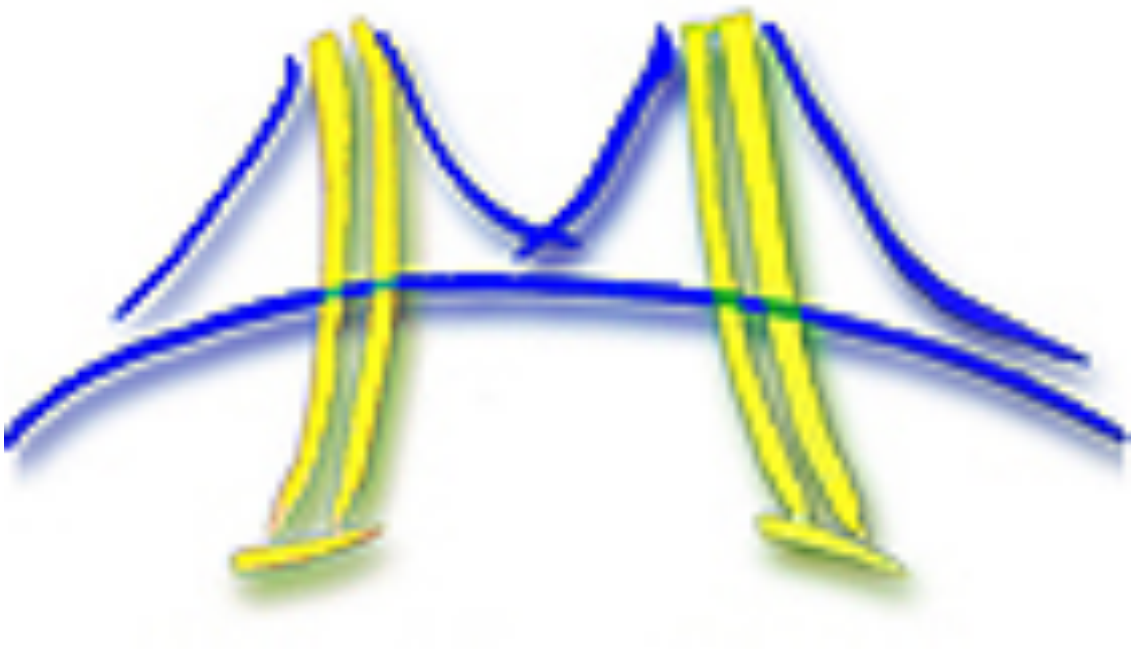
Recent Progress in Autotuning at Berkeley

Jim Demmel
UC Berkeley

bebop.cs.berkeley.edu
parlab.eecs.berkeley.edu

Outline

- Introduction to ParLab
- Summary of autotuning activities
  - Also Shoaib Kamil's talk on Thursday @ 10:15
- Dense Linear Algebra
- Sparse Linear Algebra (summary)
- Communication collectives
- Responses to questions from the organizers

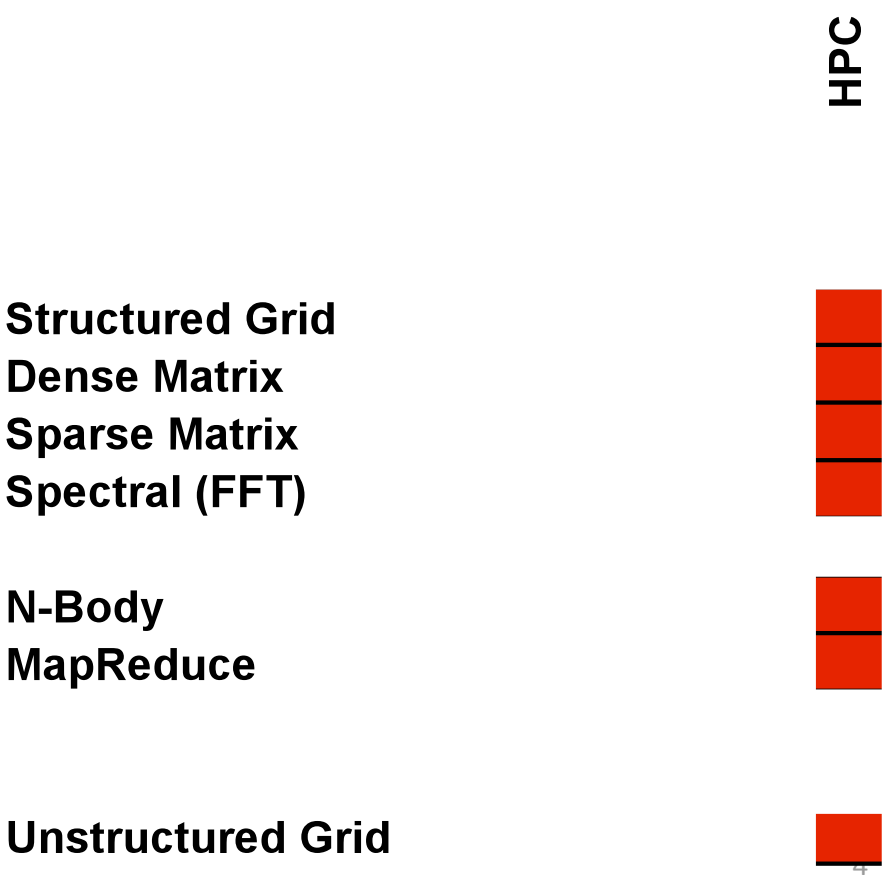


Overview of the Parallel Laboratory

Krste Asanovic, Ras Bodik, Jim Demmel,
Tom Keaveny, Kurt Keutzer, John Kubiatowicz, Edward Lee,
Nelson Morgan, George Necula, Dave Patterson,
Koushik Sen, John Wawrzynek, David Wessel, Kathy Yelick

parlab.eecs.berkeley.edu

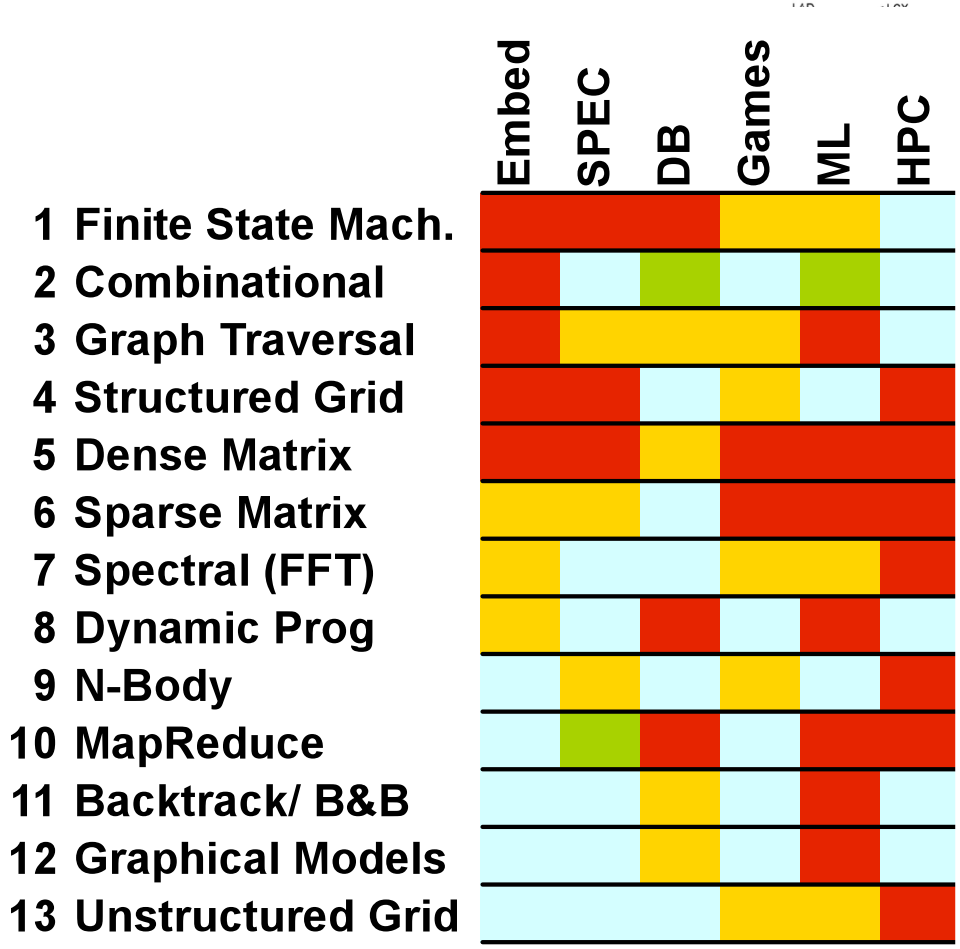
“Motif" Popularity



“Motif" Popularity

(Red Hot → Blue Cool)

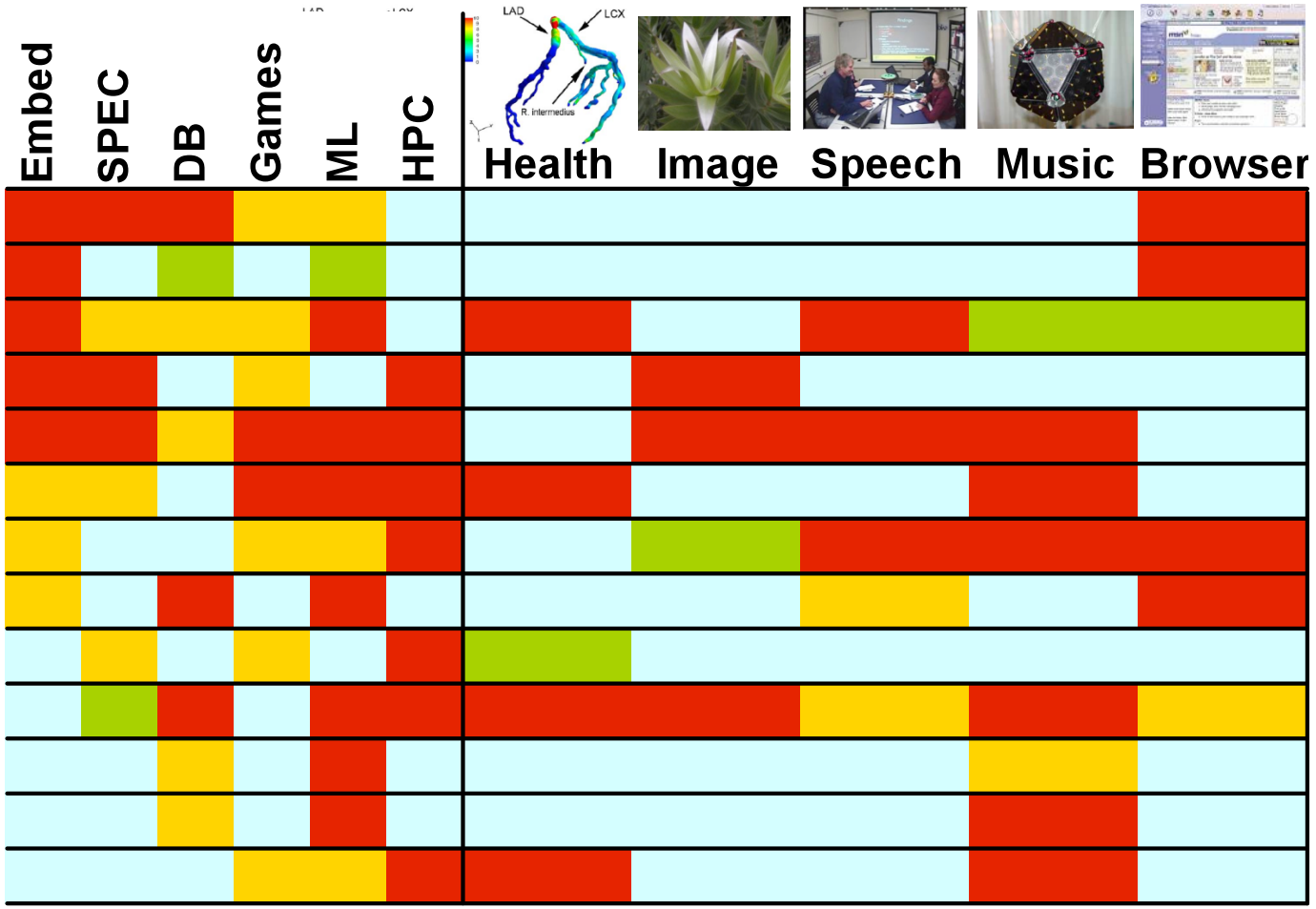
- How do compelling apps relate to 13 motifs?



“Motif" Popularity

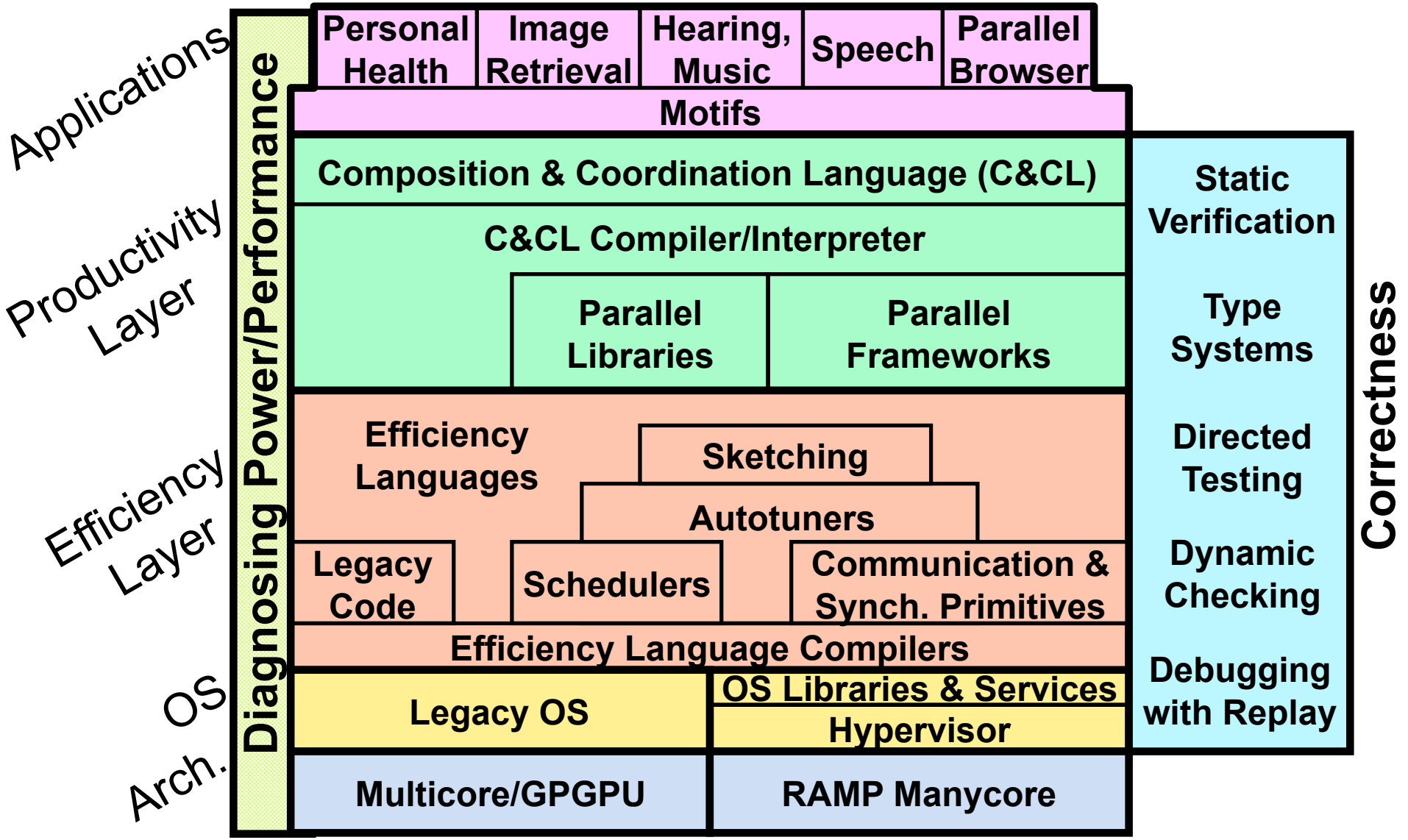
(Red Hot → Blue Cool)

- How do compelling apps relate to 13 motifs?



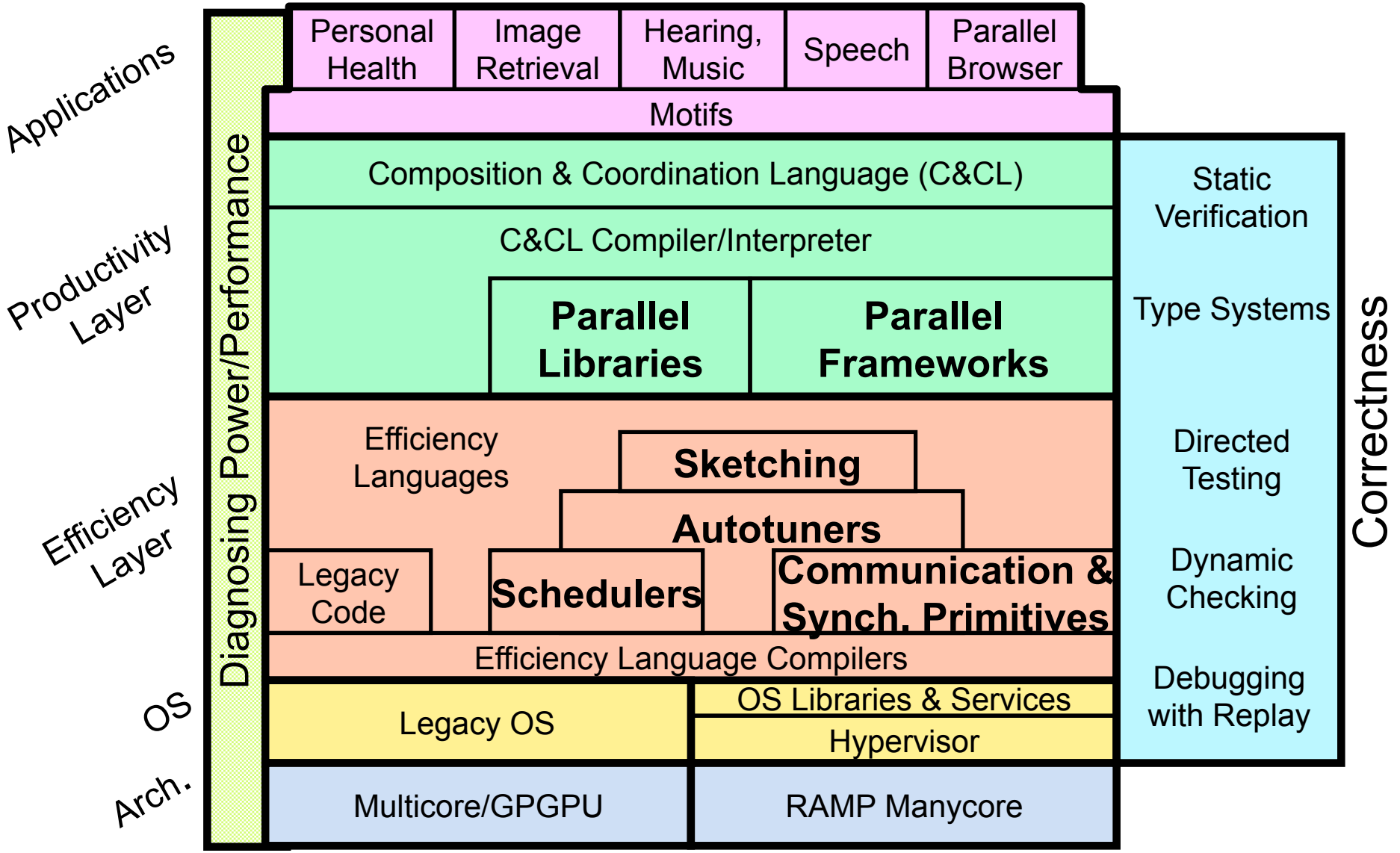
Par Lab Research Overview

Easy to write correct programs that run efficiently on manycore



Par Lab Research Overview

Easy to write correct programs that run efficiently on manycore



Summary of Autotuning (1/2)

- Dense linear algebra
  - New algorithms that attain lower bounds on communication (parallel and sequential)
  - New algorithms for GPUs
- Sparse linear algebra (summary)
  - New algorithms that attain lower bounds on comm.
- Collective communications
  - Choosing right tree for reductions/broadcasts/etc

Summary of Autotuning (2/2)

- *To be presented by Shoaib Kamil on Thursday*
- Recent work autotuning three parallel kernels
  - Sparse Matrix Vector Multiply (SpMV)
  - Lattice Boltzmann MHD
  - Stencils (Heat Equation)
- Lessons learned/commonalities between the autotuners
- Towards a framework for building autotuners
  - What is the role of the compiler?

Motivation

- Running time of an algorithm is sum of 3 terms:
  - # flops * time_per_flop
  - # words moved / bandwidth
  - # messages * latency

  } communication
- Exponentially growing gaps between
  - Time_per_flop << 1/Network BW << Network Latency
    - **Improving 59%/year vs 26%/year vs 15%/year**
  - Time_per_flop << 1/Memory BW << Memory Latency
    - **Improving 59%/year vs 23%/year vs 5.5%/year**

Motivation

- Running time of an algorithm is sum of 3 terms:
  - # flops * time_per_flop
  - # words moved / bandwidth
  - # messages * latency

  } communication
- Exponentially growing gaps between
  - Time_per_flop << 1/Network BW << Network Latency
    - **Improving 59%/year vs 26%/year vs 15%/year**
  - Time_per_flop << 1/Memory BW << Memory Latency
    - **Improving 59%/year vs 23%/year vs 5.5%/year**
- Goal : reorganize/tune motifs to *avoid* communication
  - Not just *hiding* communication (speedup ≤ 2x)
  - Arbitrary speedups possible for linear algebra
  - Success metric – attain lower bounds on communication

Linear Algebra Collaborators (so far)

- UC Berkeley
  - Kathy Yelick, Ming Gu
  - Mark Hoemmen, Marghoob Mohiyuddin, Kaushik Datta, George Petropoulos, Sam Williams, BeBOp group
  - Lenny Oliker, John Shalf
  - ParLab/UPCRC – new parallel computing research center funded by Intel and Microsoft
- CU Denver
  - Julien Langou
- INRIA
  - Laura Grigori, Hua Xiang
- Georgia Tech
  - Rich Vuduc

- Dense work $\subseteq$ ongoing development of Sca/LAPACK
  - Joint with UTK

Communication Lower Bounds for Dense Linear Algebra (1/2)

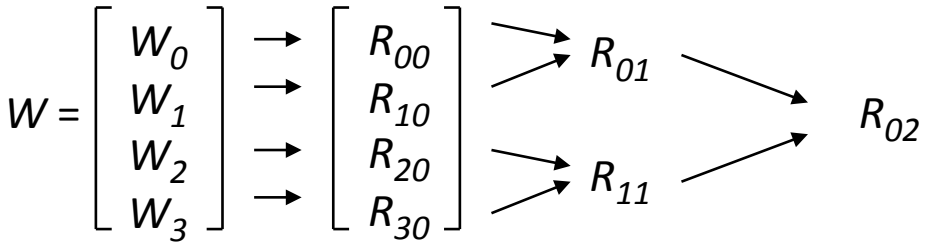
- Hong-Kung (1981), Irony/Tishkin/Toledo (2004)
- Usual matmul using $2n^3$ flops
- Sequential case, with fast memory of size $W < 3n^2$
  - Thm: #words moved = $\Omega(n^3 / W^{1/2})$
- Parallel case, P procs, $O(n^2 / P)$ memory/proc
  - Thm: #words moved = $\Omega(n^2 / P^{1/2})$
- Bandwidth lower bound $\Rightarrow$ latency lower bound
  - #messages $\geq$ # words moved / (fast,local)memory size
  - Sequential: # messages = $\Omega(n^3 / W^{3/2})$
  - Parallel: # messages = $\Omega(P^{1/2})$

Communication Lower Bounds for Dense Linear Algebra (2/2)

- Same lower bounds apply to LU and QR
  - Assumption: $O(n^3)$ algorithms; LU is easy but QR is subtle
- LAPACK and ScaLAPACK do *not* attain these bounds
  - ScaLAPACK attains bandwidth bound
    - But sends $O((mn/P)^{1/2})$ times more messages
  - LAPACK attains neither; $O((m^2/W)^{1/2})$ x more words moved
- But new algorithms do attain them, mod polylog factors
  - Parallel QR: requires new representation of Q, redundant flops
  - Sequential QR: can use new one, or recursive
  - Parallel LU: new pivoting strategy needed, stable in practice
  - Sequential LU: can use new one or recursive, but higher latency

Minimizing Communication in Parallel QR

- QR decomposition of m x n matrix W, m >> n
  - TSQR = "Tall Skinny QR"
  - P processors, block row layout
- Usual Parallel Algorithm
  - Compute Householder vector for each column
  - Number of messages $\propto$ n log P
- Communication Avoiding Algorithm
  - Reduction operation, with QR as operator
  - Number of messages $\propto$ log P - optimal



TSQR in more detail

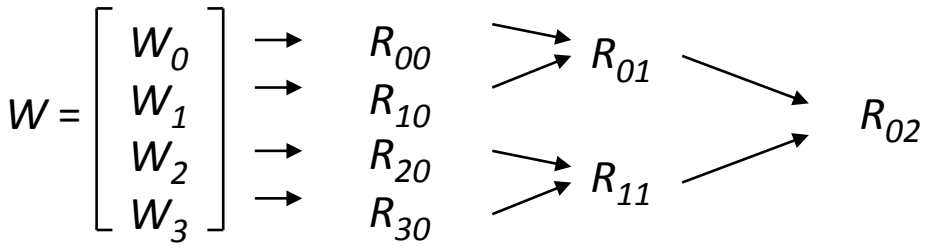
$$W = \begin{pmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{pmatrix} = \begin{pmatrix} Q_{00} & & \\ & Q_{10} & \\ & & Q_{20} \\ & & & Q_{30} \end{pmatrix} \cdot \begin{pmatrix} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{pmatrix}$$

$$\begin{pmatrix} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{pmatrix} = \begin{pmatrix} Q_{01} & \\ & Q_{11} \end{pmatrix} \cdot \begin{pmatrix} R_{01} \\ R_{11} \end{pmatrix} \qquad \begin{pmatrix} R_{01} \\ R_{11} \end{pmatrix} = Q_{02} R_{02}$$

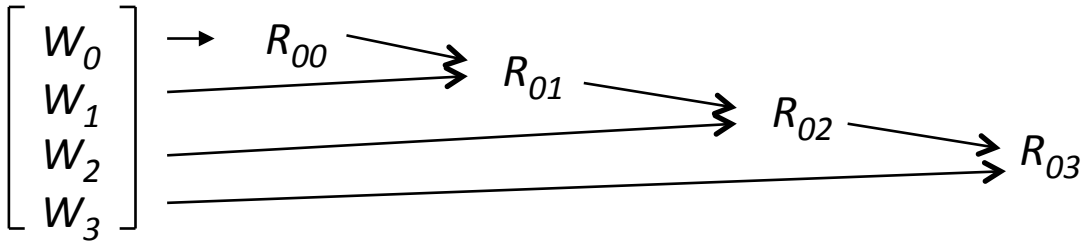
Q is represented implicitly as a product
(tree of factors)

Minimizing Communication in TSQR

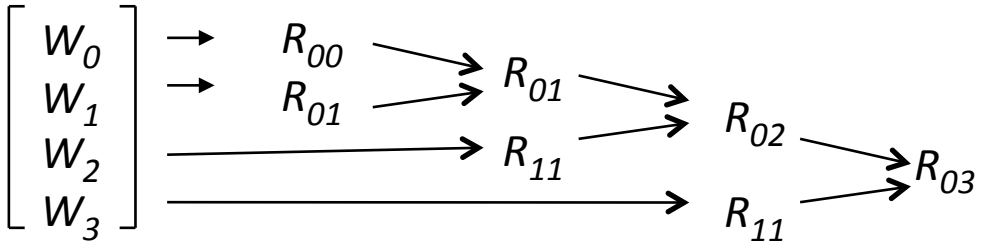
Parallel:



Sequential:



Dual Core:



Multicore / Multisocket / Multirack / Multisite / Out-of-core: ?

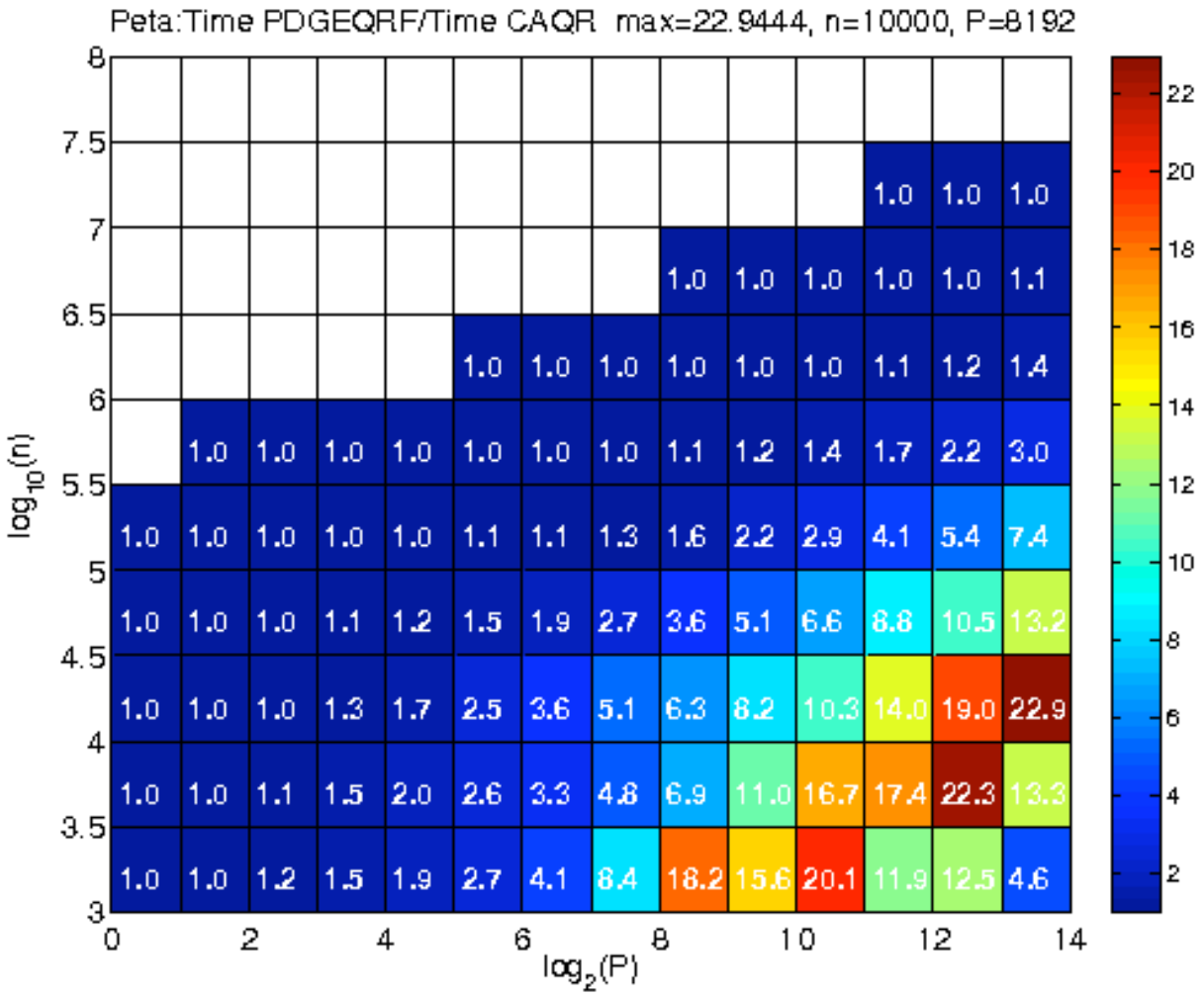
Choose reduction tree dynamically

Performance of TSQR vs Sca/LAPACK

- Parallel
  - Pentium III cluster, Dolphin Interconnect, MPICH
    - Up to **6.7x speedup** (16 procs, 100K x 200)
  - BlueGene/L
    - Up to **4x speedup** (32 procs, 1M x 50)
  - Both use Elmroth-Gustavson locally – enabled by TSQR
- Sequential
  - OOC on PowerPC laptop
    - As little as **2x slowdown vs (predicted) infinite DRAM**
- See UC Berkeley EECS Tech Report 2008-74
  - Being revised to add optimality results!

- General rectangular QR
  - TSQR for panel factorization, then right-looking
  - Implementation under way (Julien Langou)

Modeled Speedups of CAQR vs ScaLAPACK



Petascale
up to 22.9x

IBM Power 5
up to 9.7x

"Grid"
up to 11x

Petascale machine with 8192 procs, each at 500 GFlops/s, a bandwidth of 4 GB/s.

$$\gamma = 2 \cdot 10^{-12}\, s, \alpha = 10^{-5}\, s, \beta = 2 \cdot 10^{-9}\, s / word.$$

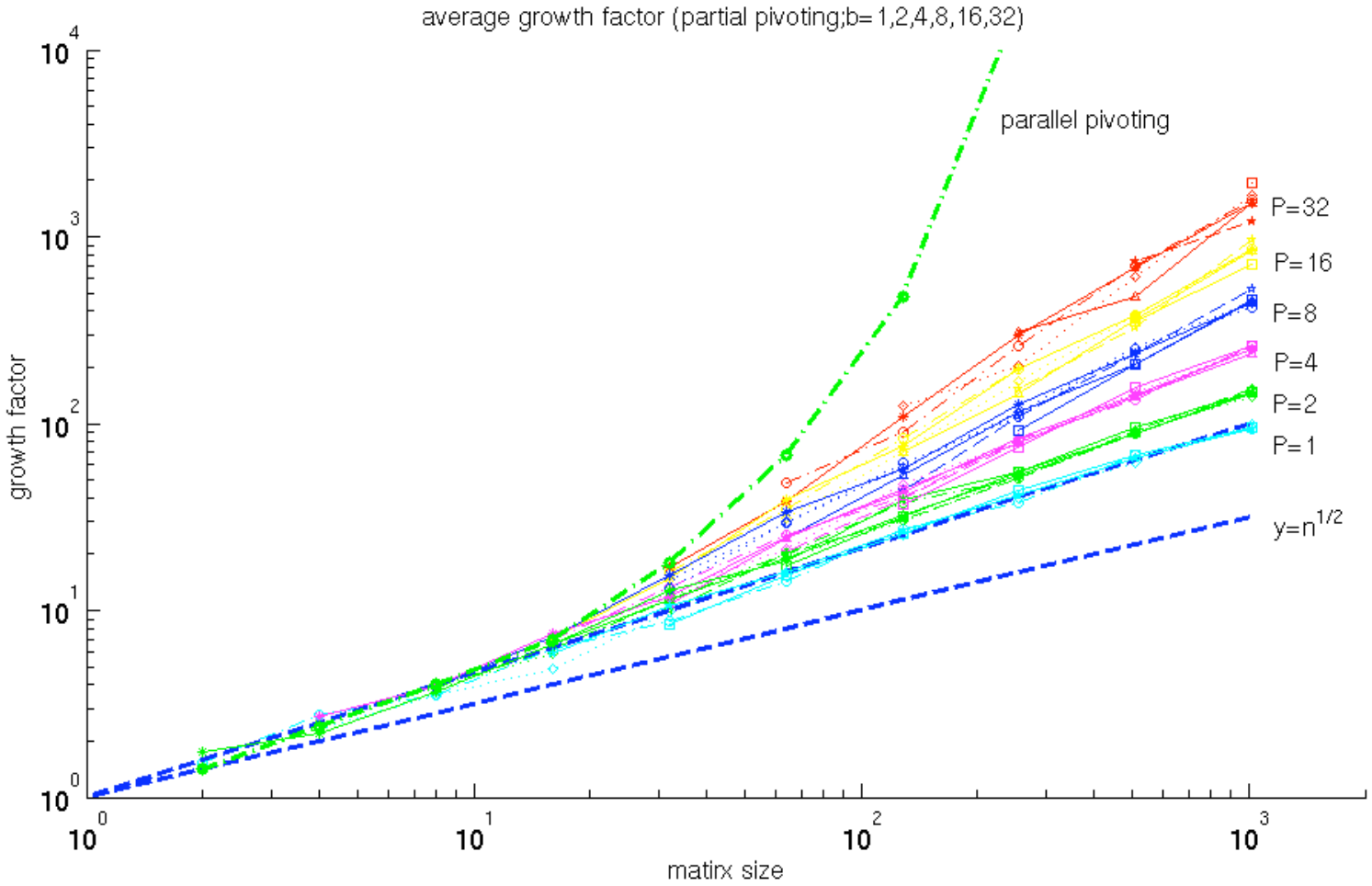
TSLU: LU factorization of a tall skinny matrix

First try the obvious generalization of TSQR:

$$W = \begin{pmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{pmatrix} = \underbrace{\begin{pmatrix} \prod_{00} & & \\ & \prod_{10} & \\ & & \prod_{20} \\ & & & \prod_{30} \end{pmatrix}}_{\Pi_0} \cdot \begin{pmatrix} L_{00} & & \\ & L_{10} & \\ & & L_{20} \\ & & & L_{30} \end{pmatrix} \cdot \begin{pmatrix} U_{00} \\ U_{10} \\ U_{20} \\ U_{30} \end{pmatrix}$$

$$\begin{pmatrix} U_{00} \\ U_{10} \\ U_{20} \\ U_{30} \end{pmatrix} = \underbrace{\begin{pmatrix} \prod_{01} & \\ & \prod_{11} \end{pmatrix}}_{\Pi_1} \cdot \begin{pmatrix} L_{01} & \\ & L_{11} \end{pmatrix} \cdot \begin{pmatrix} U_{01} \\ U_{11} \end{pmatrix} \qquad \begin{pmatrix} U_{01} \\ U_{11} \end{pmatrix} = \underbrace{\prod_{02}}_{\Pi_2} L_{02} U_{02}$$

Growth factor for TSLU based factorization



Unstable for large P and large matrices.

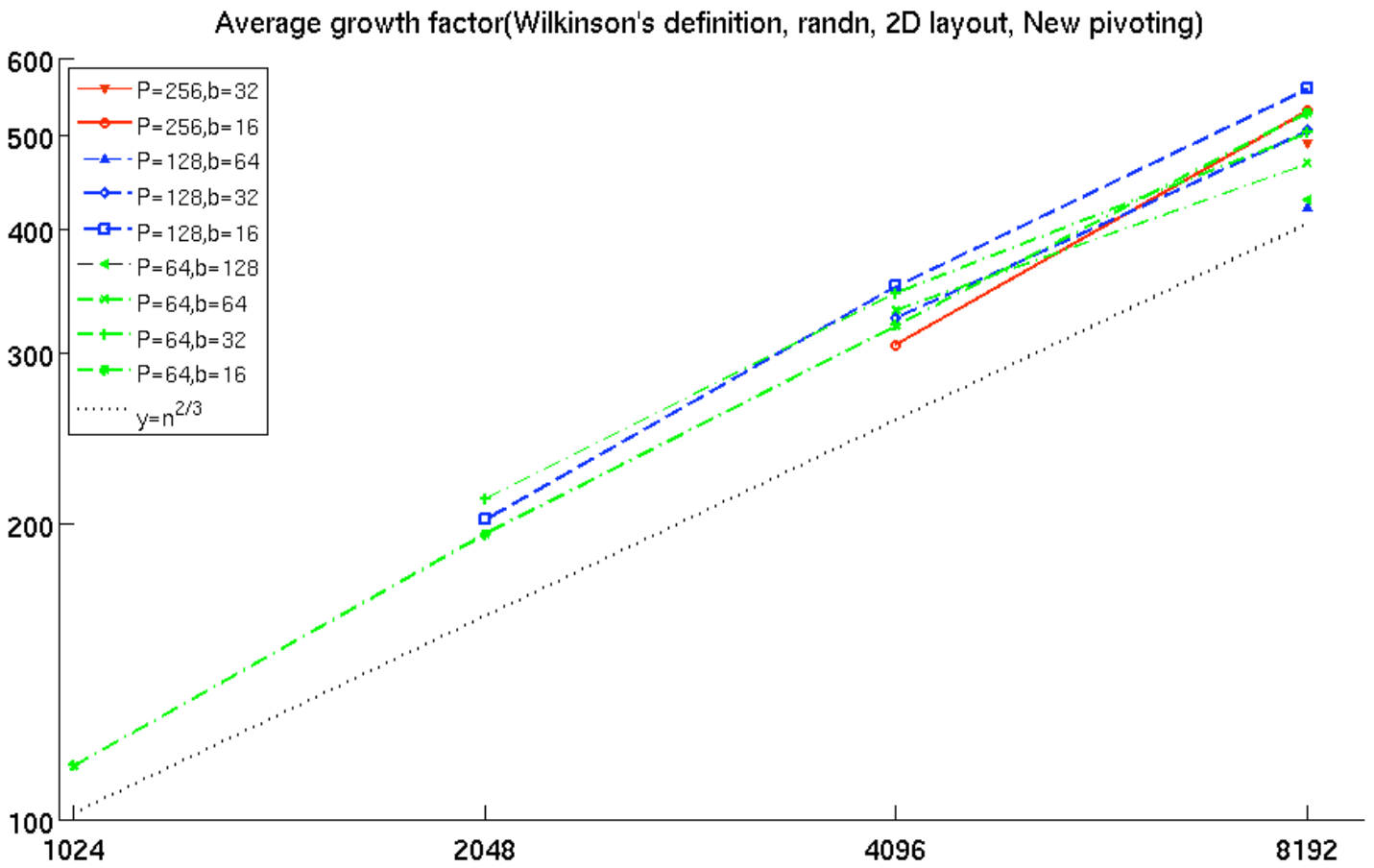
When P = # rows, TSLU is equivalent to parallel pivoting.

Courtesy of H. Xiang

Making TSLU Stable

- At each node in tree, TSLU selects b pivot rows from 2b candidates from its 2 child nodes
- At each node, do LU on 2b *original* rows selected by child nodes, not U factors from child nodes
- When TSLU done, permute b selected rows to top of original matrix, redo b steps of LU without pivoting
- CALU – Communication Avoiding LU for general A
  - Use TSLU for panel factorizations
  - Apply to rest of matrix
  - Cost: redundant panel factorizations
- Benefit:
  - Stable in practice, but *not* same pivot choice as GEPP
  - b times fewer messages overall - faster

Growth factor for better CALU approach



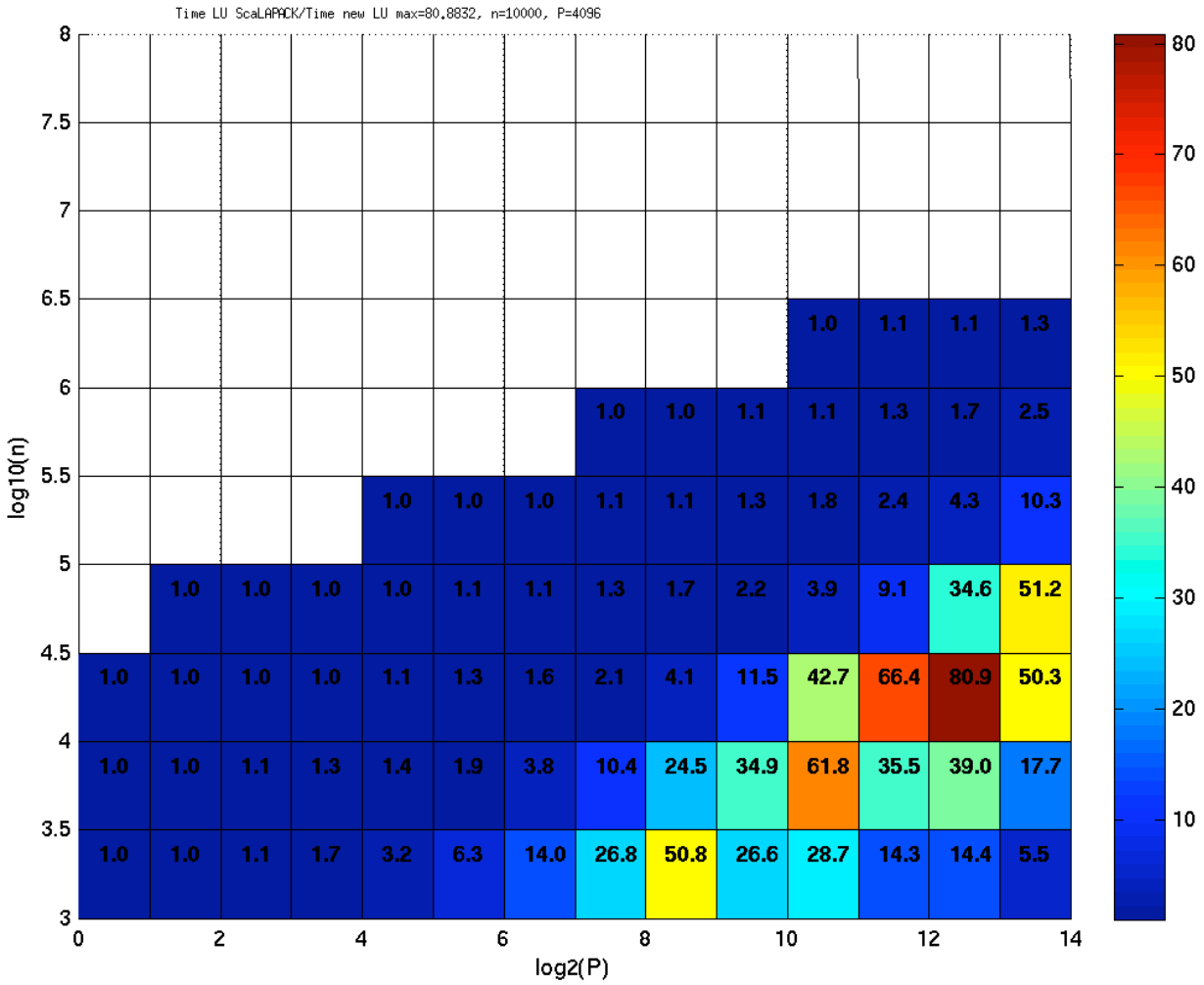
Like threshold pivoting with worst case threshold = .33 , so |L| <= 3

Testing shows about same residual as GEPP

Performance vs ScaLAPACK

- TSLU
  - IBM Power 5
    - Up to **4.37x** faster (16 procs, 1M x 150)
  - Cray XT4
    - Up to **5.52x** faster (8 procs, 1M x 150)
- CALU
  - IBM Power 5
    - Up to **2.29x** faster (64 procs, 1000 x 1000)
  - Cray XT4
    - Up to **1.81x** faster (64 procs, 1000 x 1000)
- Optimality analysis analogous to QR
- See INRIA Tech Report 6523 (2008)

Speedup prediction for a Petascale machine - up to 81x faster

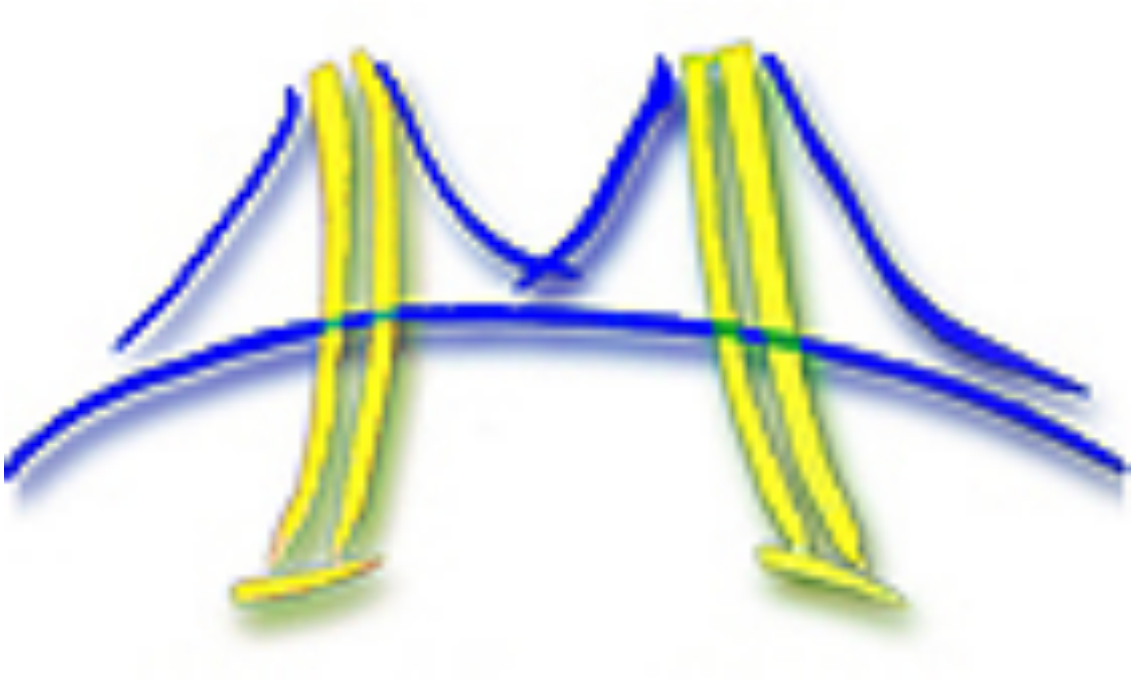


Petascale machine with 8192 procs, each at 500 GFlops/s, a bandwidth of 4 GB/s.

$$\gamma = 2 \cdot 10^{-12} s, \alpha = 10^{-5} s, \beta = 2 \cdot 10^{-9} s / word.$$

Related Work and Contributions for Dense Linear Algebra

- Related work
  - Pothen & Raghavan (1989)
  - Toledo (1997), Rabani & Toledo (2001)
  - Dunha, Becker & Patterson (2002)
  - Gunter & van de Geijn (2005)
- Our contributions
  - QR: 2D parallel, efficient 1D parallel QR
  - Unified approach to sequential, parallel, …
  - Parallel LU
  - Communication lower bounds, proving optimality



Heterogeneous (CPU+GPU) Performance Libraries

Vasily Volkov

Challenges in programming GPUs

Relatively little tuning information for GPU hardware

- Hardware exposed via abstract programming model
- Hard to develop accurate performance models
- Hardware changing rapidly

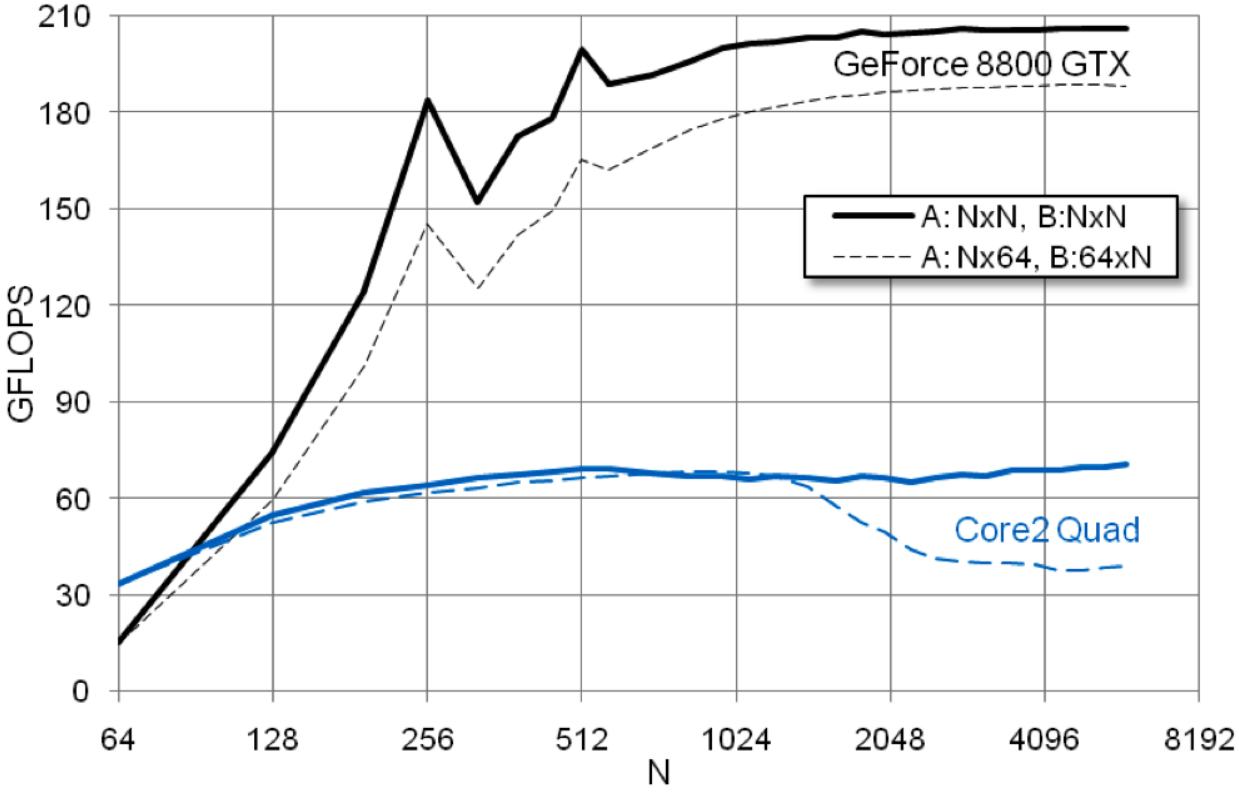
We get ~2x speedups vs. best other work on variety of libraries

- Matrix-matrix multiply (1.6x speedup)
- FFT (2x speedup)
- LU/Cholesky factorization (3x/2x speedups)
- Tridiagonal eigenvalue solver (2x speedup)

Example: partial pivoting in LU factorization on NVIDIA GPU

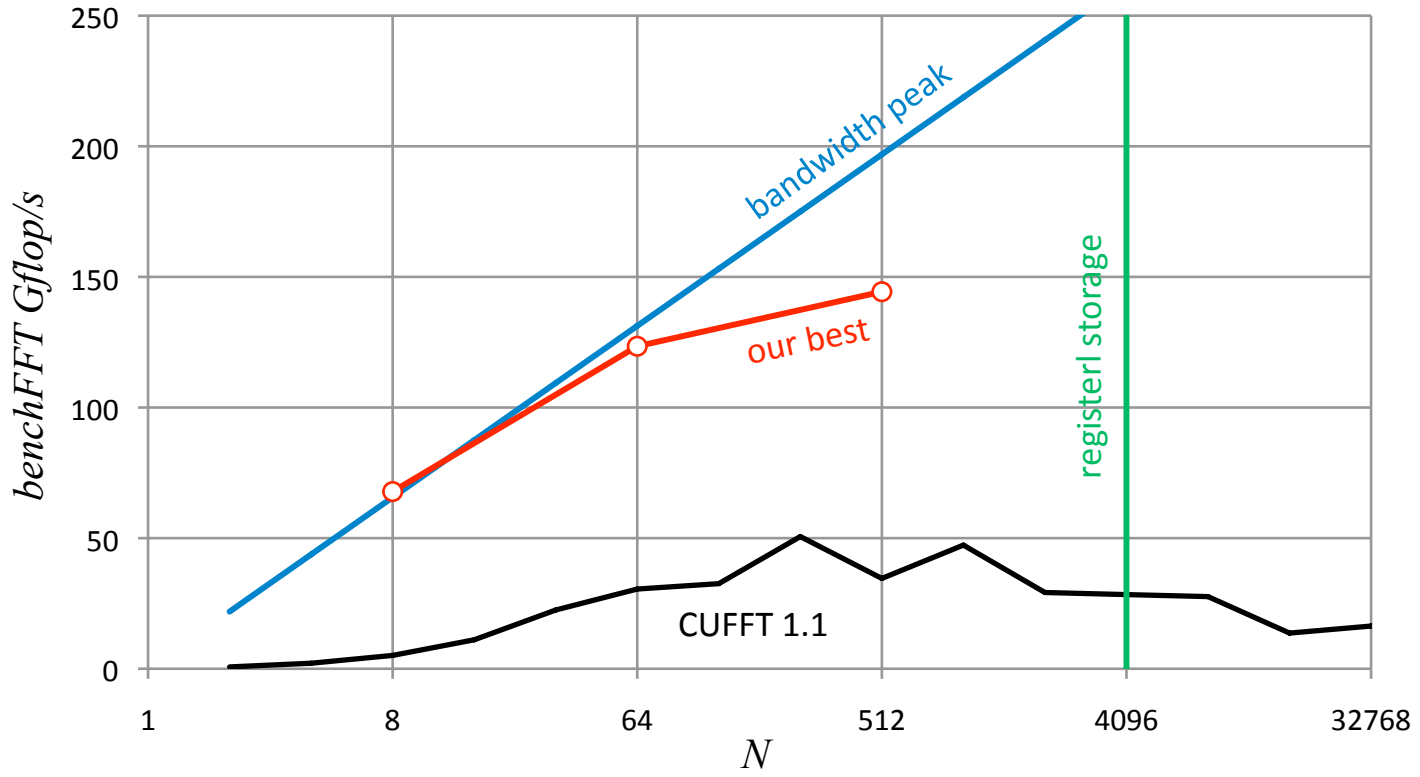
- Ad: “GPU supports gather/scatter with arbitrary access patterns”
- LAPACK code (sgetrf) spends 50% of time doing pivoting
- Gets worse with larger matrices
- Only 1% of time is spent in pivoting if matrix is transposed
    - 2x speedup!

Matrix-matrix multiply (SGEMM)



- Uses software prefetching, strip-mining, register blocking
- Resembles algorithms used for Cray X1 and IBM 3090 VF
- 60% of peak, bound by access to on-chip shared memory

FFT, complex-to-complex, batched



Results as on NVIDIA GeForce 8800GTX
Resembles algorithms used for vector processors
Radix-8 in registers, transposes using shared memory

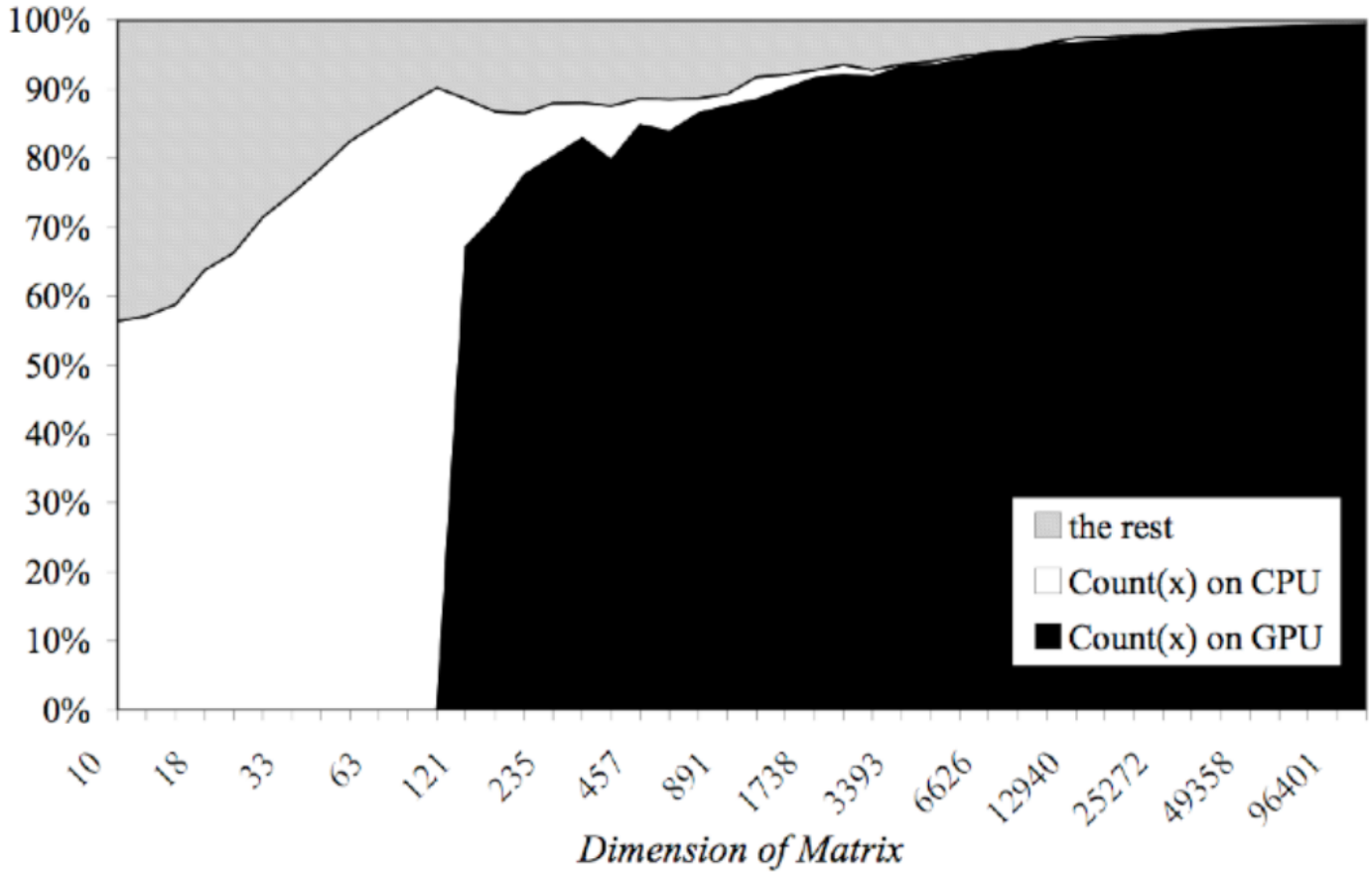
Heterogeneous Computing

Question: should you port entire applications to GPU? –
Maybe not

Example: Eigenvalue solver (bisection, LAPACK's sstebz)
- Most work ($O(n^2)$) in vectorized, embarrassingly parallel code
  - Do it on GPU and finely tune
  - May do many redundant flops on GPU to go faster
    - multisection
  - Run on CPU when problem too small for GPU
  - Use performance model to choose CPU or GPU version
- Little work ($O(n)$) in nearly serial code
  - Do it on CPU: time-consuming and non-trivial to port

Independent project:
  - Did everything on the GPU
  - Used advanced parallel algorithms for O(n) work
  - Up to 2x slower running on faster GPU (compared to us)

Breakdown of Runtime (sstebz)



"Count(x)" is the $O(n^2)$ work, use either SSE or GPU
"the rest" is the $O(n)$ work, isn't worth optimizing

LU Factorization

LU factorization:

- $O(n^3)$ work in bulky BLAS3 operations
- $O(n^2)$ work in fine-grain BLAS1 / BLAS2 (panel factorization)

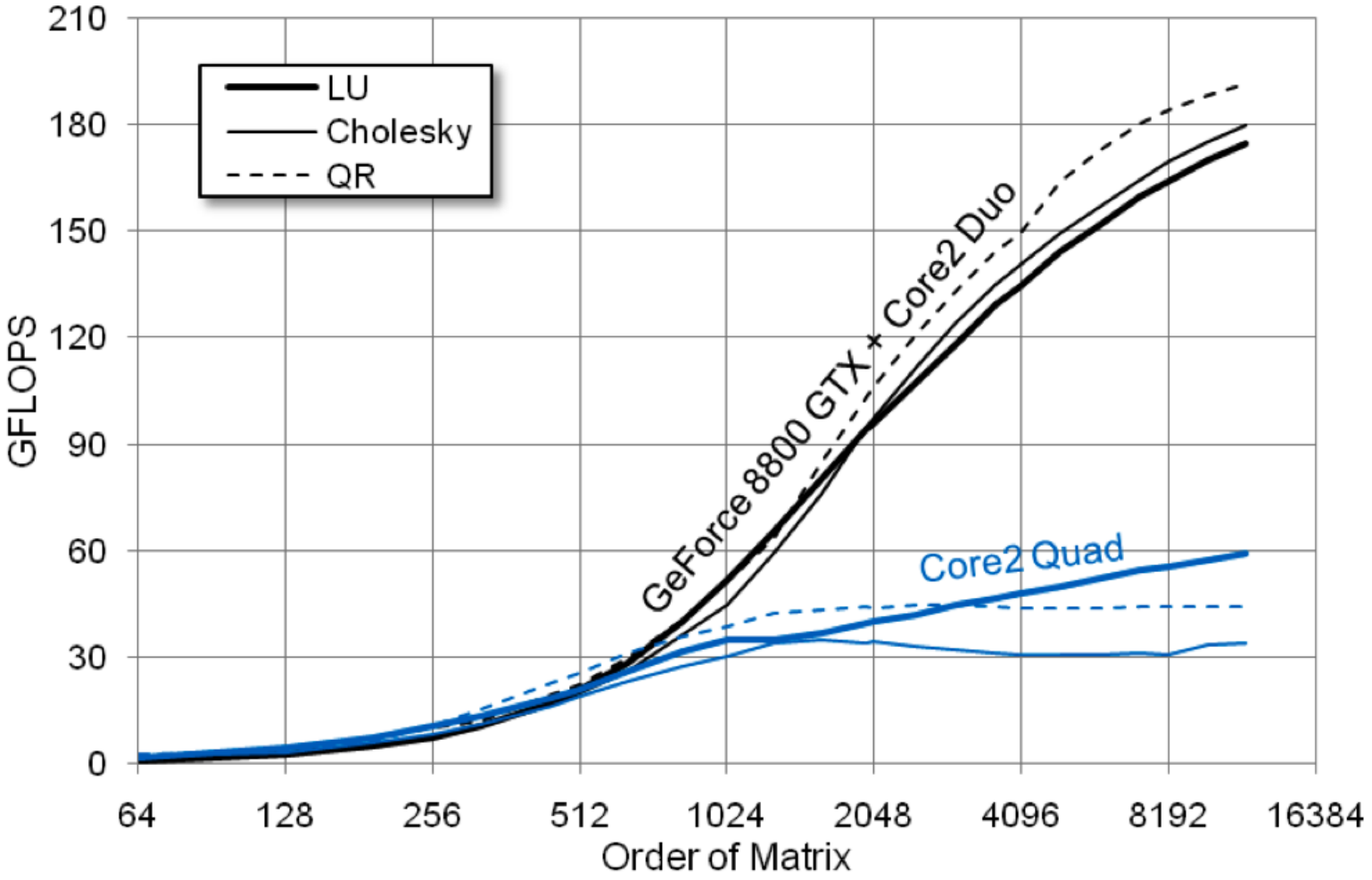
Panel factorization is usually faster on the CPU

- Peak sustained bandwidth of 8800GTX is 75GB/s
- Kernel launch overhead is 5μs
- BLAS1 and BLAS2 are bandwidth bound, so run at

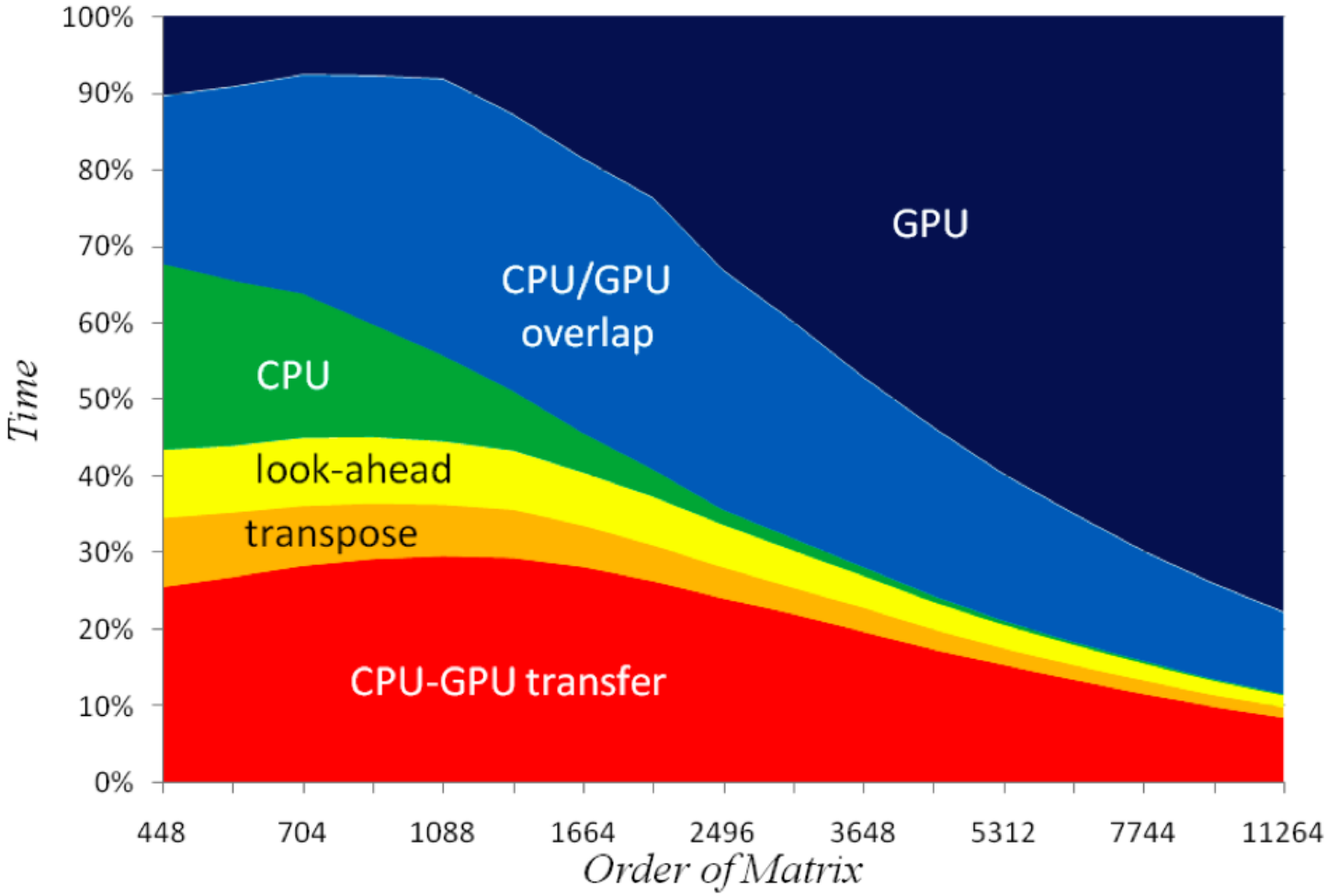
$$Time = 5\mu s + \frac{bandwidth\ \ required}{75GB/s}$$

- compare vs. CPU handicapped by CPU-GPU transfers
  - Faster on CPU up to n ~ 16K on NVIDIA 8800GTX

Overall Performance

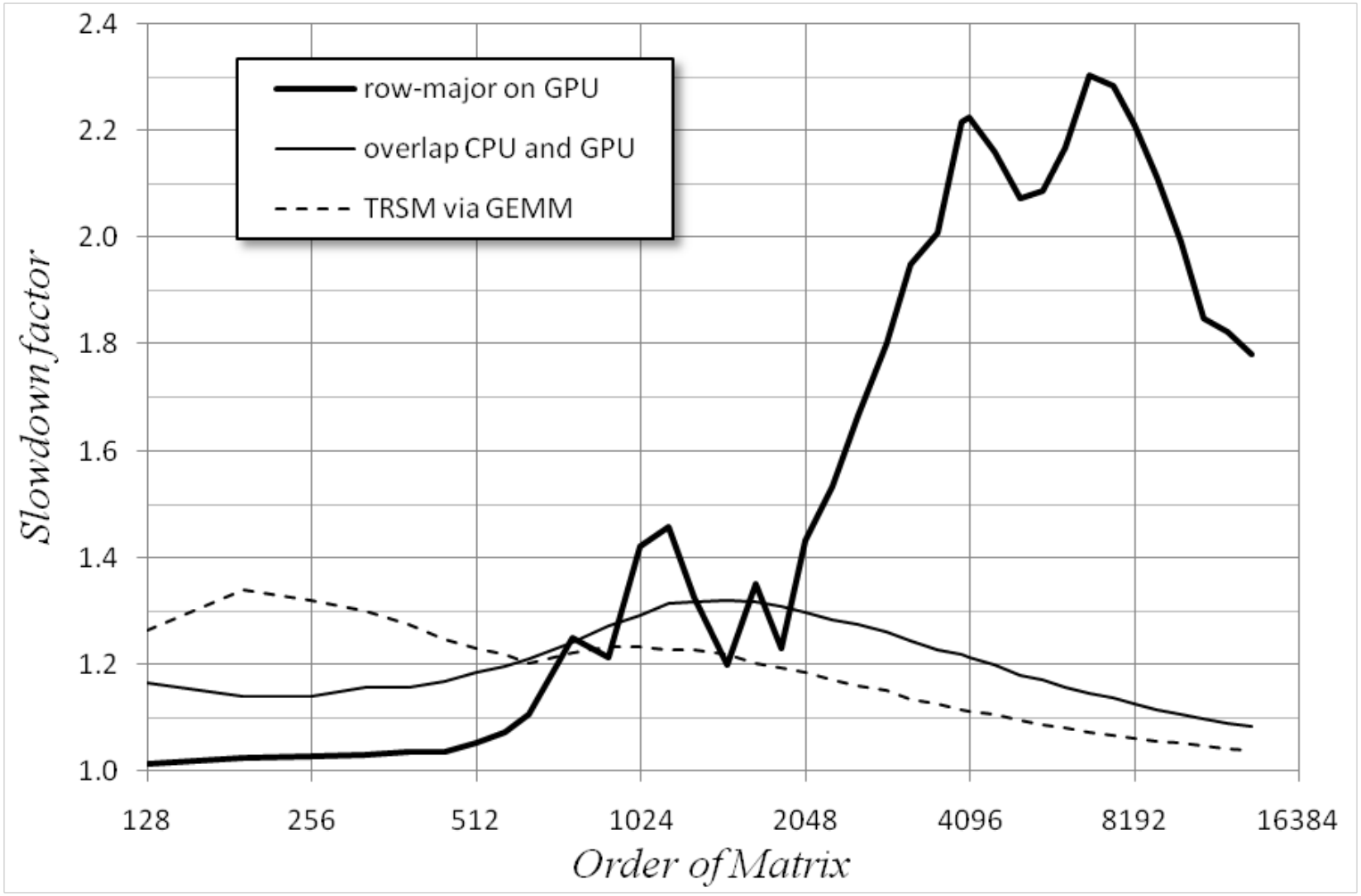


Overlap in LU factorization



Overlap panel factorization on CPU with sgemm on GPU

LU slowdowns from omitting optimizations



Summary of GPU Experience

- Heterogeneity expands tuning space
  - Dividing work between CPU and GPU
    - Depends a lot on flop-rate/bandwidth/latency/startup
  - Different data layouts may be best for CPU and GPU
    - Rowwise vs columnwise for LU
  - May want to do (many) extra flops on GPU
    - Multisection vs Bisection in eigensolver
    - TRINV + TRMM vs TRSM in LU
- Rapid evolution of GPU architectures motivates autotuning

Why all our problems are solved for dense linear algebra– in theory

- Thm (D., Dumitriu, Holtz, Kleinberg) (Numer.Math. 2007)
  - Given any matmul running in $O(n^{\omega})$ ops for some $\omega>2$, it can be made stable and still run in $O(n^{\omega+\varepsilon})$ ops, for any $\varepsilon>0$.
    - Current record: $\omega \approx 2.38$
- Thm (D., Dumitriu, Holtz) (Numer. Math. 2008)
  - Given any stable matmul running in $O(n^{\omega+\varepsilon})$ ops, can do backward stable dense linear algebra in $O(n^{\omega+\varepsilon})$ ops:
    - GEPP, QR (recursive)
    - rank revealing QR (randomized)
    - (Generalized) Schur decomposition, SVD (randomized)
- Also reduces communication to $O(n^{\omega+\varepsilon})$
- But constants?

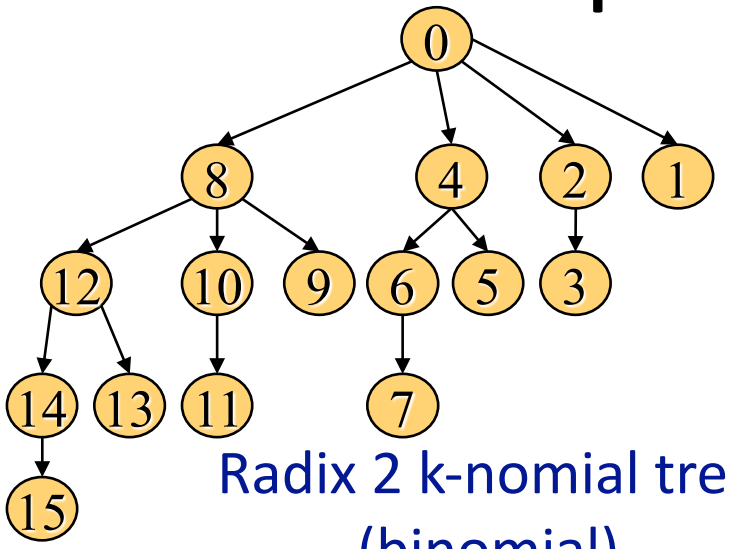
Avoiding Communication in Sparse Linear Algebra - Summary

- Take *k* steps of Krylov subspace method
  - GMRES, CG, Lanczos, Arnoldi
  - Assume matrix "well-partitioned," with modest surface-to-volume ratio
  - Parallel implementation
    - Conventional: O(k log p) messages
    - **"New"**: O(log p) messages - optimal
  - Serial implementation
    - Conventional: O(k) moves of data from slow to fast memory
    - **"New"**: O(1) moves of data – optimal
- Can incorporate some preconditioners
  - Hierarchical, semiseparable matrices …
- Lots of speed up possible (modeled and measured)
  - Price: some redundant computation

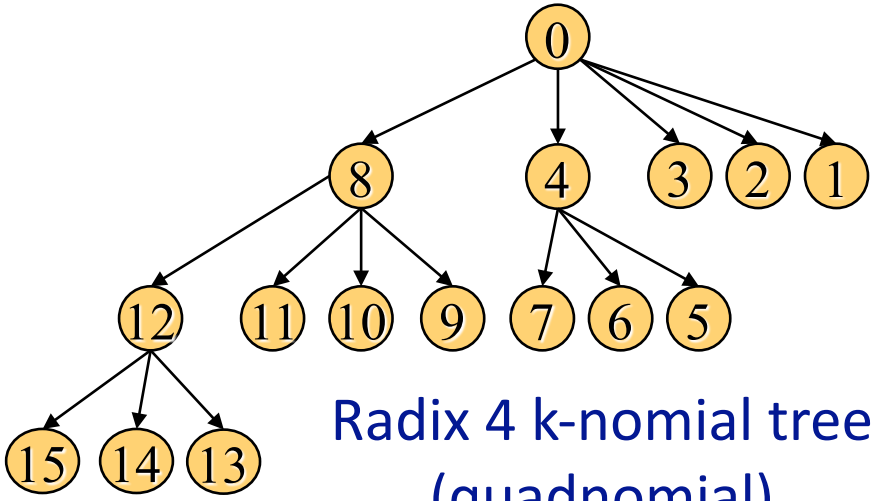
Tuning Collectives

- Rajesh Nishtala
- Collectives called by all threads to perform globally communication
  - Broadcase, reduction, all-to-all
- Need to tune because best algorithm depends on
  - #procs,
  - Network latency
  - Network bandwidth,
  - Transfer size, etc.
- Focus on PGAS languages and one-sided communication models
  - E.g. UPC, Titanium, Co-Array Fortran

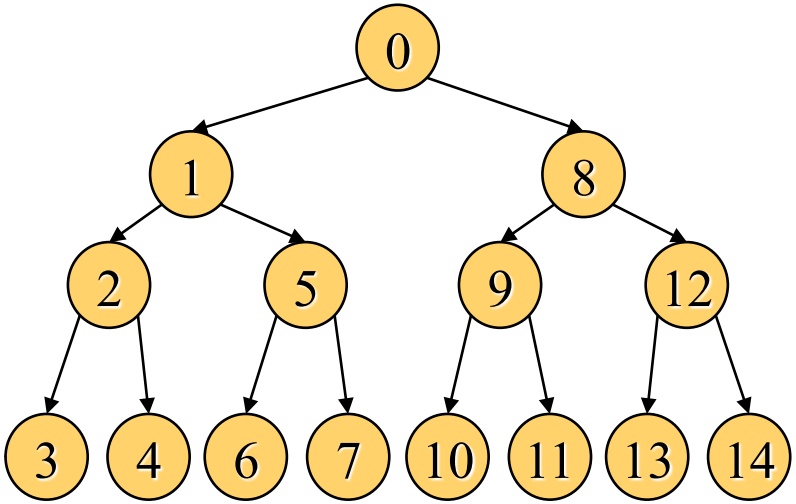
Example Tree Topologies



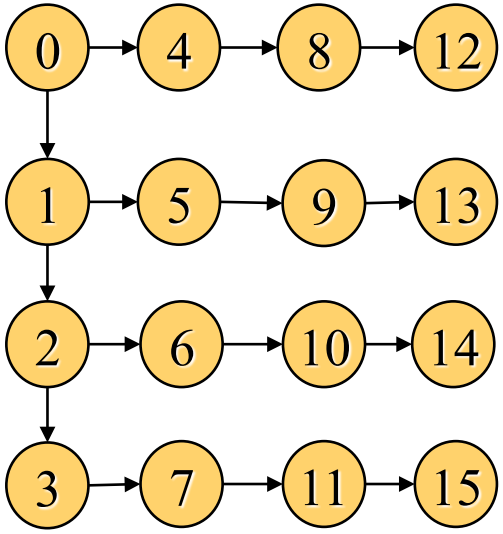
Radix 2 k-nomial tree (binomial)



Radix 4 k-nomial tree (quadnomial)



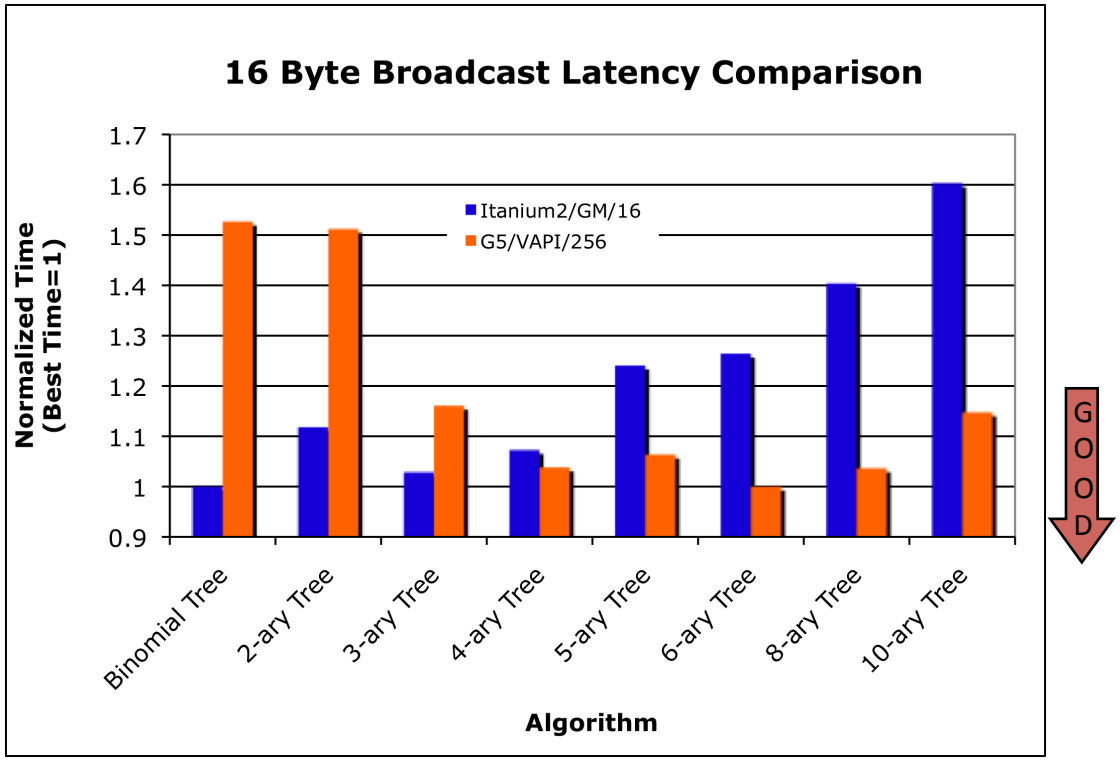
Binary Tree



Fork Tree

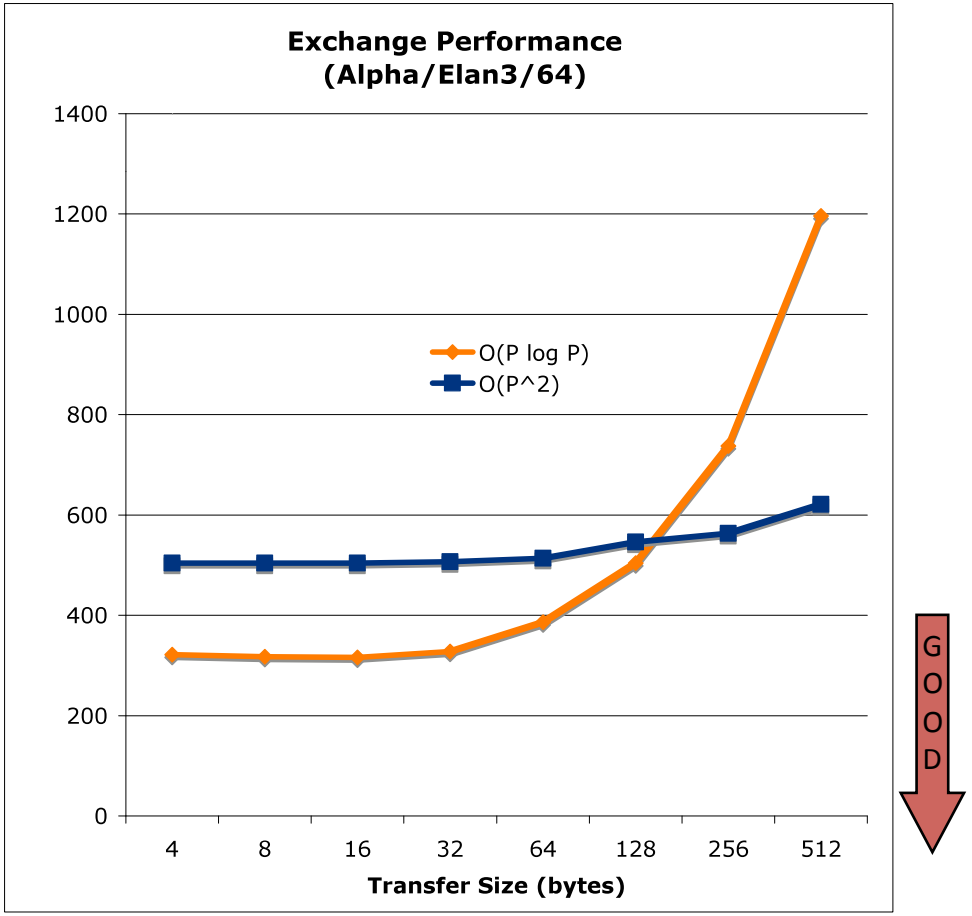
Distributed Memory Broadcast

- 1000 16 Byte broadcasts
- Best implementation is platform specific
- Low cost to inject message ⇒want flat tree (6-ary)
- High cost ⇒ want deep tree to parallelize the overhead (binomial)

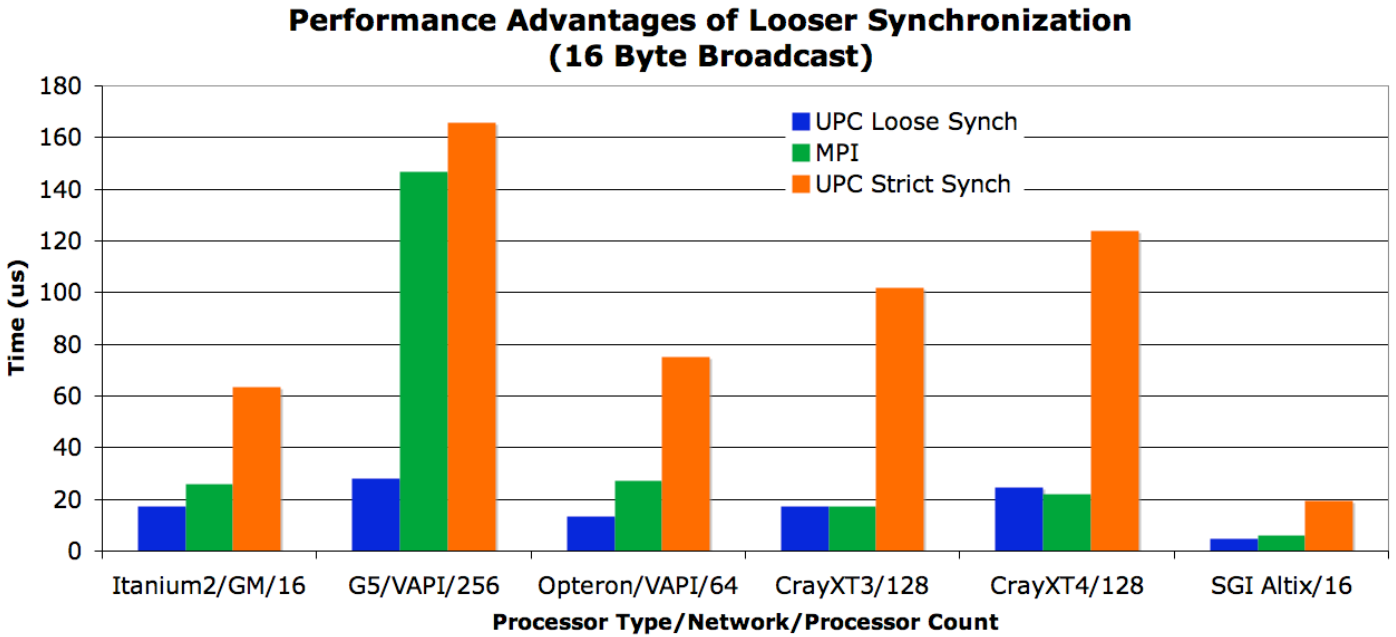


Distributed Memory All-to-All

- Optimal algorithm varies based on transfer size
  - O(P log P) algorithm doubles message size after every round
  - Expensive as the processor count grows
- Cross-over point is platform specific



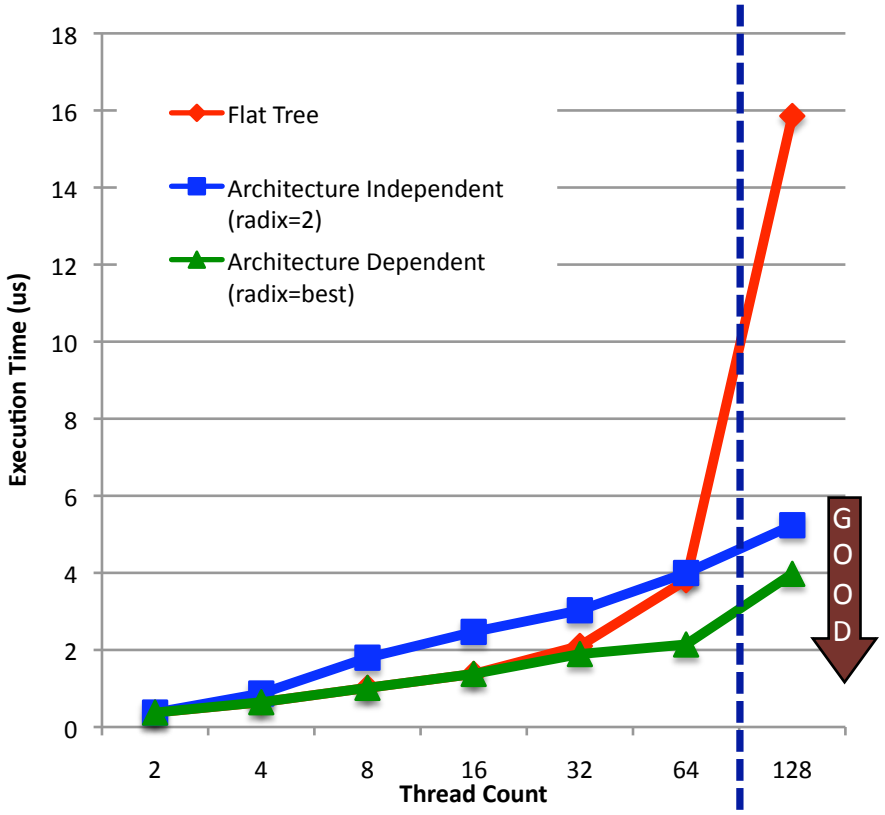
Collective Synchronization



- UPC presents new semantics for collective synchronization
  - Loose synchronization lets collective begin after any thread arrives (looser than MPI semantics)
  - Strict synchronization requires all threads to arrive before communication can start (stricter than MPI)
- Synchronization semantics affects algorithm choice

Multicore Barrier Performance Results

- Time many back-to-back barriers on Dual Socket Sun Niagra2 (128 hardware threads)
- Flat tree is just one level with all threads reporting to thread 0
  - Leverages shared memory but non -scalable
- Architecture Independent Tree (radix=2)
  - Pick a generic "good" radix that is suitable for many platforms
  - Mismatched to architecture
- Architecture Dependent Tree
  - Search overall radices to pick the tree that best matches the architecture
  - Search revealed radix 8 to be the best



Niagara2 (Maramba) Barrier Performance

Summary and Conclusions (1/2)

- Possible to minimize communication complexity of much dense and sparse linear algebra
  - Practical speedups
  - Approaching theoretical lower bounds
- Hardware trends mean the time has come to do this
- Optimal asymptotic complexity algorithms for dense linear algebra – also lower communication
- Important to optimize communication collectives themselves

Summary and Conclusions (2/2)

- Many open problems
  - How to add these to automatic tuning design space
    - PLASMA (Jakub Kurzak)
  - Extend optimality proofs, algorithms to more general architectures
    - Heterogeneity, including multiple bandwidths and latencies
  - Dense eigenvalue problems – SBR or spectral D&C?
  - Sparse direct solvers – CALU or SuperLU?
  - Which preconditioners work?
  - Other motifs

Answers to Questions from Organizers (1/5)

- What about tuning on petascale?
  - Communication avoiding algorithms help address it
  - Can leverage multicore tuning, if we can mirror hierarchy of machines in hierarchy of algorithms/tuners
- How do we measure success for tuning?
  - Performance gains attained
  - Ease of use by end-user
  - Ease of development/evolution/maintenance of tuners
- What architectures should we target?
  - Multicore and Petascale as above,
  - Emerging architectures like GPUs

Answers to Questions from Organizers (2/5)

- T/F: “Parameter Tuning” is not enough
  - Many of us have been changing data structures and algorithms for a while, which goes beyond “parameters”
- T/F: Self-tuned libraries will beat compilers
  - What is the starting point for the compiler? The most naïve possible code? Then yes (but may work for some motifs)
  - But autotuners use compilers to compile tuned source code
- Will compilers change data structures/algorithms?
  - Would they still be called compilers or something else?
  - How much wisdom about algorithms, numerical analysis and applied math can we expect compilers to incorporate?

Answers to Questions from Organizers (3/5)

- Simple performance models will make search unnecessary, eg “cache-oblivious”
  - If search is easy compared to figuring out a “simple” performance model, it is more productive
  - “Cache oblivious” would not have found new communication avoiding algorithms
  - Simple models help decide when to stop tuning
- Boundaries between SW layer too rigid
  - Yes! See ParLab structure for one take on breaking usual layering of the HW/SW stack, eg schedulers

Answers to Questions from Organizers (4/5)

- What issues/technologies are we ignoring?
  - Need more compiler-based expertise to build autotuners, so they are easier to design/develop/maintain, and not just big PERL scrips that need to be written from scratch for each new architecture or slightly different function
- What common tools should we build?
  - See answer to last question; see Pluto, talk by Chen
- Tuning systems are too narrow, invest in compilers instead
  - If compilers do not change data structures or algorithms, they will miss a lot of performance gains
  - ParLab hypothesis is that 13 motifs/tuners is enough
  - Invest in compiler tools to help build autotuners

Answers to Questions from Organizers (5/5)

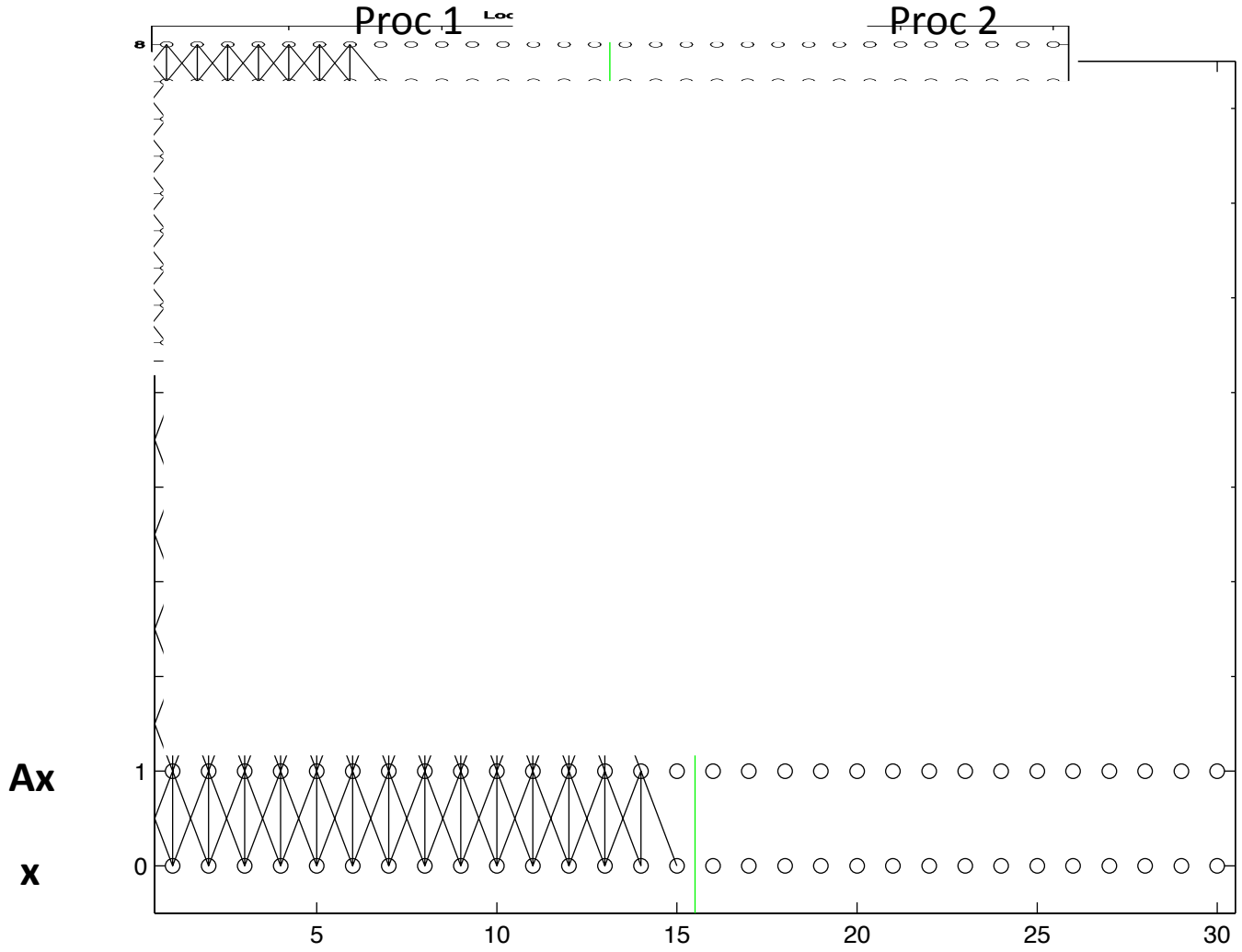
- Runtime optimization is needed
  - Yes! A number of autotuners do this now, eg JIT, OSKI
- If all layers of the SW stack are autotuned, how will they be composed?
  - A big question for ParLab too! One answer for multicore is in having schedulers for each library that can accept or give up resources (cores) on the fly so that higher level schedulers can balance work at a higher level

EXTRA SLIDES

Avoiding Communication in Sparse Linear Algebra - Summary

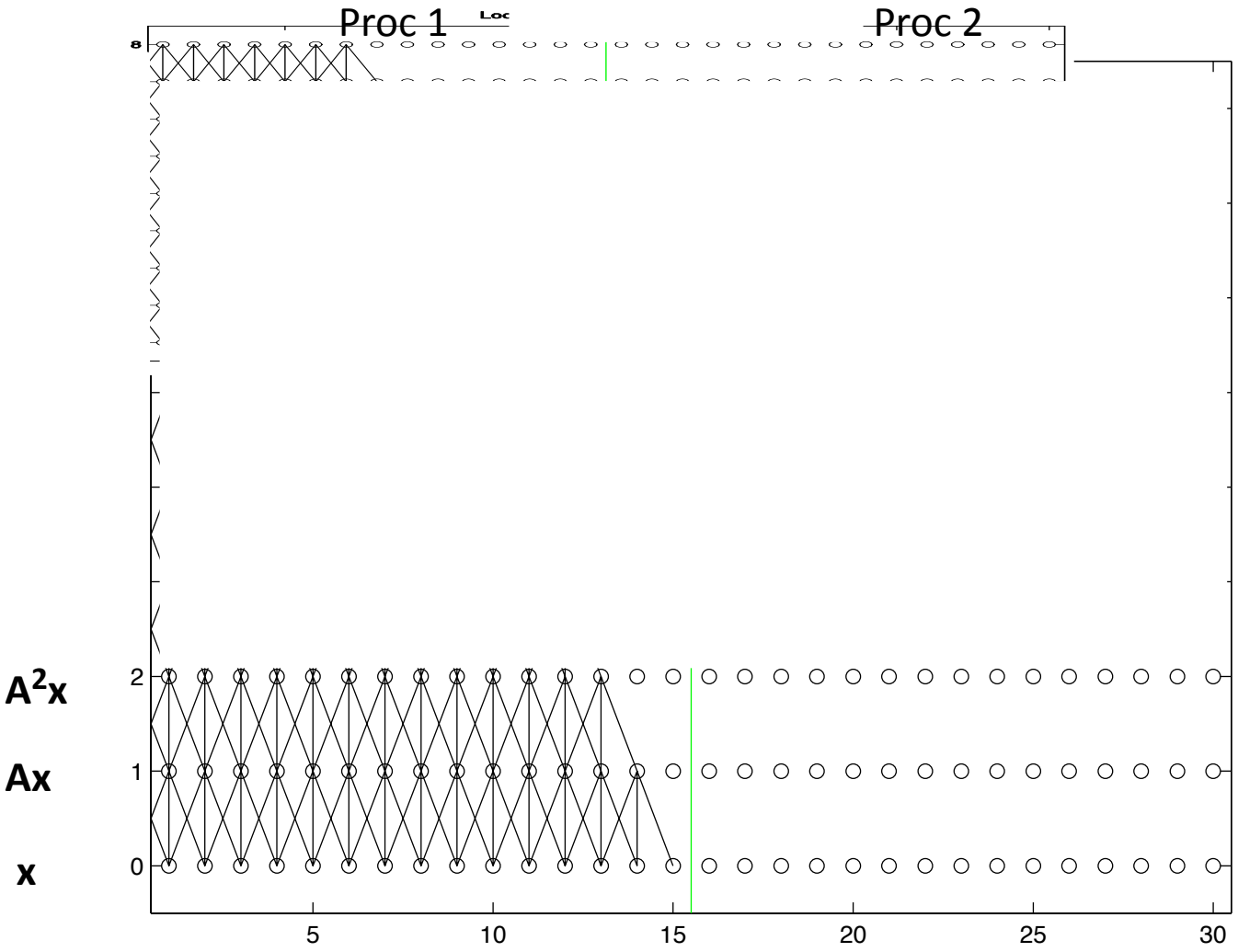
- Take *k* steps of Krylov subspace method
  - GMRES, CG, Lanczos, Arnoldi
  - Assume matrix "well-partitioned," with modest surface-to-volume ratio
  - Parallel implementation
    - Conventional: O(k log p) messages
    - **"New"**: O(log p) messages - optimal
  - Serial implementation
    - Conventional: O(k) moves of data from slow to fast memory
    - **"New"**: O(1) moves of data – optimal
- Can incorporate some preconditioners
  - Hierarchical, semiseparable matrices …
- Lots of speed up possible (modeled and measured)
  - Price: some redundant computation

Locally Dependent Entries for [x,Ax], A tridiagonal, 2 processors



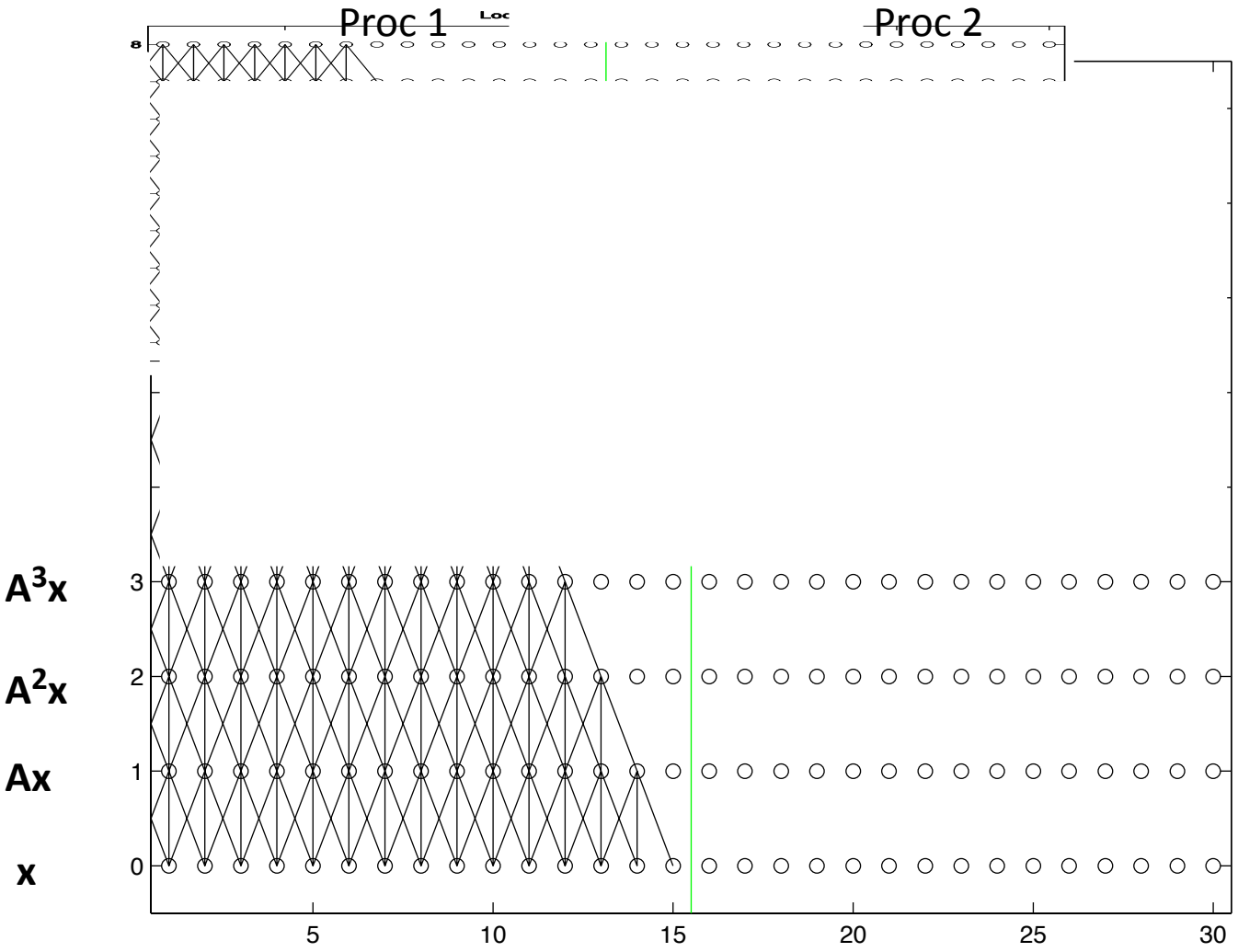
Can be computed without communication

Locally Dependent Entries for [x,Ax,A^2x], A tridiagonal, 2 processors



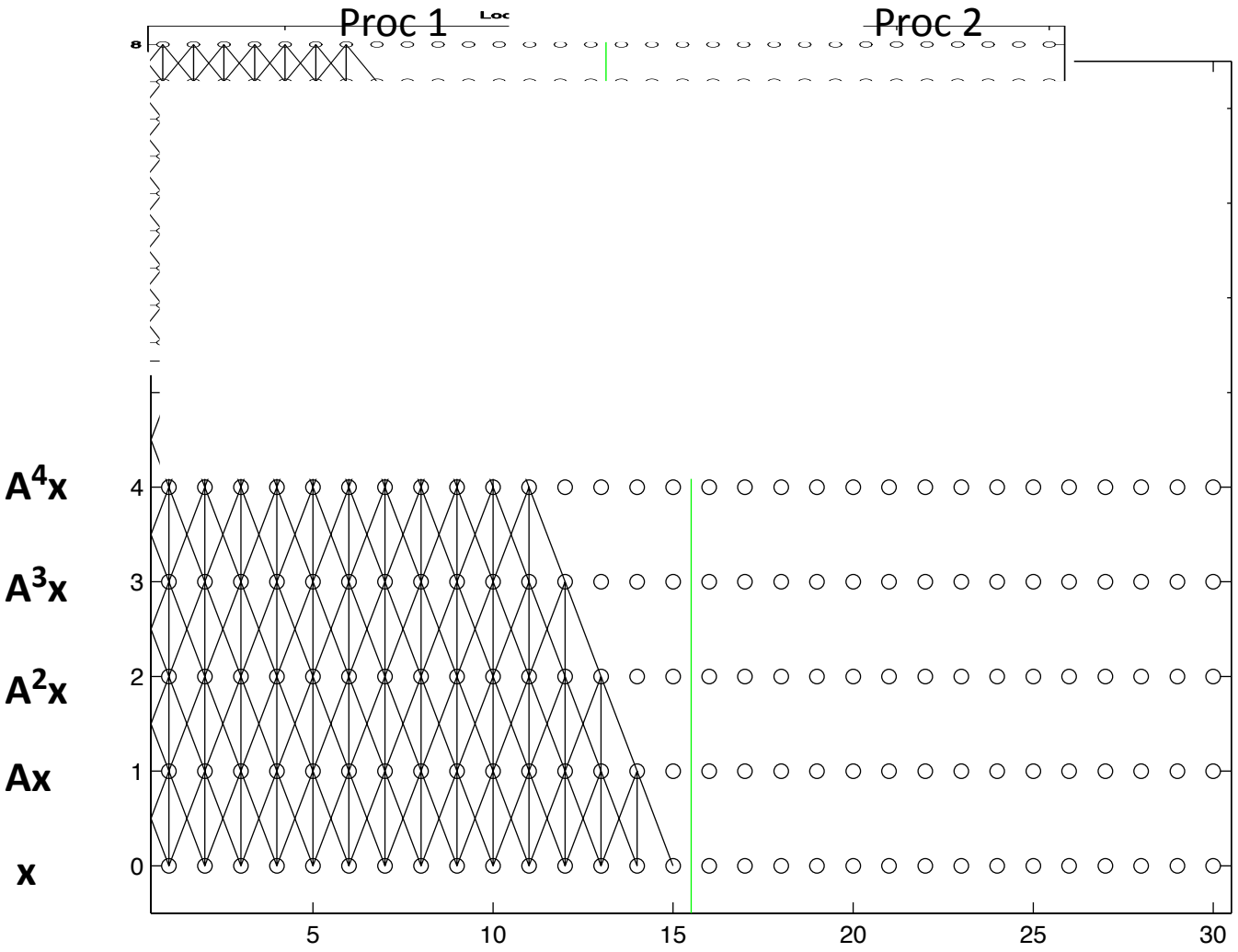
Can be computed without communication

Locally Dependent Entries for [x,Ax,…,A^3x], A tridiagonal, 2 processors



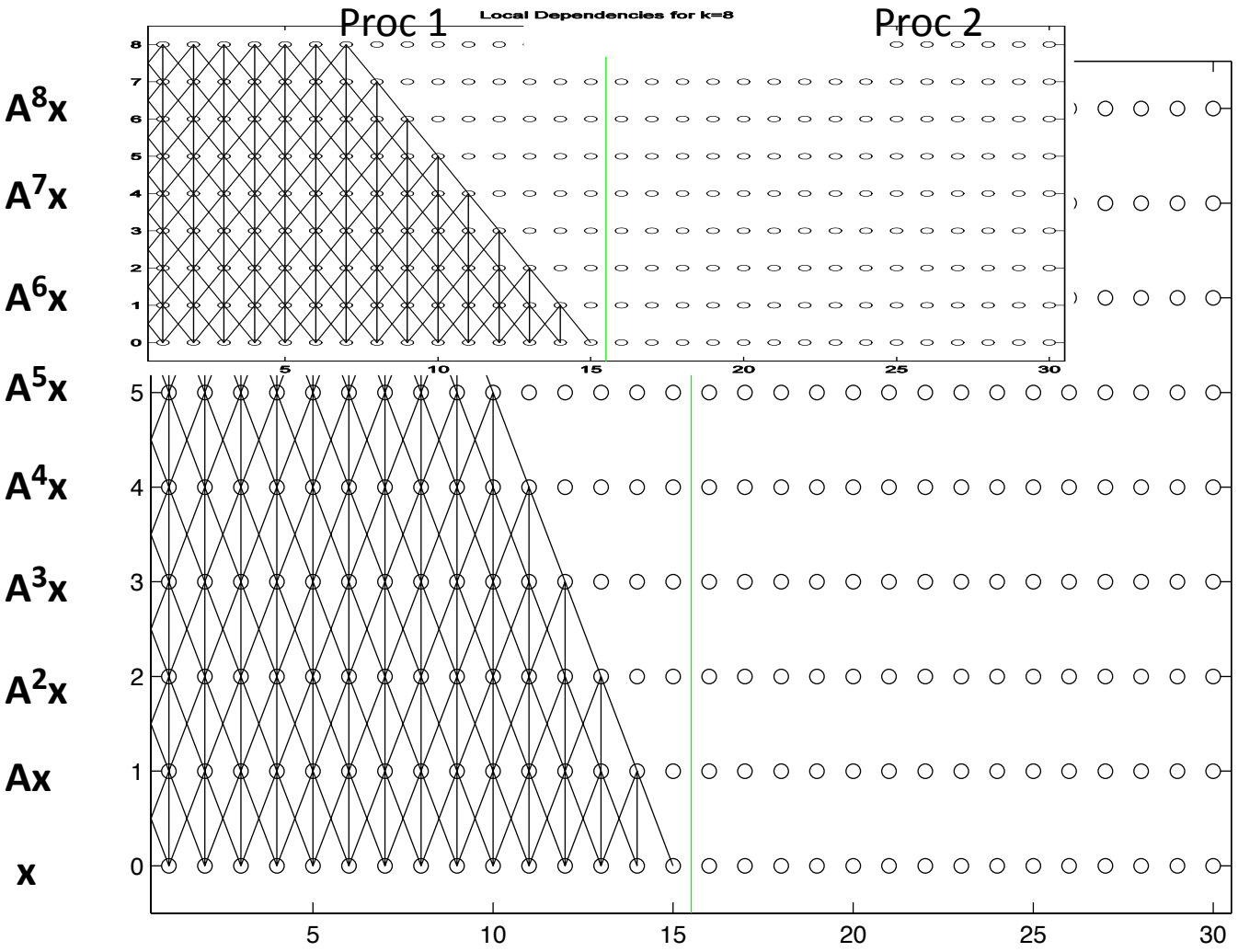
Can be computed without communication

Locally Dependent Entries for [x,Ax,…,A^4x], A tridiagonal, 2 processors



Can be computed without communication

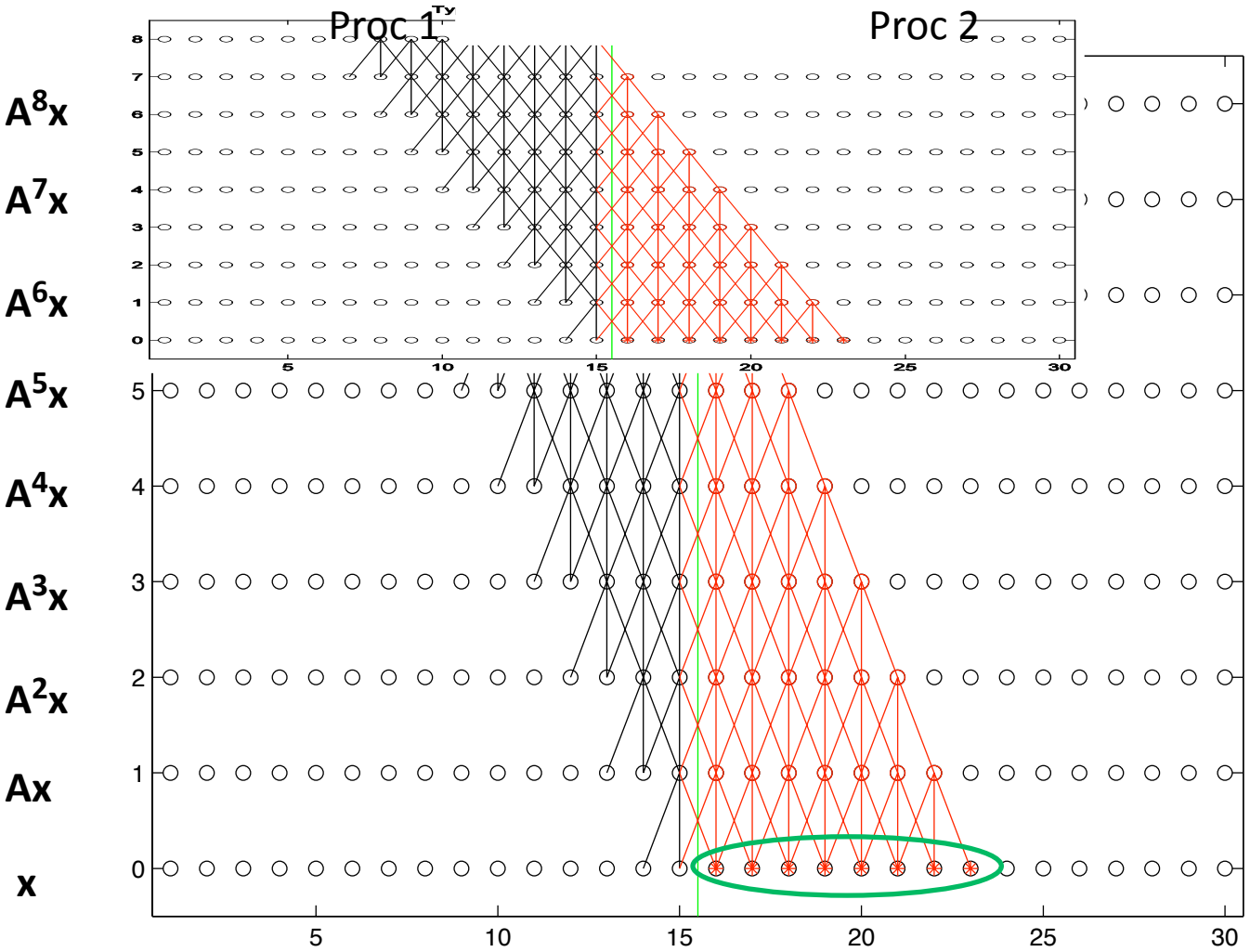
Locally Dependent Entries for [x,Ax,…,A^8x], A tridiagonal, 2 processors



Can be computed without communication

k=8 fold reuse of A

Remotely Dependent Entries for [x,Ax,…,A^{8}x], A tridiagonal, 2 processors

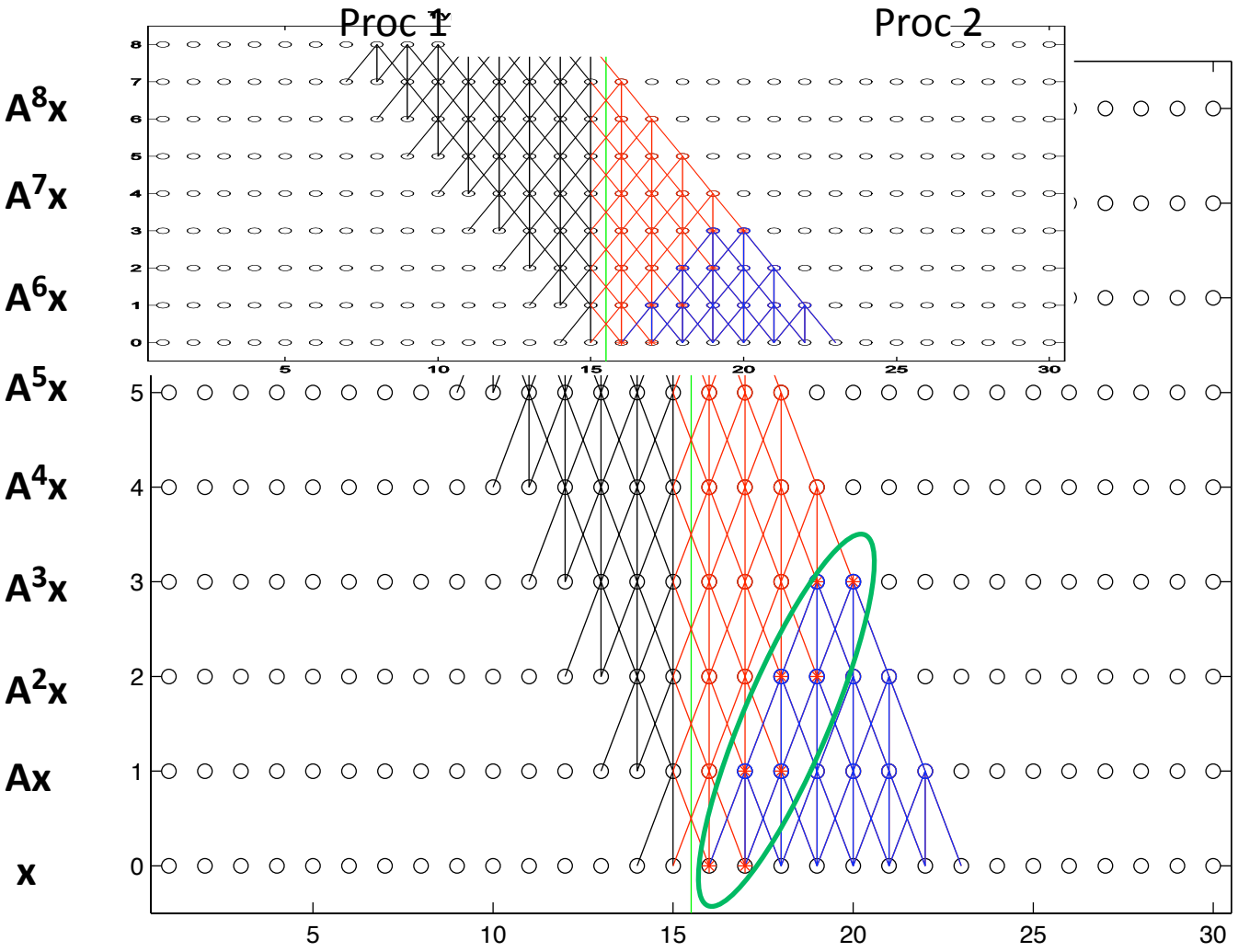


One message to get data needed to compute remotely dependent entries, ***not k=8***

Minimizes number of messages = latency cost

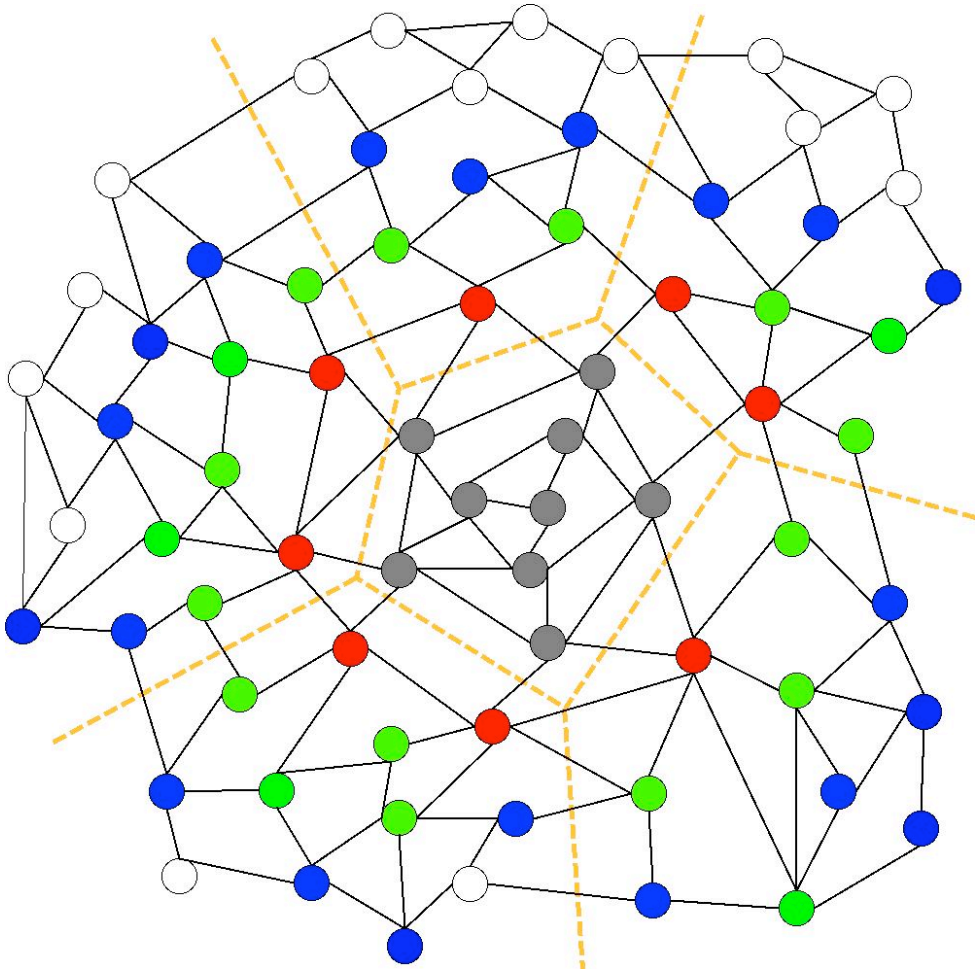
Price: redundant work ∝ "surface/volume ratio"

Fewer Remotely Dependent Entries for [x,Ax,…,A^8x], A tridiagonal, 2 processors

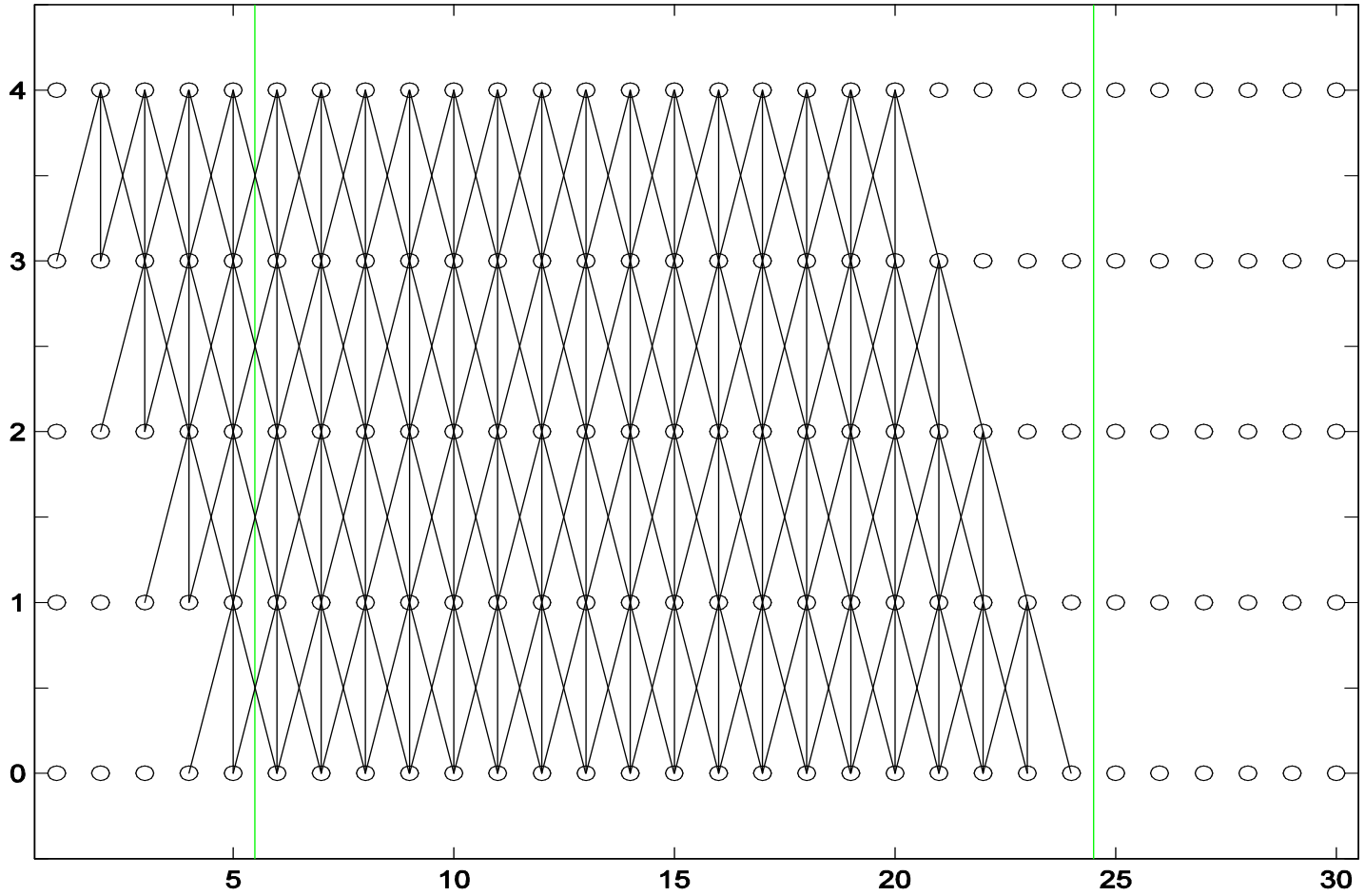


Reduce redundant work by half

Remotely Dependent Entries for $[x, Ax, A^2x, A^3x]$, A irregular, multiple processors



Sequential [x,Ax,…,A^4x], with memory hierarchy



One read of matrix from slow memory, ***not k=4***

Minimizes words moved = bandwidth cost

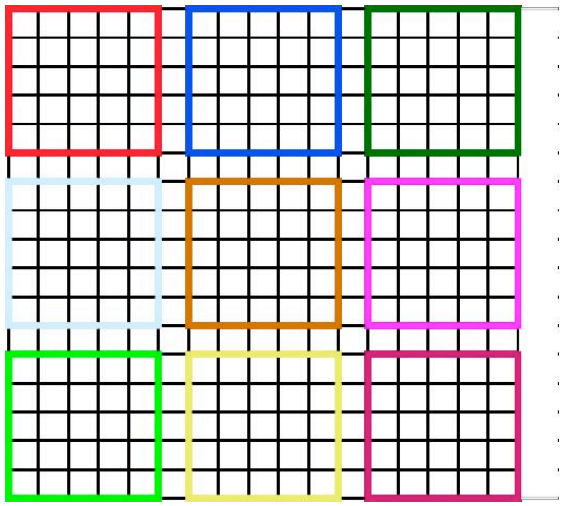
No redundant work

Performance Results

- Measured
  - Sequential/OOC speedup up to **3x**
- Modeled
  - Sequential/multicore speedup up to **2.5x**
  - Parallel/Petascale speedup up to **6.9x**
  - Parallel/Grid speedup up to **22x**
- See bebop.cs.berkeley.edu/#pubs

Optimizing Communication Complexity of Sparse Solvers

- Example: GMRES for $Ax=b$ on "2D Mesh"
  - x lives on n-by-n mesh
  - Partitioned on $p^{1/2}$ -by- $p^{1/2}$ grid
  - A has "5 point stencil" (Laplacian)
    - (Ax)(i,j) = linear_combination(x(i,j), x(i,j±1), x(i±1,j))
  - Ex: 18-by-18 mesh on 3-by-3 grid

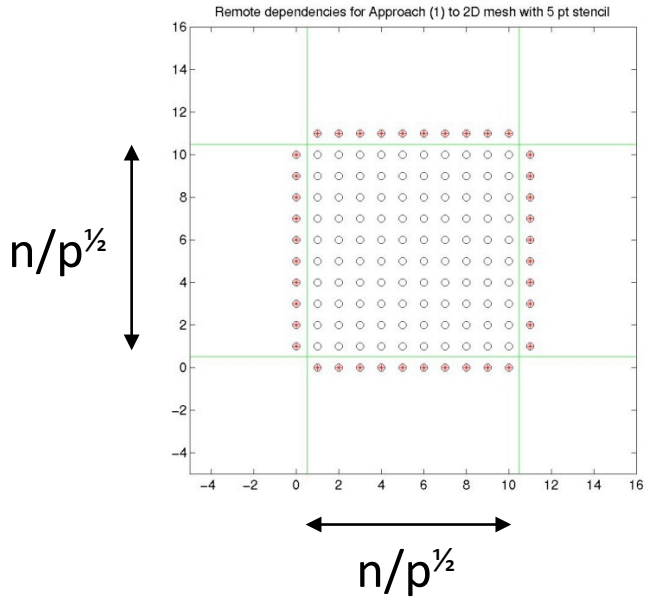


Minimizing Communication of GMRES

- What is the cost = (#flops, #words, #mess) of k steps of standard GMRES?

GMRES, ver.1:

```
for i=1 to k
   w = A * v(i-1)
   MGS(w, v(0),…,v(i-1))
   update v(i), H
endfor
solve LSQ problem with H
```



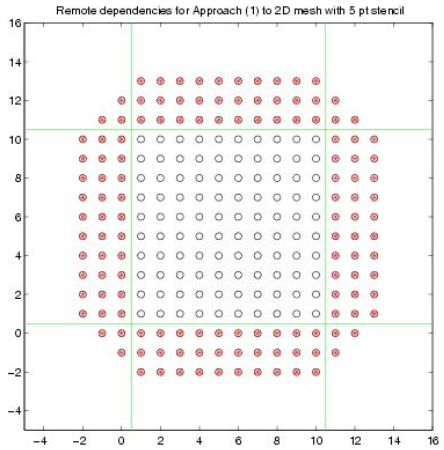
- Cost(A * v) = $k * (9n^2/p, 4n / p^{½}, 4)$
- Cost(MGS) = $k^2/2 * (4n^2/p, \log p, \log p)$
- Total cost ~ Cost(A * v) + Cost (MGS)
- Can we reduce the latency?

Minimizing Communication of GMRES

- Cost(GMRES, ver.1) = Cost(*A*v*) + Cost(MGS)
  = ($9kn^2/p$, $4kn/p^{1/2}$, 4k) + ($2k^2n^2/p$, $k^2 \log p / 2$, $k^2 \log p / 2$)

- How much latency cost from A*v can you avoid? Almost all

GMRES, ver. 2:
W = [v, Av, A^2v, … , A^kv]
[Q,R] = MGS(W)
Build H from R, solve LSQ problem



k = 3

- Cost(W) = (~ same, ~ same , 8)
  - Latency cost independent of k – **optimal**
- Cost (MGS) unchanged
- Can we reduce the latency more?

Minimizing Communication of GMRES

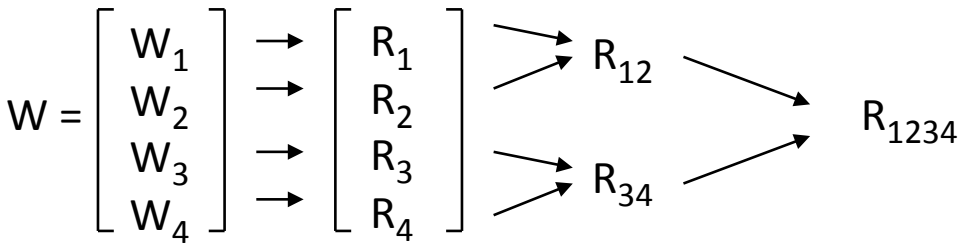
- Cost(GMRES, ver. 2) = Cost(*W*) + Cost(MGS)
  = ($9kn^2/p$, $4kn/p^{1/2}$, 8) + ($2k^2n^2/p$, $k^2 \log p / 2$, $k^2 \log p / 2$)
- How much latency cost from MGS can you avoid? Almost all

GMRES, ver. 3:

W = [v, Av, A^2v, … , A^kv]

[Q,R] = TSQR(W) … "Tall Skinny QR"

Build H from R, solve LSQ problem



- Cost(TSQR) = (~ same, ~ same , log p)
  - Latency cost independent of s - **optimal**

Minimizing Communication of GMRES

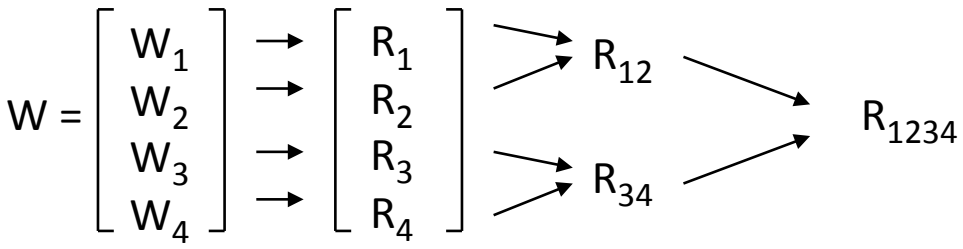
- Cost(GMRES, ver. 2) = Cost(*W*) + Cost(MGS)
  = ($9kn^2/p$, $4kn/p^{1/2}$, 8) + ($2k^2n^2/p$, $k^2 \log p / 2$, $k^2 \log p / 2$)
- How much latency cost from MGS can you avoid? Almost all

GMRES, ver. 3:

W = [v, Av, A^2v, … , A^kv]

[Q,R] = TSQR(W) … "Tall Skinny QR"

Build H from R, solve LSQ problem



- Cost(TSQR) = (~ same, ~ same , log p)
- **Oops**

Minimizing Communication of GMRES

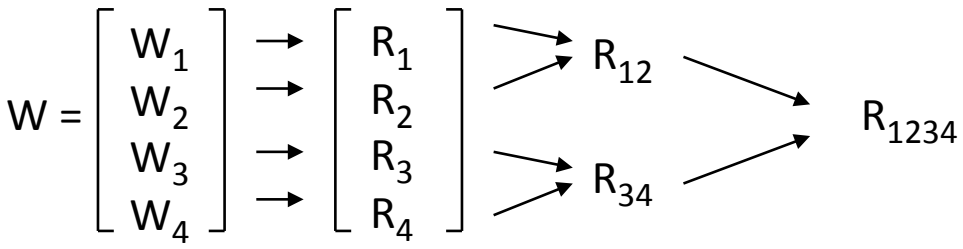
- Cost(GMRES, ver. 2) = Cost(*W*) + Cost(MGS)
  = ($9kn^2/p$, $4kn/p^{1/2}$, 8) + ($2k^2n^2/p$, $k^2 \log p / 2$, $k^2 \log p / 2$)
- How much latency cost from MGS can you avoid? Almost all

GMRES, ver. 3:

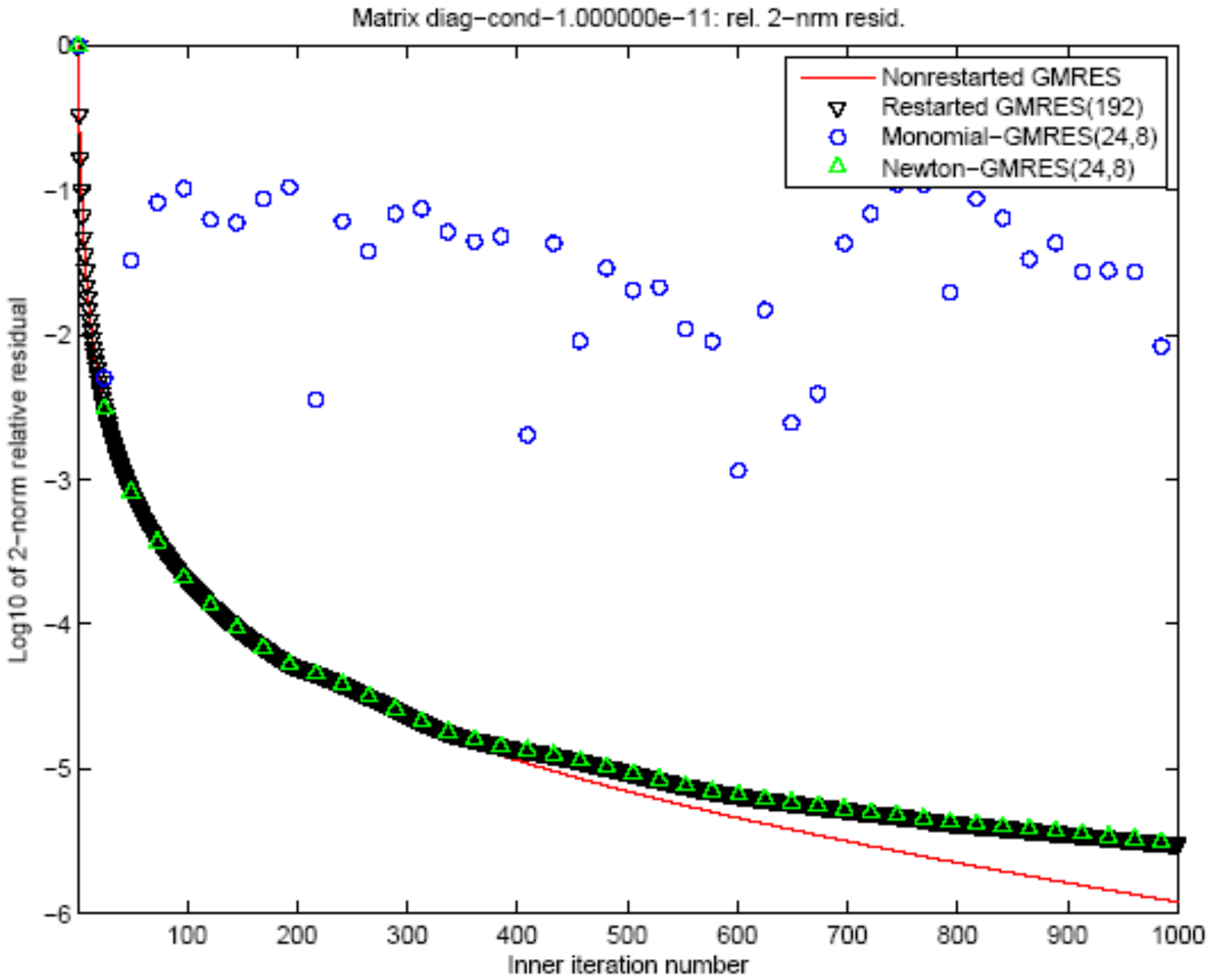
W = [v, Av, A^2v, … , A^kv]

[Q,R] = TSQR(W) … "Tall Skinny QR"

Build H from R, solve LSQ problem



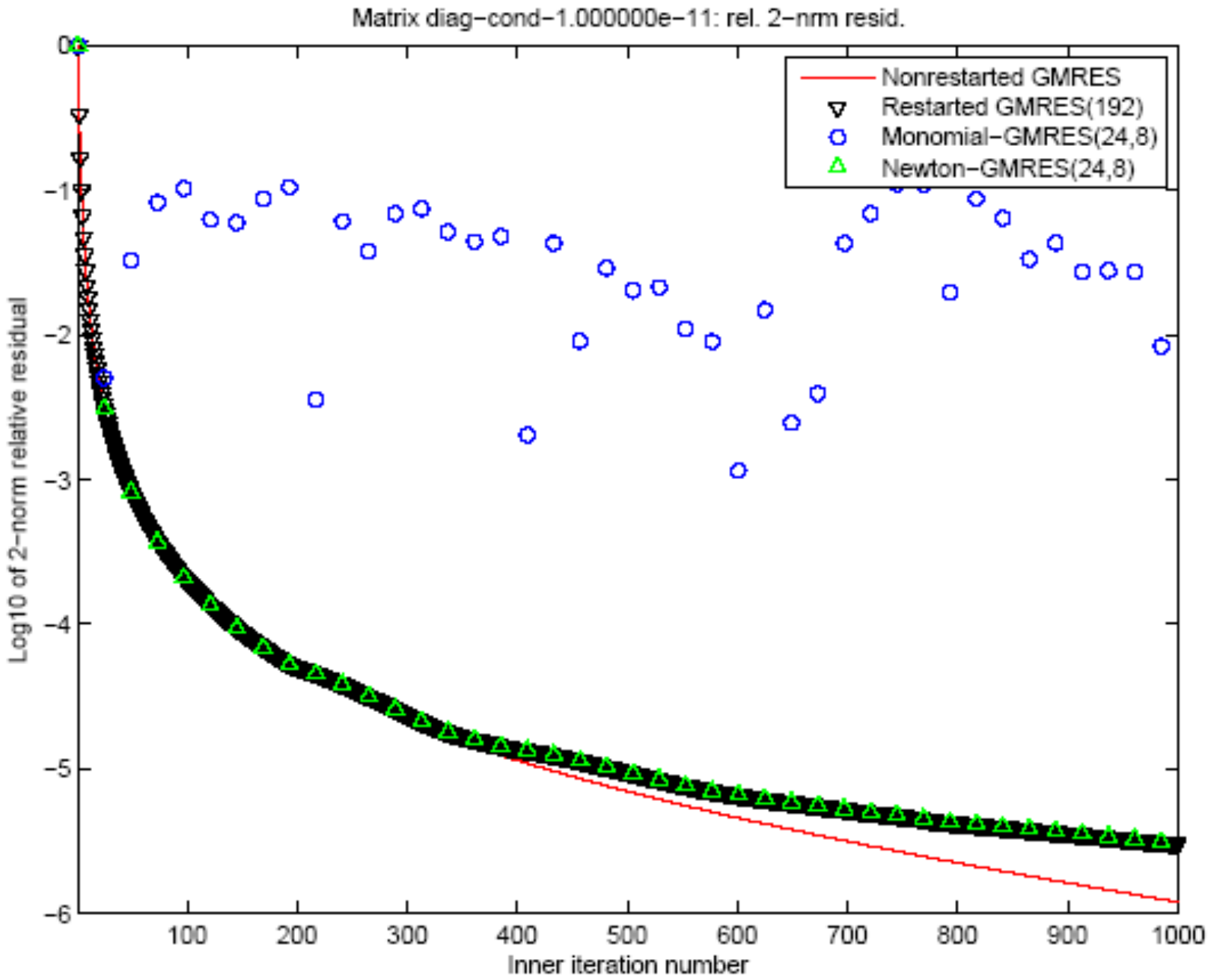
- Cost(TSQR) = (~ same, ~ same , log p)
- **Oops – W from power method, precision lost!**


Matrix diag−cond−1.000000e−11: rel. 2−nrm resid.
Nonrestarted GMRES
Restarted GMRES(192)
Monomial−GMRES(24,8)
Newton−GMRES(24,8)
Log10 of 2−norm relative residual
Inner iteration number
0
−1
−2
−3
−4
−5
−6
100
200
300
400
500
600
700
800
900
1000

Minimizing Communication of GMRES

- Cost(GMRES, ver. 3) = Cost(*W*) + Cost(TSQR)
  = ($9kn^2/p$, $4kn/p^{1/2}$, 8) + ($2k^2n^2/p$, $k^2 \log p / 2$, log p)
- Latency cost independent of k, just log p – optimal
- Oops – W from power method, so precision lost – What to do?

- Use a different polynomial basis
- Not Monomial basis W = [v, Av, A^2v, …], instead …
- Newton Basis W_N = [v, $(A - \theta_1 I)v$, $(A - \theta_2 I)(A - \theta_1 I)v$, …] or
- Chebyshev Basis W_C = [v, $T_1(v)$, $T_2(v)$, …]


Matrix diag−cond−1.000000e−11: rel. 2−nrm resid.
Nonrestarted GMRES
Restarted GMRES(192)
Monomial−GMRES(24,8)
Newton−GMRES(24,8)
Log10 of 2−norm relative residual
Inner iteration number
0
−1
−2
−3
−4
−5
−6
100
200
300
400
500
600
700
800
900
1000

Related Work and Contributions for Sparse Linear Algebra

- Related work
  - s-step GMRES: De Sturler (1991), Bai, Hu & Reichel (1991), Joubert et al (1992), Erhel (1995)
  - s-step CG: Van Rosendale (1983), Chronopoulos & Gear (1989), Toledo (1995)
  - Matrix Powers Kernel: Pfeifer (1963), Hong & Kung (1981), Leiserson, Rao & Toledo (1993), Toledo (1995), Douglas, Hu, Kowarschik, Rüde, Weiss (2000), Strout, Carter & Ferrante (2001)
- Our contributions
  - Unified approach to serial and parallel, use of TSQR
  - Optimizing serial case via TSP
  - Unified approach to stability
  - Incorporating preconditioning

Summary and Conclusions (1/2)

- Possible to minimize communication complexity of much dense and sparse linear algebra
  - Practical speedups
  - Approaching theoretical lower bounds
- Optimal asymptotic complexity algorithms for dense linear algebra – also lower communication
- Hardware trends mean the time has come to do this
- *Lots* of prior work (see pubs) – and some new

Summary and Conclusions (2/2)

- Many open problems
  - Automatic tuning - build and optimize complicated data structures, communication patterns, code automatically: bebop.cs.berkeley.edu
  - Extend optimality proofs to general architectures
  - Dense eigenvalue problems – SBR or spectral D&C?
  - Sparse direct solvers – CALU or SuperLU?
  - Which preconditioners work?
  - Why stop at linear algebra?