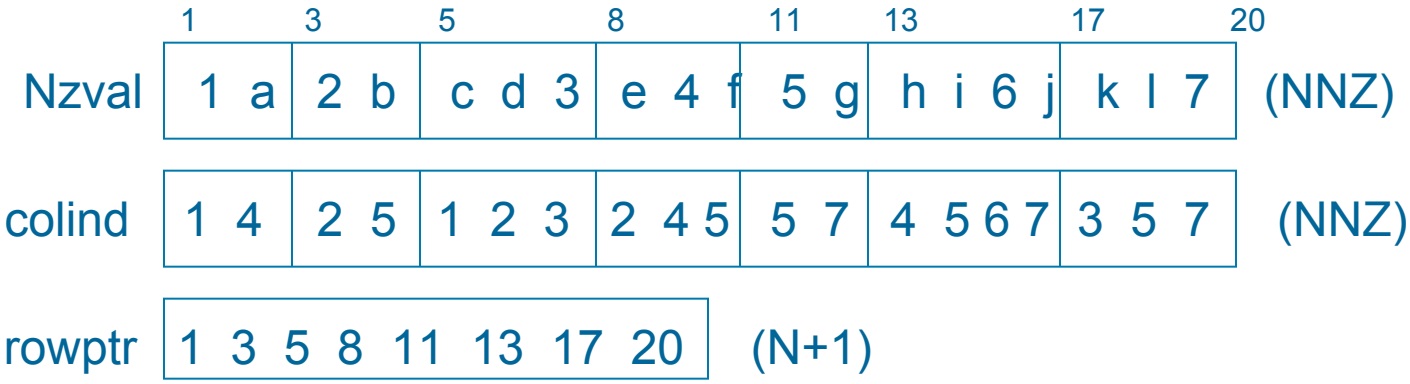
Compressed Row Storage (CRS)

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

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

	1		3		5		8		11		13		17		20	
Nzval	1	a	2	b	c	d	3	e	4	f	5	g	h	i	6	j	k	l	7	(NNZ)
colind	1	4	2	5	1	2	3	2	4	5	5	7	4	5	6	7	3	5	7	(NNZ)
rowptr	1	3	5	8	11	13	17	20												(N+1)

SpMV (y = Ax) with CRS




```
do  i = 1, N    . . .  row i of A
    sum = 0.0
    do  j = rowptr(i),  rowptr(i+1) – 1
        sum = sum + nzval(j) * x(colind(j))
    enddo
    y(i) = sum
enddo
```

- "dot product"
- No locality for x
- Vector length usually short
- Memory-bound: 3 reads, 2 flops

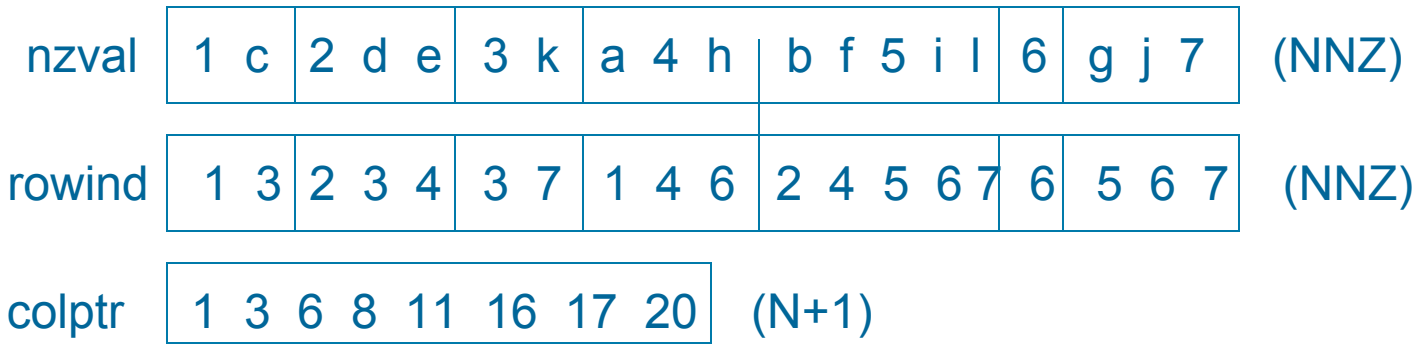
Compressed Column Storage (CCS)

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

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

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

SpMV (y = Ax) with CCS

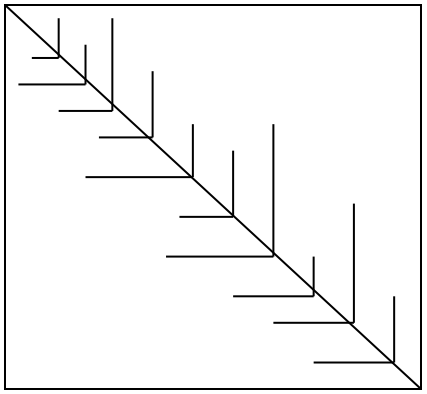



```
y(i) = 0.0,   i = 1…N
do  j = 1, N    . . .   column j of A
    t = x(j)
    do  i = colptr(j),  colptr(j+1) – 1
        y(rowind(i)) = y(rowind(i)) + nzval(i) * t
    enddo
enddo
```

- "SAXPY"
- No locality for y
- Vector length usually short
- Memory-bound: 3 reads, 1 write, 2 flops

Other Representations

- *"Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods", R. Barrett et al. (online)*
  - ELLPACK, segmented-sum, etc.
- "Block entry" formats (e.g., multiple degrees of freedom are associated with a single physical location)
  - Constant block size (BCRS)
  - Variable block sizes (VBCRS)
- Skyline (or profile) 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 zero) are stored



SpMV optimization – mitigate memory access bottleneck

BeBOP (Berkeley Benchmark and Optimization group):

http://bebop.cs.berkeley.edu

Software: OSKI / pOSKI – Optimized Sparse Kernel Interface

- Matrix reordering: up to 4x over CSR
- Register blocking: find dense blocks, pad zeros if needed, 2.1x over CSR
- Cache blocking: 2.8x over CSR
- Multiple vectors (SpMM): 7x over CSR
- Variable block splitting
- Symmetry: 2.8x over CSR
- …

Graphs

A graph G = (V, E) consists of a finite set V , called the vertex set and a finite, binary relation E on V , called the edge set.

Three standard graph models

- Undirected graph: The edges are unordered pair of vertices, i.e.

  $$\{u, v\} \in E, \text{ for some } u, v \in V$$

- Directed graph: The edges are ordered pair of vertices, that is, (u, v) and (v, u) are two different edges
- Bipartite graph: G = (U **U** V;E) consists of two disjoint vertex sets U and V such that for each edge $\{u, v\} \in E, u \in U \text{ and } v \in V$

An ordering or labelling of G = (V, E) having n vertices, i.e., |V| = n, is a mapping of V onto 1,2, …, n.

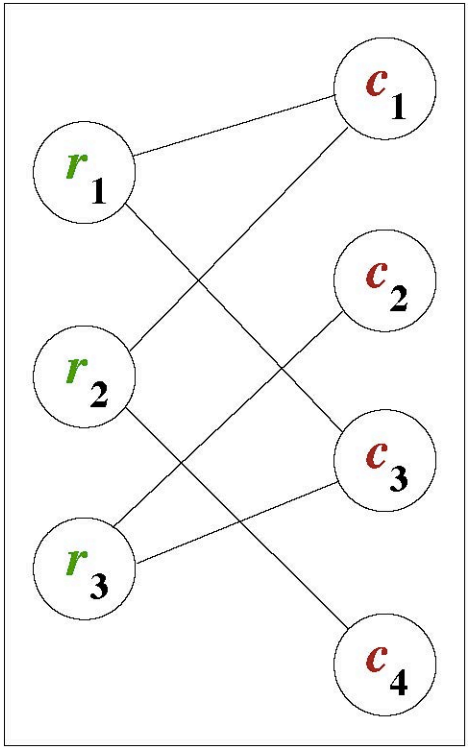
Graph for rectangular matrix

- Bipartite graph

  Rows = vertex set U, columns = vertex set V

  each nonzero A(i,j) = an edge (r_i,c_j), r_i in U and c_j in V

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} \bullet & & \bullet & \\ \bullet & & & \bullet \\ & \bullet & \bullet & \end{pmatrix} \end{array}$$

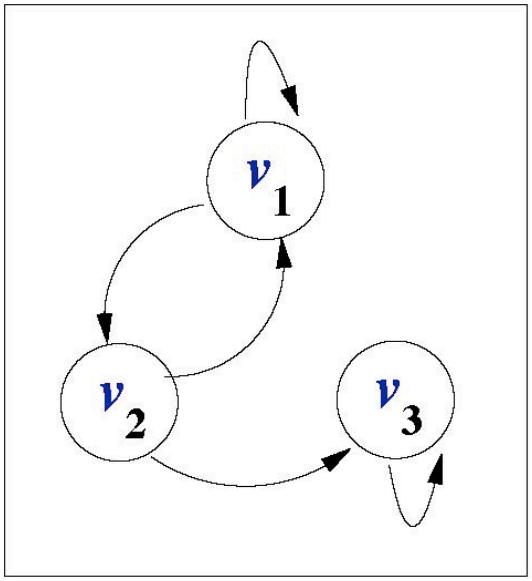


Graphs for square, pattern nonsymmetric matrix

- Bipartite graph as before

- Directed graph:
  Rows / columns = vertex set V
  each nonzero A(i,j) = an ordered edge (v_i, v_j) directed $v_i \rightarrow v_j$

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} \bullet & \bullet & \\ \bullet & & \bullet \\ & & \bullet \end{pmatrix} \end{array}$$

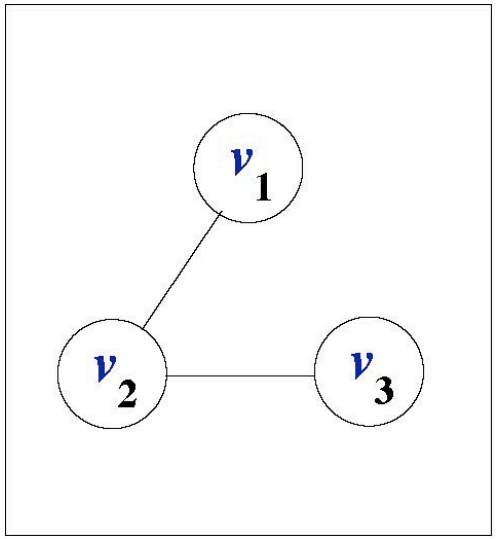


Graphs for square, pattern symmetric matrix

- Bipartite graph as before

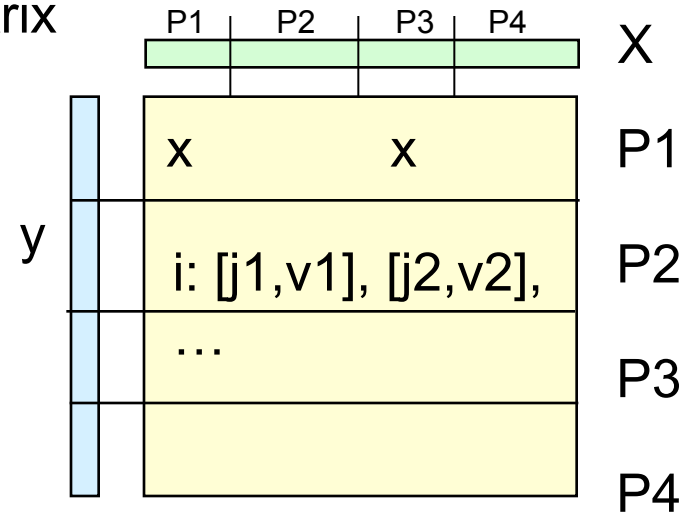
- Undirected graph:
  Rows / columns = vertex set V
  each nonzero A(i,j) = an edge $\{v_i, v_j\}$

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} & \bullet & \\ \bullet & \bullet & \bullet \\ & \bullet & \bullet \end{pmatrix} \end{array}$$



Parallel sparse matrix-vector multiply

- y = A*x, where A is a sparse n x n matrix

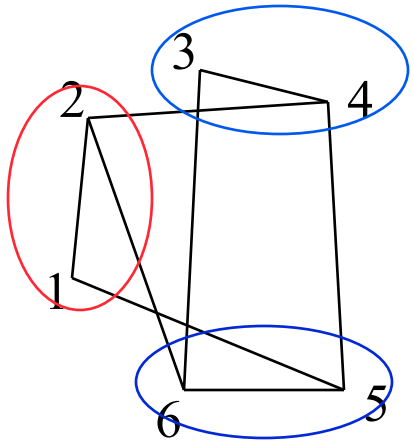


- Questions
  - which processors store
    - **y[i], x[i], and A[i,j]**
  - which processors compute
    - **y[i] = (row i of A) * x … a sparse dot product**
- Partitioning
  - Partition index set {1,…,n} = N1 ∪ N2 ∪ … ∪ Np.
  - For all i in Nk, Processor k stores y[i], x[i], and row i of A
  - For all i in Nk, Processor k computes y[i] = (row i of A) * x
  - "owner computes" rule: Processor k compute y[i]s it owns

May need communication

Graph partitioning and sparse matrices

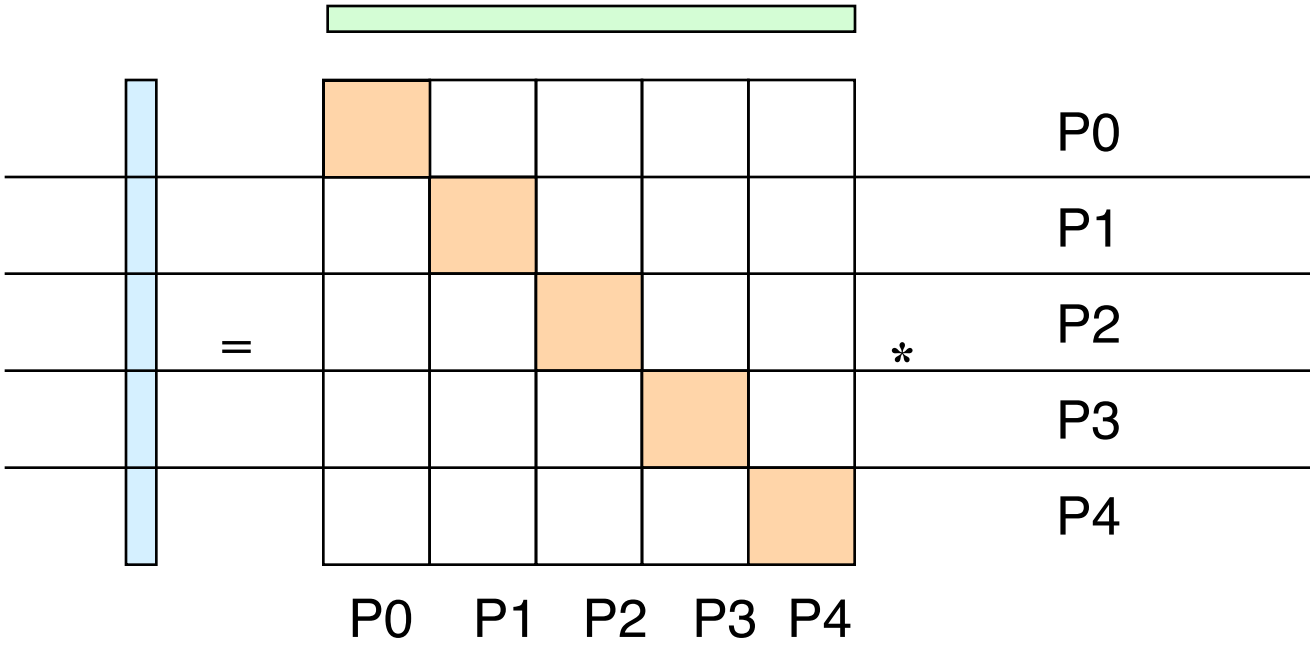
	1	2	3	4	5	6
1	1	1			1	
2	1	1		1		1
3			1	1		1
4		1	1	1	1	
5	1			1	1	1
6		1	1		1	1



- A "good" partition of the graph has
  - equal (weighted) number of nodes in each part (load and storage balance).
  - minimum number of edges crossing between (minimize communication).
- Reorder the rows/columns by putting all nodes in one partition together.

Matrix reordering via graph partitioning

- "Ideal" matrix structure for parallelism: block diagonal
  - p (number of processors) blocks, can all be computed locally.
  - If no non-zeros outside these blocks, no communication needed
- Can we reorder the rows/columns to get close to this?
  - Most nonzeros in diagonal blocks, very few outside



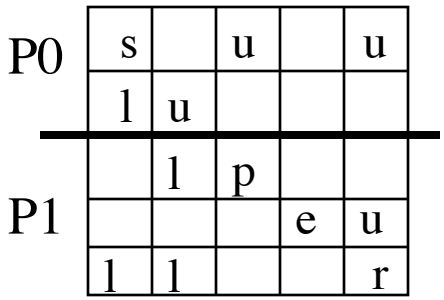
Distributed Compressed Row Storage

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

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

Distributed Compressed Row Storage

A is distributed on 2 cores:



- 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 }**

Sparse matrices in MATLAB

- In matlab, “A = sparse()”, create a sparse matrix A
  - Type “help sparse”, or “doc sparse”
- Storage: compressed column (CCS)
- Operation on sparse (full) matrices returns sparse (full) matrix
  operation on mixed sparse & full matrices returns full matrix
- Ordering: amd, symamd, symrcm, colamd
- Factorization: lu, chol, qr, …
- Utilities: spy

Summary

- Many representations of sparse matrices
  - Depending on application/algorithm needs
- Strong connection of sparse matrices and graphs
  - Many graph algorithms are applicable

References

- Barrett, et al., “Templates for the solution of linear systems: Building Blocks for Iterative Methods, 2nd Edition”, SIAM, 1994 (book online)
- Sparse BLAS standard: http://www.netlib.org/blas/blast-forum
- BeBOP: http://bebop.cs.berkeley.edu/
- J.R. Gilbert, C. Moler, R. Schreiber, “Sparse Matrices In MATLAB: Design And Implementation”, SIAM J. Matrix Anal. Appl, 13, 333-356, 1992.

Exercises

1. Write a program that converts a matrix in CCS format to CRS format, see code in sparse_CCS/ directory
2. Write a program to compute y = A^T*x without forming A^T
   - A can be stored in your favorite compressed format
3. Write a SpMV code with ELLPACK representation
4. SpMV roofline model on your machine
5. Write an OpenMP program for SpMV
6. Run the MPI SpMV code in the Hands-On-Exercises/ directory

EXTRA SLIDES

ELLPACK

- ELLPACK: software for solving elliptic problems [Purdue]
- Force all rows to have the same length as the longest row, then columns are stored contiguously

$$\begin{pmatrix} 1 & & & a & & & \\ & 2 & & & b & & \\ c & d & 3 & & & & \\ & e & & 4 & f & & \\ & & & & 5 & & g \\ & & & h & i & 6 & j \\ & & k & & l & & 7 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & a & 0 & 0 \\ 2 & b & 0 & 0 \\ c & d & 3 & 0 \\ e & 4 & f & 0 \\ 5 & g & 0 & 0 \\ h & i & 6 & j \\ k & l & 7 & 0 \end{pmatrix}$$

- 2 arrays: nzval(N,L) and colind(N,L), where L = max row length
  - **N*L reals, N*L integers**
- Usually L << N

SpMV with ELLPACK

```
y(i) = 0.0,  i = 1…N
do j = 1, L
    do i = 1, N
        y(i) = y(i) + nzval(i, j) * x(colind(i, j))
    enddo
enddo
```

$$\begin{pmatrix} 1 & a & 0 & 0 \\ 2 & b & 0 & 0 \\ c & d & 3 & 0 \\ e & 4 & f & 0 \\ 5 & g & 0 & 0 \\ h & i & 6 & j \\ k & l & 7 & 0 \end{pmatrix}$$

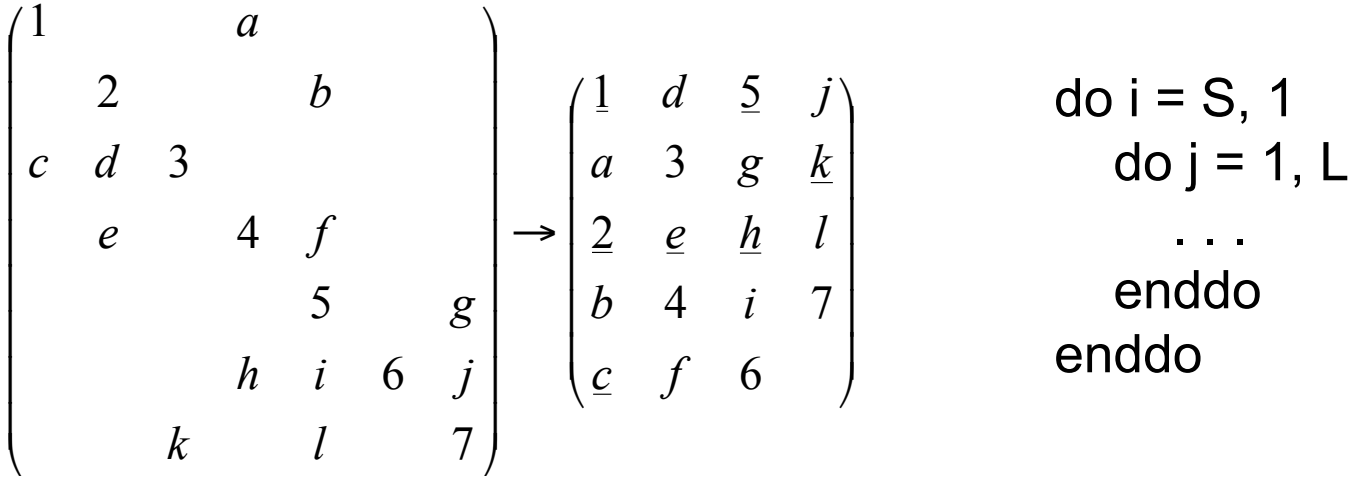
- Neither “dot” nor “SAXPY”
- Good for vector processor: long vector length (N)
- Extra memory, flops for padded zeros, especially bad if row lengths vary a lot

Segmented-Sum [Blelloch et al.]

- Data structure is an augmented form of CRS, computational structure is similar to ELLPACK
- Each row is treated as a ***segment*** in a long vector
- Underlined elements denote the beginning of each segment (i.e., a row in A)
- Dimension: S * L ~ NNZ, where L is chosen to approximate the hardware vector length

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

SpMV with Segmented-Sum



- 2 arrays: nzval(S, L) and colind(S, L), where S*L ~ NNZ
  - **NNZ reals, NNZ integers**
- SpMV is performed bottom-up, with each "row-sum" (dot) of Ax stored in the beginning of each segment
  - **Similar to ELLPACK, but with more control logic in inner-loop**
- Good for vector processors

4th Gene Golub SIAM Summer School, 7/22 – 8/7, 2013, Shanghai

Lecture 3

Sparse Direct Method: Combinatorics

Xiaoye Sherry Li
Lawrence Berkeley National Laboratory, USA
xsli@lbl.gov

crd-legacy.lbl.gov/~xiaoye/G2S3/

Lecture outline

- Linear solvers: direct, iterative, hybrid
- Gaussian elimination
- Sparse Gaussian elimination: elimination graph, elimination tree
- Symbolic factorization, ordering, graph traversal
  - only integers, no FP involved

Strategies of sparse linear solvers

- Solving a system of linear equations Ax = b
  - **Sparse: many zeros in A; worth special treatment**
- Iterative methods (CG, GMRES, …)
  - **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 (factorization) to precondition iterative method
  - Solve an easy system: $M^{-1}Ax = M^{-1}b$

Gaussian Elimination (GE)

- Solving a system of linear equations Ax = b

- First step of GE

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

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

- Repeat GE on C
- Result 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

Numerical Stability: Need for Pivoting

- One step of GE:

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

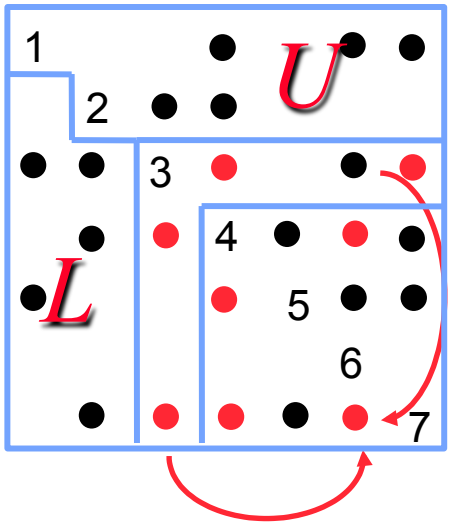
- $C = B - \frac{v \cdot w^T}{\alpha}$
  - **If α small, some entries in B may be lost from addition**

- Pivoting: swap the current diagonal with a larger entry from the other part of the matrix

- Goal: control element growth (pivot growth) in L & U

Sparse GE

- Goal: Store only nonzeros and perform operations only on nonzeros
- Scalar algorithm: 3 nested loops
  - Can re-arrange loops to get different variants: left-looking, right-looking, . . .
- Fill: new nonzeros in factor




```
for i = 1 to n
    A(:,j) = A(:,j) / A(j,j)   % cdiv(j) col_scale
    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)
```

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

Useful tool to discover fill: Reachable Set

- Given certain elimination order (x_1, x_2, . . ., x_n), how do you determine the fill-ins using original graph of A ?
  - **An implicit elimination model**
- Definition: Let S be a subset of the node set. The reachable set of y through S is:

  Reach(y, S) = { x | there exists a directed path (y,v_1,…v_k, x), v_i in S}

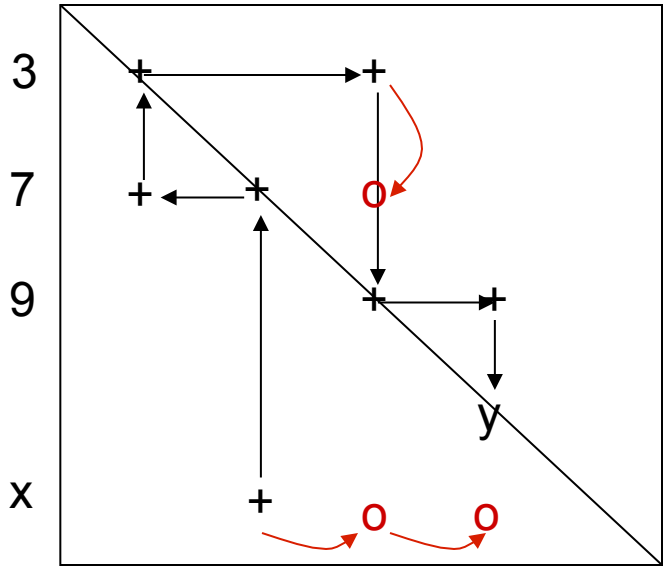
- "Fill-path theorem" [Rose/Tarjan ' 78] (general case):

  Let G(A) = (V,E) be a directed graph of A, then an edge (v,w) exists in the filled graph $G^+(A)$ if and only if

$$w \in \mathrm{Re}\,ach(v, \{v_1,\ldots v_k\}), \quad \text{where}, \; v_i < \min(v,w), \quad 1 \le i \le k$$

  - **$G^+(A)$ = graph of the {L,U} factors**

Concept of reachable set, fill-path



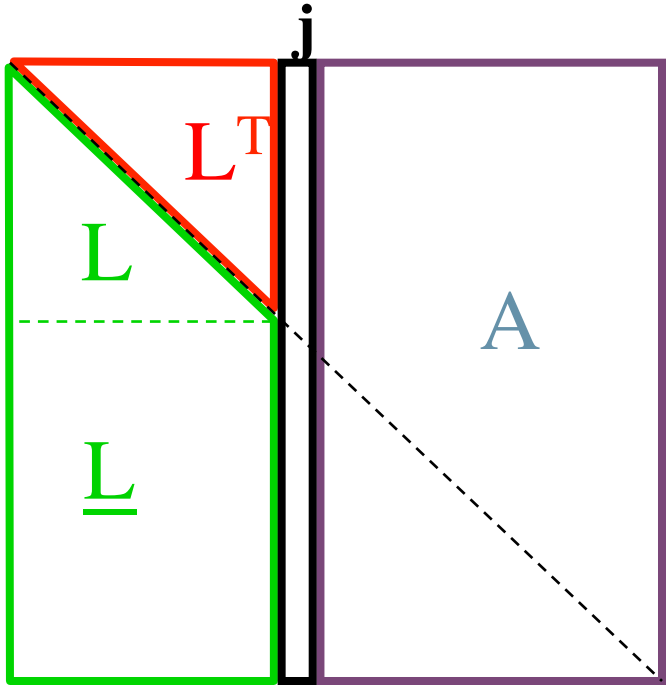
Edge (x,y) exists in filled graph G^+ due to the path: x → 7 → 3 → 9 → y

- Finding fill-ins ←→ finding transitive closure of G(A)

Sparse Column Cholesky Factorization LL^T

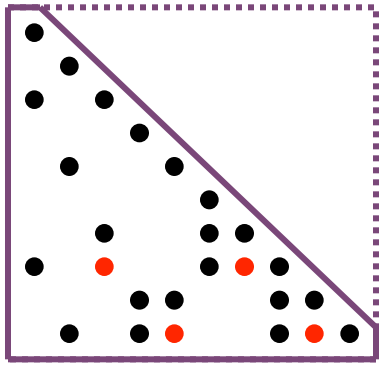
```
for j = 1 : n
   L(j:n, j) = A(j:n, j);
   for k < j with L(j, k) nonzero
      % sparse cmod(j,k)
      L(j:n, j) = L(j:n, j) – L(j, k) * L(j:n, k);
   end;
   % sparse cdiv(j)
   L(j, j) = sqrt(L(j, j));
   L(j+1:n, j) = L(j+1:n, j) / L(j, j);
end;
```

Column j of A becomes column j of L

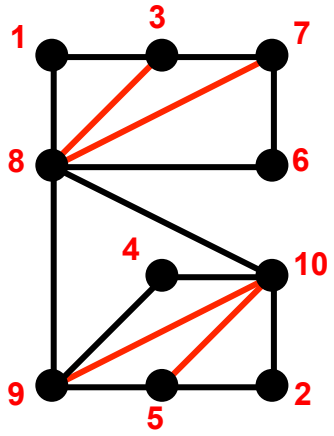


- Fill-path theorem [George ’80] (symmetric case)
  After $x_1, \ldots, x_i$ are eliminated, the set of nodes adjacent to y in the elimination graph is given by Reach(y, $\{x_1, \ldots, x_i\}$), x_i<min(x,y)

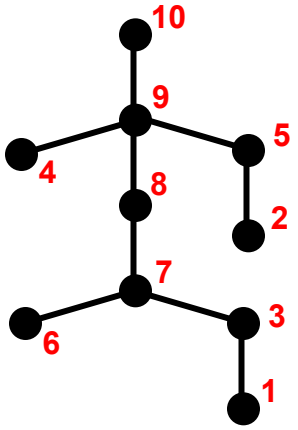
Elimination Tree



Cholesky factor L



$G^+(A)$



T(A)

$T(A)$: parent(j) = min { i > j : (i, j) in $G^+(A)$ }

parent(col j) = first nonzero row below diagonal in L

- T describes dependencies among columns of factor
- Can compute $G^+(A)$ easily from T
- Can compute T from G(A) in almost linear time

Symbolic Factorization

precursor to numerical factorization

- Elimination tree
- Nonzero counts
- Supernodes
- Nonzero structure of {L, U}

- Cholesky [Davis'06 book, George/Liu'81 book]
  - **Use elimination graph of L and its transitive reduction (elimination tree)**
  - **Complexity linear in output: O(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 nnz(L+U), but much smaller than flops(LU)**

Can we reduce fill?

- Reordering, permutation

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 2 & & & \\ 3 & & 3 & & \\ 4 & & & 4 & \\ 5 & & & & 5 \end{pmatrix}$$ (all filled after elimination)

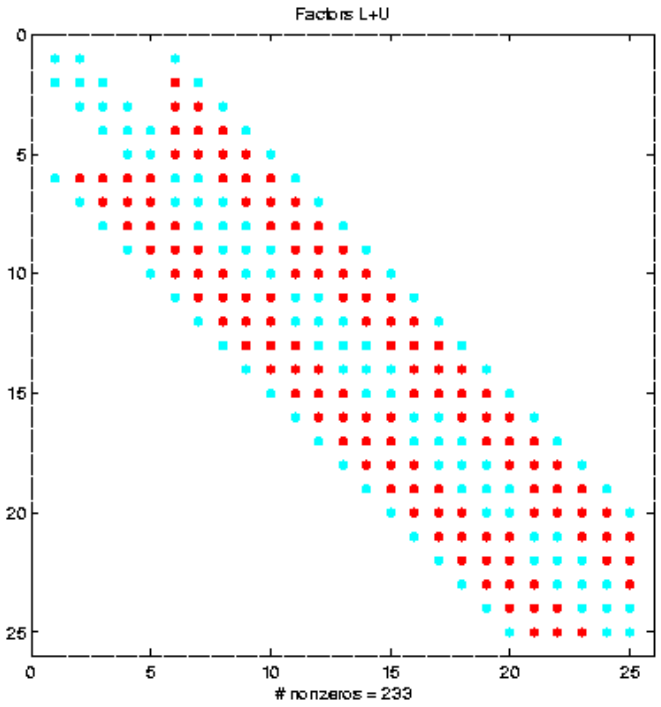
$$\Rightarrow \begin{pmatrix} & & & & 1 \\ & & & 1 & \\ & & 1 & & \\ & 1 & & & \\ 1 & & & & \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 2 & & & \\ 3 & & 3 & & \\ 4 & & & 4 & \\ 5 & & & & 5 \end{pmatrix} \begin{pmatrix} & & & & 1 \\ & & & 1 & \\ & & 1 & & \\ & 1 & & & \\ 1 & & & & \end{pmatrix} = \begin{pmatrix} 5 & & & & 5 \\ & 4 & & & 4 \\ & & 3 & & 3 \\ & & & 2 & 2 \\ 5 & 4 & 3 & 2 & 1 \end{pmatrix}$$

(no fill after elimination)

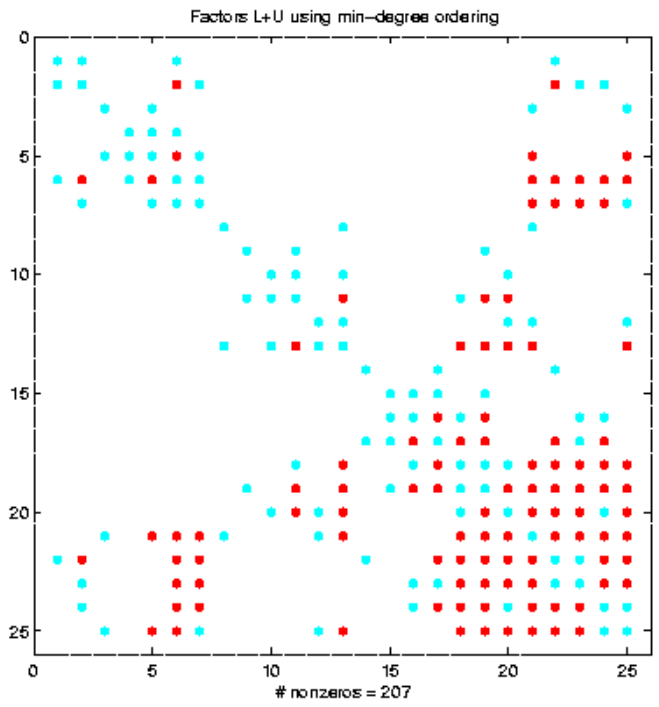
Fill-in in Sparse GE

- Original zero entry A_{ij} becomes nonzero in L or U
  - Red: fill-ins

Natural order: NNZ = 233

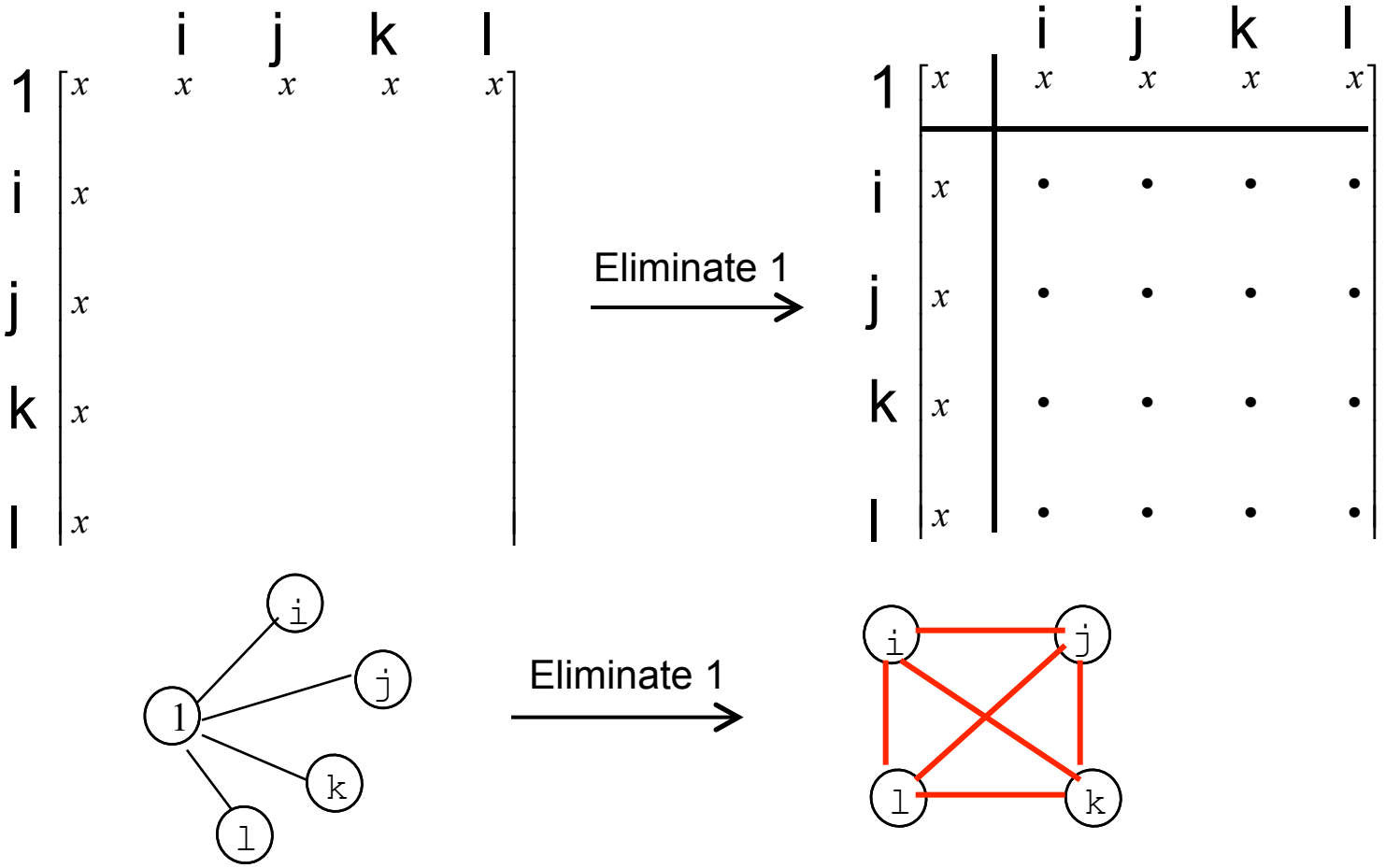


Min. Degree order: NNZ = 207



Ordering : Minimum Degree (1/3)

Graph game:



Maximum fill: all the edges between neighboring vertices (“clique”)

Minimum Degree Ordering (2/3)

- Greedy approach: do the best locally
  - Best for modest size problems
  - Hard to parallelize
- At each step
  - Eliminate the vertex with the smallest degree
  - Update degrees of the neighbors
- Straightforward implementation is slow and requires too much memory
  - Newly added edges are more than eliminated vertices

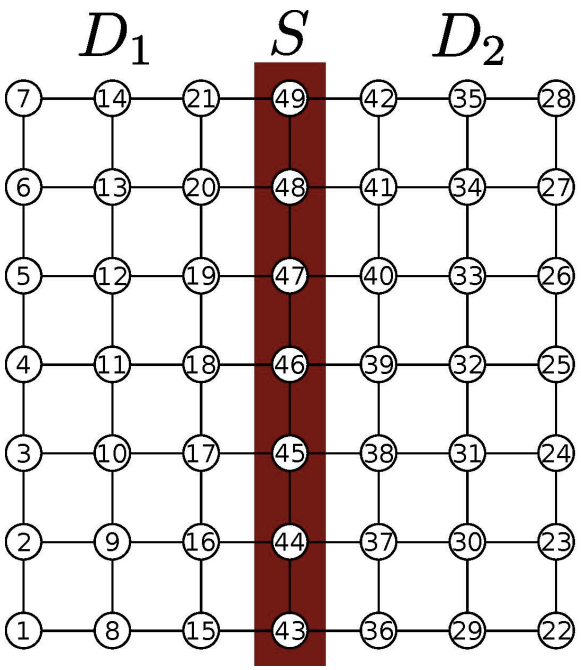
Minimum Degree Ordering (3/3)

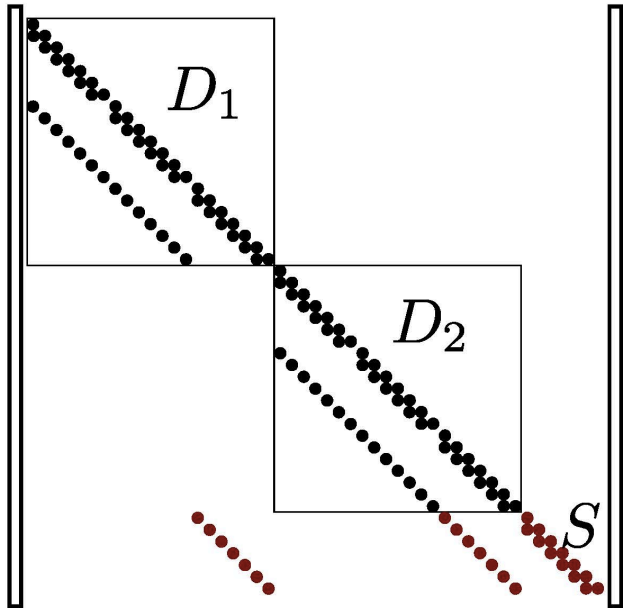
- Use quotient graph (QG) as a compact representation [George/Liu ’78]
- Collection of cliques resulting from the eliminated vertices affects the degree of an uneliminated vertex
- Represent each connected component in the eliminated subgraph by a single “supervertex”
- Storage required to implement 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., . .
- Extended the QG model to nonsymmetric using bipartite graph [Amestoy/Li/Ng `07]

Ordering : Nested Dissection

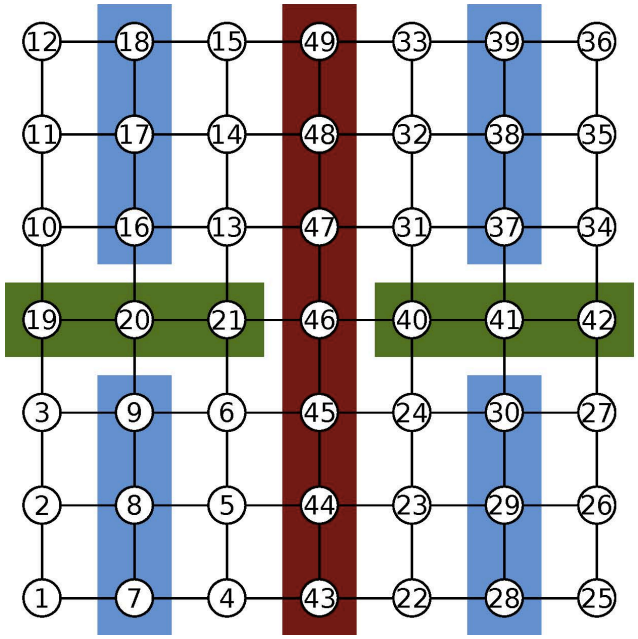
- Model problem: discretized system Ax = b from certain PDEs, e.g., 5-point stencil on k x k grid, n = k^2
- Recall fill-path theorem:

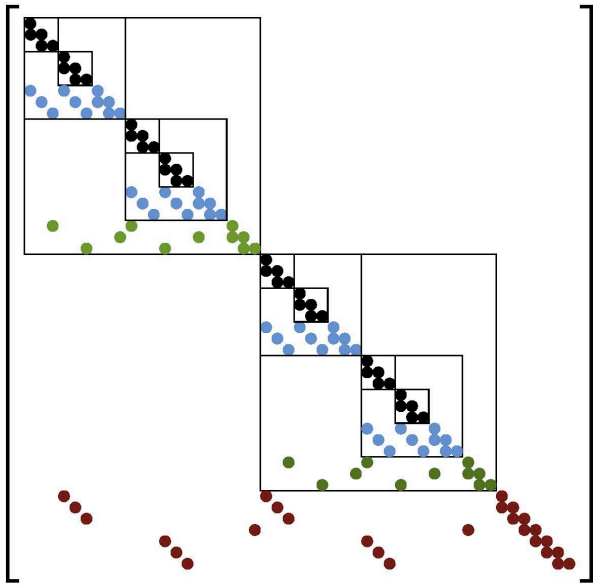
  After x_1, …, x_i are eliminated, the set of nodes adjacent to y in the elimination graph is given by Reach(y, {x_1, …, x_i}), x_i<min(x,y)



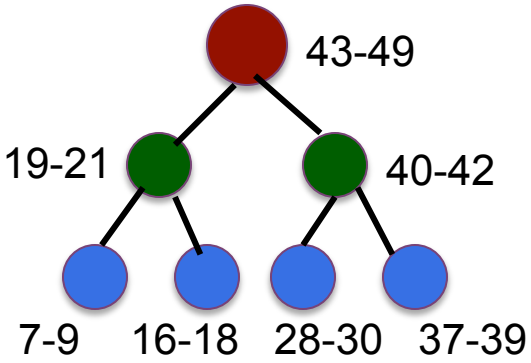


ND ordering: recursive application of bisection





- ND gives a separator tree (i.e elimination tree)



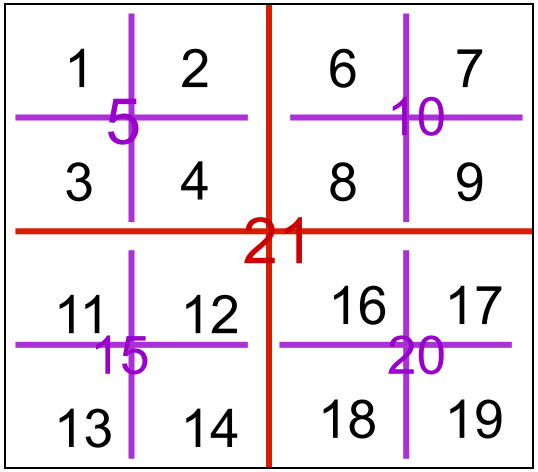
ND analysis on a square grid (k x k = n)

- Theorem [George '73, Hoffman/Martin/Ross]: ND ordering gave optimal complexity in exact factorization.

  Proof:
  - **Apply ND by a sequence of "+" separators**
  - **By "reachable set" argument, all the separators are essentially dense submatrices**
  - **Fill-in estimation: add up the nonzeros in the separators**

$$k^2 + 4(k/2)^2 + 4^2(k/4)^2 + \cdots = O(k^2 \log_2 k) = O(n \log_2 n)$$

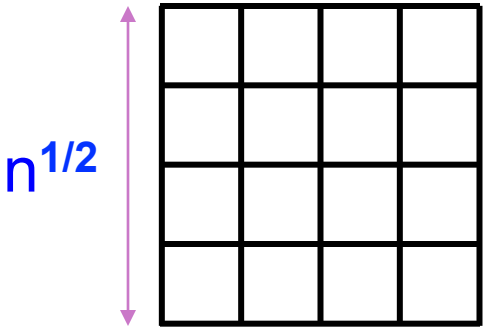
(more precisely: $31/4\,(k^2 \log_2 k) + O(k^2)$)

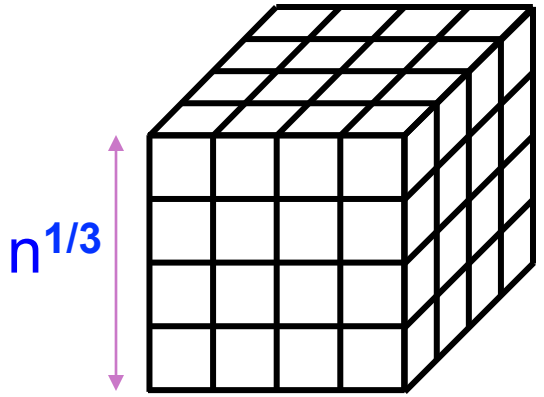


Similarly: Operation count: $O(k^3) = O(n^{3/2})$

Complexity of direct methods

Time and space to solve any problem on any well-shaped finite element mesh

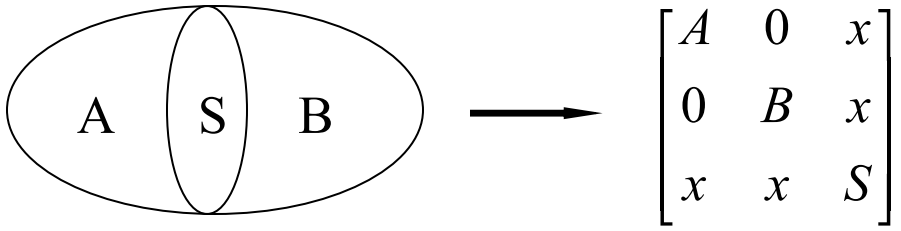




	2D	3D
Space (fill):	$O(n \log n)$	$O(n^{4/3})$
Time (flops):	$O(n^{3/2})$	$O(n^{2})$

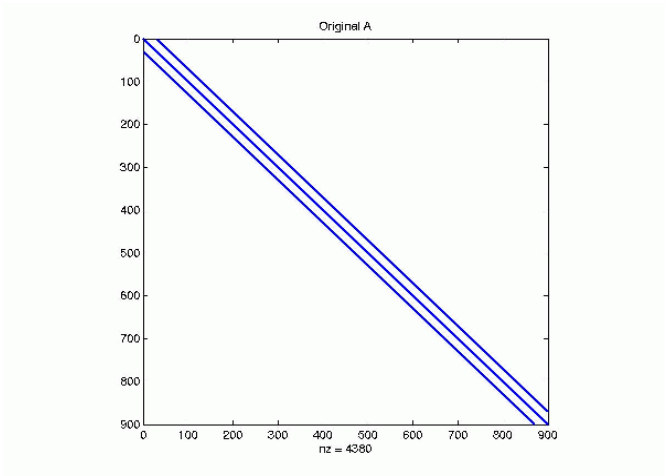
ND Ordering: generalization

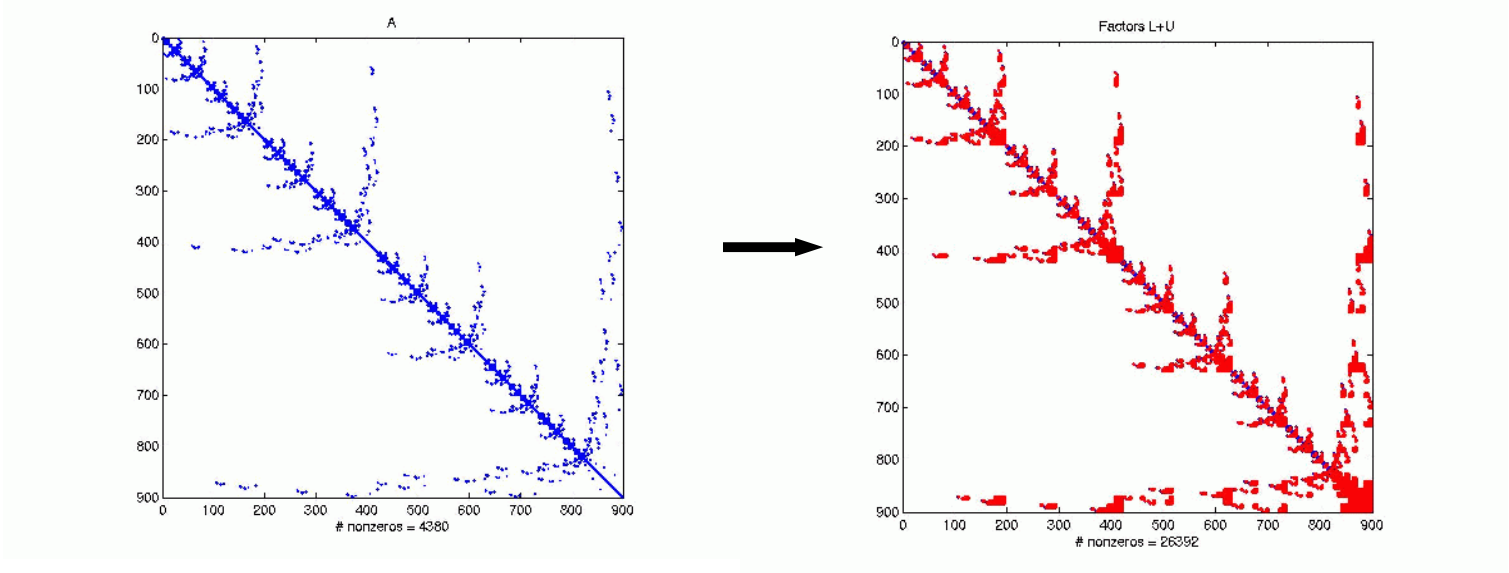
- Generalized nested dissection [Lipton/Rose/Tarjan ’79]
  - **Global graph partitioning: top-down, divide-and-conqure**
  - **First level**



  - **Recurse on A and B**
- Goal: find the smallest possible separator S at each level
  - **Multilevel schemes:**
    - **(Par)Metis [Karypis/Kumar `95], Chaco [Hendrickson/ Leland `94], (PT-)Scotch [Pellegrini et al.`07]**
  - **Spectral bisection [Simon et al. `90-`95]**
  - **Geometric and spectral bisection [Chan/Gilbert/Teng `94]**

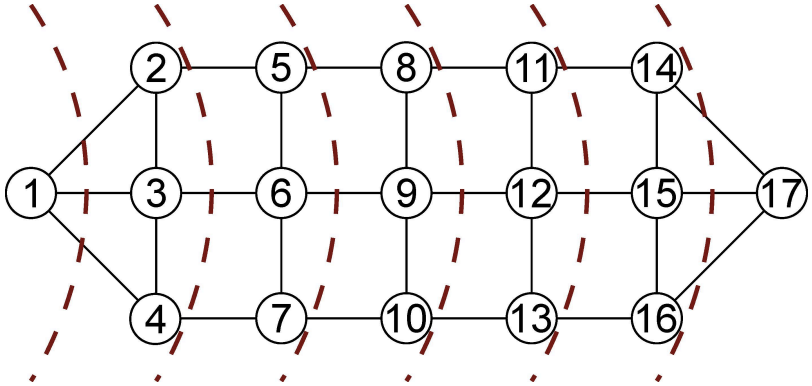
ND Ordering


Original A
nz = 4380

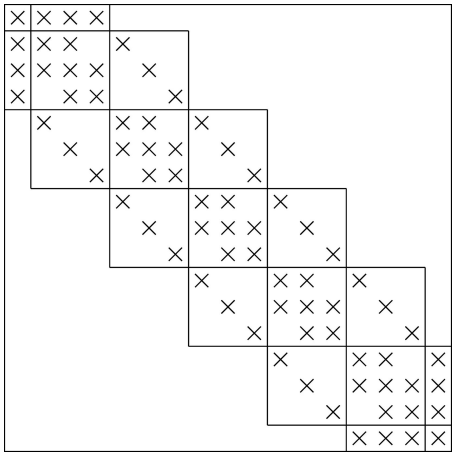

A
nonzeros = 4380
Factors L+U
nonzeros = 26392

CM / RCM Ordering

- Cuthill-McKee, Reverse Cuthill-McKee
- Reduce bandwidth
  - Construct level sets via breadth-first search, start from the vertex of minimum degree
  - At any level, priority is given to a vertex with smaller number of neighbors



[Duff, Erisman, Reid]

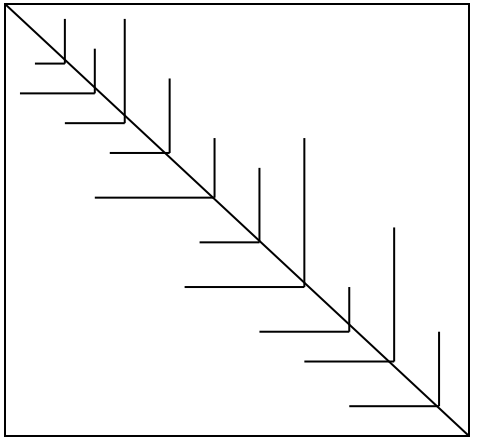


- RCM: Simply reverse the ordering found by CM

RCM good for envelop (profile) Solver
(also good for SpMV)

- Define bandwidth for each row or column
  - Data structure a little more sophisticated than band solver, but simpler than general sparse 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 the fill is contained inside the profile



- A good ordering would be based on bandwidth reduction
  - E.g., Reverse Cuthill-McKee

Envelop (profile) solver (2/2)

- Lemma: env(L+U) = env(A)
  - **No more fill-ins generated outside the envelop!**

Inductive proof: After N-1 steps,

$$A = \begin{pmatrix} A_1 & w \\ v^T & s \end{pmatrix} = \begin{pmatrix} L_1 & \\ v_1^T & 1 \end{pmatrix} \begin{pmatrix} U_1 & w_1 \\ & t \end{pmatrix}, \quad s.t.\ A_1 = L_1 U_1$$

Then,

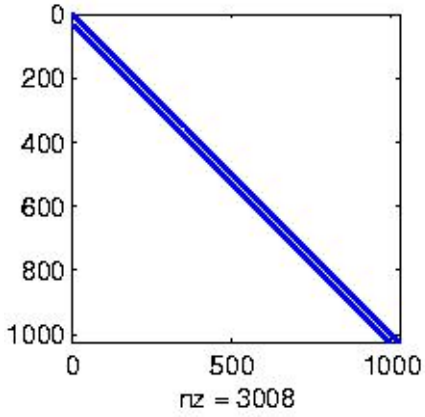
solve $L_1\ w_1 = w$, first nonzero position of w_1 is the same as w

solve $U_1^T\ v_1 = v$, first nonzero position of v_1 is the same as v

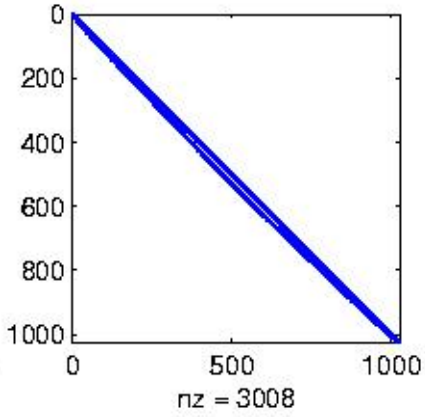
Envelop vs. general solvers

- Example: 3 orderings (natural, RCM, MD)
- Envelop size = sum of bandwidths

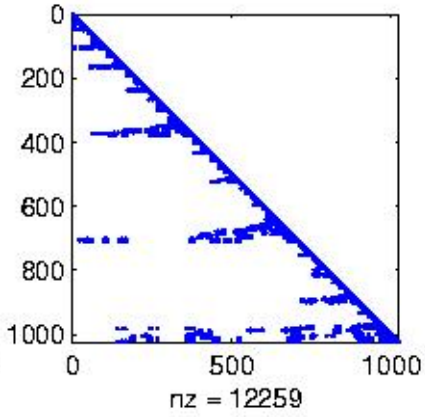
Env = 31775



Env = 22320



Env = 61066
NNZ(L, MD) = 12259



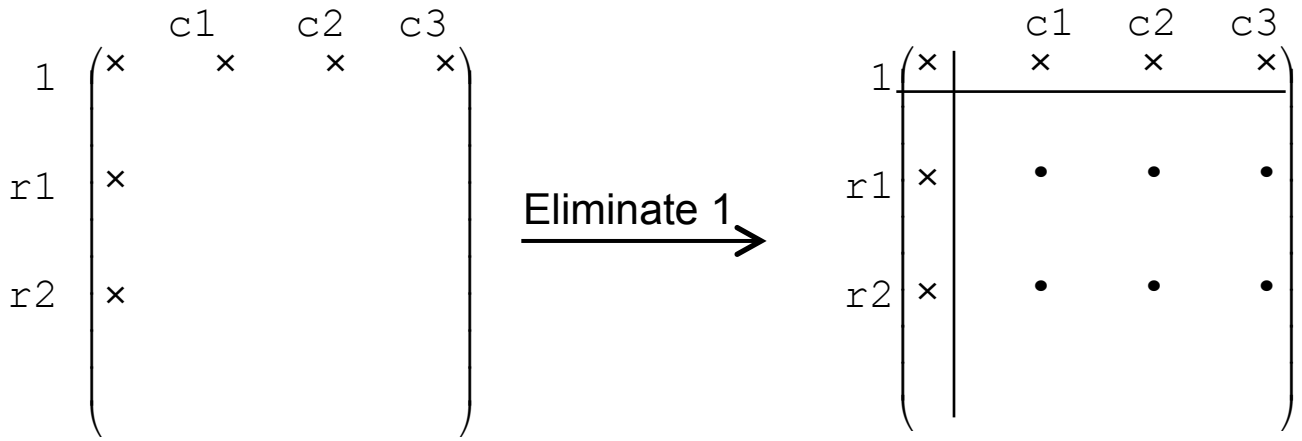
Ordering for unsymmetric LU – symmetrization

- Can use a symmetric ordering on a symmetrized matrix . . .

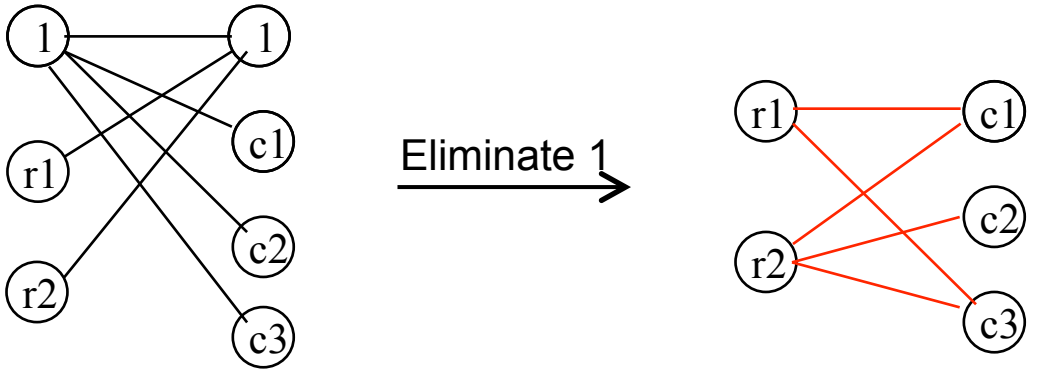
- Case of partial pivoting (sequential SuperLU):
  Use ordering based on A^TA
  - If $R^TR = A^TA$ and $PA = LU$, then for any row permutation P, struct(L+U) $\subseteq$ struct(R^T+R) [George/Ng `87]
  - Making R sparse tends to make L & U sparse . . .

- Case of diagonal pivoting (static pivoting in SuperLU_DIST):
  Use ordering based on A^T+A
  - If $R^TR = A^T+A$ and $A = LU$, then struct(L+U) $\subseteq$ struct(R^T+R)
  - Making R sparse tends to make L & U sparse . . .

Unsymmetric variant of "Min Degree" ordering
(Markowitz scheme)



- Bipartite graph
- After a vertex is eliminated, all the row & column vertices adjacent to it become fully connected – "bi-clique" (assuming diagonal pivot)
- The edges of the bi-clique are the potential fill-ins (upper bound !)



Results of Markowitz ordering [Amestoy/Li/Ng'02]

- Extend the QG model to bipartite quotient graph
- Same asymptotic complexity as symmetric MD
  - **Space is bounded by 2*(m + n)**
  - **Time is bounded by O(n * m)**

- For 50+ unsym. matrices, compared with MD on A' +A:
  - **Reduction in fill:** average 0.88, best 0.38
  - **Reduction in FP operations:** average 0.77, best 0.01

- How about graph partitioning for unsymmetric LU?
  - **Hypergraph partition [Boman, Grigori, et al. `08]**
  - **Similar to ND on A^TA, but no need to compute A^TA**

Remark: Dense vs. Sparse GE

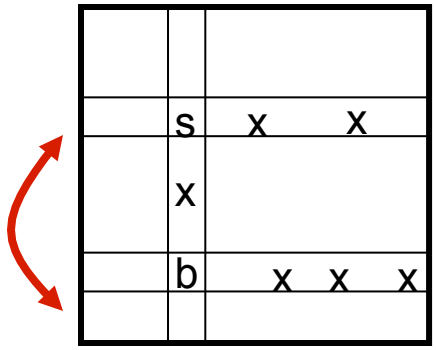
- Dense GE: $P_r A P_c = LU$
  - P_r and P_c are permutations chosen to maintain stability
  - Partial pivoting suffices in most cases : $P_r A = LU$

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

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 $\sqrt{\varepsilon}\|A\|$, without changing data structures for L & U**
  - **If needed, use a few steps of iterative refinement after the first solution**

  ➔ **quite stable in practice**

Use many heuristics

- Finding an optimal fill-reducing ordering is NP-complete → use heuristics:
  - Local approach: Minimum degree
  - Global approach: Nested dissection (optimal in special case), RCM
  - Hybrid: First permute the matrix globally to confine the fill-in, then reorder small parts using local heuristics
    - **Local methods effective for smaller graph, global methods effective for larger graph**

- Numerical pivoting: trade-off stability with sparsity and parallelism
  - Partial pivoting too restrictive
  - Threshold pivoting
  - Static pivoting
  - …

Algorithmic phases in sparse GE

1. 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)

2. Predict the fill-in positions in L & U
   - 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
   - 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 my be interleaved.

References

- T. Davis, Direct Methods for Sparse Linear Systems, SIAM, 2006. (book)
- A. George and J. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, 1981. (book)
- I. Duff, I. Erisman and J. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, 1986. (book)
- C. Chevalier, F. Pellegrini, "PT-Scotch", Parallel Computing, 34(6-8), 318-331, 2008. http://www.labri.fr/perso/pelegrin/scotch/
- G. Karypis, K. Schloegel, V. Kumar, ParMetis: Parallel Graph Partitioning and Sparse Matrix Ordering Library", University of Minnesota. http://www-users.cs.umn.edu/~karypis/metis/parmetis/
- E. Boman, K. Devine, et al., "Zoltan, Parallel Partitioning, Load Balancing and Data-Management Services", Sandia National Laboratories.

  http://www.cs.sandia.gov/Zoltan/
- J. Gilbert, "Predicting structures in sparse matrix computations", SIAM. J. Matrix Anal. & App, 15(1), 62–79, 1994.
- J.W.H. Liu, "Modification of the minimum degree algorithm by multiple elimination", ACM Trans. Math. Software, Vol. 11, 141-153, 1985.
- T.A. Davis, J.R. Gilbert, S. Larimore, E. Ng, "A column approximate minimum degree ordering algorithm", ACM Trans. Math. Software, 30 (3), 353-376, 2004
- P. Amestoy, X.S. Li and E. Ng, "Diagonal Markowitz Scheme with Local Symmetrization", SIAM J. Matrix Anal. Appl., Vol. 29, No. 1, pp. 228-244, 2007.

Exercises

- Homework3 in Hands-On-Exercises/ directory
- Show that:
  If $R^TR = A^T+A$ and $A = LU$, then struct$(L+U) \subseteq$ struct(R^T+R)
- Show that: [George/Ng `87]
  If $R^TR = A^TA$ and $PA = LU$, then for any row permutation P,
  struct$(L+U) \subseteq$ struct(R^T+R)

4th Gene Golub SIAM Summer School, 7/22 – 8/7, 2013, Shanghai

Lecture 4

Sparse Factorization: Data-flow Organization

Xiaoye Sherry Li
Lawrence Berkeley National Laboratory, USA
xsli@lbl.gov

crd-legacy.lbl.gov/~xiaoye/G2S3/

Lecture outline

- Dataflow organization: left-looking, right-looking
- Blocking for high performance
  - Supernode, multifrontal
- Triangular solution

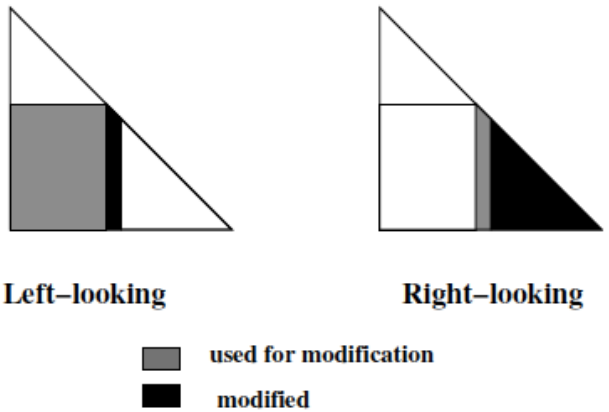
Dense Cholesky

- Left-looking Cholesky

for k = 1,…,n do
  for i = k,…,n do
    for j = 1,…k-1 do
      $a_{ik}^{(k)} = a_{ik}^{(k)} - l_{ij} \cdot l_{kj}$
    end for
  end for
  $l_{kk} = \sqrt{a_{kk}^{(k-1)}}$
  for i = k+1,…,n do
    $l_{ik} = a_{ik}^{(k-1)} / l_{kk}$
  end for
end for

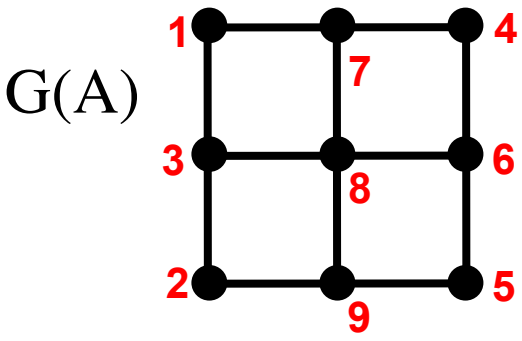
- Right-looking Cholesky

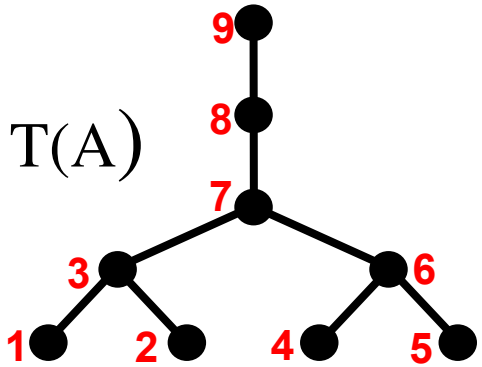
for k = 1,…,n do
  $l_{kk} = \sqrt{a_{kk}^{(k-1)}}$
  for i = k+1,…,n do
    $l_{ik} = a_{ik}^{(k-1)} / l_{kk}$
    for j = k+1,…,i do
      $a_{ij}^{(k)} = a_{ij}^{(k)} - l_{ik} \cdot l_{jk}$
    end for
  end for
end for

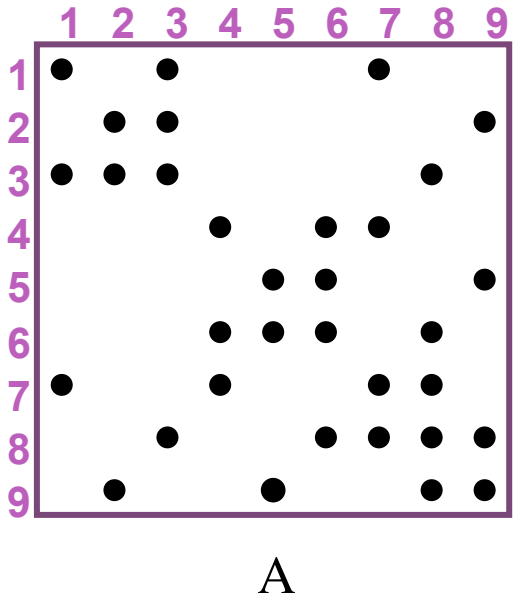


Sparse Cholesky

- Reference case: regular 3 x 3 grid ordered by nested dissection. Nodes in the separators are ordered last



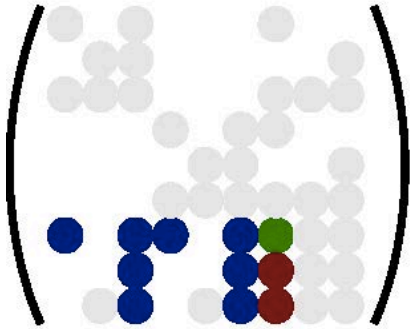


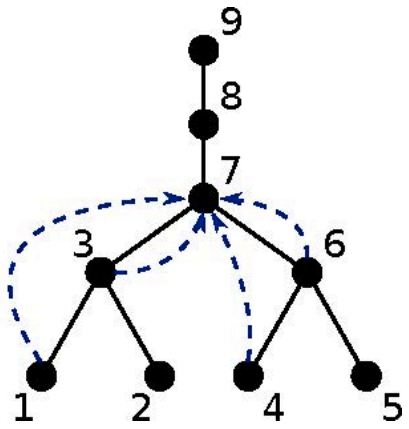


- Notation:
  - cdiv(j) : divide column j by a scalar
  - cmod(j, k) : update column j with column k
  - struct(L(1:k), j)) : the structure of L(1:k, j) submatrix

Sparse left-looking Cholesky

```
for j = 1 to n do
    for k in struct(L(j, 1 : j-1)) do
        cmod(j, k)
    end for
    cdiv(j)
end for
```



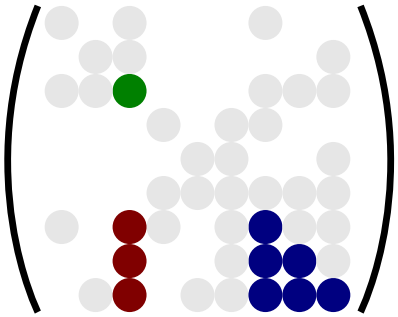


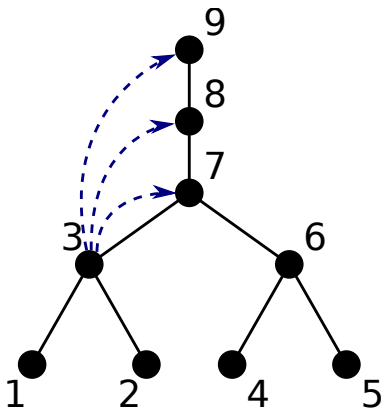
Before variable j is eliminated, column j is updated with all the columns that have a nonzero on row j. In the example above, struct(L(7,1:6)) = **{1**; **3**; **4**; **6}**.

- This corresponds to receiving updates from nodes lower in the subtree rooted at j
- The filled graph is necessary to determine the structure of each row

Sparse right-looking Cholesky

```
for k = 1 to n do
    cdiv(k)
    for j in struct(L(k+1 : n, k)) do
        cmod(j,k)
    end for
end for
```





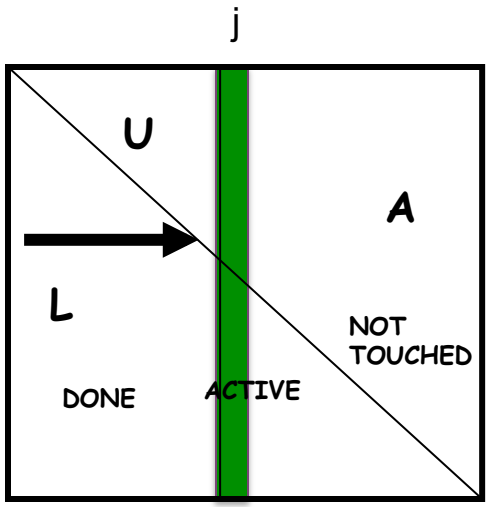
After variable k is eliminated, column k is used to update all the columns corresponding to nonzeros in column k. In the example above, struct(L(4:9,3))={**7**; **8**; **9**}.

- This corresponds to sending updates to nodes higher in the elimination tree
- The filled graph is necessary to determine the structure of each column

Sparse LU

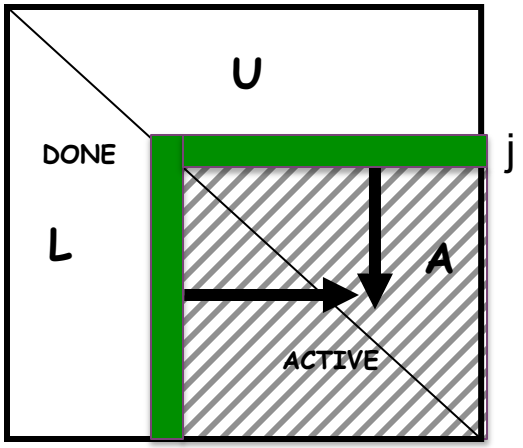
- Left-looking: many more reads than writes

```
U(1:j-1, j) = L(1:j-1, 1:j-1) \ A(1:j-1, j)
for j = 1 to n do
    for k in struct(U(1:j-1, j)) do
        cmod(j, k)
    end for
    cdiv(j)
end for
```



- Right-looking: many more writes than reads

```
for k = 1 to n do
    cdiv(k)
    for j in struct(U(k, k+1:n)) do
        cmod(j, k)
    end for
end for
```



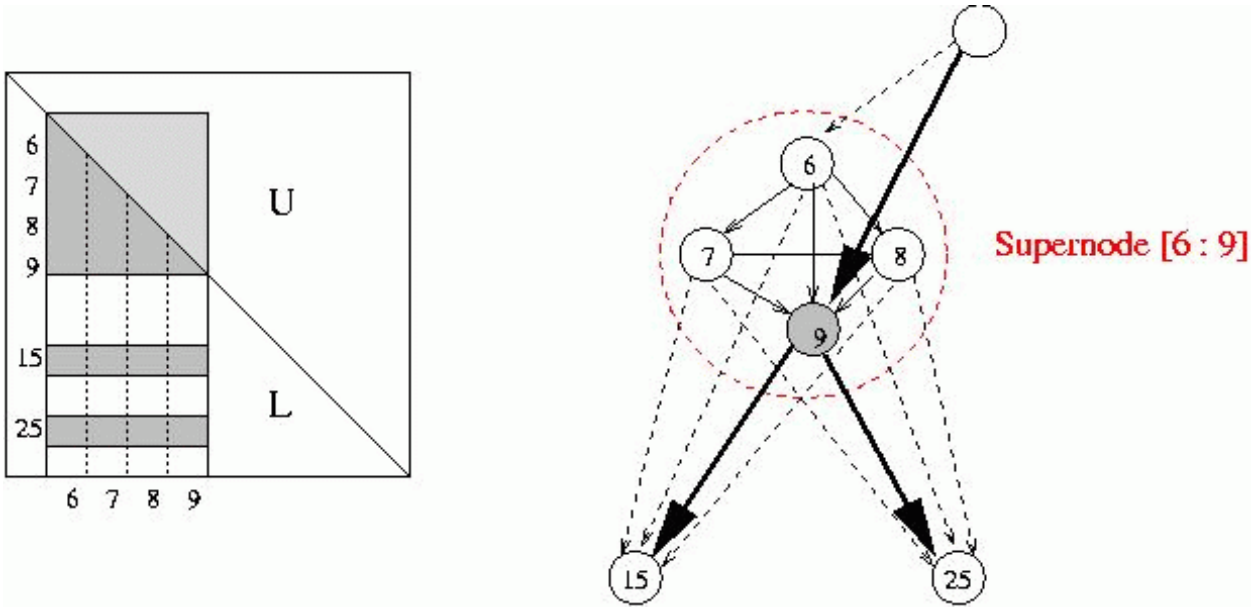
High Performance Issues: Reduce Cost of Memory Access & Communication

- Blocking to increase number of floating-point operations performed for each memory access
- 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

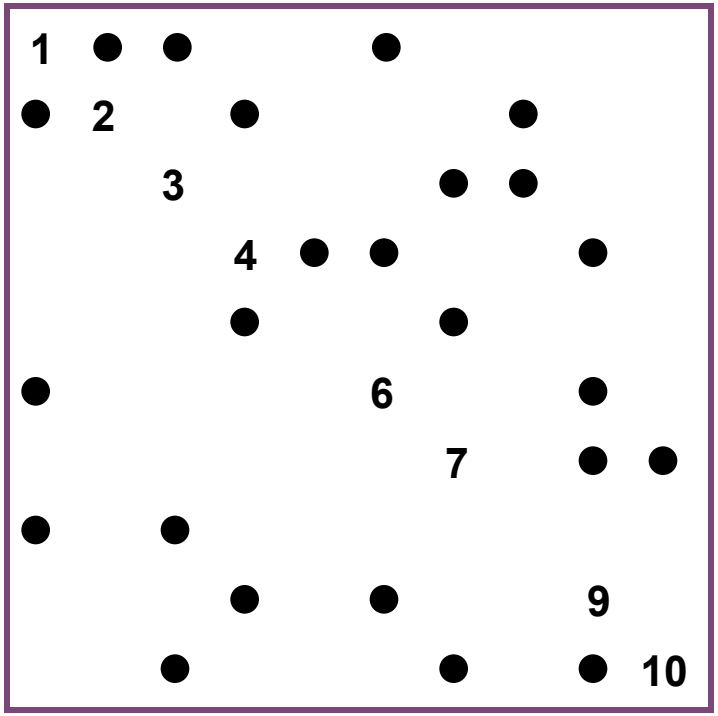
Blocking: supernode

- Use (blocked) CRS or CCS, and any ordering method
  - Leave room for fill-ins ! (symbolic factorization)

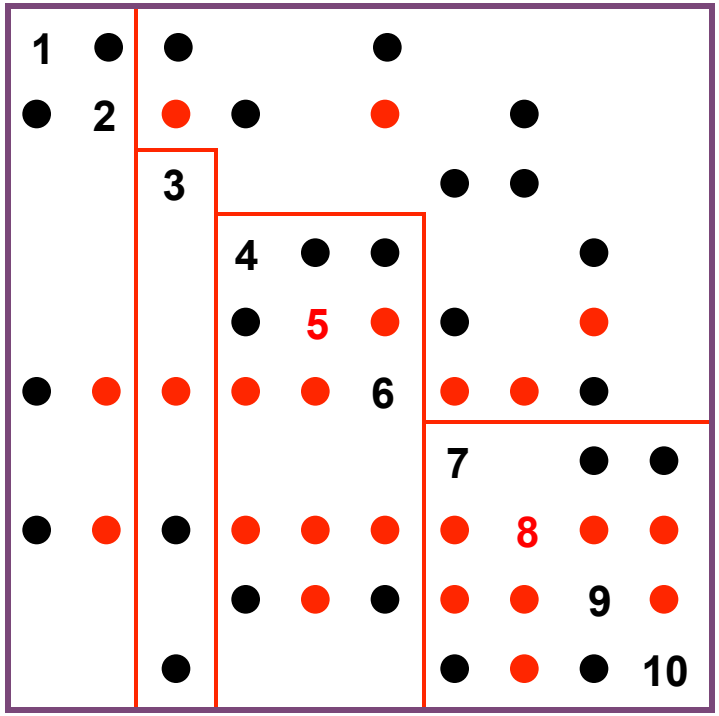
- Exploit "supernodal" (dense) structures in the factors
  - Can use Level 3 BLAS
  - Reduce inefficient indirect addressing (scatter/gather)
  - Reduce graph traversal time using a coarser graph



Nonsymmetric supernodes

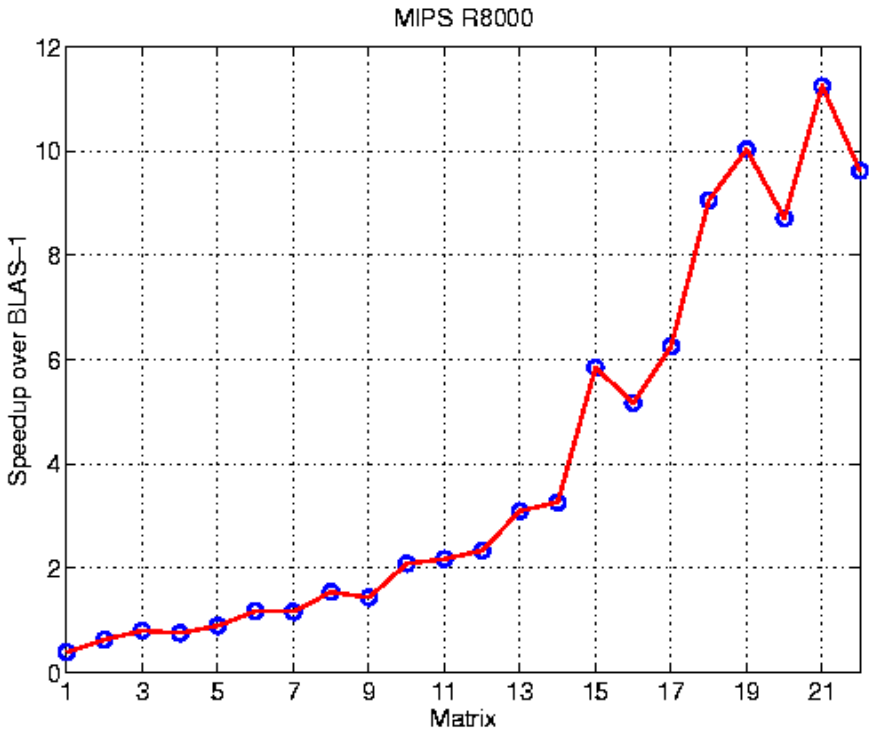


Original matrix A



Factors L+U

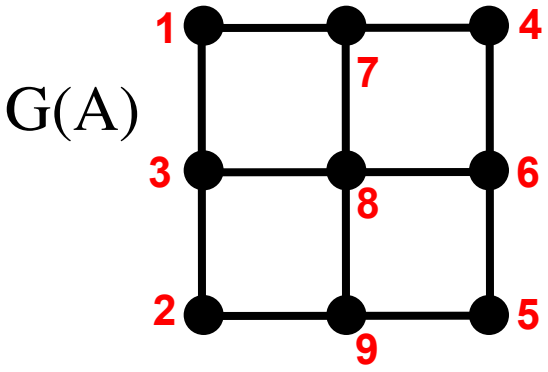
SuperLU speedup over unblocked code

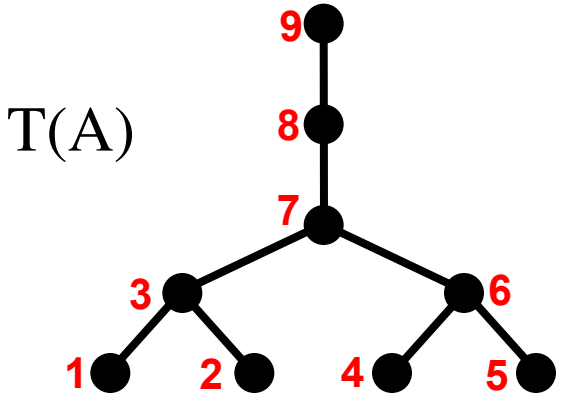


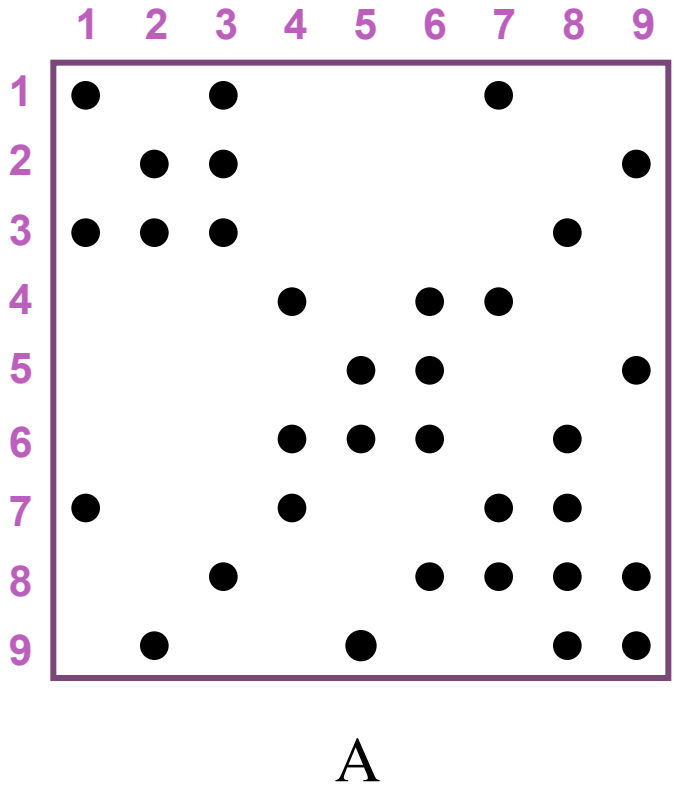
- Sorted in increasing "reuse ratio" = #Flops/nonzeros
  - ~ Arithmetic Intensity
- Up to 40% of machine peak on large sparse matrices on IBM RS6000/590, MIPS R8000

Symmetric-pattern multifrontal factorization

[John Gilbert's lecture]

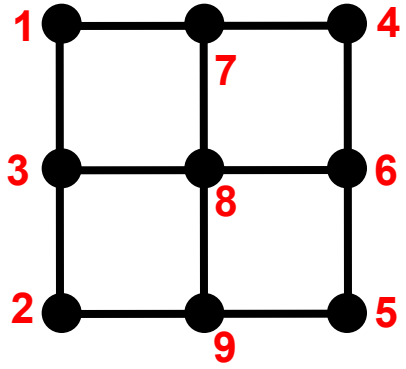




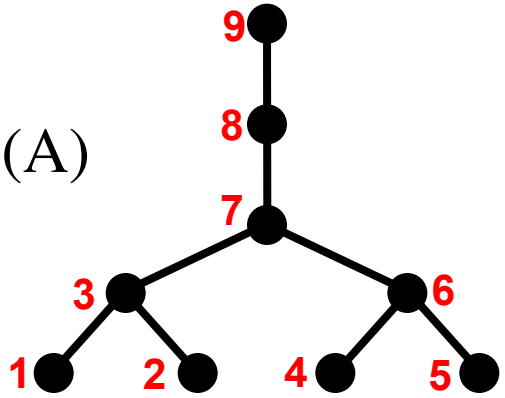


Symmetric-pattern multifrontal factorization

G(A)



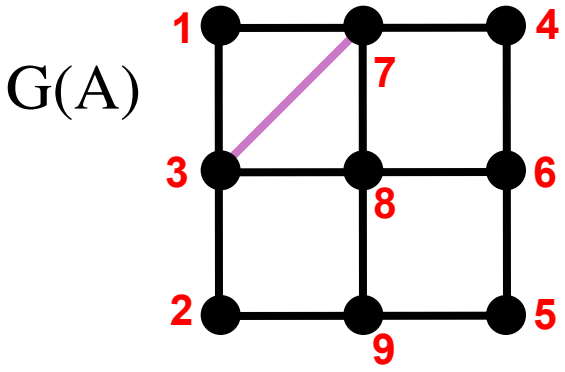
T(A)



For each node of T from leaves to root:

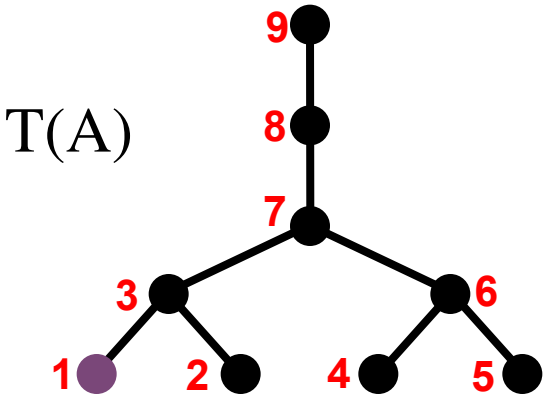
- Sum own row/col of A with children's *Update* matrices into *Frontal* matrix
- Eliminate current variable from *Frontal* matrix, to get *Update* matrix
- Pass *Update* matrix to parent

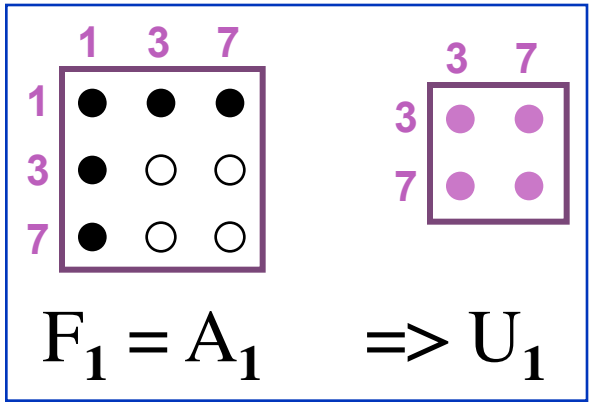
Symmetric-pattern multifrontal factorization



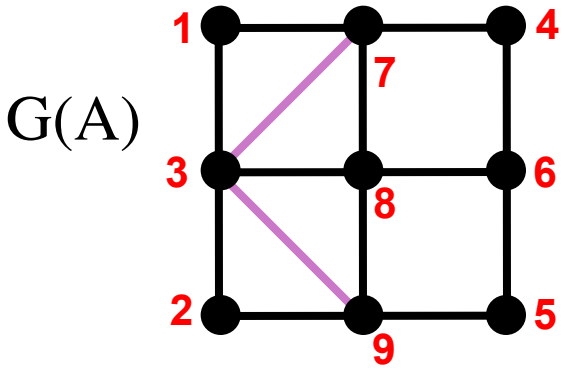
For each node of T from leaves to root:

- Sum own row/col of A with children's *Update* matrices into *Frontal* matrix
- Eliminate current variable from *Frontal* matrix, to get *Update* matrix
- Pass *Update* matrix to parent



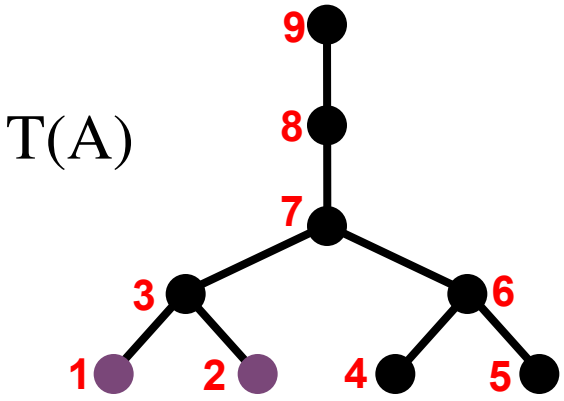


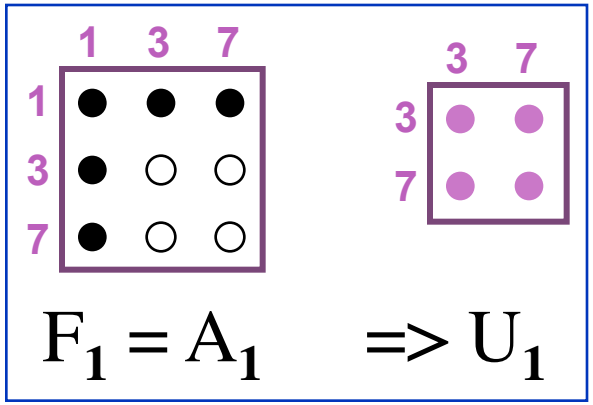
Symmetric-pattern multifrontal factorization

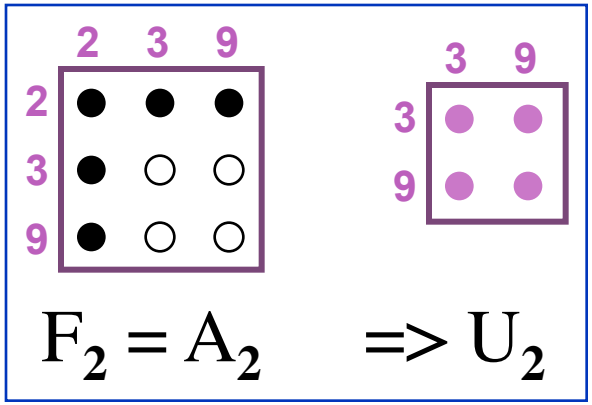


For each node of T from leaves to root:

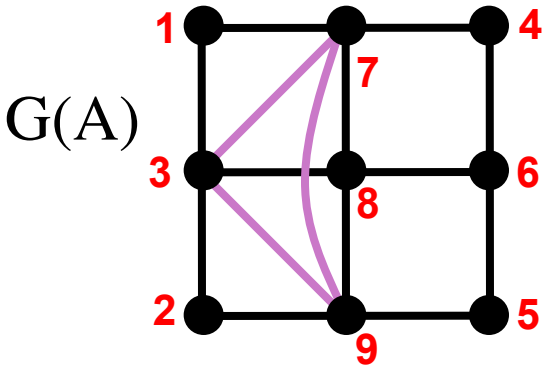
- Sum own row/col of A with children's *Update* matrices into *Frontal* matrix
- Eliminate current variable from *Frontal* matrix, to get *Update* matrix
- Pass *Update* matrix to parent

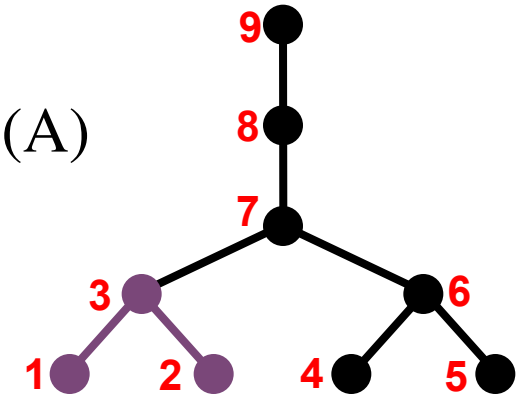


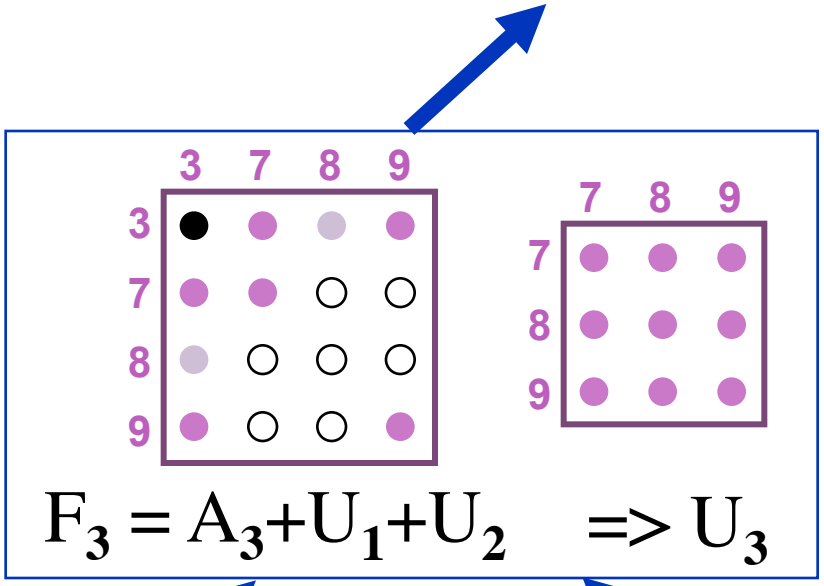


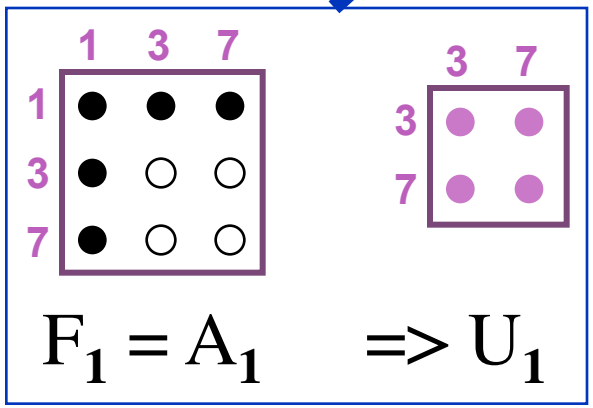


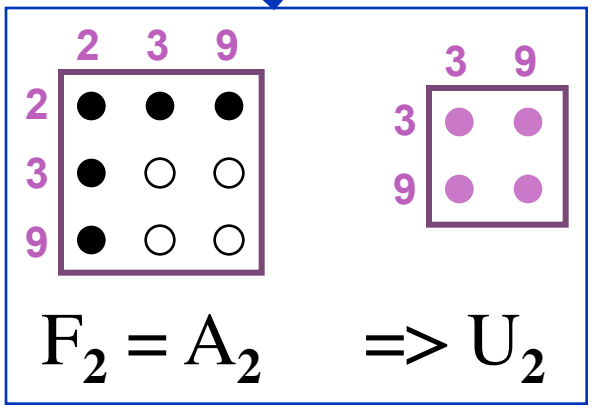
Symmetric-pattern multifrontal factorization



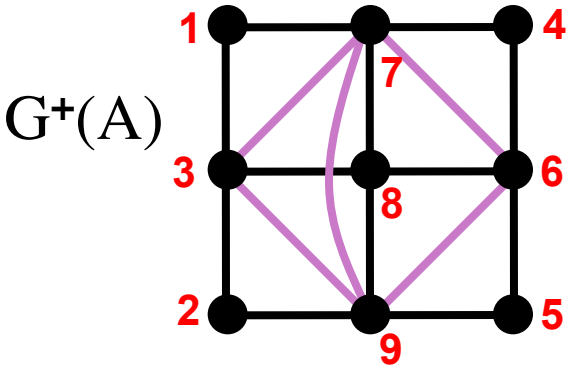


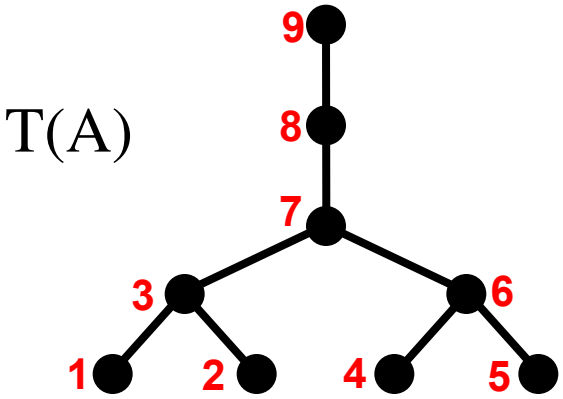


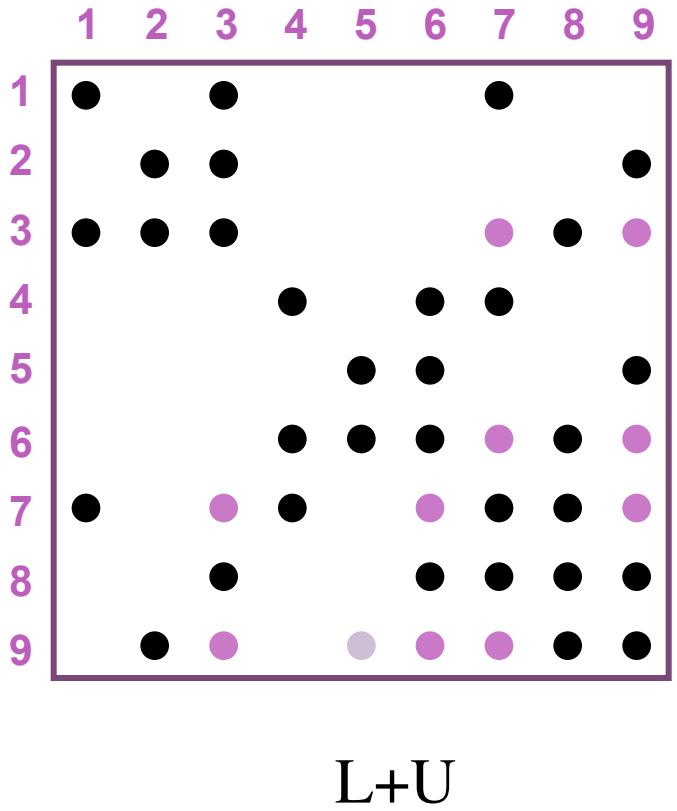




Symmetric-pattern multifrontal factorization

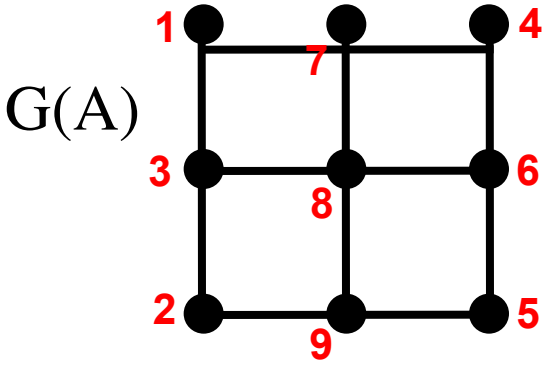


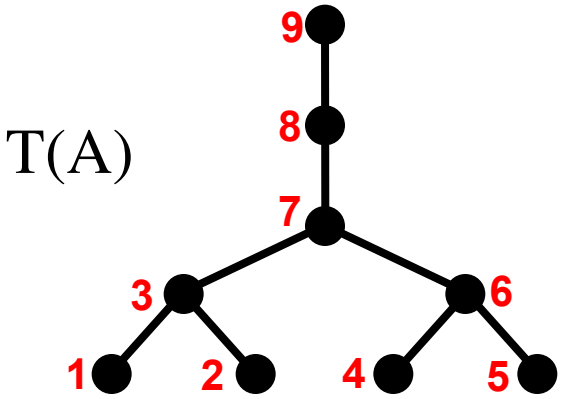




L+U

Symmetric-pattern multifrontal factorization



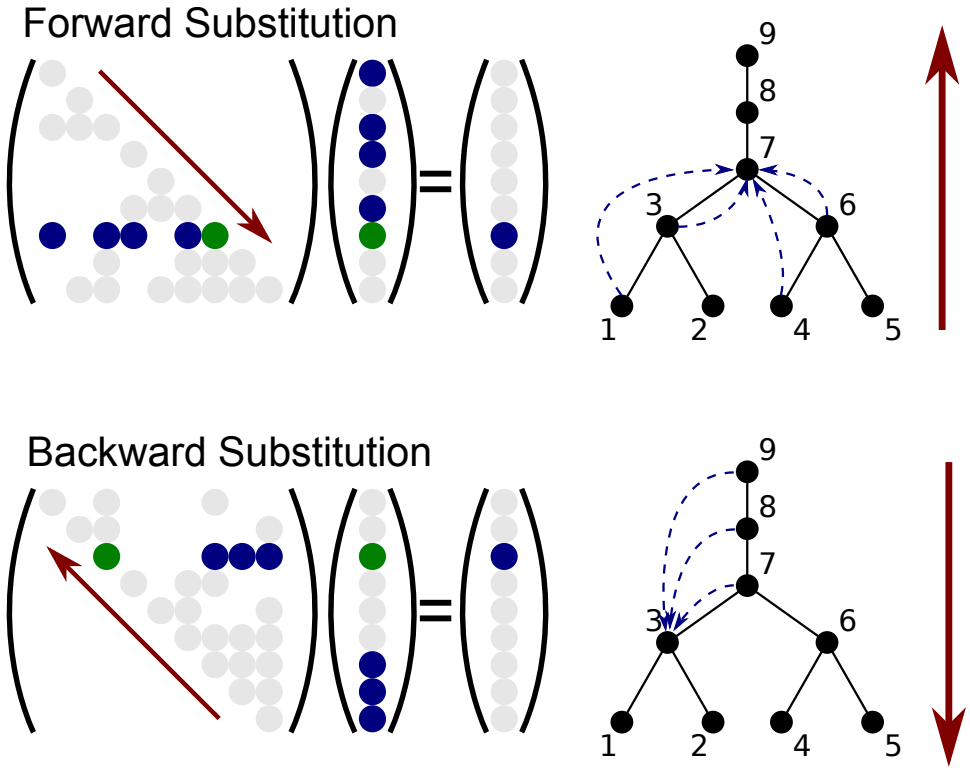


- variant of right-looking
- Really uses supernodes, not nodes
- All arithmetic happens on dense square matrices.
- Needs extra memory for a stack of pending update matrices
- Potential parallelism:
  1. between independent tree branches
  2. parallel dense ops on frontal matrix

Sparse triangular solution

- Forward substitution for x = L \ b (back substitution for x = U \ b)
- Row-oriented = dot-product = left-looking

```
for  i = 1 to n do
     x(i) = b(i);
     // dot-product
     for j = 1 to i-1 do
          x(i) = x(i) – L(i, j) * x(j);
     end for
     x(i) = x(i) / L(i, i);
end for
```

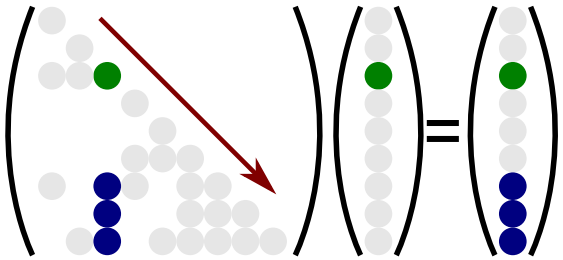


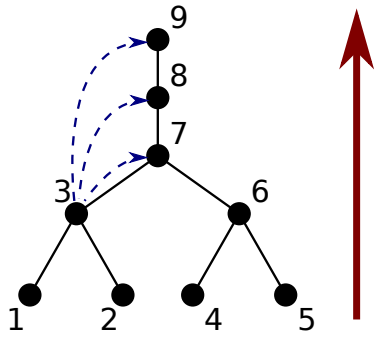
Sparse triangular solution: x = L \ b

- column-oriented = saxpy = right-looking
- Either way works in O(nnz(L)) time

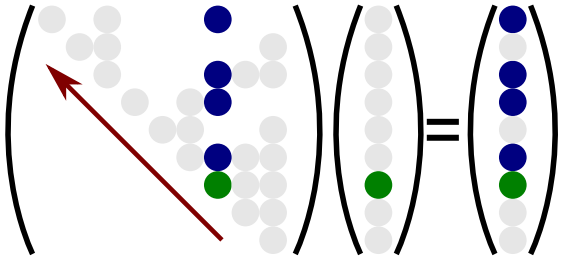
```
x(1:n) = b(1:n);
for j = 1 to n do
    x(j) = x(j) / L(j, j);
    // saxpy
    x(j+1:n) = x(j+1:n) –
                L(j+1:n, j) * x(j);
 end for
```

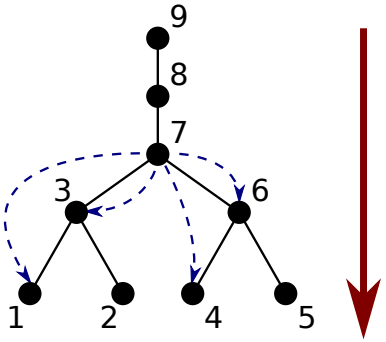
Forward Substitution





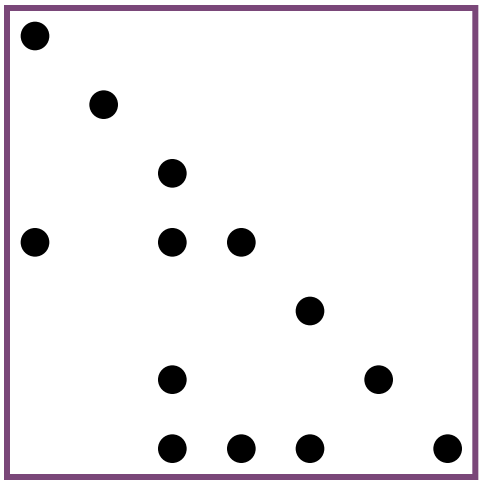
Backward Substitution



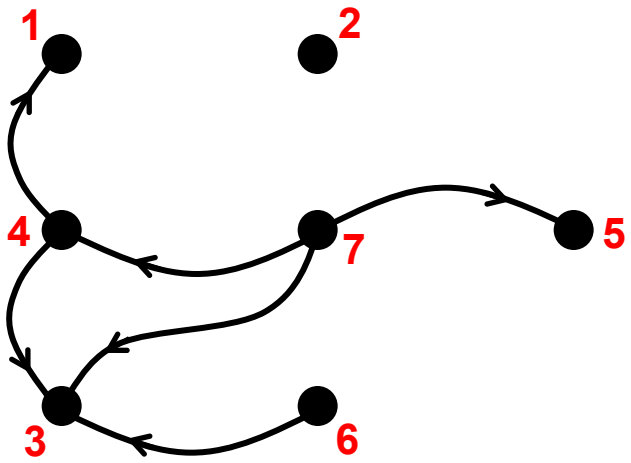


Sparse right-hand side: x = L \ b, b sparse

Use Directed Acyclic Graph (DAG)



A

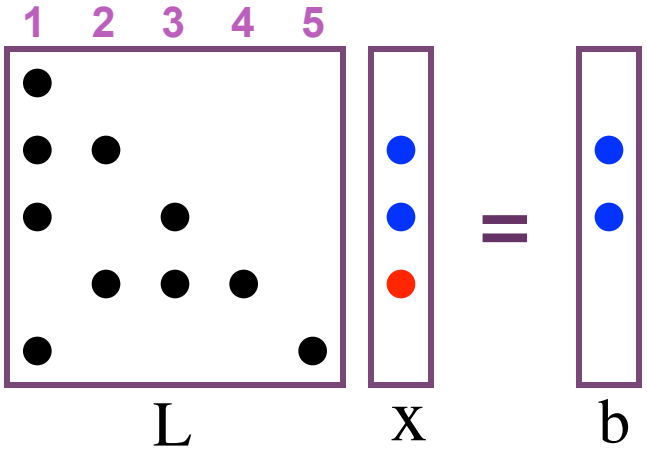


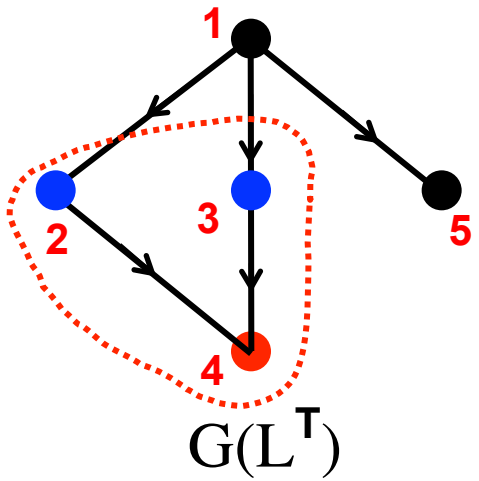
G(A)

- If A is triangular, G(A) has no cycles
- Lower triangular => edges directed from higher to lower #s
- Upper triangular => edges directed from lower to higher #s

Sparse right-hand side: x = L \ b, b sparse

b is sparse, x is also sparse, but may have fill-ins





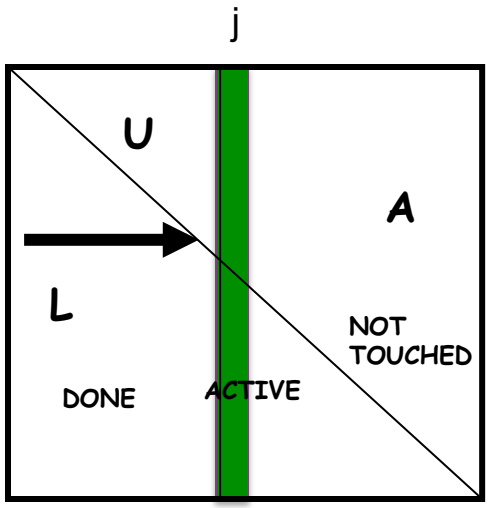
1. Symbolic:
   - Predict structure of x by depth-first search from nonzeros of b
2. Numeric:
   - Compute values of x in topological order

Time = O(flops)

Recall: left-looking sparse LU

sparse right-hand side

```
U(1:j-1, j) = L(1:j-1, 1:j-1) \ A(1:j-1, j)
for j = 1 to n d
    for k in struct(U(1:j-1, j)) do
        cmod(j, k)
    end for
    cdiv(j)
end for
```



- Used in symbolic factorization to find nonzeros in column j

References

- M.T. Heath, E. Ng., B.W. Peyton, “Parallel Algorithms for Sparse Linear Systems”, SIAM Review, Vol. 33 (3), pp. 420-460, 1991.
- E. Rothberg and A. Gupta, “Efficient Sparse Matrix Factorization on High-Performance Workstations--Exploiting the Memory Hierarchy”, ACM. Trans. Math, Software, Vol. 17 (3), pp. 313-334, 1991
- E. Rothberg and A. Gupta, “An Efficient Block-Oriented Approach to Parallel Sparse Cholesky Factorization, SIAM J. Sci. Comput., Vol. 15 (6), pp. 1413-1439, 1994.
- J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W.H. Liu, “A Supernodal Approach to Sparse Partial Pivoting'', SIAM J. Matrix Analysis and Applications, vol. 20 (3), pp. 720-755, 1999.
- J. W. H. Liu, “The Multifrontal Method for Sparse Matrix Solution: theory and Practice”, SIAM Review, Vol. 34 (1), pp. 82-109, 1992.

Exercises

1. Study and run the OpenMP code of dense LU factorization in Hands-On-Exercises/ directory

4th Gene Golub SIAM Summer School, 7/22 – 8/7, 2013, Shanghai

Lecture 5

Parallel Sparse Factorization, Triangular Solution

Xiaoye Sherry Li
Lawrence Berkeley National Laboratory, USA
xsli@lbl.gov

crd-legacy.lbl.gov/~xiaoye/G2S3/

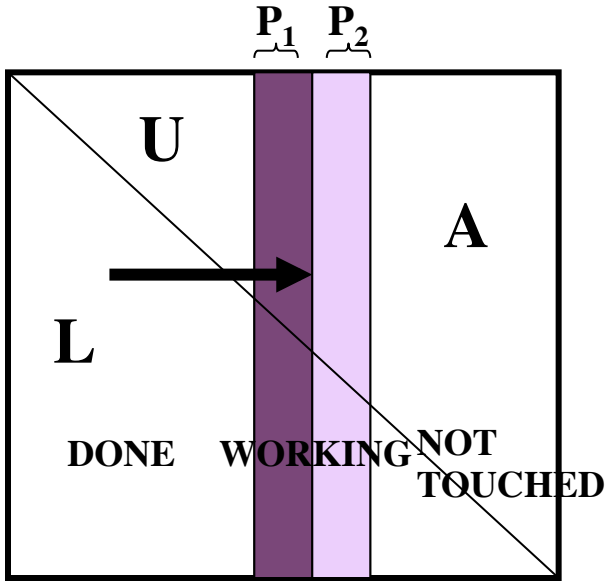
Lecture outline

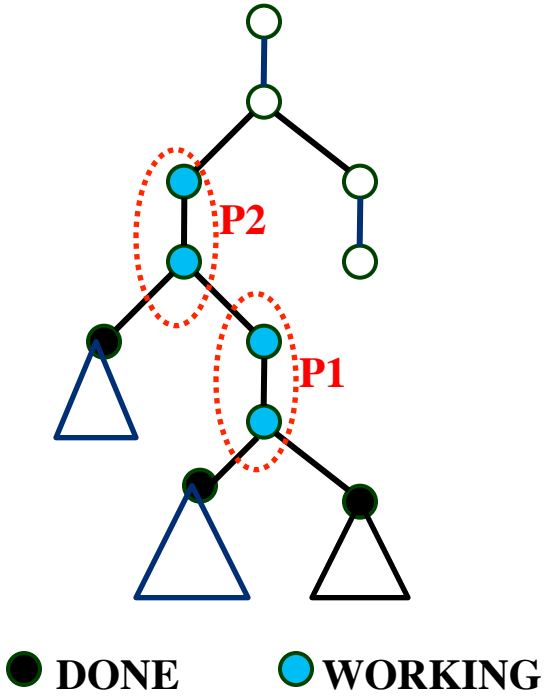
- Shared-memory
- Distributed-memory
- Distributed-memory triangular solve

- Collection of sparse codes, sparse matrices

SuperLU_MT [Li, Demmel, Gilbert]

- Pthreads 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)





Benchmark matrices

	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

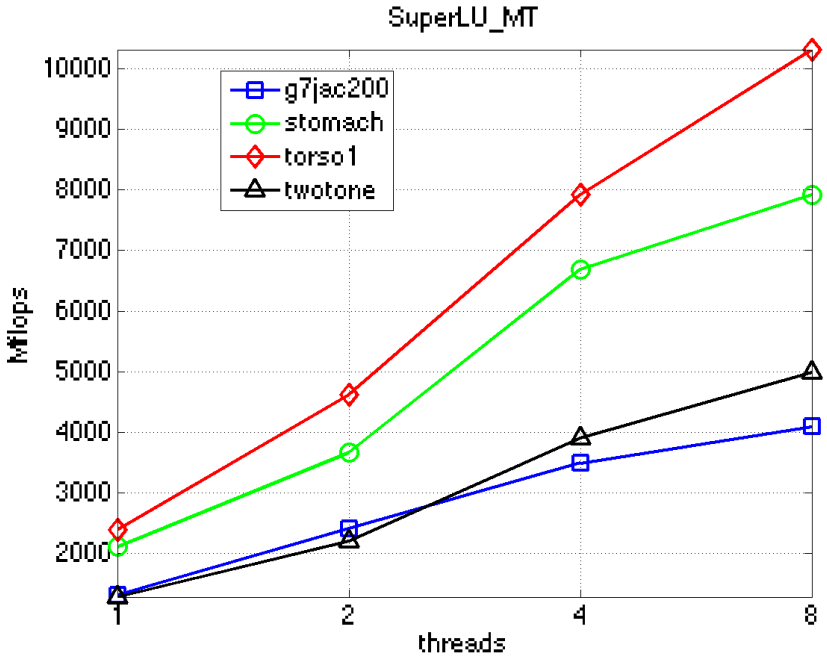
Multicore platforms

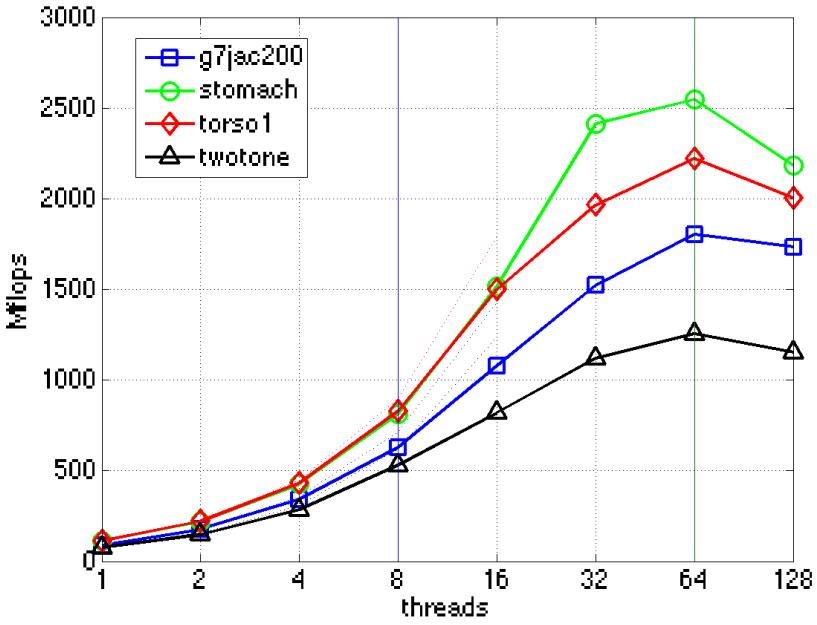
- Intel Clovertown (Xeon 53xx)
  - 2.33 GHz Xeon, 9.3 Gflops/core
  - 2 sockets x 4 cores/socket
  - L2 cache: 4 MB/2 cores

- Sun Niagara 2 (UltraSPARC T2):
  - 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

Intel Clovertown, Sun Niagara 2

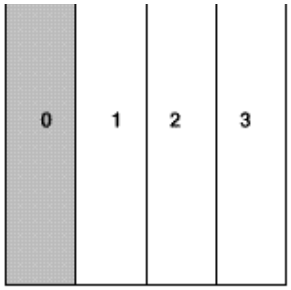
- Maximum speed up 4.3 (Intel), 20 (Sun)
- Question: tools to analyze resource contention?

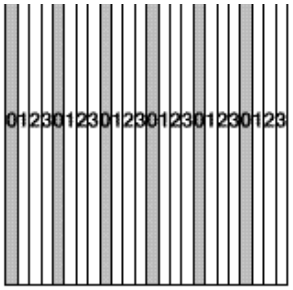




Matrix distribution on large distributed-memory machine

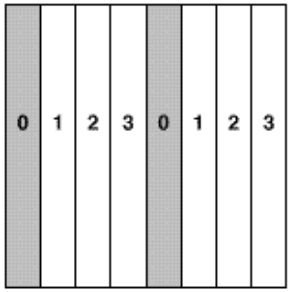
1D blocked

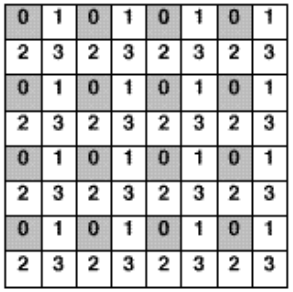




1D cyclic

1D block cyclic



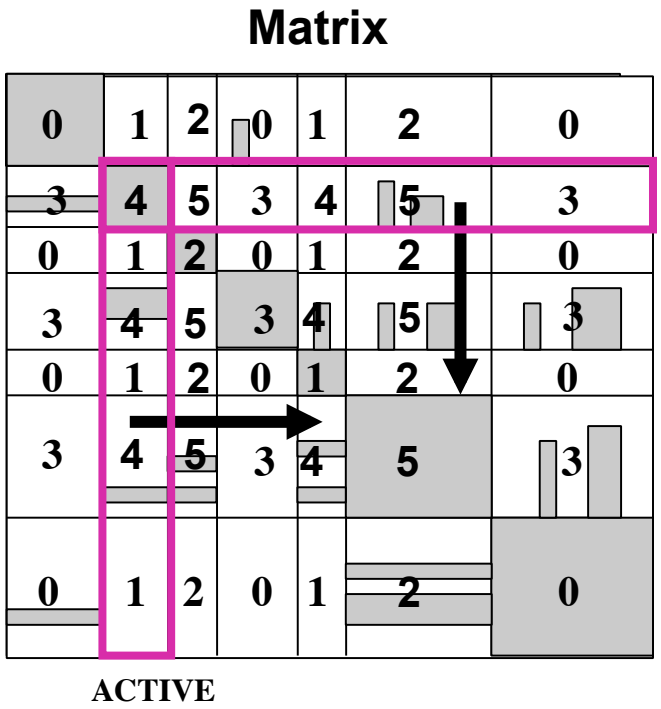


2D block cyclic

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

2D Block Cyclic Distr. for Sparse L & U

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



Process(or) mesh

0	1	2
3	4	5

SuperLU_DIST: GE with static pivoting

[Li, Demmel, Grigori, Yamazaki]

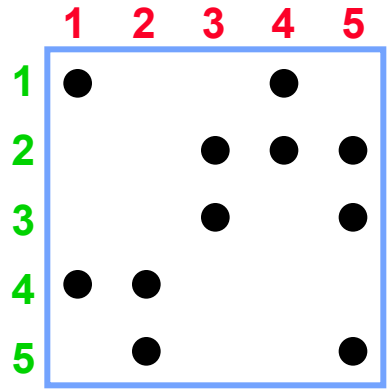
- Target: Distributed-memory multiprocessors
- Goal: No pivoting during numeric factorization

1. Permute A unsymmetrically to have large elements on the diagonal (using weighted bipartite matching)
2. Scale rows and columns to equilibrate
3. Permute A symmetrically for sparsity
4. Factor A = LU with no pivoting, fixing up small pivots:

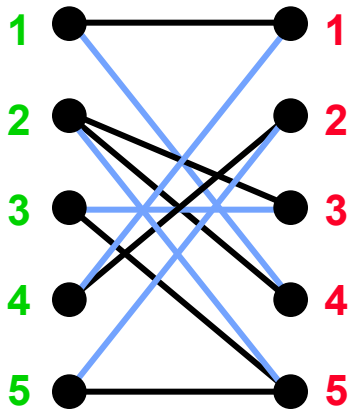
   if $|a_{ii}| < \varepsilon \cdot ||A||$ then replace a_{ii} by $\pm\varepsilon^{1/2} \cdot ||A||$
5. Solve for x using the triangular factors: Ly = b, Ux = y
6. Improve solution by iterative refinement

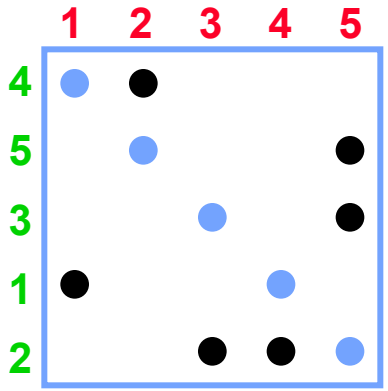
Row permutation for heavy diagonal

[Duff, Koster]



A





PA

- Represent A as a weighted, undirected bipartite graph (one node for each row and one node for each column)
- Find matching (set of independent edges) with maximum product of weights
- Permute rows to place matching on diagonal
- Matching algorithm also gives a row and column scaling to make all diag elts =1 and all off-diag elts <=1

SuperLU_DIST: GE with static pivoting

[Li, Demmel, Grigori, Yamazaki]

- Target: Distributed-memory multiprocessors
- Goal: No pivoting during numeric factorization

1. Permute A unsymmetrically to have large elements on the diagonal (using weighted bipartite matching)
2. Scale rows and columns to equilibrate
3. Permute A symmetrically for sparsity
4. Factor A = LU with no pivoting, fixing up small pivots:

   if $|a_{ii}| < \varepsilon \cdot \|A\|$ then replace a_{ii} by $\pm\varepsilon^{1/2} \cdot \|A\|$
5. Solve for x using the triangular factors: Ly = b, Ux = y
6. Improve solution by iterative refinement

SuperLU_DIST: GE with static pivoting

[Li, Demmel, Grigori, Yamazaki]

- Target: Distributed-memory multiprocessors
- Goal: No pivoting during numeric factorization

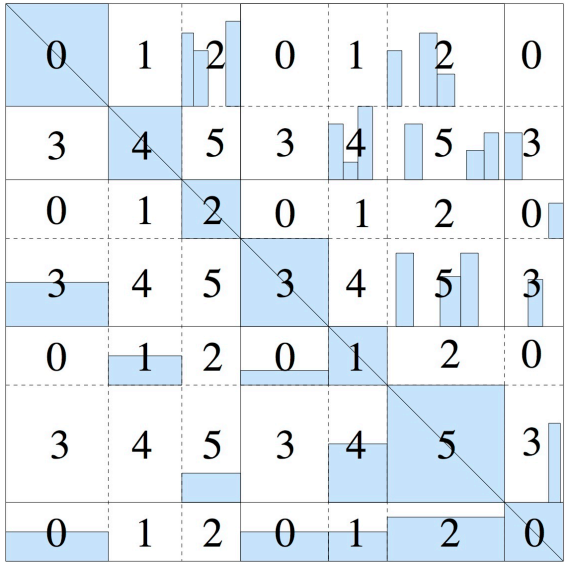
1. Permute A unsymmetrically to have large elements on the diagonal (using weighted bipartite matching)
2. Scale rows and columns to equilibrate
3. Permute A symmetrically for sparsity
4. Factor A = LU with no pivoting, fixing up small pivots:

   if $|a_{ii}| < \varepsilon \cdot \|A\|$ then replace a_{ii} by $\pm\varepsilon^{1/2} \cdot \|A\|$
5. Solve for x using the triangular factors: Ly = b, Ux = y
6. Improve solution by iterative refinement

SuperLU_DIST steps to solution

1. **Matrix preprocessing**
   - static pivoting/scaling/permutation to improve numerical stability and to preseve sparsity
2. **Symbolic factorization**
   - compute e-tree, structure of LU, static comm. & comp. scheduling
   - find supernodes (6-80 cols) for efficient dense BLAS operations
3. **Numerical factorization (dominate)**
   - Right-looking, outer-product
   - 2D block-cyclic MPI process grid
4. **Triangular solve with forward, back substitutions**

2x3 process grid



SuperLU_DIST right-looking factorization

for j = 1, 2, . . . , Ns (# of supernodes)

// panel factorization (row and column)

- factor A(j,j)=L(j,j)*U(j,j), and ISEND to $P_C(j)$ and $P_R(j)$

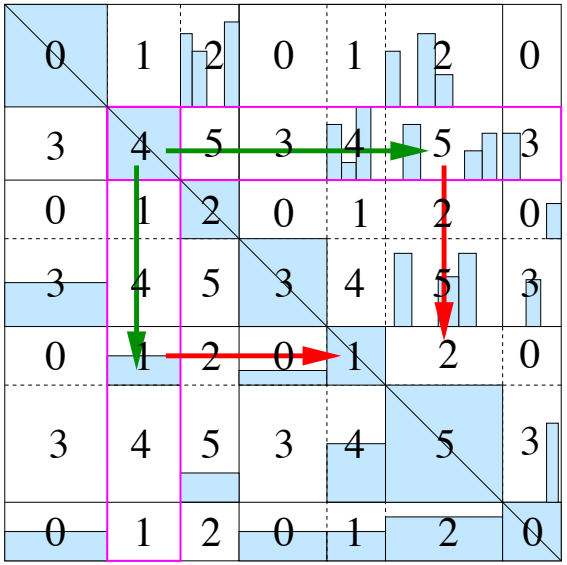
- WAIT for $L_{j,j}$ and factor row $A_{j,\ j+1:Ns}$
  and SEND right to P_C (:)
- WAIT for $U_{j,j}$ and factor column $A_{j+1:Ns,\ j}$
  and SEND down to P_R(:)

// trailing matrix update

- update $A_{j+1:Ns,\ j+1:Ns}$

end for

2x3 process grid

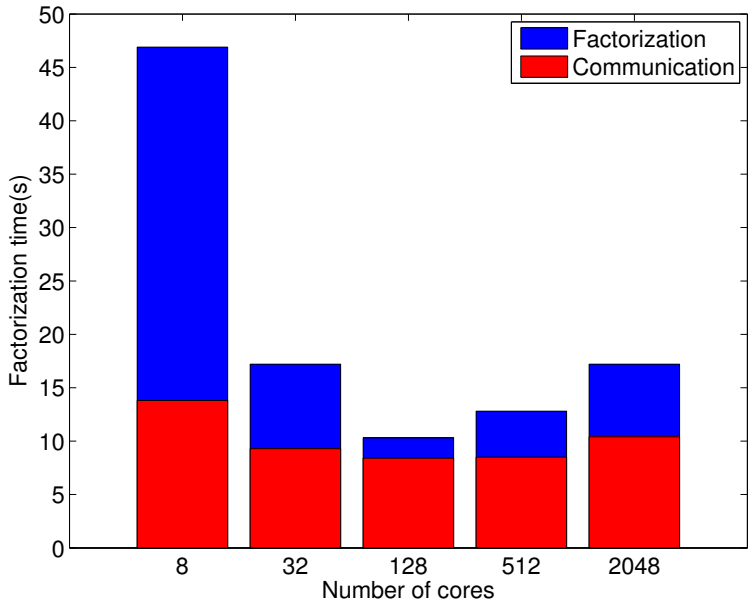


Scalability bottleneck:

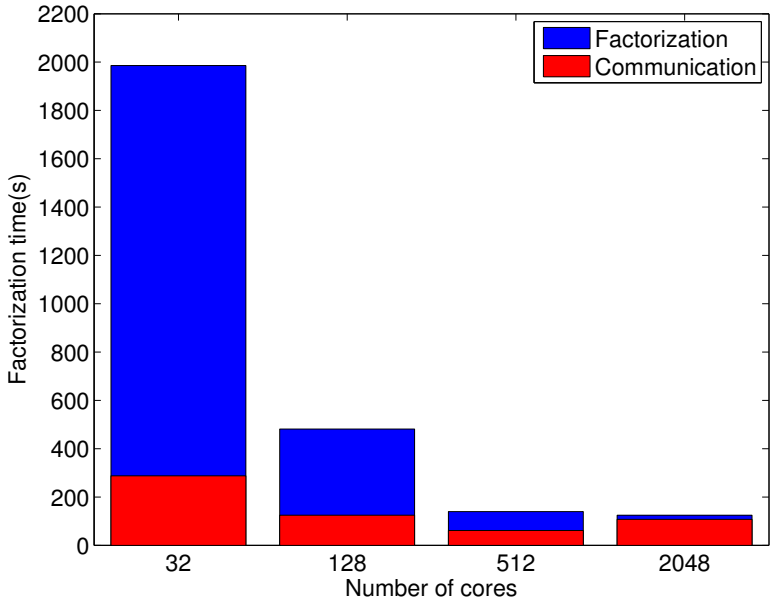
- Panel factorization has sequential flow and limited parallelism.
- All processes wait for diagonal factorization & panel factorization

SuperLU_DIST 2.5 on Cray XE6

- Profiling with IPM
- Synchronization dominates on a large number of cores
  - up to 96% of factorization time



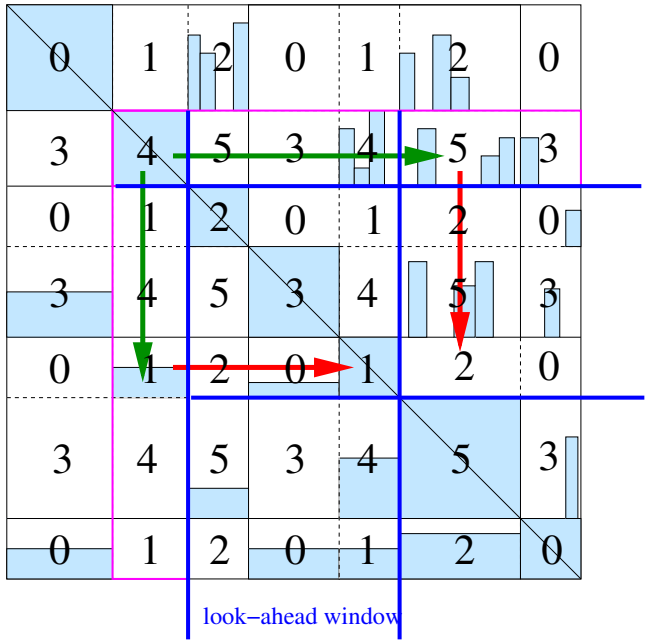
Accelerator (sym), n=2.7M, fill-ratio=12



DNA, n = 445K, fill-ratio= 609

Look-ahead factorization with window size n_w

for j = 1, 2, . . . , Ns (# of supernodes)
  // look-ahead row factorization
  for k = j+1 to j+n_w do
    if ($L_{k,k}$ has arrived) factor $A_{k,(k+1):Ns}$ and ISEND to $P_C(:)$
  end for
  // synchronization
  - factor $A_{j,j}$ =$L_{j,j}U_{j,j}$, and ISEND to $P_C(j)$ and $P_R(j)$
  - WAIT for $L_{j,j}$ and factor row $A_{j,\ j+1:Ns}$
  - WAIT for $L_{:,\ j}$ and $U_{j,\ :}$
  // look-ahead column factorization
  for k = j+1 to j+n_w do
    update $A_{:,k}$
    if ($A_{:,k}$ is ready) factor $A_{k:Ns,k}$ and ISEND to $P_{R(:)}$
  end for
  // trailing matrix update
  - update remaining $A_{\ j+nw+1:Ns,\ j+nw+1:Ns}$
end for



- At each j-th step, factorize all "ready" panels in the window
  - reduce idle time; overlap communication with computation; exploit more parallelism

Expose more "Ready" panels in window

- Schedule tasks with better order as long as tasks dependencies are respected

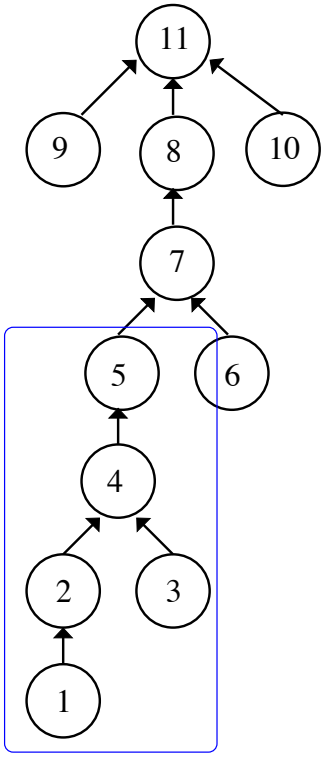
Dependency graphs:

1. LU DAG: all dependencies
2. Transitive reduction of LU DAG: smallest graph, removed all redundant edges, but expensive to compute
3. Symmetrically pruned LU DAG (rDAG): in between LU DAG and its transitive reduction, cheap to compute
4. Elimination tree (e-tree):
   - symmetric case: e-tree = transitive reduction of Cholesky DAG, cheap to compute
   - unsymmetric case: e-tree of $|A|^T+|A|$, cheap to compute

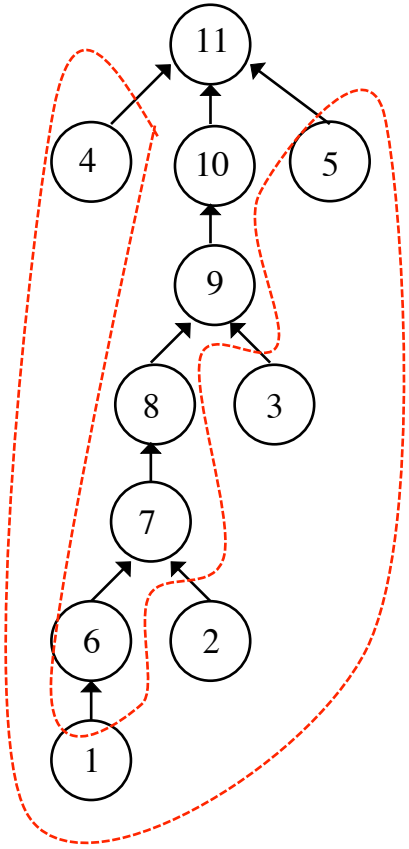
Example: reordering based on e-tree

Window size = 5

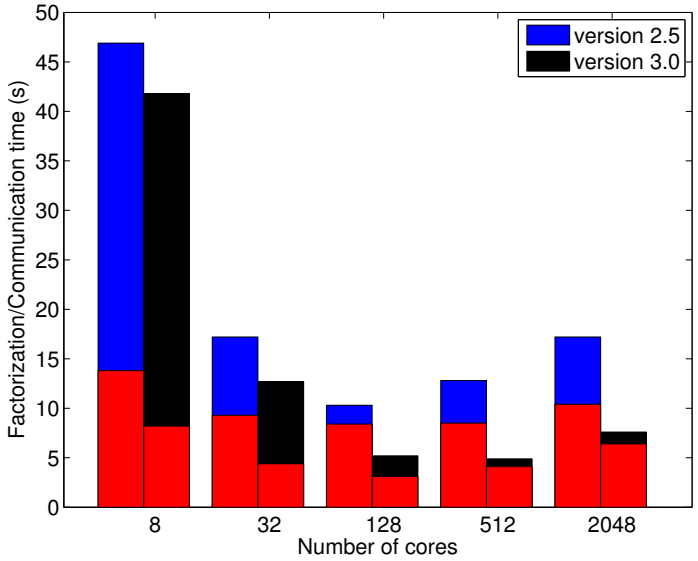
- Postordering based on depth-first search



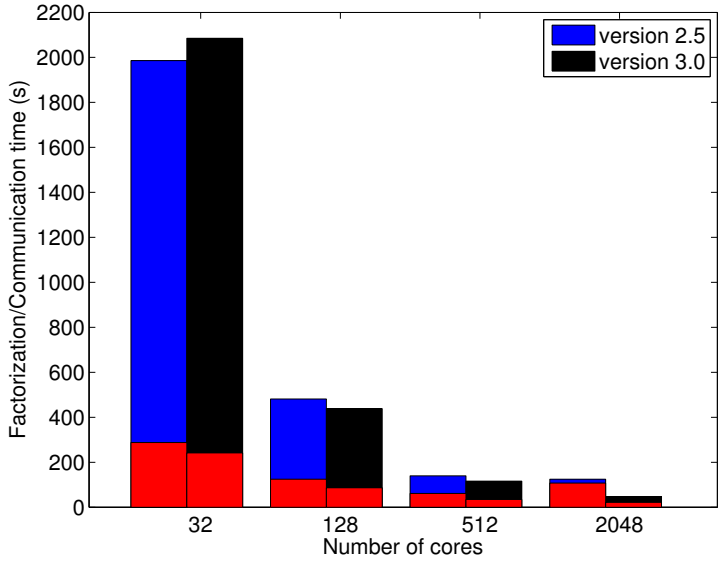
- Bottomup level-based ordering



SuperLU_DIST 2.5 and 3.0 on Cray XE6



Accelerator (sym), n=2.7M, fill-ratio=12



DNA, n = 445K, fill-ratio= 609

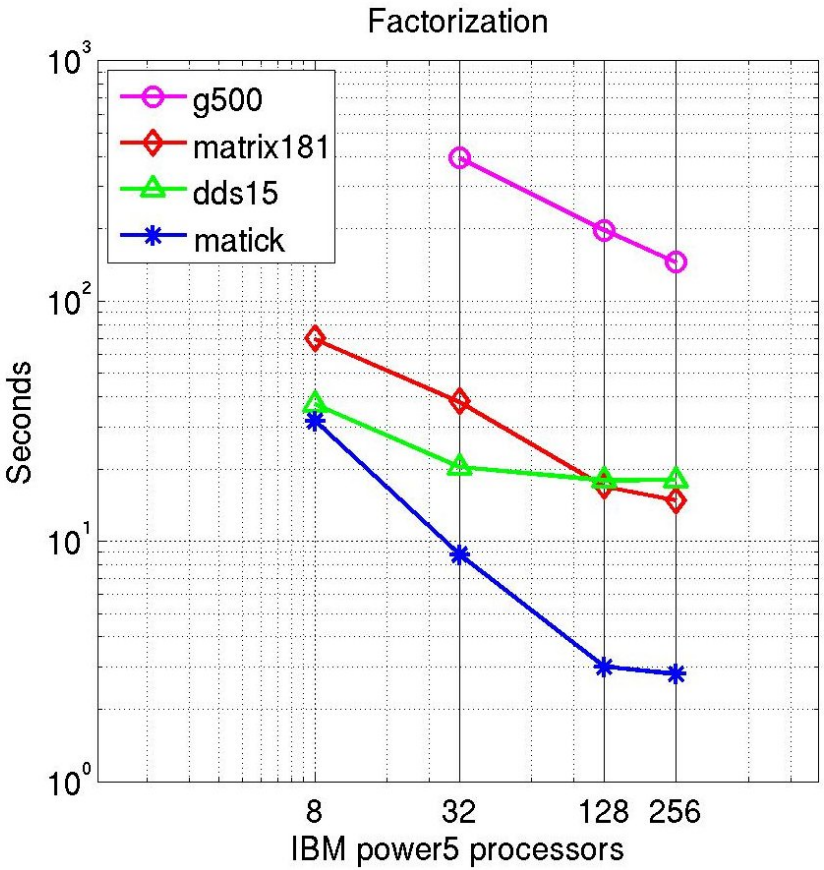
- Idle time was significantly reduced (speedup up to 2.6x)
- To further improve performance:
  - more sophisticated scheduling schemes
  - hybrid programming paradigms

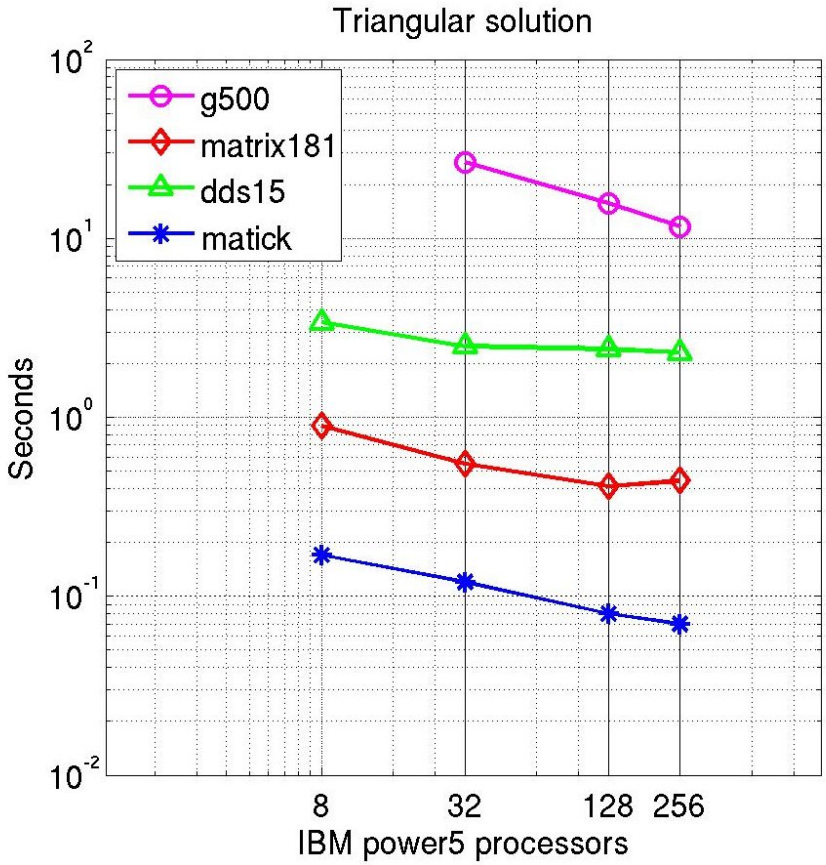
Examples

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
matick	Circuit sim. MNA method (IBM)	Complex	16,019	4005	64.3	1.0

- Sparsity-preserving ordering: MeTis applied to structure of A’ +A

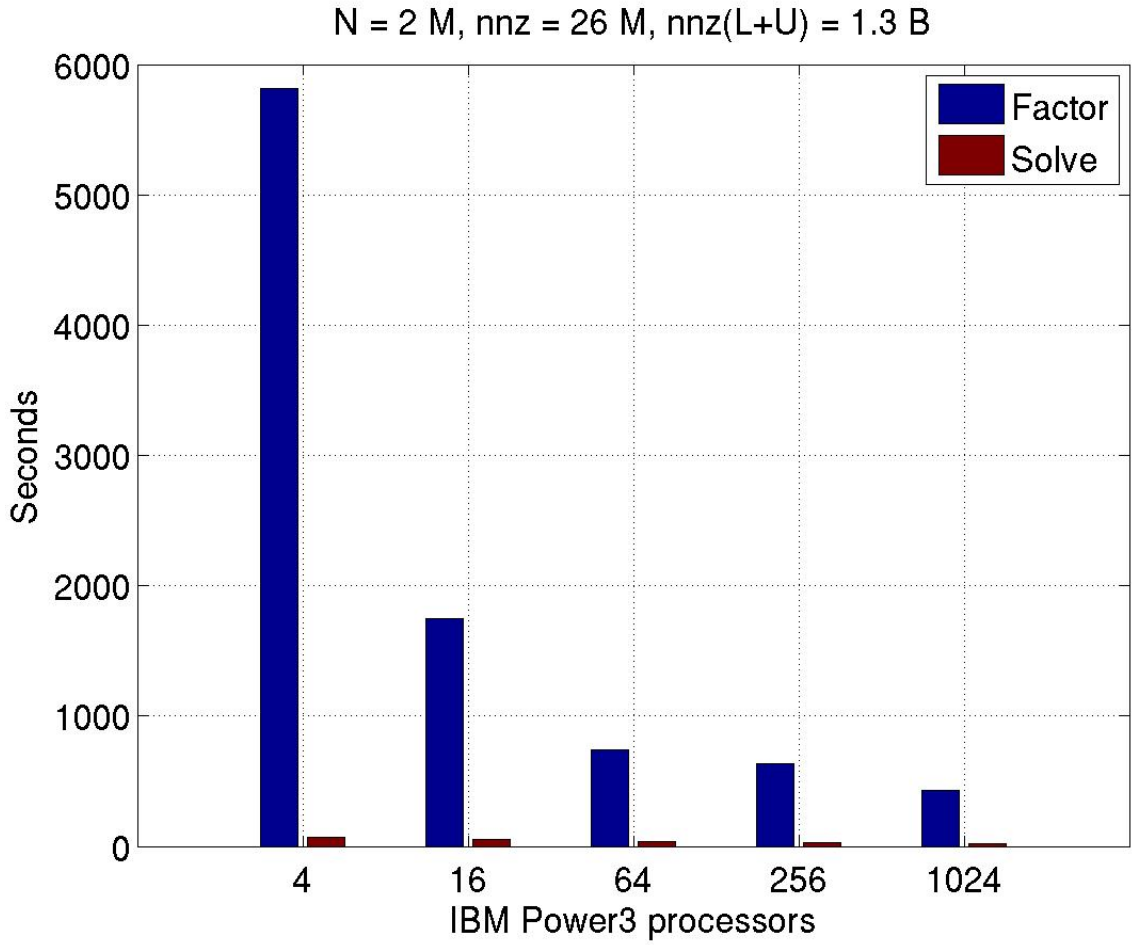
Performance on IBM Power5 (1.9 GHz)





- Up to 454 Gflops factorization rate

Performance on IBM Power3 (375 MHz)



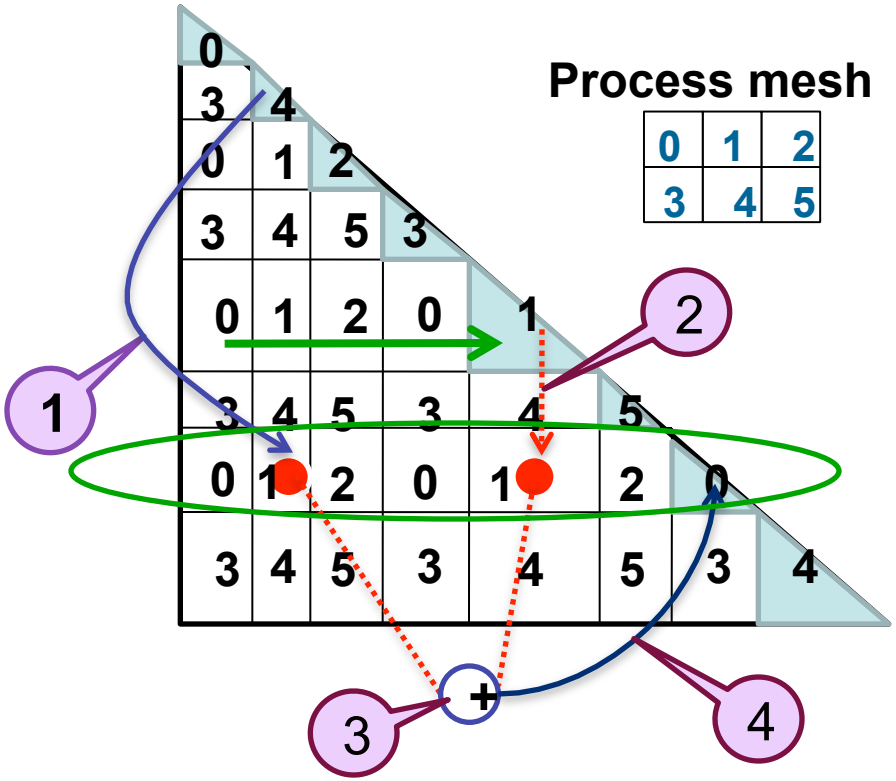
- Quantum mechanics, complex

Distributed triangular solution

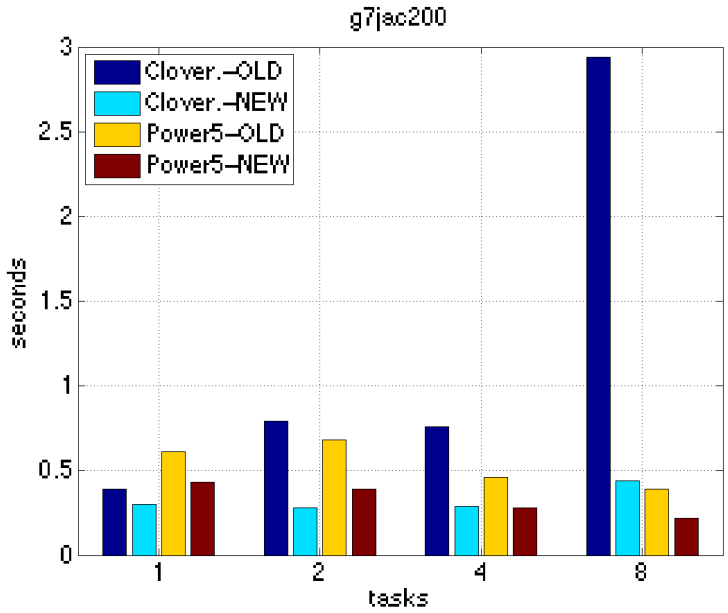
- Challenge: higher degree of dependency

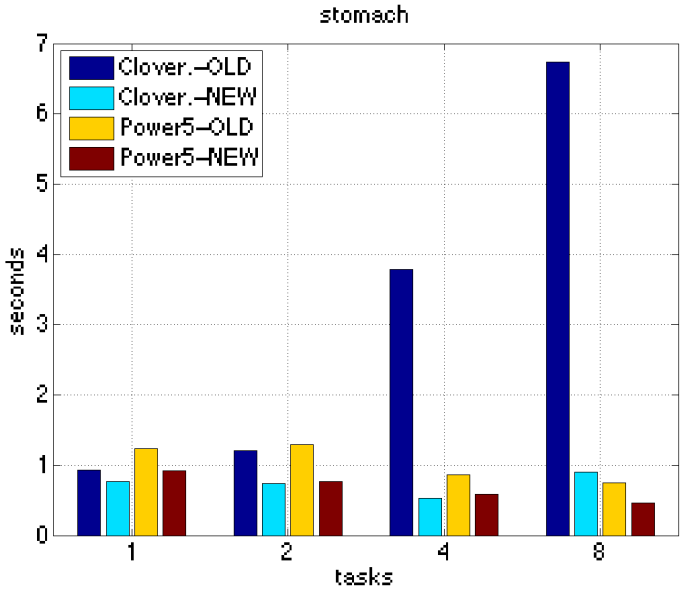
$$x_i = \frac{b_i - \sum_{j=1}^{i-1} L_{ij} \cdot x_j}{L_{ii}}$$

- Diagonal process computes the solution



Parallel triangular solution





- Clovertown: 8 cores; IBM Power5: 8 cpus/node

- OLD code: many MPI_Reduce of one integer each, accounting for 75% of time on 8 cores
- NEW code: change to one MPI_Reduce of an array of integers

- Scales better on Power5

MUMPS: distributed-memory multifrontal

[Current team: Amestoy, Buttari, Guermouche, L'Excellent, Uçar]

- Symmetric-pattern multifrontal factorization
- Parallelism both from tree and by sharing dense ops
- Dynamic scheduling of dense op sharing
- Symmetric preordering
- For nonsymmetric matrices:
  - optional weighted matching for heavy diagonal
  - expand nonzero pattern to be symmetric
  - numerical pivoting only within supernodes if possible (doesn't change pattern)
  - failed pivots are passed up the tree in the update matrix

Collection of software, test matrices

- Survey of different types of direct solver 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, distributed-memory, out-of-core
    - **Accelerators such as GPU, FPGA become active, have papers, no public code yet**

- The University of Florida Sparse Matrix Collection
  http://www.cise.ufl.edu/research/sparse/matrices/

References

- X.S. Li, “An Overview of SuperLU: Algorithms, Implementation, and User Interface”, ACM Transactions on Mathematical Software, Vol. 31, No. 3, 2005, pp. 302-325.
- X.S. Li and J. Demmel, “SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems”, ACM Transactions on Mathematical Software, Vol. 29, No. 2, 2003, pp. 110-140.
- X.S. Li, “Evaluation of sparse LU factorization and triangular solution on multicore platforms”, VECPAR'08, June 24-27, 2008, Toulouse.
- I. Yamazaki and X.S. Li, “New Scheduling Strategies for a Parallel Right-looking Sparse LU Factorization Algorithm on Multicore Clusters”, IPDPS 2012, Shanghai, China, May 21-25, 2012.
- L. Grigori, X.S. Li and J. Demmel, “Parallel Symbolic Factorization for Sparse LU with Static Pivoting”. SIAM J. Sci. Comp., Vol. 29, Issue 3, 1289-1314, 2007.
- P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, and J. Koster, “A fully asynchronous multifrontal solver using distributed dynamic scheduling”, SIAM Journal on Matrix Analysis and Applications, 23(1), 15-41 (2001).
- P. Amestoy, I.S. Duff, A. Guermouche, and T. Slavova. Analysis of the Solution Phase of a Parallel Multifrontal Approach. Parallel Computing, No 36, pages 3-15, 2009.
- A. Guermouche, J.-Y. L'Excellent, and G.Utard, Impact of reordering on the memory of a multifrontal solver. Parallel Computing, 29(9), pages 1191-1218.
- F.-H. Rouet, Memory and Performance issues in parallel multifrontal factorization and triangular solutions with sparse right-hand sides, PhD Thesis, INPT, 2012.
- P. Amestoy, I.S. Duff, J-Y. L'Excellent, X.S. Li, “Analysis and Comparison of Two General Sparse Solvers for Distributed Memory Computers”, ACM Transactions on Mathematical Software, Vol. 27, No. 4, 2001, pp. 388-421.

Exercises

1. Download and install SuperLU_MT on your machine, then run the examples in EXAMPLE/ directory.
2. Run the examples in SuperLU_DIST_3.3 directory.

4th Gene Golub SIAM Summer School, 7/22 – 8/7, 2013, Shanghai

Lecture 6

Incomplete Factorization

Xiaoye Sherry Li
Lawrence Berkeley National Laboratory, USA
xsli@lbl.gov

crd-legacy.lbl.gov/~xiaoye/G2S3/

Lecture outline

- Supernodal LU factorization (SuperLU)
- Supernodal ILUTP with adaptive dual dropping
  - Threshold dropping in supernode
  - Secondary dropping for memory concern
- Variants: Modified ILU (MILU)
- Extensive experiments, comparison with other approaches
  - 232 matrices

Preconditioner

- Improve efficiency and robustness of iterative solvers
- Solve a transformed linear system, hopefully easier:
  - $M^{-1}Ax = M^{-1}b$ …. Left preconditioning
  - $AM^{-1}u = b$, $x = M^{-1}u$ …. Right preconditioning
- Goal: find preconditioner M ~ A, so that the eigenvalue spectrum of $M^{-1}A$ is improved.
  1. Cheap to construct, store, "invert", parallelize
  2. Good approximation of A

  contradictory goals → tradeoff
- Standard design strategy:
  - Start with a complete factorization
  - Add approximations to make it cheaper (cf. 1) while (hopefully/ provably) affecting little 2.
- We will present two approaches
  - Incomplete factorization
  - Low-rank approximations

ILU preconditioner

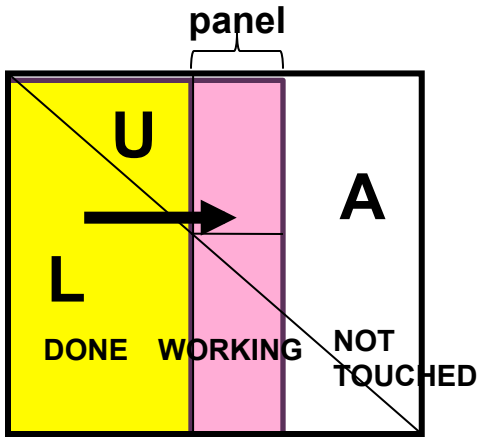
- Structure-based dropping: level-of-fill
  - ILU(0), ILU(k)
  - Rationale: the higher the level, the smaller the entries
  - Separate symbolic factorization to determine fill-in pattern
- Value-based dropping: drop truly small entries
  - Fill-in pattern must be determined on-the-fly

- ILUTP [Saad]: among the most sophisticated, and (arguably) robust; implementation similar to direct solver
  - "T" = threshold, "P" = pivoting
  - Dual dropping: ILUTP(p, tau)
    1) **Remove elements smaller than tau**
    2) **At most p largest kept in each row or column**

SuperLU [Demmel/Eisenstat/Gilbert/Liu/Li '99]

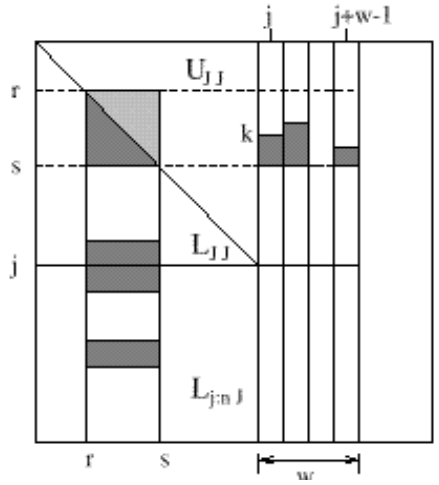
http://crd.lbl.gov/~xiaoye/SuperLU

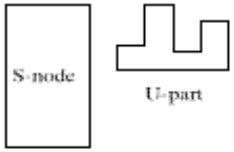
- **Left-looking, supernode**



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





Primary dropping rule: S-ILU(tau)

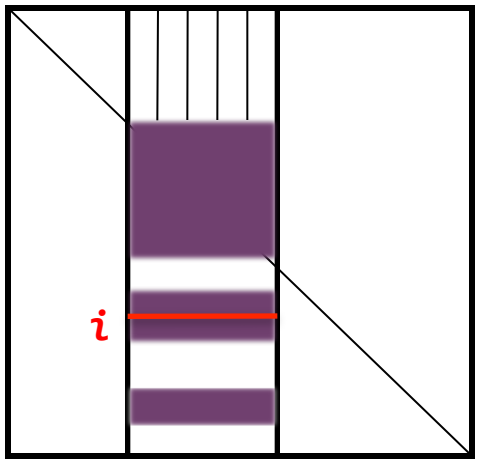
- Similar to ILUTP, adapted to supernode
  1. U-part:

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

  2. L-part: retain supernode

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

- Remarks
  1) Delayed dropping
  2) Entries computed first, then dropped.
     May not save many flops compared to LU
  3) Many choices for RowSize() metric



Dropping in supernode

Supernode $L(:, s:t)$, if $RowSize(i, s:t) < \tau$, then set the entire i-th row to zero

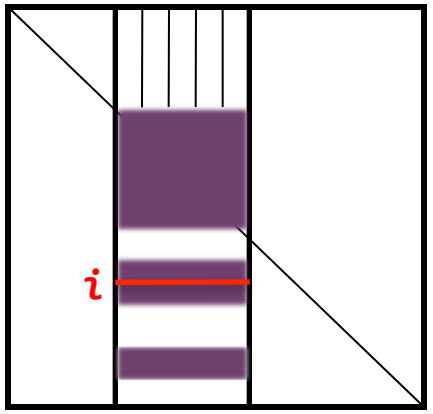
RowSize() metric: let m = t-s+1 be the supernode size

1) Mean: $RowSize(x) = \frac{\| x \|_1}{m}$ [used by Gupta/George for IC]

2) Generalized-mean: $RowSize(x) = \frac{\| x \|_2}{\sqrt{m}}$

3) Infinity-norm: $RowSize(x) = \| x \|_\infty$

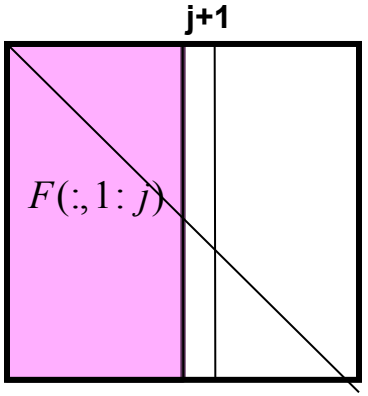
Every dropped entry in L would also be dropped in a column-wise algorithm



Since $\frac{\| x \|_1}{m} \le \frac{\| x \|_2}{\sqrt{m}} \le \| x \|_\infty$, 1) is most aggressive, 3) is conservative

Secondary dropping rule: S-ILU(p,tau)

- Control fill ratio with a user-desired upper bound γ
- Earlier work, column-based
  - [Saad]: ILU(p, tau), at most p largest nonzeros allowed in each row
  - [Gupta/George]: p adaptive for each column $p(j) = \gamma \cdot nnz(A(:, j))$
    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:

    $$fr(j) = nnz(F(:, 1:j)) / nnz(A(:, 1:j))$$

  - Define adaptive upper bound function $f(j) \in [1, \gamma]$
    If $fr(j)$ exceeds $f(j)$, retain only p largest, such that $fr(j) \le f(j)$

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

Experiments: GMRES + ILU

- Use restarted GMRES with ILU as a right preconditioner
  Solve $PA(\widetilde{L}\widetilde{U})^{-1}y = Pb$
- Size of Krylov subspace set to 50
- Initial guess is a 0-vector
- Stopping criteria: $\|b - Ax_k\|_2 \le 10^{-8}\|b\|_2$ and ≤ 500 iterations

- 232 unsymmetric test matrices; RHS is generated so the true solution is 1-vector
  - 227 from Univ. of Florida Sparse Matrix Collection
    dimension 5K – 1M, condition number below 10^{15}
  - 5 from MHD calculation in tokamak design for plasma fusion energy
- AMD Opteron 2.4 GHz quad-core (Cray XT5), 16 GBytes memory, PathScale pathcc and pathf90 compilers

Compare with column C-ILU(p, tau)

- C-ILU: set maximum supernode size to be 1
- Maxsuper = 20, gamma = 10, tau = 1e-4

	Factor construction				GMRES		Total Sec.
	Fill-ratio	S-node Cols	Flops (10^9)	Fact. sec.	Iters	Iter sec.	
	$RowSize(x) = \| x \|_2 / \sqrt{m}$			138 matrices succeeded			
S-ILU	4.2	2.8	7.60	39.69	21.6	2.93	42.68
C-ILU	3.7	1.0	2.65	65.15	20.0	2.55	67.75
	$RowSize(x) = \| x \|_\infty$			134 matrices succeeded			
S-ILU	4.2	2.7	9.45	54.44	20.5	3.4	57.0
C-ILU	3.6	1.0	2.58	74.10	19.8	2.88	77.04

Supernode vs. column

- Less benefit using supernode compared to complete LU
  - Better, but Less than 2x speedup

- What go against supernode?
  - The average supernode size is smaller than in LU.
  - The row dropping rule in S-ILU tends to leave more fill-ins and operations than C-ILU … we must set a smaller “maxsuper” parameter.

    e.g., 20 in ILU vs. 100 in LU

S-ILU for extended MHD calculation (fusion)

- ILU parameters: $\tau = 10^{-4}, \gamma = 10$
- Up to 9x smaller fill ratio, and 10x faster

Problems	order	Nonzeros (millions)	ILU time	ILU fill-ratio	GMRES time	GMRES iters	SuperLU time	SuperLU fill-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	-

Performance profile

E.D. Dolan and J.J. More, "Benchmarking optimization software with performance profiles", Mathematical Programming, 91(2):201–203, 2002.

- Visually compare solvers X inputs

- Let M = set of matrices, S = set of solvers
- fr(m, s) and t(m, s) denote the fill ratio and total time needed to solve matrix "m" by solver s.
- Calculate the cumulative distribution functions for each solver s:
  - fraction of the problems that s could solve within the fill ratio x

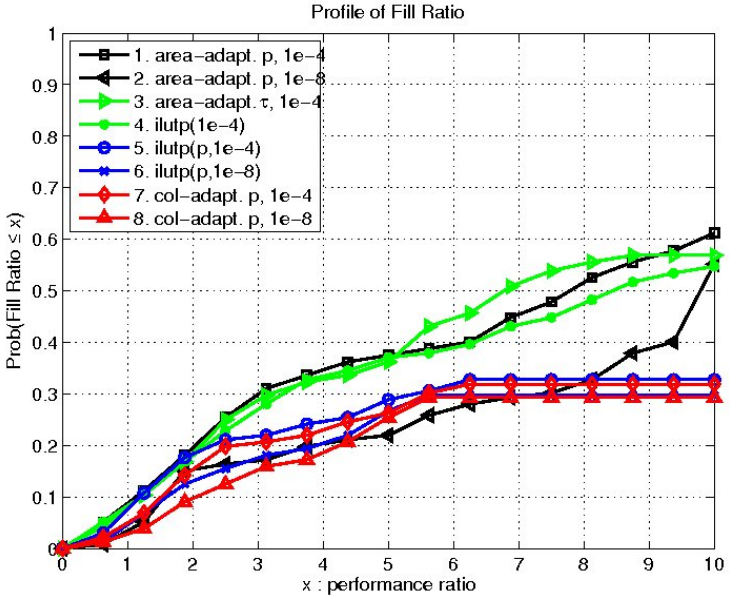
$$\Pr_f(s,x) = \frac{\#\{m \in M : fr(m,s) \le x\}}{\#M}, \quad x \in R$$

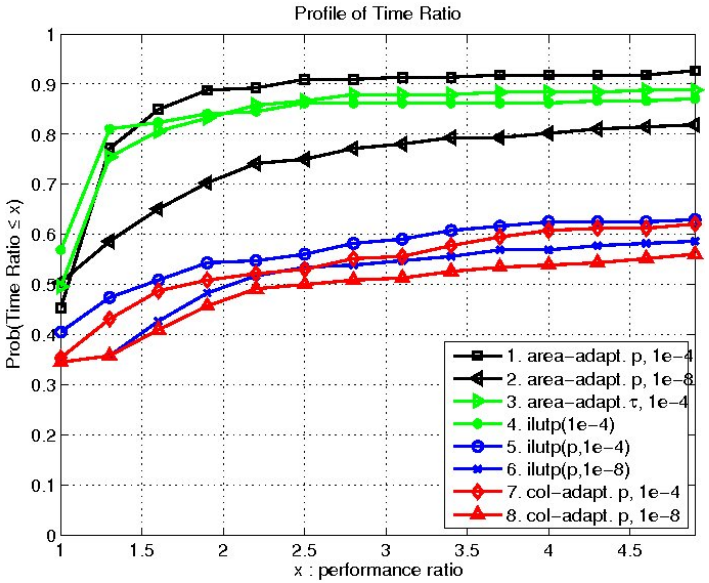
  - fraction of the problems that s could solve within a factor of x of the best solution time among all the solvers

$$\Pr_t(s,x) = \frac{\#\left\{m \in M : \dfrac{t(m,s)}{\min_{s \in S}\{t(m,s)\}} \le x\right\}}{\#M}, \quad x \in R$$

S-ILU comprehensive tests

- Performance profile of fill ratio – fraction of the problems a solver could solve within a fill ratio of X
- Performance profile of runtime – fraction of the problems a solver could solve within a factor X of the best solution time





- Conclusion:
  - New area-based heuristic is much more robust than column-based one
  - ILUTP(tau) is reliable; but need secondary dropping to control memory

Other features in the software

- Zero pivot ?

$$\text{if } u_{jj} = 0, \text{ set it to } \hat{\tau}(j) \| A(:, j) \|_\infty$$

$$\hat{\tau}(j) = 10^{-2(1-j/n)}, \text{ adaptive, increasing with } j, \text{ so } U \text{ is not too ill-conditioned}$$

- Threshold partial pivoting

- Preprocecssing with MC64 [Duff-Koster]
  - With MC64, 203 matrices converge, avg. 12 iterations
  - Without MC64, 170 matrices converge, avg. 11 iterations

- Modified ILU (MILU)
  - Reduce number of zero pivots

Modified ILU (MILU)

- Reduce the effect of dropping: for a row or column, add up the dropped elements to the diagonal of U

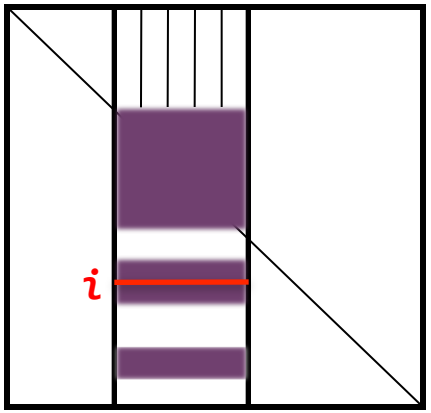
- Classical approach has the following property:
  - Maintain row-sum for a row-wise algorithm: $\widetilde{L}\widetilde{U}e = Ae$
  - Maintain column-sum for a column-wise algorithm: $e^T\widetilde{L}\widetilde{U} = e^T A$

- Another twist … proposed for MIC

  Maintain $LUx = Ax + \Lambda Dx$ for any x, using diagonal perturbations
  - Dupont-Kendall, Axelsson-Gustafsson, Notay (DRIC)
  - Reduce condition number of elliptic discretization matrices by order of magnitude (i.e., from $O(h^{-2})$ to $O(h^{-1})$)

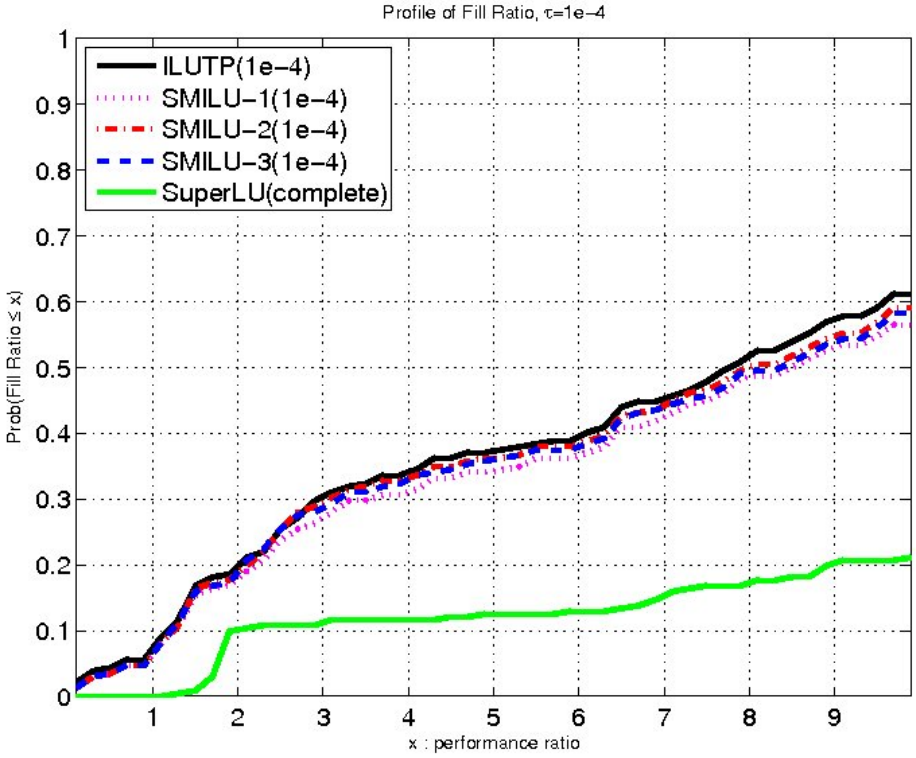
MILU algorithm

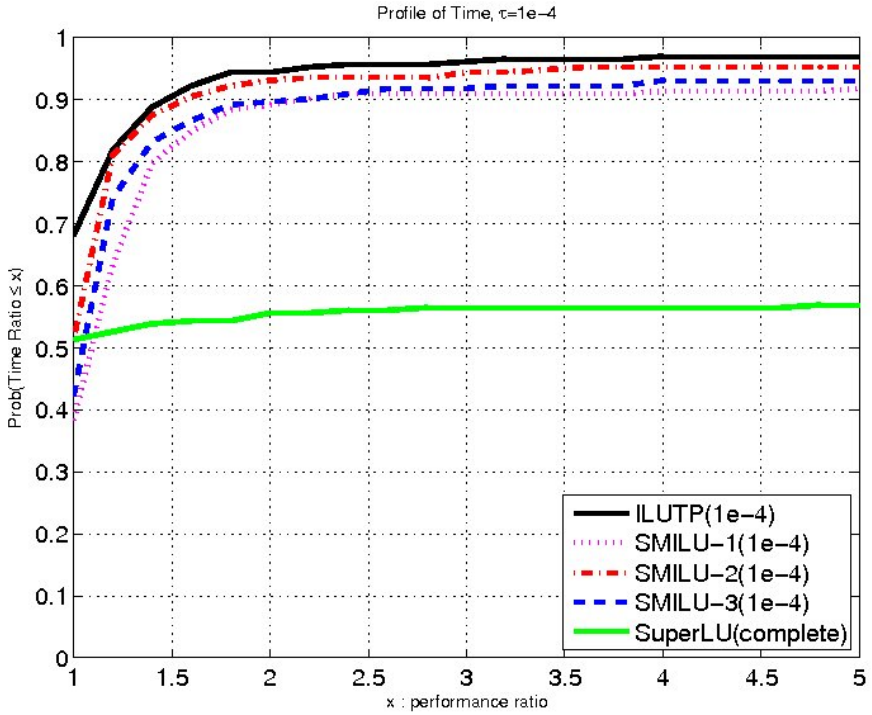
- C-MILU:
  1) Obtain filled column F(:, j), drop from F(:, j)
  2) Add up the dropped entries: $s = \sum_{dropped} f_{ij}$; Set $f_{ij} := f_{ij} + s$
  3) Set U(1:j, j) := F(1:j, j); L(j+1:n, j) := F(j+1: n, j) / F(j, j)
- S-MILU:
  1) First drop from U, $s = \sum_{dropped} U(:,j)$
     Set $u_{jj} := f_{jj} + s$;
  2) When a supernode is formed in L, drop more rows in L, add the dropped entries to diagonal of U



- Our variants:
  - S-MILU-1: $s = \sum_{dropped} U(:,j)$
  - S-MILU-2: $s = |\sum_{dropped} U(:,j)|$, $u_{jj} := f_{ij} + sign(f_{jj})*s$
  - S-MILU-3: $s = \sum_{dropped} |U(:,j)|$, $u_{jj} := f_{ij} + sign(f_{jj})*s$

Modified ILU (MILU)





Another look at MILU – 232 matrices

	Converge	Slow	Diverge	Zero pivots	Average iterations
S-ILU	133	51	46	1737	35
S-MILU-1	125	72	33	1058	34
S-MILU-2	127	71	31	296	30
S-MILU-3	129	73	28	289	33

Compare with the other preconditioners

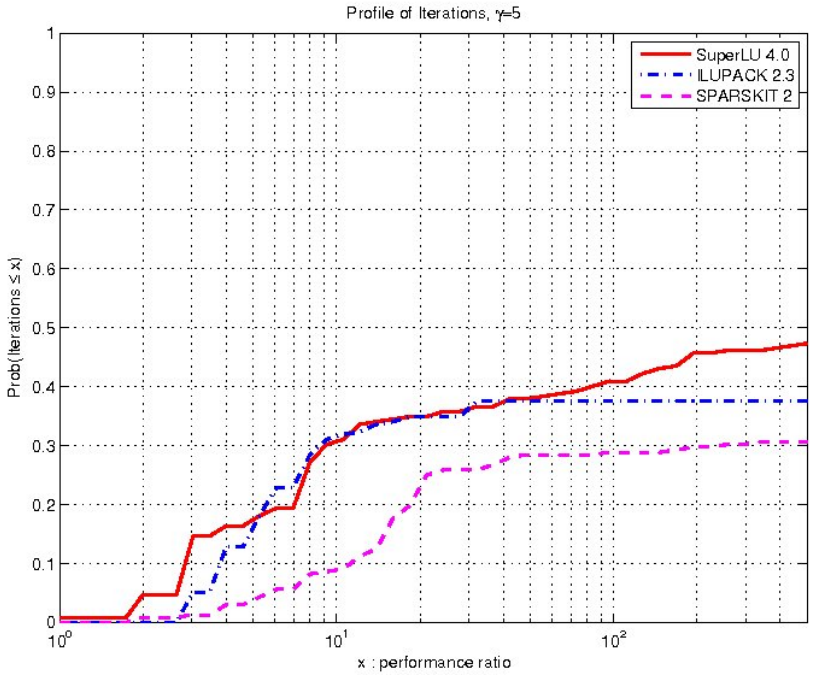
- SPARSKIT [saad] : ILUTP, closest to ours
  - Row-wise algorithm, no supernode
  - Secondary dropping uses a fixed p for each row
- ILUPACK [Bolhoefer et al.] : very different
  - Inverse-based approach: monitor the norm of the k-th row of L^{-1}, if too large, delay pivot to next level
  - Multilevel: restart the delayed pivots in a new level
- ParaSails [Chow]: very different
  - Sparse approximate inverse: $M \sim A^{-1}$
  - Pattern of powers of sparsified A as the pattern of M

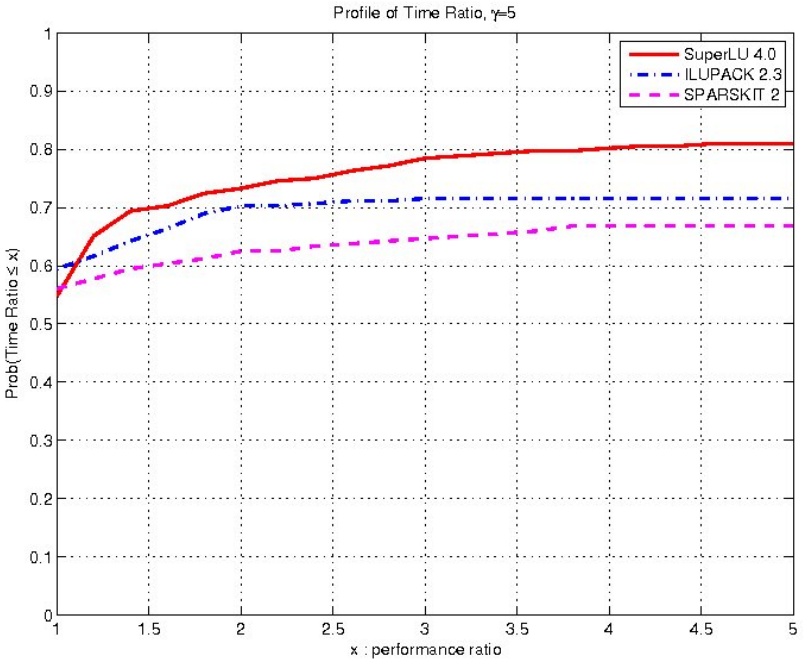
    "thresh" to sparsify A, "nlevels" to keep level of neighbors
  - Default setting: thresh = 0.1, nlevels = 1

    Only 39 matrices converge, 62 hours to construct M, 63 hours after GMRES
  - Smaller thresh and larger nlevels help, but too expensive

Compare with SPARSKIT, ILUPACK

- S-ILU: $\tau = 10^{-4}, \gamma = 5,$ diag_thresh $\eta = 0.1$
- ILUPACK : $\tau = 10^{-4}, \ \gamma = 5, \ \nu = 5$
- SPARSKIT : $\tau = 10^{-4}, \ \gamma = 5, \ p = \gamma \cdot \dfrac{nnz}{n}$





Comparison (cont) … a closer look …

- S-ILU and ILUPACK are comparable: S-ILU is slightly faster, ILUPACK has slightly lower fill
- None of the preconditioners works for all problems … unlike direct methods
- They do not solve the same set of problems
  - S-ILU succeeds with 142
  - ILUPACK succeeds with 130
  - Both succeed with 100 problems

- Remark

  Two methods complimentary to one another, both have their place in practice

Summary

- Secondary dropping: area-based, adaptive-p, adaptive-tau
  - More reliable
- Empirical study of MILU
  - Limited success, disappointing in general

Summary

- 60-70% success with S-ILUTP for 232 matrices.
  When it works, much more efficient than direct solver.
- Supernode
  - Useful, but to less extend compared with complete LU
- Secondary dropping: area-based, adaptive-p, adaptive-tau
  - More reliable
- Software
  - Available in serial SuperLU V4.0, June 2009
  - Same can be done for SuperLU_MT (left-looking, multicore)
- Scalable parallel ILUTP?
  - How to do this with right-looking, multifrontal algorithms?
    e.g., SuperLU_DIST, MUMPS
  - Even lower Arithmetic Intensity than complete LU

References

- X.S. Li and M. Shao, “A Supernodal Approach to Incomplete LU Factorization with Partial Pivoting”, ACM Trans. Math. Software, Vol. 37, No. 4, Article No. 43, 2011.
- Y. Saad, “Iterative methods for sparse linear systems (2nd edition)”, SIAM, 2003. (book)
- M. Benzi, “Preconditioning techniques for large linear systems: a survey”, Journal of Computational Physics 182 (2), 418-477, 2002.
  - Contains 296 references

Exercises

1. Run the ILU + GMRES example in SuperLU_4.3/EXAMPLE/ directory
2. Study the incomplete Cholesky code (CholIC) in Hands-On-Exercises/ directory
   - Implement the subroutine Chol_IC() in the file graph_facto_mod.f90
   - Compile and run your completed program

Lecture 7

Low Rank Approximate Factorizations

Xiaoye Sherry Li
Lawrence Berkeley National Laboratory, USA
xsli@lbl.gov

Introduction

Consider solving large sparse linear system $A\mathbf{u} = \mathbf{b}$ with Gaussian elimination: $A = LU$

- Deliver reliable solution, error bounds, condition estimation, efficient for many RHS, . . .

- **Complexity wall** ... not linear time
  [George '73] For model problems, (exact) sparse LU with best ordering Nested Dissection gives optimal complexity:
  - 2D ($kxk = n$ grids): $\mathcal{O}(n \log n)$ Fill, $\mathcal{O}(n^{3/2})$ Flops

  Fill: adding up the dense submatrices of all the "+" separators:

$$k^2 + 4\left(\frac{k}{2}\right)^2 + 4^2\left(\frac{k}{4}\right)^2 + \ldots = \sum_{i=0} 4^i \left(\frac{k}{2^i}\right)^2 = O(k^2 \log k)$$

  Flops: dominated by cubic term of factorizing top-level separator: $O(k^3)$

Approximation

Exploit "data-sparseness" structure in separators

- data-sparse: matrix may be dense, but has a compressed representation smaller than N^2

Low-rank matrices as basic building blocks

- If B has exact rank at most k:
  - Outer-product form: $B_{m\times n} = U_{m\times k} V^T_{k\times n}, k \leq n$
  - Orthonormal outer-product form:
    $B_{m\times n} = U_{m\times k} X_{k\times k} V^T_{k\times n}, \quad U^T U = V^T V = I_k$
- If A has numerical low rank k (called ε-rank):
  $A = U\Sigma V^T \approx A_k := U\Sigma_k V^T, \Sigma = diag(\sigma_1, \ldots, \sigma_k, \sigma_{k+1}, \ldots, \sigma_n)$
  $\Sigma_k = diag(\sigma_1, \ldots, \sigma_k, 0, \ldots, 0)$, with $\sigma_k > \varepsilon$

Algorithms:

- truncated SVD
- rank-revealing QR
- randomized sampling, ...

Approximations by LR matrices

- Singular Value Decomposition (SVD)
  $A = U\Sigma V^T \approx A_k := U\Sigma_k V^T$
  $\Sigma = diag(\sigma_1, \ldots, \sigma_k, \sigma_{k+1}, \ldots, \sigma_n)$
  $\Sigma_k = diag(\sigma_1, \ldots, \sigma_k, 0, \ldots, 0)$
  - **accuracy**: $\|A - A_k\|_2 = \sigma_{k+1}$
  - **cost**: $O(m^2 n) \quad (m \leq n)$

- Rank-Revealing QR decomposition (RRQR)
  $A\Pi = QR, \quad R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}, \quad \Pi$ permutation matrix
  Choose $U = Q(:, 1:k), V = \Pi[R_{11} \quad R_{12}]^T$
  - **accuracy**: $\|A - UV^T\|_2 = \|R_{22}\|_2 \leq \mathbf{c}\, \sigma_{k+1}$
  - **cost**: $O(kmn) \quad (m \leq n,\ k \approx m)$

LR Matrices (con't)

- Randomized sampling
  1. Pick random matrix $\Omega_{n\times(k+p)}$, p small, e.g. 10
  2. Sample matrix $S = A\Omega$, with slight oversampling p
  3. Compute $Q = \text{ON-basis}(S)$

  - **accuracy:** with *probability* $\geq 1 - 6 \cdot p^{-p}$,
    $\|A - QQ^*A\| \leq [1 + 11\sqrt{k+p} \cdot \sqrt{\min\{m,n\}}]\sigma_{k+1}$
  - **cost:** $O(kmn)$

- **Remarks**
  - Kernel: All have same asymptotic cost with explicit matrix
    - ⋆ RS can be faster when fast matrix-vector available
    - ⋆ RS useful when only matrix-vector available
  - Putting in sparse solver: costs will be different ...

LR Matrices (con't)

- Randomized sampling
  1. Pick random matrix $\Omega_{n\times(k+p)}$, p small, e.g. 10
  2. Sample matrix $S = A\Omega$, with slight oversampling p
  3. Compute $Q = \text{ON-basis}(S)$

  - **accuracy:** with *probability* $\geq 1 - 6 \cdot p^{-p}$,
    $\|A - QQ^*A\| \leq [1 + 11\sqrt{k+p} \cdot \sqrt{\min\{m,n\}}]\sigma_{k+1}$
  - **cost:** $O(kmn)$

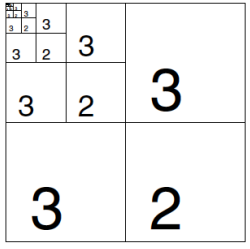
- **Remarks**
  - Kernel: All have same asymptotic cost with explicit matrix
    - ★ RS can be faster when fast matrix-vector available
    - ★ RS useful when only matrix-vector available
  - Putting in sparse solver: costs will be different ...

Data-sparse representations

Hierarchical matrices: $\mathcal{H}$-matrix, $\mathcal{H}^2$-matrix

[Bebendorf, Borm, Grasedyck, Hackbusch, Le Borne, Martinsson, Tygert, ...]

- allow **Fast** matrix-vector multiplication, factorization, inversion, ...
- $\mathcal{H}$-matrix : Given a "suitable" partition $P : I \times J$ of row and column dimensions, ranks of all blocks $A_b \leq k$. (low-rank blocks chosen independently from each other)
  - Example [Bebendorf 2008]: Hilbert matrix $h_{ij} = \frac{1}{i+j-1}$ and the blockwise ranks:
  - Flops of matrix-vector multiplication: $O(k(|I| \log |I| + |J| \log |J|))$



- $\mathcal{H}^2$-matrix is a uniform $\mathcal{H}$-matrix with **nested** cluster bases
  - more restrictive but faster than $\mathcal{H}$-matrix
  - Flops of matrix-vector multiplication: $O(k(|I| + |J|))$ (algebraic generalization of the **Fast Multipole** method)

Data-sparse representations

(Hierarchically) Semi-Separable matrices

[Bini, Chandrasekaran, Dewilde, Eidelman, Gemignani, Gohberg, Gu, Kailath, Olshevsky, van der Veen, Van Barel, Vandebril, White, et al.]

- SS matrix: $S = triu(UV^T) + tril(WZ^T)$, where $U, V, W,$ and Z are rank-k matrices.
  - Example: can be used to represent the inverse of a banded matrix
- HSS matrix: the bases are required to be **nested**
  - special case of $\mathcal{H}^2$-matrix

Other low-rank factorization ideas:

- BLR (Block LR) (Amestoy et al.)
- MLR (Multilevel LR) (Saad et al.)

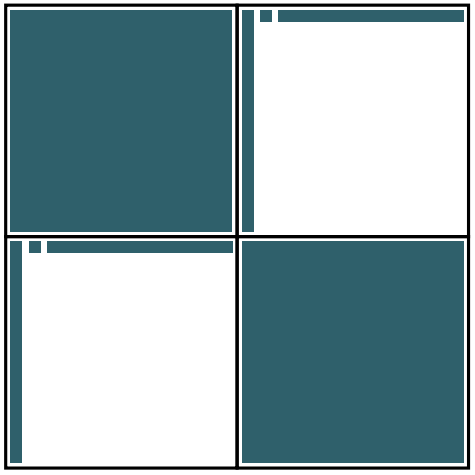
Outline

- How it works operationally?
  - Hierarchical matrix representation, factorization
  - HSS-embedded multifrontal factorization
    - ★ targeting at nonsymetric systems (with PDE behind)
- Theory
  - Schur monotonicity
  - conditioning analysis
  - rank analysis for discretized PDEs
- Practice
  - ordering within separator
  - parallelization
  - preconditioning
- Summary

Hierarchically Semi-Separable matrices

An HSS matrix A is a dense matrix whose off-diagonal blocks are low-rank.
High-level structure: 2×2 blocks

$$A = \left[\begin{array}{c|c} D_1 & U_1 B_1 V_1^T \\ \hline U_2 B_2 V_1^T & D_2 \end{array}\right]$$



Fundamental property required for efficiency: nested bases

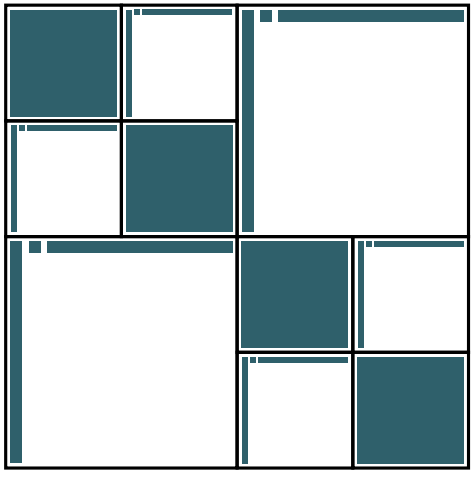
$$U_3 = \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix} U_3^{small}, \; U_3^{small} : 2k \times k$$

Same for U_3, U_6, V_6 and recursively at subsequent levels.

Hierarchically Semi-Separable matrices

An HSS matrix A is a dense matrix whose off-diagonal blocks are low-rank.
Recursion

$$A = \left[\begin{array}{cc|cc} D_1 & U_1 B_1 V_2^T & \multicolumn{2}{c}{\multirow{2}{*}{$U_3 B_3 V_6^T$}} \\ U_2 B_2 V_1^T & D_2 & & \\ \hline \multicolumn{2}{c|}{\multirow{2}{*}{$U_6 B_6 V_3^T$}} & D_4 & U_4 B_4 V_5^T \\ & & U_5 B_5 V_4^T & D_5 \end{array}\right]$$



Fundamental property required for efficiency: nested bases

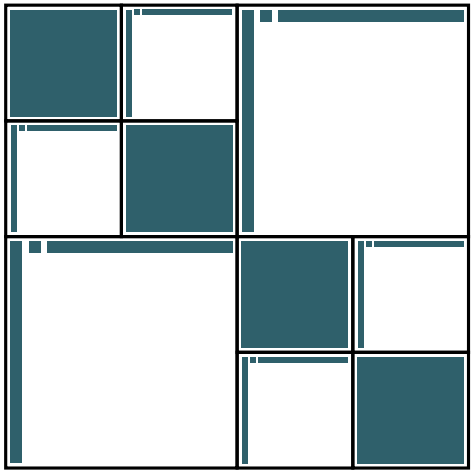
$$U_3 = \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix} U_3^{small}, U_3^{small} : 2k \times k$$

Same for U_3, U_6, V_6 and recursively at subsequent levels.

Hierarchically Semi-Separable matrices

An HSS matrix A is a dense matrix whose off-diagonal blocks are low-rank.
Recursion

$$A = \left[\begin{array}{cc|cc} D_1 & U_1 B_1 V_2^T & \multicolumn{2}{c}{\multirow{2}{*}{$U_3 B_3 V_6^T$}} \\ U_2 B_2 V_1^T & D_2 & & \\ \hline \multicolumn{2}{c|}{\multirow{2}{*}{$U_6 B_6 V_3^T$}} & D_4 & U_4 B_4 V_5^T \\ & & U_5 B_5 V_4^T & D_5 \end{array}\right]$$

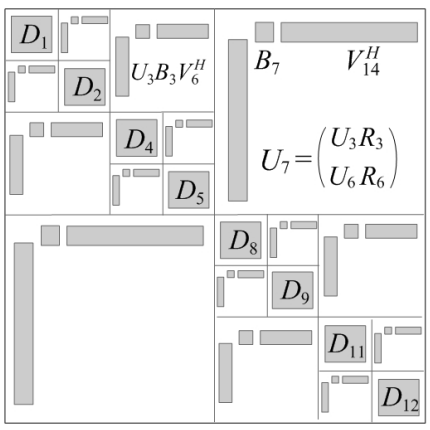


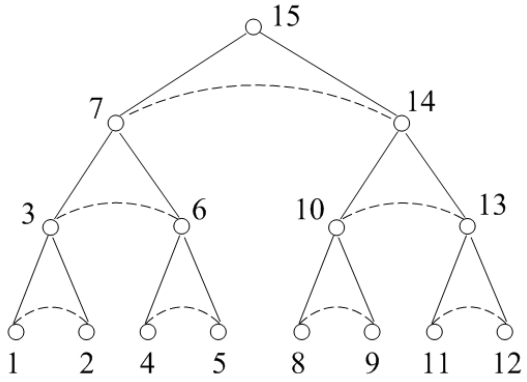
Fundamental property required for efficiency: nested bases

$$U_3 = \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix} U_3^{small}, \; U_3^{small} : 2k \times k$$

Same for U_3, U_6, V_6 and recursively at subsequent levels.

Hierarchical bases, HSS tree





For efficiency, require:

$$U_3 = \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix} U_3^{small}, \; U_3^{small} : 2k \times k, \quad U_6 = \begin{bmatrix} U_4 & 0 \\ 0 & U_5 \end{bmatrix} U_6^{small}, \quad U_6^{small} : 2k \times k$$

$$U_7 = \begin{bmatrix} U_3 & 0 \\ 0 & U_6 \end{bmatrix} U_7^{small}, \quad U_7^{small} : 2k \times k$$

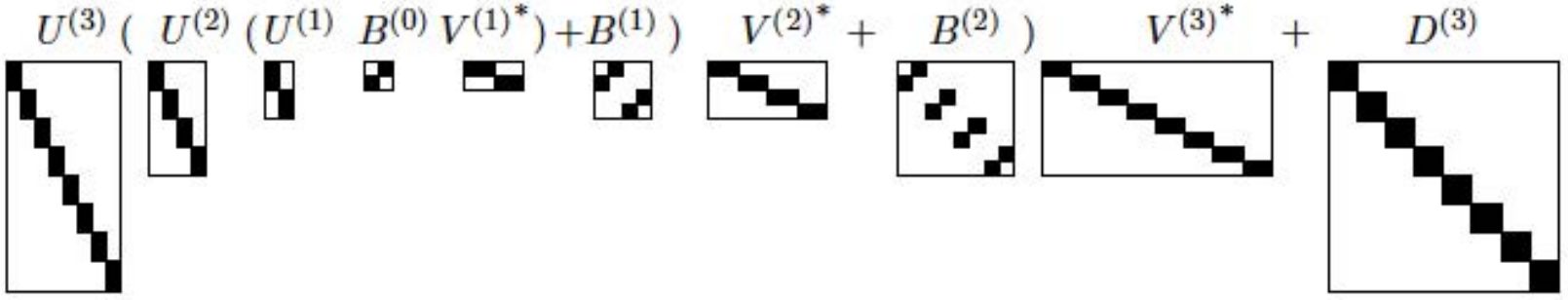
Each basis is a product of descendents' bases:

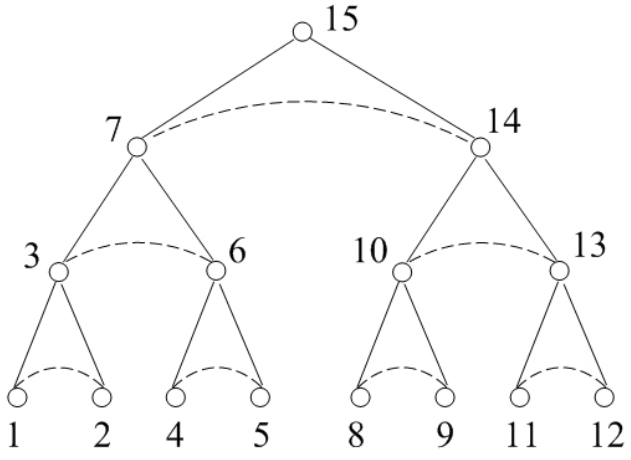
$$U_7 = \begin{bmatrix} U_1 & 0 & 0 & 0 \\ 0 & U_2 & 0 & 0 \\ 0 & 0 & U_4 & 0 \\ 0 & 0 & 0 & U_5 \end{bmatrix} \begin{bmatrix} U_3^{small} & 0 \\ 0 & U_6^{small} \end{bmatrix} U_7^{small},$$

Not to multiply out!

HSS explicit representation (construction)

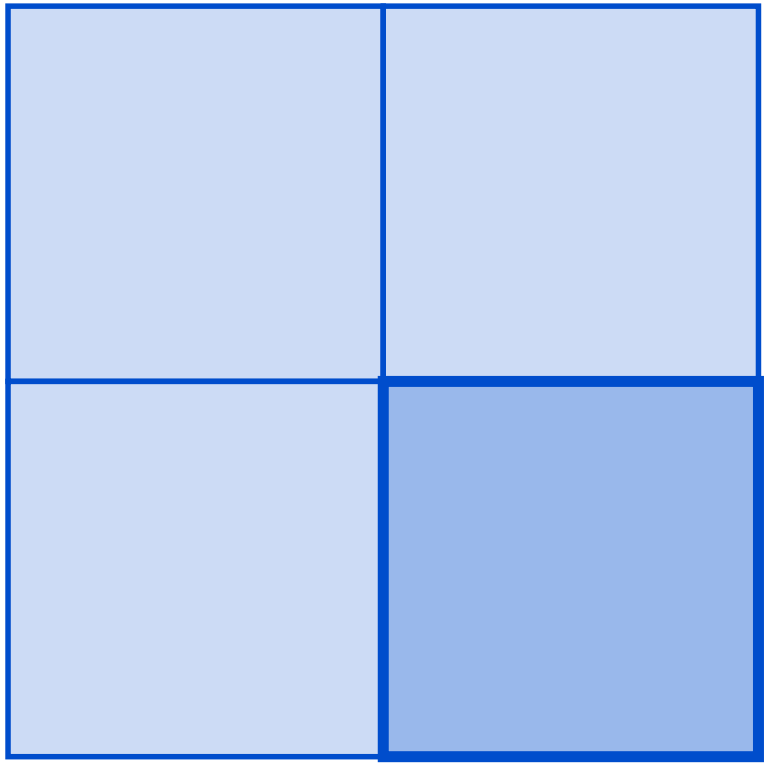
[Martinsson]



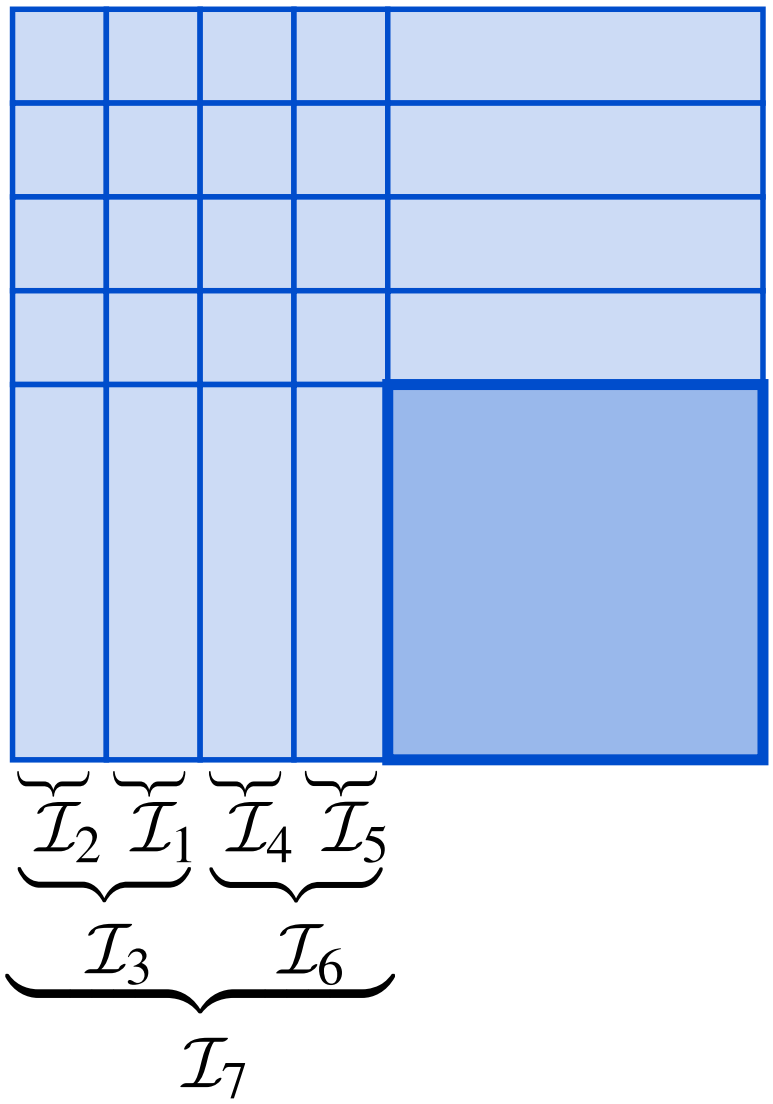


- keep it as an unevaluated product & sum
- operations going up / down the HSS tree

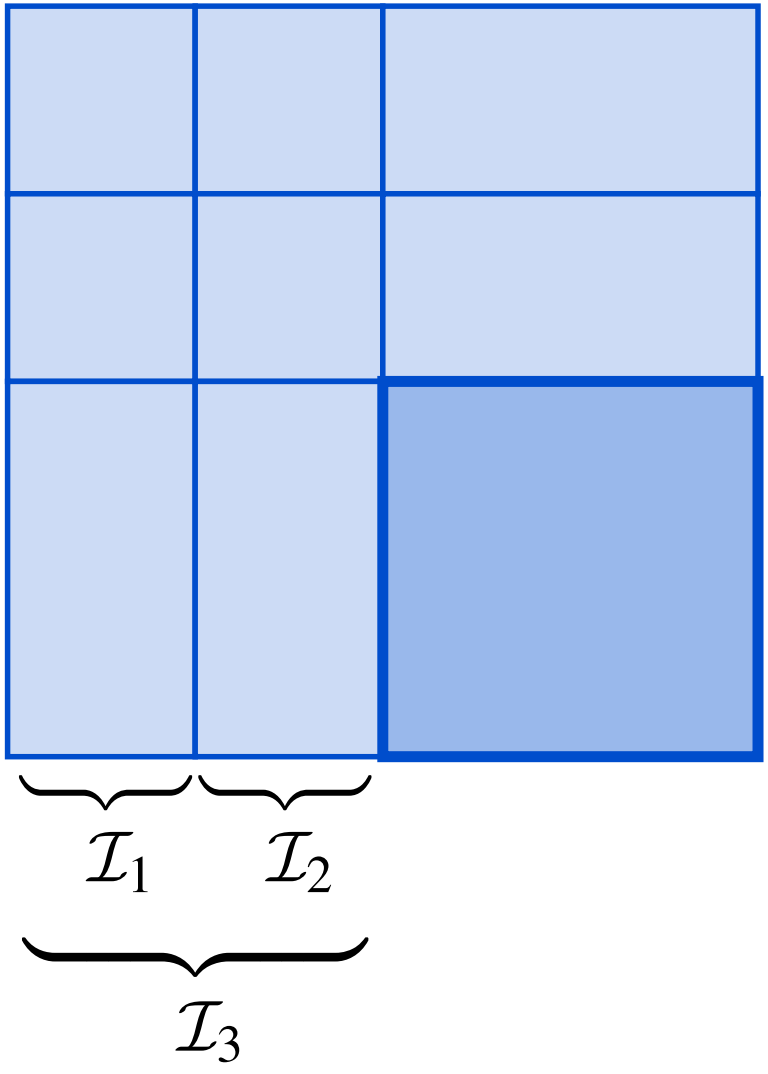
Structured factorization



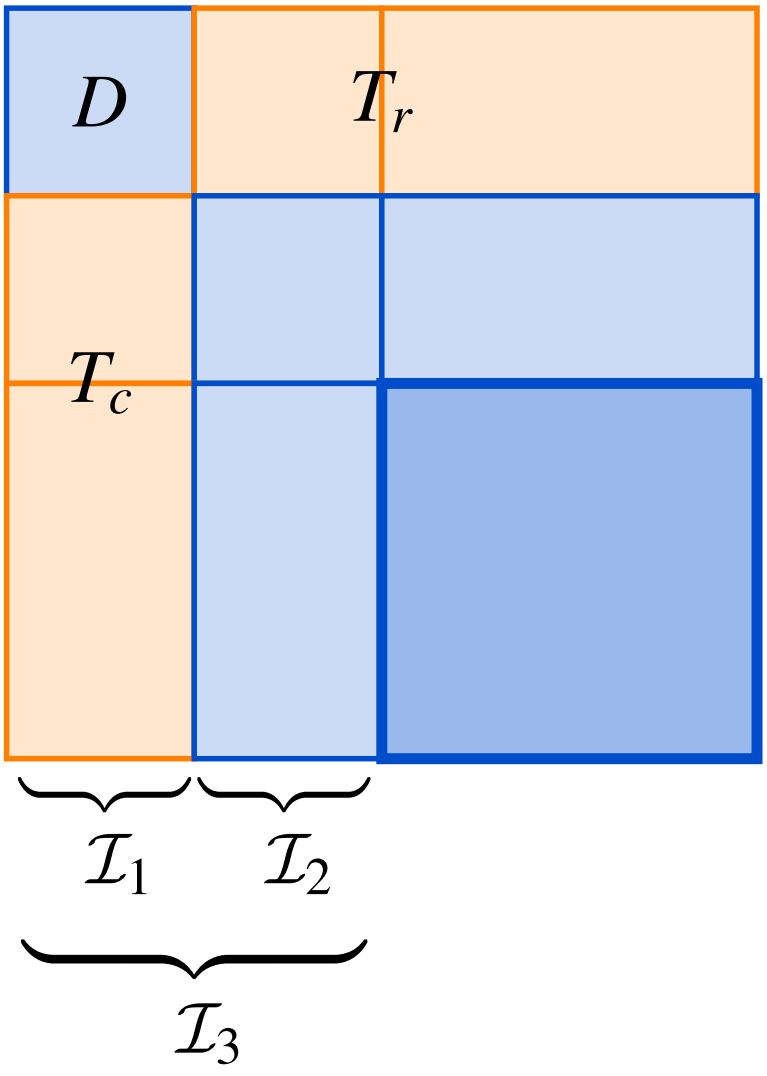
Structured factorization



Structured factorization



Structured factorization



HSS node 1:

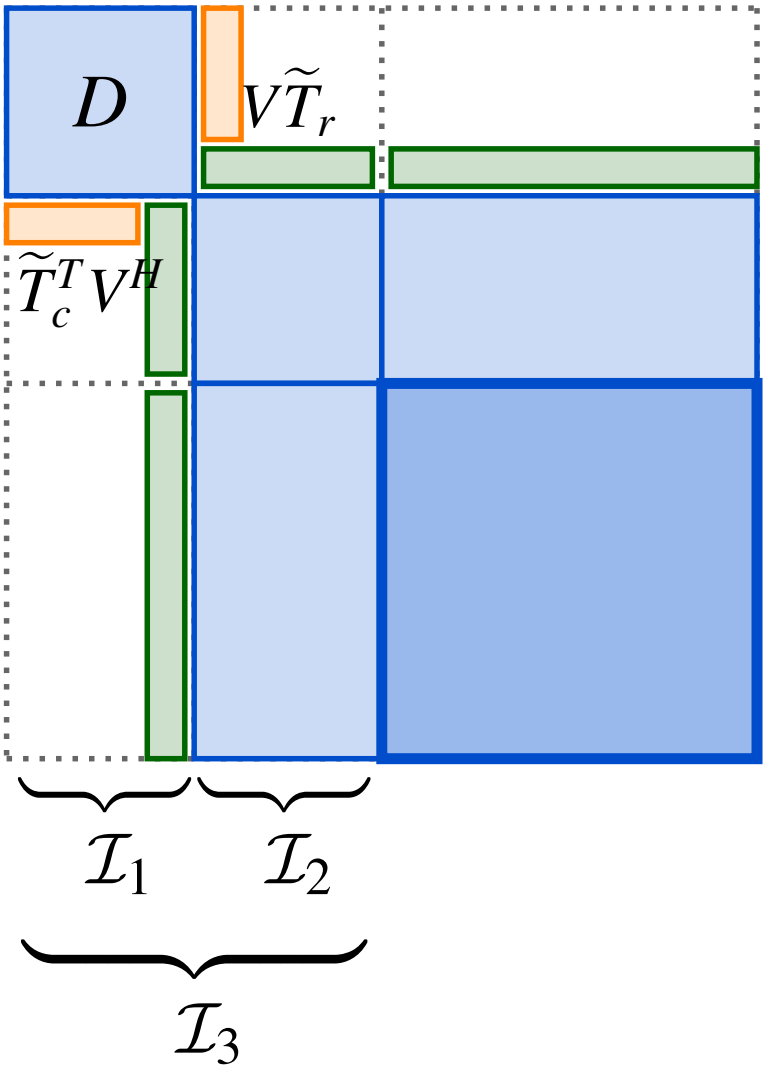
1. Approximate:
$(T_r , T_c^H) \approx V(\widetilde{T}_r , \widetilde{T}_c^H)$

2. Compl. basis with $V_\perp$:
$(V_\perp , V)$ is unitary

3. Change basis:
$\widetilde{D} = (V_\perp , V)^H (V_\perp , V)$

4. Partially factorize $\widetilde{D}$:
$\begin{pmatrix} D_L & \\ D_c & \widetilde{T}_D \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \widetilde{D}$

Structured factorization



HSS node 1:

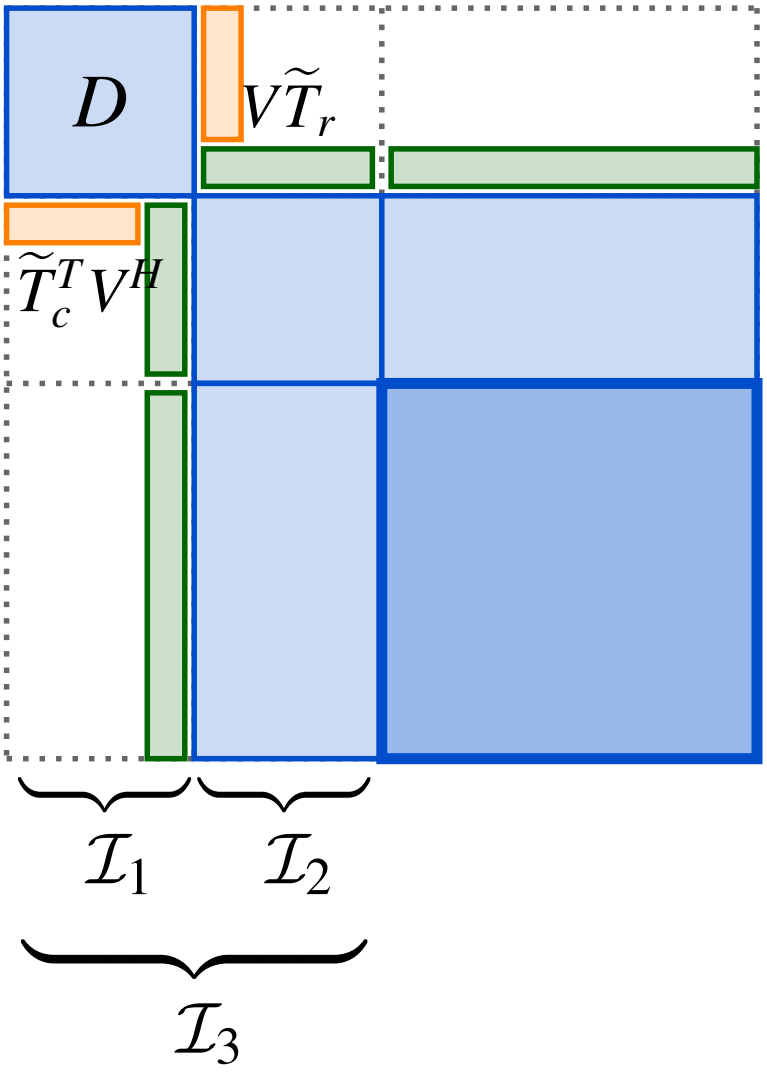
1. Approximate:
$(T_r , T_c^H) \approx V(\widetilde{T}_r , \widetilde{T}_c^H)$

2. Compl. basis with $V_\perp$:
$(V_\perp , V)$ is unitary

3. Change basis:
$\widetilde{D} = (V_\perp , V)^H (V_\perp , V)$

4. Partially factorize $\widetilde{D}$:
$$\begin{pmatrix} D_L & \\ D_c & \widetilde{T}_D \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \widetilde{D}$$

Structured factorization



HSS node 1:

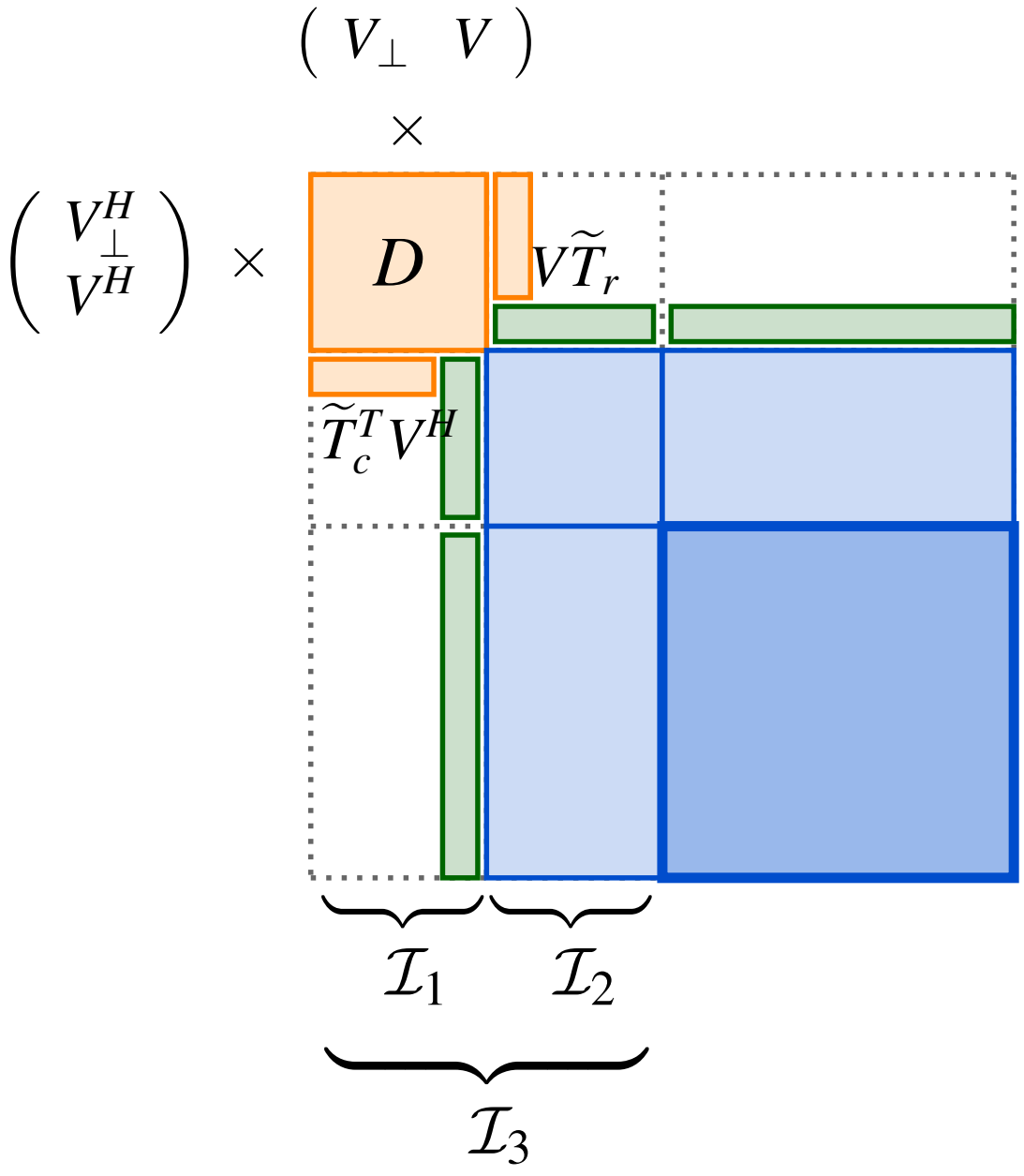
1. Approximate:
$(T_r , T_c^H) \approx V(\widetilde{T}_r , \widetilde{T}_c^H)$

2. Compl. basis with $V_\perp$:
$(V_\perp , V)$ is unitary

3. Change basis:
$\widetilde{D} = (V_\perp , V)^H (V_\perp , V)$

4. Partially factorize $\widetilde{D}$:
$\begin{pmatrix} D_L & \\ D_c & \widetilde{T}_D \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \widetilde{D}$

Structured factorization



HSS node 1:

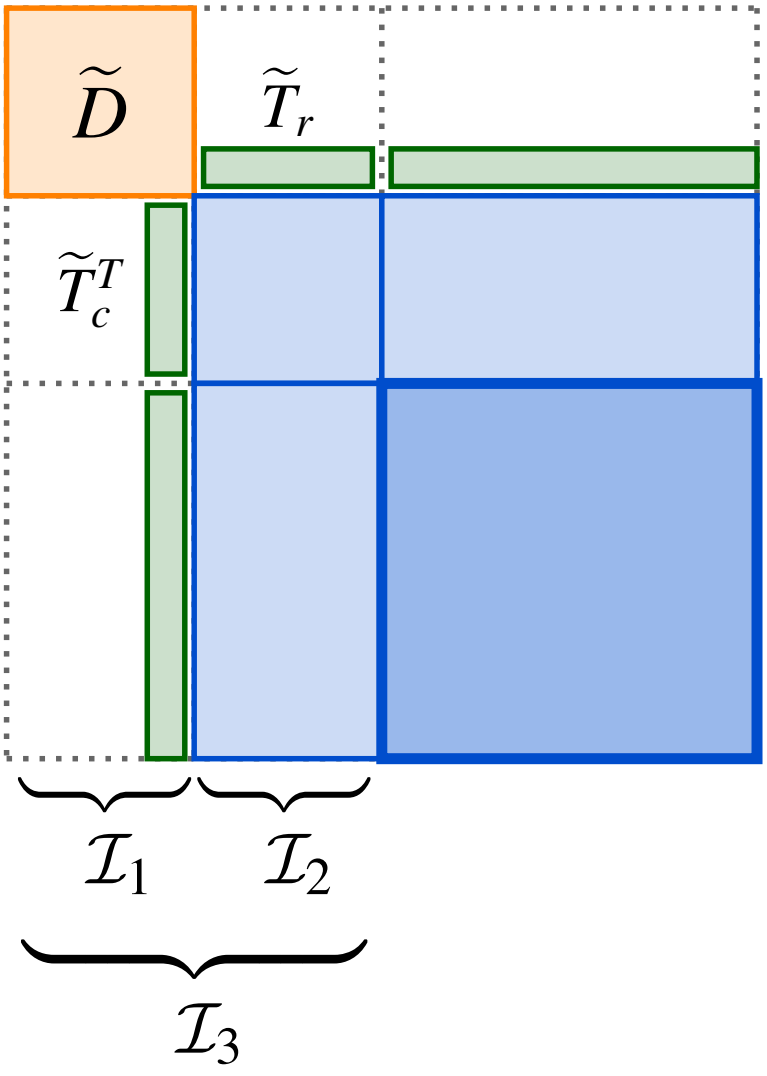
1. Approximate:
$(T_r , T_c^H) \approx V(\widetilde{T}_r , \widetilde{T}_c^H)$

2. Compl. basis with $V_\perp$:
$(V_\perp , V)$ is unitary

3. Change basis:
$\widetilde{D} = (V_\perp , V)^H D (V_\perp , V)$

4. Partially factorize $\widetilde{D}$:
$\begin{pmatrix} D_L & \\ D_c & \widetilde{T}_D \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \widetilde{D}$

Structured factorization



HSS node 1:

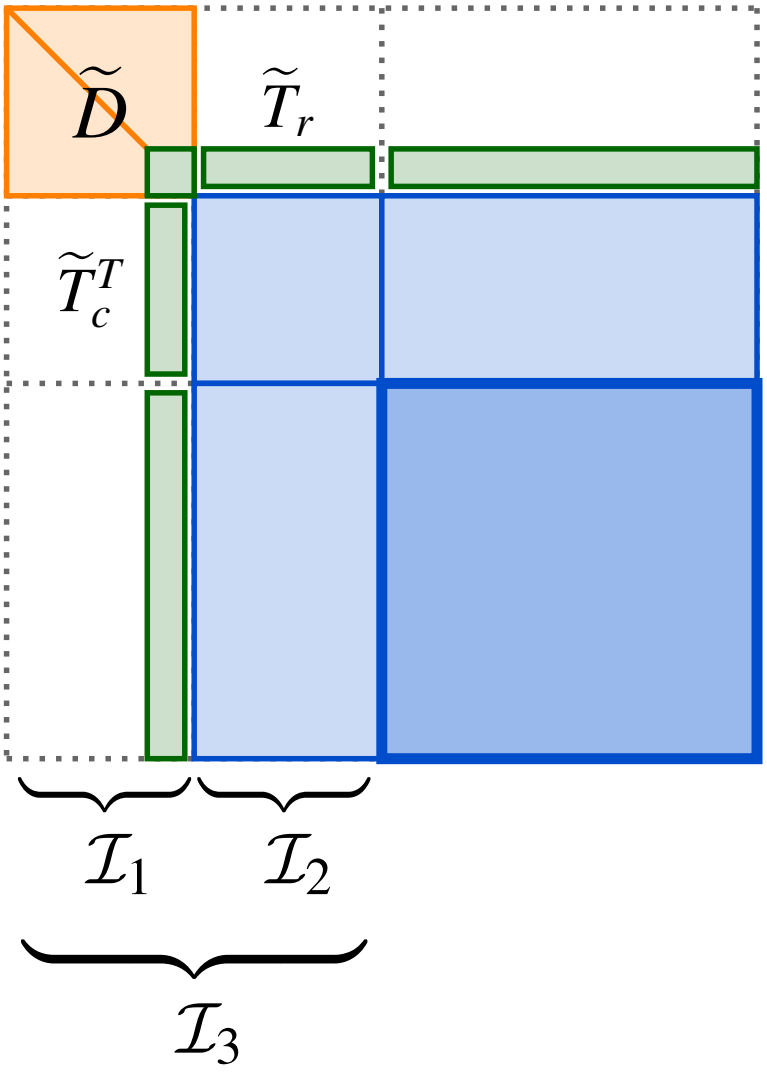
1. Approximate:
$(T_r , T_c^H) \approx V(\widetilde{T}_r , \widetilde{T}_c^H)$

2. Compl. basis with $V_\perp$:
$(V_\perp , V)$ is unitary

3. Change basis:
$\widetilde{D} = (V_\perp , V)^H D (V_\perp , V)$

4. Partially factorize $\widetilde{D}$:
$$\begin{pmatrix} D_L & \\ D_C & \widetilde{T}_D \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \widetilde{D}$$

Structured factorization



HSS node 1:

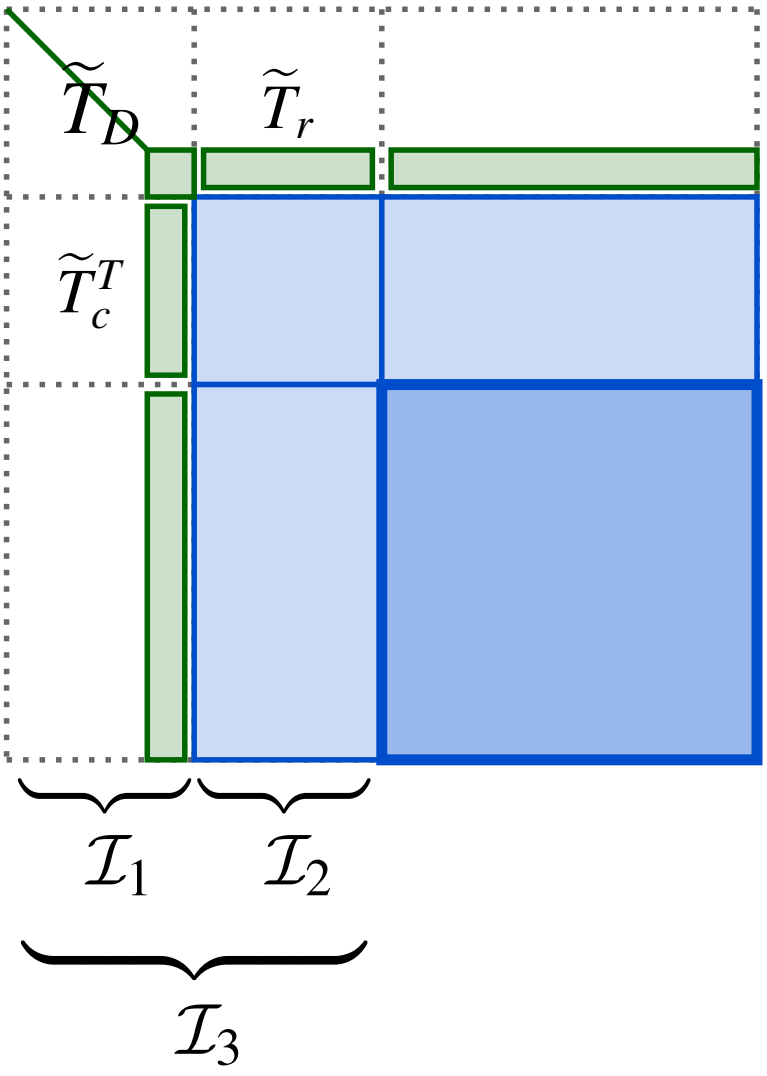
1. Approximate:
$(T_r , T_c^H) \approx V(\widetilde{T}_r , \widetilde{T}_c^H)$

2. Compl. basis with $V_\perp$:
$(V_\perp , V)$ is unitary

3. Change basis:
$\widetilde{D} = (V_\perp , V)^H D (V_\perp , V)$

4. Partially factorize $\widetilde{D}$:
$$\begin{pmatrix} D_L & \\ D_c & \widetilde{T}_D \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \widetilde{D}$$

Structured factorization



HSS node 1:

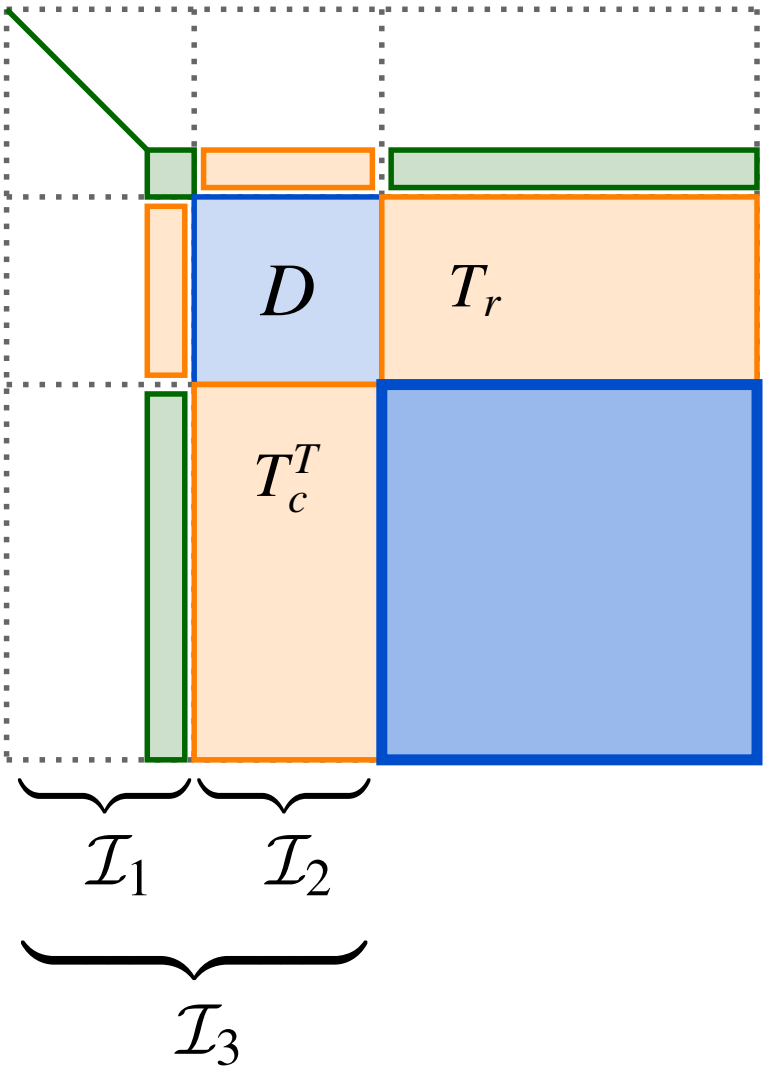
1. Approximate:
$(T_r , T_c^H) \approx V(\widetilde{T}_r , \widetilde{T}_c^H)$

2. Compl. basis with $V_\perp$:
$(V_\perp , V)$ is unitary

3. Change basis:
$\widetilde{D} = (V_\perp , V)^H D (V_\perp , V)$

4. Partially factorize $\widetilde{D}$:
$$\begin{pmatrix} D_L & \\ D_c & \widetilde{T}_D \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \widetilde{D}$$

Structured factorization



HSS node 2:

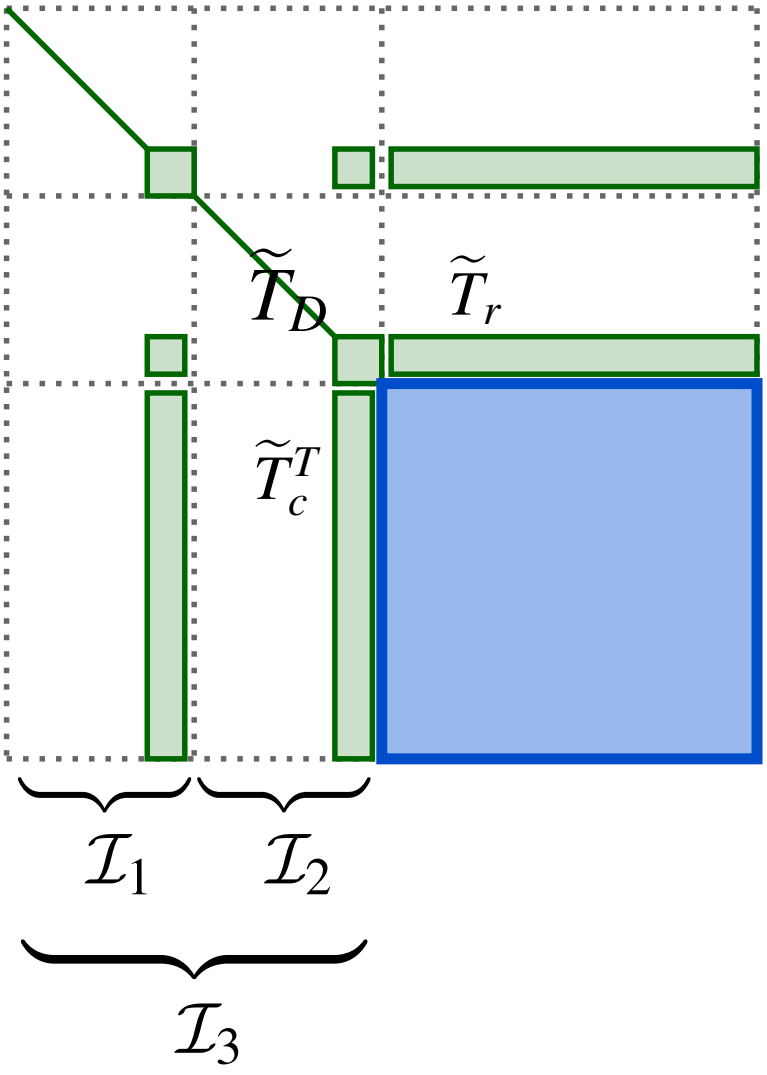
1. Approximate:
$(T_r , T_c^H) \approx V(\widetilde{T}_r , \widetilde{T}_c^H)$

2. Compl. basis with $V_\perp$:
$(V_\perp , V)$ is unitary

3. Change basis:
$\widetilde{D} = (V_\perp , V)^H D (V_\perp , V)$

4. Partially factorize $\widetilde{D}$:
$$\begin{pmatrix} D_L & \\ D_c & \widetilde{T}_D \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \widetilde{D}$$

Structured factorization



HSS node 2:

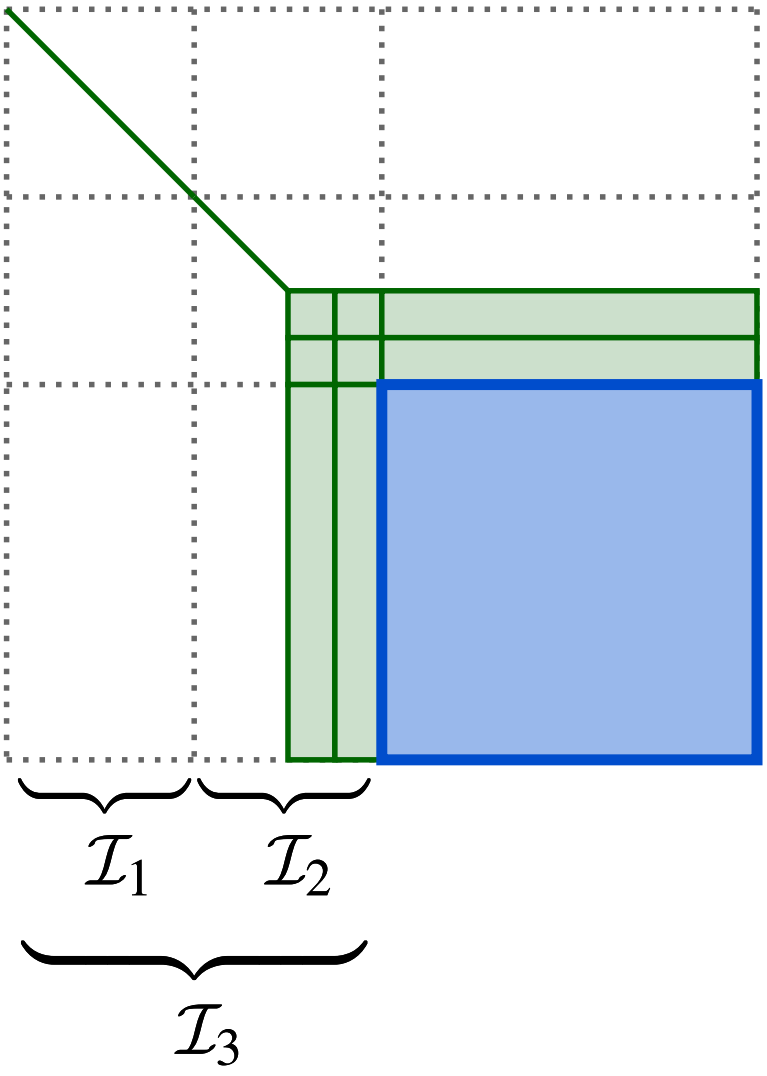
1. Approximate:
$(T_r , T_c^H) \approx V(\widetilde{T}_r , \widetilde{T}_c^H)$

2. Compl. basis with $V_\perp$:
$(V_\perp , V)$ is unitary

3. Change basis:
$\widetilde{D} = (V_\perp , V)^H D (V_\perp , V)$

4. Partially factorize $\widetilde{D}$:
$$\begin{pmatrix} D_L & \\ D_c & \widetilde{T}_D \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \widetilde{D}$$

Structured factorization



HSS node 2:

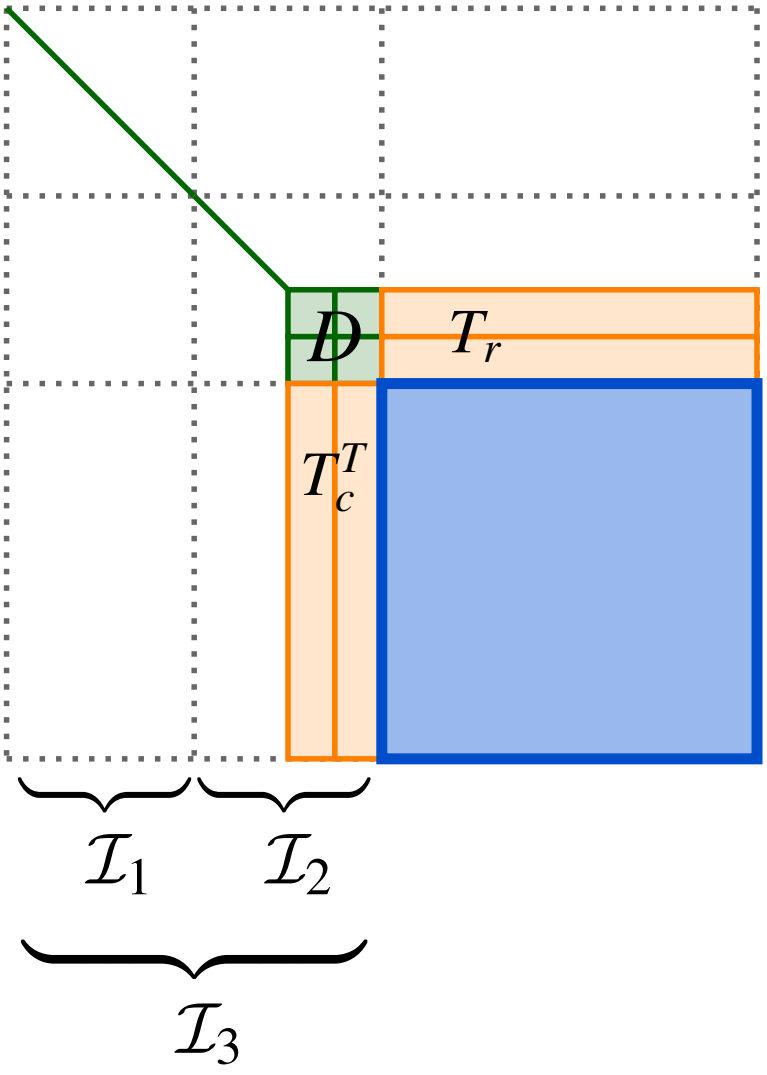
1. Approximate:
$(T_r , T_c^H) \approx V(\widetilde{T}_r , \widetilde{T}_c^H)$

2. Compl. basis with $V_\perp$:
$(V_\perp , V)$ is unitary

3. Change basis:
$\widetilde{D} = (V_\perp , V)^H D (V_\perp , V)$

4. Partially factorize $\widetilde{D}$:
$$\begin{pmatrix} D_L & \\ D_c & \widetilde{T}_D \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \widetilde{D}$$

Structured factorization



HSS node 3:

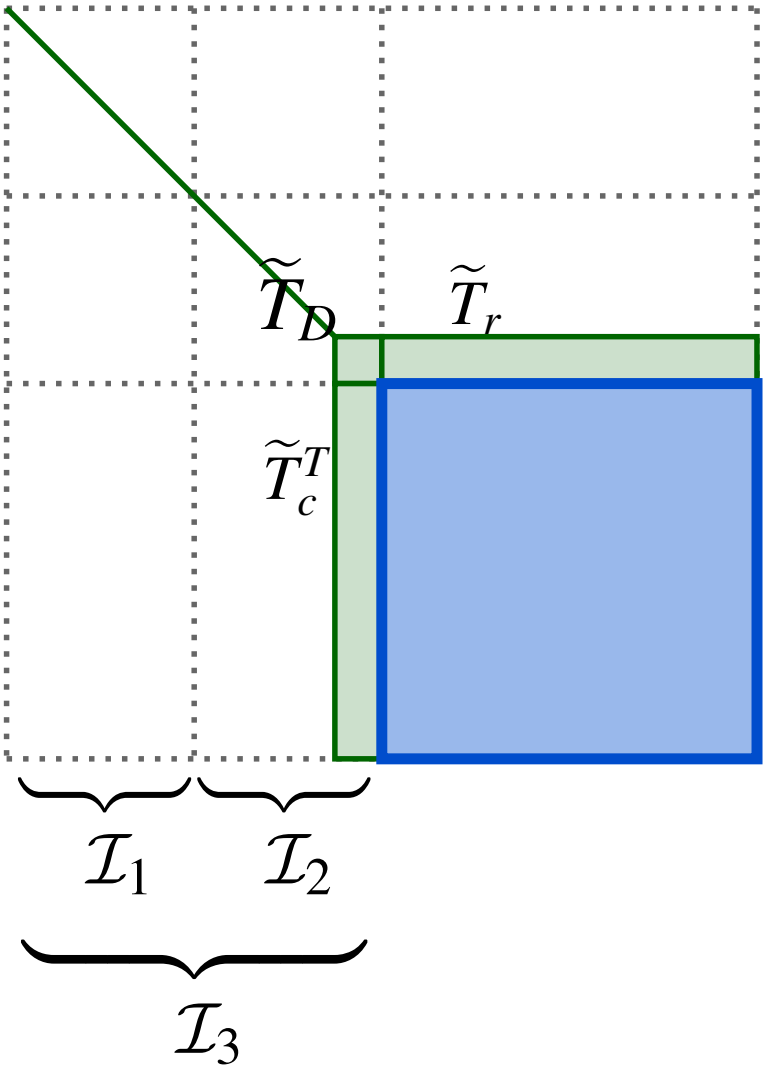
1. Approximate:
$(T_r , T_c^H) \approx V(\widetilde{T}_r , \widetilde{T}_c^H)$

2. Compl. basis with $V_\perp$:
$(V_\perp , V)$ is unitary

3. Change basis:
$\widetilde{D} = (V_\perp , V)^H D (V_\perp , V)$

4. Partially factorize $\widetilde{D}$:
$$\begin{pmatrix} D_L & \\ D_c & \widetilde{T}_D \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \widetilde{D}$$

Structured factorization



HSS node 3:

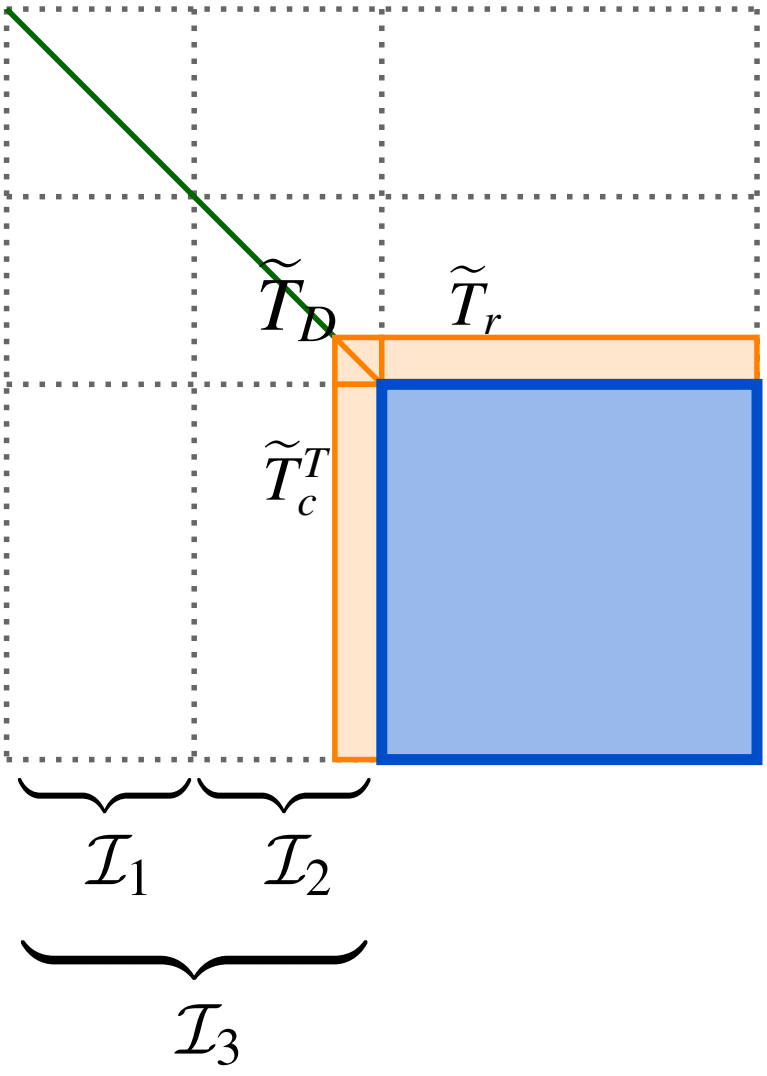
1. Approximate:
$(T_r , T_c^H) \approx V(\widetilde{T}_r , \widetilde{T}_c^H)$

2. Compl. basis with $V_\perp$:
$(V_\perp , V)$ is unitary

3. Change basis:
$\widetilde{D} = (V_\perp , V)^H D (V_\perp , V)$

4. Partially factorize $\widetilde{D}$:
$$\begin{pmatrix} D_L & \\ D_c & \widetilde{T}_D \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \widetilde{D}$$

Structured factorization



HSS node 3:

5. Partially factorize:

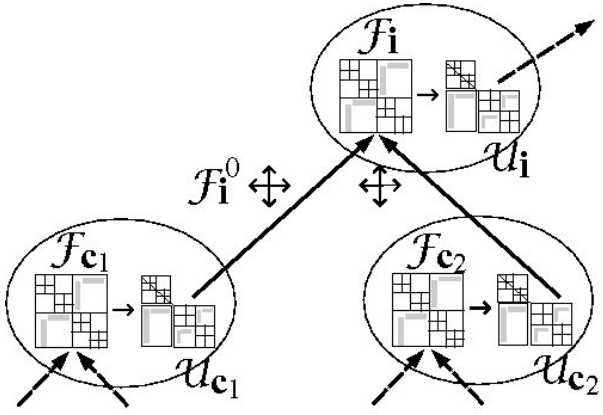
$$\begin{pmatrix} D_L & \\ D_c & \mathcal{U} \end{pmatrix} \begin{pmatrix} D_U & D_r \\ & I \end{pmatrix} = \begin{pmatrix} \widetilde{T}_D & \widetilde{T}_r \\ \widetilde{T}_c & \widetilde{T}_U \end{pmatrix}$$

Embedding HSS in multifrontal

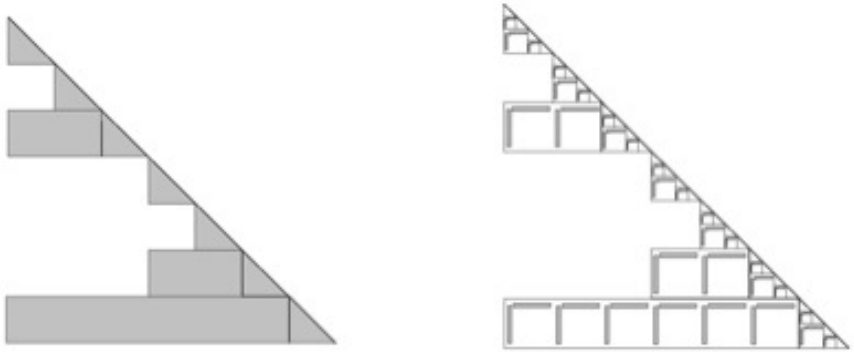
Approximate **Frontal** & **Update** matrices by HSS

Need following operations:

- **frontal HSS factorization of** F_i
- **extend-add of two HSS update matrices** U_i **and** U_j



Final Cholesky factor: Classical vs HSS-embedded



Theory

- Schur monotonicity for approximate Cholesky factorization
- conditioning analysis
- rank analysis for discretized PDEs

Schur monotonicity for approximate Cholesky $A = R^T R$

$$R = \begin{pmatrix} R_{1,1} & R_{1,2} & \cdots & R_{1,n} \\ & R_{2,2} & \cdots & R_{2,n} \\ & & \ddots & \vdots \\ & & & R_{n,n} \end{pmatrix}, \quad R \approx \widetilde{R} = \begin{pmatrix} R_1 & \tilde{R}_{1,2} & \cdots & \tilde{R}_{1,n} \\ & R_2 & \cdots & \tilde{R}_{2,n} \\ & & \ddots & \vdots \\ & & & R_n \end{pmatrix}$$

First approximation step: $R_1^T R_1 = A_{11}$ and $H_1 = D_1^{-T} A_{1,2:n}$

$$H_1 = \begin{pmatrix} U_1 & \widehat{U}_1 \end{pmatrix} \begin{pmatrix} \mathcal{Q}_1 & \widehat{\mathcal{Q}}_1 \end{pmatrix}^T, \quad H_1^T H_1 = \mathcal{Q}_1 \mathcal{Q}_1^T + \widehat{\mathcal{Q}}_1 \widehat{\mathcal{Q}}_1 e^T, \quad \|\widehat{\mathcal{Q}}_1\|_2 \leq \tau$$

Orthogonal dropping: $\widetilde{H}_1 = U_1 \mathcal{Q}_1 \longrightarrow \widetilde{H}_1^T (H_1 - \widetilde{H}_1) = 0$

Schur complement: $\mathcal{A}_1 = A_{2:n,2:n} - H_1^T H_1 = A_{2:n,2:n} - \mathcal{Q}_1 \mathcal{Q}_1^T - \widehat{\mathcal{Q}}_1 \widehat{\mathcal{Q}}_1^T$.

Approximate $\mathcal{A}_1$ by $\widetilde{\mathcal{A}}_1 = A_{2:n,2:n} - \mathcal{Q}_1 \mathcal{Q}_1^T = \mathcal{A}_1 + \widehat{\mathcal{Q}}_1 \widehat{\mathcal{Q}}_1^T = \mathcal{A}_1 + O(\tau^2)$

Nice Property: Successive Schur complements do not decrease in SPD sense
$\Rightarrow$ factorization is breakdown free

1. M. Gu, X.S. Li, P. Vassilevski, "Direction-Preserving and Schur-Monotonic Semiseparable Approximations of Symmetric Positive Definite Matrices", SIMAX, 31 (5), 2650-2664, 2010.
2. J. Xia, M. Gu, "Robust approximate Cholesky factorization of rank-structured symmetric positive definite matrices", SIMAX, 31 (5), 2899-2920, 2010.

Conditioning analysis when $R^T R$ as preconditioner (Napov)

- Goal: analyze spectral condition number $\kappa(R^{-T}AR^{-1})$
- Sketch: look at approximation after each step k of total l step; capture different approximation order:

$$B_k = \begin{pmatrix} \mathcal{R}_{11}^{(k)^T} & \\ \widetilde{\mathcal{R}}_{12}^{(k)^T} & \widetilde{S}_B^{(k)} \end{pmatrix} \begin{pmatrix} \mathcal{R}_{11}^{(k)} & \widetilde{\mathcal{R}}_{12}^{(k)} \\ & I \end{pmatrix}, \quad \widetilde{S}_B^{(k)} = A_{i_k+1:n,i_k+1:n} - \widetilde{\mathcal{R}}_{12}^{(k)^T}\widetilde{\mathcal{R}}_{12}^{(k)}$$

- SSS bound (sequential order):

$$\kappa(R^{-T}AR^{-1}) \leq \prod_{k=1}^{l} \frac{1+\gamma_k}{1-\gamma_k}, \quad \text{where } \gamma_k = \|(\mathcal{R}_{12}^{(k)} - \widetilde{\mathcal{R}}_{12}^{(k)})\widetilde{S}_B^{(k)^{-1/2}}\| < 1$$

- HSS bound: can be computed numerically using good estimates
- γ_k estimate: $\gamma_k \leq \left\|(\mathcal{R}_{12}^{(k)} - \widetilde{\mathcal{R}}_{12}^{(k)}\right\| \|A^{-1}\|^{1/2}$
  - can estimate $\|A^{-1}\|$ with a few iterations of Conjugate Gradient
- Adaptive threshold strategy based on γ_k estimate

A. Napov, "Conditioning analysis of incomplete Cholesky factorizations with orthogonal dropping", SIMAX, Vol. 34, No. 3, 1148-1173, 2013.

Rank analysis for some PDEs

Consider $n \times m$ grid, lexicographical (layer-by-layer) order gives rise to block tridiagonal matrix:

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & & & \\ A_{2,1} & A_{2,2} & \ddots & & \\ & \ddots & \ddots & A_{m-1,m} & \\ & & A_{m,m-1} & A_{m,m} \end{pmatrix}$$

, each $n \times n$ Schur compl. $S_{i+1} = A_{i,i} - A_{i,i-1} S_i^{-1} A_{i,i-1}$

Model problem (Chandrasekaran et al.):

$A_{i,i} = A_{j,j}, A_{i-1,i} = A_{j-1,j}, A_{i,i-1} = A_{j,j-1}$

- In 2D, ε-rank of the off-diagonal Hankel block is constant, for $n \to \infty$
- In 3D (k^3), ε-rank of the strip Hankel block is bounded by $O(k)$

Helmholtz equations with constant velocity (Engquist,Ying):

look at the Green's function of the Helmholtz operator.

- In 2D (k^2), ε-rank of the off-diagonal block bounded by $O(\log k)$
- In 3D (k^3), $O(k)$

1. S. Chandrasekaran, P. Dewilde, M. Gu, and N. Somasunderam, "On the Numerical Rank of the Off-Diagonal Blocks of Schur Complements of Discretized Elliptic PDEs", SIMAX, 31 (5), 2261-2290, 2010.
2. B. Engquist, L. Ying, "Sweeping preconditioner for the Helmholtz equation: Hierarchical matrix representation", Communications in Pure and Applied Mathematics 64 (2011).

Complexity of HSS-embedded multifrontal factorization

- With ND order, the intermediate Schur complements my have slightly higher ranks, but no more than twice:

$$A = \begin{pmatrix} A_{11} & & A_{33} \\ & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}, \quad \text{Schur compl. } S = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{32}$$

  Each contribution $-A_{31}A_{11}^{-1}A_{13}$ (or $-A_{32}A_{22}^{-1}A_{32}$) satisfies the off-diagonal rank bound, together, the off-diagonal rank bound of S is at most twice as that of layer-by-layer order.
- Given the rank bounds, can show the following cost of the HSS-MF factorization algorithm:

Problem		r	MF		HSS-MF	
			flops	fill	flops	fill
2D (k^2)	Elliptic	$O(1)$	$O(n^{3/2})$	$O(n \log n)$	$O(n \log n)$	$O(n \log \log n)$
	Helmholtz	$O(\log k)$				
3D (k^3)	Elliptic	$O(k)$	$O(n^2)$	$O(n^{4/3})$	$O(n^{4/3} \log n)$	$O(n \log n)$
	Helmholtz	$O(k)$				

J. Xia, "Efficient structured multifrontal factorization for general large sparse matrices", SISC, 35 (2), A832-A860, 2012.

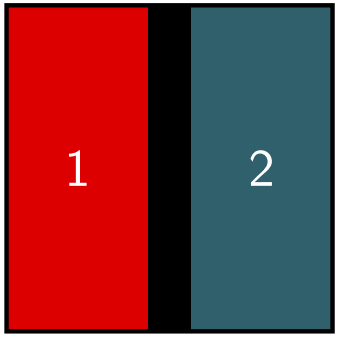
Warning: The constant prefactor may be large: $\sim O(100s)$ ($\sim$ 10 for classical)

Practice

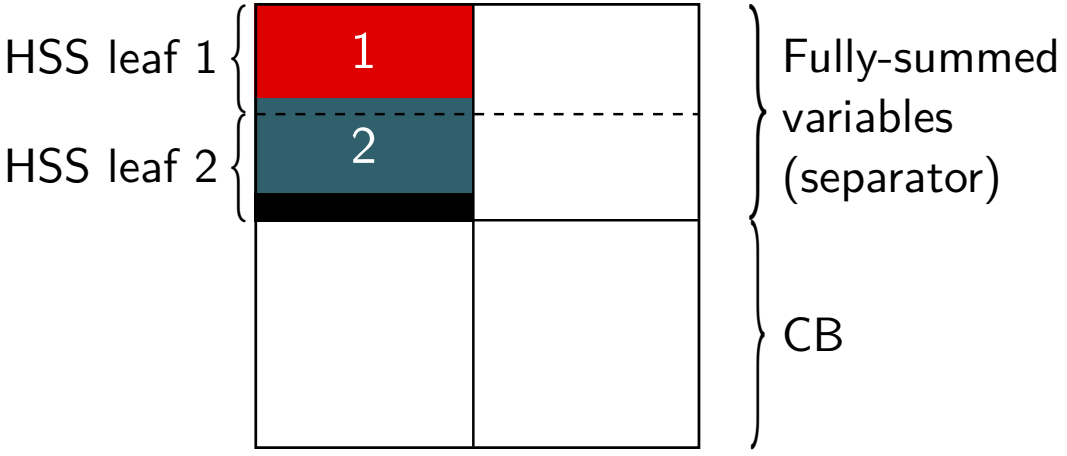
- ordering within separator
- parallelization
- preconditioning

Separator ordering: vertex-based vs edge-based

Vertex-based approach:

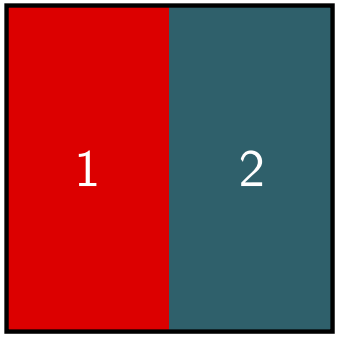


2D separator from a 3D domain.

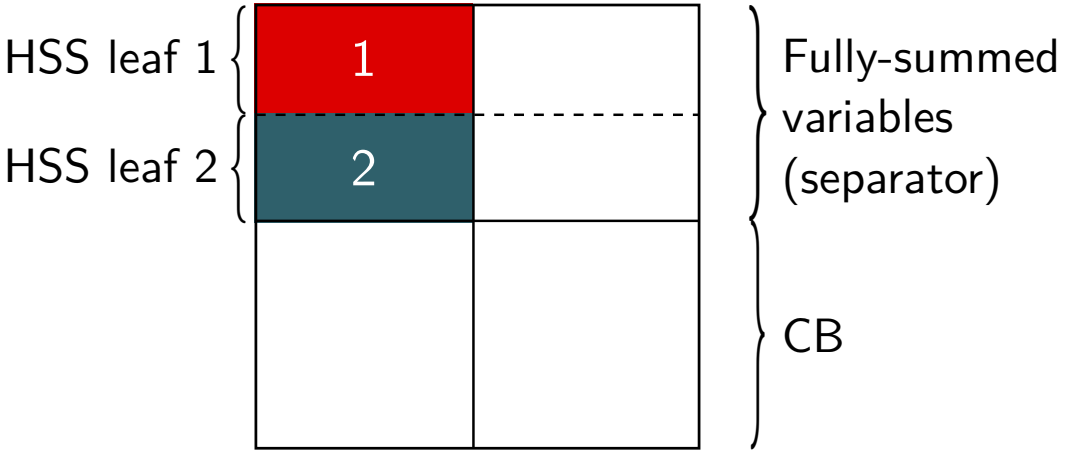


Frontal matrix.

Edge-based approach:



2D separator from a 3D domain.

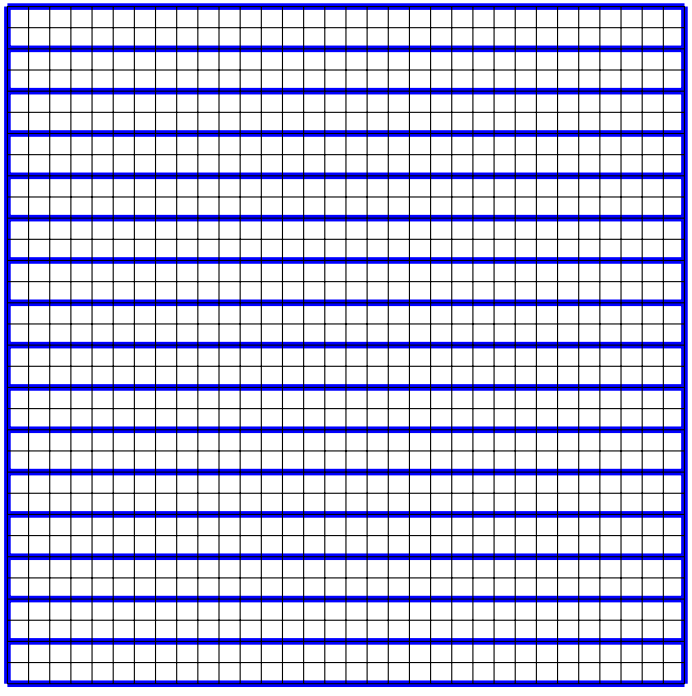


Frontal matrix.

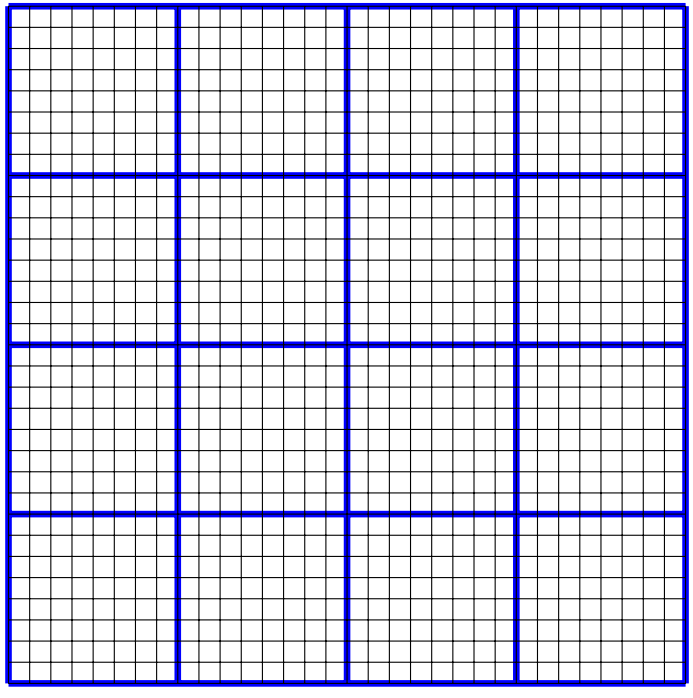
An edge-based ordering allows us to simply match parts of the separators with nodes of the HSS tree.

Ordering of separators: shape

In order to ensure some kind of admissibility condition, parts should have a small diameter.



Large diameters.



Small diameters.

For simplicity, we divide the separator into square blocks (chessboard).

Ordering of separators: ordering of blocks

In the HSS compression stage, blocks are merged two-by-two following a tree flow. Merged blocks should also have small diameter, thus the partioning should have some recursive property.

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

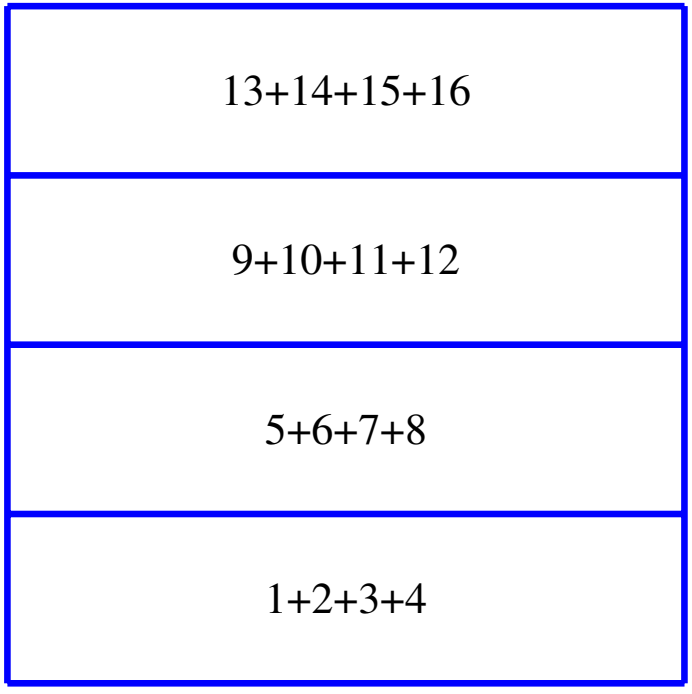
10	11	15	16
9	10	13	14
3	4	7	8
1	2	5	6

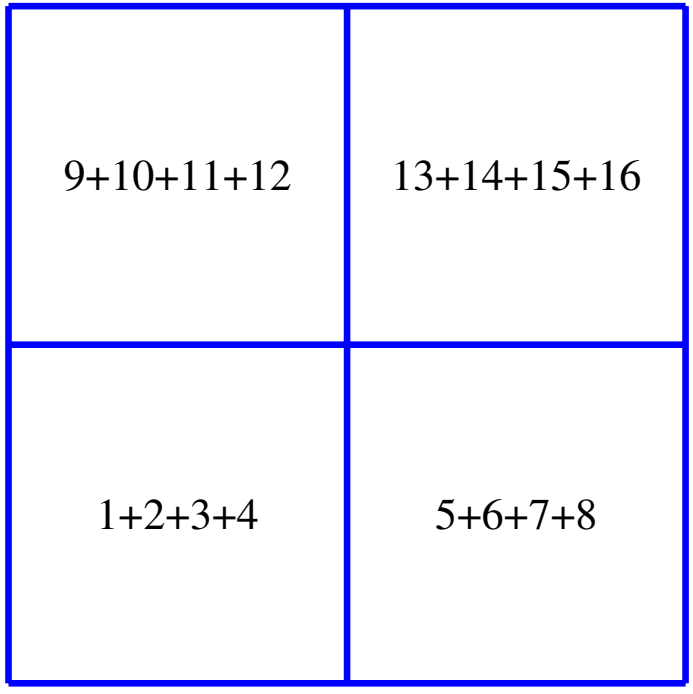
Leaf level.

We use an edge-based Nested Dissection (we cut the domain into squares, and order these squares using ND/Morton ordering).

Ordering of separators: ordering of blocks

In the HSS compression stage, blocks are merged two-by-two following a tree flow. Merged blocks should also have small diameter, thus the partioning should have some recursive property.





Two levels above leaves: blocks are merged four-by-four.

We use an edge-based Nested Dissection (we cut the domain into squares, and order these squares using ND/Morton ordering).

Results - topmost separator

Topmost separator of a 200^3 domain (200×200 plane). We compare:

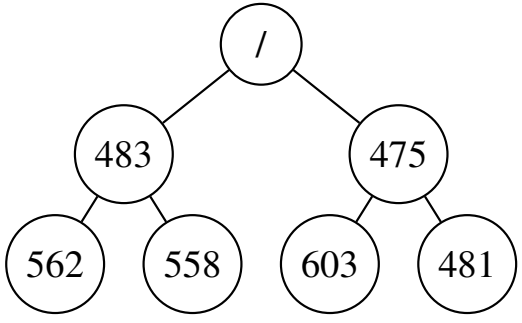
VND: vertex-based ND.

Nat: square blocks ordering in natural/lexicographic order.

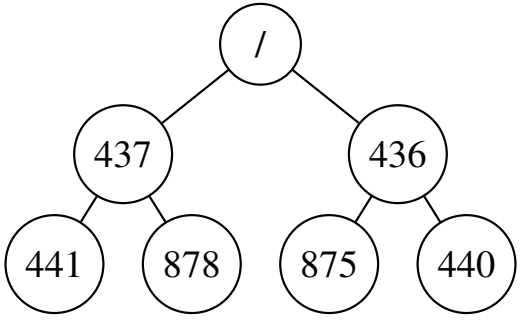
END: edge-based ND: square blocks ordered in ND.

	VND	Nat	END
Total HSS time (s)	55.0	51.8	32.3
Max rank	731	893	646
Min time in RRQR (s)	15.2	20.3	11.0
Max time in RRQR (s)	53.0	50.2	30.7

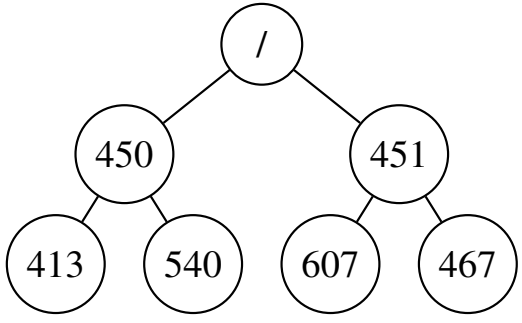
Ranks at the top of HSS trees:



VND.



Nat.



END.

END yields lower rank and better balance of ranks.

Results - complete problem

Helmholtz equations with PML boundary

$$\left(-\Delta - \frac{\omega}{v(x)^2}\right) u(x,\omega) = s(x,\omega)$$

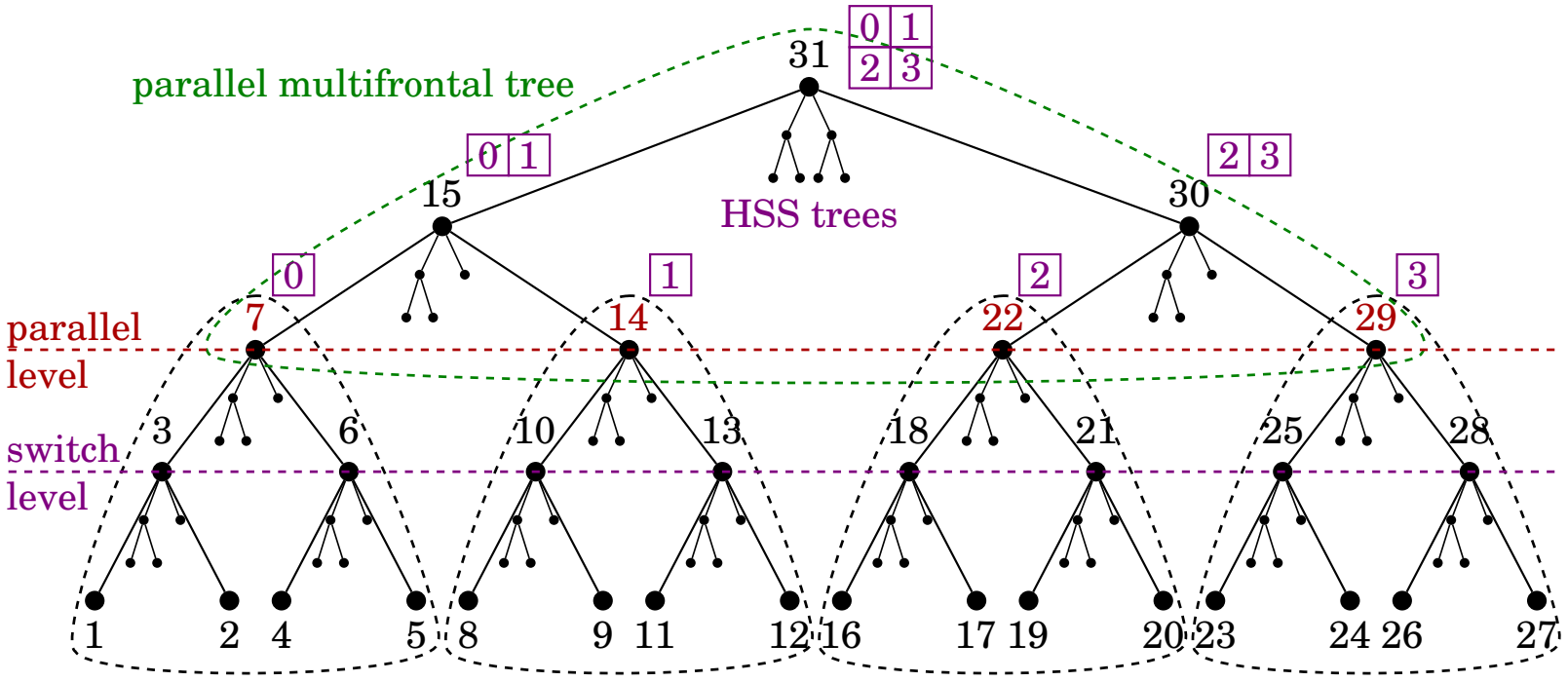
On the complete problem, with 256 cores and HSS compression at the 8 topmost levels of the tree:

	VND	Nat	END
Total factorization time (s)	984.8	978.5	938.0
Max rank	865	893	868
Min time in RRQR (s)	304.8	322.1	310.9
Max time in RRQR (s)	674.9	683.8	654.7

END: marginal (5%) improvement in run time but better workload balance, so hopefully more potential for strong scaling.

Parallelization: two types of tree-based parallelism

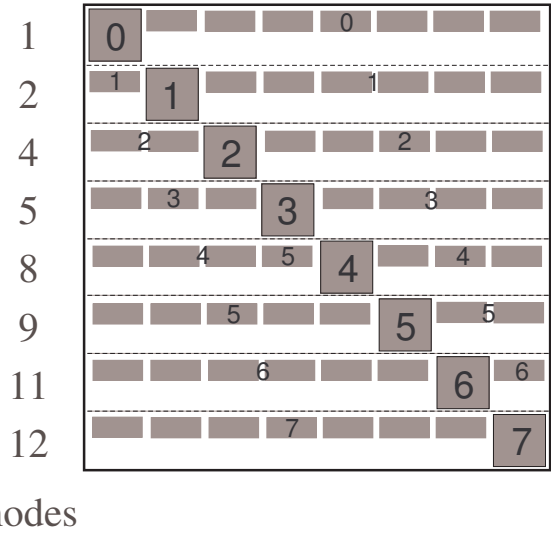
- **Outer tree**: separator tree for multifrontal factorization
- **Inner tree**: HSS tree at each internal separator node



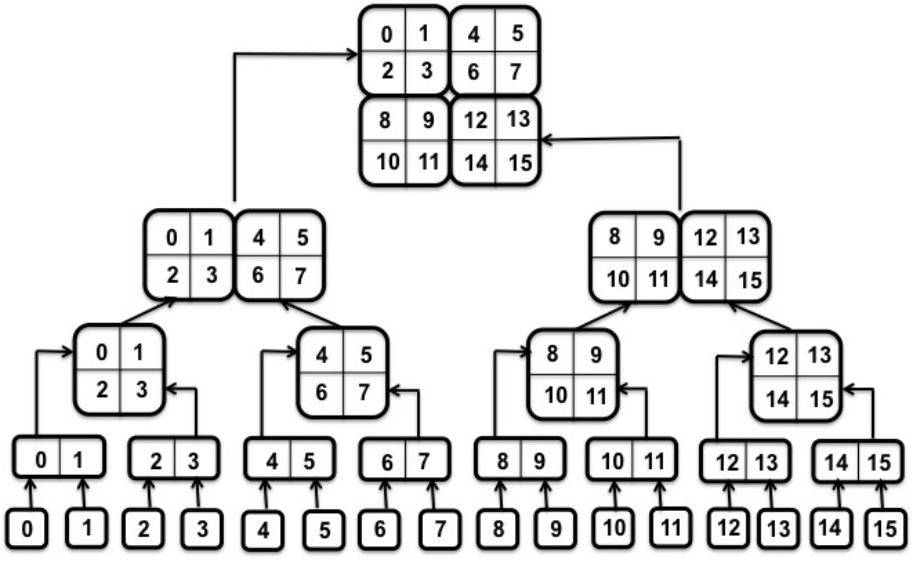
Parallelization strategy for HSS

- Work along the HSS tree level-wise, bottom up.
- 2D block-cyclic distribution at each tree node ($\#Levels = \log P$)
  - each P_i works on the bottom level leaf node i
  - every 2 processes cooperate on a Level 2 node
  - every 4 processes cooperate on a Level 3 node

Level 1: local $F_i = U_i \tilde{F}_i$

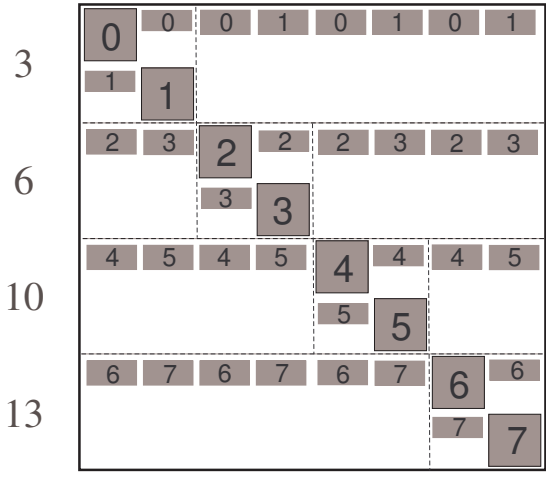


2D-procs mapped to HSS tree nodes

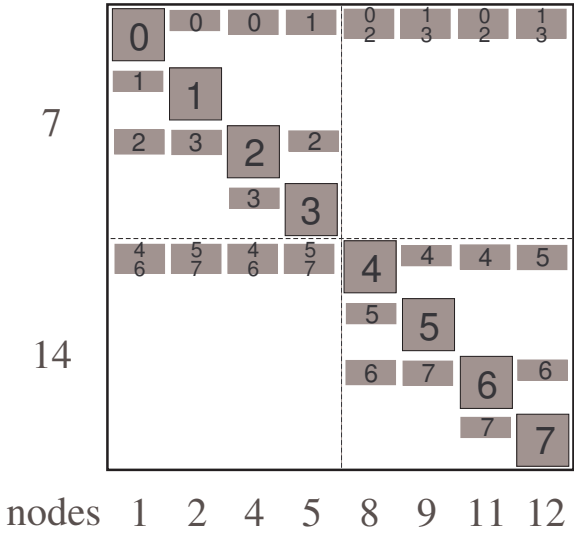


Parallel row compression (cont)

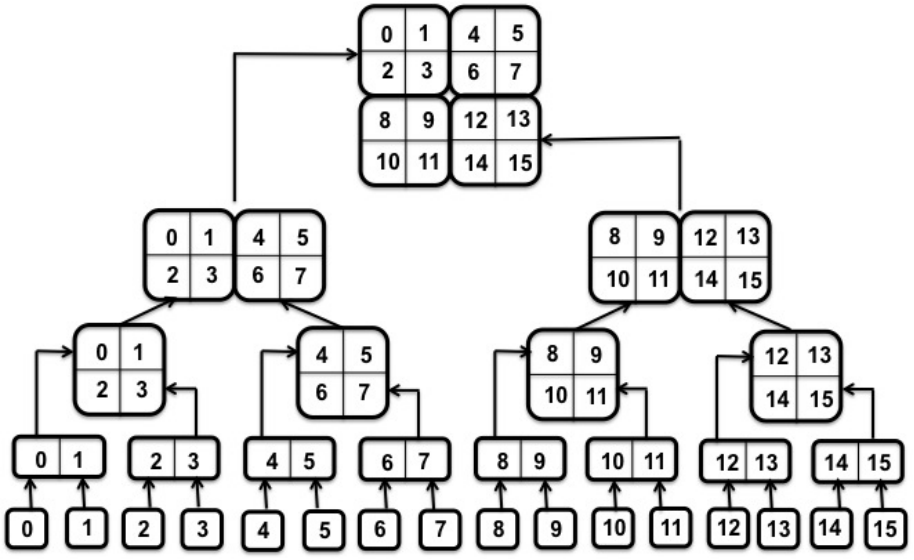
- Level 2 (2-cores/group)



- Level 3 (4-cores/group)



2D-procs mapped to HSS tree nodes



Summary of parallel row compression & complexity

- Each step involves RRQR and redistribution
  - pairwise exchange
- Flop count: $\mathcal{O}(\frac{r N^2}{P})$
- Communication in row compression:

  $\#msg = \mathcal{O}(r \log^2 P)$

  $\#words = \mathcal{O}(r N \log P)$

  (assume no overlap between comm. and comp.)

Arithmetic Intensity: $\mathcal{O}(\frac{N}{P \log P})$

(c.f. ScaLAPACK dense LU: $\mathcal{O}(\frac{N}{\sqrt{P}})$)

Parallel test

- Cray XE6 (hopper at NERSC)
- Example: Helmholtz equation with PML boundary

$$\left(-\Delta - \frac{\omega^2}{v(x)^2}\right) u(x,\omega) = s(x,\omega), \tag{1}$$

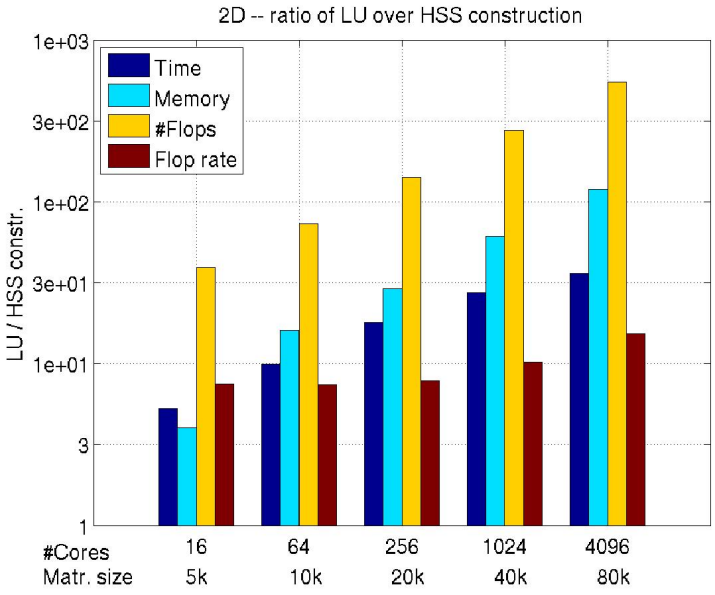
Δ: Laplacian
ω: angular frequency
$v(x)$: seismic velocity field
$u(x,\omega)$: time-harmonic wavefield solution

- FD discretized linear system:
  - Complex, pattern-symmetric, non-Hermitian,
  - Indefinite, ill-conditioned

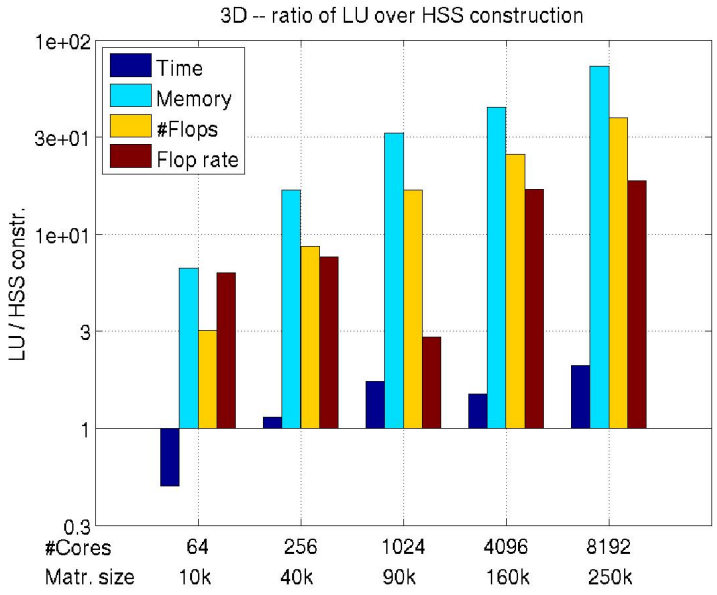
Parallel HSS performance

- HSS constrcution on the last Schur complement corresp. to the top separator.

Performance ratio of LU over HSS:



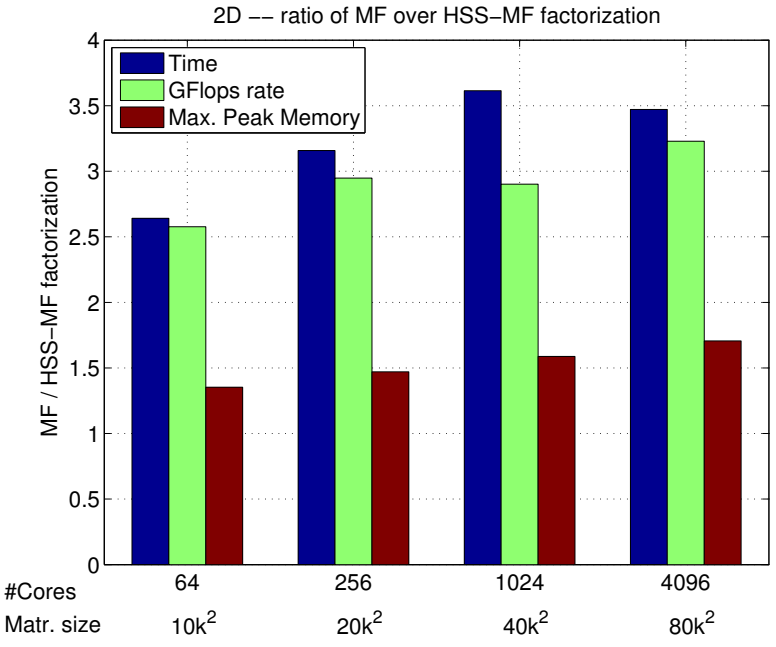
(a) 2D, max_rank=7



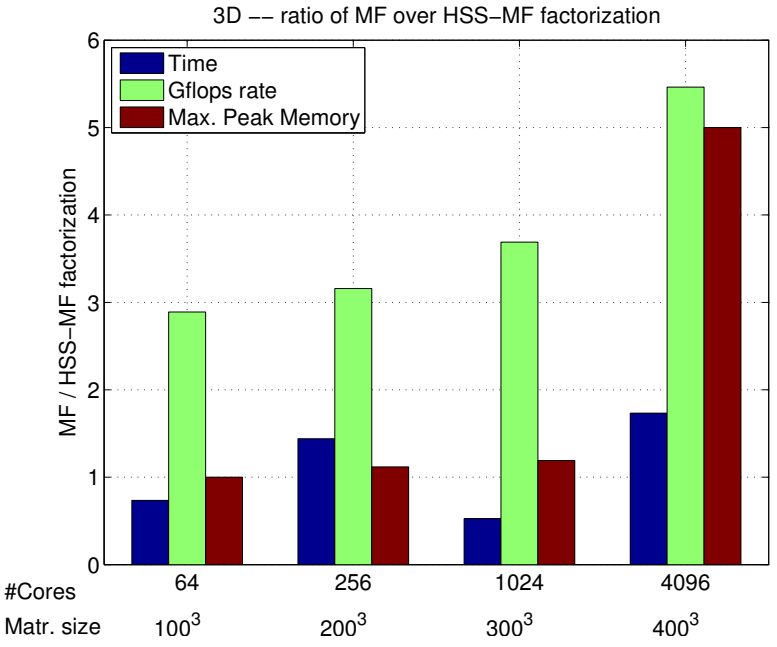
(b) 3D, max_rank=848

S. Wang, X.S. Li, J. Xia, and M.V. de Hoop, "Efficient scalable algorithms for solving linear systems with hierarchically semiseparable matrices", SISC, Nov. 2012. (revised)

Parallel HSS-MF performance



(c) 2D Helmholtz, 10Hz



(d) 3D Helmholtz, 5Hz

HSS-MF succeeded with 600^3 on 16,384 cores, while MF failed.

S. Wang, X.S. Li, F.-H. Rouet, J. Xia, and M. de Hoop, "A Parallel Geometric Multifrontal Solver Using Hierarchically Semiseparable Structure", ACM TOMS, June 2013. (in submission)

Sparse results - 2D problems

2D Helmholtz problems on square grids (mesh size k, $N = k^2$), 10 Hz.

k		10,000	20,000	40,000	80,000
P		64	256	1,024	4,096
MF	Factorization (s)	258.6	544.8	1175.8	2288.5
	Gflops/s	507.3	2109.3	8185.6	31706.9
	Solution+refinement (s)	10.4	10.8	11.5	11.6
	Factors size (GB)	120.1	526.7	2291.2	9903.7
	Max. peak (GB)	2.3	2.5	2.7	2.9
	Communication volume (GB)	136.2	1202.5	9908.1	79648.4
HSS	HSS+ULV (s)	97.9	172.5	325.3	659.3
	Gflops/s	196.9	715.6	2820.7	9820.6
	Solution+refinement (s)	20.2	55.4	61.4	115.8
	Steps	3	3	9	9
	Factors size (GB)	66.2	267.7	1333.2	4572.3
	Max. peak (GB)	1.7	1.7	1.7	1.7
	Communication volume (GB)	74.2	573.8	4393.4	41955.8
	HSS rank	258	503	1013	2015
	$\|\|x - x_{\text{MF}}\|\|/\|\|x_{\text{MF}}\|\|$	1.5×10^{-5}	2.2×10^{-5}	3.1×10^{-5}	3.5×10^{-6}
	$\max_i \frac{\lvert Ax-b\rvert_i}{(\lvert A\rvert\lvert x\rvert+\lvert b\rvert)_i}$	7.1×10^{-6}	1.0×10^{-5}	2.0×10^{-6}	3.5×10^{-6}

Results - 3D problems

3D Helmholtz problems on cubic grids (mesh size k, $N = k^3$), 5 Hz.

k		100	200	300	400
P		64	256	1,024	4,096
MF	Factorization (s)	88.4	1528.0	1175.8	6371.6
	Gflops/s	600.6	2275.7	9505.6	35477.3
	Solution+refinement (s)	0.6	2.2	3.5	4.8
	Factors size (GB)	16.6	280.0	1450.1	4636.1
	Max. peak (GB)	0.5	1.9	2.5	2.0
	Communication volume (GB)	83.1	2724.7	26867.8	165299.3
HSS	HSS+ULV (s)	120.4	1061.3	2233.8	3676.5
	Gflops/s	207.8	720.4	2576.6	6494.8
	Solution+refinement (s)	2.3	8.2	31.5	182.8
	Steps	4	5	10	6
	Factors size (GB)	10.7	112.9	434.3	845.3
	Max. peak (GB)	0.5	1.7	2.1	0.4
	Communication volume (GB)	93.6	2241.2	18621.1	143300.0
	HSS rank	481	925	1391	1860
	$\|\|x - x_{\text{MF}}\|\|/\|\|x_{\text{MF}}\|\|$	6.2×10^{-6}	9.4×10^{-7}	1.1×10^{-6}	1.7×10^{-6}
	$\max_i \frac{\lvert Ax-b\rvert_i}{(\lvert A\rvert\lvert x\rvert+\lvert b\rvert)_i}$	1.5×10^{-7}	5.7×10^{-7}	9.7×10^{-7}	3.7×10^{-6}

Preconditioning results

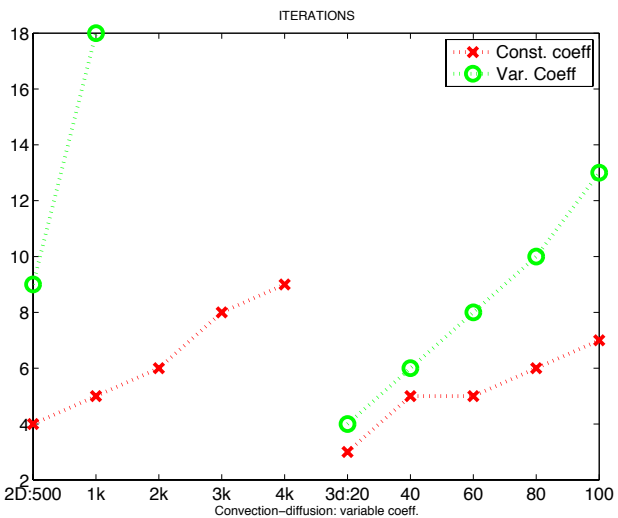
- Test matrices: 2D & 3D
  - model problems
  - convection-diffusion: constant coefficient, variable coefficient
  - curl-curl edge elements (Nedelec elements)
  - Helmholtz
  - general matrices
- RHS $= (1, 1, \ldots)^T$
- GMRES(30)
  - right preconditioner
  - initial $x_0 = (0, 0, \ldots)^T$
  - stopping criterion: $\frac{\|r_k\|_2}{\|b\|_2} \leq 10^{-6}$

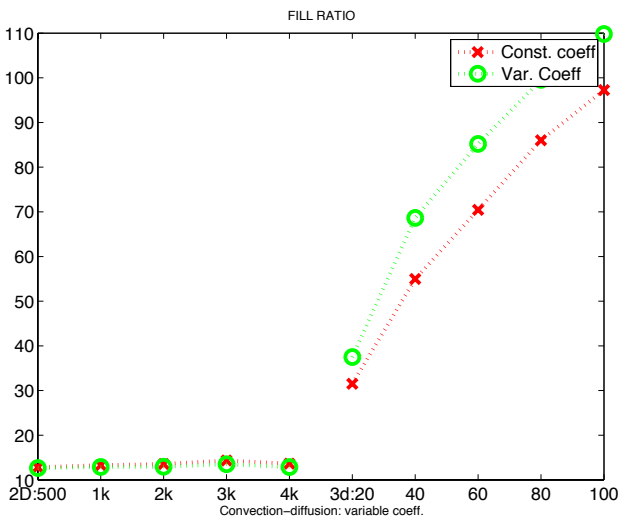
Convection-diffusion

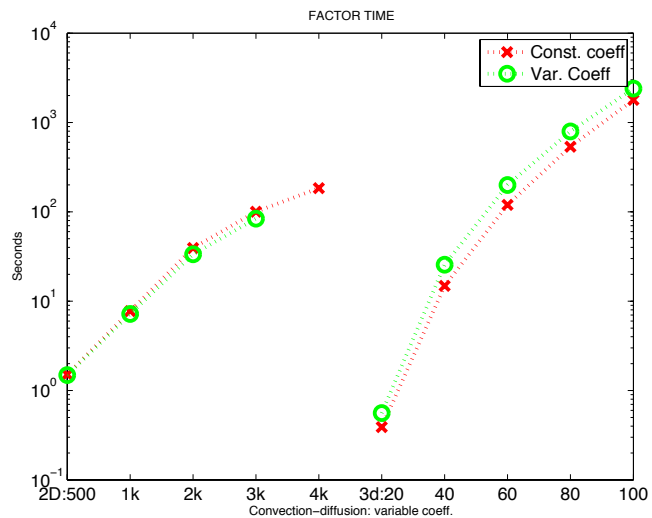
$$-\nu\Delta u + \mathbf{v}\cdot\nabla u = \mathbf{b} \quad \text{on } \Omega\,.$$

$\mathbf{v} = \ldots$

r	constant coeff.	variable coeff.
2D	$(1/\sqrt{2} \quad 1/\sqrt{2})$	$(\ x(1-x)(2y-1) \quad y(1-y)(2x-1)\ )$
3D	$(1/2 \quad 1/2 \quad 1/\sqrt{2})$	$(\ x(1-x)(2y-1)z \quad y(y-1)(2x-1) \quad (2x-1)(2y-1)z(z-1)\ )$

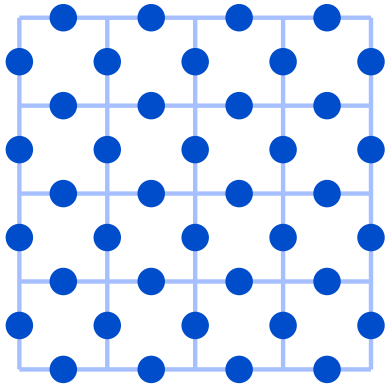






Curl-curl edge element (Nedelec element)

$\nabla \times \nabla \times u + \beta \ = \mathbf{b}$ on Ω
$\tau = 10^{-8}$



mesh size	HSS-rank	fill-ratio	factor (s)	Its	GMRES (s)
500^2	59	14.8e	5.8	2	0.5
1000^2	52	14.9	25.0	3	3.3
2000^2	60	14.9	106.1	3	13.3
3000^2	50	14.4	114.6	5	48.7
20^3	388	78.9	6.7	2	0.1
40^3	824	125.6	261.5	3	1.7
60^3	804	156.1	2055.8	3	7.4

Helmholtz

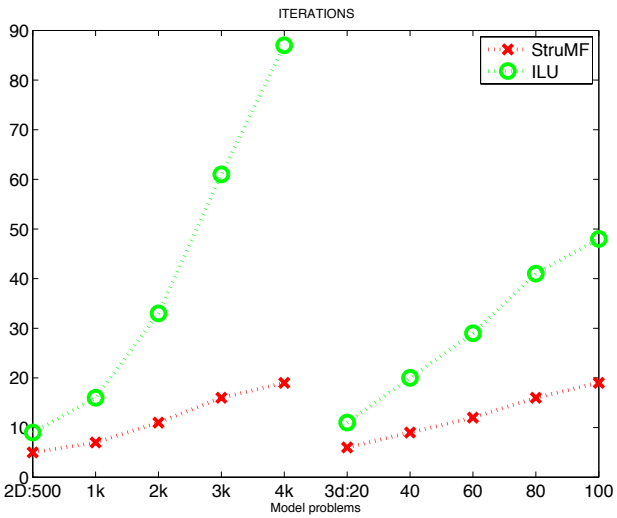
$$\left(-\Delta - \frac{\omega}{v(x)^2}\right) u(x,\omega) = s(x,\omega)$$

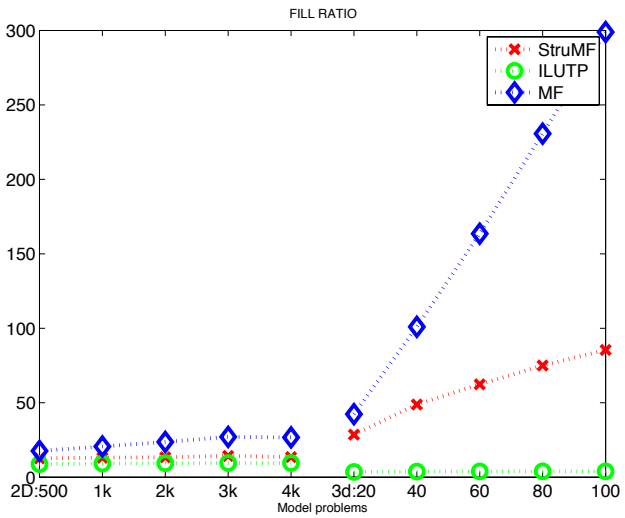
$\tau = 10^{-4}$

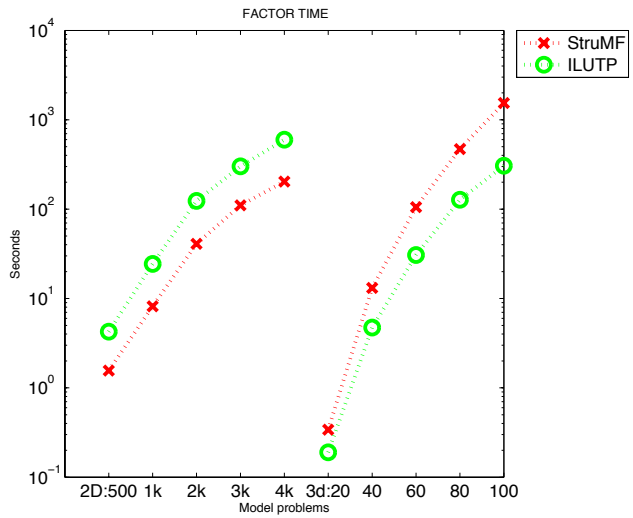
mesh size	HSS-rank	fill-ratio	factor (s)	Its	GMRES (s)
500^2	85	8.8	8.6	4	1.6
1000^2	210	9.5	53.1	4	6.5
2000^2	229	9.7	307.1	71	500.1
3000^2	380	10.0	950.2	139	2464.1
20^3	275	13.3	2.4	3	0.1
40^3	533	27.0	151.9	3	1.1
60^3	1039	38.8	1434.6	3	5.3
80^3	1167	47.3	7708.1	3	16.8

Model problems: $\Delta u = f$

- Compare to ILU in SuperLU (Li/Shao 10)
  - supernode-based ILUTP, threshold, partial pivoting
  - 10^{-4} for HSS trunction, and ILU threshold





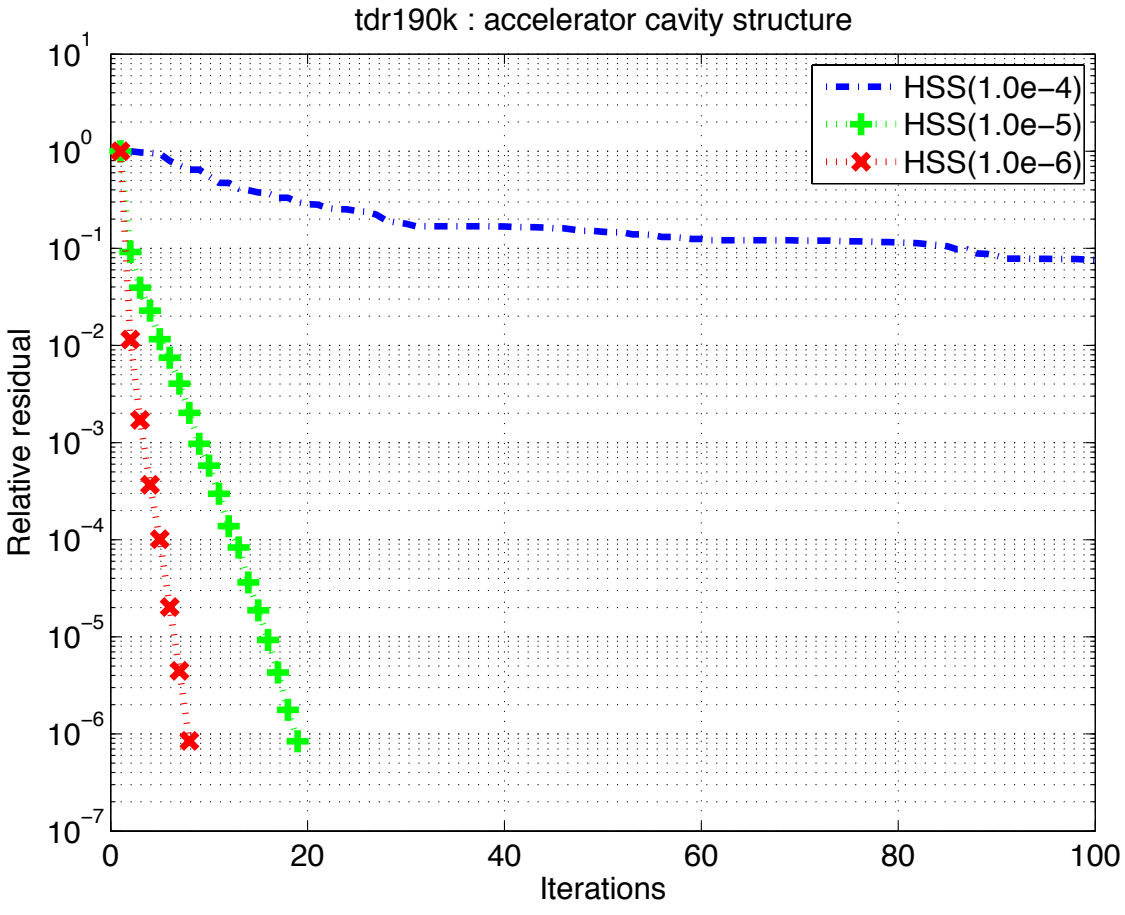


General matrices

Matr.	Descr.	N	HSS-rank	Fill-ratio		Factor (s)		Its	
				HSS	ILU	HSS	ILU	HSS	ILU
add32	circuit	4,690	0	2.1	1.3	0.01	0.01	1	2
mchln85ks17	car tire	84,180	948	13.5	12.3	133.8	216.1	4	39
mhd500	plasma	250,000	100	11.6	15.6	2.5	7.9	2	8
poli_large	economic	15,575	64	4.8	1.6	0.04	0.02	1	2
stomach	bio eng.	213,360	92	12.1	2.9	13.8	18.7	2	2
tdr190k	accelerator	1,100242	596	14.1	–	629.2	–	7	–
torso3	bio eng.	259,156	136	22.6	2.4	86.7	63.7	2	2
utm5940	TOKAMAK	5,940	123	6.7	8.0	0.1	0.16	3	15
wang4	device	26,068	385	45.3	23.1	4.4	6.4	3	4

HSS truncation tolerance

tdr190k – Maxwell equations in frequency domain, eigenvalue problem



Summary

- More theory has been developed
- In practice: very promising for large problems, large machines
  - demonstrated that it is implementable in parallel, with reduced communication
- Compare to ILU preconditioner
  - breakdown free
  - More parallel
  - Dropping operation may be more expensive (row/col vs. entry-wise in ILU)

Future work

- Parallel low-rank factorization using randomized sampling
- Analyze communication lower bound for HSS-structured sparse factorization
  - Classical sparse Cholesky (Gupta et al.'97):
    3D model problem: $\mathcal{O}(\frac{N^{4/3}}{\sqrt{P}})$ COMM-Volume
- Black-box preconditioner?
  - Apply to broader simulation problems: accelerator, fusion, etc.
  - compare to other preconditioners, e.g., ILU, multigrid
- Precondition the Communication-Avoiding Krylov algorithms [with Demmel's group]
- Compare to sparse solvers using $\mathcal{H}$-matrix [Weisbecker et al., Ying et al.]
- Resilience at extreme scale

Rank-revealing via randomized sampling

1. Pick random matrix $\Omega_{n\times(k+p)}$, p small, e.g. 10
2. Sample matrix $S = A\Omega$, with slight oversampling p
3. Compute $Q = \text{ON-basis}(S)$

- **accuracy:** with *probability* $\geq 1 - 6 \cdot p^{-p}$,
  $\|A - QQ^*A\| \leq [1 + 11\sqrt{k+p} \cdot \sqrt{\min\{m,n\}}]\sigma_{k+1}$
- **cost:** $O(kmn)$

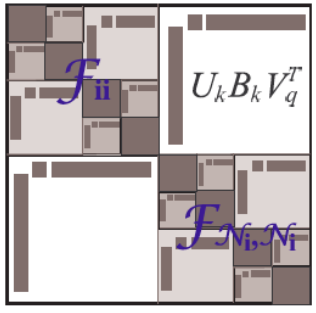
Randomized sampling simplies extend-add

- HSS construction via RS [Martinsson'11]
  - SRRQR repeatedly applied to matrices with **reduced column dimention**

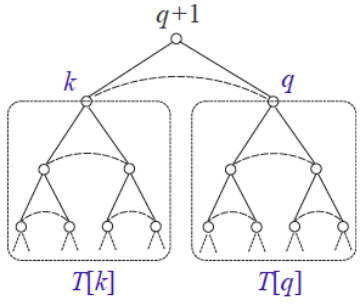
- Multifrontal HSS via RS
  1. Compress frontal $\mathcal{F}_i$:
     Form sample matrix $Y_i = \mathcal{F}_i X_i$, where $X_i = (X_i^{(1)} \quad X_i^{(2)})^T$ random,
     Construct HSS of $\mathcal{F}_i$ with help of Y_i
  2. ULV factorize $\mathcal{F}_i(1,1)$
  3. HSS approximation of $\mathcal{U}_i$
  4. Form sample matrix $Z_i = \mathcal{U}_i X_i^{(2)}$, where $X_i^{(2)}$ is a submatrix of X_i corresponding to $\mathcal{U}_i$
  5. **extend-add of sample matrices to parent:**
     $Y_p \equiv \mathcal{F}_p X_p = (A_p X_p) \oplus Z_i \oplus Z_j$

At Child

- Compression of F_i

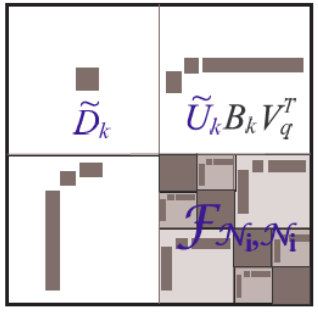


(i) HSS form for $\mathcal{F}_i$

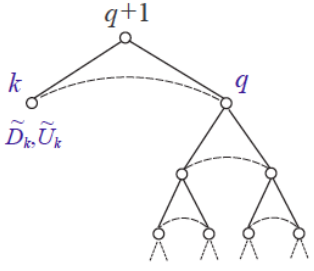


(ii) HSS tree T for $\mathcal{F}_i$ and subtrees $T[k]$ and $T[q]$

- Partial elimination of F_i



(i) After the partial ULV factorization of $\mathcal{F}_{\mathrm{ii}}$

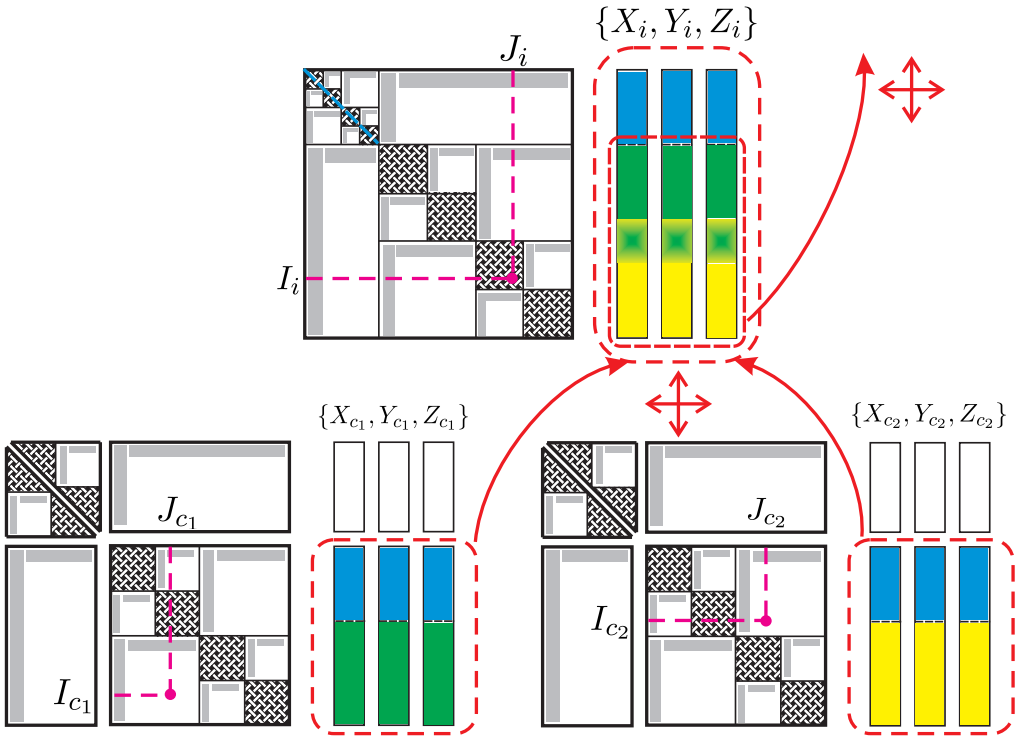


(ii) The resulting tree

- Compute update matrix U_i: $\mathcal{U}_i = \mathcal{F}_i(2,2) - U_q B_k^T (\tilde{U}_k^T \tilde{D}_k^{-1} \tilde{U}_k) B_k U_q^T$
  - fast low-rank update: obtain $\mathcal{U}_i$ generators **directly** from $\mathcal{F}_i(2,2)$ generators

At Parent: extend-add of sample matrices from children

RS simplies extend-add: $Y_p \equiv F_p X_p = (A_p X_p) \oplus Z_i \oplus Z_j$



$$
\begin{aligned}
Z_i &= \mathcal{U}_i X_i^{(2)} = \mathcal{F}_i(2,2) X_i^{(2)} - U_q B_k^T (\tilde{U}_k^T \tilde{D}_k^{-1} \tilde{U}_k) B_k U_q^T X_i^{(2)} \\
&= Y_i^{(2)} - U_q B_k^T U_k^T X_i^{(1)} - U_q B_k^T (\tilde{U}_k^T \tilde{D}_k^{-1} \tilde{U}_k) B_k U_q^T X_i^{(2)} \\
&= Y_i^{(2)} - U_q B_k^T \left[U_k^T X_i^{(1)} + (\tilde{U}_k^T \tilde{D}_k^{-1} \tilde{U}_k) B_k U_q^T X_i^{(2)} \right]
\end{aligned}
$$

References

- M. Bebendorf, “Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems”, Lecture Notes in Computational Science and Engineering, Springer, 2008.
- M. Gu, X.S. Li, P. Vassilevski, “Direction-Preserving and Schur-Monotonic Semiseparable Approximations of Symmetric Positive Definite Matrices”, SIMAX, 31 (5), 2650-2664, 2010.
- J. Xia, M. Gu, “Robust approximate Cholesky factorization of rank-structured symmetric positive definite matrices”, SIMAX, 31 (5), 2899-2920, 2010.
- J. Xia, “Efficient structured multifrontal factorization for general large sparse matrices”, SISC, 35 (2), A832-A860, 2012.
- A. Napov, “Conditioning analysis of incomplete Cholesky factorizations with orthogonal dropping”, to appear in SIMAX.
- Martinsson, “A Fast Randomized Algorithm for Computing A Hierarchically Semiseparable Representation of A Matrix”, SIMAX, Vol.32, No.4, 1251-1274, 2011.
- S. Wang, X.S. Li, J. Xia, and M.V. de Hoop, “Efficient scalable algorithms for solving linear systems with hierarchically semiseparable matrices”, SISC, Nov. 2012. (revised)
- S. Wang, X.S. Li, F.-H. Rouet, J. Xia, and M. de Hoop, “A Parallel Geometric Multifrontal Solver Using Hierarchically Semiseparable Structure”, ACM TOMS, June 2013. (in submission)
- P.R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L’Excellent, C. Weisbecker, “Improving Multifrontal Methods by Means of Block Low-Rank Representations”, SISC, submitted, 2012. Tech report, RT/APO/12/6, ENSEEIHT, Toulouse, France.

4th Gene Golub SIAM Summer School, 7/22 – 8/7, 2013, Shanghai

Lecture 8

Hybrid Solvers based on Domain Decomposition

Xiaoye Sherry Li
Lawrence Berkeley National Laboratory, USA
xsli@lbl.gov

crd-legacy.lbl.gov/~xiaoye/G2S3/

Lecture outline

- Hybrid solver based on Schur complement method
  - Design target: indefinite problems, high degree concurrency
- Combinatorial problems in hybrid solver
  - Multi-constraint graph partitioning
  - Sparse triangular solution with sparse right-hand sides

Schur complement method

- a.k.a iterative substructuring method

  or, non-overlapping domain decomposition

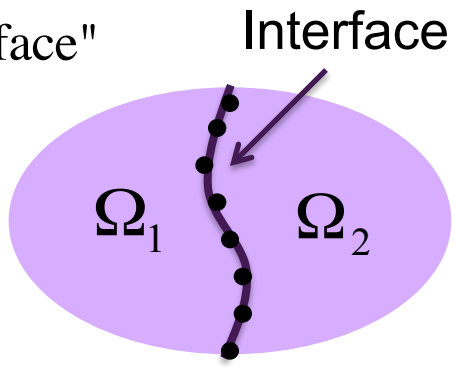
- Divide-and-conquer paradigm . . .
  - Divide entire problem (domain, graph) into subproblems (subdomains, subgraphs)
  - Solve the subproblems
  - Solve the interface problem (Schur complement)

- Variety of ways to solve subdomain problems and Schur complement … lead to a powerful poly-algorithm or hybrid solver framework

Structural analysis view

- Case of two subdomains

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



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,

   Local Schur complements (unassembled): $S^{(k)} = A_{II}^{(k)} - A_{Ii}^{(k)} (A_{ii}^{(k)})^{-1} A_{iI}^{(k)}$

   Global Schur complement (assembled): $S = S^{(1)} + S^{(2)}$

Algebraic view

1. Reorder into 2x2 block system, A_{11} is block diagonal

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

2. Schur complement

$$S = A_{22} - A_{21} A_{11}^{-1} A_{12} = A_{22} - (U_{11}^{-T} A_{21}^{T})^{T} (L_{11}^{-1} A_{12}) = A_{22} - W \cdot G$$

where $A_{11} = L_{11} U_{11}$

S corresponds to interface (separator) variables, no need to form explicitly

3. Compute the solution

(1) $x_2 = S^{-1}(b_2 - A_{21} A_{11}^{-1} b_1)$ ← iterative solver

(2) $x_1 = A_{11}^{-1}(b_1 - A_{12} x_2)$ ← direct solver

Solving the Schur complement system

- SPD, conditioning property [Smith/Bjorstad/Gropp'96]

  For an SPD matrix, condition number of a Schur complement is no larger than that of the original matrix.
  - S is SPD, much reduced in size, better conditioned, but denser, good for preconditioned iterative solver
- Two approaches to preconditioning S
  1. Global approximate S (e.g., PDSLin [Yamazaki/Li.'10], HIPS [Henon/Saad'08])
     - **general algebraic preconditioner, more robust, e.g. ILU(S)**
  2. Local S (e.g. MaPHys [Giraud/Haidary/Pralet'09])
     - **restricted preconditioner, more parallel**
     - **e.g., additive Schwarz preconditioner**

$$S = S^{(1)} \oplus S^{(2)} \oplus S^{(3)} \ldots$$

$$M = S^{(1)^{-1}} \oplus S^{(2)^{-1}} \oplus S^{(3)^{-1}} \ldots$$

Related work

PDSLin (LBNL)	MaPHyS (INRIA/CERFACS)	HIPS (INRIA)
• Multi. procs/subdom, • Smaller Schur compl.	• Multi. procs/subdom, • Smaller Schur compl.	• Multi. subdoms/proc, • Larger Schur compl., • Good load balance
• Threshold “ILU”, • Global approx. Schur • Robust	• Additive Schwartz, • Local Schur • Scalable	• Level-based ILU, • Global approx. Schur • Scalable

- **PDSLin**
  - Uses two levels of parallelization and load-balancing techniques for tackling large-scale systems
  - Provides a robust preconditioner for solving highly-indefinite or ill-conditioned systems
- **Future work:** compare the 3 solvers

Parallelization with serial subdomain

- No. of subdomains increases with increasing core count.
  - ➔ Schur complement size and iteration count increase

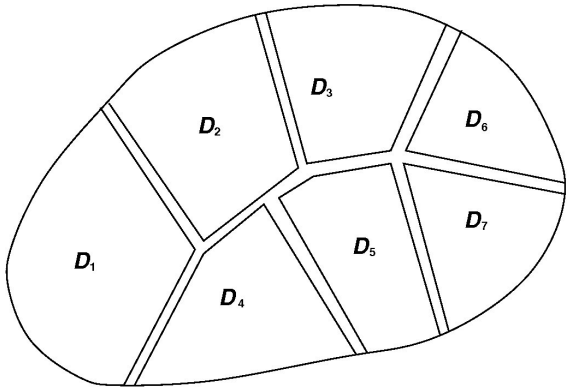
- HIPS (serial subdomain) vs. PDSLin (parallel subdomain)
  - M3D-C[1], Extended MHD to model fusion reactor tokamak, 2D slice of 3D torus
  - Dimension 801k, 70 nonzeros per row, real unsymmetric

P	N_S	HIPS 1.0 sec (iter)	PDSLin sec (iter)
8	13k	284.6 (26)	79.9 (15)
32	29k	55.4 (64)	25.3 (16)
128	62k	--	17.1 (16)
512	124k	--	21.9 (17)

Parallelism – extraction of multiple subdomains

- Partition adjacency graph of $|A|+|A^T|$

  Multiple goals: reduce size of separator, balance size of subdomains
  - Nested Dissection (e.g., PT-Scotch, ParMetis)
  - k-way partition (preferred)



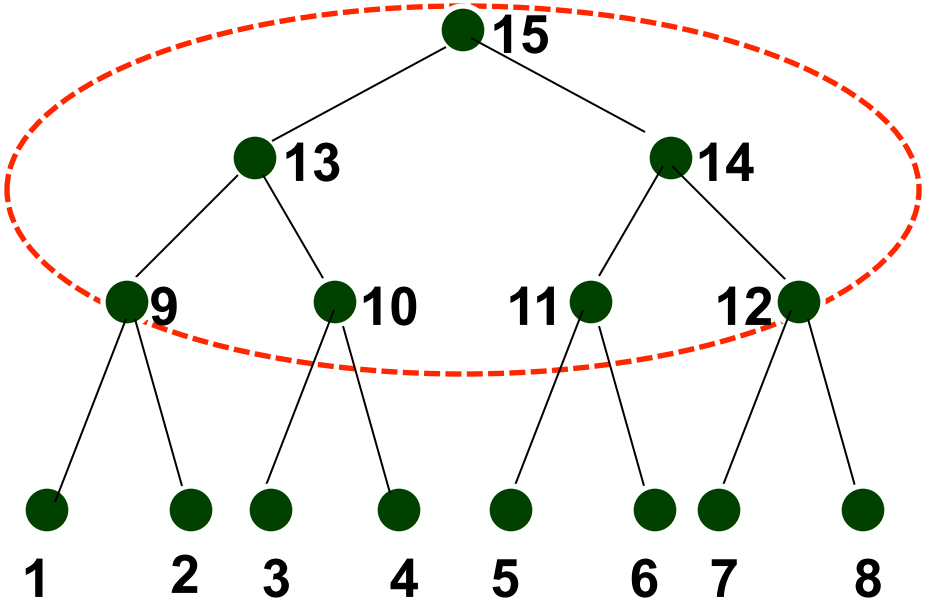
$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{cccc|c} D_1 & & & & E_1 \\ & D_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & D_k & E_k \\ \hline F_1 & F_2 & \cdots & F_k & A_{22} \end{array}\right)$$

- Memory requirement: fill is restricted within
  - "small" diagonal blocks of A_{11}, and
  - ILU(S), maintain sparsity via numerical dropping

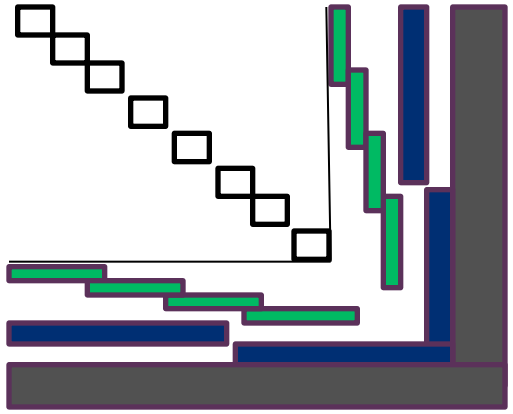
Ordering

- Permute all the separators to the end

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{cccc|c} D_1 & & & & E_1 \\ & D_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & D_k & E_k \\ \hline F_1 & F_2 & \cdots & F_k & A_{22} \end{array}\right)$$

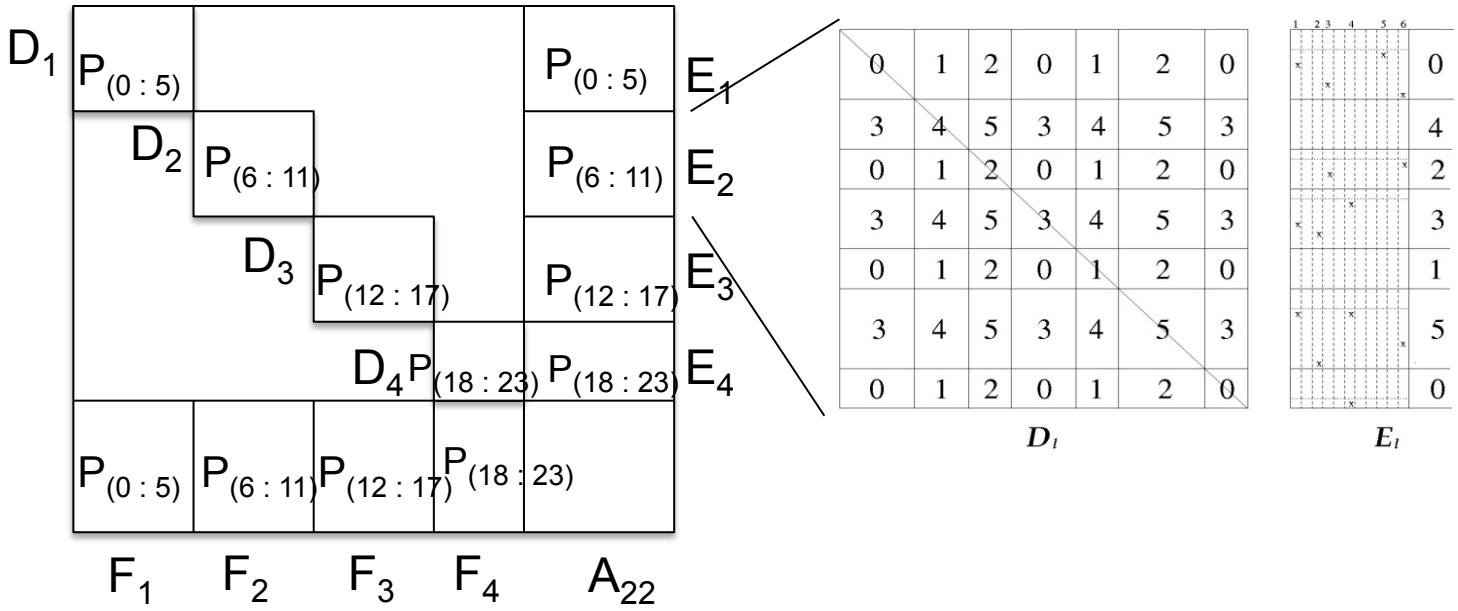


Separator tree



Hierarchical parallelism

- Multiple processors per subdomain
  - one subdomain with 2x3 procs (e.g. SuperLU_DIST, MUMPS)



- Advantages:
  - Constant #subdomains, Schur size, and convergence rate, regardless of core count.
  - Need only modest level of parallelism from direct solver.

Combinatorial problems

- K-way, multi-constraint graph partitioning
  - Small separator
  - Similar subdomains
  - Similar connectivity
- Sparse triangular sol. with many sparse RHS (intra-subdomain)

$$S = A_{22} - \sum_l (U_l^{-T} F_l^T)^T \, (L_l^{-1} E_l) = \sum_l W_l \cdot G_l, \quad \text{where} \quad D_l = L_l U_l$$

- Sparse matrix–matrix multiplication (inter-subdomain)

$$\tilde{W} \leftarrow \text{sparsify}(W, \ \sigma_1); \quad \tilde{G} \leftarrow \text{sparsify}(G, \ \sigma_1)$$

$$T^{(p)} \leftarrow \tilde{W}^{(p)} \cdot \tilde{G}^{(p)}$$

$$\hat{S}^{(p)} \leftarrow A_{22}^{(p)} - \sum_q T^{(q)}(p) \ ; \quad \tilde{S} \leftarrow \text{sparsify}(\hat{S}, \ \sigma_2)$$

I. Yamazali, F.-H. Rouet, X.S. Li, B. Ucar, “On partitioning and reordering problems in a hierarchically parallel hybrid linear solver”, IPDPS / PDSEC Workshop, May 24, 2013.

PDSLin package

http://crd-legacy.lbl.gov/FASTMath-LBNL/Software/

- Parallel Domain decomposition Schur complement based Linear solver
  - C and MPI, with Fortran interface.
  - Unsymmetric / symmetric, real / complex, multiple RHSs.

- Features
  - parallel graph partitioning:
    - **PT-Scotch**
    - **ParMetis**
  - subdomain solver options:
    - **SuperLU, SuperLU_MT, SuperLU_DIST**
    - **MUMPS**
    - **PDSLin**
    - **ILU (inner-outer)**
  - Schur complement solver options:
    - **PETSc**
    - **SuperLU_DIST**

PDSLin encompass Hybrid, Iterative, Direct

Hybrid

Default
Subdomain: LU
Schur: Krylov

Iterative

User Options

(1) num_doms = 0
Schur = A: Krylov

(2) Subdomain: ILU
Schur: Krylov
(FGMRES inner-outer)

Direct

User Options

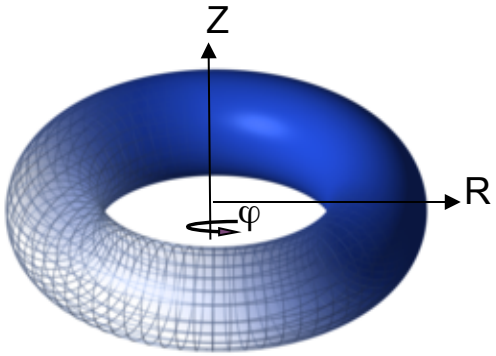
(1) Subdomain: LU
Schur: LU
drop_tol = 0.0

(2) num_doms = 1
Domain: LU

$$\begin{pmatrix} D_1 & & & & E_1 \\ & D_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & D_k & E_k \\ \hline F_1 & F_2 & \dots & F_k & A_{22} \end{pmatrix}$$

Application 1: Burning plasma for fusion energy

- DOE SciDAC project: Center for Extended Magnetohydrodynamic Modeling (CEMM), PI: S. Jardin, PPPL
- Develop simulation codes to predict microscopic MHD instabilities of burning magnetized plasma in a confinement device (e.g., tokamak used in ITER experiments).
  - Efficiency of the fusion configuration increases with the ratio of thermal and magnetic pressures, but the MHD instabilities are more likely with higher ratio.
- Code suite includes M3D-C^1, NIMROD



(S. Jardin)

- At each φ = constant plane, scalar 2D data is represented using 18 degree of freedom quintic triangular finite elements Q_{18}
- Coupling along toroidal direction

2-Fluid 3D MHD Equations

$$\frac{\partial n}{\partial t}+\nabla\bullet(nV)=0 \qquad \text{continuity}$$

$$\frac{\partial B}{\partial t}=-\nabla\times E,\quad \nabla\bullet B=0,\quad \mu_0 J=\partial\times B \qquad \text{Maxwell}$$

$$nM_t\left(\frac{\partial V}{\partial t}+V\bullet\nabla V\right)+\nabla p=J\times B-\nabla\bullet\Pi_{GV}-\nabla\bullet\Pi_{\mu} \qquad \text{Momentum}$$

$$E+V\times B=\eta J+\frac{1}{ne}(J\times B-\nabla p_e-\nabla\bullet\Pi_e) \qquad \text{Ohm's law}$$

$$\frac{3}{2}\frac{\partial p_e}{\partial t}+\nabla\bullet\left(\frac{3}{2}p_eV\right)=-p_e\nabla\bullet\nabla+\eta J^2-\nabla\bullet q_e+Q_{\Delta} \qquad \text{electron energy}$$

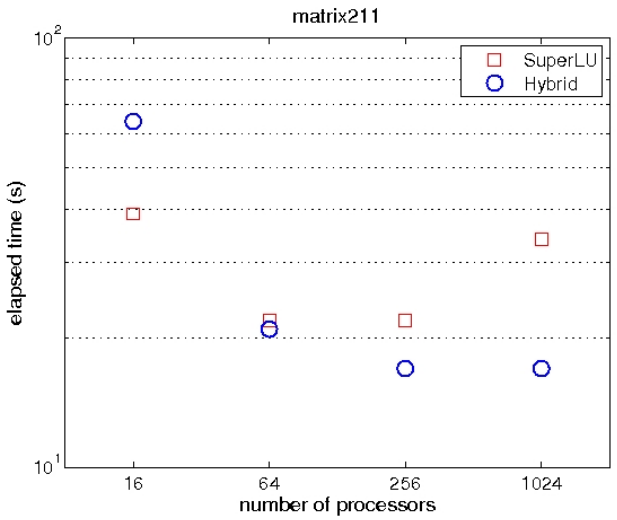
$$\frac{3}{2}\frac{\partial p_i}{\partial t}+\nabla\bullet\left(\frac{3}{2}p_iV\right)=-p_i\nabla\bullet\nabla-\Pi_{\mu}\bullet\nabla V-\nabla\bullet q_i-Q_{\Delta} \qquad \text{ion energy}$$

The objective of the M3D-C^1 project is to solve these equations as accurately as possible in 3D toroidal geometry with realistic B.C. and optimized for a low-β torus with a strong toroidal field.

PDSLin vs. SuperLU_DIST

- Cray XT4 at NERSC
- Matrix211 : extended MHD to model burning plasma
  - dimension = 801K, nonzeros = 56M, real, unsymmetric
  - PT-Scotch extracts 8 subdomains of size ≈ 99K, S of size ≈ 13K
  - SuperLU_DIST to factorize each subdomain, and compute preconditioner LU($\widetilde{S}$)
  - BiCGStab of PETSc to solve Schur system on 64 processors with residual < 10^{-12} , converged in 10 iterations

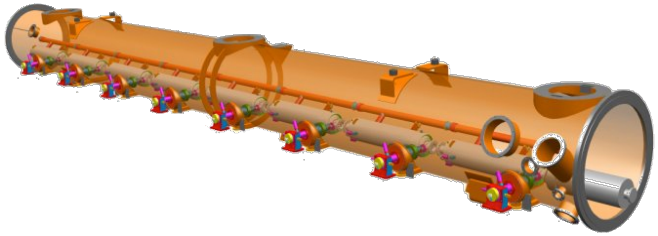
- Needs only 1/3 memory of direct solver



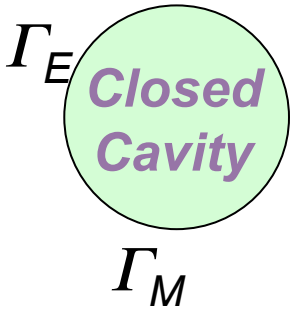
Application 2: Accelerator cavity design

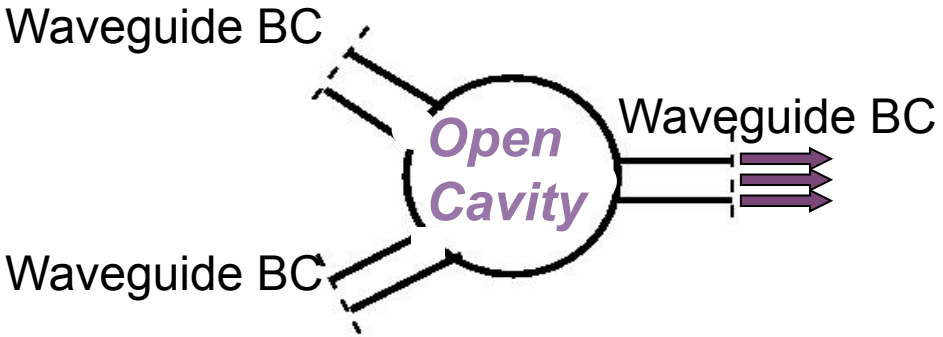
- DOE SciDAC: Community Petascale Project for Accelerator Science and Simulation (ComPASS), PI: P. Spentzouris, Fermilab
- Development of a comprehensive computational infrastructure for accelerator modeling and optimization
- RF cavity: Maxwell equations in electromagnetic field
- FEM in frequency domain leads to large sparse eigenvalue problem; needs to solve shifted linear systems

(L.-Q. Lee)



RF unit in ILC





linear eigenvalue problem

$$(K_0 - \sigma^2 M_0)\, x = M_0\, b$$

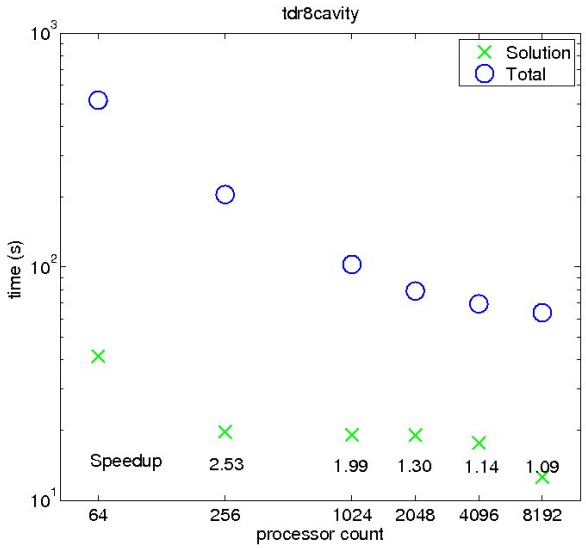
nonlinear complex eigenvalue problem

$$(K_0 + i\,\sigma W - \sigma^2 M_0)\, x = b$$

PDSLin for RF cavity (strong scaling)

- Cray XT4 at NERSC; used 8192 cores
- Tdr8cavity : Maxwell equations to model cavity of International Linear Collider
  - dimension = 17.8M, nonzeros = 727M
  - PT-Scotch extracts 64 subdomains of size ≈ 277K, S of size ≈ 57K
  - BiCGStab of PETSc to solve Schur system on 64 processors with residual < 10^{-12}, converged in 9 ~ 10 iterations

- Direct solver failed !



PDSLin for largest system

- Matrix properties:
  - 3D cavity design in Omega3P, 3rd order basis function for each matrix element
  - dimension = 52.7 M, nonzeros = 4.3 B (~80 nonzeros per row), real, symmetric, highly indefinite
- Experimental setup:
  - 128 subdomains by PT-Scotch (size ~410k)
  - Each subdomain by SuperLU_DIST, preconditioner LU($\tilde{S}$) of size 247k (32 cores)
  - BiCGStab to solve Sy = d by PETSc
- Performance:
  - Fill-ratio (nnz(Precond.)/nnz(A)): ~ 250
  - Using 2048 cores:
    - preconditioner construction: 493.1 sec.
    - solution: 108.1 second (32 iterations)

Combinatorial problems . . .

- K-way graph partitioning with multiple objectives
  - Small separator
  - Similar subdomains
  - Similar connectivity

- Sparse triangular solution with many sparse RHS

$$S = A_{22} - \sum_l (U_l^{-T} F_l^T)^T \, (L_l^{-1} E_l) = A_{22} - \sum_l G \cdot W, \quad \text{where} \quad D_l = L_l U_l$$

- Sparse matrix–matrix multiplication

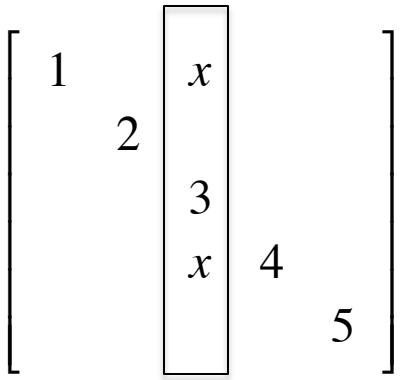
$$\tilde{G} \leftarrow \text{sparsify}(G, \sigma_1); \quad \tilde{W} \leftarrow \text{sparsify}(W, \sigma_1)$$

$$T^{(p)} \leftarrow \tilde{W}^{(p)} \times \tilde{G}^{(p)}$$

$$\hat{S}^{(p)} \leftarrow A_{22}{}^{(p)} - \sum_q T^{(q)}(p); \quad \tilde{S} \leftarrow \text{sparsify}(\hat{S}, \sigma_2)$$

Two graph models

- Standard graph : G=(V, E)
  - GPVS: graph partitioning with vertex separator
  - GPES: graph partitioning with edge separator
- Hypergraph : H = (V, N), net = “edge”, may include more than two vertices
  - Column-net hypergraph: H = (R, C)
    rows = vertices, columns = nets
    n_3 = {1, 3, 4}
  - Row-net hypergraph: H = (C, R)
    columns = vertices, rows = nets



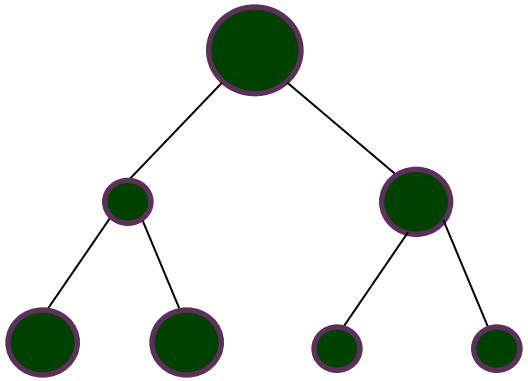
- Partition problem: π(V) = {V_1, V_2, . . . , V_k}, disjoint parts
  - Graph: a cut edge connects two parts
  - Hypergraph: a cut net connects multiple parts

➔ Want to minimize the cutsize in some metric (Objective), and keep equal weights among the parts (Constraints).

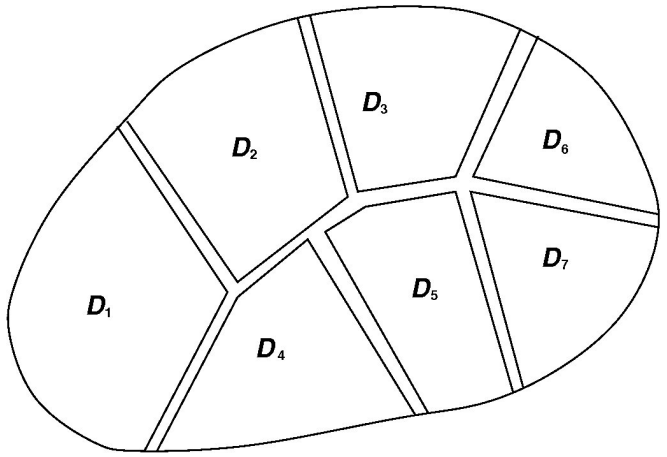
1. K-way subdomain extraction

- **Problem with ND:**

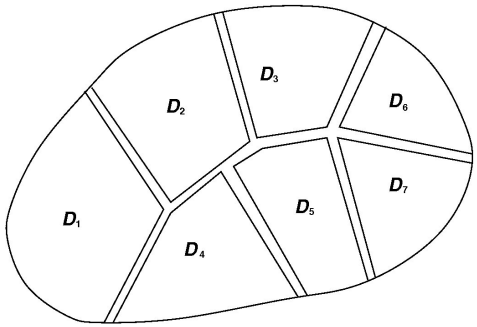
  Imbalance in separator size at different branches → Imbalance in subdomain size



- **Alternative: directly partition into K parts, meet multiple constraints:**
  1. **Compute k-way partitioning → balanced subdomains**
  2. **Extract separator → balanced connectivity**



k-way partition: objectives, constraints



$$\begin{pmatrix} D_1 & & & & E_1 \\ & D_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & D_k & E_k \\ F_1 & F_2 & \dots & F_k & A_{22} \end{pmatrix}$$

- Initial partition to extract vertex-separator impacts **load balance:**

Objectives to minimize:

- number of interface vertices → separator size
- number of interface edges → nonzeros in interface

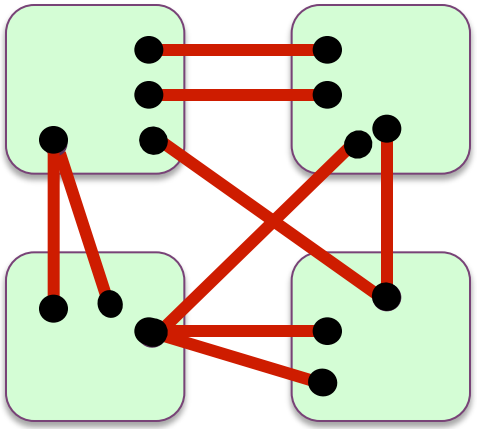
Balance constraints:

- number of interior vertices and edges → LU of D_i
- number of interface vertices and edges → local update matrix

$$(U_l^{-T} F_l^T)^T \; (L_l^{-1} E_l)$$

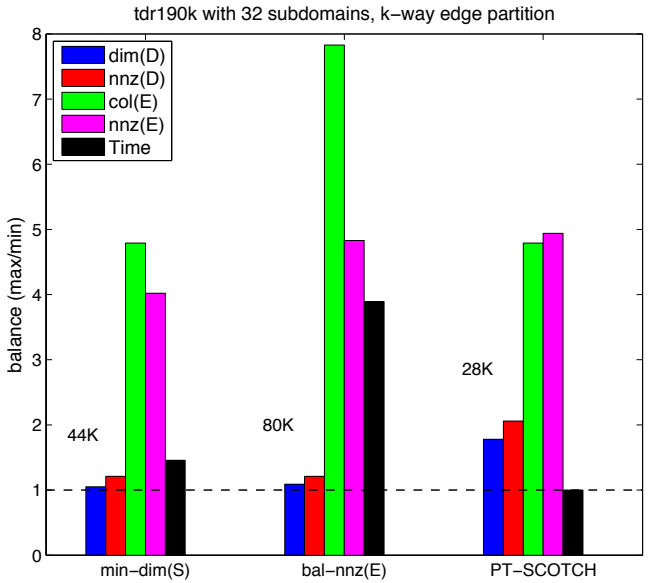
K-way edge partition

- Extract vertex-separator from k-way edge partition
  - Compute k-way edge partition satisfying balanced subdomains (e.g., PT-Scotch, Metis)
  - Extract vertex-separator from edge-separators to minimize and balance interfaces (i.e. minimum vertex cover)
- Heuristics to pick the next vertex: pick a vertex from largest subdomain to maintain balance among subdomains
  - pick the vertex with largest degree to minimize separator size
  - pick the vertex to obtain best balance of interfaces (e.g., nnz).



Balance & Solution time with edge part. + VC

- tdr190k from accelerator cavity design: N = 1.1M, k = 32
- Compared to ND of SCOTCH
- balance = max value$_\ell$ / min value$_\ell$, for ℓ = 1, 2, . . . , k
- Results
  - Improved balance of subdomains, but not of interfaces due to larger separator → total time not improved
  - Post-processing already-computed partition is not effective to balance multiple constraints



Recursive Hypergraph Bisection

- Column-net HG partition to permute an m-by-n matrix M to a singly-bordered form (e.g., Patoh):

$$\mathrm{P}_r M P_c^T = \begin{pmatrix} M_1 & & & C_1 \\ & M_2 & & C_2 \\ & & \ddots & \vdots \\ & & M_k & C_k \end{pmatrix} \rightarrow P_c(M^T M)P_c^T = \begin{pmatrix} M_1^T M_1 & & & M_1^T C_1 \\ & M_2^T M_2 & & M_2^T C_2 \\ & & \ddots & \vdots \\ & & M_k^T M_k & M_k^T C_k \\ C_1^T M_1 & C_2^T M_2 & \cdots & \sum C_l^T C_l \end{pmatrix}$$

- Recursive hypergraph bisection (RHB)
  - Compute structural decomposition str(A) = str($M^T M$)
    - e.g., using edge clique cover of G(A) [Catalyurek '09]
  - Compute 2-way partition of M (with multiple constraints) into a singly-bordered form based on recursive bisection
  - Permute A as:

$$\mathrm{str}(\mathrm{P_c A P_c^T}) = \begin{pmatrix} D_1 & & E_1 \\ & D_2 & E_2 \\ F_1 & F_2 & C \end{pmatrix} = \begin{pmatrix} M_1^T & \\ & M_2^T \\ C & C_2^T \end{pmatrix} \bullet \begin{pmatrix} M_1 & & C_1 \\ & M_2 & C_2 \end{pmatrix}$$

RHB: objectives, constraints

- Objective: minimize cutsize

  Metrics of cutsize:
  - Connectivity – 1, $\sum_{n_j \in N} (\lambda(j) - 1)$ (interface nz columns)
  - Cut-net, $\sum_{n_j \in N, \lambda(j) > 1} 1$ (separator size)
  - Sum-of-external degree (soed), $\sum_{n_j \in N, \lambda(j) > 1} \lambda(j)$ (sum of above)

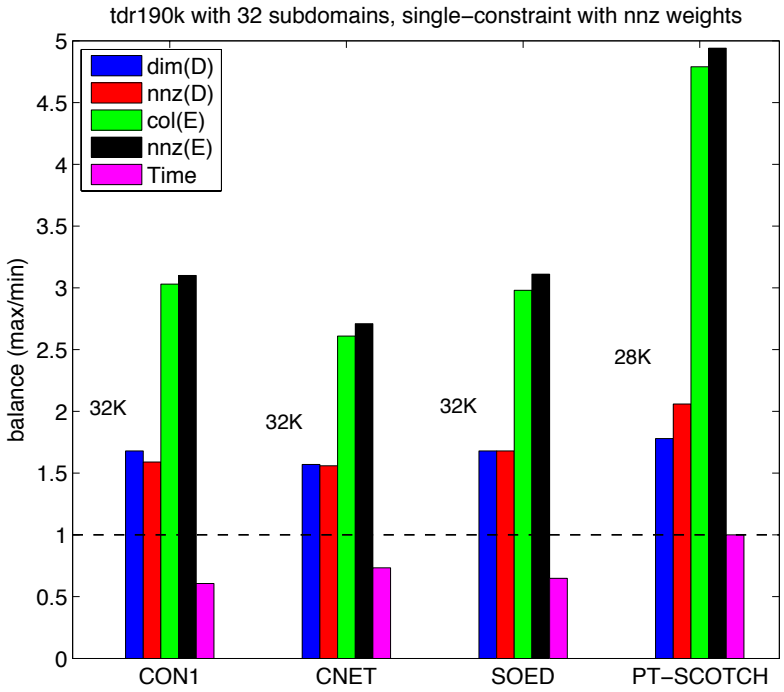
  Where, j-th net n_j is in the cut, and $\boldsymbol{\lambda}_j$ is the number of parts to which n_j is connected

- Constraints: equal weights of different parts

  i-th vertex weights (based on previous partition):
  - unit weight (subdomain dimension)
  - nnz(M_k(i, :)) (subdomain nnz)
  - nnz(M_k(i, :)) + nnz(C_k(i, :)) (interface nnz)

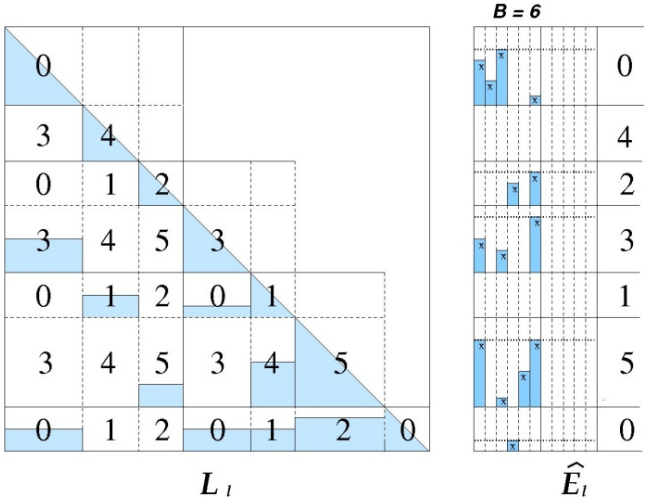
Balance & Time results with RHB

- tdr190k from accelerator cavity design: N = 1.1M, k = 32
- Compared to ND of SCOTCH
- Results
  - Single-constraint improves balance without much increase of separator size → **1.7x faster than ND**
  - Multi-constraints improves balance, but larger separator



2. Sparse triangular solution with sparse RHS (intra-group within a subdomain)

- RHS vectors E_ℓ and F_ℓ are sparse (e.g., about 20 nnz per column); There are many RHS vectors (e.g., $O(10^4)$ columns)

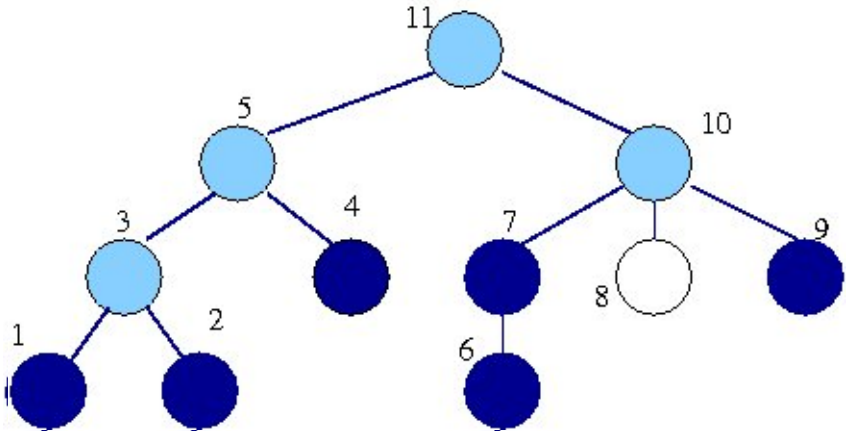


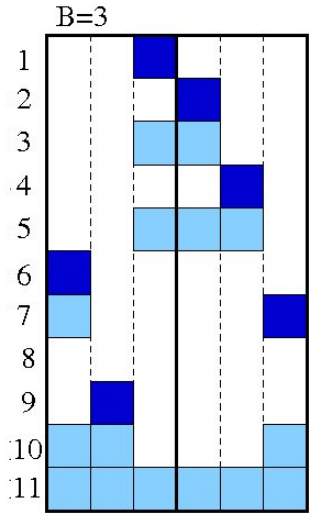
- Blocking RHS vectors
  - Reduce number of calls to the symbolic routine and number of messages, and improve read reuse of the LU factors
  - ➢ Achieved over 5x speedup
  - ✖ zeros must be padded to fill the block → memory cost !

Sparse triangular solution with sparse RHSs

- Objective: Reorder columns of E_ℓ to maximize structural similarity among the adjacent columns.
- Where are the fill-ins?

  Path Theorem [Gilbert'94] Given the elimination tree of D_l, fills generated in G_l at the positions associated with nodes on the path from nodes of the nonzeros in E_l to the root

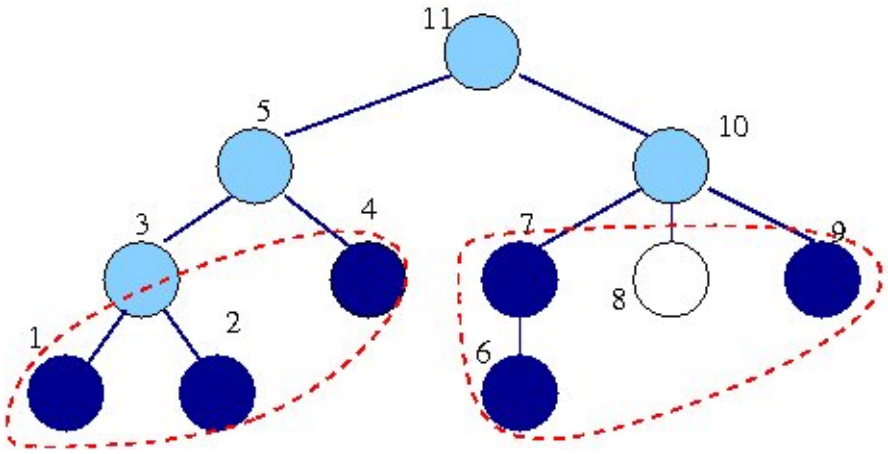


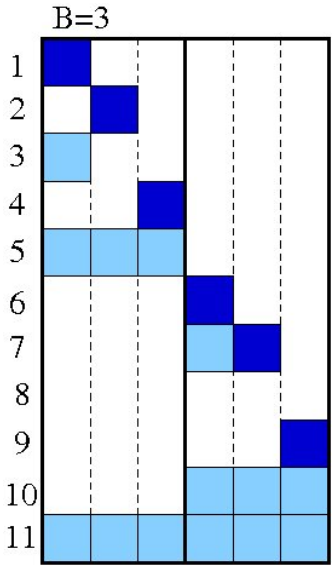


24 padded zeros

Sparse RHS … postordering

- Postorder-conforming ordering of the RHS vectors
  - Postorder elimination tree of D_l
  - Permute columns of E_l s.t. row indices of the first nonzeros are in ascending order
- Increased overlap of the paths to the root, fewer padded zeros
- 20–40% reduction in solution time over ND





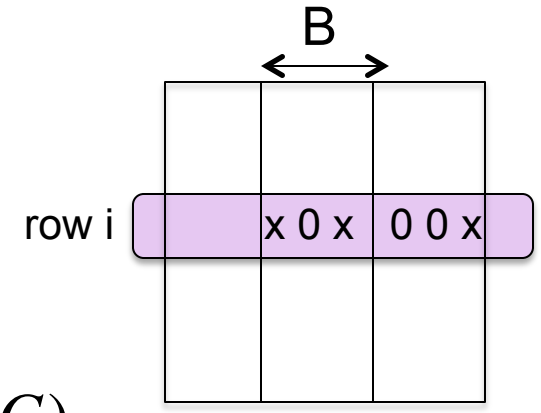
13 padded zeros

Sparse triangular solution … Hypergraph

- Partition/group columns using row-net HG
- Define a cost function ≈ padded zeros

$$cost(\Pi) = \sum_{\text{row } i=1}^{n} \left(\lambda_i B - nnz(G_l(i,:)) \right)$$

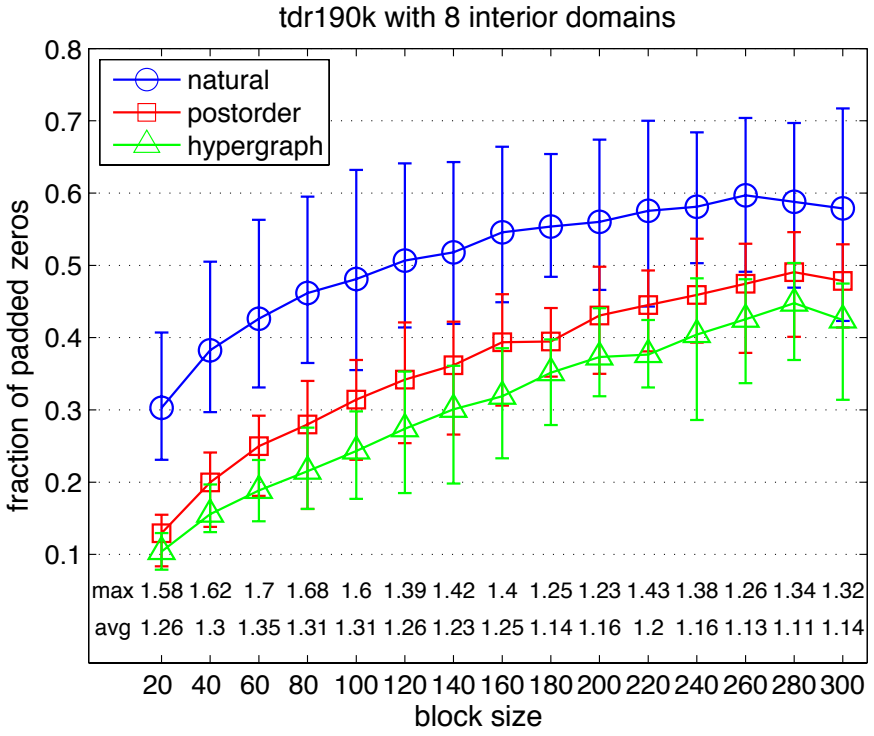
$$= \underbrace{\sum_{i=1}^{n} (\lambda_i - 1) B}_{\text{“connectivity-1” metric}} + \underbrace{\sum_{i=1}^{n} B + nnz(G)}_{\text{constant}}$$

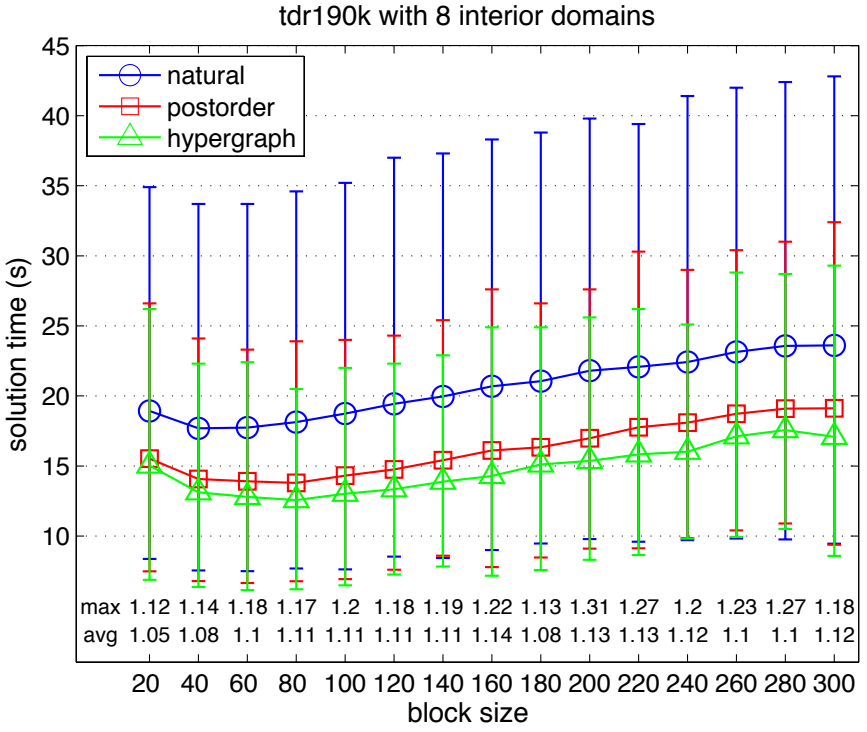


- Minimize cost(π) using Patoh
- Additional 10% reduction in time

Sparse RHS: memory saving

- tdr190k from accelerator cavity design: N = 1.1M, k = 8
- Fraction of padded zeros, with different block size





Summary

- Graph partitioning
  - Direct edge partition + min. vertex cover not effective
  - Recursive Hypergraph Bisection: 1.7x faster
- Reordering sparse RHS in sparse triangular solution
  - postordering: 20-40% faster; hypergraph: additional 10% faster

Remarks

- Direct solvers can scale to 1000s cores
- Domain-decomposition type of hybrid solvers can scale to 10,000s cores
  - Can maintain robustness too
- Beyond 100K cores: Working on AMG combined with low-rank approximate factorization preconditioner

References

- B. Smith, P. Bjorstad, W. Gropp, “Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations”, Cambridge University Press, 1996. (book)
- I. Yamazaki and X.S. Li, “On techniques to improve robustness and scalability of the Schur complement method, VECPAR'10, June 22-25, 2010, Berkeley.
- I. Yamazaki, X.S. Li, F.-H. Rouet, and B. Ucar, “On partitioning and reordering problems in a hierarchically parallel hybrid linear solver'', PDSEC Workshop at IPDPS, 2013, Boston.

Exercises

1. Show that for a symmetric positive definite matrix, the condition number of a Schur complement is no larger than that of the original matrix.

Acknowledgements

Funded through DOE SciDAC projects:

Solvers:

- **TOPS (Towards Optimal Petascale Simulations)**
- **FASTMath (Frameworks, Algorithms, and Scalable Technologies for Mathematics)**

Application partnership:

- **CEMM (Center for Extended MHD Modeling, fusion energy)**
- **ComPASS (Community Petascale Project for Accelerator Science and Simulation)**