New optimal complexity algorithms for linear algebra

Jim Demmel

CScADS

11 July 2007

Collaborators

- Sparse: Kathy Yelick, Mark Hoemmen, Marghoob Mohiyuddin, BEBOP group
- Dense: Ioana Dumitriu, Laura Grigori, Olga Holtz, Robert Kleinberg, Julien Langou, Jessica Schoen, LAPACK group

Outline

- Tuning $(x,A,k) \rightarrow [x,Ax,A^2x,\ldots A^kx]$
- Optimal communication complexity algorithms for sparse linear algebra

- Optimal communication complexity algorithms for dense linear algebra
- Optimal arithmetic complexity algorithms for dense linear algebra

Outline

- Tuning $(x,A,k) \rightarrow [x, Ax, A^2x, \ldots A^kx]$
- Optimal communication complexity algorithms for sparse linear algebra

- Optimal communication complexity algorithms for dense linear algebra
- Optimal arithmetic complexity algorithms for dense linear algebra

- So what if they're optimal, are they fast?

Outline

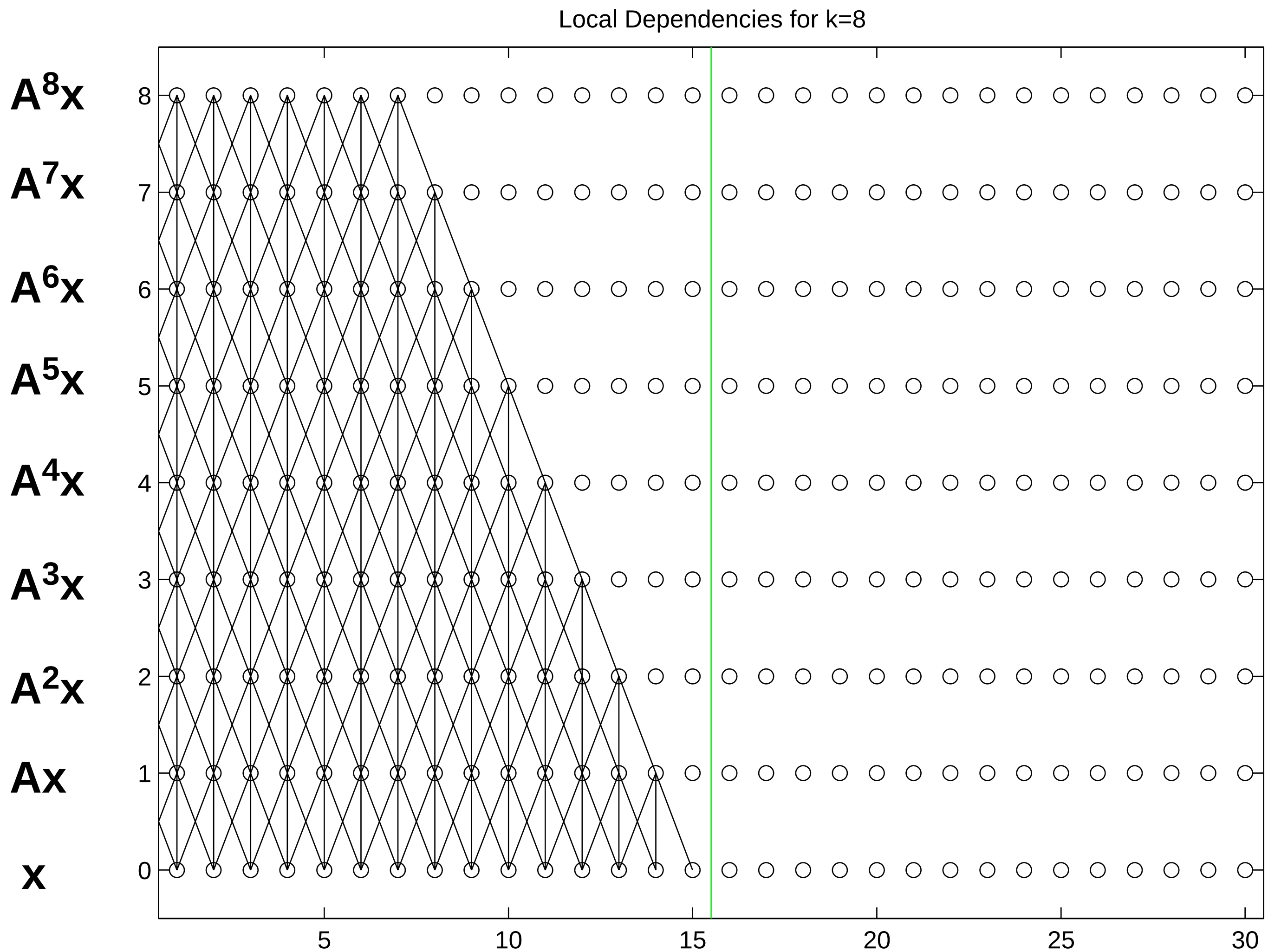
- Tuning $(x,A,k) \rightarrow [x,Ax,A^2x,\ldots A^kx]$
- Optimal communication complexity algorithms for sparse linear algebra

- Optimal communication complexity algorithms for dense linear algebra
- Optimal arithmetic complexity algorithms for dense linear algebra

- So what if they're optimal, are they fast?

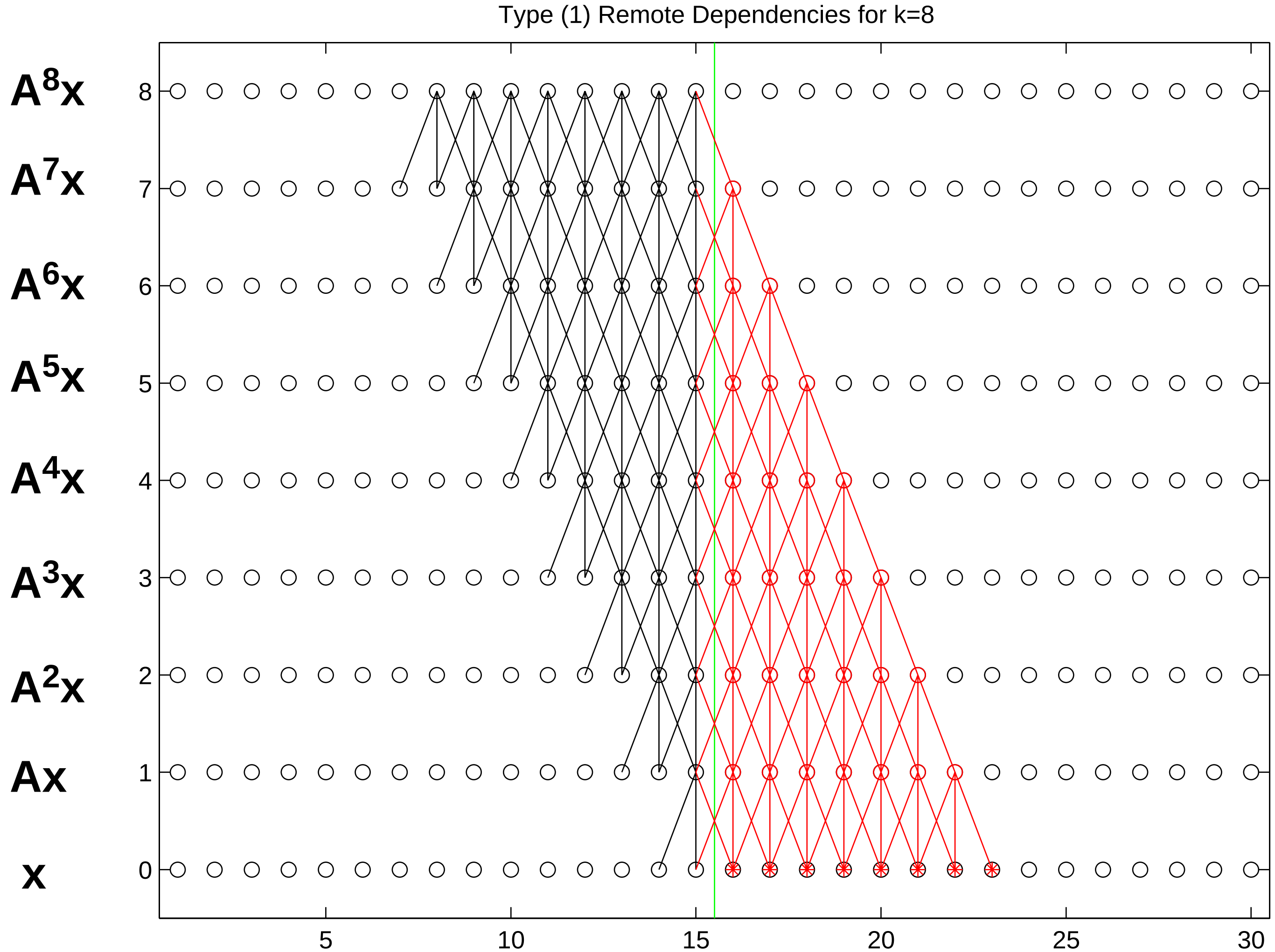
- Tuning opportunities in Sca/LAPACK

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



Can be computed without communication
k=8 fold reuse of A

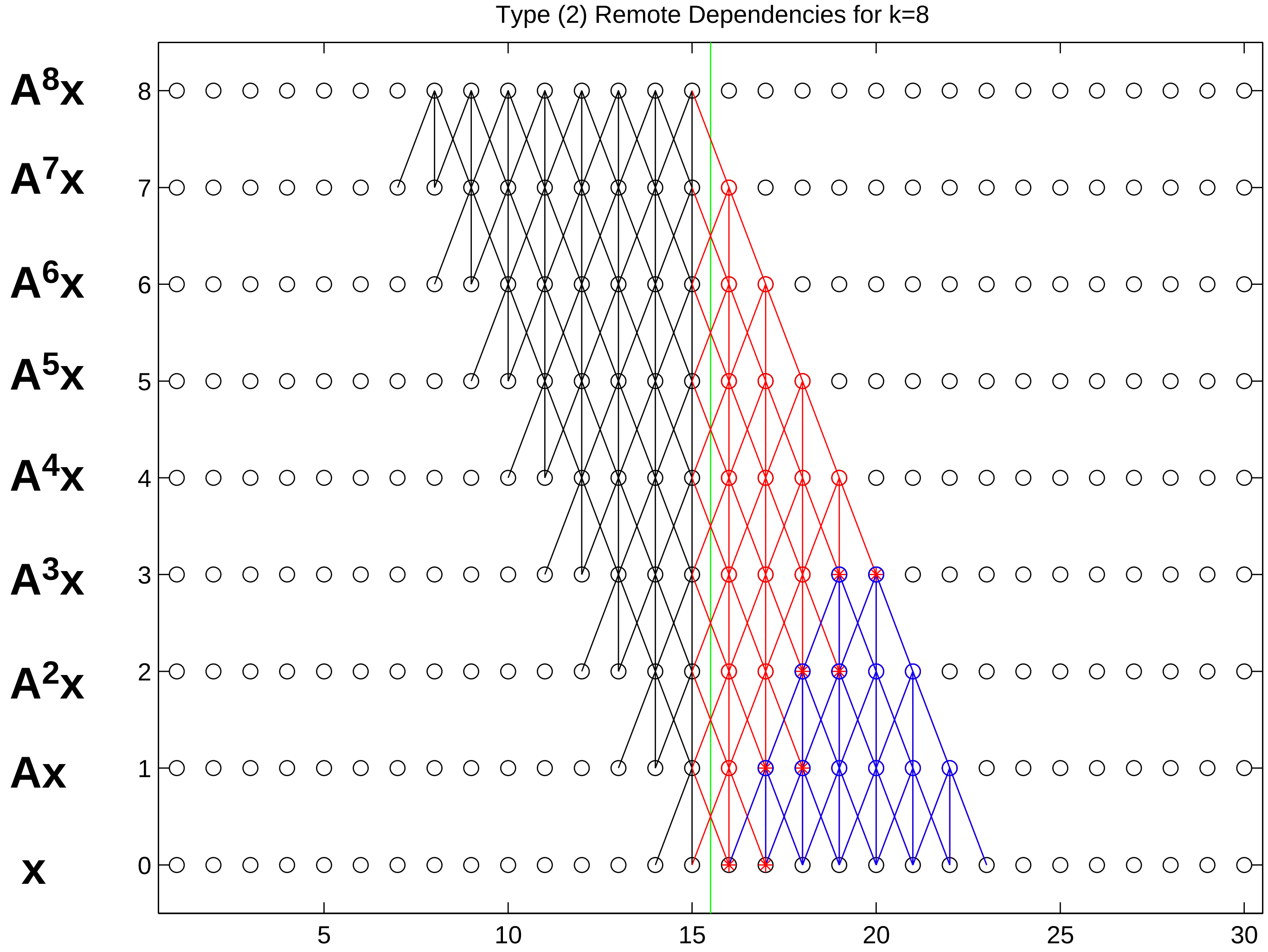
Remotely Dependent Entries for [x,Ax,…,A^8x], A tridiagonal



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

Price: redundant work

Fewer Remotely Dependent Entries for $[x,Ax,\ldots,A^8x]$, A tridiagonal

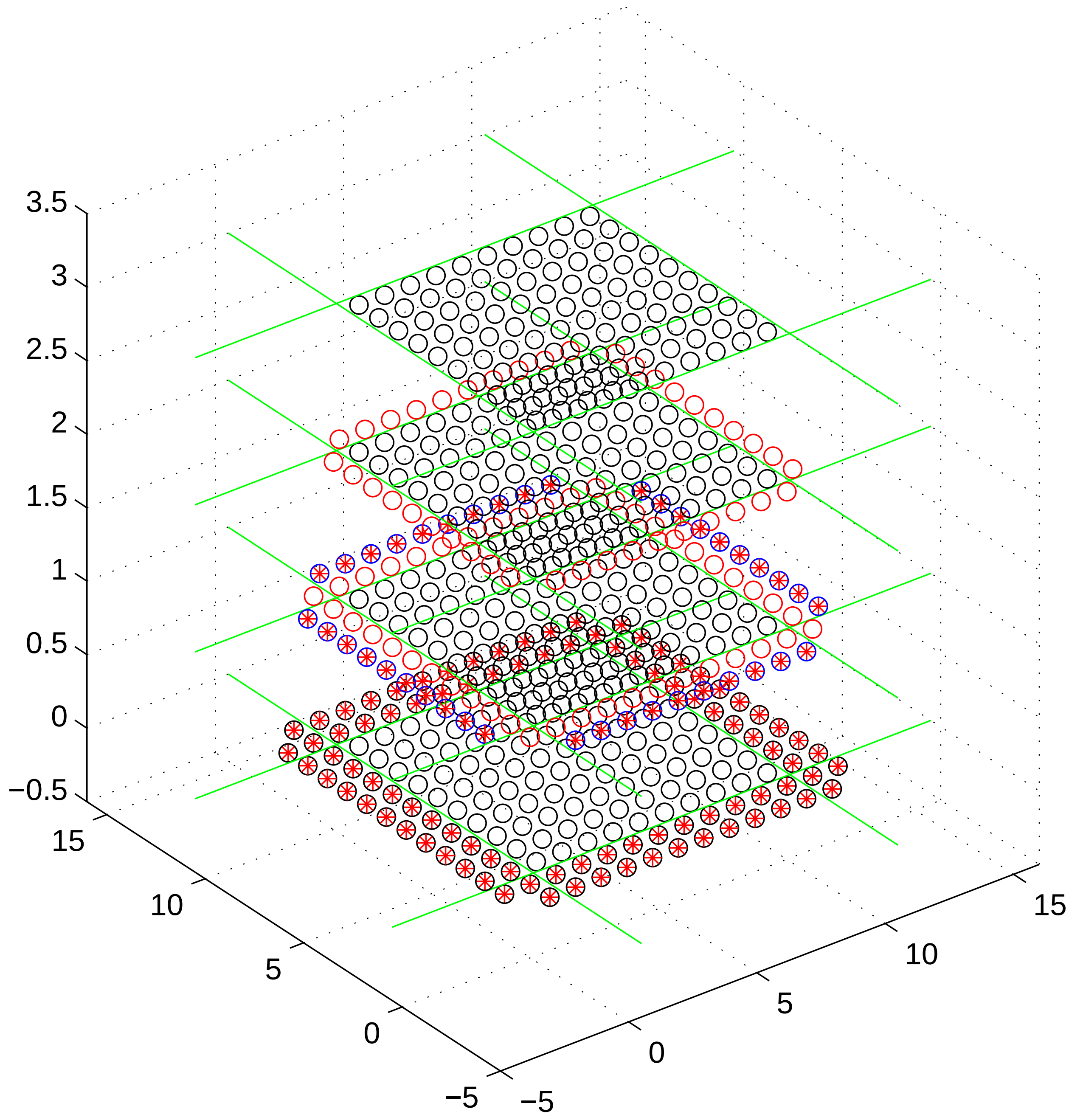


Reduce redundant work by half

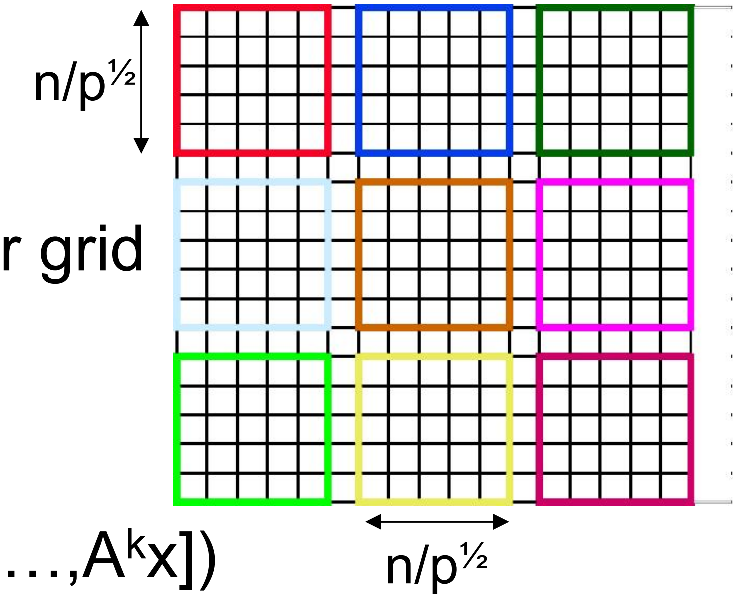
Latency Avoiding Parallel Kernel for $[x, Ax, A^2x, \ldots, A^kx]$

- Compute locally dependent entries needed by nghbrs
- Send data to nghbrs, receive from nghbrs
- Compute remaining locally dependent entries
- Wait for receive
- Compute remotely dependent entries

Remote dependencies for Approach (2) to 2D mesh with 5 pt stencil, 3D view



Parallel Complexity



- Example matrix – "2D mesh"
  - *x* lives on n-by-n mesh
  - Partitioned on $p^{1/2}$ -by- $p^{1/2}$ processor grid
  - *A* has "5 point stencil" (Laplacian)
  - Ex: 18-by-18 mesh on 3-by-3 grid
- Cost = (flops, #words, #messages)
- Cost(conventional algorithm for $[x,Ax,\ldots,A^kx]$)

  $= (9kn^2/p,\ 4kn/p^{1/2},\ 4k)$

  = (O(k·volume), O(k·surface), O(k))
- Cost(new algorithm for $[x,Ax,\ldots,A^kx]$)

  $= (9kn^2/p + 9k^2n/p^{1/2},\ 4kn/p^{1/2} + 2k^2,\ 8)$

  = (O(k·volume + k^2·surface), O(k·surface), O(1))
- Latency cost of new algorithm is O(1), optimal

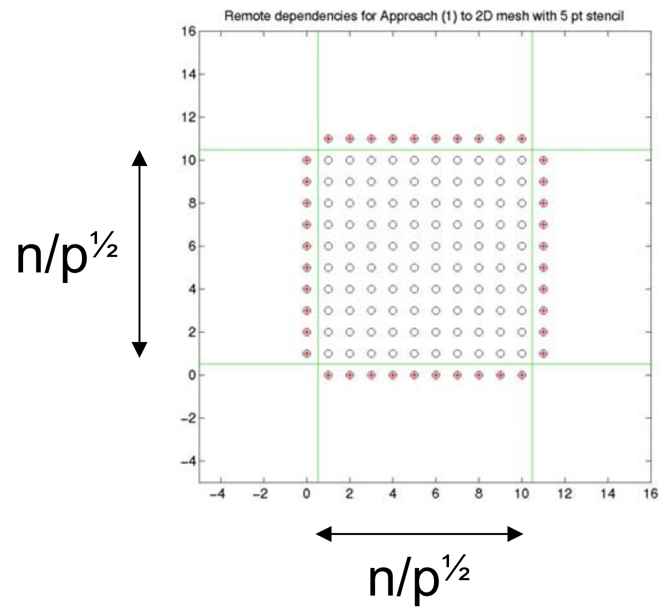
Optimal Communication Complexity Algorithms for Sparse Linear Algebra

- Consider Sparse Iterative Methods for *Ax*=*b*
  - Use Krylov Subspace Methods like GMRES, CG
  - Can we lower the communication costs?
    - Latency of communication, for a parallel machine
    - Latency and bandwidth, for a memory hierarchy
- Example: GMRES for *Ax*=*b* on “2D Mesh”

Minimizing Communication

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

GMRES, v1:
  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*



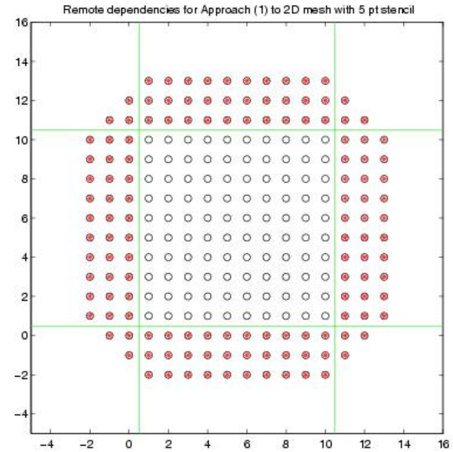
- Cost(*A* * *v*) = k * ($9n^2/p$, $4n/p^{1/2}$, 4)
- Cost(MGS) = $k^2/2$ * ($4n^2/p$, log p , log p)
- Total cost ~ Cost(*A* * *v*) + Cost (MGS)
- Can we reduce the latency?

Minimizing Communication

- Cost(GMRES, v1) = 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, v2:
$W = [\ v, Av, A^2v, \ldots, A^kv\ ]$
$[Q,R] = \text{MGS}(W)$
Build H from R, solve LSQ problem



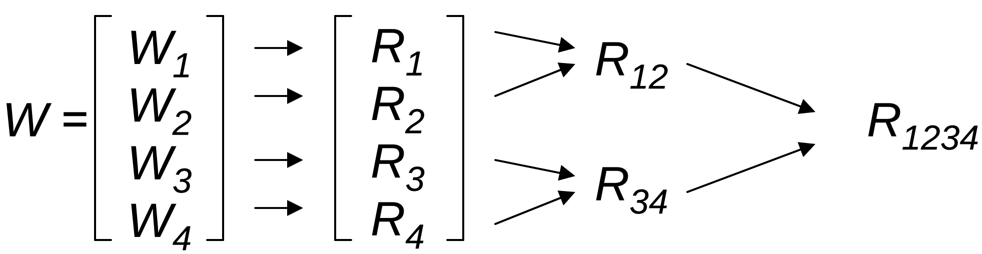
s = 3

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

Minimizing Communication

- Cost(GMRES, v2) = 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, v3:
  $W = [\ v, Av, A^2v, \ldots, A^kv\ ]$
  $[Q,R]$ = TSQR(W) … "Tall Skinny QR"
  Build H from R, solve LSQ problem



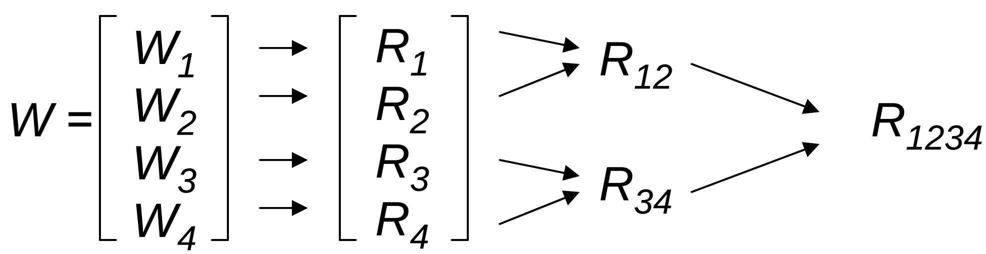
- Cost(TSQR) = (~ same, ~ same , log p)
  - Latency cost *independent* of k - ***optimal***

Minimizing Communication

- Cost(GMRES, v2) = 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, v3:
$W = [\, v, Av, A^2v, \ldots, A^kv \,]$
$[Q,R]$ = TSQR(W) … "Tall Skinny QR"
Build H from R, solve LSQ problem



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

Minimizing Communication

- Cost(GMRES, v2) = Cost(W) + Cost(MGS)

  = ($9kn^2/p$, $4kn / p^{1/2}$, 8) + ($2k^2n^2/p$, $k^2 \log p / 2$, $k^2 \log p / 2$)

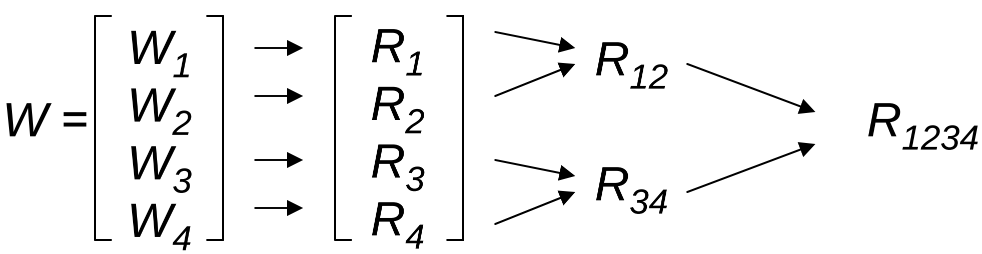
- How much latency cost from MGS can you avoid? Almost all

GMRES, v3:

$W = [\, v, Av, A^2v, \ldots , A^kv \,]$

$[Q,R]$ = TSQR(W) … "Tall Skinny QR"

Build H from R, solve LSQ problem

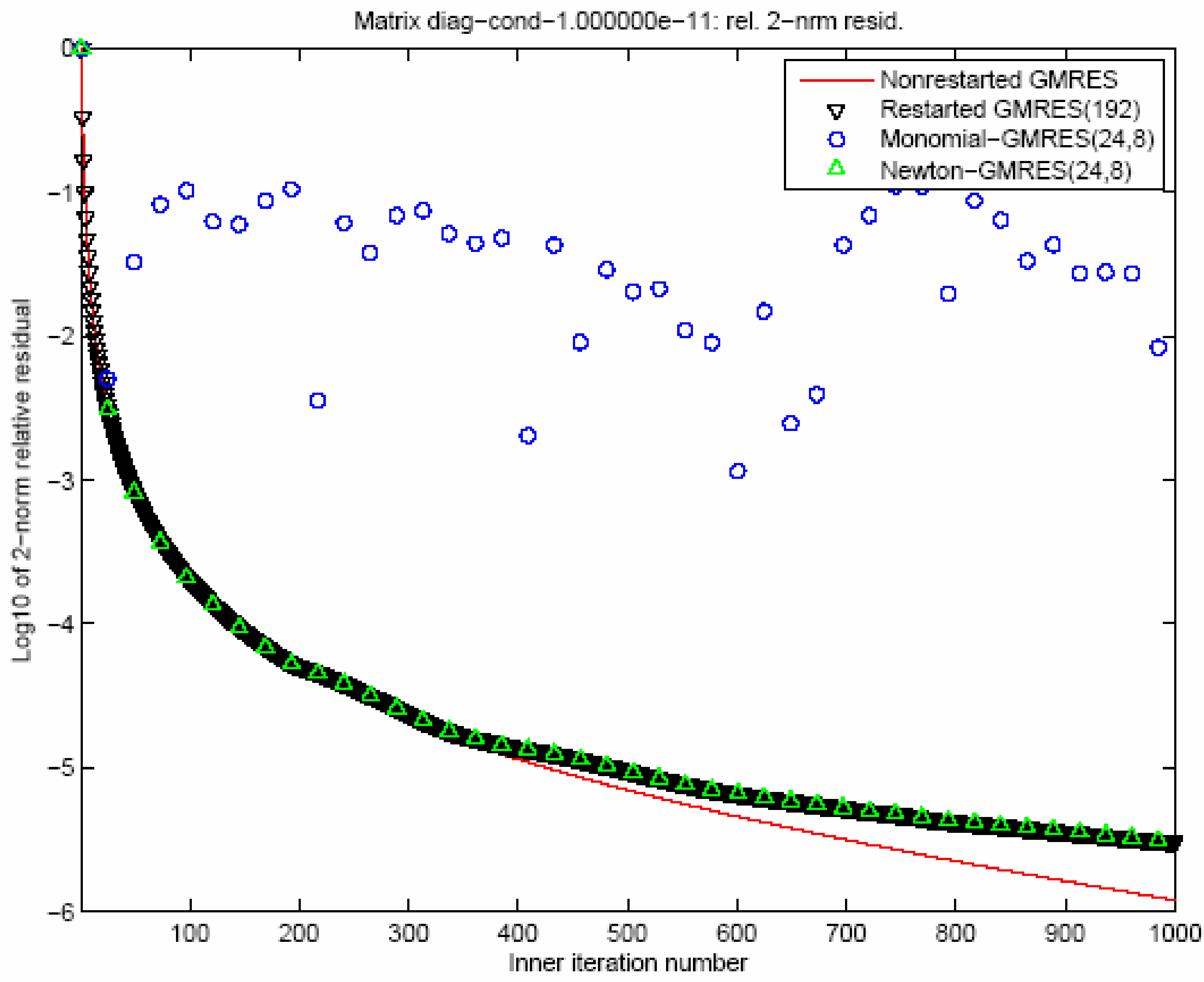


- Cost(TSQR) = (~ same, ~ same , $\log p$)
- ***Oops – W from power method, precision lost!***

Minimizing Communication

- Cost(GMRES, v3) = 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 $W = [v, Av, A^2v, \ldots]$, instead …
- Newton Basis $W_N = [v, (A - \theta_1 I)v , (A - \theta_2 I)(A - \theta_1 I)v, \ldots]$ or
- Chebyshev Basis $W_C = [v, T_1(v), T_2(v), \ldots]$

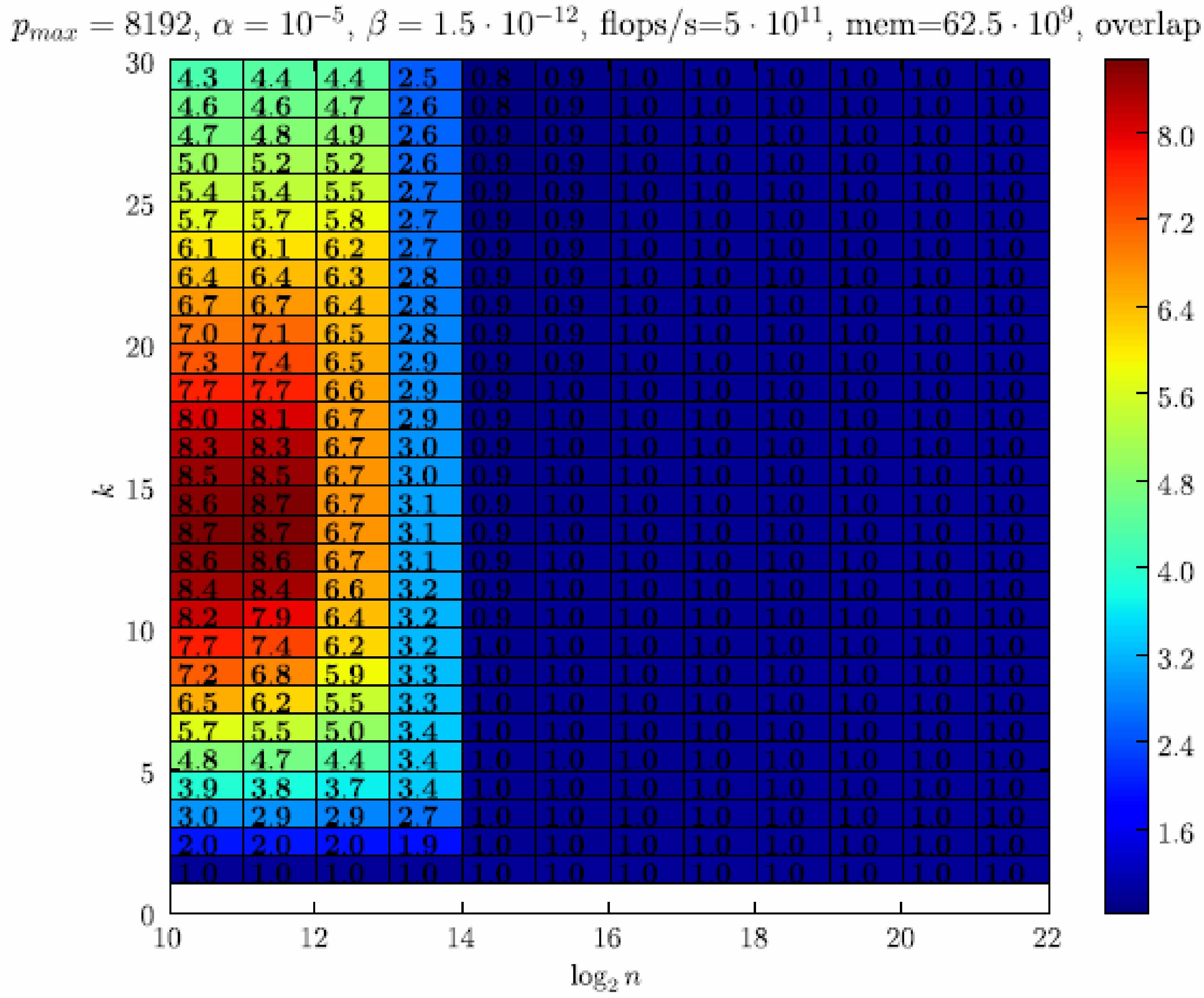


Matrix diag−cond−1.000000e−11: rel. 2−nrm resid.
Nonrestarted GMRES
Restarted GMRES(192)
Monomial−GMRES(24,8)
Newton−GMRES(24,8)
Log10 of 2−norm relative residual
Inner iteration number
0
-1
-2
-3
-4
-5
-6
100
200
300
400
500
600
700
800
900
1000

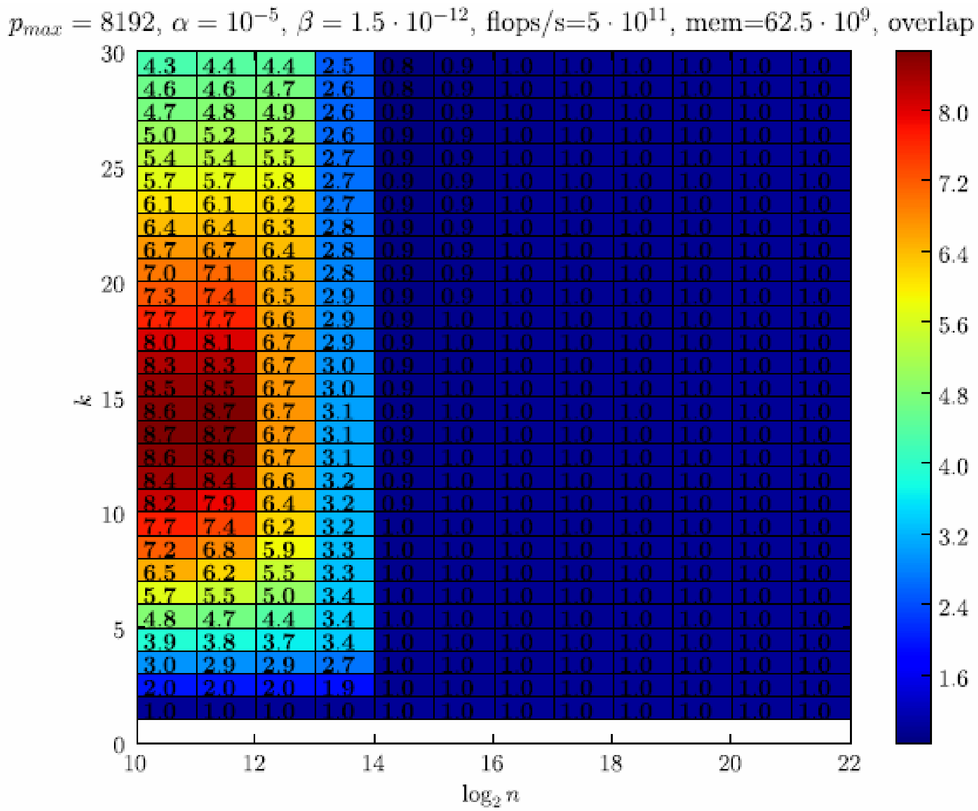
Performance Modeling

- Petascale
  - Max # processor =8100
  - Memory/processor = $6.25 \cdot 10^{9}$ words
  - Flop time = $2 \cdot 10^{-12}$ secs (.5 TFlops/s)
  - Latency = 10^{-5} secs
  - 1/Bandwidth = $1.5 \cdot 10^{-12}$ secs (.67 TWords/s)
    - Should be 4GB/s = .5 GW/s = 2 e-9 secs
- Grid
  - Max # processor = 125
  - Memory/processor = $1.2 \cdot 10^{12}$ words
  - Flop time = 10^{-13} secs (10 TFlops/s)
  - Latency = .1 secs
  - 1/Bandwidth = $3 \cdot 10^{-9}$ secs (.33 GWords/s)
    - Should be (40GB/s / 125 / 8) = 40MWords/s = 25 e-9 secs
    - Could be as high as 100 e-9 secs

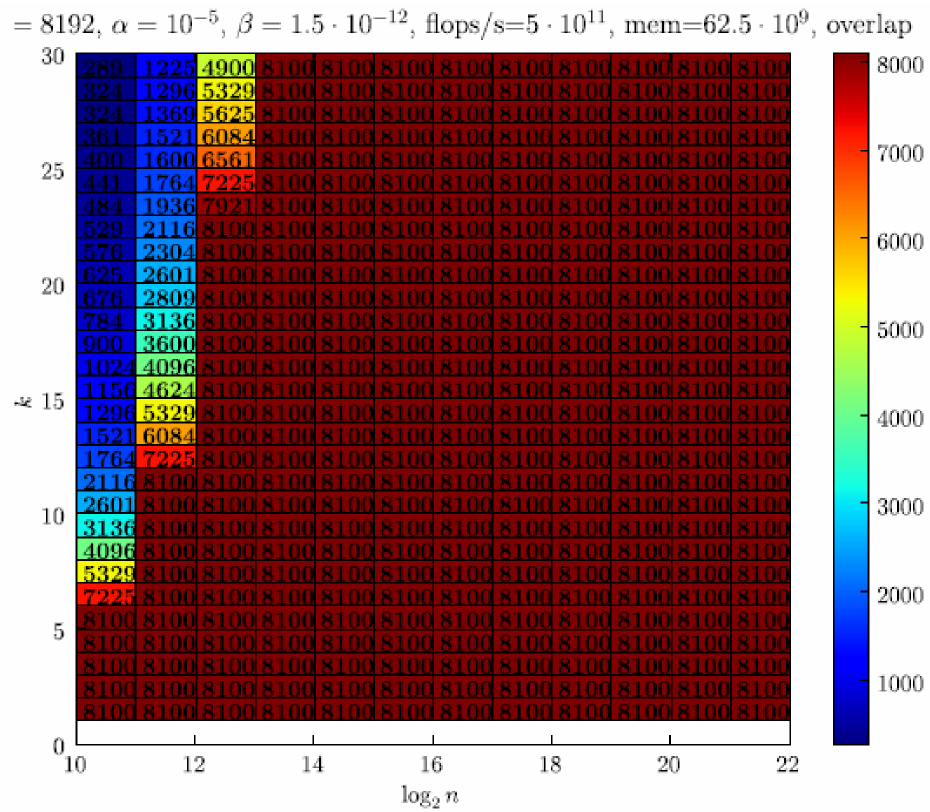
Speedup of 2D Mesh, 9pt stencil, on Petascale, with overlap



Speedup of 2D Mesh, 9pt stencil, on Petascale, with overlap

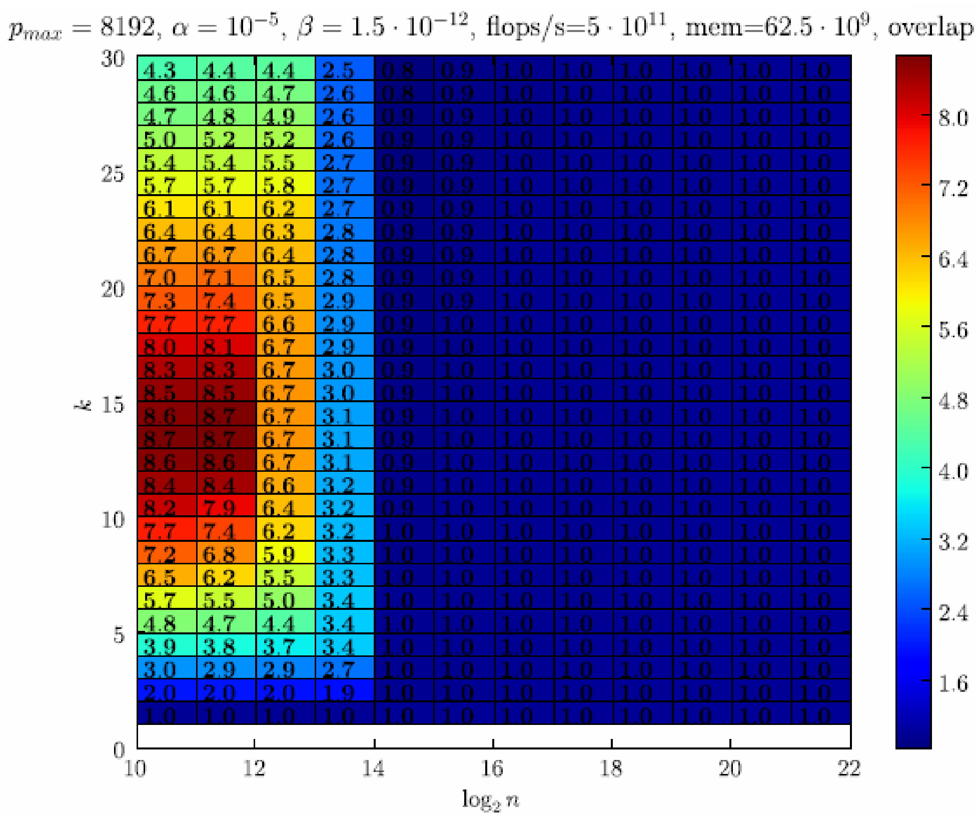


Speedup

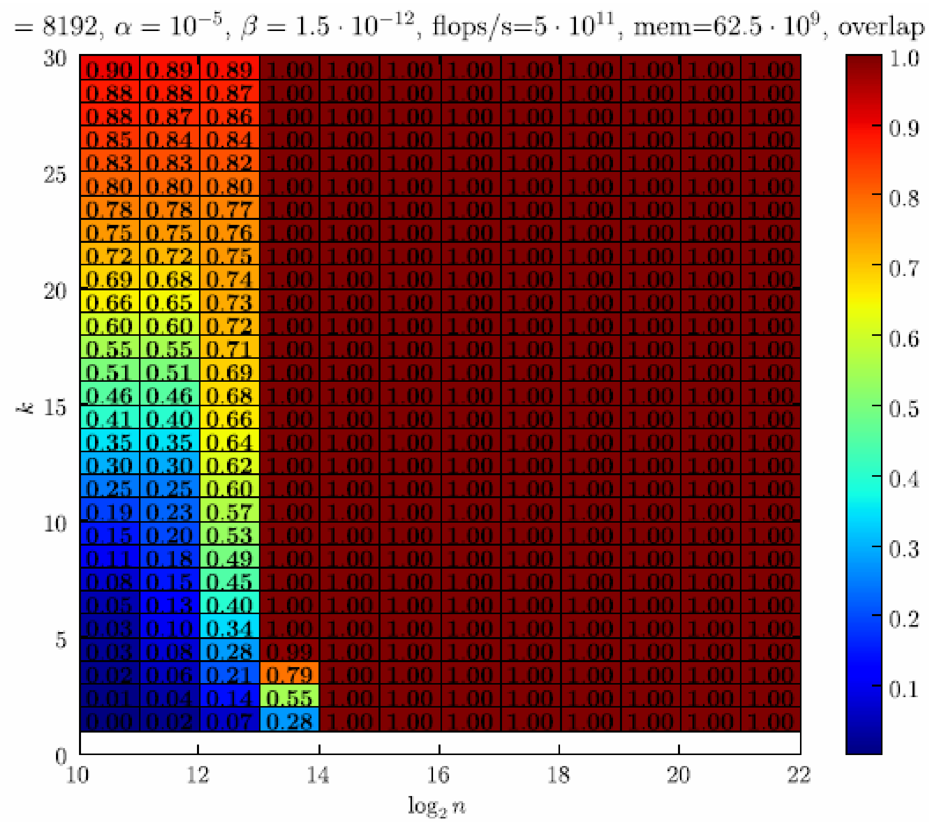


Optimal p (≤8100)

Speedup of 2D Mesh, 9pt stencil, on Petascale, with overlap

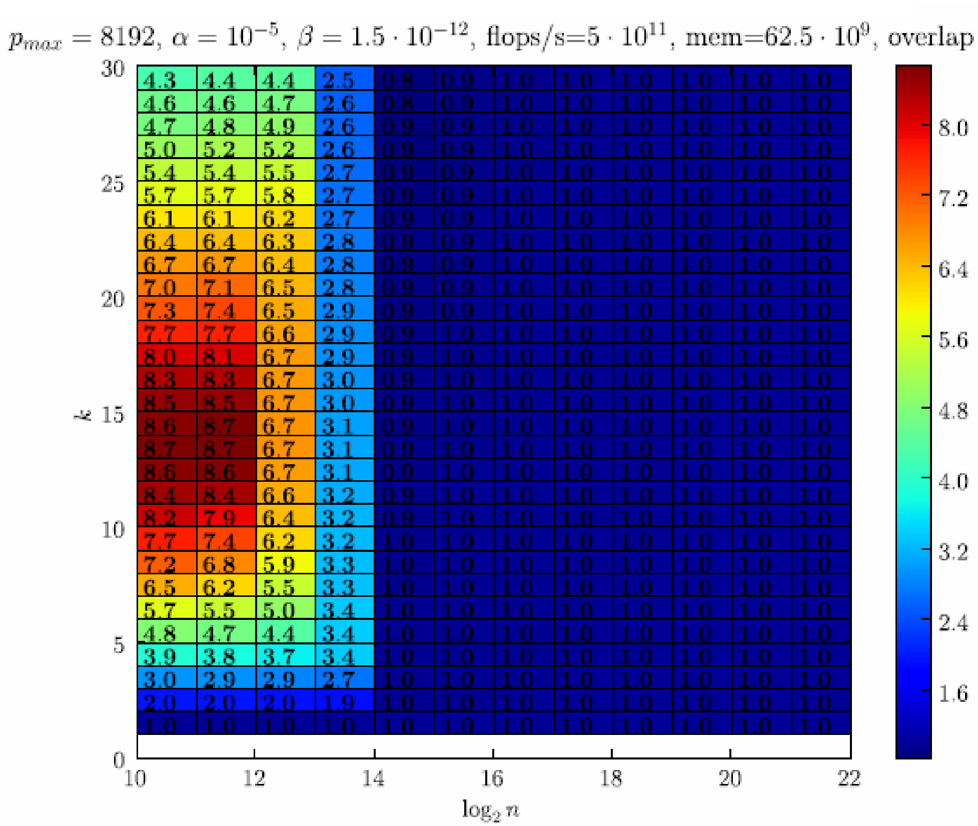


Speedup

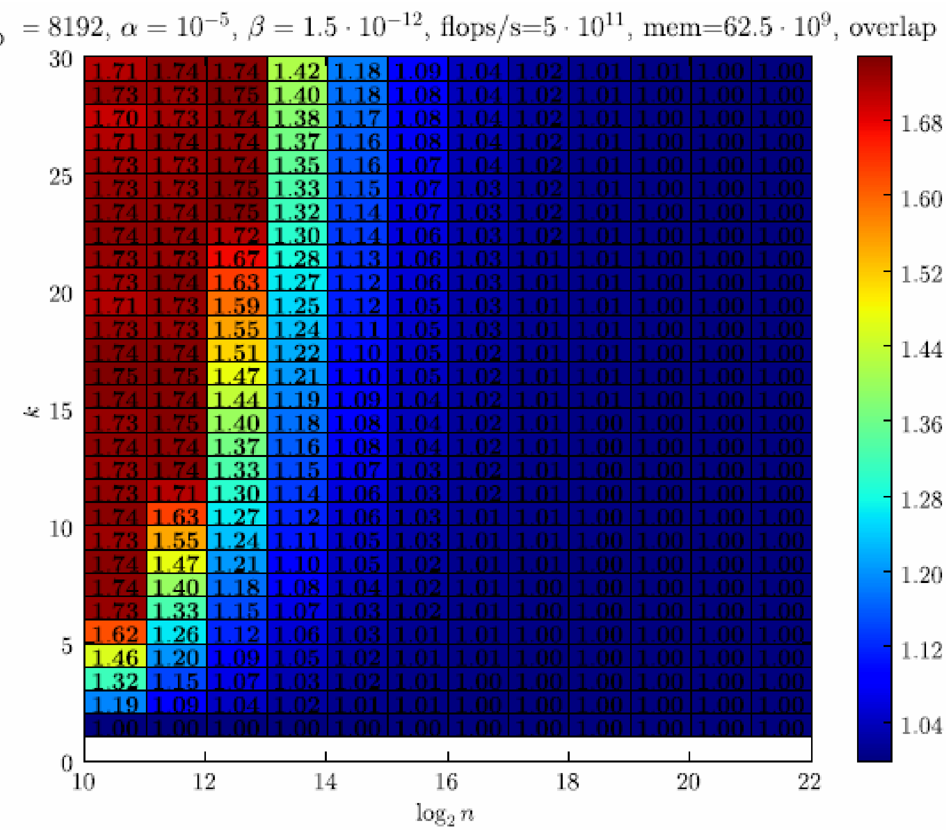


Time(flops) / Total Time

Speedup of 2D Mesh, 9pt stencil, on Petascale, with overlap

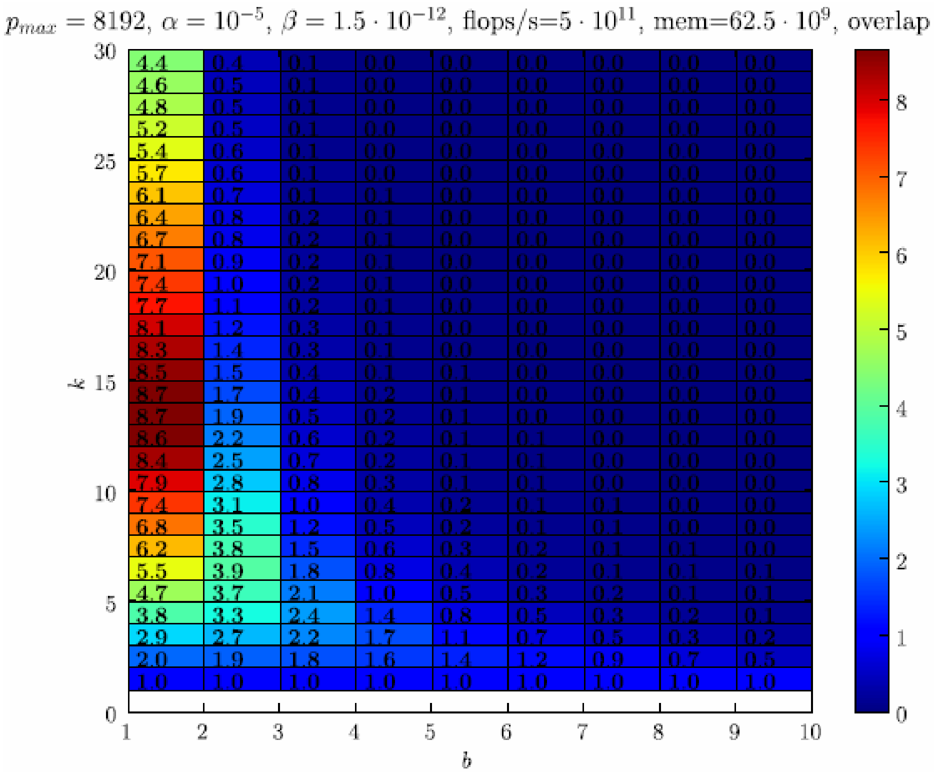


Speedup

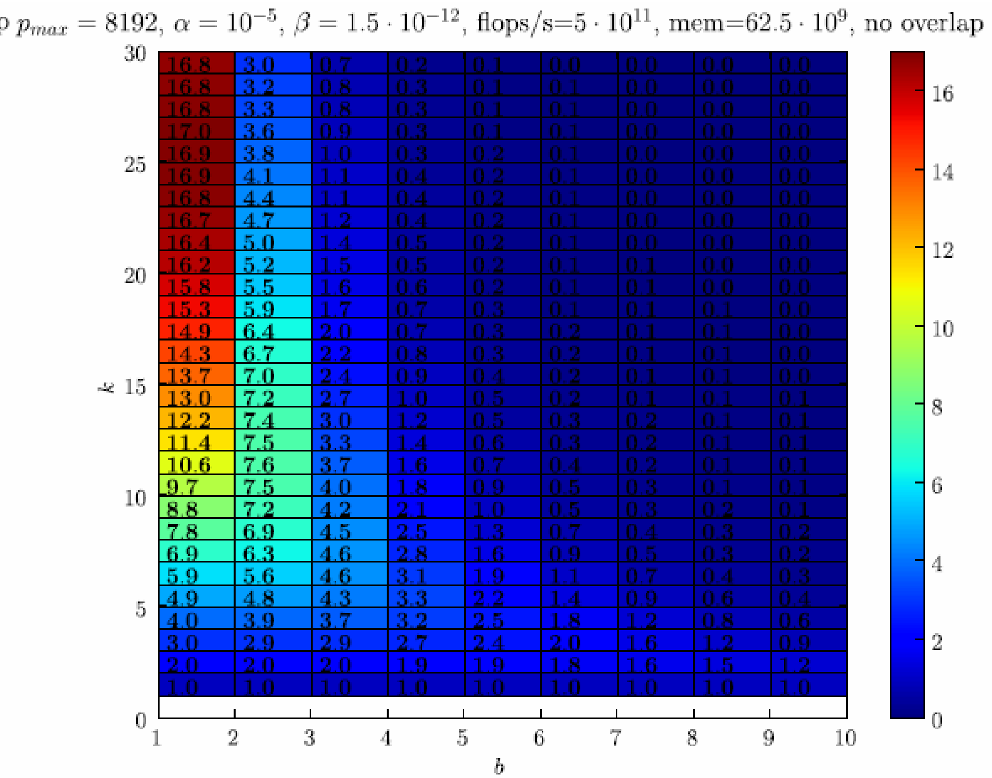


#flops in new alg / #flops in old alg

Speedup of 2D mesh, $(2b+1)^2$ pt stencil, on Petascale



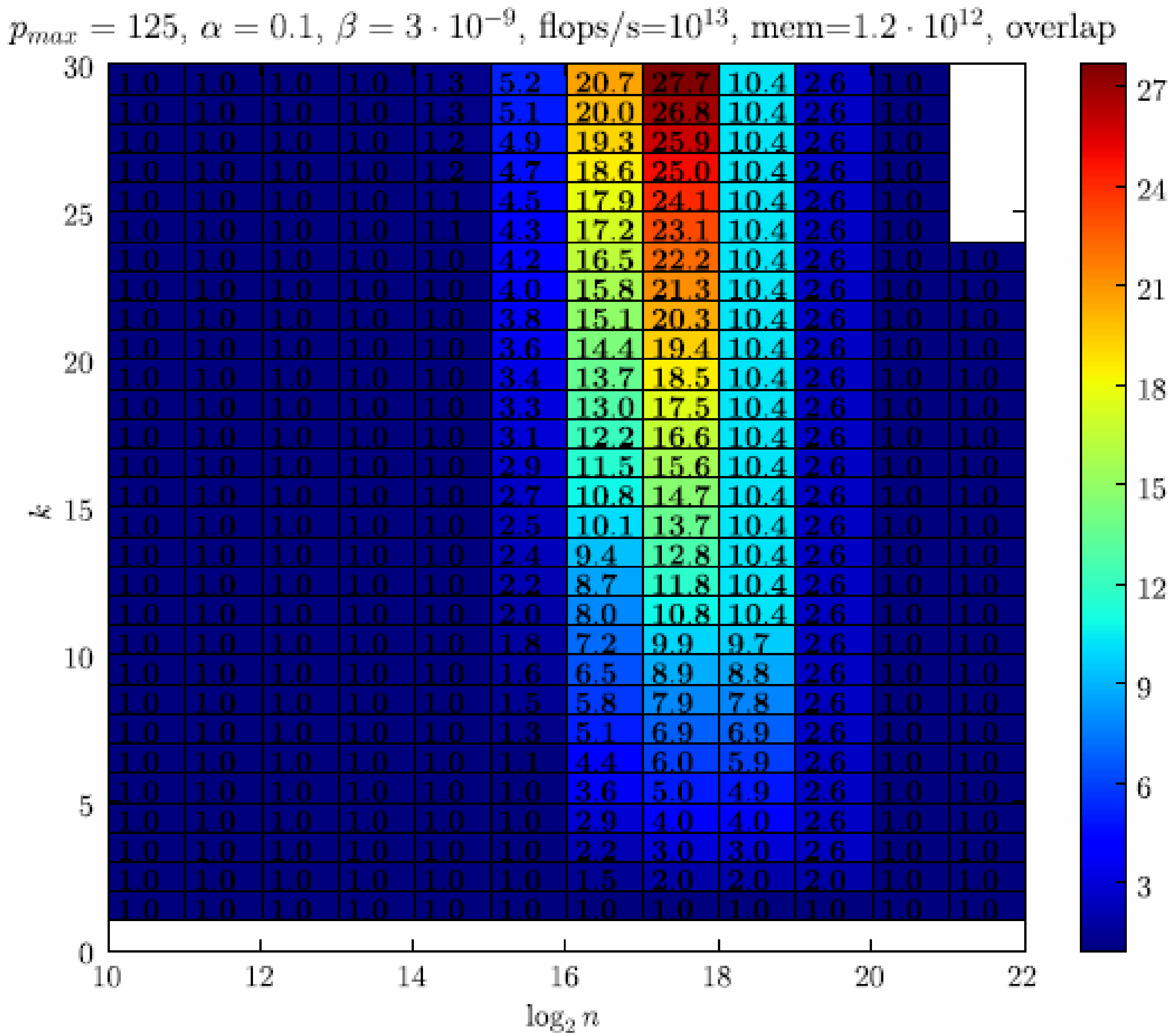
With overlap



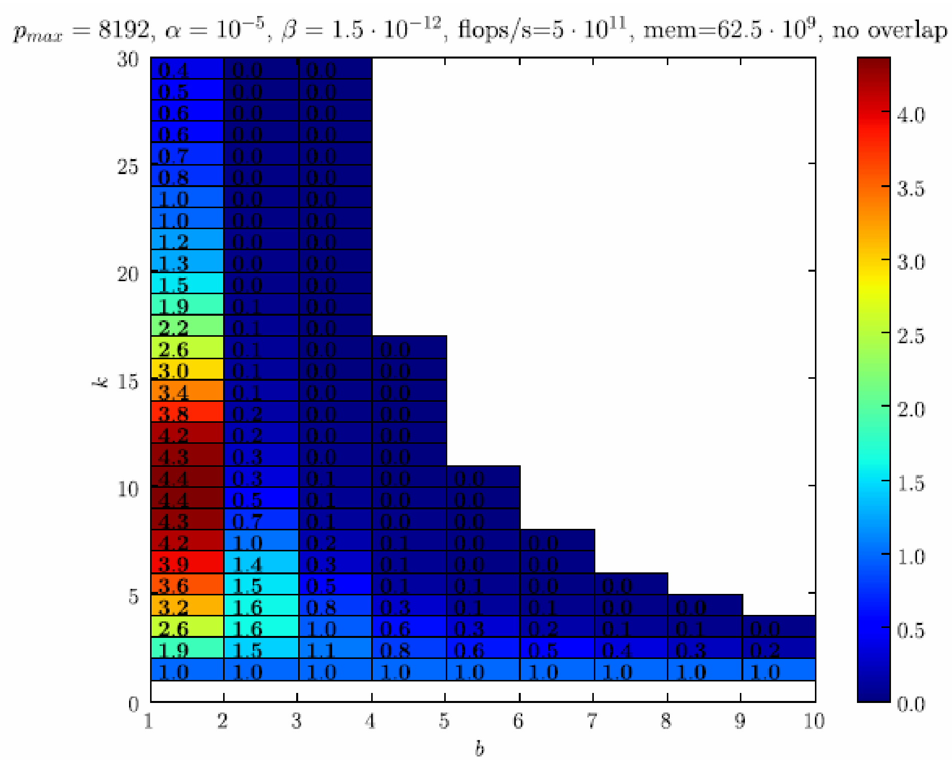
Without overlap

$n = 2^{11}$

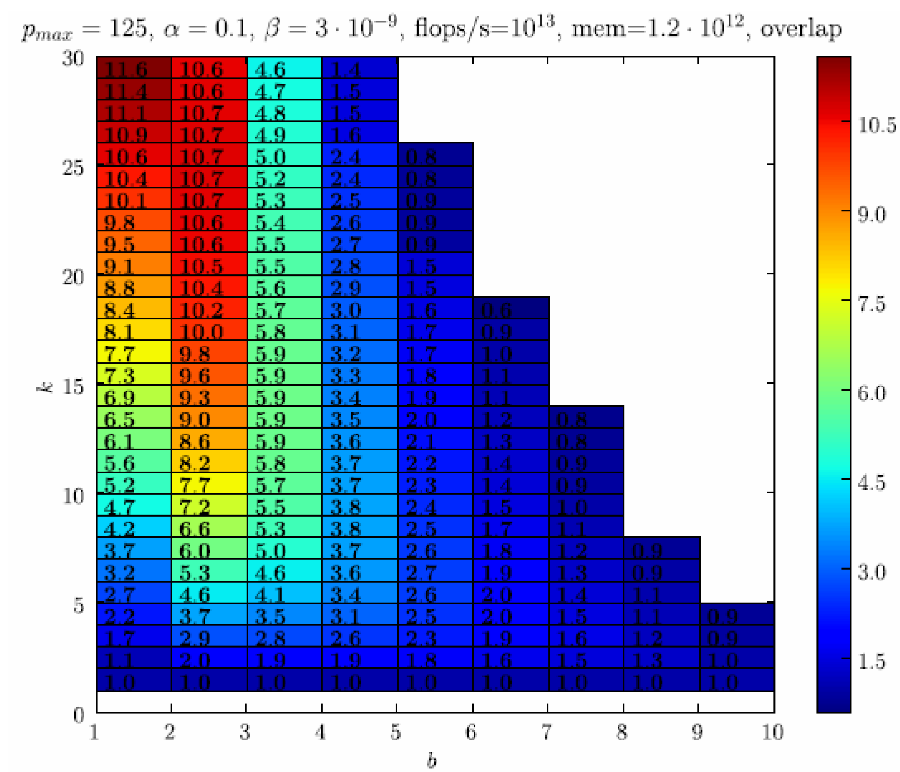
Speedup of 2D Mesh, 9pt stencil, on Grid, with overlap



Speedup on 3D mesh, $(2b+1)^3$ pt stencil



Petascale, without overlap
n=512
(no speedup with overlap!)



Grid, with overlap
n=1024

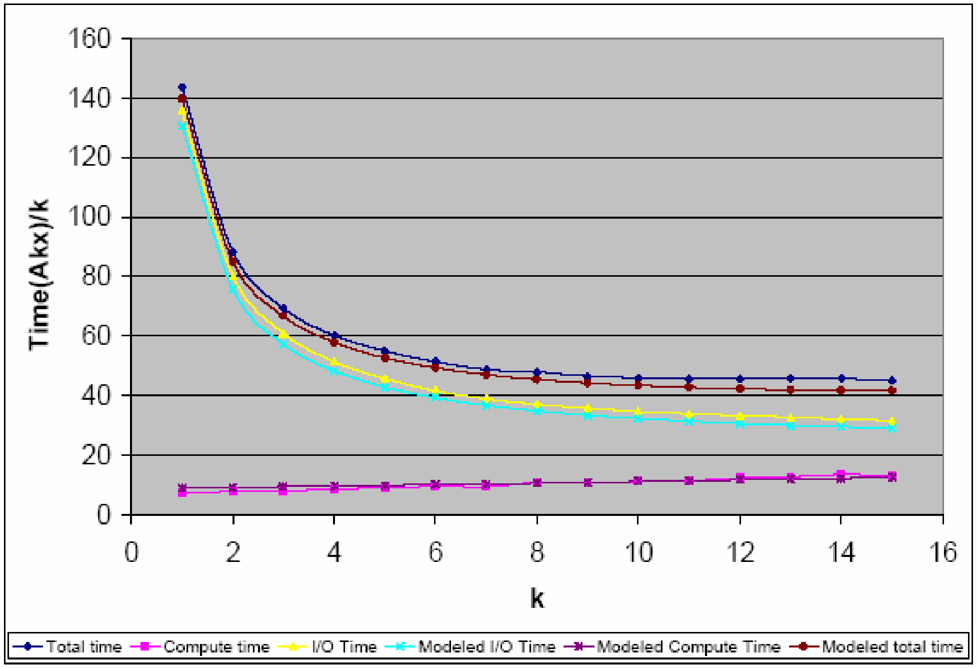
Latency *and Bandwidth* Avoiding Sequential Kernel for [x, Ax, … , A^kx]

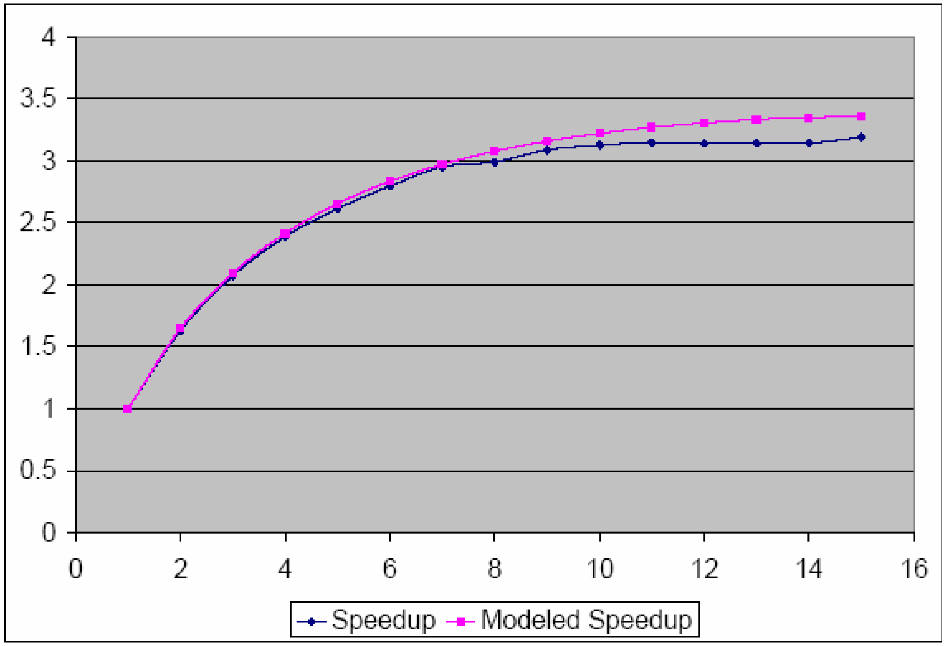
- Mimic parallel algorithm:
  - For i = 1 to #blocks of x
    - Load rows of A needed to compute block i of [Ax,…,A^kx] (including remotely dependent entries)
    - Load block i of x and parts of x from neighboring blocks needed to compute remotely dependent entries of [Ax,…,A^kx]
    - Compute block i of [Ax,…,A^kx]
- #Blocks chosen to fit as much of A and [x,Ax,…,A^kx] in fast memory as possible
  - Double buffering, other optimizations possible
- Optimal in sense that all data moved between fast and slow memory ≈once
  - 1 + (k·surface/volume) times
  - Increase computational intensity k-fold

Measured and Modeled Performance

5.2 GFlop Itanium2, 4GB memory, Disk

1/f= 300MFlops/s, BW_{read} = 140 MB/s, BW_{write} = 30 MB/s,   disk latency irrelevant





3D mesh, 27-pt stencil, n = 368, p = 64 blocks,

Measured Speedup up to 3.2x (flop time ≈ ½ bandwidth time)

Summary of Optimal Sparse Algorithms

- Tuning and algorithmic design interact
- Can eliminate latency from GMRES, CG, … maintaining stability
  - Ideas go back to Van Rosendale (1983), Chronopoulos & Gear (1989), many others, but without simultaneous stability & optimality
- Extends to preconditioned methods
  - Kernel becomes $[x, Ax, MAx, AMAx, MAMAx, \ldots, (MA)^k x]$
  - But only some preconditioners let us eliminate latency, not raise flop count a lot (work in progress)
- Lots of tuning opportunities
  - All SpMV techniques, plus choosing k, polynomial in kernel, partitioning, overlapping communication and computation, …

Minimizing Communication in Dense Linear Algebra

- Communication costs of current ScaLAPACK
  - LU & QR: O(n log p) messages
  - Cholesky: O(n/b log p) messages
- New "LU" and "QR" algorithms
  - As few messages as Cholesky
  - "QR" returns QR but represented differently
  - "LU" "equivalent" to LU in complexity, stability TBD
- "Optimal" communication complexity, but fast?

Minimizing Arithmetic in Dense Linear Algebra

- Long known (Strassen) how to invert matrices as fast as matmul, but unstably:

$$\begin{bmatrix} T_{11} & T_{12} \\ & T_{22} \end{bmatrix}^{-1} = \begin{bmatrix} T_{11}^{-1} & -T_{11}^{-1} T_{12} T_{22}^{-1} \\ & T_{22}^{-1} \end{bmatrix}$$

- New results
  - Can make solving Ax=b, least squares, eigenvalue problems as fast as fastest matmul, and stable (even if matmul unstable!)
- “Optimal” arithmetic complexity, but fast?

What *could* go into a dense linear algebra library?

For all linear algebra problems

For all matrix structures

For all data types

For all architectures and networks

For all programming interfaces

Produce best algorithm(s) w.r.t. performance and accuracy (including condition estimates, etc)

Need to prioritize, automate!

How do we best explore this large tuning space?

- **Algorithm tuning space includes**
  - **Numerous block sizes, not just in underlying BLAS**
  - **Many possible layers of parallelism, many mappings to HW**
  - **Different traversals of underlying DAGs**
    - **Left and right looking two of many; asynchronous algorithms**
  - **“Redundant” algorithms for GPUs**
  - **Recursive, parallel layouts and algorithms**
  - **New “optimal” algorithms for variations on standard factorizations**
  - **New and old eigenvalue algorithms**
  - **Mixed precision (for speed or accuracy)**

- **Is there a concise set of abstractions to describe, generate tuning space?**
  - **Block matrices, factorizations (partial, tree, …), DAGs, …**
  - **FLAME, CSS, Spiral, Sequoia, Telescoping languages, Bernoulli, Rose, …**

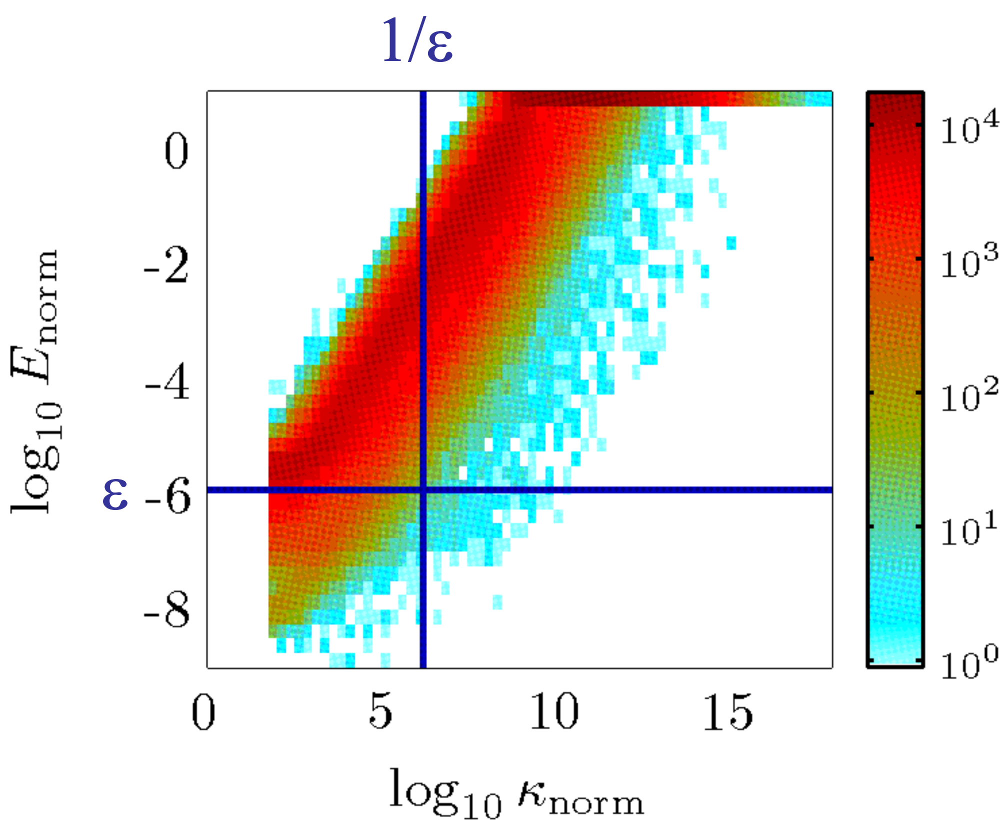
- **Question: What fraction of dense linear algebra can be generated/tuned?**
  - **Lots more than when we started**
    - **Sequential BLAS -> Parallel BLAS -> LU -> other factorizations -> …**
  - **Most of dense linear algebra?**
    - **Not eigenvalue algorithms (on compact forms)**
    - **What fraction of LAPACK can be done?**
    - **Rest of loop “for all linear algebra problems…”**
  - **For all interesting architectures…?**

Exploiting GPUs

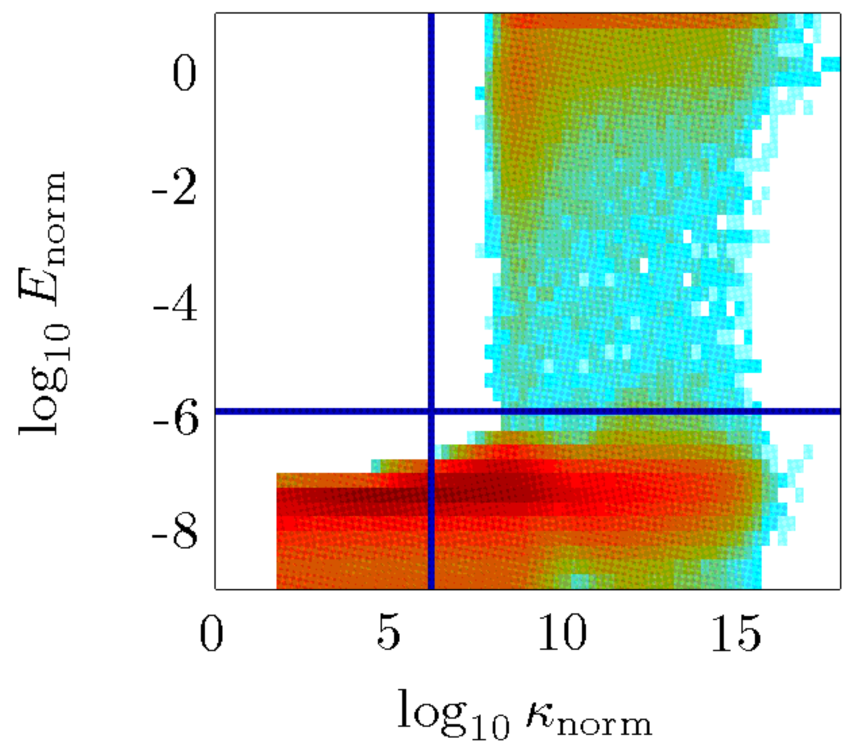
- Numerous emerging co-processors
  - Cell, SSE, Grape, GPU, “physics coprocessor,” …
- When can we exploit them?
  - LIttle help if memory is bottleneck
  - Various attempts to use GPUs for dense linear algebra
- Bisection on GPUs for symmetric tridiagonal eigenproblem
  - Evaluate Count(x) = #(evals < x) for many x
  - Very little memory traffic, but much redundant work
  - Speedups up to 100x   (Volkov)
    - 43 Gflops on ATI Radeon X1900 vs running on 2.8 GHz Pentium 4
    - Overall eigenvalue solver 6.8x faster
- Port of CLAPACK to NVIDIA underway

Iterative Refinement: For Accuracy

Conventional Gaussian Elimination



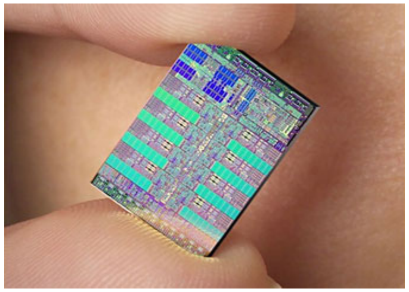
With extra precise
iterative refinement



$\varepsilon = \mathbf{n}^{1/2}\, 2^{-24}$

Iterative Refinement: For Speed

- **What if double precision much slower than single?**
  - **Cell processor in Playstation 3**
    - **256 GFlops single, 25 GFlops double**



  - **Pentium SSE2: single twice as fast as double**
- **Given Ax=b in double precision**
  - **Factor in single, do refinement in double**
  - **If $\kappa(A) < 1/\varepsilon_{single}$, runs at speed of single**
- **8x speedup on Cell, 1.9x on Intel-based laptop**
- **Applies to many algorithms, if difference large**

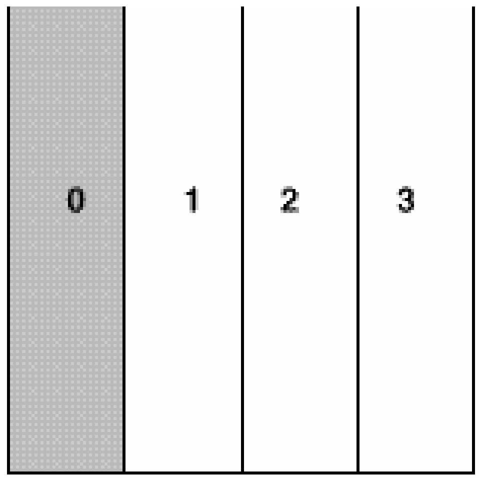
New algorithm for roots(p)

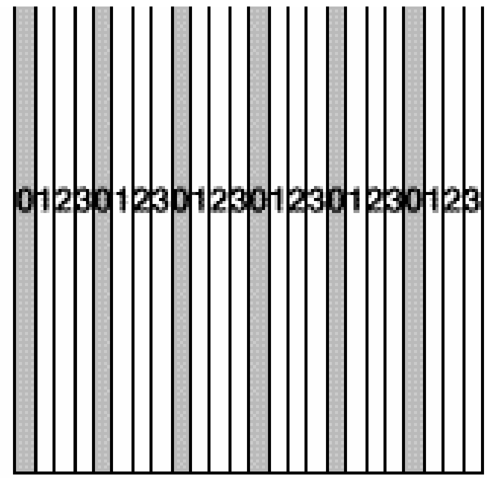
$$C(p)=\begin{pmatrix} -p_1 & -p_2 & \dots & -p_d \\ 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 1 & 0 \end{pmatrix}$$

- **To find roots of polynomial p**
  - **Roots(p) calls eig(C(p))**
  - **Costs O(n^3), stable, reliable**
- **O(n^2) Alternatives**
  - **Newton, Jenkins-Traub, Laguerre, …**
  - **Stable? Reliable?**
- **New: Exploit "semiseparable" structure of C(p)**
  - **Low rank of any submatrix of upper triangle of C(p) preserved under QR iteration**
  - **Complexity drops from O(n^3) to O(n^2), stable in practice**
- **Related work: Gemignani, Bini, Pan, et al**
- **Ming Gu, Shiv Chandrasekaran, Jiang Zhu, Jianlin Xia, David Bindel, David Garmire, Jim Demmel**

ScaLAPACK Data Layouts

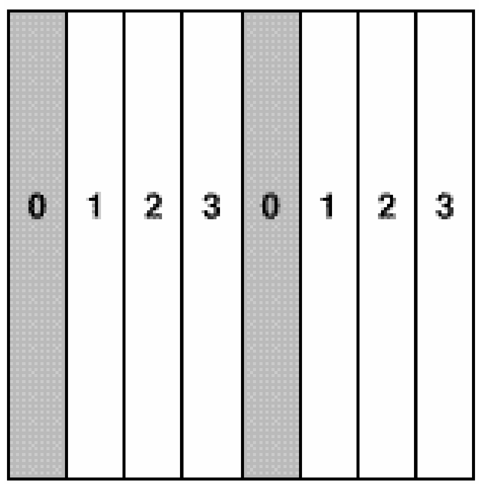
1D Block

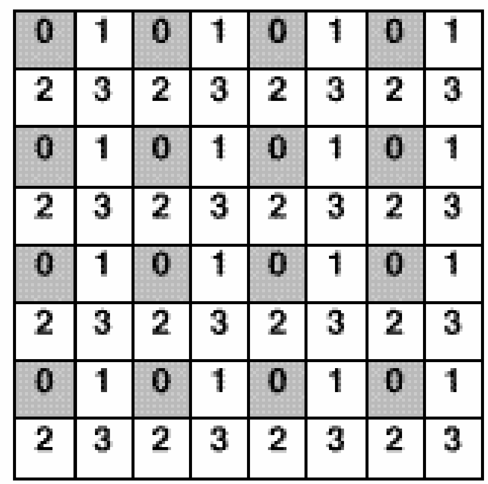




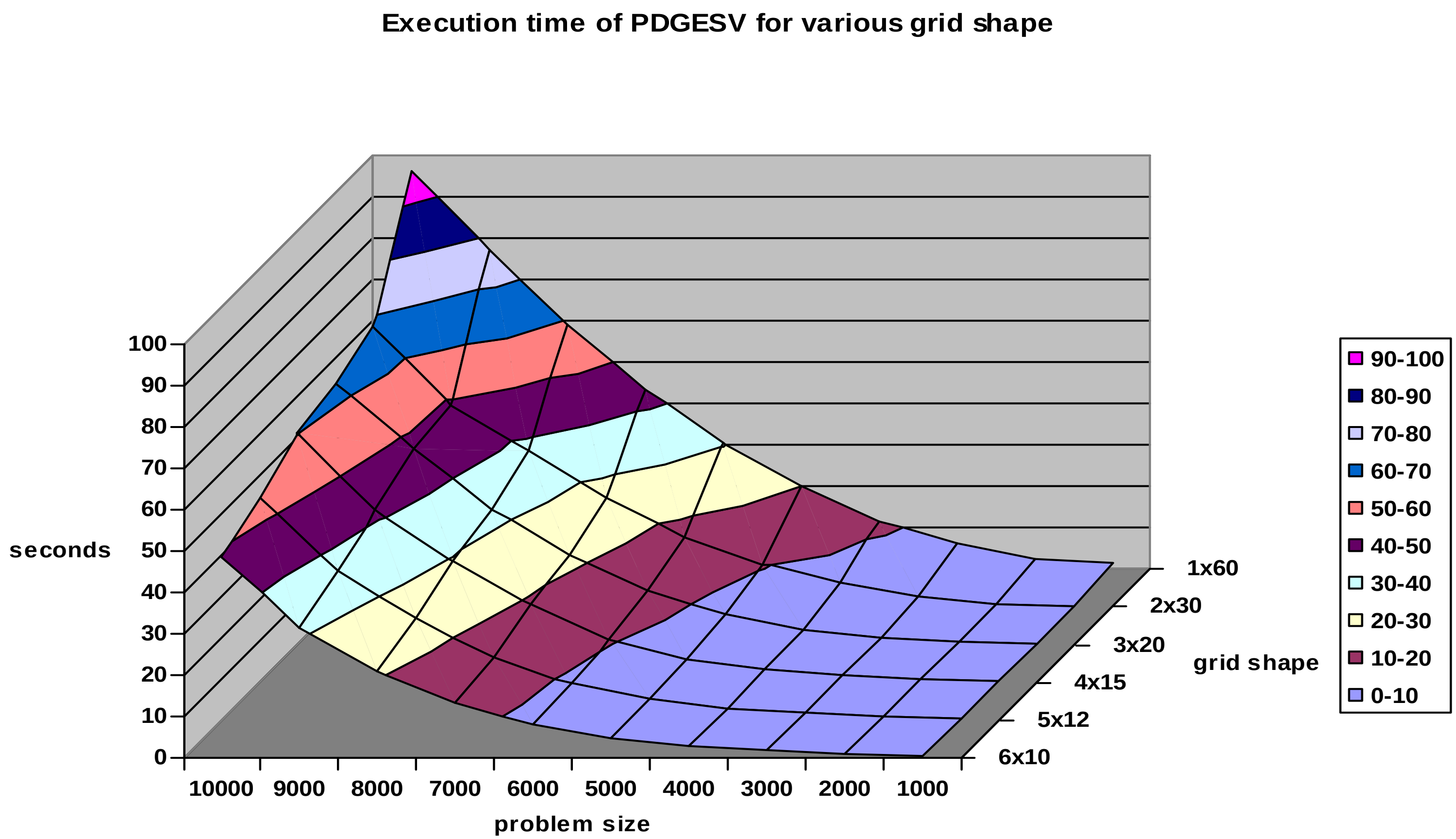
1D Cyclic

1D Block Cyclic





2D Block Cyclic



Speedups for using 2D processor grid range from 2x to 8x
Cost of redistributing from 1D to best 2D layout 1% - 10%

Times obtained on:

60 processors, Dual AMD Opteron 1.4GHz Cluster w/Myrinet Interconnect

2GB Memory

Extra Slides

New optimal algorithms (1)

- Long known (Strassen) how to invert matrices as fast as matmul, but unstably:

$$\begin{bmatrix} T_{11} & T_{12} \\ & T_{22} \end{bmatrix}^{-1} = \begin{bmatrix} T_{11}^{-1} & -T_{11}^{-1} T_{12} T_{22}^{-1} \\ & T_{22}^{-1} \end{bmatrix}$$

- New results
  - Can make solving Ax=b, least squares, eigenvalue problems as fast as fastest matmul, and stable
- "Optimal" arithmetic complexity, but fast?

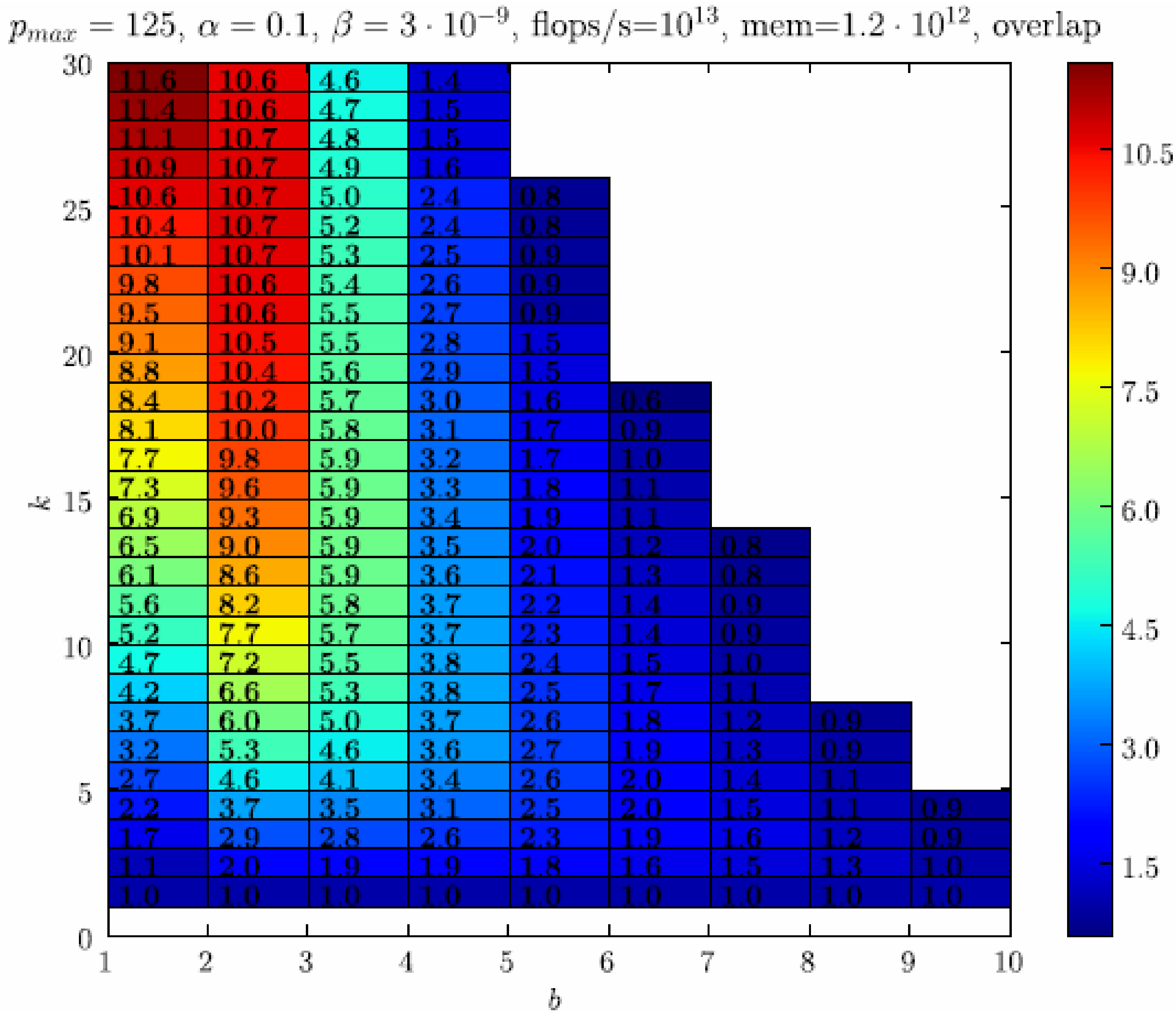
New optimal algorithms (2)

- Communication costs of current ScaLAPACK
  - LU & QR: O(n log p) messages
  - Cholesky: O(n/b log p) messages
- New “LU” and “QR” algorithms
  - As few messages as Cholesky
  - “QR” returns QR but represented differently
  - “LU” “equivalent” to LU in complexity, stability TBD
- “Optimal” communication complexity, but fast?

Goal 2 – Automate Performance Tuning

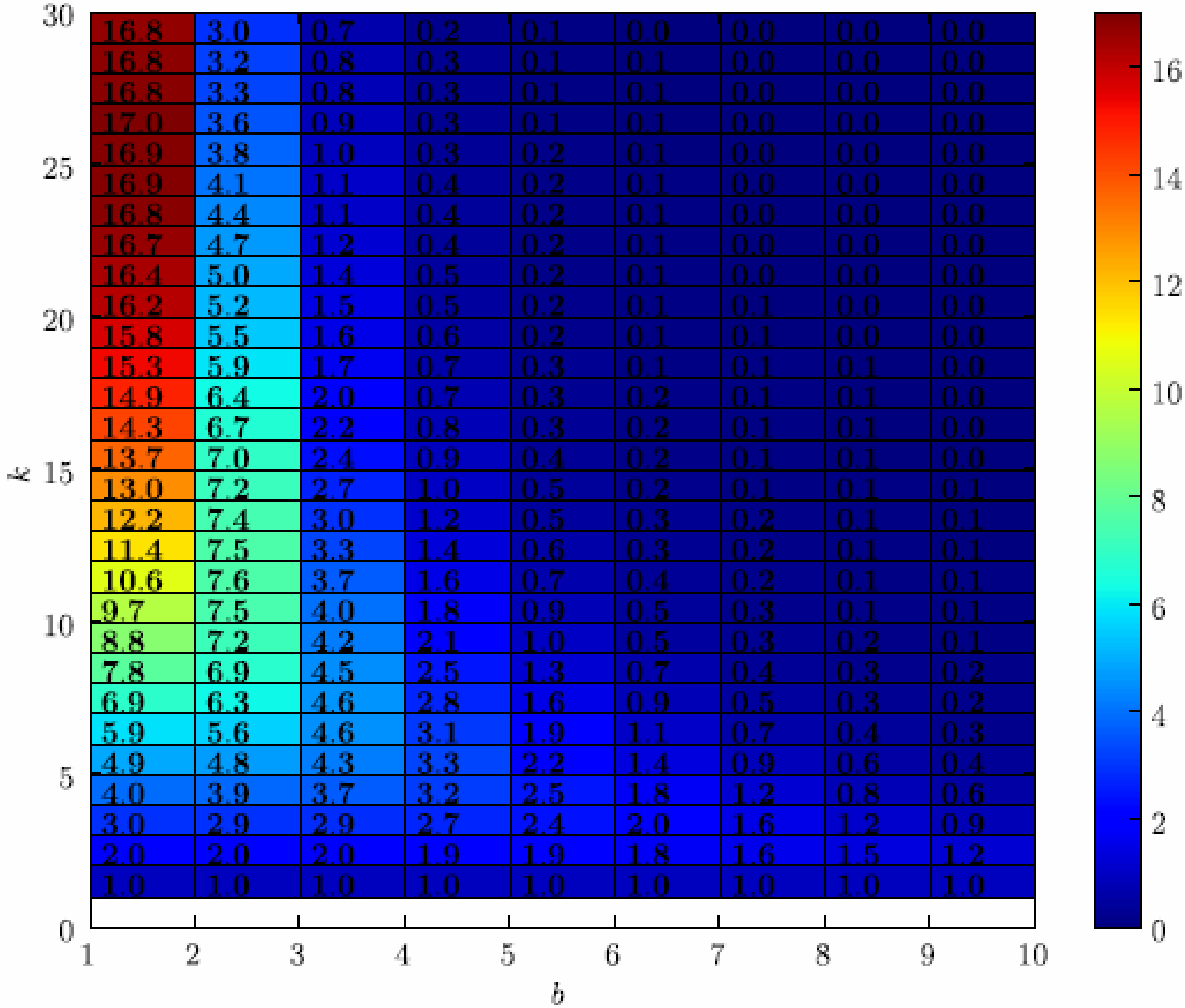
- 1300 calls to ILAENV() to get block sizes, etc.
  - Never been systematically tuned
- Extend automatic tuning techniques of ATLAS, etc. to these other parameters
  - Automation important as architectures evolve
- Convert ScaLAPACK data layouts on the fly
  - Important for ease-of-use too

Grid_wo_3d_bw_perf

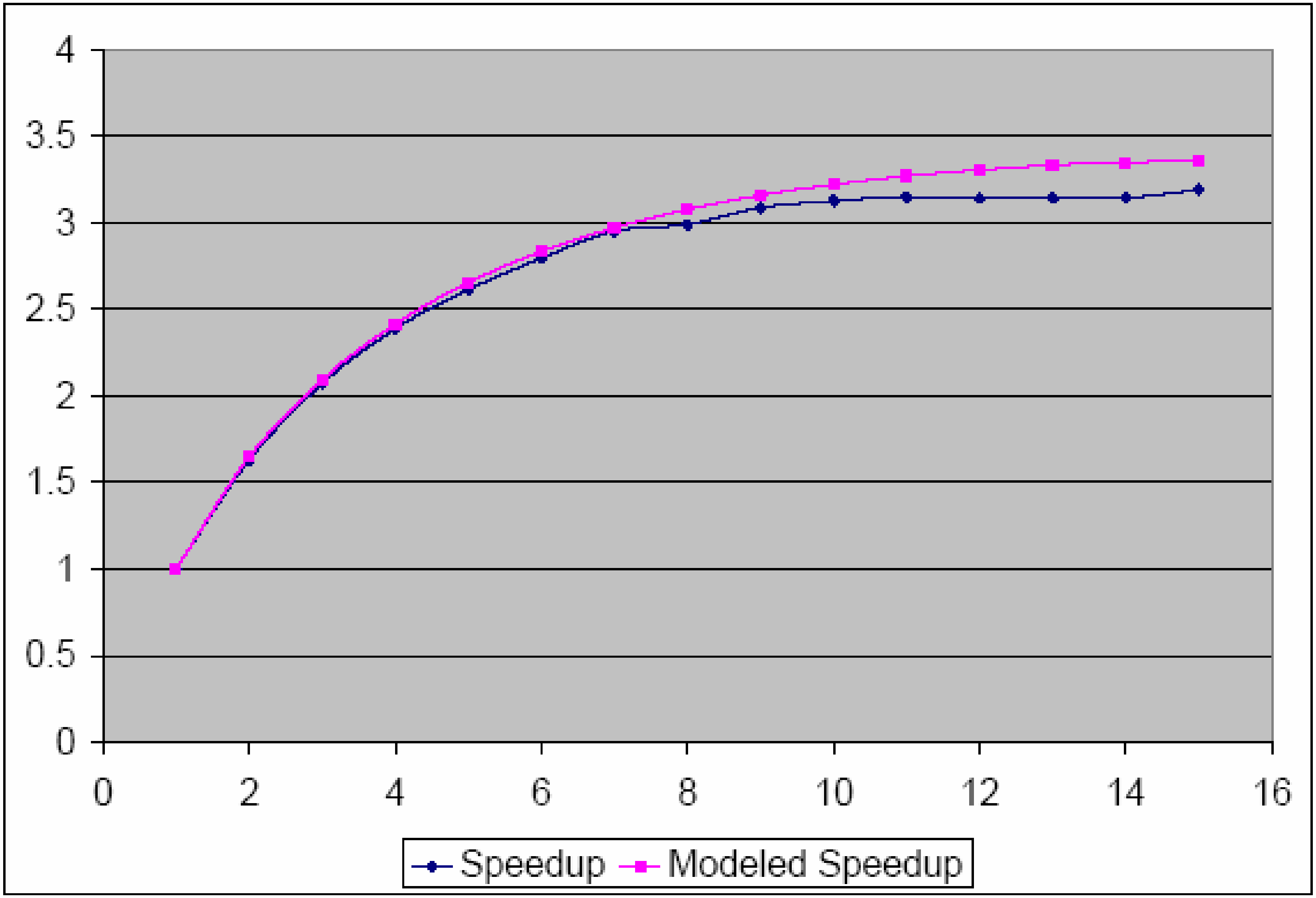


Foo_woo_2d_bw_perf

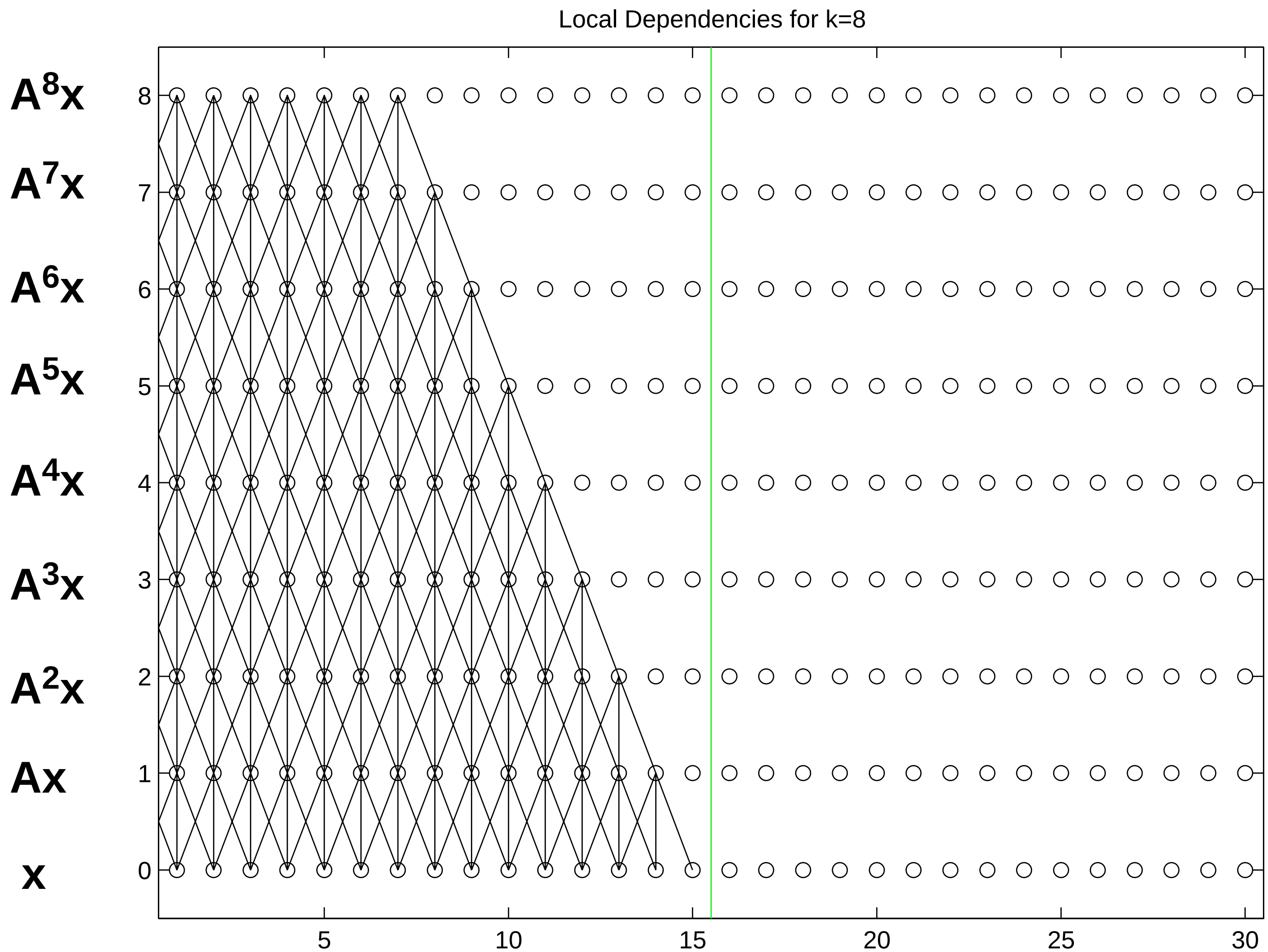
$p_{max} = 8192$, $\alpha = 10^{-5}$, $\beta = 1.5 \cdot 10^{-12}$, flops/s=$5 \cdot 10^{11}$, mem=$62.5 \cdot 10^{9}$, no overlap



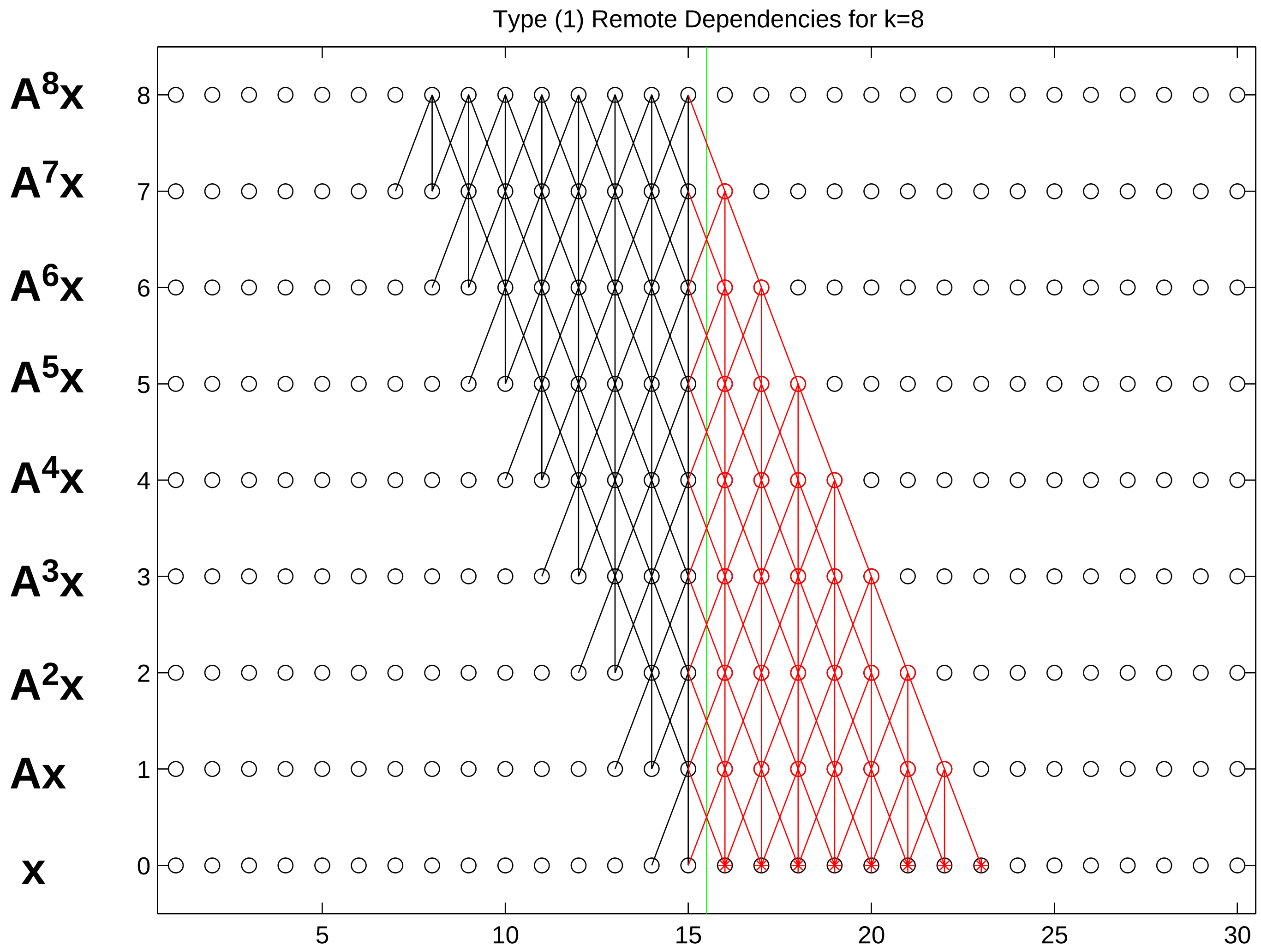
speedup_368_4



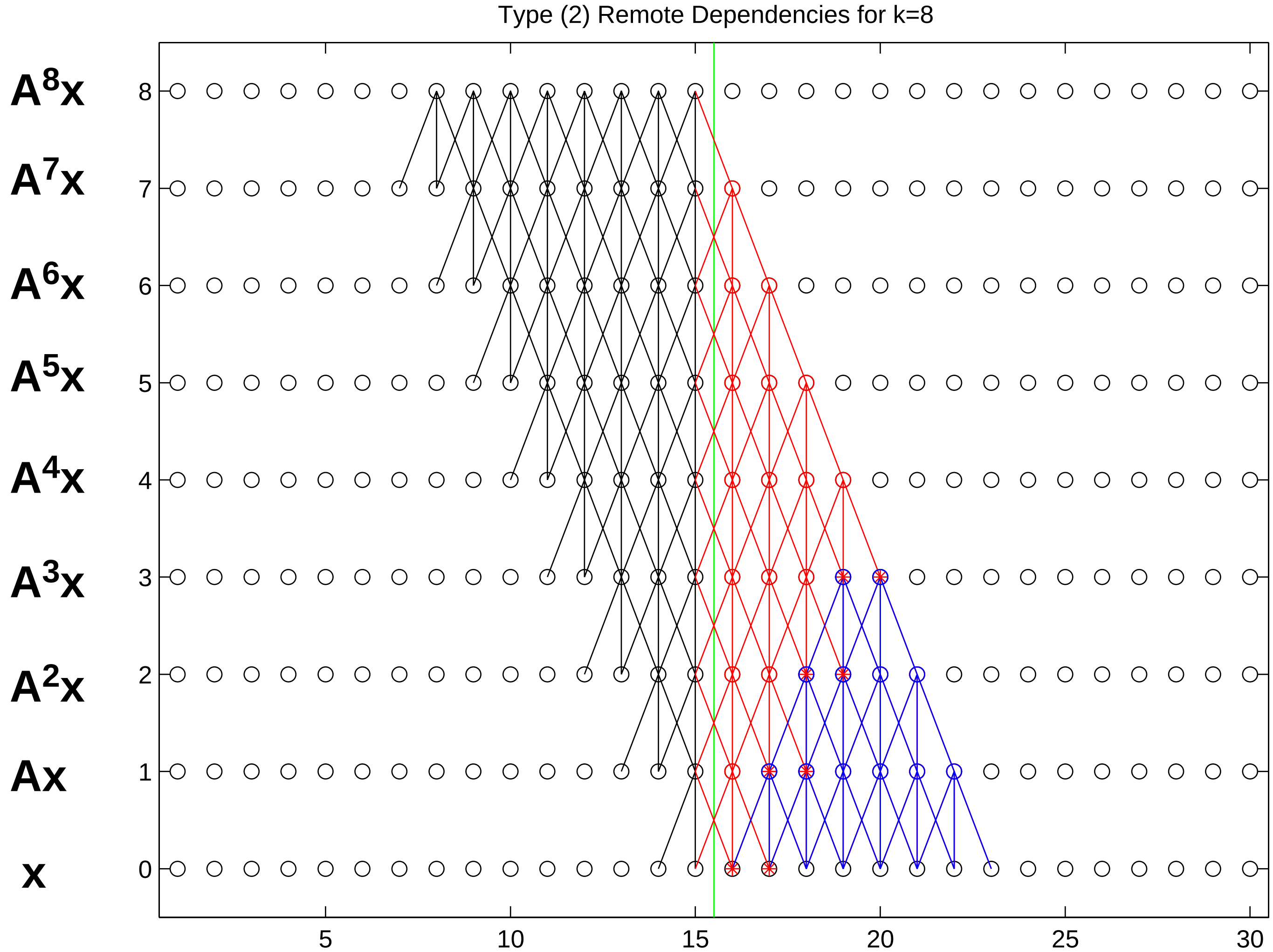
Locally Dependent Entries for $[x, Ax, \ldots, A^8x]$, A tridiagonal



Remotely Dependent Entries for [x,Ax,…,A⁸x], A tridiagonal



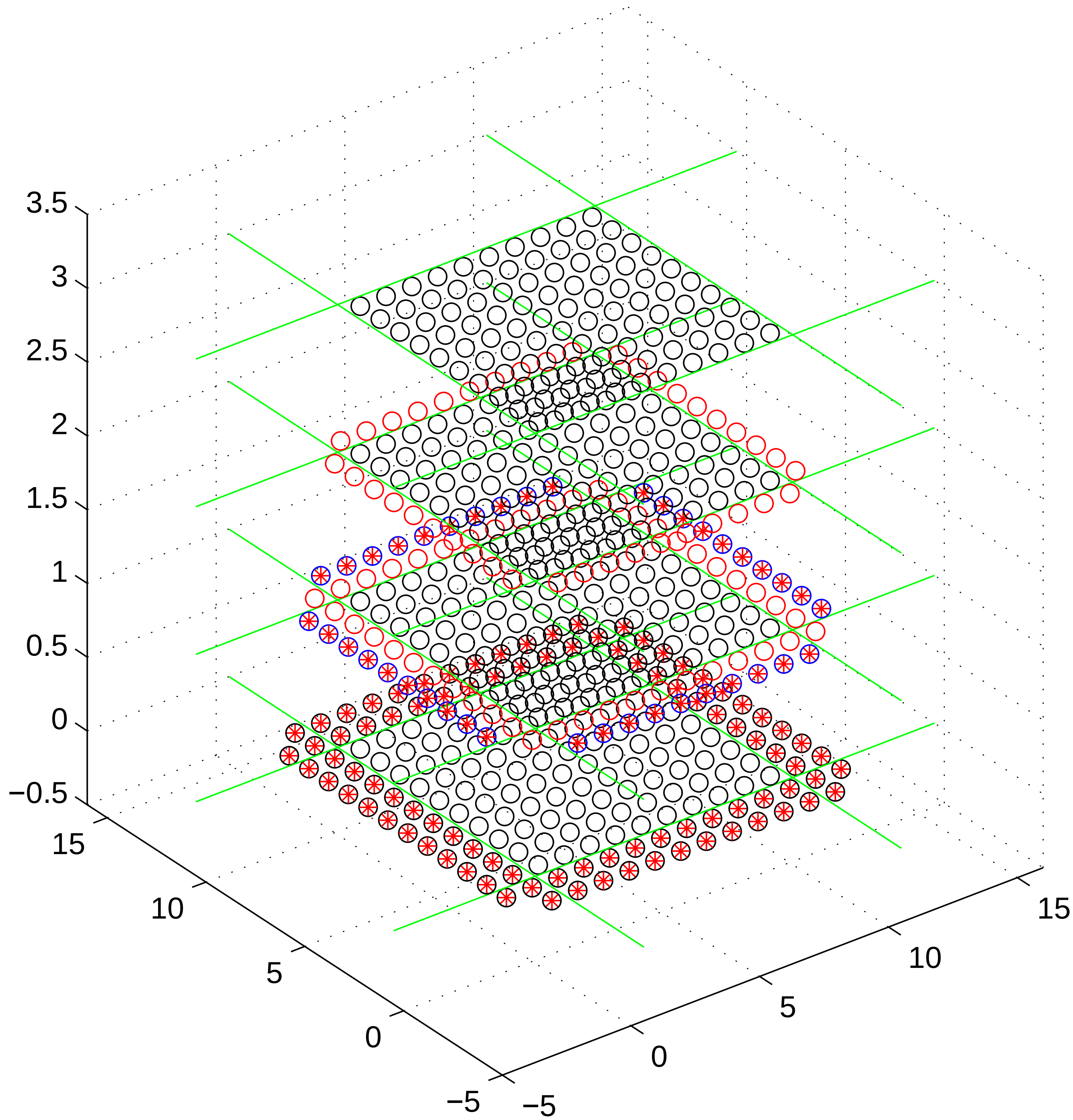
Fewest Remotely Dependent Entries for $[x, Ax, \ldots, A^8x]$, A tridiagonal

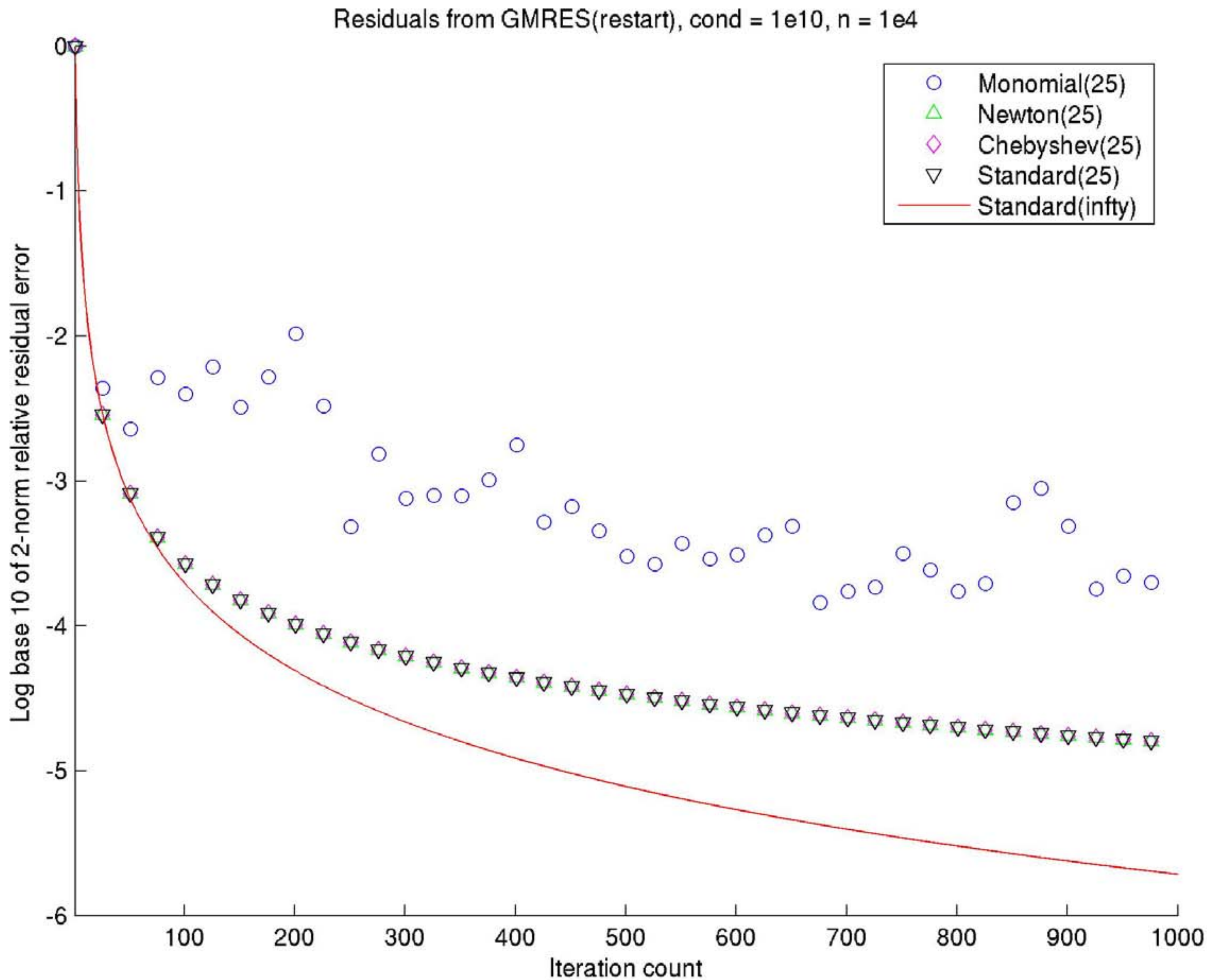


Latency Avoiding Parallel Kernel for $[x, Ax, A^2x, \ldots, A^kx]$

- Compute locally dependent entries needed by nghbrs
- Send data to nghbrs, receive from nghbrs
- Compute remaining locally dependent entries
- Wait for receive
- Compute remotely dependent entries

Remote dependencies for Approach (2) to 2D mesh with 5 pt stencil, 3D view




Residuals from GMRES(restart), cond = 1e10, n = 1e4
Monomial(25)
Newton(25)
Chebyshev(25)
Standard(25)
Standard(infty)
Log base 10 of 2-norm relative residual error
0
-1
-2
-3
-4
-5
-6
100
200
300
400
500
600
700
800
900
1000
Iteration count

Future Work in Automatic Performance Tuning for Sparse Matrix Algorithms

- **Include more important kernels**
  - **Better code generators / special purpose compilers**
- **Emerging architectures**
  - **Multicore, GPU, Petascale, …**
- **Change the interface to the machine**
  - **Put hardware into tuning loop (with RAMP)**
- **Change the interface to the application**
  - **If using SpMV as abstraction limits optimizations, change it**
  - **So we need new numerical algorithms too!**