Introduction to Communication-Avoiding Algorithms

www.cs.berkeley.edu/~demmel/SC12_tutorial

Jim Demmel

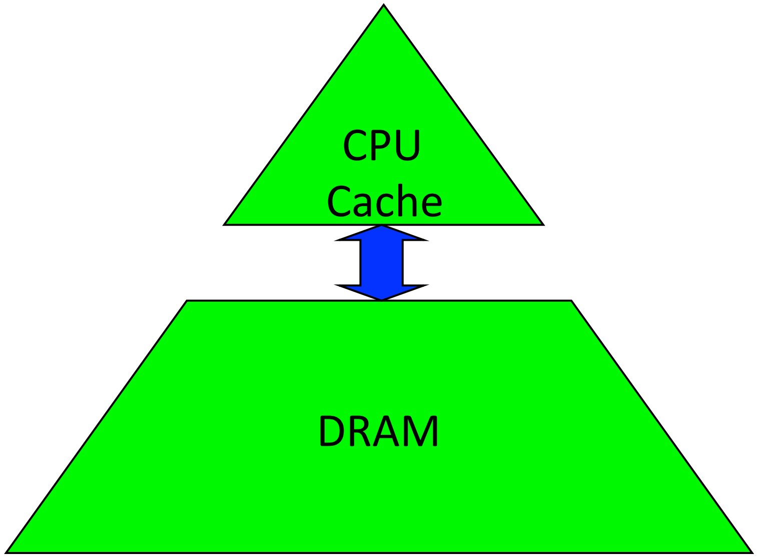
EECS & Math Departments

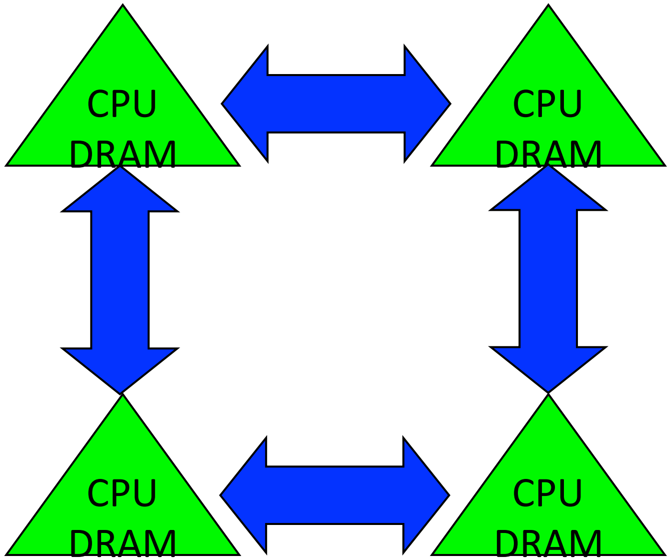
UC Berkeley

Why avoid communication? (1/2)

Algorithms have two costs (measured in time or energy):

1. Arithmetic (FLOPS)
2. Communication: moving data between
   - levels of a memory hierarchy (sequential case)
   - processors over a network (parallel case).





Why avoid communication? (2/3)

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

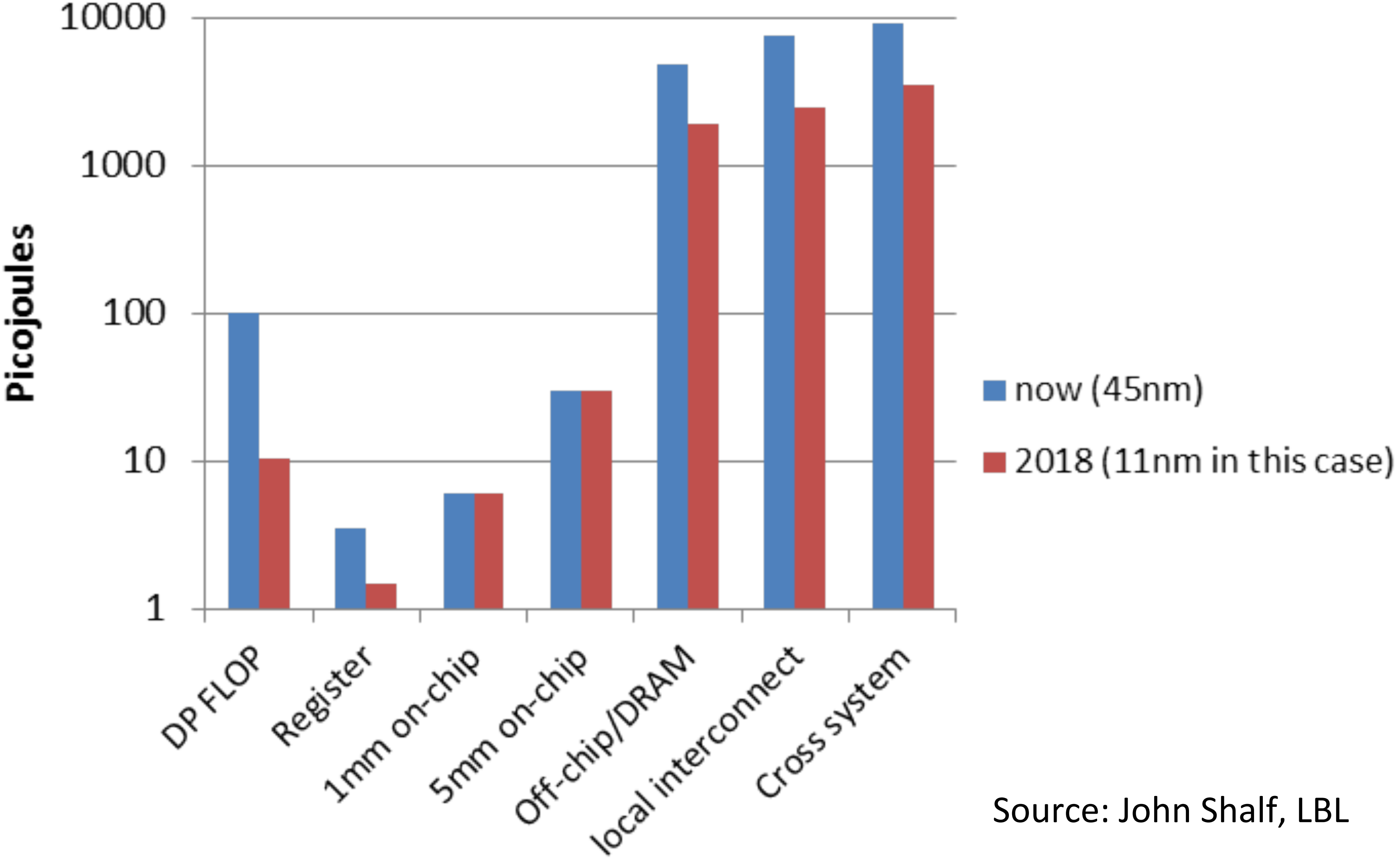
communication

- Time_per_flop << 1/ bandwidth << latency
  - Gaps growing exponentially with time [FOSC]

Annual improvements			
Time_per_flop		Bandwidth	Latency
59%	Network	26%	15%
	DRAM	23%	5%

- Avoid communication to save time

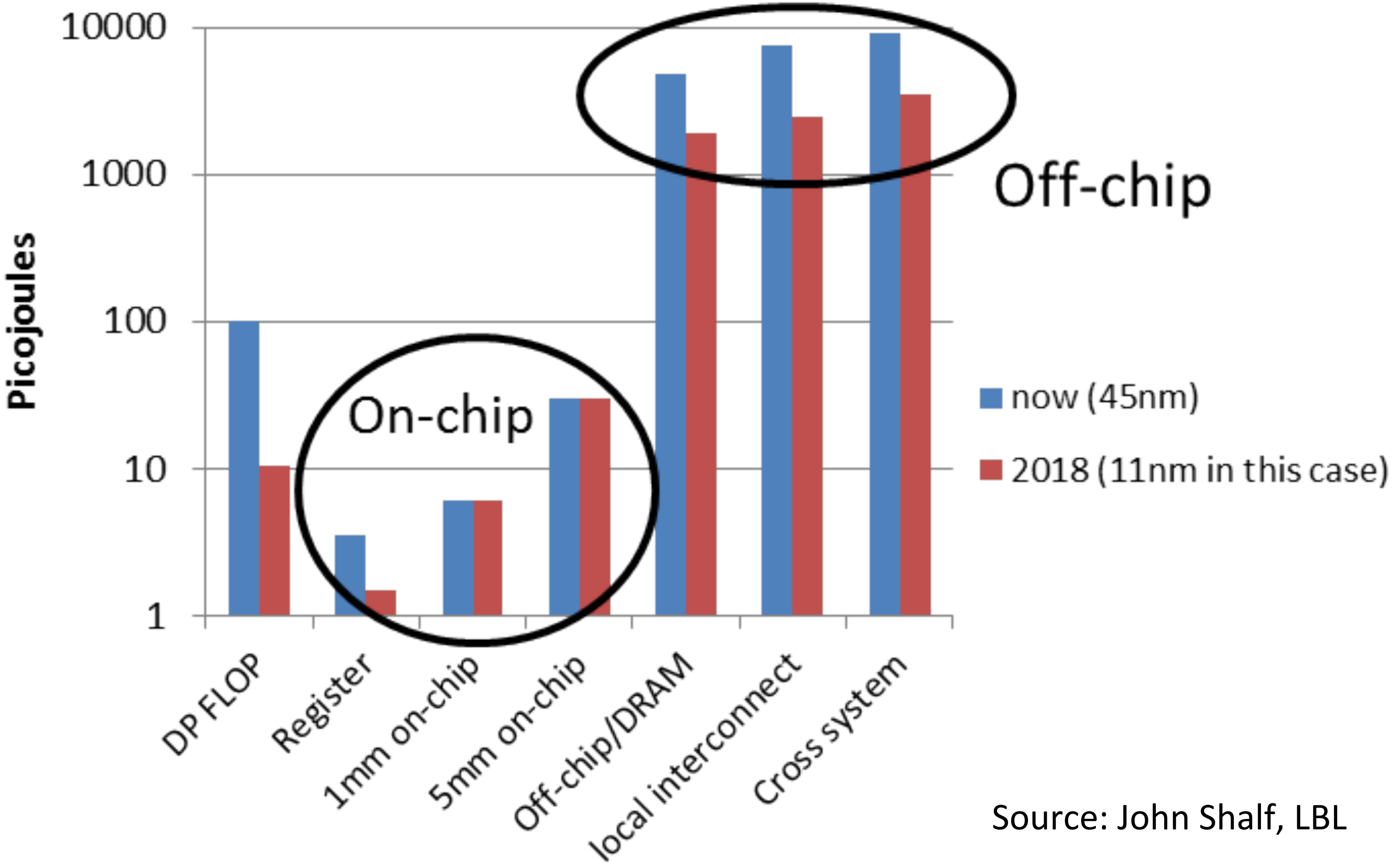
Why Minimize Communication? (2/2)



Source: John Shalf, LBL

Why Minimize Communication? (2/2)

Minimize communication to save energy



Source: John Shalf, LBL

Goals

- Redesign algorithms to *avoid* communication
  - Between all memory hierarchy levels
    - L1 ↔ L2 ↔ DRAM ↔ network, etc
- Attain lower bounds if possible
  - Current algorithms often far from lower bounds
  - Large speedups and energy savings possible

President Obama cites Communication-Avoiding Algorithms in the FY 2012 Department of Energy Budget Request to Congress:

“New Algorithm Improves Performance and Accuracy on Extreme-Scale Computing Systems. **On modern computer architectures, communication between processors takes longer than the performance of a floating point arithmetic operation by a given processor.** ASCR researchers have developed a new method, derived from commonly used linear algebra methods, to **minimize communications between processors and the memory hierarchy, by reformulating the communication patterns specified within the algorithm**. This method has been implemented in the TRILINOS framework, a highly-regarded suite of software, which provides functionality for researchers around the world to solve large scale, complex multi-physics problems.”

FY 2010 Congressional Budget, Volume 4, FY2010 Accomplishments, Advanced Scientific Computing Research (ASCR), pages 65-67.

CA-GMRES (Hoemmen, Mohiyuddin, Yelick, JD)
“Tall-Skinny” QR (Grigori, Hoemmen, Langou, JD)

Collaborators and Supporters

- Michael Christ, Jack Dongarra, Ioana Dumitriu, Armando Fox, David Gleich, Laura Grigori, Ming Gu, Mike Heroux, Mark Hoemmen, Olga Holtz, Kurt Keutzer, Julien Langou, Tom Scanlon, Michelle Strout, Sam Williams, Hua Xiang, Kathy Yelick
- Michael Anderson, Grey Ballard, Austin Benson, Abhinav Bhatele, Aydin Buluc, Erin Carson, Maryam Dehnavi, Michael Driscoll, Evangelos Georganas, Nicholas Knight, Penporn Koanantakool, Ben Lipshitz, Marghoob Mohiyuddin, Oded Schwartz, Edgar Solomonik
- Other members of ParLab, BEBOP, CACHE, EASI, FASTMath, MAGMA, PLASMA, TOPS projects

- Thanks to NSF, DOE, UC Discovery, Intel, Microsoft, Mathworks, National Instruments, NEC, Nokia, NVIDIA, Samsung, Oracle

- bebop.cs.berkeley.edu

Summary of CA Algorithms

- "Direct" Linear Algebra
  - Lower bounds on communication for linear algebra problems like Ax=b, least squares, Ax = λx, SVD, etc
  - New algorithms that attain these lower bounds
    - Being added to libraries: Sca/LAPACK, PLASMA, MAGMA
    - Large speed-ups possible
  - Autotuning to find optimal implementation
- Ditto for "Iterative" Linear Algebra
- Ditto (work in progress) for programs accessing arrays (eg n-body)

Outline

- "Direct" Linear Algebra
  - Lower bounds on communication
  - New algorithms that attain these lower bounds
- Ditto for "Iterative" Linear Algebra
- Ditto (work in progress) for programs accessing arrays (eg n-body)

Outline

- “Direct” Linear Algebra
  - Lower bounds on communication
  - New algorithms that attain these lower bounds
- Ditto for “Iterative” Linear Algebra
- Ditto (work in progress) for programs accessing arrays (eg n-body)

Lower bound for all “direct” linear algebra

- Let M = “fast” memory size (per processor)

#words_moved (per processor) = Ω(#flops (per processor) / $M^{1/2}$)

- Parallel case: assume either load or memory balanced

- Holds for
  - Matmul

Lower bound for all "direct" linear algebra

- Let M = "fast" memory size (per processor)

#words_moved (per processor) = Ω(#flops (per processor) / $M^{1/2}$)

#messages_sent ≥ #words_moved / largest_message_size

- Parallel case: assume either load or memory balanced

- Holds for
  - Matmul, BLAS, LU, QR, eig, SVD, tensor contractions, …
  - Some whole programs (sequences of these operations, no matter how individual ops are interleaved, eg A^k)
  - Dense and sparse matrices (where #flops << n^3)
  - Sequential and parallel algorithms
  - Some graph-theoretic algorithms (eg Floyd-Warshall)

Lower bound for all "direct" linear algebra

- Let M = "fast" memory size (per processor)

#words_moved (per processor) = Ω(#flops (per processor) / $M^{1/2}$)

#messages_sent (per processor) = Ω(#flops (per processor) / $M^{3/2}$)

- Parallel case: assume either load or memory balanced

- Holds for
  - Matmul, BLAS, LU, QR, eig, SVD, tensor contractions, …
  - Some whole programs (sequences of these operations, no matter how individual ops are interleaved, eg A^k)

SIAM SIAG/Linear Algebra Prize, 2012

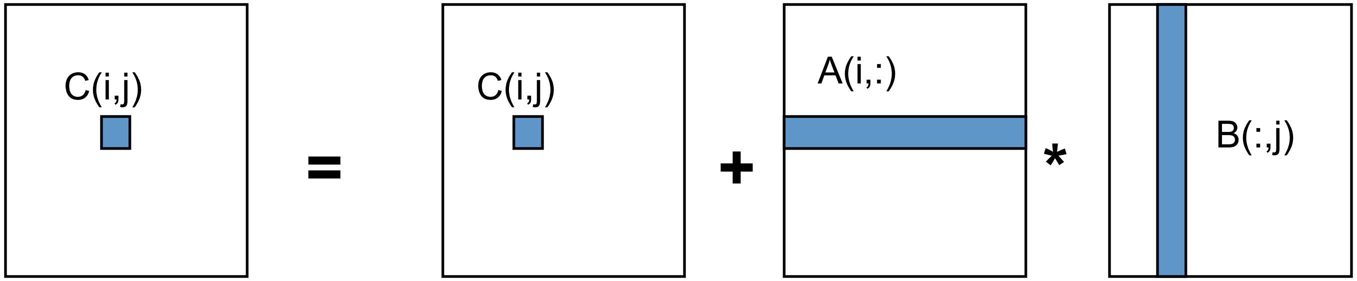
Ballard, D., Holtz, Schwartz

Can we attain these lower bounds?

- Do conventional dense algorithms as implemented in LAPACK and ScaLAPACK attain these bounds?
  - Often not
- If not, are there other algorithms that do?
  - Yes, for much of dense linear algebra
  - New algorithms, with new numerical properties, new ways to encode answers, new data structures
  - Not just loop transformations
- Only a few sparse algorithms so far
- Lots of work in progress
- Case study: Matrix Multiply

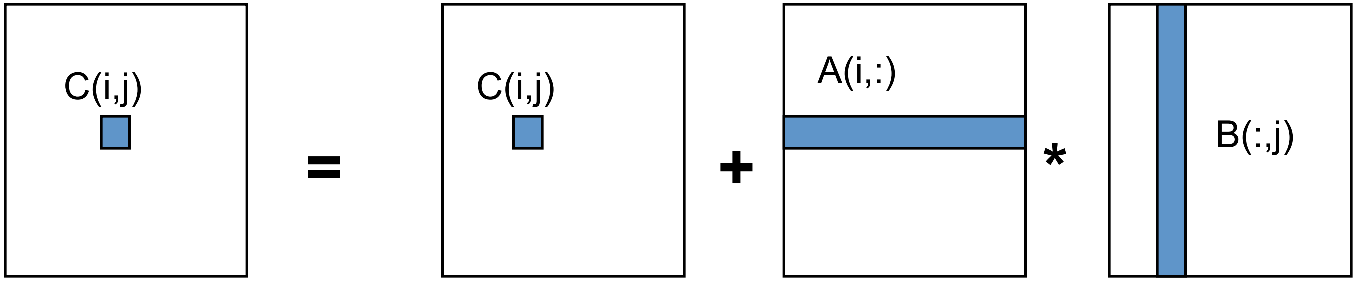
Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
```



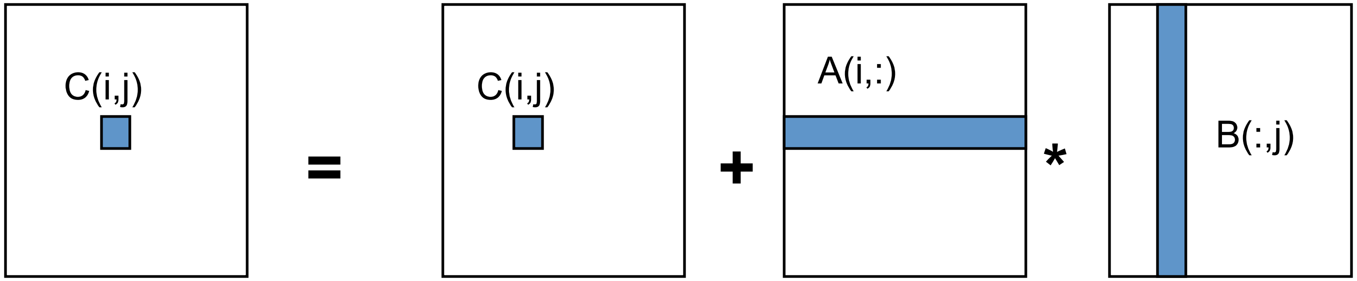
Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
      {read C(i,j) into fast memory}
      {read column j of B into fast memory}
      for k = 1 to n
         C(i,j) = C(i,j) + A(i,k) * B(k,j)
      {write C(i,j) back to slow memory}
```



Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}             … $n^2$ reads altogether
   for j = 1 to n
     {read C(i,j) into fast memory}              … $n^2$ reads altogether
     {read column j of B into fast memory}       … $n^3$ reads altogether
     for k = 1 to n
        C(i,j) = C(i,j) + A(i,k) * B(k,j)
     {write C(i,j) back to slow memory}          … $n^2$ writes altogether
```

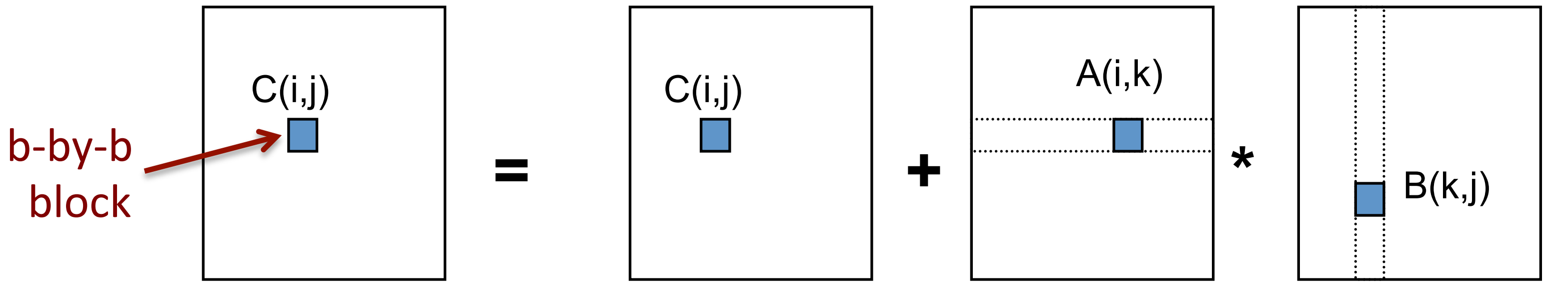


$n^3 + 3n^2$ reads/writes altogether – dominates $2n^3$ arithmetic

Blocked (Tiled) Matrix Multiply

Consider A,B,C to be n/b-by-n/b matrices of b-by-b subblocks where b is called the block size; assume 3 b-by-b blocks fit in fast memory

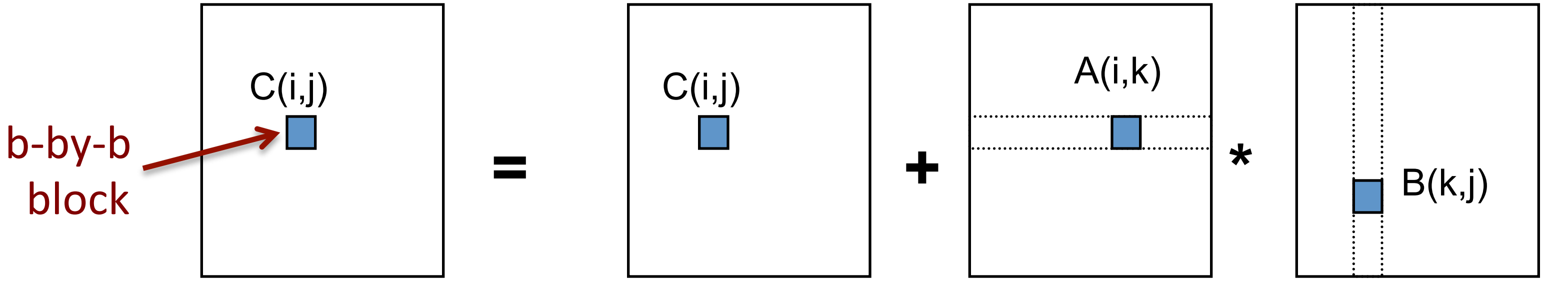
```
for i = 1 to n/b
  for j = 1 to n/b
    {read block C(i,j) into fast memory}
    for k = 1 to n/b
        {read block A(i,k) into fast memory}
        {read block B(k,j) into fast memory}
         C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
     {write block C(i,j) back to slow memory}
```



Blocked (Tiled) Matrix Multiply

Consider A,B,C to be n/b-by-n/b matrices of b-by-b subblocks where b is called the block size; assume 3 b-by-b blocks fit in fast memory

```
for i = 1 to n/b
   for j = 1 to n/b
      {read block C(i,j) into fast memory}          … $b^2 \times (n/b)^2 = n^2$ reads
      for k = 1 to n/b
         {read block A(i,k) into fast memory}    … $b^2 \times (n/b)^3 = n^3/b$ reads
         {read block B(k,j) into fast memory}    … $b^2 \times (n/b)^3 = n^3/b$ reads
         C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
      {write block C(i,j) back to slow memory}  … $b^2 \times (n/b)^2 = n^2$ writes
```



$2n^3/b + 2n^2$ reads/writes << $2n^3$ arithmetic - Faster!

Does blocked matmul attain lower bound?

- Recall: if 3 b-by-b blocks fit in fast memory of size M, then #reads/writes = $2n^3/b + 2n^2$
- Make b as large as possible: $3b^2 \leq M$, so #reads/writes $\geq 3^{1/2}n^3/M^{1/2} + 2n^2$
- Attains lower bound $= \Omega$ (#flops / $M^{1/2}$)

- But what if we don't know M?
- Or if there are multiple levels of fast memory?
- How do we write the algorithm?

Recursive Matrix Multiplication (RMM) (1/2)

- For simplicity: square matrices with $n = 2^m$
- $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

$$= \begin{pmatrix} A_{11}\cdot B_{11} + A_{12}\cdot B_{21} & A_{11}\cdot B_{12} + A_{12}\cdot B_{22} \\ A_{21}\cdot B_{11} + A_{22}\cdot B_{21} & A_{21}\cdot B_{12} + A_{22}\cdot B_{22} \end{pmatrix}$$

- True when each A_{ij} etc 1x1 or n/2 x n/2

```
func C = RMM (A, B, n)
   if n = 1, C = A * B, else
      {  $C_{11}$ = RMM ($A_{11}$ , $B_{11}$ , n/2) + RMM ($A_{12}$ , $B_{21}$ , n/2)
         $C_{12}$ = RMM ($A_{11}$ , $B_{12}$ , n/2) + RMM ($A_{12}$ , $B_{22}$ , n/2)
         $C_{21}$ = RMM ($A_{21}$ , $B_{11}$ , n/2) + RMM ($A_{22}$ , $B_{21}$ , n/2)
         $C_{22}$ = RMM ($A_{21}$ , $B_{12}$ , n/2) + RMM ($A_{22}$ , $B_{22}$ , n/2)  }
   return
```

Recursive Matrix Multiplication (RMM) (2/2)

```
func C = RMM (A, B, n)
    if n=1, C = A * B, else
        {  $C_{11}$ = RMM ($A_{11}$ , $B_{11}$ , n/2) + RMM ($A_{12}$ , $B_{21}$ , n/2)
           $C_{12}$ = RMM ($A_{11}$ , $B_{12}$ , n/2) + RMM ($A_{12}$ , $B_{22}$ , n/2)
           $C_{21}$ = RMM ($A_{21}$ , $B_{11}$ , n/2) + RMM ($A_{22}$ , $B_{21}$ , n/2)
           C    = RMM (A    , B    , n/2) + RMM (A    , B    , n/2)  }
```

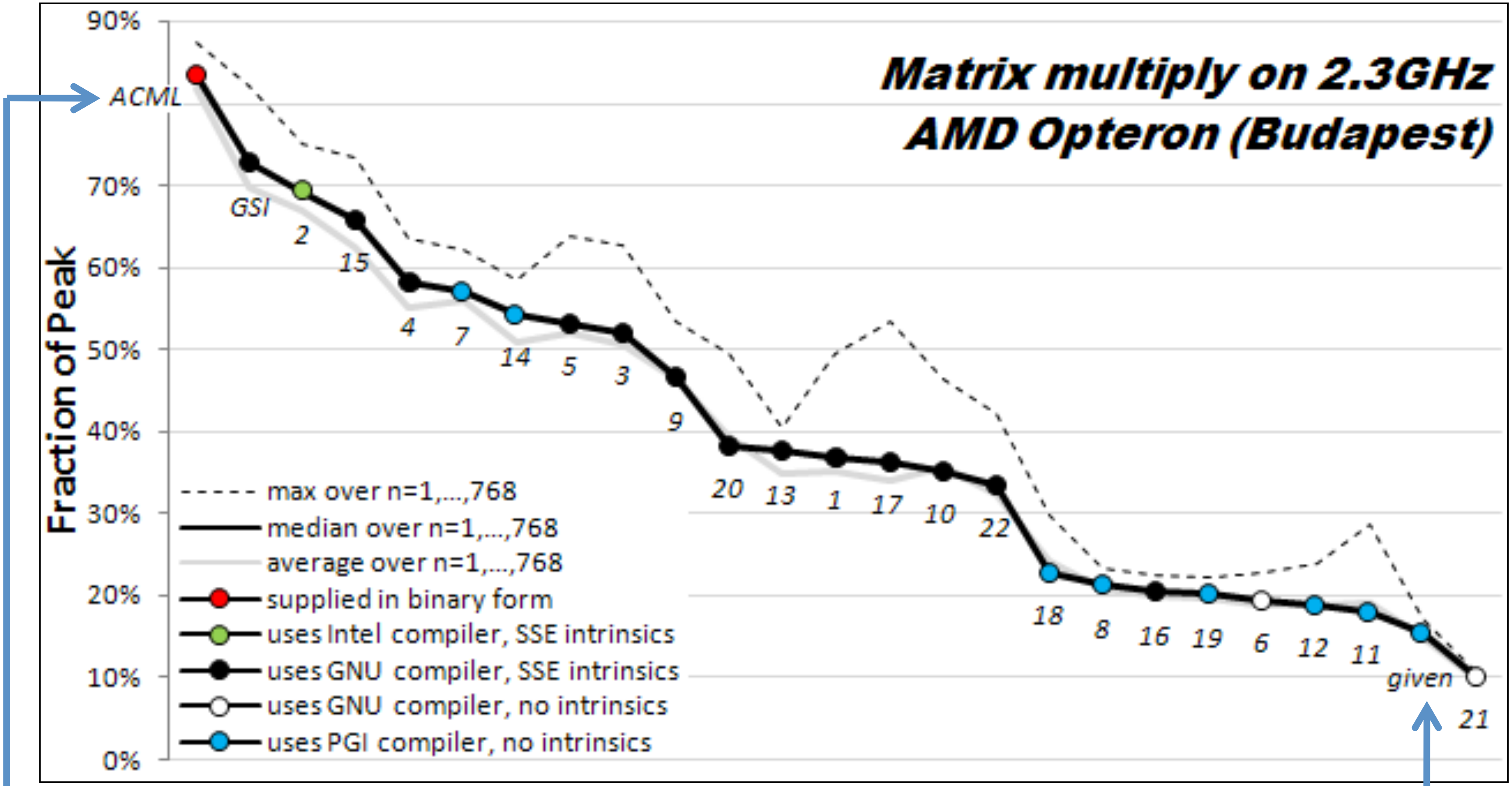
For big speedups, see SC12 poster on “Beating MKL and ScaLAPACK at Rectangular Matmul” 11/13 at 5:15-7pm

, n)

$= O(n^3 / M^{1/2} + n^2)$ … same as blocked matmul

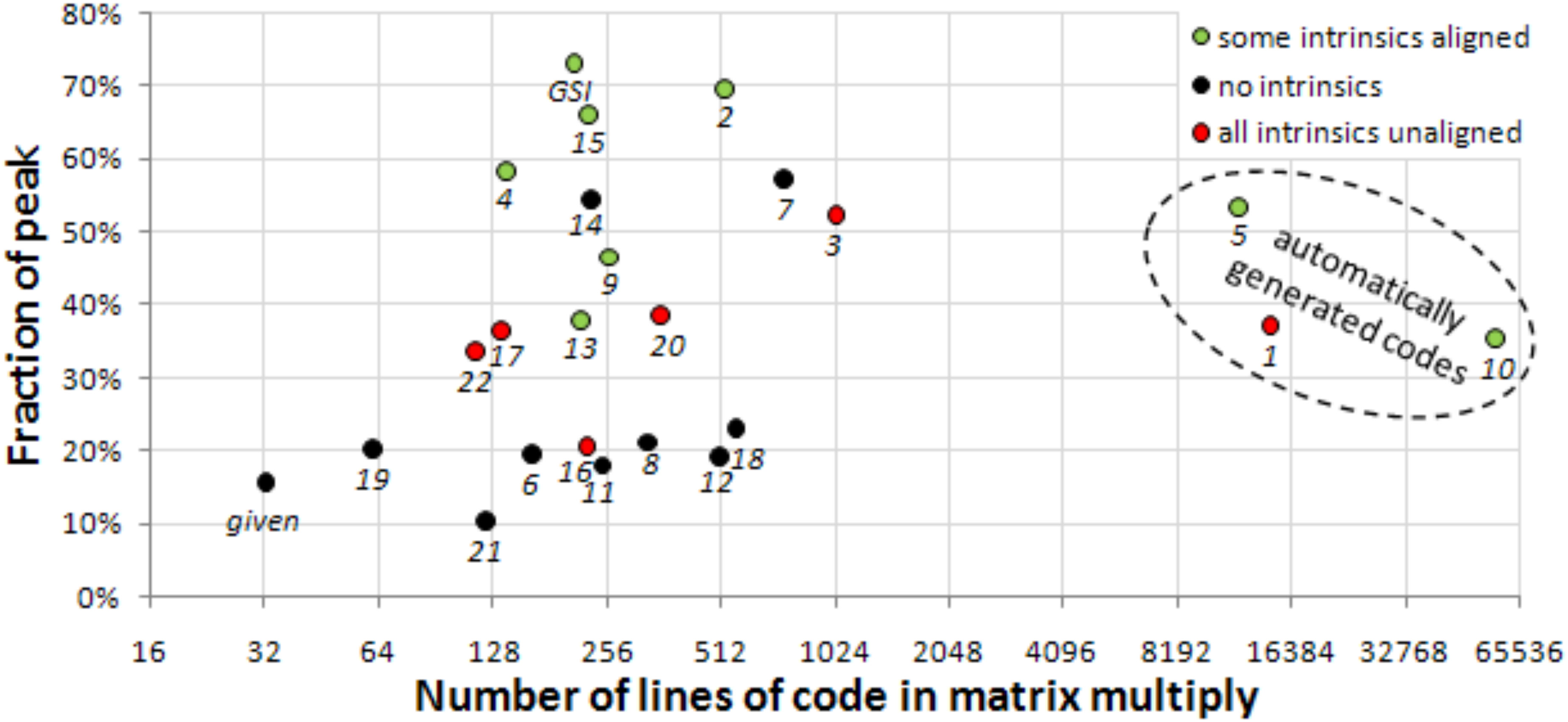
“Cache oblivious”, works for memory hierarchies, but not panacea

How hard is hand-tuning matmul, anyway?



- **Results of 22 student teams trying to tune matrix-multiply, in CS267 Spr09**
- **Students given "blocked" code to start with (7x faster than naïve)**
  - **Still hard to get close to vendor tuned performance (ACML) (another 6x)**
- **For more discussion, see www.cs.berkeley.edu/~volkov/cs267.sp09/hw1/results/**

How hard is hand-tuning matmul, anyway?



Parallel MatMul with 2D Processor Layout

- P processors in $P^{1/2}$ x $P^{1/2}$ grid
  - Processors communicate along rows, columns
- Each processor owns $n/P^{1/2}$ x $n/P^{1/2}$ submatrices of A,B,C
- Example: P=16, processors numbered from P_{00} to P_{33}
  - Processor P_{ij} owns submatrices A_{ij}, B_{ij} and C_{ij}

C = A * B

C

P_{00}	P_{01}	P_{02}	P_{03}
P_{10}	P_{11}	P_{12}	P_{13}
P_{20}	P_{21}	P_{22}	P_{23}
P_{30}	P_{31}	P_{32}	P_{33}

A

P_{00}	P_{01}	P_{02}	P_{03}
P_{10}	P_{11}	P_{12}	P_{13}
P_{20}	P_{21}	P_{22}	P_{23}
P_{30}	P_{31}	P_{32}	P_{33}

B

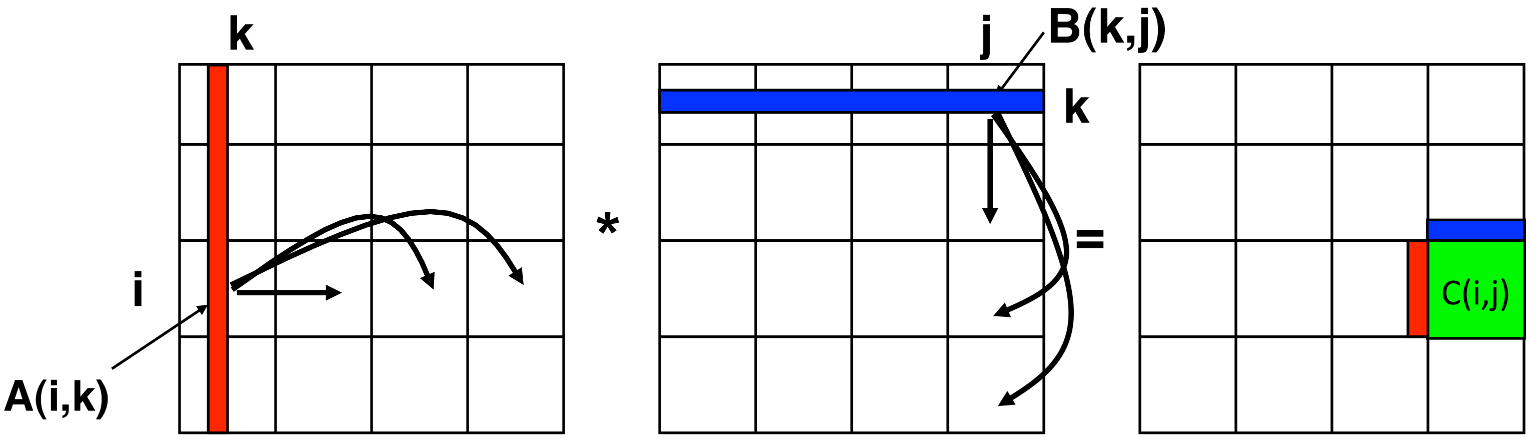
P_{00}	P_{01}	P_{02}	P_{03}
P_{10}	P_{11}	P_{12}	P_{13}
P_{20}	P_{21}	P_{22}	P_{23}
P_{30}	P_{31}	P_{32}	P_{33}

SUMMA Algorithm

- SUMMA = Scalable Universal Matrix Multiply
  - Comes within factor of log P of lower bounds:
    - Assume fast memory size M = $O(n^2/P)$ per processor – 1 copy of data
    - #words_moved = $\Omega(\ \#\text{flops} / M^{1/2}\ ) = \Omega(\ (n^3/P) / (n^2/P)^{1/2}\ ) = \Omega(\ n^2 / P^{1/2}\ )$
    - #messages = $\Omega(\ \#\text{flops} / M^{3/2}\ ) = \Omega(\ (n^3/P) / (n^2/P)^{3/2}\ ) = \Omega(\ P^{1/2}\ )$
  - Can accommodate any processor grid, matrix dimensions & layout
  - Used in practice in PBLAS = Parallel BLAS
    - www.netlib.org/lapack/lawns/lawn{96,100}.ps

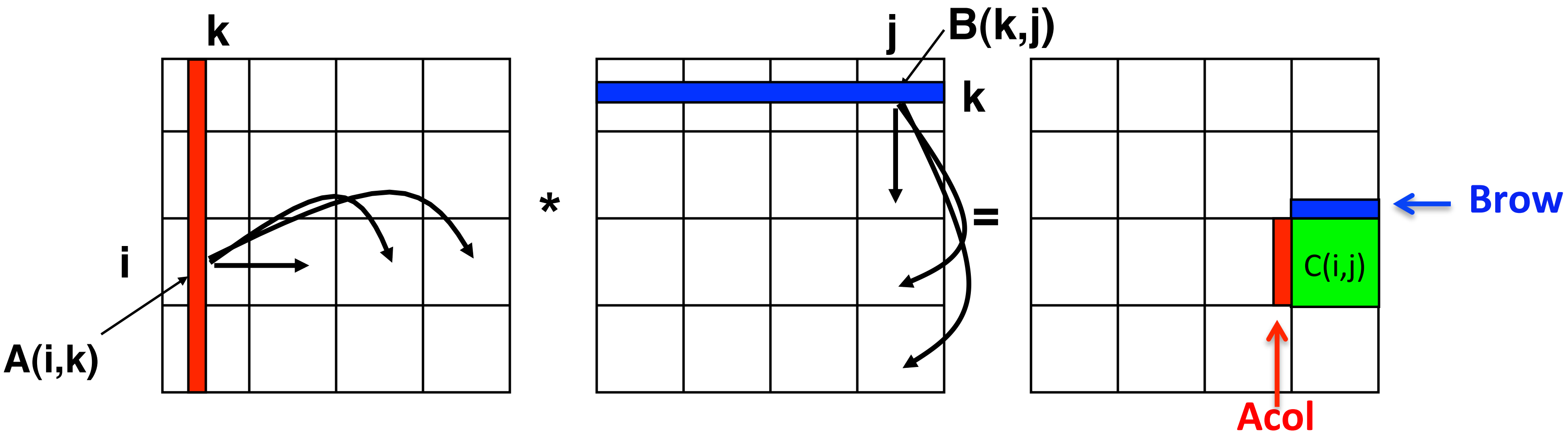
- Comparison to Cannon's Algorithm
  - Cannon attains lower bound
  - But Cannon harder to generalize to other grids, dimensions, layouts, and Cannon may use more memory

SUMMA – n x n matmul on $P^{1/2}$ x $P^{1/2}$ grid



- C(i, j) is n/$P^{1/2}$ x n/$P^{1/2}$ submatrix of C on processor P_{ij}
- A(i,k) is n/$P^{1/2}$ x b submatrix of A
- B(k,j) is b x n/$P^{1/2}$ submatrix of B
- $C(i,j) = C(i,j) + \Sigma_k A(i,k)*B(k,j)$
  - summation over submatrices
- Need not be square processor grid

SUMMA– n x n matmul on $P^{1/2}$ x $P^{1/2}$ grid



```
For k=0 to n/b-1
    for all i = 1 to P^{1/2}
           owner of A(i,k) broadcasts it to whole processor row (using binary tree)
    for all j = 1 to  P^{1/2}
          owner of B(k,j) broadcasts it to whole processor column (using bin. tree)
    Receive A(i,k) into Acol
    Receive B(k,j) into Brow
    C_myproc = C_myproc + Acol * Brow
```

SUMMA Communication Costs

```
For k=0 to n/b-1
   for all i = 1 to $P^{1/2}$
        owner of A(i,k) broadcasts it to whole processor row (using binary tree)
         … #words  = $\log P^{1/2} * b * n/P^{1/2}$ ,     #messages = $\log P^{1/2}$
   for all j = 1 to  $P^{1/2}$
        owner of B(k,j) broadcasts it to whole processor column (using bin. tree)
         …  same #words and #messages
   Receive A(i,k) into Acol
   Receive B(k,j) into Brow
   C_myproc = C_myproc + Acol * Brow
```

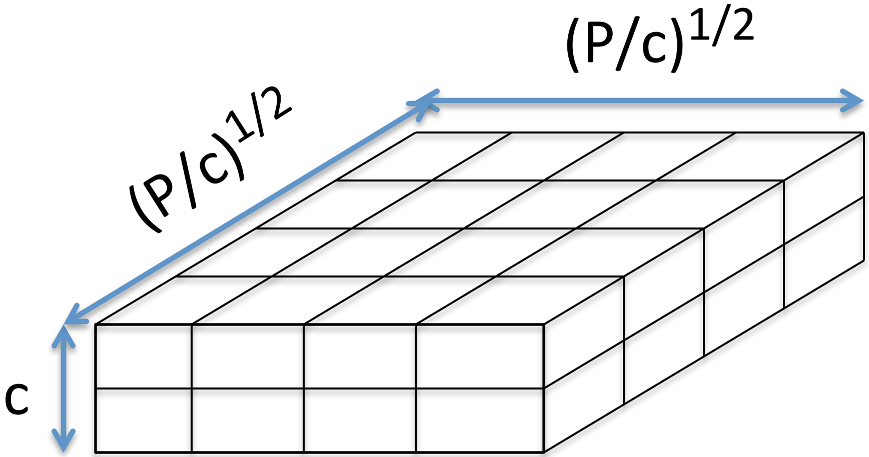
- Total #words = $\log P * n^2 / P^{1/2}$
  - Within factor of log P of lower bound
  - (more complicated implementation removes log P factor)
- Total #messages = $\log P * n/b$
  - Choose b close to maximum, $n/P^{1/2}$, to approach lower bound $P^{1/2}$

Can we do better?

- Lower bound assumed 1 copy of data: $M = O(n^2/P)$ per proc.
- What if matrix small enough to fit c>1 copies, so $M = cn^2/P$?
  - #words_moved = Ω(#flops / $M^{1/2}$) = $\Omega(n^2 / (c^{1/2} P^{1/2}))$
  - #messages = Ω(#flops / $M^{3/2}$) = $\Omega(P^{1/2} / c^{3/2})$
- Can we attain new lower bound?
  - Special case: "3D Matmul": $c = P^{1/3}$
    - Dekel, Nassimi, Sahni [81], Bernsten [89], Agarwal, Chandra, Snir [90], Johnson [93], Agarwal, Balle, Gustavson, Joshi, Palkar [95]
    - Processors arranged in $P^{1/3} \times P^{1/3} \times P^{1/3}$ grid
    - Processor (i,j,k) performs $C(i,j) = C(i,j) + A(i,k)*B(k,j)$, where each submatrix is $n/P^{1/3} \times n/P^{1/3}$
  - Not always that much memory available…

2.5D Matrix Multiplication

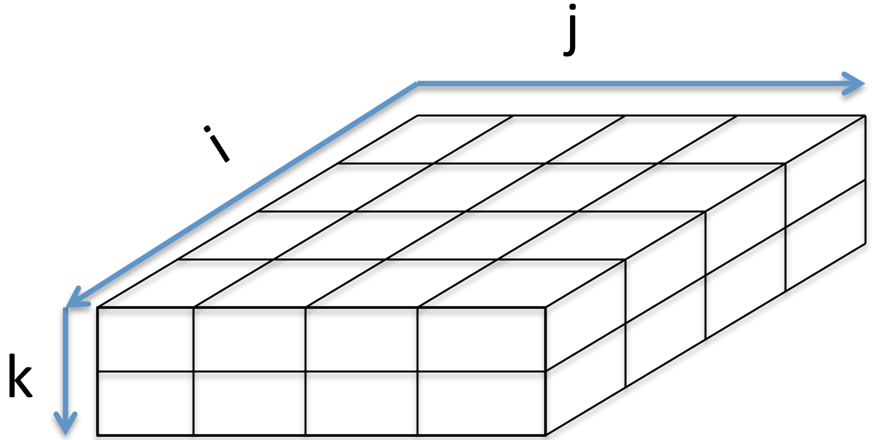
- Assume can fit cn^2/P data per processor, c>1
- Processors form $(P/c)^{1/2}$ x $(P/c)^{1/2}$ x c grid



Example: P = 32, c = 2

2.5D Matrix Multiplication

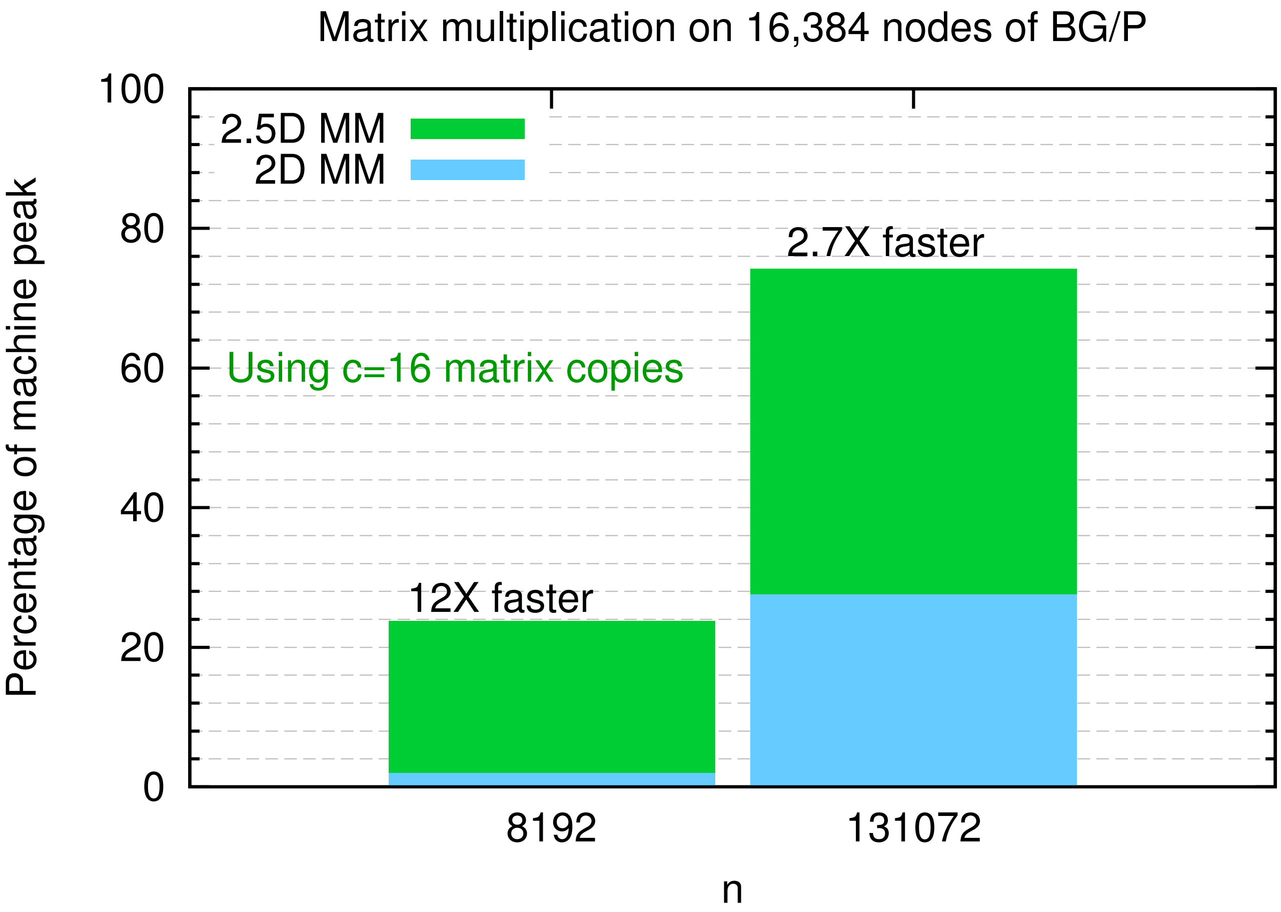
- Assume can fit cn^2/P data per processor, $c > 1$
- Processors form $(P/c)^{1/2}$ x $(P/c)^{1/2}$ x c grid



Initially P(i,j,0) owns A(i,j) and B(i,j) each of size $n(c/P)^{1/2}$ x $n(c/P)^{1/2}$

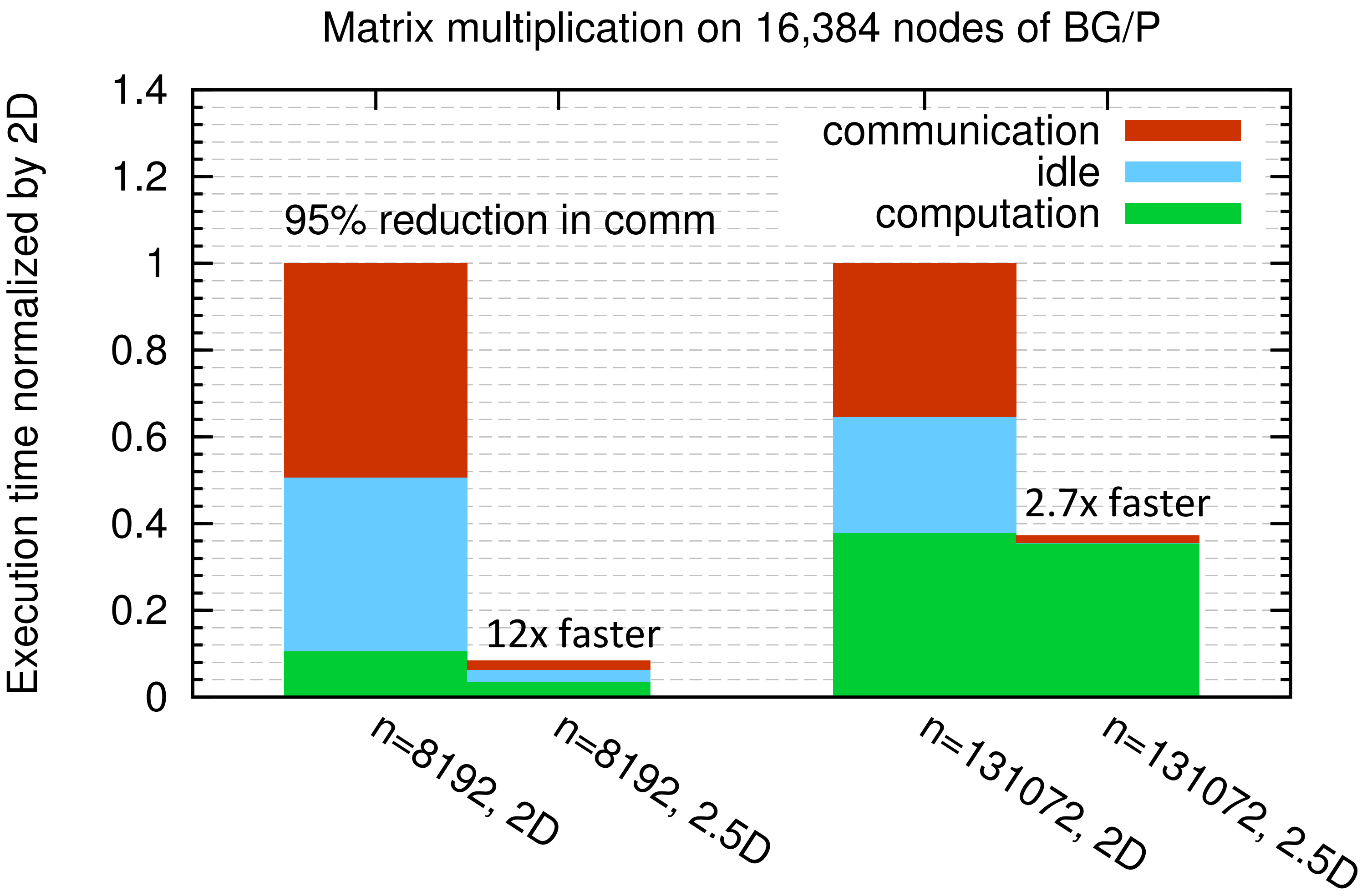
(1) P(i,j,0) broadcasts A(i,j) and B(i,j) to P(i,j,k)
(2) Processors at level k perform 1/c-th of SUMMA, i.e. 1/c-th of Σ_m A(i,m)*B(m,j)
(3) Sum-reduce partial sums Σ_m A(i,m)*B(m,j) along k-axis so P(i,j,0) owns C(i,j)

2.5D Matmul on BG/P, 16K nodes / 64K cores



2.5D Matmul on BG/P, 16K nodes / 64K cores

c = 16 copies



95% reduction in comm

12x faster

2.7x faster

Distinguished Paper Award, EuroPar'11

SC'11 paper by Solomonik, Bhatele, D.

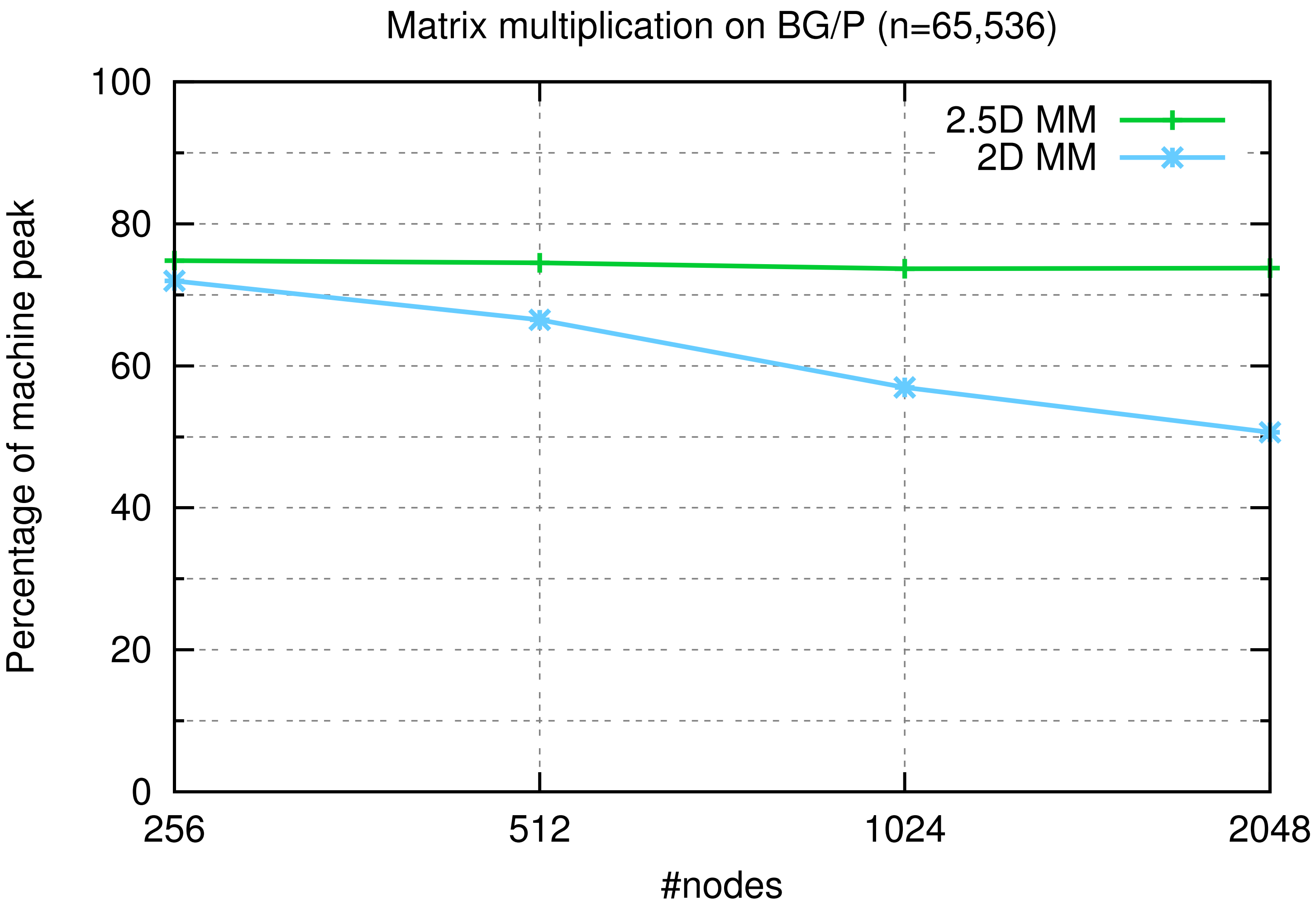
Perfect Strong Scaling – in Time and Energy (1/2)

- Every time you add a processor, you should use its memory M too
- Start with minimal number of procs: $PM = 3n^2$
- Increase P by a factor of c ➔ total memory increases by a factor of c
- Notation for timing model:
  - γ_T , β_T , α_T = secs per flop, per word_moved, per message of size m
- $T(cP) = n^3/(cP)\ [\ \gamma_T + \beta_T/M^{1/2} + \alpha_T/(mM^{1/2})\ ]$

  $= T(P)/c$
- Notation for energy model:
  - γ_E , β_E , α_E = joules for same operations
  - δ_E = joules per word of memory used per sec
  - ε_E = joules per sec for leakage, etc.
- $E(cP) = cP\ \{\ n^3/(cP)\ [\ \gamma_E + \beta_E/M^{1/2} + \alpha_E/(mM^{1/2})\ ] + \delta_E M T(cP) + \varepsilon_E T(cP)\ \}$

  $= E(P)$

Perfect Strong Scaling in Time of 2.5D Matmul on BG/P, n=64K



Perfect Strong Scaling – in Time and Energy (2/2)

- Perfect scaling extends to N-body, Strassen, …
- We can use these models to answer many questions, including:

- What is the minimum energy required for a computation?
- Given a maximum allowed runtime **T** , what is the minimum energy **E** needed to achieve it?
- Given a maximum energy budget **E** , what is the minimum runtime **T** that we can attain?
- The ratio **P** = **E/T** gives us the average power required to run the algorithm. Can we minimize the average power consumed?
- Given an algorithm, problem size, number of processors and target energy efficiency (GFLOPS/W), can we determine a set of architectural parameters to describe a conforming computer architecture?

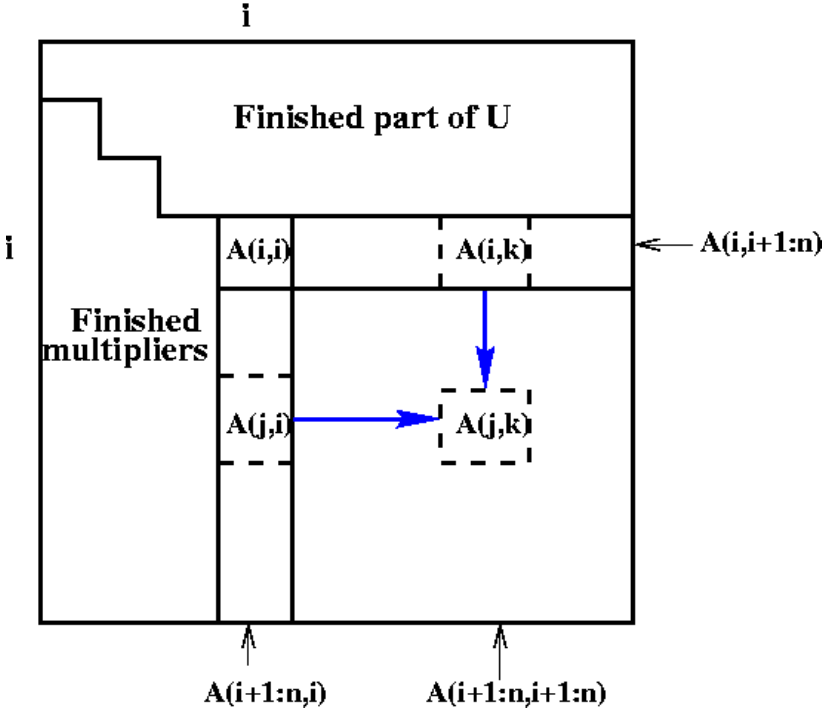
Extending to rest of Direct Linear Algebra
Naïve Gaussian Elimination: A =LU

```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) * ( 1 / A(i,i) )
          …  scale a vector
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n )
           - A(i+1:n , i) * A(i , i+1:n)
           … rank-1 update
```

→

```
for  i=1 to n-1
      update column i
      update trailing  matrix
```

Work at step i of Gaussian Elimination



Communication in sequential One-sided Factorizations (LU, QR, …)

- Naive Approach

```
for i=1 to n-1
    update column i
    update trailing matrix
```

- #words_moved = $O(n^3)$

- Blocked Approach (LAPACK)

```
for i=1 to n/b - 1
    update block i of b columns
    update trailing matrix
```

- #words moved = $O(n^3/M^{1/3})$

- Recursive Approach

```
func factor(A)
    if A has 1 column, update it
else
        factor(left half of A)
        update right half of A
        factor(right half of A)
```

- #words moved = $O(n^3/M^{1/2})$

- None of these approaches
  - minimizes #messages
  - handles eig() or svd()
  - works in parallel
- Need more ideas

TSQR: QR of a Tall, Skinny matrix

$$W = \left(\begin{array}{c} W_0 \\ \hline W_1 \\ \hline W_2 \\ \hline W_3 \end{array}\right)$$

$$\left(\begin{array}{c} R_{00} \\ R_{10} \\ \hline R_{20} \\ R_{30} \end{array}\right) = \left(\begin{array}{c} Q_{01}\ R_{01} \\ \hline Q_{11}\ R_{11} \end{array}\right)$$

$$\left(\begin{array}{c} R_{01} \\ \hline R_{11} \end{array}\right) = \left(\ Q_{02}\ R_{02}\ \right)$$

TSQR: QR of a Tall, Skinny matrix

$$W = \left(\begin{array}{c} W_0 \\ \hline W_1 \\ \hline W_2 \\ \hline W_3 \end{array}\right) = \left(\begin{array}{c} Q_{00}\,R_{00} \\ \hline Q_{10}\,R_{10} \\ \hline Q_{20}\,R_{20} \\ \hline Q_{30}\,R_{30} \end{array}\right) = \left(\begin{array}{cccc} Q_{00} & & & \\ \hline & Q_{10} & & \\ \hline & & Q_{20} & \\ \hline & & & Q_{30} \end{array}\right) \cdot \left(\begin{array}{c} R_{00} \\ \hline R_{10} \\ \hline R_{20} \\ \hline R_{30} \end{array}\right)$$

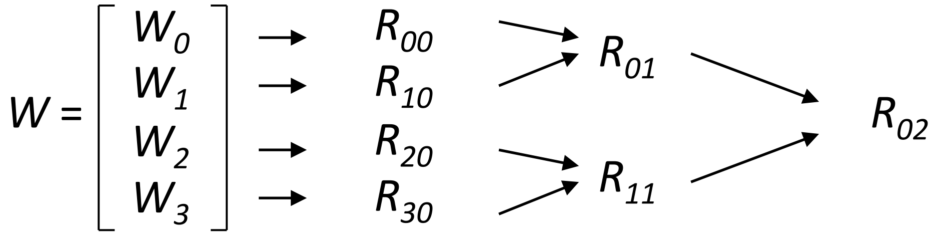
$$\left(\begin{array}{c} R_{00} \\ R_{10} \\ \hline R_{20} \\ R_{30} \end{array}\right) = \left(\begin{array}{c} Q_{01}\,R_{01} \\ \hline Q_{11}\,R_{11} \end{array}\right) = \left(\begin{array}{cc} Q_{01} & \\ \hline & Q_{11} \end{array}\right) \cdot \left(\begin{array}{c} R_{01} \\ \hline R_{11} \end{array}\right)$$

$$\left(\begin{array}{c} R_{01} \\ \hline R_{11} \end{array}\right) = \left(Q_{02}\,R_{02} \right)$$

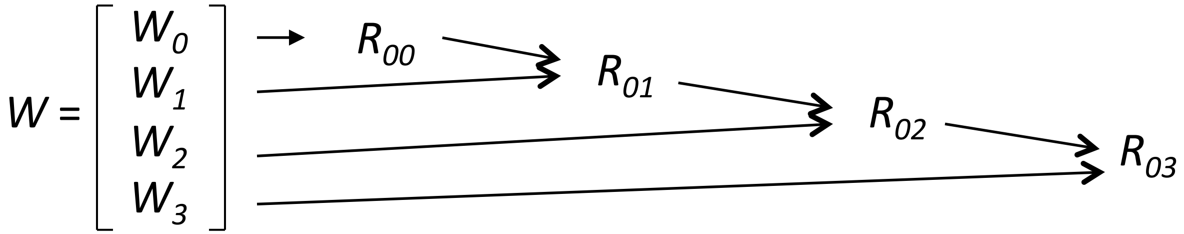
$$\text{Output} = \{\, Q_{00}, Q_{10}, Q_{20}, Q_{30}, Q_{01}, Q_{11}, Q_{02}, R_{02} \,\}$$

TSQR: An Architecture-Dependent Algorithm

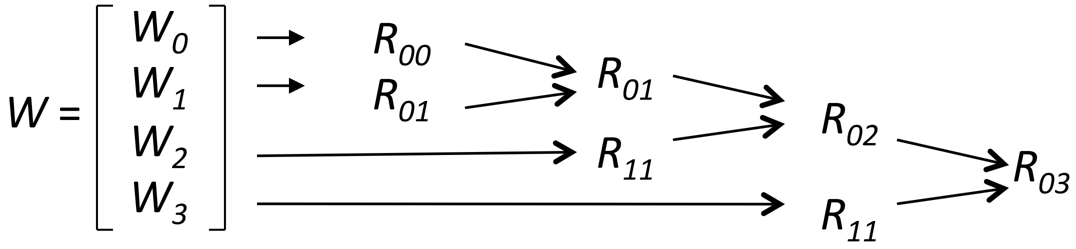
Parallel:


$W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix}$
R_{00}
R_{10}
R_{20}
R_{30}
R_{01}
R_{11}
R_{02}

Sequential:


$W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix}$
R_{00}
R_{01}
R_{02}
R_{03}

Dual Core:


$W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix}$
R_{00}
R_{01}
R_{01}
R_{11}
R_{02}
R_{11}
R_{03}

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

Can choose reduction tree dynamically

TSQR Performance Results

- Parallel
  - Intel Clovertown
    - Up to **8x** speedup (8 core, dual socket, 10M x 10)
  - 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)
  - Tesla C 2050 / Fermi
    - Up to **13x** (110,592 x 100)
  - Grid – **4x** on 4 cities (Dongarra et al)
  - Cloud – **1.6x slower than accessing data twice** (Gleich and Benson)
- Sequential
  - "**Infinite speedup**" for out-of-Core on PowerPC laptop
    - As little as 2x slowdown vs (predicted) infinite DRAM
    - LAPACK with virtual memory never finished
- SVD costs about the same
- Joint work with Grigori, Hoemmen, Langou, Anderson, Ballard, Keutzer, others

Back to LU: Using similar idea for TSLU as TSQR: Use reduction tree, to do "Tournament Pivoting"

$$W^{nxb} = \left(\begin{array}{c} W_1 \\ \hline W_2 \\ \hline W_3 \\ \hline W_4 \end{array}\right) = \left(\begin{array}{c} P_1 \cdot L_1 \cdot U_1 \\ \hline P_2 \cdot L_2 \cdot U_2 \\ \hline P_3 \cdot L_3 \cdot U_3 \\ \hline P_4 \cdot L_4 \cdot U_4 \end{array}\right)$$

Choose b pivot rows of W_1, call them W_1'
Choose b pivot rows of W_2, call them W_2'
Choose b pivot rows of W_3, call them W_3'
Choose b pivot rows of W_4, call them W_4'

$$\left(\begin{array}{c} W_1' \\ W_2' \\ \hline W_3' \\ W_4' \end{array}\right) = \left(\begin{array}{c} P_{12} \cdot L_{12} \cdot U_{12} \\ \hline P_{34} \cdot L_{34} \cdot U_{34} \end{array}\right)$$

Choose b pivot rows, call them W_{12}'

Choose b pivot rows, call them W_{34}'

$$\left(\begin{array}{c} W_{12}' \\ W_{34}' \end{array}\right) = P_{1234} \cdot L_{1234} \cdot U_{1234}$$

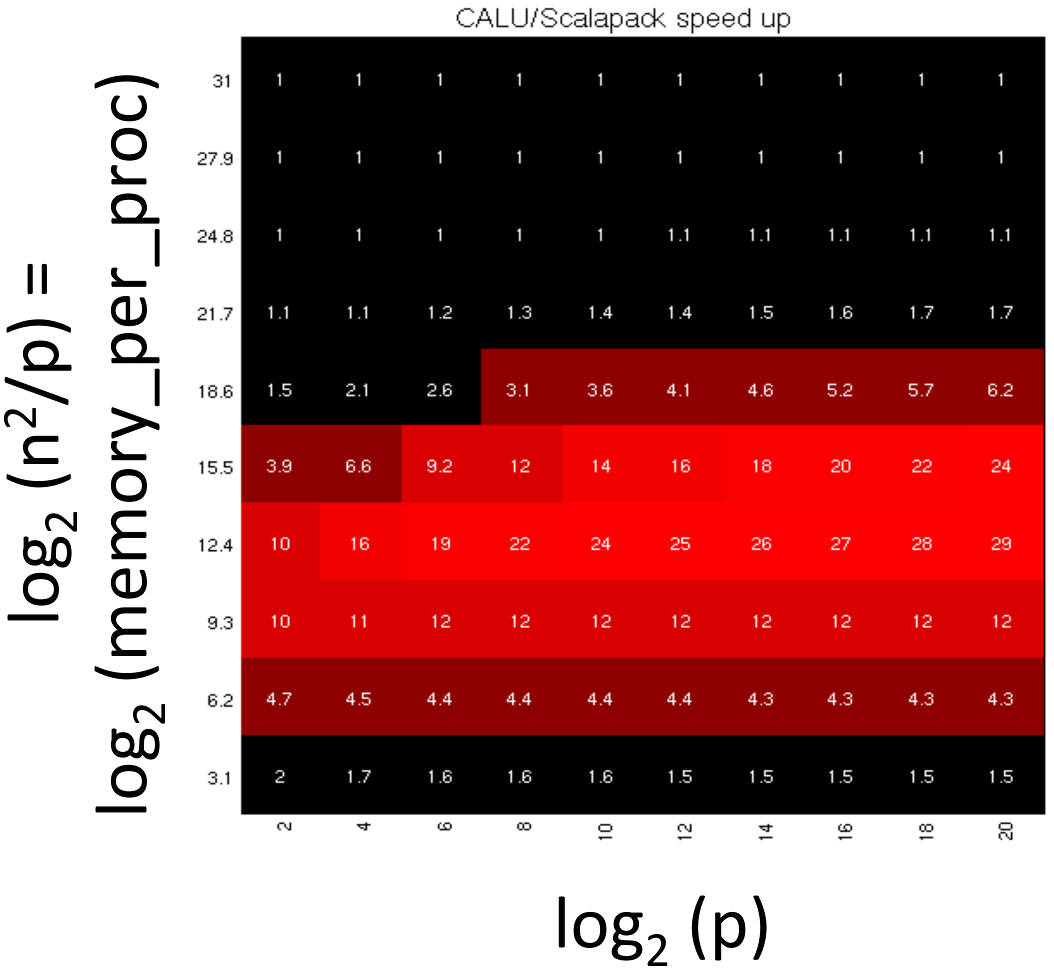
Choose b pivot rows

- Go back to W and use these b pivot rows
  - Move them to top, do LU without pivoting
  - Extra work, but lower order term
- Thm: As numerically stable as Partial Pivoting on a larger matrix

Exascale Machine Parameters Source: DOE Exascale Workshop

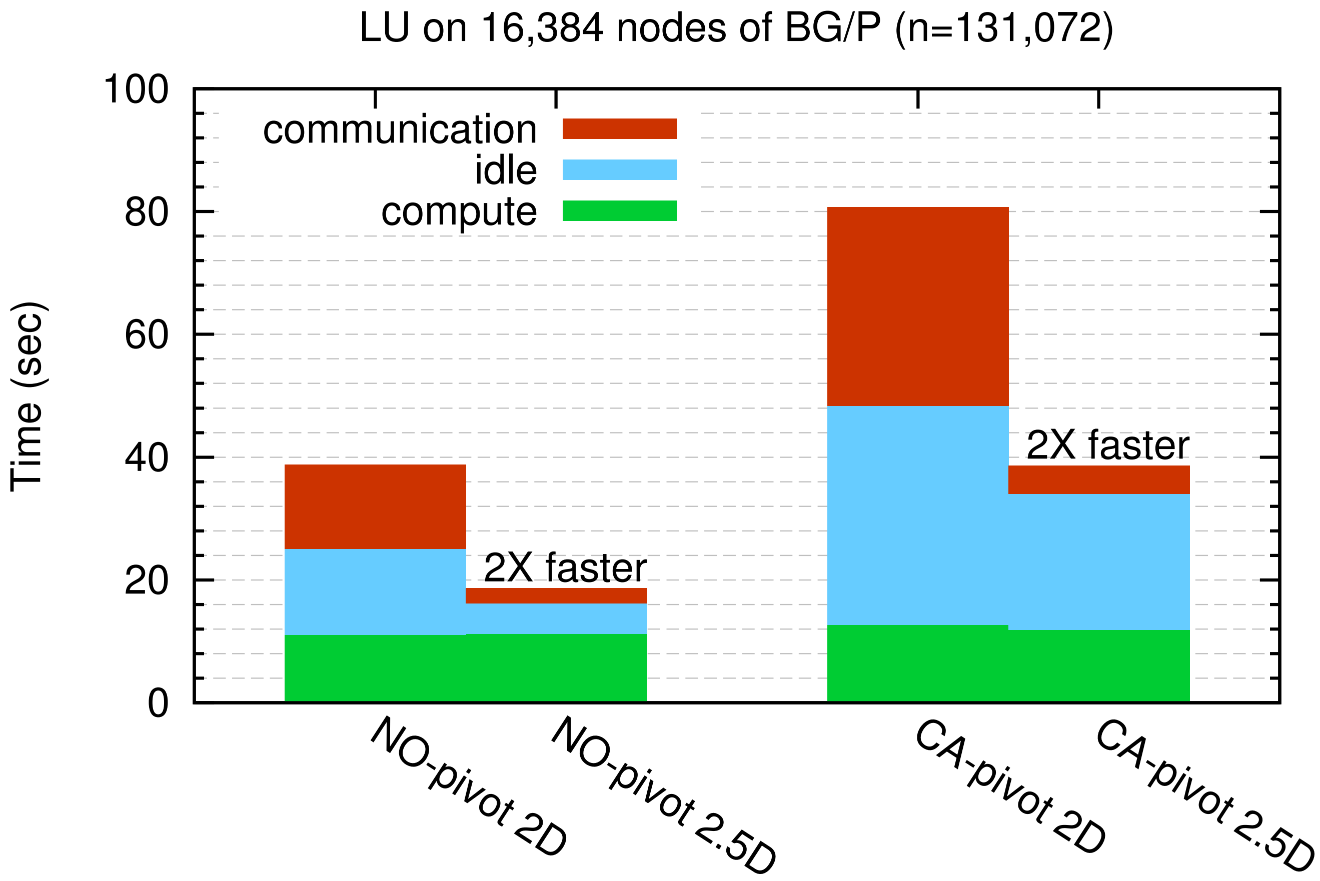
- 2^20 ≈ 1,000,000 nodes
- 1024 cores/node (a billion cores!)
- 100 GB/sec interconnect bandwidth
- 400 GB/sec DRAM bandwidth
- 1 microsec interconnect latency
- 50 nanosec memory latency
- 32 Petabytes of memory
- 1/2 GB total L1 on a node

Exascale predicted speedups for Gaussian Elimination: 2D CA-LU vs ScaLAPACK-LU



	2	4	6	8	10	12	14	16	18	20
31	1	1	1	1	1	1	1	1	1	1
27.9	1	1	1	1	1	1	1	1	1	1
24.8	1	1	1	1	1	1.1	1.1	1.1	1.1	1.1
21.7	1.1	1.1	1.2	1.3	1.4	1.4	1.5	1.6	1.7	1.7
18.6	1.5	2.1	2.6	3.1	3.6	4.1	4.6	5.2	5.7	6.2
15.5	3.9	6.6	9.2	12	14	16	18	20	22	24
12.4	10	16	19	22	24	25	26	27	28	29
9.3	10	11	12	12	12	12	12	12	12	12
6.2	4.7	4.5	4.4	4.4	4.4	4.4	4.3	4.3	4.3	4.3
3.1	2	1.7	1.6	1.6	1.6	1.5	1.5	1.5	1.5	1.5

2.5D vs 2D LU
With and Without Pivoting



Summary of dense *sequential* algorithms attaining communication lower bounds

- Algorithms shown minimizing # Messages use (recursive) block layout
  - Not possible with columnwise or rowwise layouts
- *Many* references (see reports), only some shown, plus ours
- Cache-oblivious are underlined, Green are ours, ? is unknown/future work

Algorithm	2 Levels of Memory		Multiple Levels of Memory	
	#Words Moved	and # Messages	#Words Moved	and #Messages
BLAS-3				
Cholesky				
LU with pivoting				
QR Rank-revealing				
Eig, SVD				

Summary of dense *parallel* algorithms attaining communication lower bounds

- Assume nxn matrices on P processors
- Minimum Memory per processor = $M = O(n^2 / P)$
- Recall lower bounds:

  #words_moved = $\Omega((n^3/ P) / M^{1/2}) = \Omega(n^2 / P^{1/2})$

  #messages = $\Omega((n^3/ P) / M^{3/2}) = \Omega(P^{1/2})$

Algorithm	**Reference**	**Factor exceeding lower bound for #words_moved**	**Factor exceeding lower bound for #messages**
Matrix Multiply			
Cholesky			
LU			
QR			
Sym Eig, SVD			
Nonsym Eig			

Summary of dense *parallel* algorithms attaining communication lower bounds

- Assume nxn matrices on P processors (conventional approach)
- Minimum Memory per processor = $M = O(n^2 / P)$
- Recall lower bounds:

  #words_moved = $\Omega((n^3/ P) / M^{1/2}) = \Omega(n^2 / P^{1/2})$

  #messages = $\Omega((n^3/ P) / M^{3/2}) = \Omega(P^{1/2})$

Algorithm	**Reference**	**Factor exceeding lower bound for #words_moved**	**Factor exceeding lower bound for #messages**
Matrix Multiply	**[Cannon, 69]**	**1**	
Cholesky	**ScaLAPACK**	**log P**	
LU	**ScaLAPACK**	**log P**	
QR	**ScaLAPACK**	**log P**	
Sym Eig, SVD	**ScaLAPACK**	**log P**	
Nonsym Eig	**ScaLAPACK**	**$P^{1/2}$ log P**	

Summary of dense *parallel* algorithms attaining communication lower bounds

- Assume nxn matrices on P processors (conventional approach)
- Minimum Memory per processor = $M = O(n^2 / P)$
- Recall lower bounds:

  #words_moved = $\Omega((n^3/ P) / M^{1/2}) = \Omega(n^2 / P^{1/2})$

  #messages = $\Omega((n^3/ P) / M^{3/2}) = \Omega(P^{1/2})$

Algorithm	**Reference**	**Factor exceeding lower bound for #words_moved**	**Factor exceeding lower bound for #messages**
Matrix Multiply	**[Cannon, 69]**	**1**	**1**
Cholesky	**ScaLAPACK**	**log P**	**log P**
LU	**ScaLAPACK**	**log P**	**n log P / $P^{1/2}$**
QR	**ScaLAPACK**	**log P**	**n log P / $P^{1/2}$**
Sym Eig, SVD	**ScaLAPACK**	**log P**	**n / $P^{1/2}$**
Nonsym Eig	**ScaLAPACK**	**$P^{1/2}$ log P**	**n log P**

Summary of dense *parallel* algorithms attaining communication lower bounds

- Assume nxn matrices on P processors (better)
- Minimum Memory per processor = $M = O(n^2 / P)$
- Recall lower bounds:

$$\#\text{words\_moved} = \Omega((n^3/ P) / M^{1/2}) = \Omega(n^2 / P^{1/2})$$

$$\#\text{messages} = \Omega((n^3/ P) / M^{3/2}) = \Omega(P^{1/2})$$

Algorithm	Reference	Factor exceeding lower bound for #words_moved	Factor exceeding lower bound for #messages
Matrix Multiply	**[Cannon, 69]**	**1**	**1**
Cholesky	**ScaLAPACK**	**log P**	**log P**
LU	**[GDX10]**	**log P**	**log P**
QR	**[DGHL08]**	**log P**	**$\log^3 P$**
Sym Eig, SVD	**[BDD11]**	**log P**	**$\log^3 P$**
Nonsym Eig	**[BDD11]**	**log P**	**$\log^3 P$**

Can we do even better?

- Assume nxn matrices on P processors
- Use c copies of data: $M = O(cn^2 / P)$ per processor
- Increasing M reduces lower bounds:

$$\#\text{words\_moved} = \Omega((n^3/ P) / M^{1/2}) = \Omega(n^2 / (c^{1/2} P^{1/2}))$$

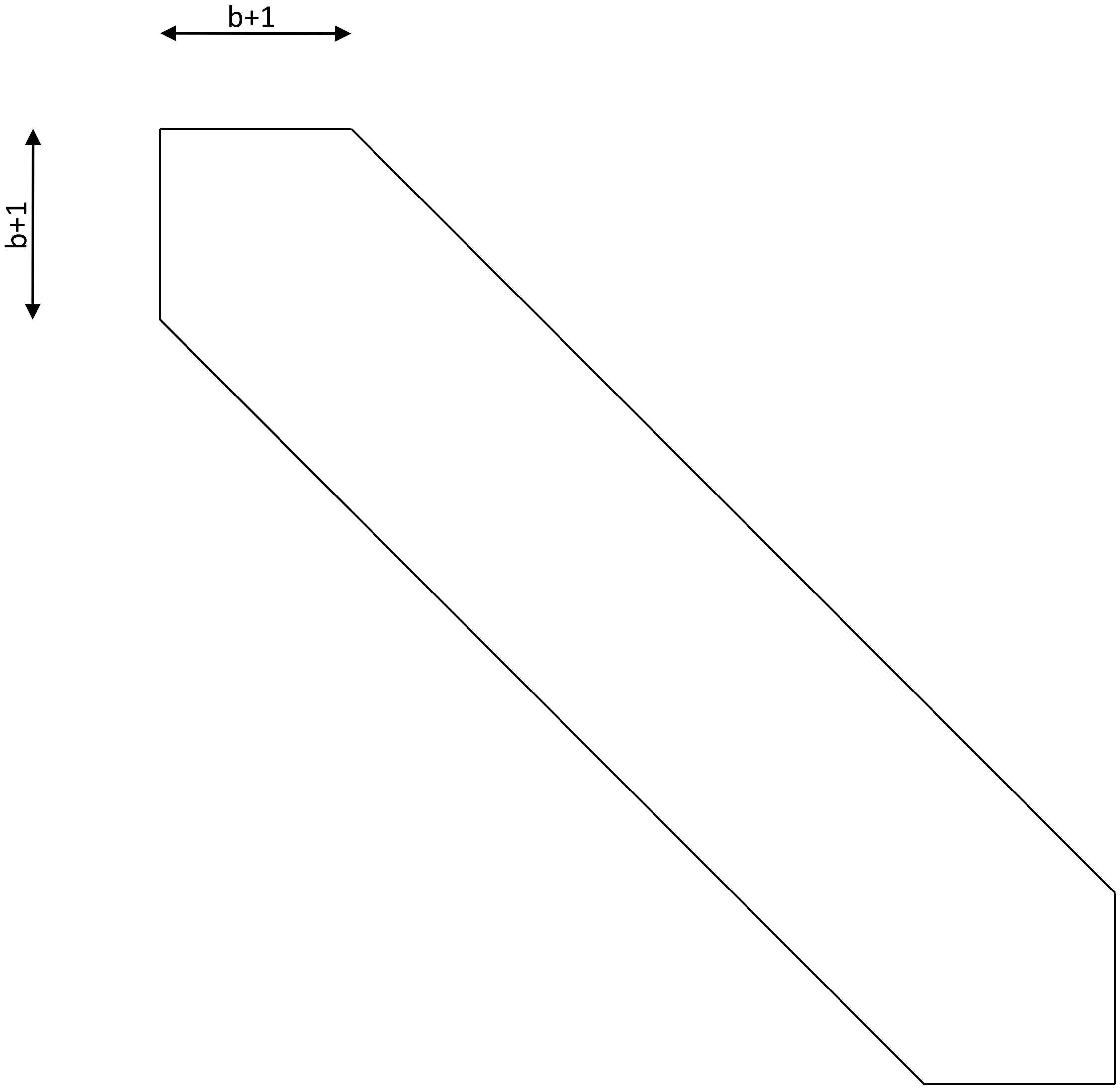
$$\#\text{messages} = \Omega((n^3/ P) / M^{3/2}) = \Omega(P^{1/2} / c^{3/2})$$

Algorithm	Reference	Factor exceeding lower bound for #words_moved	Factor exceeding lower bound for #messages
Matrix Multiply	[DS11,SBD11]	polylog P	polylog P
Cholesky	[SD11, in prog.]	polylog P	c^2 polylog P – optimal!
LU	[DS11,SBD11]	polylog P	c^2 polylog P – optimal!
QR	Via Cholesky QR	polylog P	c^2 polylog P – optimal!
Sym Eig, SVD	?		
Nonsym Eig	?		

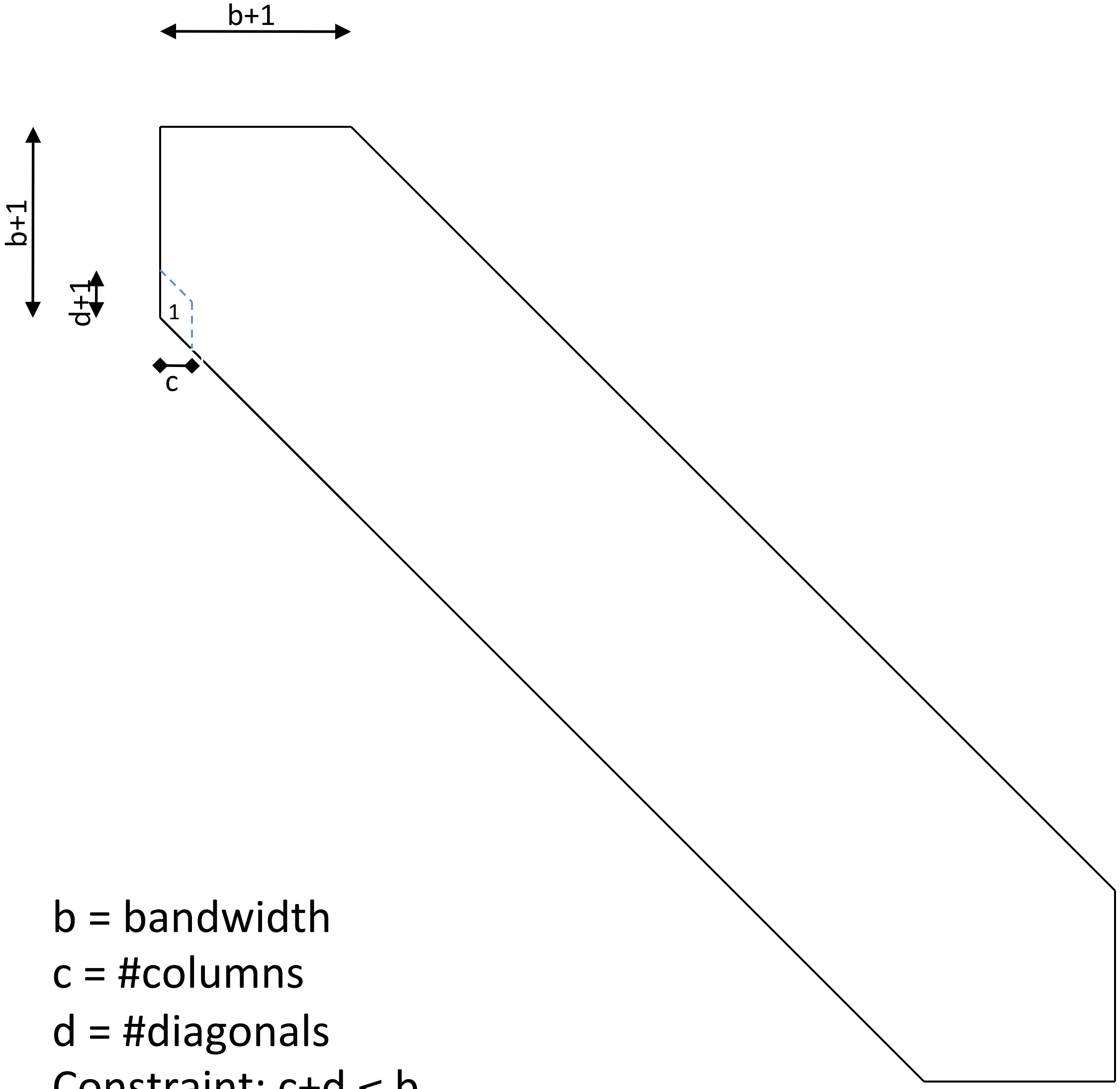
Symmetric Band Reduction

- Grey Ballard and Nick Knight
- A $\Rightarrow$ $QAQ^T = T$, where
  - $A=A^T$ is banded
  - T tridiagonal
  - Similar idea for SVD of a band matrix
- Use alone, or as second phase when A is dense:
  - Dense $\Rightarrow$ Banded $\Rightarrow$ Tridiagonal
- Implemented in LAPACK's sytrd
- Algorithm does not satisfy communication lower bound theorem for applying orthogonal transformations
  - It can communicate even less!

Successive Band Reduction

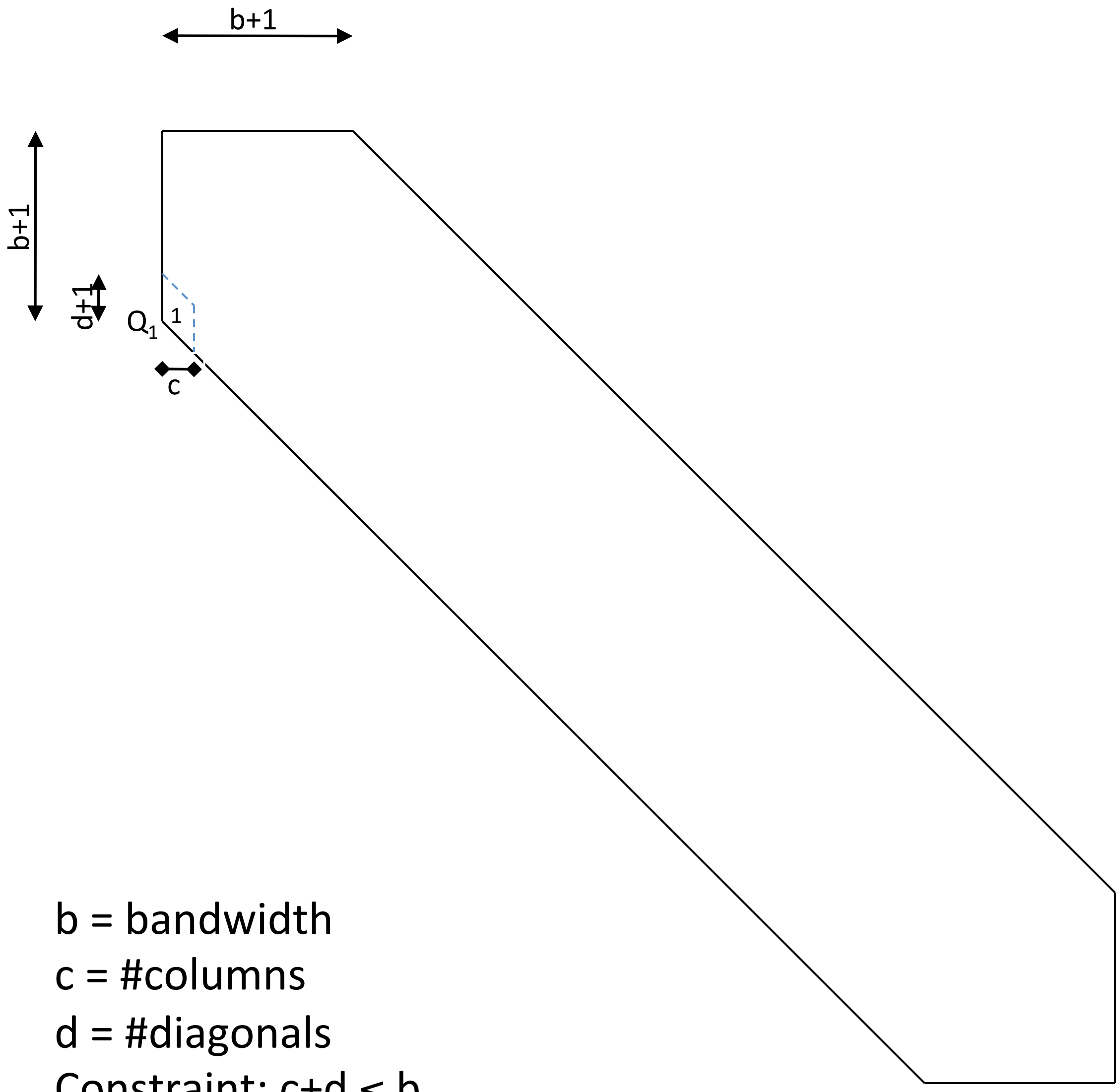


Successive Band Reduction



b = bandwidth
c = #columns
d = #diagonals
Constraint: c+d ≤ b

Successive Band Reduction



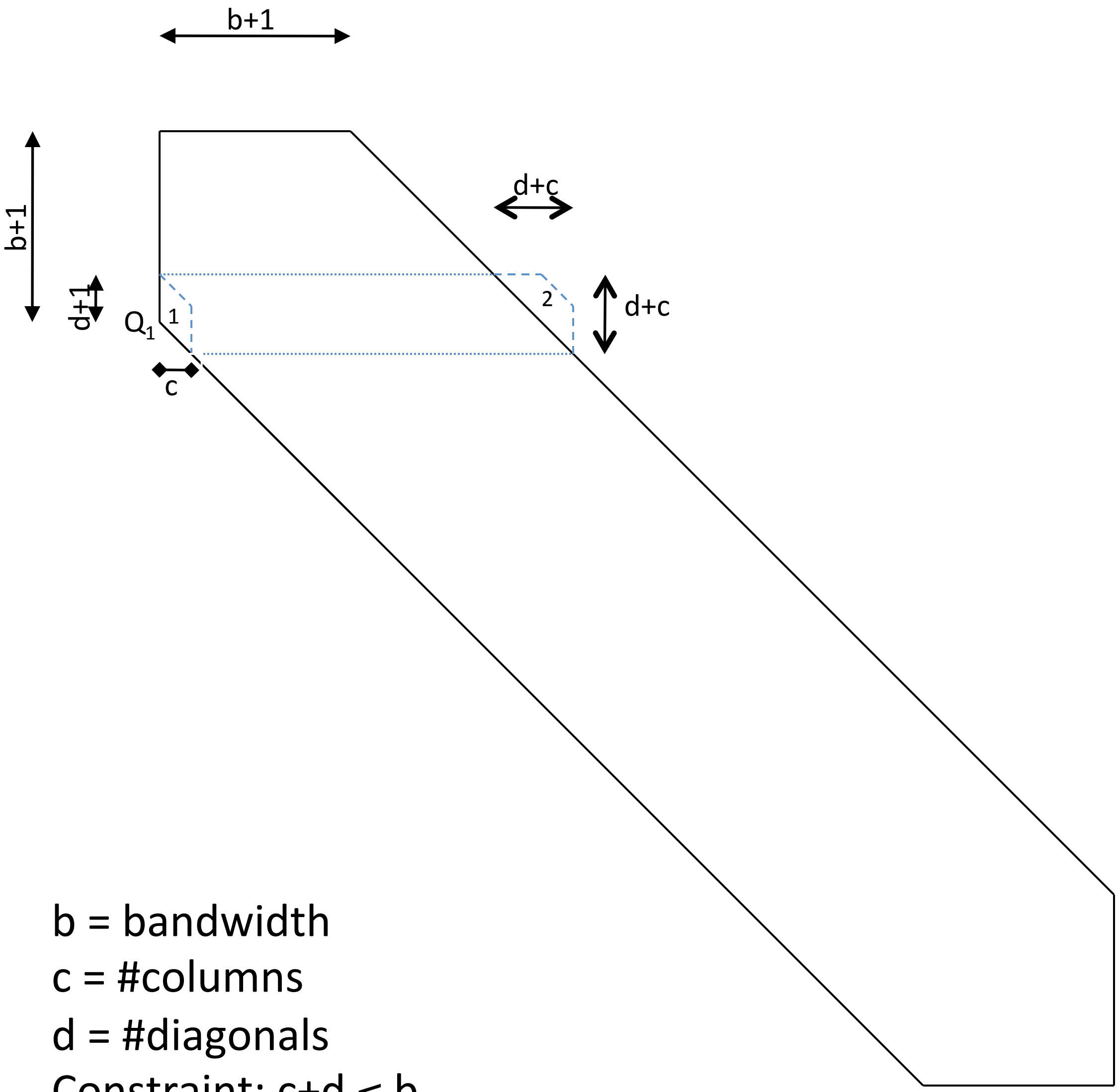
b = bandwidth

c = #columns

d = #diagonals

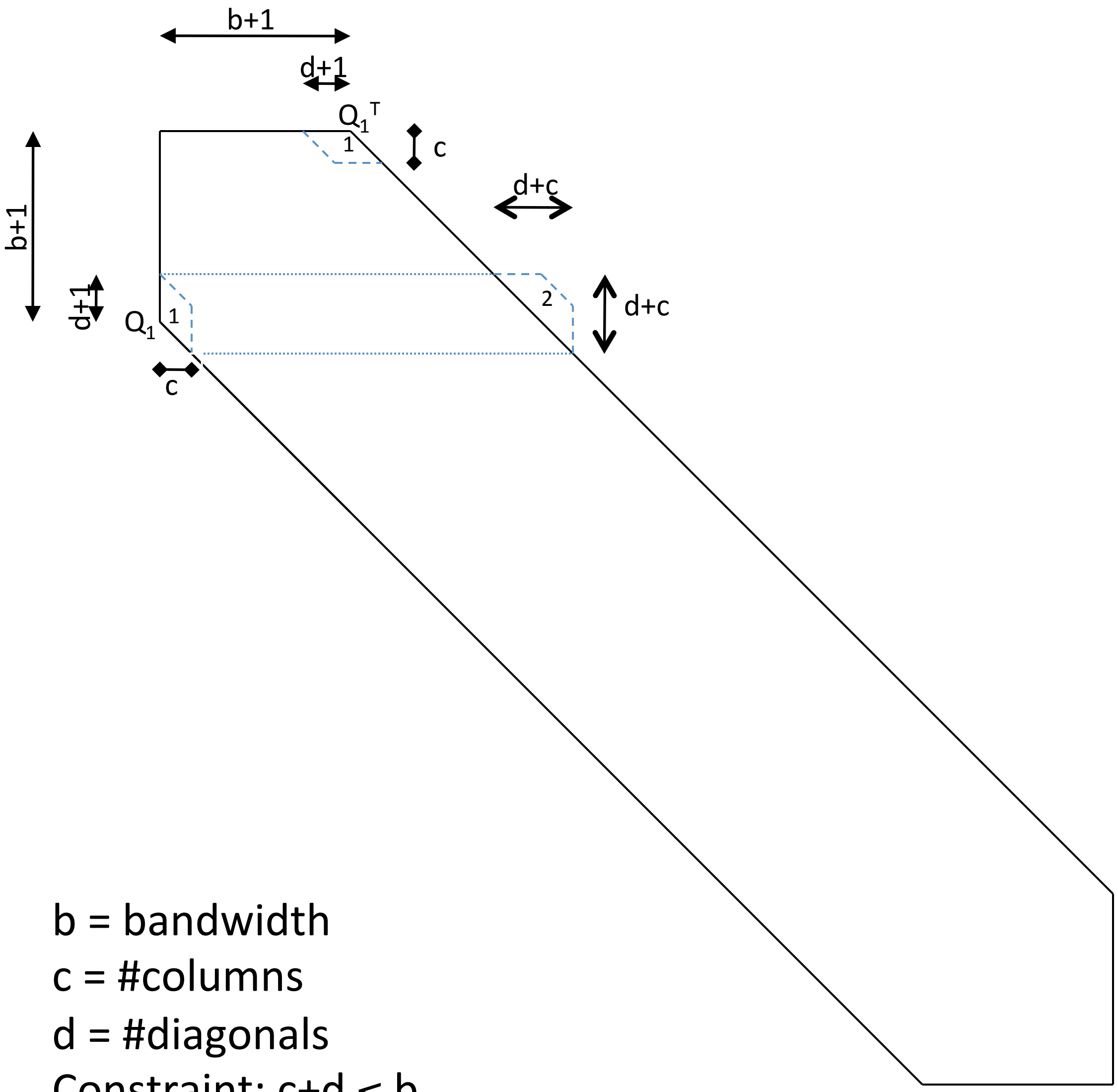
Constraint: c+d ≤ b

Successive Band Reduction



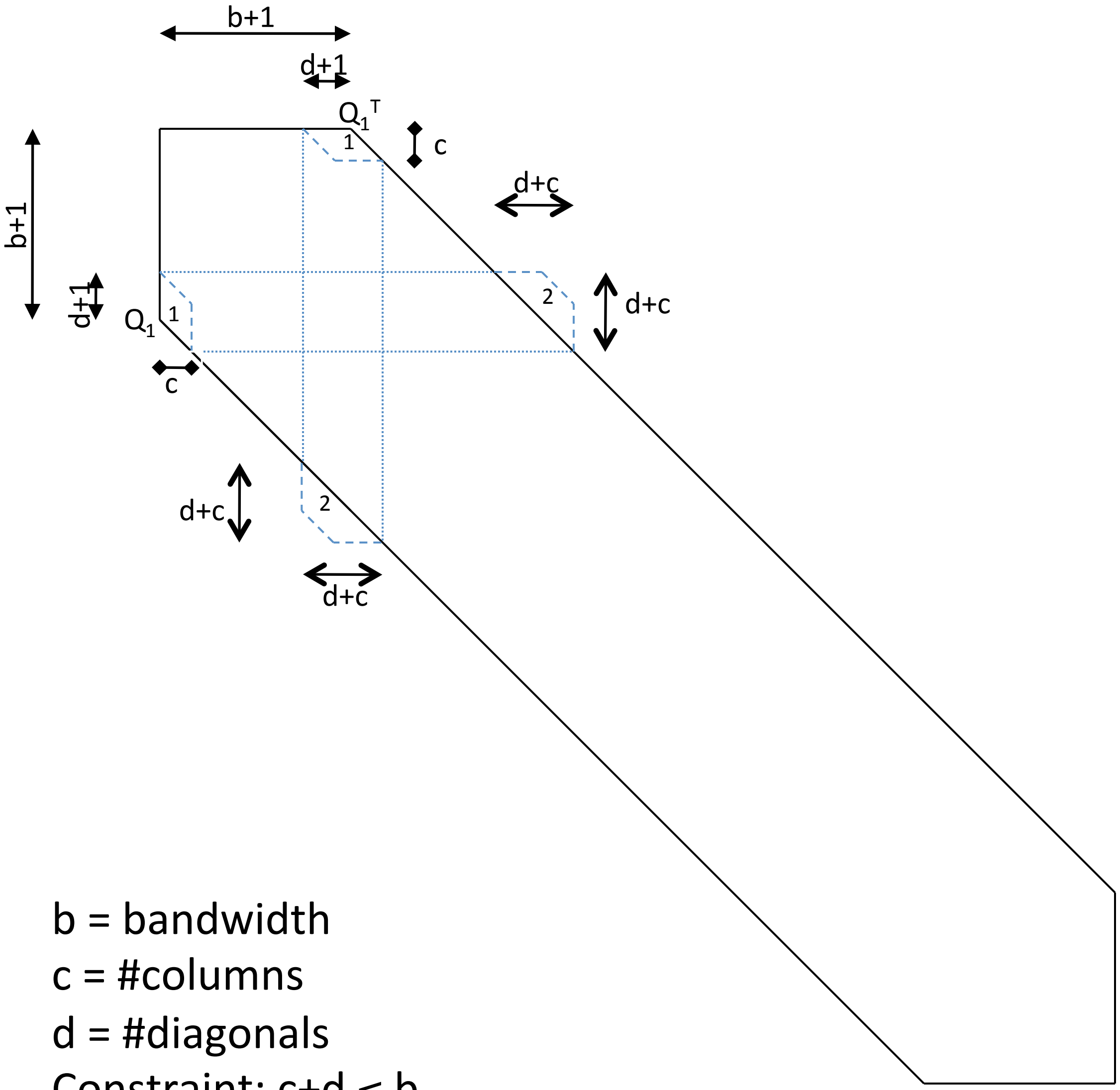
b = bandwidth
c = #columns
d = #diagonals
Constraint: c+d ≤ b

Successive Band Reduction



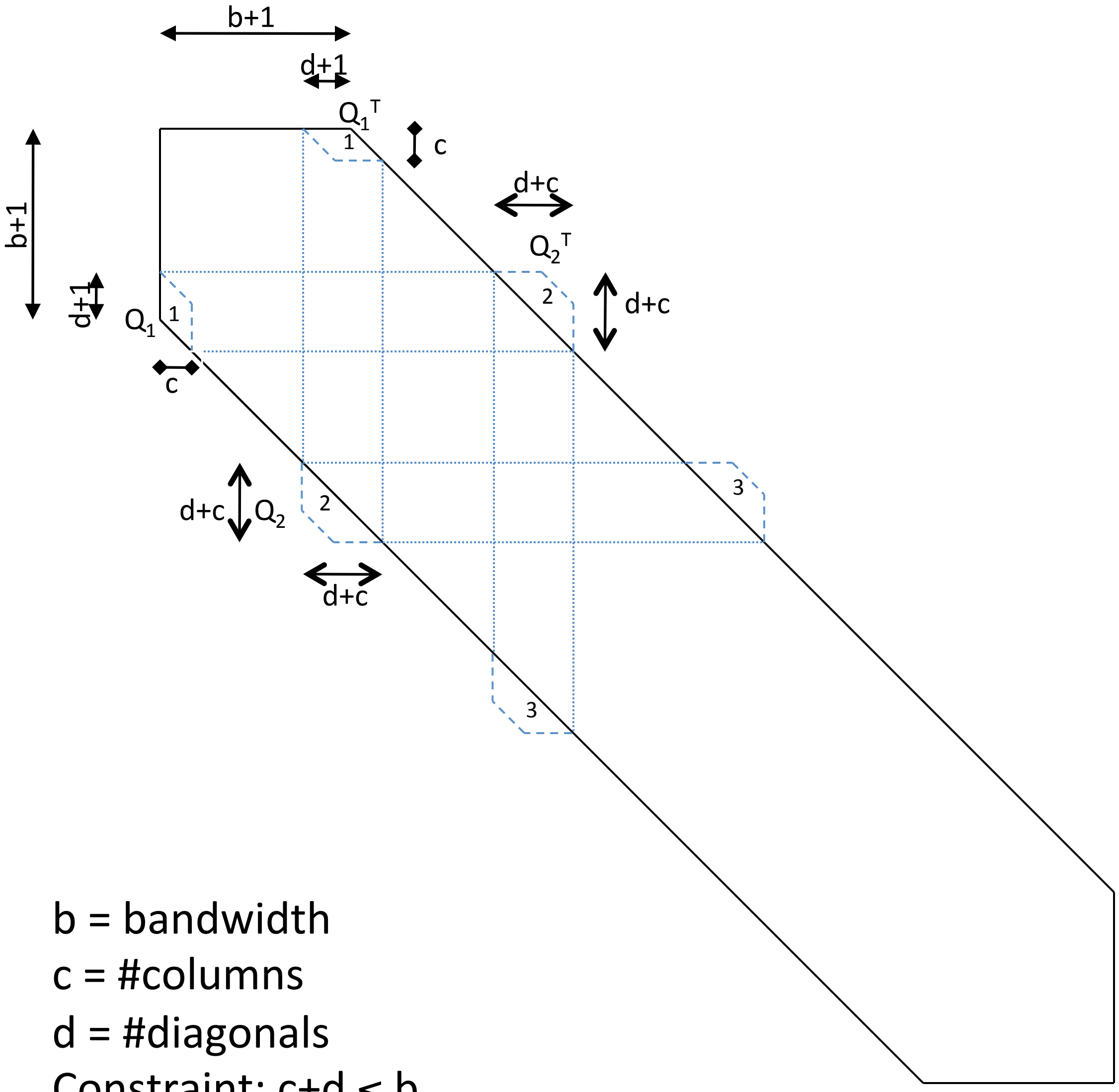
b = bandwidth
c = #columns
d = #diagonals
Constraint: c+d ≤ b

Successive Band Reduction



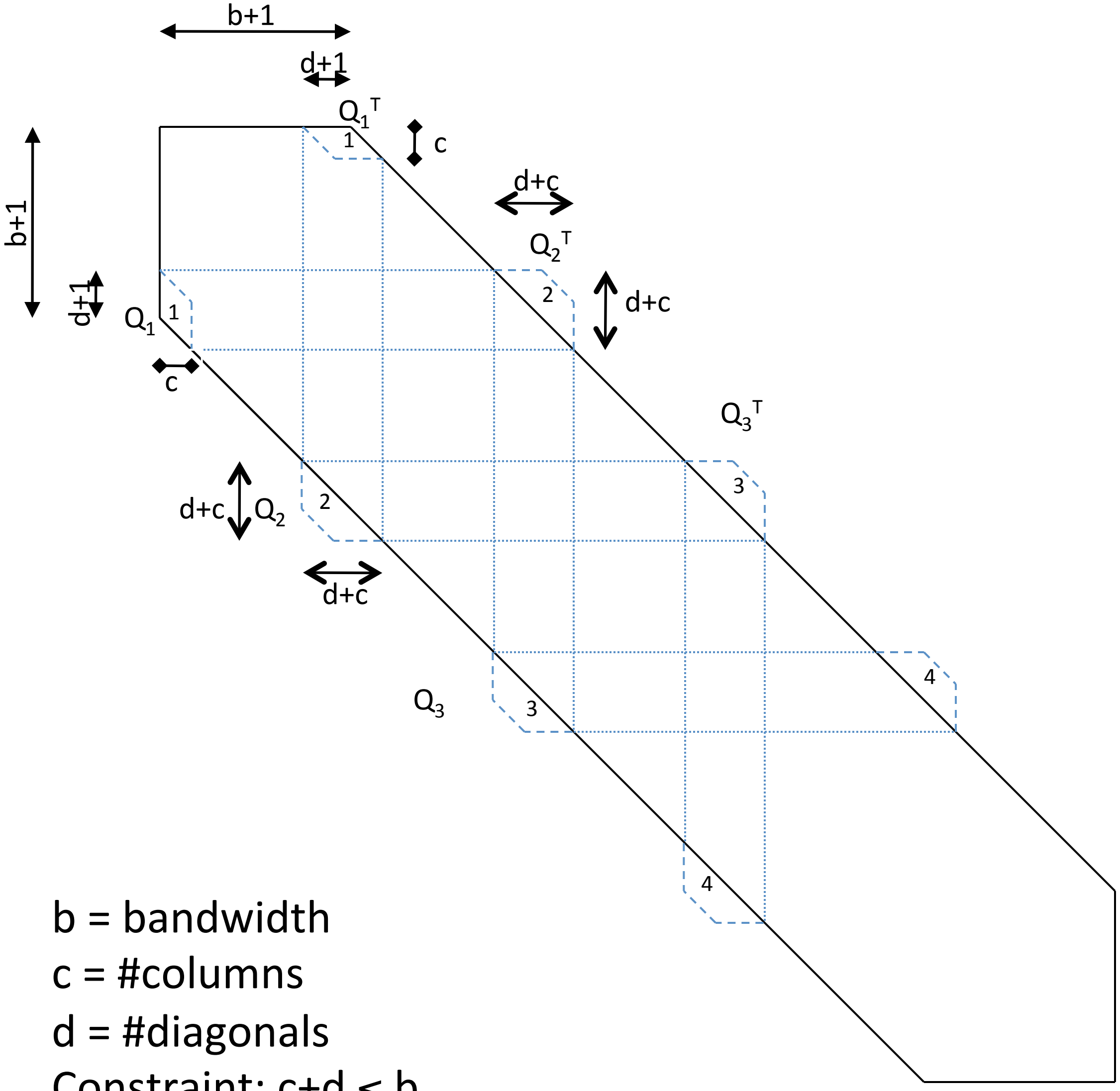
b = bandwidth
c = #columns
d = #diagonals
Constraint: c+d ≤ b

Successive Band Reduction



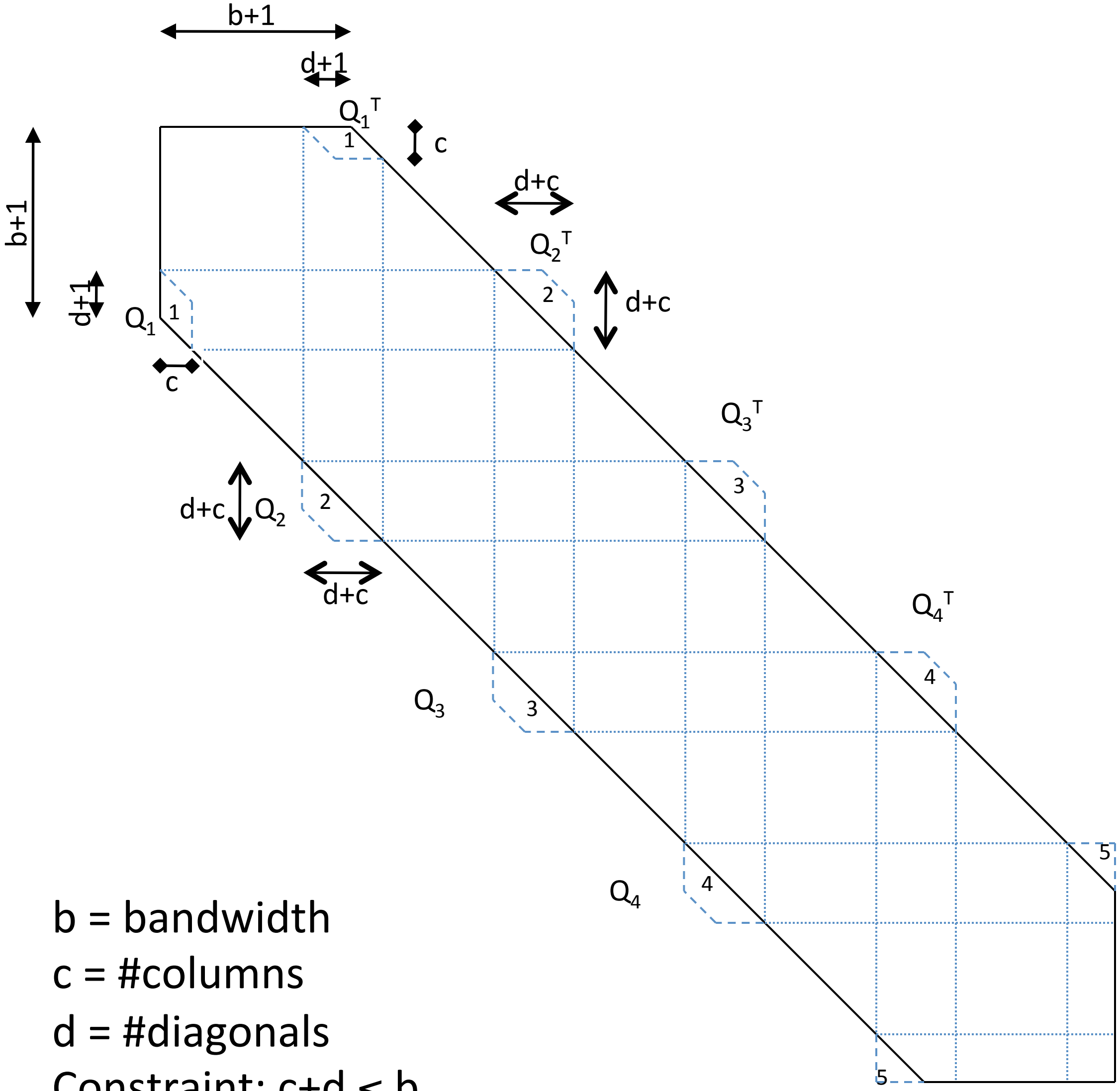
b = bandwidth
c = #columns
d = #diagonals
Constraint: c+d ≤ b

Successive Band Reduction



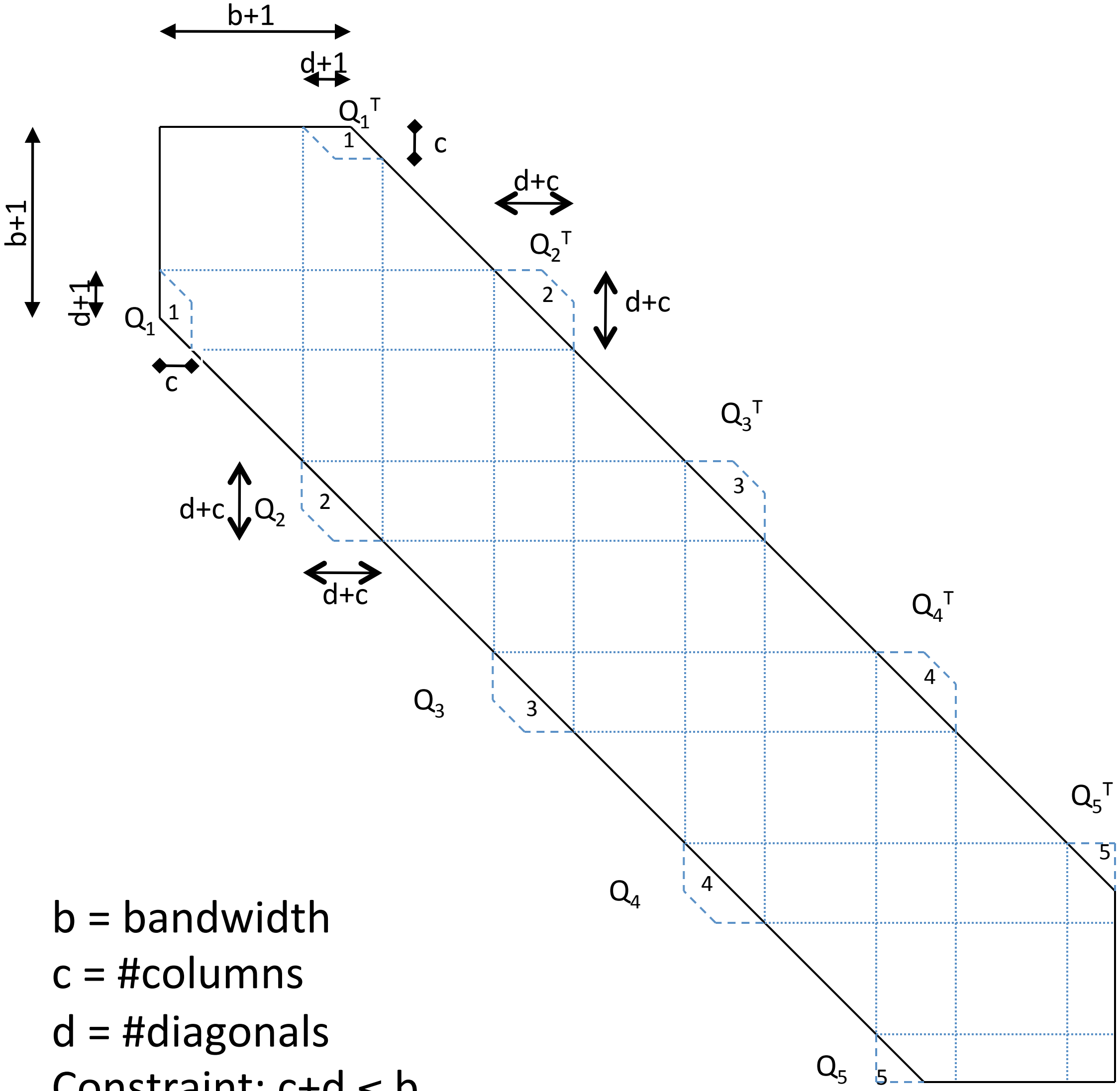
b = bandwidth
c = #columns
d = #diagonals
Constraint: c+d ≤ b

Successive Band Reduction



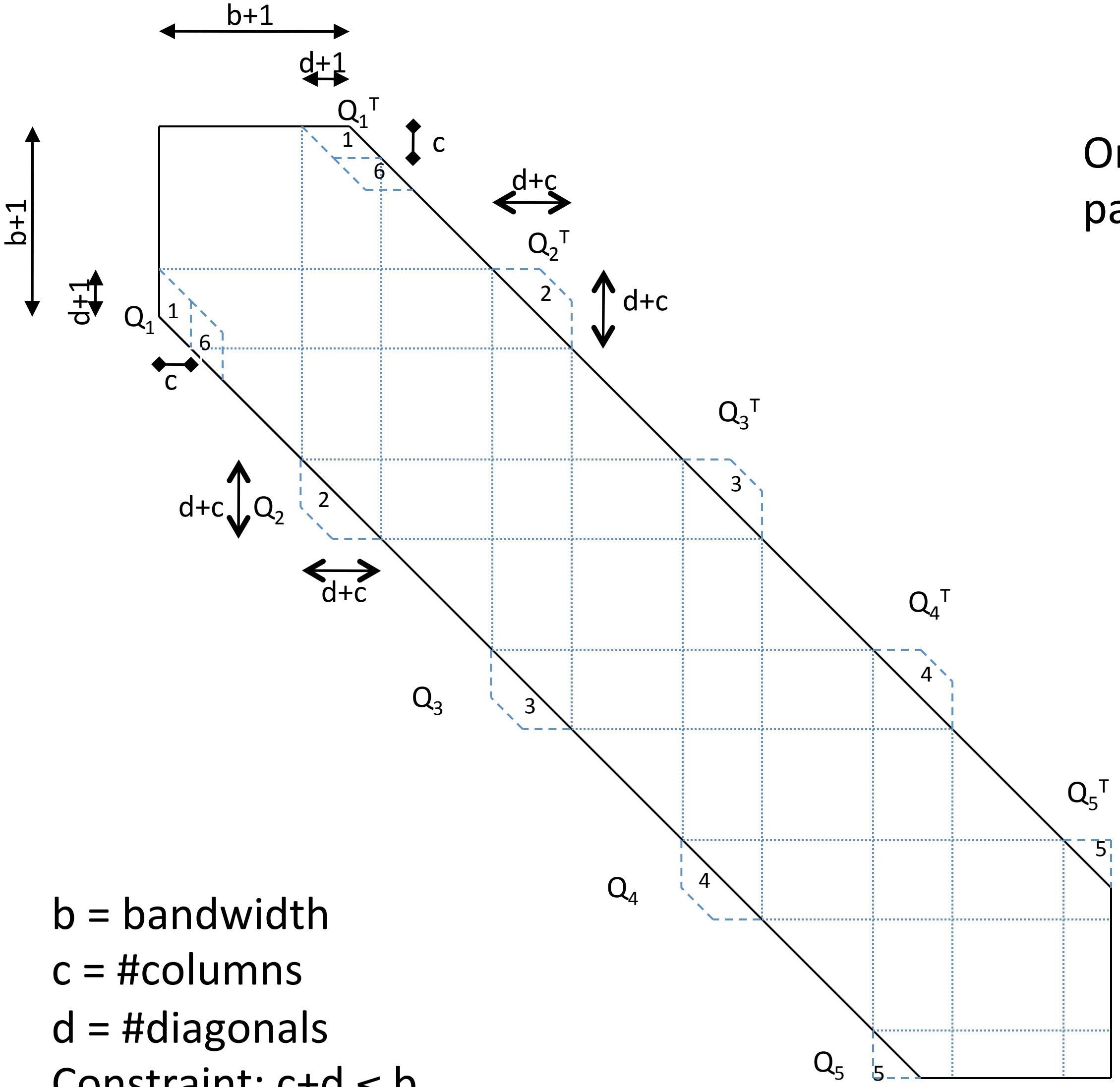
b = bandwidth
c = #columns
d = #diagonals
Constraint: c+d ≤ b

Successive Band Reduction

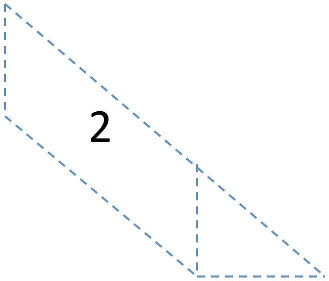


b = bandwidth
c = #columns
d = #diagonals
Constraint: $c+d \leq b$

Successive Band Reduction



Only need to zero out leading parallelogram of each trapezoid:



b = bandwidth

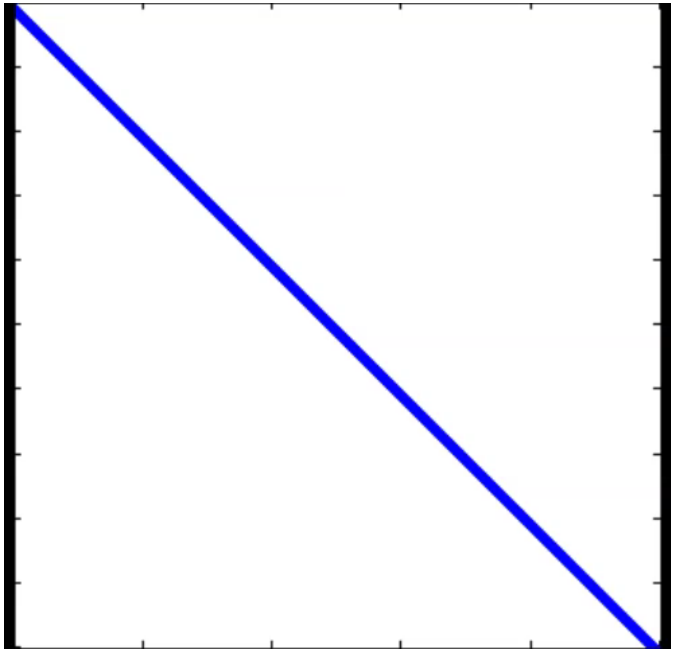
c = #columns

d = #diagonals

Constraint: c+d ≤ b

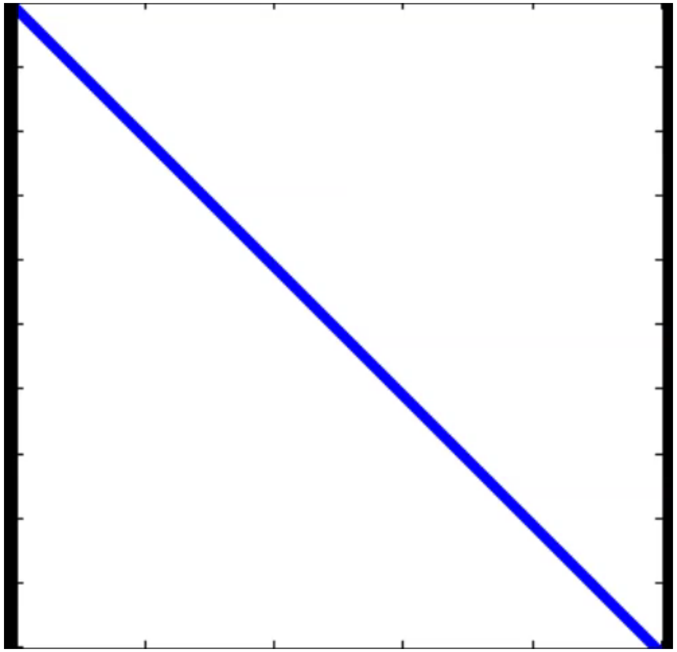
Conventional vs CA - SBR

Touch all data 4 times



Conventional

Touch all data once



Communication-Avoiding

Many tuning parameters:

Number of “sweeps”, #diagonals cleared per sweep, sizes of parallelograms

#bulges chased at one time, how far to chase each bulge

Right choices reduce #words_moved by factor M/bw, not just $M^{1/2}$

Speedups of Sym. Band Reduction vs DSBTRD

- Up to **17x** on Intel Gainestown, vs MKL 10.0
  - n=12000, b=500, 8 threads
- Up to **12x** on Intel Westmere, vs MKL 10.3
  - n=12000, b=200, 10 threads
- Up to **25x** on AMD Budapest, vs ACML 4.4
  - n=9000, b=500, 4 threads
- Up to **30x** on AMD Magny-Cours, vs ACML 4.4
  - n=12000, b=500, 6 threads

- Neither MKL nor ACML benefits from multithreading in DSBTRD
  - Best sequential speedup vs MKL: **1.9x**
  - Best sequential speedup vs ACML: **8.5x**

Communication Lower Bounds for Strassen-like matmul algorithms

Classical
$O(n^{3})$ matmul:

#words_moved =
$\Omega\ (M(n/M^{1/2})^{3}/P)$

Strassen's
$O(n^{\lg 7})$ matmul:

#words_moved =
$\Omega\ (M(n/M^{1/2})^{\lg 7}/P)$

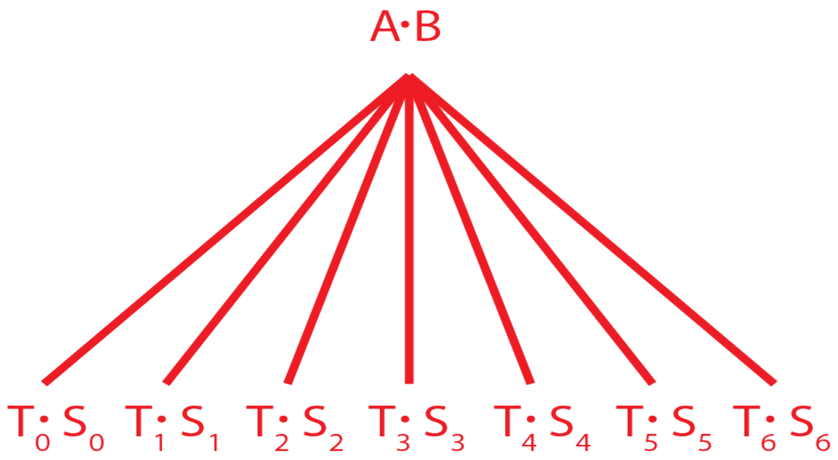
Strassen-like
$O(n^{\omega})$ matmul:

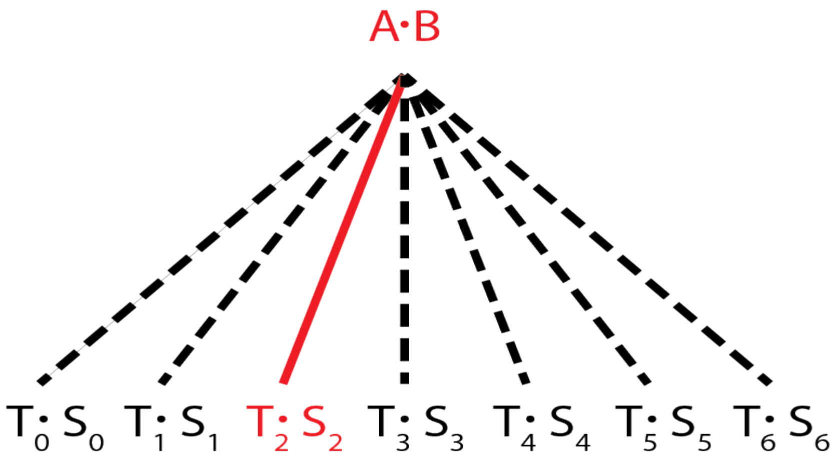
#words_moved =
$\Omega\ (M(n/M^{1/2})^{\omega}/P)$

- Proof: graph expansion (different from classical matmul)
  - Strassen-like: DAG must be "regular" and connected
- Extends up to $M = n^2 / p^{2/\omega}$
- Best Paper Prize (SPAA'11), Ballard, D., Holtz, Schwartz
  to appear in JACM
- Is the lower bound attainable?

Communication Avoiding Parallel Strassen *(CAPS)*

BFS vs. DFS





Runs all 7 multiplies in parallel
Each on P/7 processors
Needs 7/4 as much memory

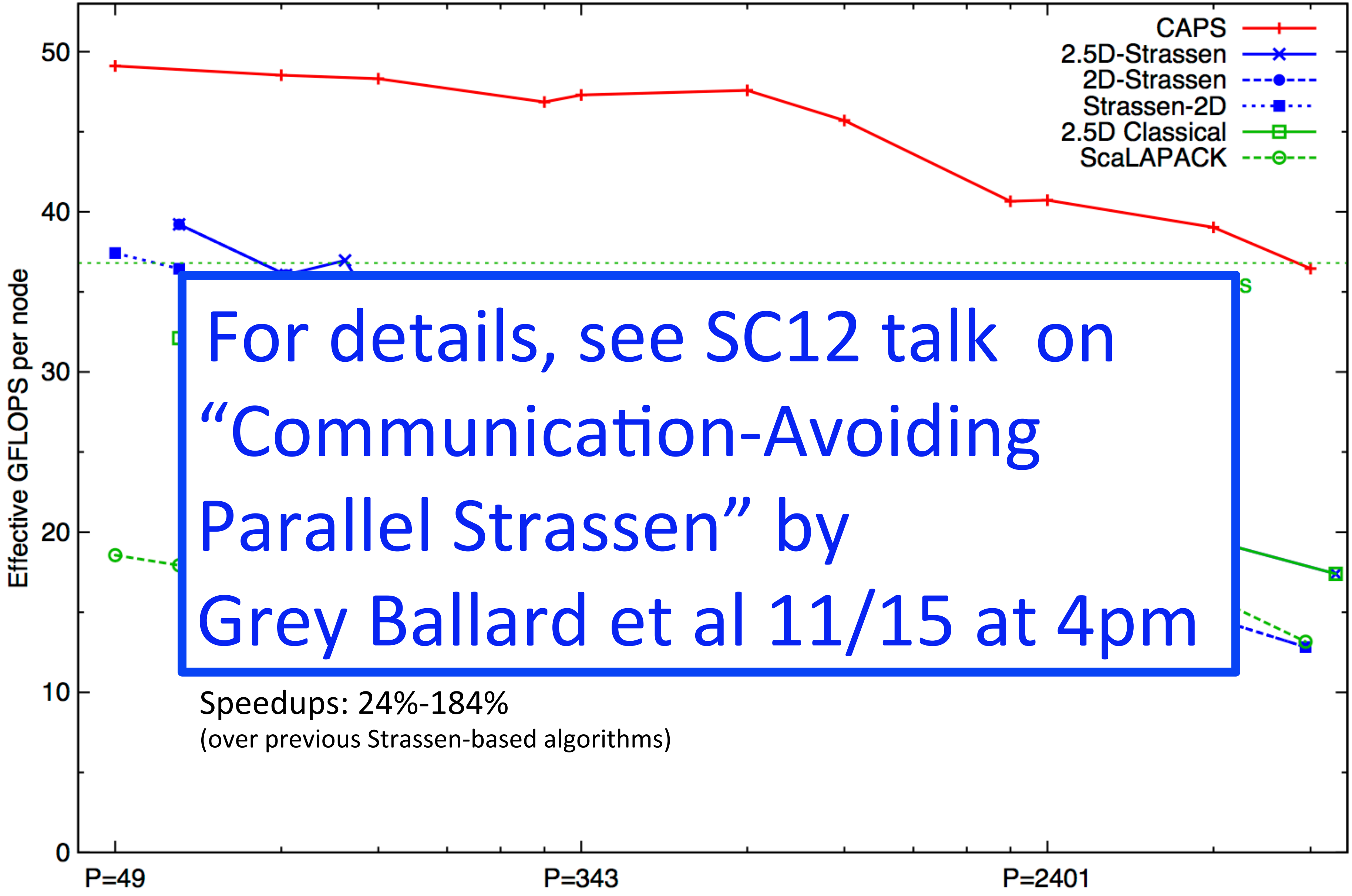
Runs all 7 multiplies sequentially
Each on all P processors
Needs 1/4 as much memory

```
CAPS
  If EnoughMemory and P ≥ 7
    then BFS step
    else DFS step
  end if
```

In practice, how to best interleave BFS and DFS is a “tuning parameter”

Performance Benchmarking, Strong Scaling Plot

Franklin (Cray XT4) n = 94080



For details, see SC12 talk on “Communication-Avoiding Parallel Strassen” by Grey Ballard et al 11/15 at 4pm

Speedups: 24%-184%

(over previous Strassen-based algorithms)

Summary of Direct Linear Algebra

- New lower bounds, optimal algorithms, big speedups in theory and practice
- Lots of ongoing work on
  - Algorithms:
    - LDL^T, QR with pivoting, other pivoting schemes, eigenproblems, …
    - All-pairs-shortest-path, …
    - Both 2D (c=1) and 2.5D (c>1)
    - But only bandwidth may decrease with c>1, not latency
    - Sparse matrices
  - Platforms:
    - Multicore, cluster, GPU, cloud, heterogeneous, low-energy, …
  - Software:
    - Integration into Sca/LAPACK, PLASMA, MAGMA,…
- Integration into applications (on IBM BG/Q)
  - Qbox (with LLNL, IBM): molecular dynamics
  - CTF (with ANL): symmetric tensor contractions

Outline

- “Direct” Linear Algebra
    - Lower bounds on communication
    - New algorithms that attain these lower bounds
- Ditto for “Iterative” Linear Algebra
- Ditto (work in progress) for programs accessing arrays (eg n-body)

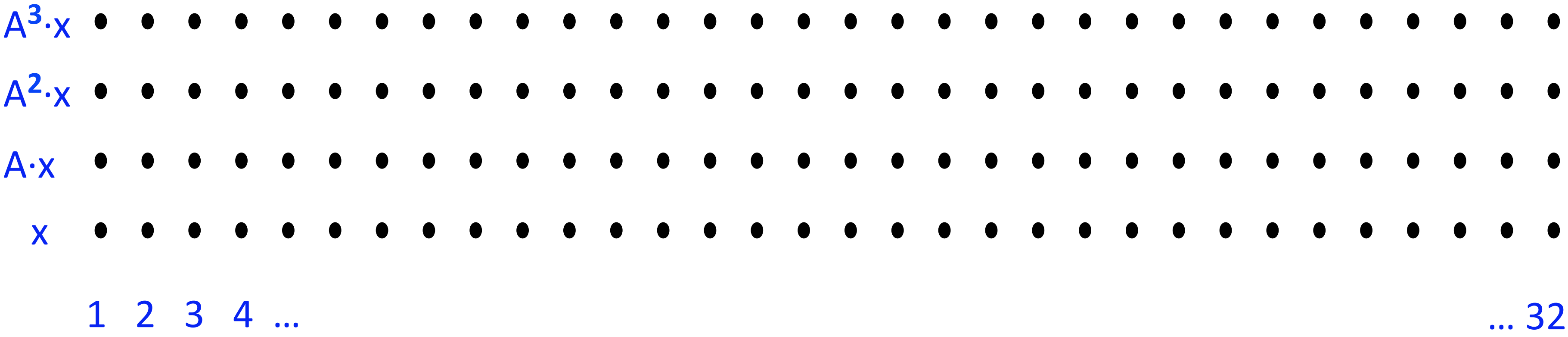
Avoiding Communication in Iterative Linear Algebra

- k-steps of iterative solver for sparse Ax=b or Ax=λx
  - Does k SpMVs with A and starting vector
  - Many such "Krylov Subspace Methods"
- Goal: minimize communication
  - Assume matrix "well-partitioned"
  - Serial implementation
    - Conventional: O(k) moves of data from slow to fast memory
    - **New**: **O(1) moves of data – optimal**
  - Parallel implementation on p processors
    - Conventional: O(k log p) messages (k SpMV calls, dot prods)
    - **New: O(log p) messages - optimal**
- Lots of speed up possible (modeled and measured)
  - Price: some redundant computation

Communication Avoiding Kernels:

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$

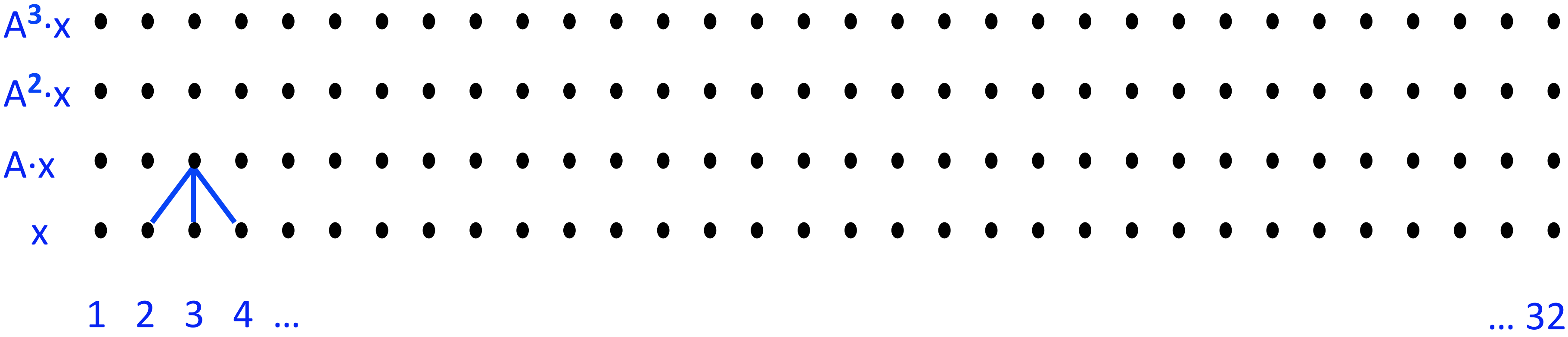


- Example: A tridiagonal, n=32, k=3
- Works for any “well-partitioned” A

Communication Avoiding Kernels:

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

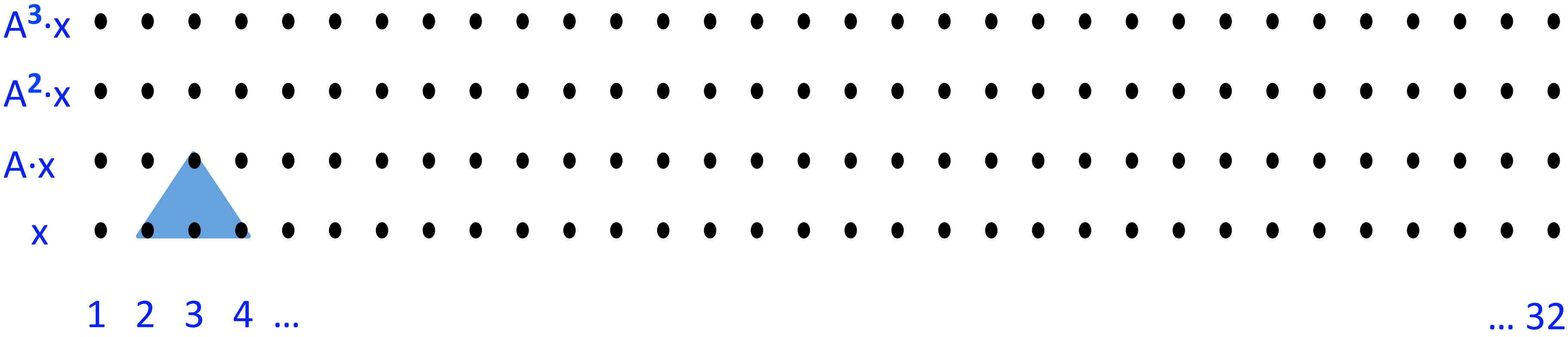
- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$



- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^{2}x, …, A^{k}x]

- Replace k iterations of y = A·x with [Ax, A^{2}x, …, A^{k}x]

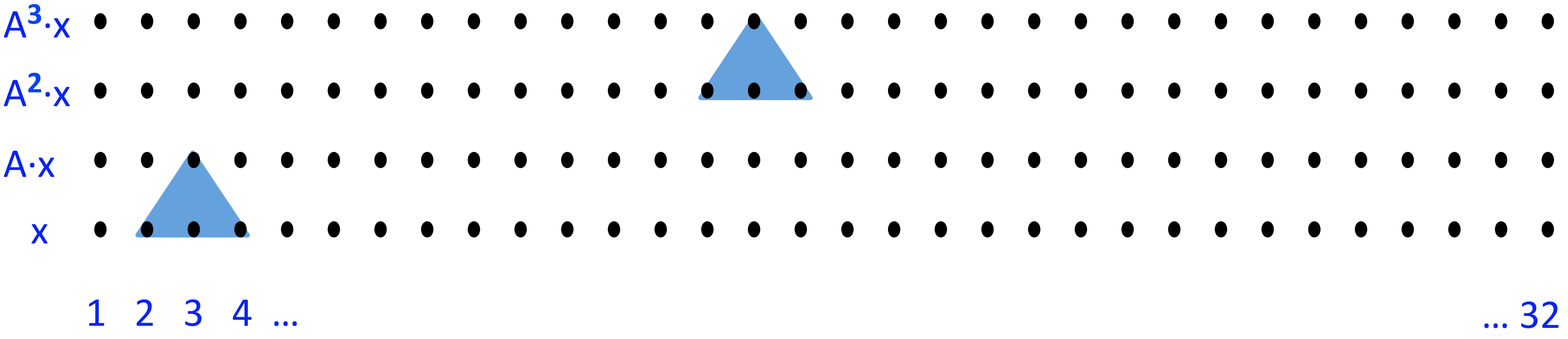


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

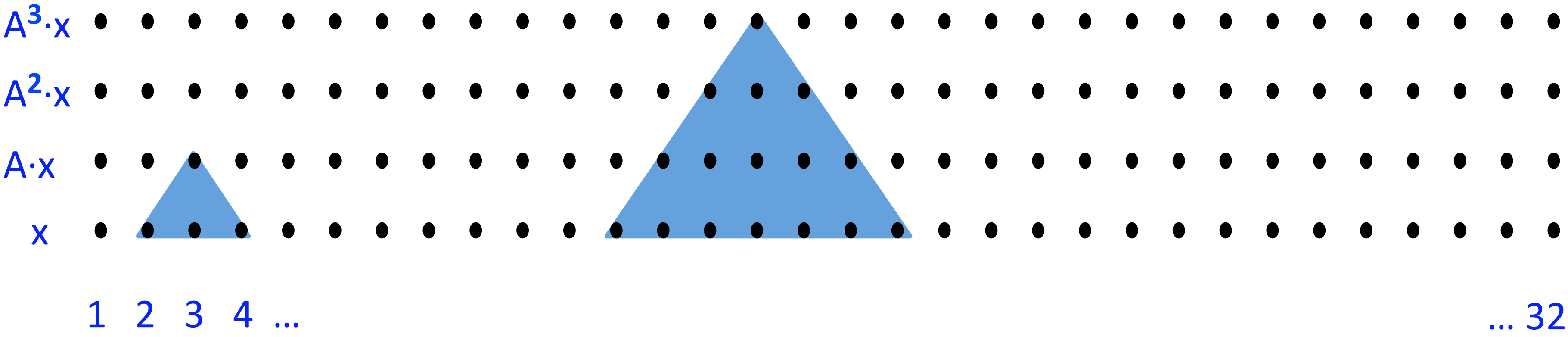
- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$



- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$

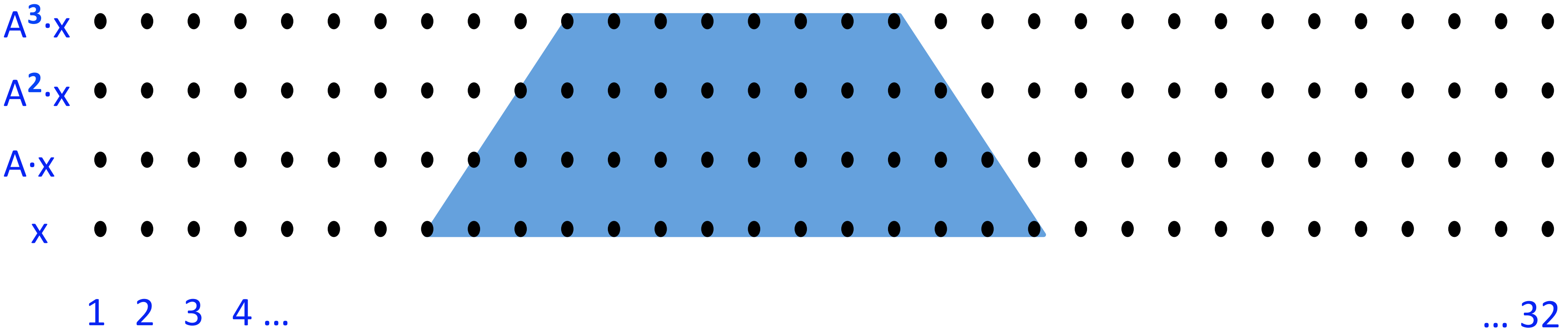


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$

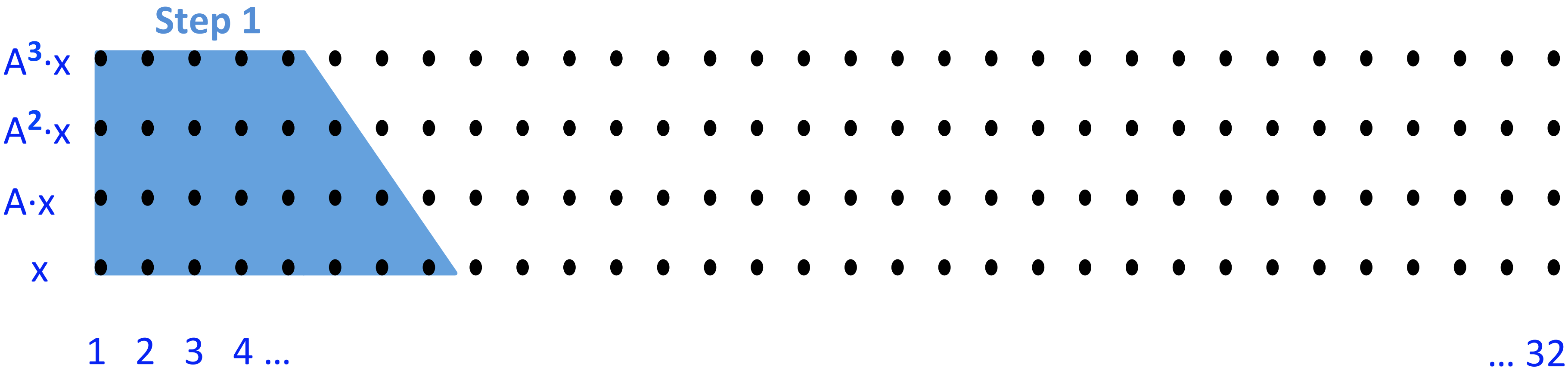


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]
- Sequential Algorithm

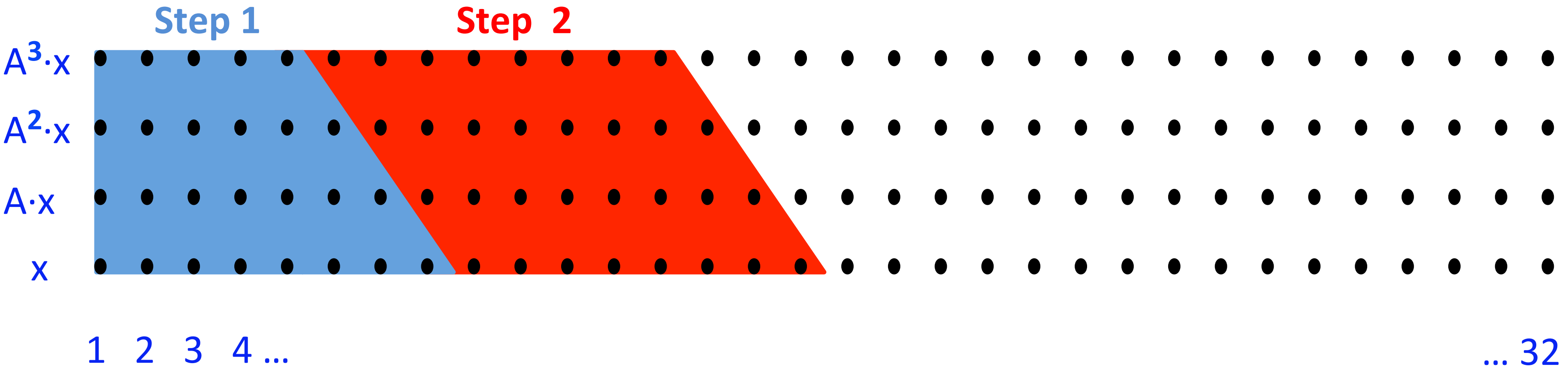


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$
- Sequential Algorithm

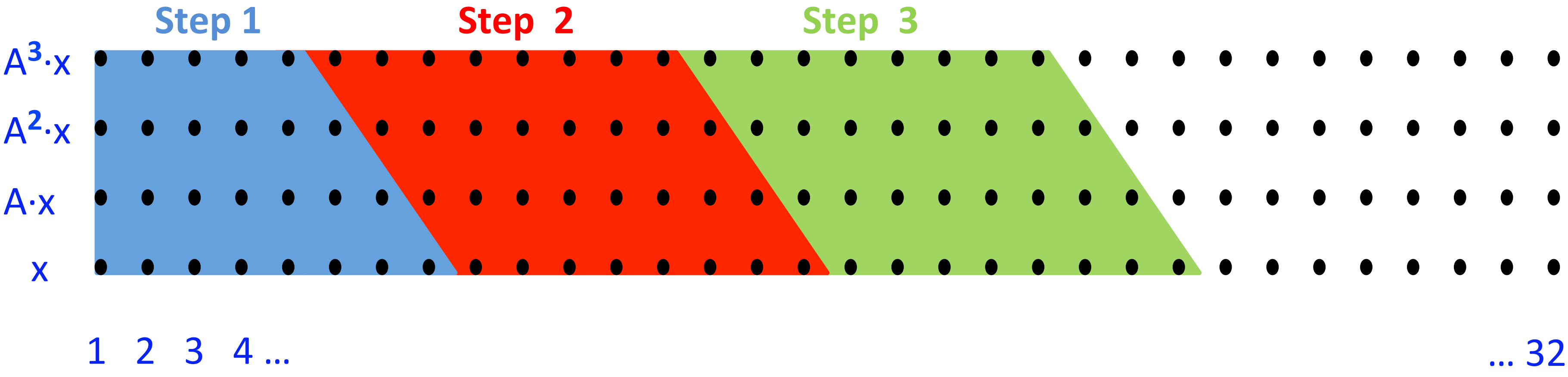


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

- Replace k iterations of $y = A\cdot x$ with $[Ax, A^2x, \ldots, A^kx]$
- Sequential Algorithm

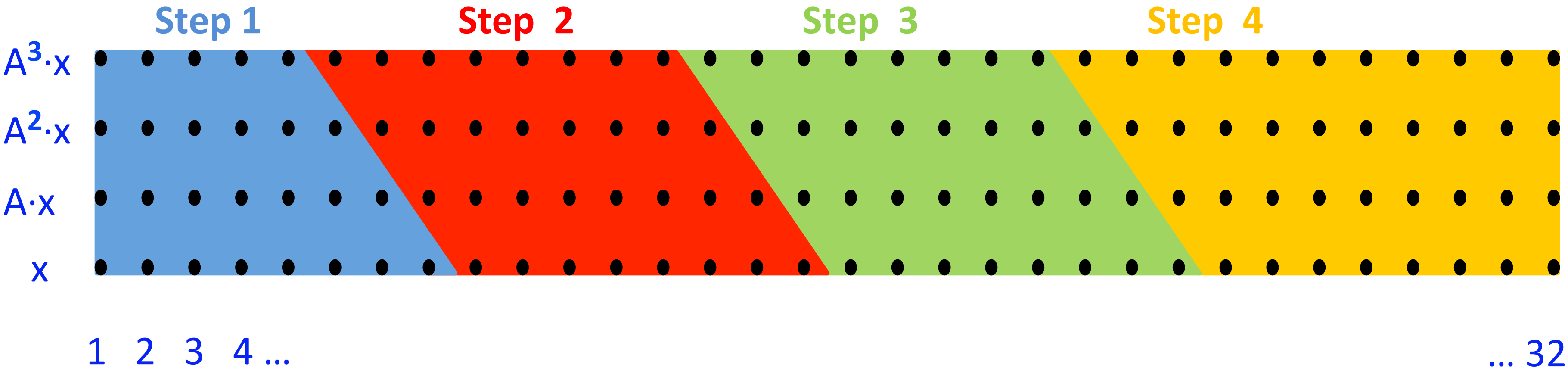


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

The Matrix Powers Kernel : $[Ax, A^2x, \dots, A^kx]$

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \dots, A^kx]$
- Sequential Algorithm

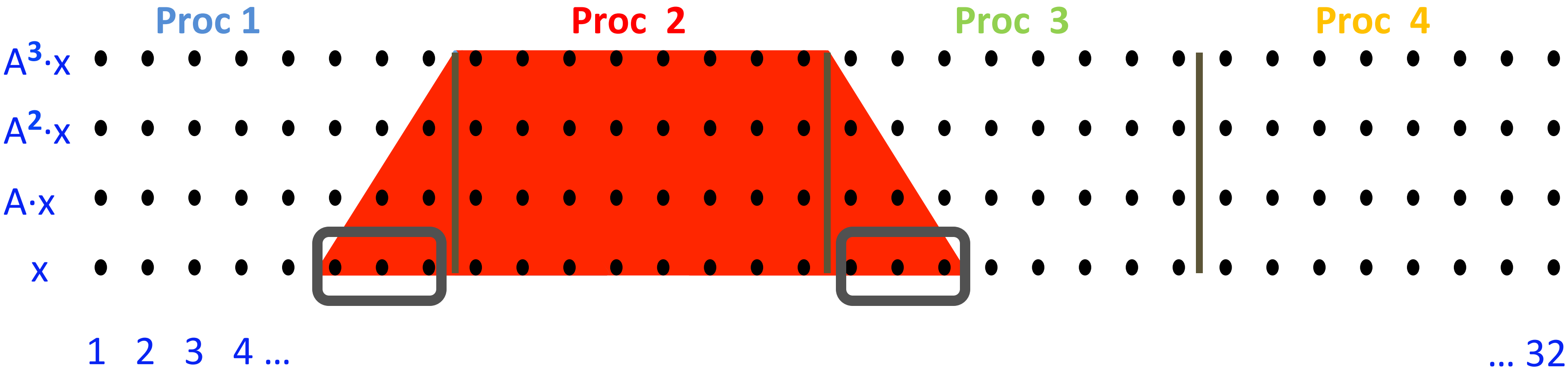


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

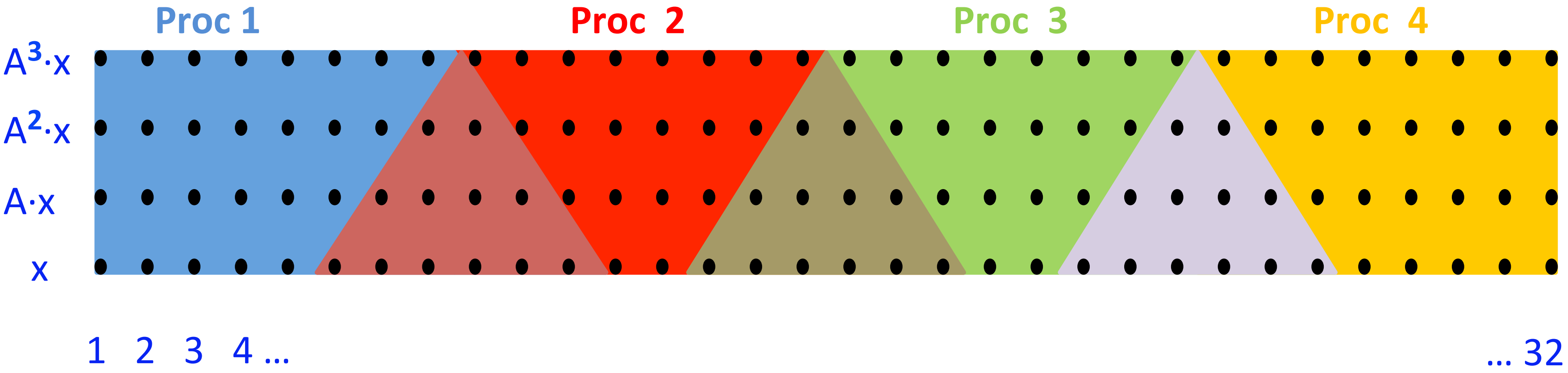
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]
- Parallel Algorithm



- Example: A tridiagonal, n=32, k=3
- Each processor communicates once with neighbors

Communication Avoiding Kernels:
The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$
- Parallel Algorithm



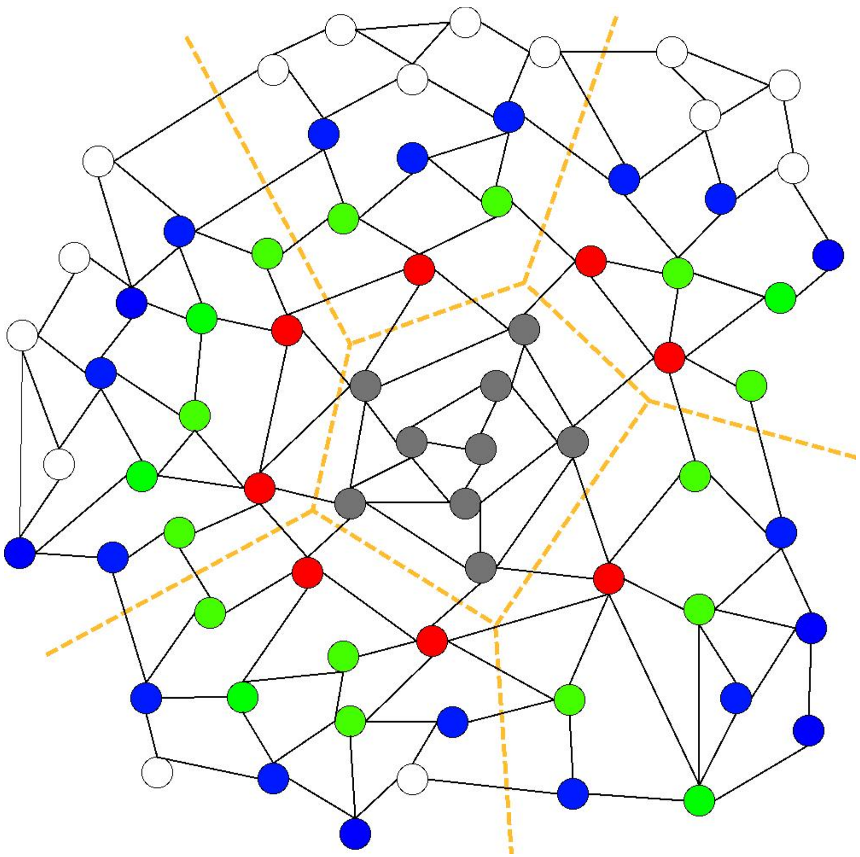
- Example: A tridiagonal, n=32, k=3
- Each processor works on (overlapping) trapezoid

Communication Avoiding Kernels:
The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

Same idea works for general sparse matrices

Simple block-row partitioning ➔
(hyper)graph partitioning

Top-to-bottom processing ➔
Traveling Salesman Problem



Minimizing Communication of GMRES to solve Ax=b

- GMRES: find x in span{b,Ab,…,A^kb} minimizing || Ax-b $||_2$

Standard GMRES

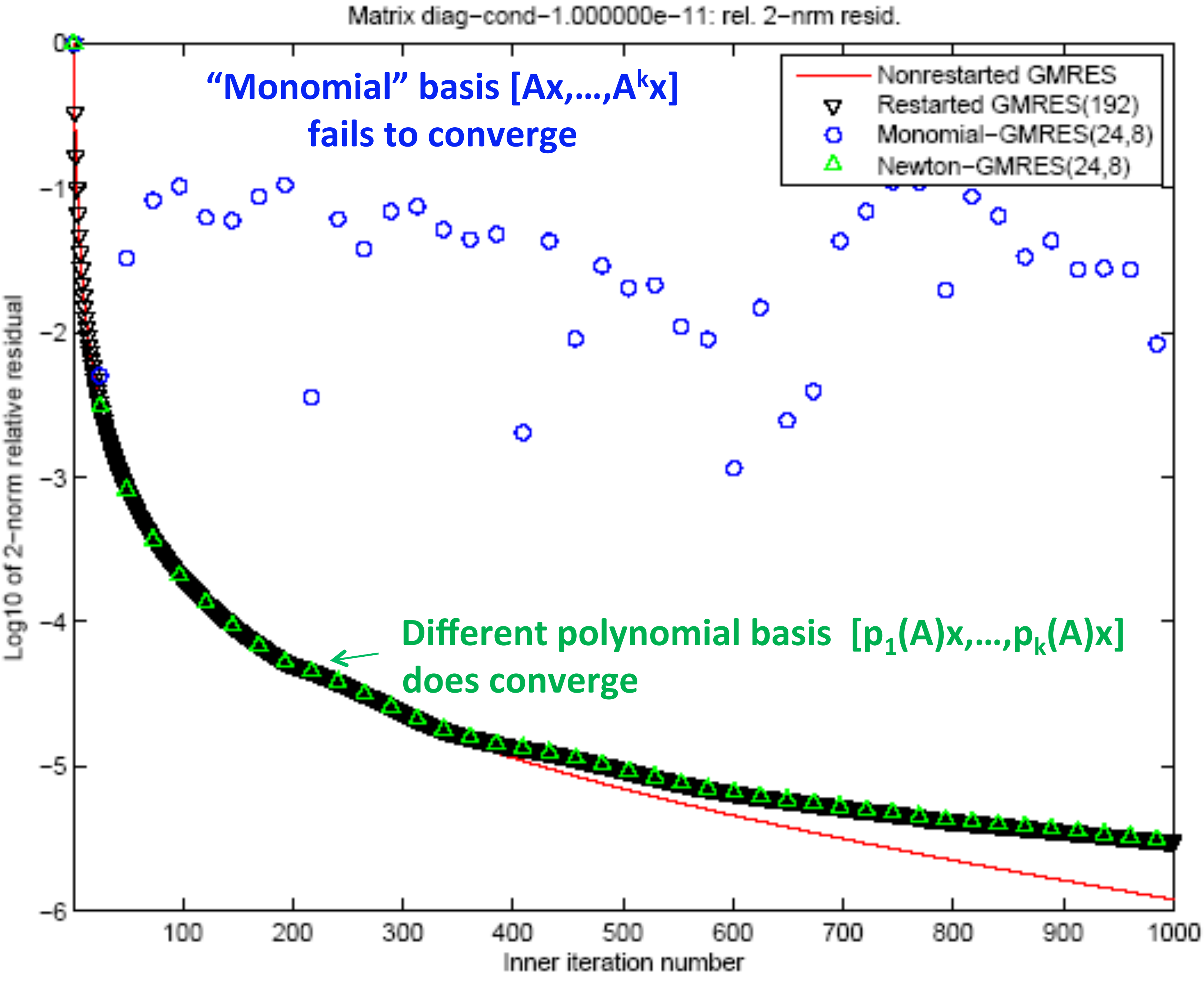
```
for i=1 to k
   w = A · v(i-1)   … SpMV
   MGS(w, v(0),…,v(i-1))
   update v(i), H
endfor
solve LSQ problem with H
```

Communication-avoiding GMRES

```
W = [ v, Av, A^2v, … , A^kv ]
[Q,R] = TSQR(W)
      …  "Tall Skinny QR"
build H from R
solve LSQ problem with H
```

Sequential case: #words moved decreases by a factor of k
Parallel case: #messages decreases by a factor of k

- Oops – W from power method, precision lost!



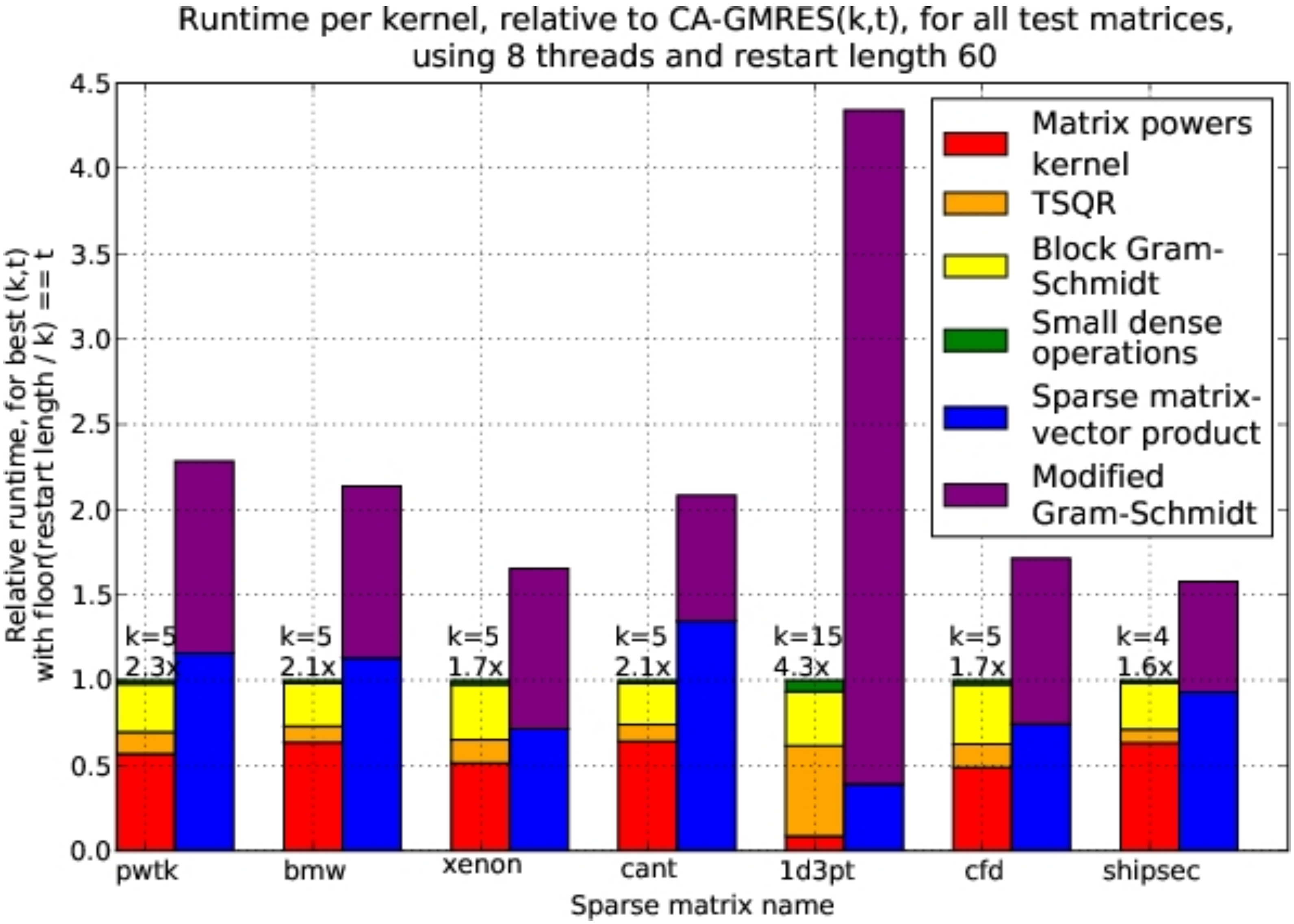
“Monomial” basis $[Ax,\ldots,A^k x]$ fails to converge

Different polynomial basis $[p_1(A)x,\ldots,p_k(A)x]$ does converge

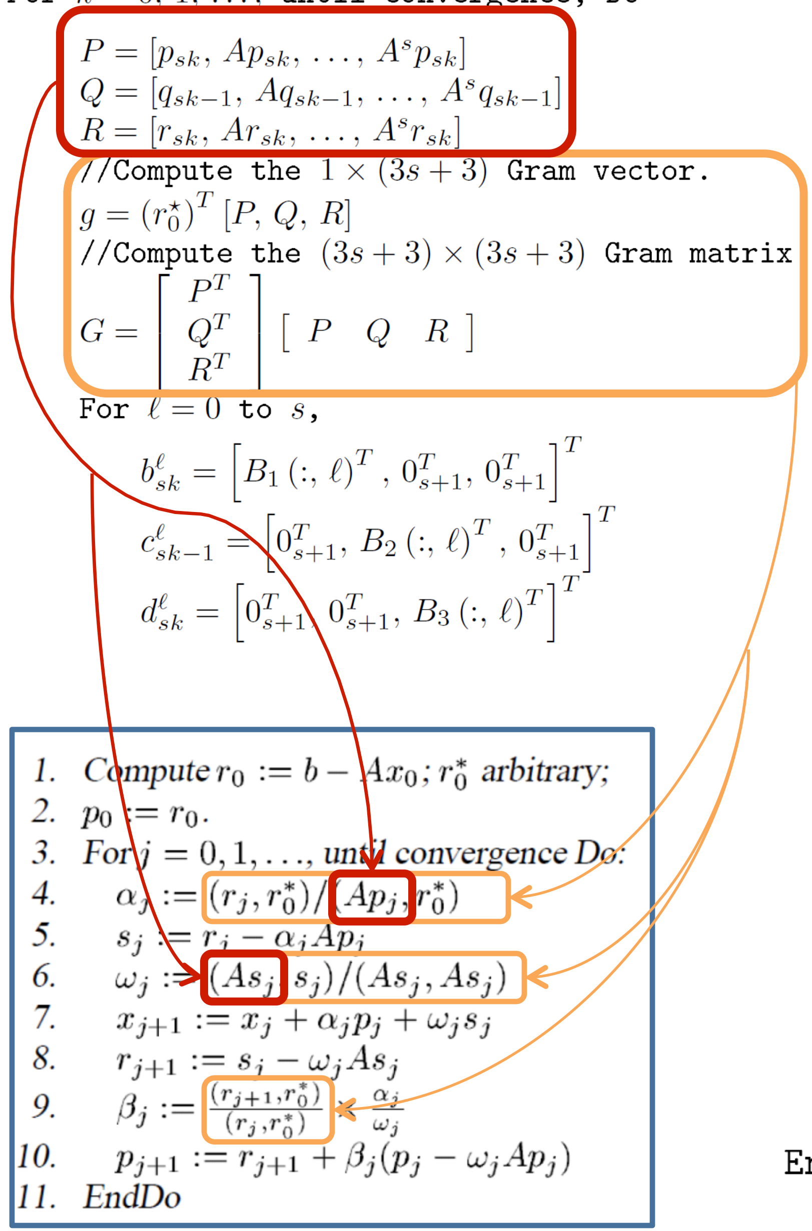
Speed ups of GMRES on 8-core Intel Clovertown

Requires Co-tuning Kernels

[MHDY09]

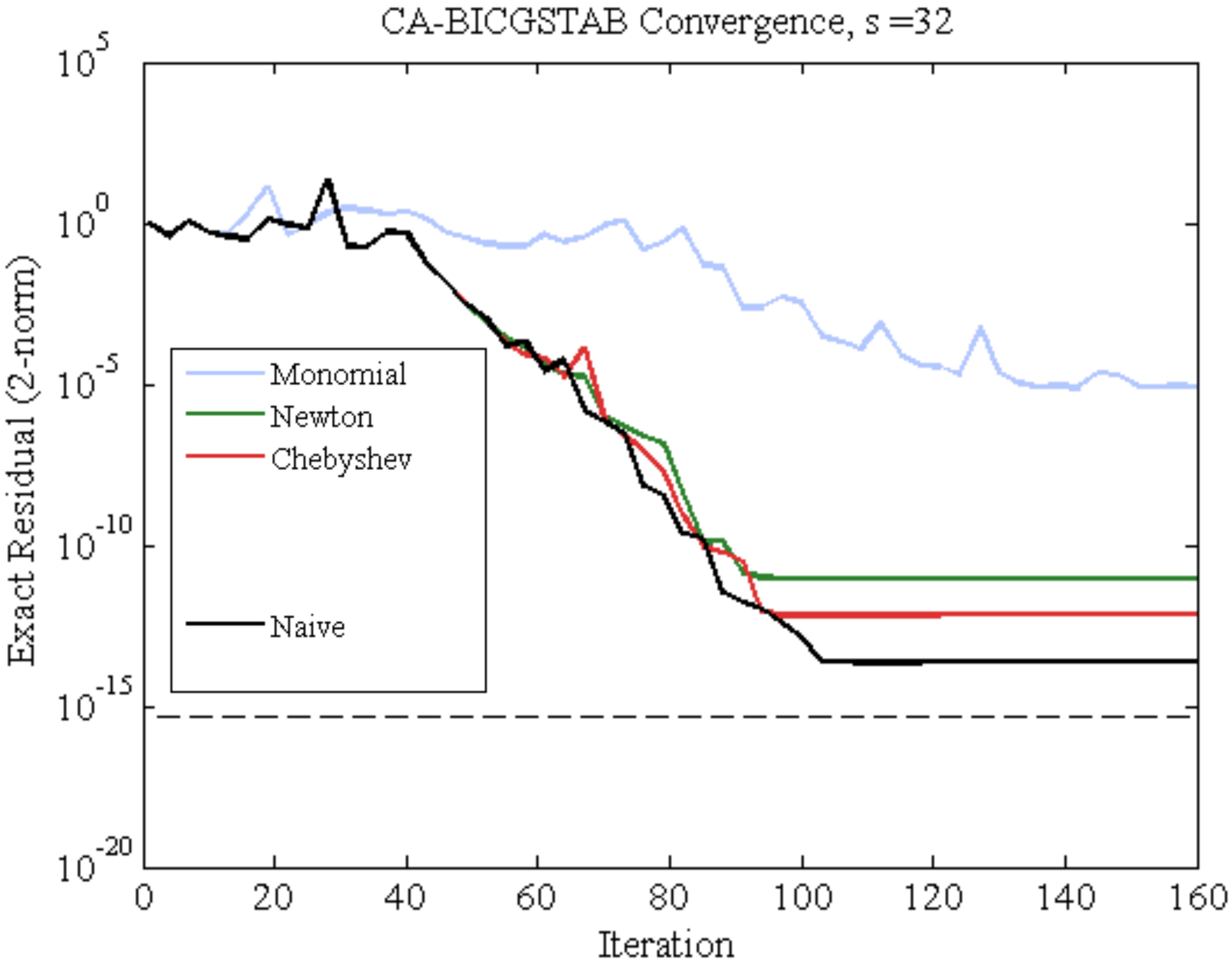


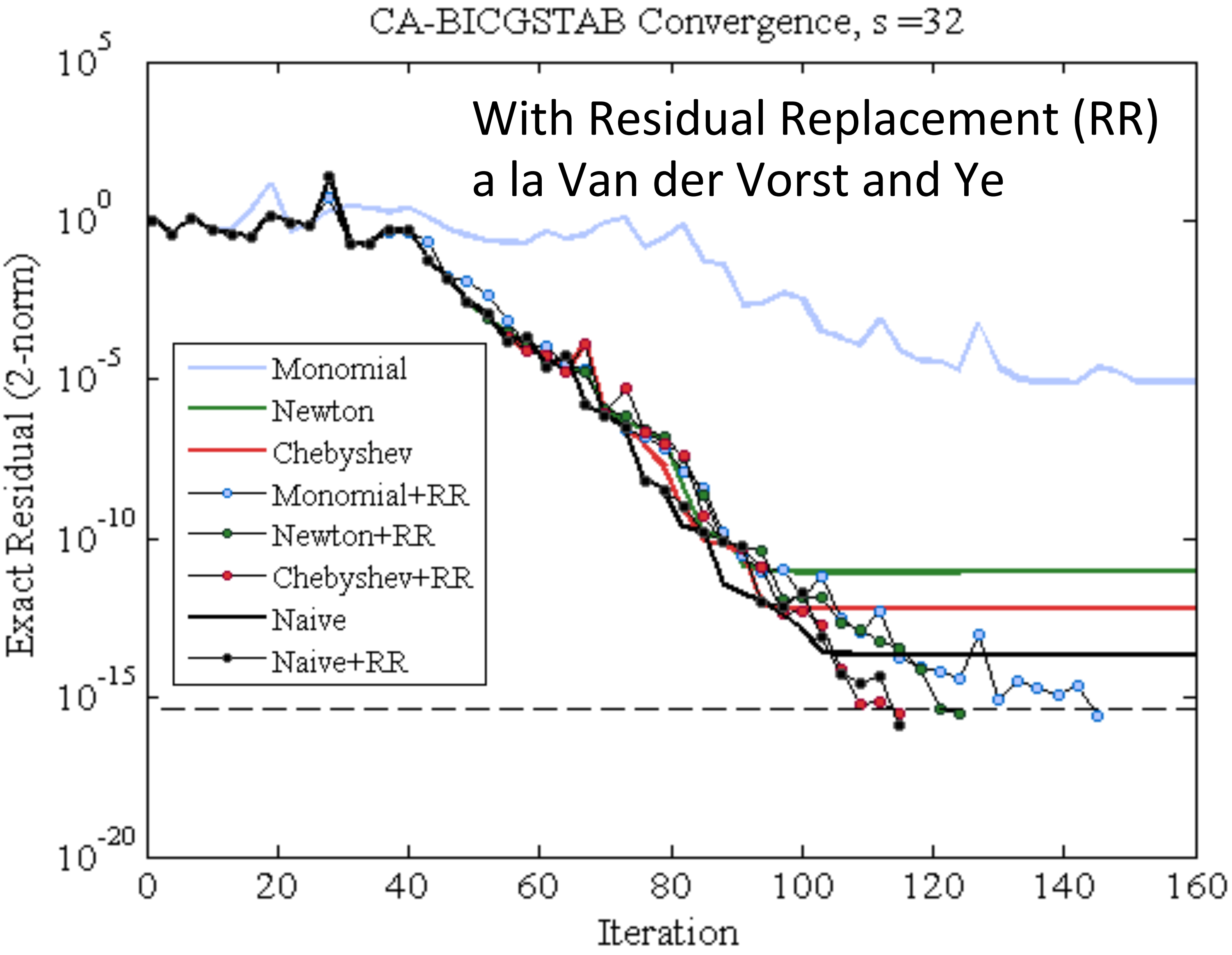
CA-BiCGStab



```
Compute $r_0 = b - Ax_0$. Choose $r_0^\star$ arbitrary.
Set $p_0 = r_0$, $q_{-1} = 0_{N \times 1}$.
For $k = 0, 1, \ldots,$ until convergence, Do
    $P = [p_{sk}, Ap_{sk}, \ldots, A^s p_{sk}]$
    $Q = [q_{sk-1}, Aq_{sk-1}, \ldots, A^s q_{sk-1}]$
    $R = [r_{sk}, Ar_{sk}, \ldots, A^s r_{sk}]$
    //Compute the $1 \times (3s+3)$ Gram vector.
    $g = (r_0^\star)^T [P, Q, R]$
    //Compute the $(3s+3) \times (3s+3)$ Gram matrix
    $G = \begin{bmatrix} P^T \\ Q^T \\ R^T \end{bmatrix} \begin{bmatrix} P & Q & R \end{bmatrix}$
    For $\ell = 0$ to $s$,
        $b^\ell_{sk} = \left[B_1(:,\ell)^T, 0^T_{s+1}, 0^T_{s+1}\right]^T$
        $c^\ell_{sk-1} = \left[0^T_{s+1}, B_2(:,\ell)^T, 0^T_{s+1}\right]^T$
        $d^\ell_{sk} = \left[0^T_{s+1}, 0^T_{s+1}, B_3(:,\ell)^T\right]^T$
    For $j = 0$ to $\lfloor \frac{s}{2} \rfloor - 1$, Do
        $\alpha_{sk+j} = \frac{<g, d^0_{sk+j}>}{<g, b^1_{sk+j}>}$
        $q_{sk+j} = r_{sk+j} - \alpha_{sk+j}[P, Q, R]b^1_{sk+j}$
        For $\ell = 0$ to $s - 2j + 1$, Do
            $c^\ell_{sk+j} = d^\ell_{sk+j} - \alpha_{sk+j} b^{\ell+1}_{sk+j-1}$
            //such that $[P, Q, R]\, c^\ell_{sk+j} = A^\ell q_{sk+j}$
        $\omega_{sk+j} = \frac{<c^1_{sk+j+1}, Gc^0_{sk+j+1}>}{<c^1_{sk+j+1}, Gc^1_{sk+j+1}>}$
        $x_{sk+j+1} = x_{sk+j} + \alpha_{sk+j} p_{sk+j} + \omega_{sk+j} q_{sk+j}$
        $r_{sk+j+1} = q_{sk+j} - \omega_{sk+j}[P, Q, R]c^1_{sk+j+1}$
        For $\ell = 0$ to $s - 2j$, Do
            $d^\ell_{sk+j+1} = c^\ell_{sk+j+1} - \omega_{sk+j} c^{\ell+1}_{sk+j+1}$
            //such that $[P, Q, R]\, d^\ell_{sk+j+1} = A^\ell r_{sk+j+1}$
        $\beta_{sk+j} = \frac{<g, d^0_{sk+j+1}>}{<g, d^0_{sk+j}>} \times \frac{\alpha}{\omega}$
        $p_{sk+j+1} = r_{sk+j+1} + \beta_{sk+j} p_{sk+j} - \beta_{sk+j}\omega_{sk+j}[P, Q, R]b^1_{sk+j}$
        For $\ell = 0$ to $s - 2j$, Do
            $b^\ell_{sk+j+1} = d^\ell_{sk+j+1} + \beta_{sk+j} b^\ell_{sk+j} - \beta_{sk+j}\omega_{sk+j} b^{\ell+1}_{sk+j}$
            //such that $[P, Q, R]\, b^\ell_{sk+j+1} = A^\ell p_{sk+j+1}$.
    EndDo
EndDo
```

```
Compute $r_0 := b - Ax_0$; $r_0^*$ arbitrary;
$p_0 := r_0$.
For $j = 0, 1, \ldots,$ until convergence Do:
    $\alpha_j := \boxed{(r_j, r_0^*)/\boxed{(Ap_j,} r_0^*)}$
    $s_j := r_j - \alpha_j Ap_j$
    $\omega_j := \boxed{\boxed{(As_j,} s_j)/(As_j, As_j)}$
    $x_{j+1} := x_j + \alpha_j p_j + \omega_j s_j$
    $r_{j+1} := s_j - \omega_j As_j$
    $\beta_j := \boxed{\frac{(r_{j+1}, r_0^*)}{(r_j, r_0^*)}} \frac{\alpha_j}{\omega_j}$
    $p_{j+1} := r_{j+1} + \beta_j(p_j - \omega_j Ap_j)$
EndDo
```


CA-BICGSTAB Convergence, s =32
Exact Residual (2-norm)
10^5
10^0
10^-5
10^-10
10^-15
10^-20
0
20
40
60
80
100
120
140
160
Iteration
Monomial
Newton
Chebyshev
Naive



With Residual Replacement (RR)
a la Van der Vorst and Ye

	Naive	**Monomial**	**Newton**	**Chebyshev**
Replacement Its.	74 **(1)**	[7, 15, 24, 31, …, 92, 97, 103] **(17)**	[67, 98] **(2)**	68 **(1)**

Tuning space for Krylov Methods

- Classifications of sparse operators for avoiding communication
  - Explicit indices or nonzero entries cause most communication, along with vectors
  - Ex: With stencils (all implicit) all communication for vectors

		Indices	
		Explicit (O(nnz))	Implicit (o(nnz))
Nonzero entries	Explicit (O(nnz))		
	Implicit (o(nnz))		

- Operations
  - $[x, Ax, A^2x, \ldots, A^kx]$ or $[x, p_1(A)x, p_2(A)x, \ldots, p_k(A)x]$
  - Number of columns in x
  - $[x, Ax, A^2x, \ldots, A^kx]$ and $[y, A^Ty, (A^T)^2y, \ldots, (A^T)^ky]$, or $[y, A^TAy, (A^TA)^2y, \ldots, (A^TA)^ky]$,
  - return all vectors or just last one
- Cotuning and/or interleaving
  - $W = [x, Ax, A^2x, \ldots, A^kx]$ and $\{\mathrm{TSQR}(W)$ or W^TW or … $\}$
  - Ditto, but throw away W
- Preconditioned versions

Summary of Iterative Linear Algebra

- New Lower bounds, optimal algorithms, big speedups in theory and practice
- Lots of other progress, open problems
  - Many different algorithms reorganized
    - More underway, more to be done
  - Need to recognize stable variants more easily
  - Preconditioning
    - Hierarchically Semiseparable Matrices
  - Autotuning and synthesis
    - pOSKI for SpMV – available at bebop.cs.berkeley.edu
    - Different kinds of "sparse matrices"

Outline

- "Direct" Linear Algebra
  - Lower bounds on communication
  - New algorithms that attain these lower bounds
- Ditto for "Iterative" Linear Algebra
- Ditto (work in progress) for programs accessing arrays (eg n-body)

Recall optimal sequential Matmul

- Naïve code

```
for i=1:n, for j=1:n, for k=1:n, C(i,j)+=A(i,k)*B(k,j)
```

- "Blocked" code

```
for i1 = 1:b:n,  for j1 = 1:b:n,   for k1 = 1:b:n
  for i2 = 0:b-1,  for j2 = 0:b-1,   for k2 = 0:b-1  ┐
    i=i1+i2,  j = j1+j2,  k = k1+k2                   }  b x b matmul
    C(i,j)+=A(i,k)*B(k,j)                            ┘
```

- Thm: Picking b = $M^{1/2}$ attains lower bound:
  #words_moved = $\Omega(n^3/M^{1/2})$
- Where does **1/2** come from?

New Thm applied to Matmul

- for i=1:n, for j=1:n, for k=1:n, C(i,j) += A(i,k)*B(k,j)
- Record array indices in matrix Δ

$$\Delta = \begin{array}{cccc} i & j & k & \\ 1 & 0 & 1 & A \\ 0 & 1 & 1 & B \\ 1 & 1 & 0 & C \end{array}$$

- Solve LP for $x = [xi,xj,xk]^T$: max $\mathbf{1}^T x$ s.t. $\Delta\, x \le \mathbf{1}$
  - Result: $x = [1/2, 1/2, 1/2]^T$, $\mathbf{1}^T x = 3/2 = s$
- Thm: #words_moved = $\Omega(n^3/M^{S-1}) = \Omega(n^3/M^{1/2})$

Attained by block sizes $M^{xi}, M^{xj}, M^{xk} = M^{1/2}, M^{1/2}, M^{1/2}$

New Thm applied to Direct N-Body

- for i=1:n, for j=1:n, F(i) += force(P(i) , P(j))
- Record array indices in matrix Δ

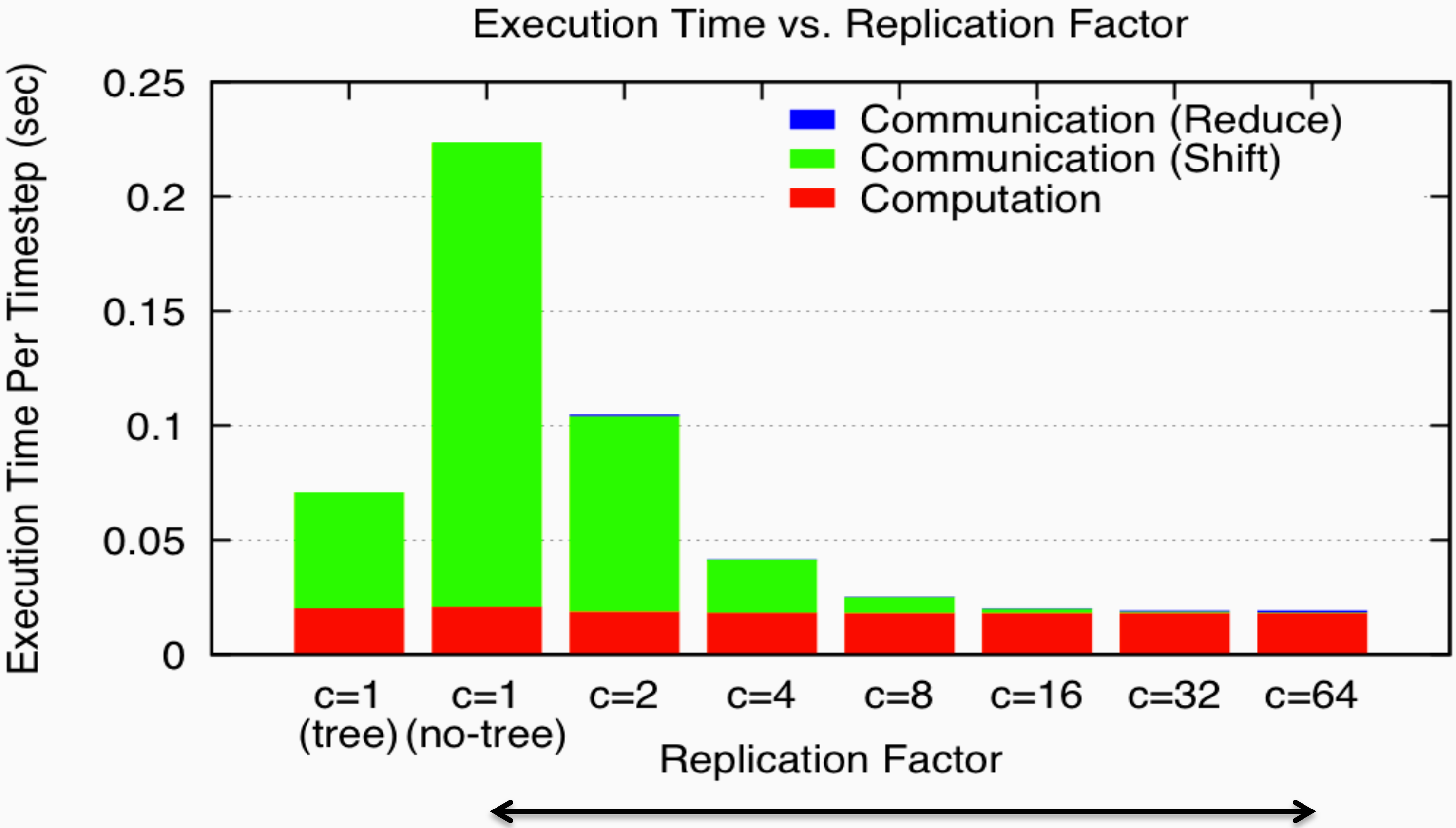
$$\begin{array}{cccc} & i & j & \\ \Delta = & 1 & 0 & F \\ & 1 & 0 & P(i) \\ & 0 & 1 & P(j) \end{array}$$

- Solve LP for $x = [xi,xj]^T$: max $\mathbf{1}^T x$ s.t. $\Delta\, x \leq \mathbf{1}$
  - Result: $x = [1,1]$, $\mathbf{1}^T x = 2 = s$
- Thm: #words_moved = $\Omega(n^2/M^{S-1}) = \Omega(n^2/M^1)$

  Attained by block sizes $M^{xi}, M^{xj} = M^1, M^1$

N-Body Speedups on IBM-BG/P (Intrepid) 8K cores, 32K particles

K. Yelick, E. Georganas, M. Driscoll, P. Koanantakool, E. Solomonik



11.8x speedup

New Thm applied to Random Code

- for i1=1:n, for i2=1:n, … , for i6=1:n

  A1(i1,i3,i6) += func1(A2(i1,i2,i4),A3(i2,i3,i5),A4(i3,i4,i6))

  A5(i2,i6) += func2(A6(i1,i4,i5),A3(i3,i4,i6))

- Record array indices in matrix Δ

$$\Delta = \begin{array}{cccccc|c} i1 & i2 & i3 & i4 & i5 & i6 & \\ \hline 1 & 0 & 1 & 0 & 0 & 1 & A1 \\ 1 & 1 & 0 & 1 & 0 & 0 & A2 \\ 0 & 1 & 1 & 0 & 1 & 0 & A3 \\ 0 & 0 & 1 & 1 & 0 & 1 & A3,A4 \\ 0 & 0 & 1 & 1 & 0 & 1 & A5 \\ 1 & 0 & 0 & 1 & 1 & 0 & A6 \end{array}$$

- Solve LP for $x = [x1,\dots,x7]^T$: max $\mathbf{1}^T x$ s.t. $\Delta\, x \le \mathbf{1}$
  - Result: $x = [2/7,3/7,1/7,2/7,3/7,4/7]$, $\mathbf{1}^T x = 15/7 = s$
- Thm: #words_moved = $\Omega(n^6/M^{S-1}) = \Omega(n^6/M^{8/7})$

  Attained by block sizes $M^{2/7},M^{3/7},M^{1/7},M^{2/7},M^{3/7},M^{4/7}$

Approach to generalizing lower bounds

- Matmul

```
 for i=1:n, for j=1:n, for k=1:n,
       C(i,j)+=A(i,k)*B(k,j)
```

=> for (i,j,k) in S = subset of Z^3

Access locations indexed by (i,j), (i,k), (k,j)

- General case

```
for i1=1:n,  for i2 = i1:m, … for ik = i3:i4
      C(i1+2*i3-i7) = func(A(i2+3*i4,i1,i2,i1+i2,…),B(pnt(3*i4)),…)
      D(something else) = func(something else),  …
```

=> for (i1,i2,…,ik) in S = subset of Z^k

Access locations indexed by “projections”, eg

ϕ_C (i1,i2,…,ik) = (i1+2*i3-i7)

ϕ_A (i1,i2,…,ik) = (i2+3*i4,i1,i2,i1+i2,…), …

General Communication Bound

- Def: Hölder-Brascamp-Lieb Linear Program (HBL-LP)

  for $s_1,\ldots,s_m$:

  for all subgroups $H < Z^k$, $\text{rank}(H) \leq \Sigma_j\, s_j * \text{rank}(\phi_j(H))$
- Thm: Given a program with array refs given by ϕ_j, choose s_j to minimize $s_{HBL} = \Sigma_j\, s_j$ subject to HBL-LP. Then

  $$\text{\#words\_moved} = \Omega\,(\text{\#iterations}/M^{s_{HBL}-1})$$

  - Proof depends on recent result in pure mathematics by Christ/Tao/Carbery/Bennett

Is this bound attainable (1/2)?

- But first: Can we write it down?
  - Thm: (bad news) Reduces to Hilbert's 10th problem over Q (conjectured to be undecidable)
  - Thm: (good news) Can write it down explicitly in many cases of interest (eg all ϕ_j = {subset of indices})
  - Thm: (good news) Easy to approximate
    - If you miss a constraint, the lower bound may be too large (i.e. s_{HBL} too small) but still worth trying to attain, because your algorithm will still communicate less

Is this bound attainable (2/2)?

- Depends on loop dependencies
- Best case: none, or reductions (matmul)
- Thm: When all ϕ_j = {subset of indices}, dual of HBL-LP gives optimal tile sizes:

  HBL-LP: minimize $1^T * s$ s.t. $s^T * \Delta \geq 1^T$

  Dual-HBL-LP: maximize $1^T * x$ s.t. $\Delta * x \leq 1$

  Then for sequential algorithm, tile i_j by M^{x_j}
- Ex: Matmul: $s = [\ 1/2\ ,\ 1/2\ ,\ 1/2\ ]^T = x$
- Extends to unimodular transforms of indices

Ongoing Work

- Identify more decidable cases
  - Works for any 3 nested loops, or 3 different subscripts
- Automate generation of approximate LPs
- Extend "perfect scaling" results for time and energy by using extra memory
- Have yet to find a case where we cannot attain lower bound – can we prove this?
- Incorporate into compilers

For more details

- Bebop.cs.berkeley.edu
- CS267 – Berkeley's Parallel Computing Course
  - Live broadcast in Spring 2013
    - www.cs.berkeley.edu/~demmel
  - Prerecorded version planned in Spring 2013
    - www.xsede.org
    - Free supercomputer accounts to do homework!

Summary

Time to redesign all linear algebra, n-body,…
algorithms and software
(and compilers…)

Don’t Communic…