Minimizing Communication in Sparse Matrix Solvers

Marghoob Mohiyuddin, Mark Hoemmen,
James Demmel, Kathy Yelick
marghoob@eecs.berkeley.edu

EECS Department, University of California at Berkeley

SC09, Nov 17, 2009

Outline

Outline

1. Background
2. The Kernels
   - The matrix powers kernel
   - Tall skinny QR
   - Block Gram-Schmidt orthogonalization
3. Integrated Solver (GMRES)
4. Conclusions

What is communication?

Algorithms incur 2 costs:

What is communication?

Algorithms incur 2 costs:

1. Arithmetic (flops)
2. Communication (data movement)

What is communication?

Algorithms incur 2 costs:

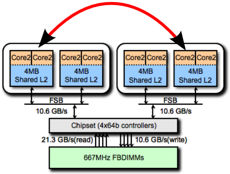
1. Arithmetic (flops)
2. Communication (data movement)
   - Bandwidth (#words) and latency (#messages) components

What is communication?

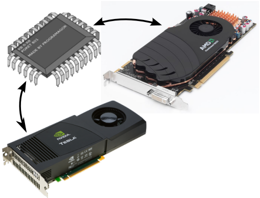
Algorithms incur 2 costs:

1. Arithmetic (flops)
2. Communication (data movement)
   - Bandwidth (#words) and latency (#messages) components

Parallel



Between CPUs



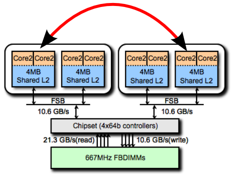
Between CPUs and coprocessors

What is communication?

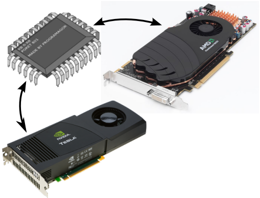
Algorithms incur 2 costs:

1. Arithmetic (flops)
2. Communication (data movement)
   - Bandwidth (#words) and latency (#messages) components

Parallel

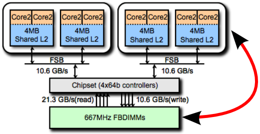


Between CPUs

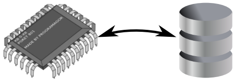


Between CPUs and coprocessors

Sequential



Between cache and DRAM



Between DRAM and disk

Communication is expensive, computation is cheap

- Time per flop $\gg$ 1/bandwidth $\gg$ latency
- Gap between processing power and communication cost increasing exponentially

Annual improvements	
Flop rate	59%
DRAM bandwidth	26%
DRAM latency	5%

- Reduce communication $\Rightarrow$ improve efficiency
- Trading off communication for computation is okay

The problem with sparse iterative solvers

Conventional GMRES (solve for $Ax = b$)

1. **for** $i = 1$ to r
2. $\quad w = Av_{i-1}$ */* SpMV */*
3. $\quad$ Orthogonalize w against $\{v_0, \ldots, v_{i-1}\}$ */* MGS */*
4. $\quad$ Update vector v_i, matrix H
5. Use H, $\{v_0, \ldots, v_r\}$ to construct the solution

The problem with sparse iterative solvers

Conventional GMRES (solve for $Ax = b$)

1. **for** $i = 1$ to r
2. $w = Av_{i-1}$ */* SpMV */*
3. Orthogonalize w against $\{v_0, \ldots, v_{i-1}\}$ */* MGS */*
4. Update vector v_i, matrix H
5. Use H, $\{v_0, \ldots, v_r\}$ to construct the solution

- Repeated calls to sparse matrix vector multiply (SpMV) & Modified Gram Schmidt orthogonalization (MGS)
  - SpMV: performs 2 flops/matrix nonzero entry $\Rightarrow$ communication bound
  - MGS: vector dot-products (BLAS level 1) $\Rightarrow$ communication bound

The problem with sparse iterative solvers

Conventional GMRES (solve for $Ax = b$)

1. **for** $i = 1$ to r
2. $w = Av_{i-1}$ */* SpMV */*
3. Orthogonalize w against $\{v_0, \ldots, v_{i-1}\}$ */* MGS */*
4. Update vector v_i, matrix H
5. Use H, $\{v_0, \ldots, v_r\}$ to construct the solution

Solution

- Replace SpMV and MGS by new kernels:
  - SpMV by matrix powers
  - MGS by block Gram-Schmidt + TSQR
- Reformulate to use the new kernels

Outline

Outline

The matrix powers kernel

- Usual kernel $y = Ax$ communication-bound for large matrices
  - Large $\Rightarrow$ does not fit in cache
  - Need to read stream through the matrix
- Given sparse matrix A, vector x, integer $k > 0$, compute $[p_1(A)x, p_2(A)x, \ldots, p_k(A)x]$, $p_i(A)$ degree i polynomial iA
- Easier to consider the special case: $[Ax, A^2x, \ldots, A^kx]$

Naïve parallel algorithm

Example: tridiagonal matrix, $k = 3$, 4 processors

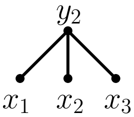
$$\underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{x}$$

Tridiagonal only for illustration

Naïve parallel algorithm

Example: tridiagonal matrix, $k = 3$, 4 processors

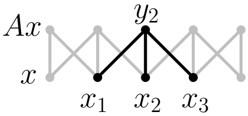
$$\underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{x} \Rightarrow$$



Naïve parallel algorithm

Example: tridiagonal matrix, $k = 3$, 4 processors

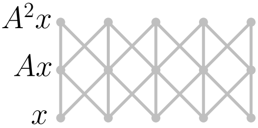
$$\underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{x} \Rightarrow$$



Naïve parallel algorithm

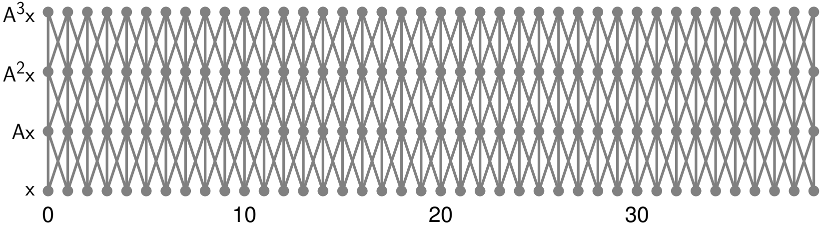
Example: tridiagonal matrix, $k = 3$, 4 processors

$$\underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{x} \Rightarrow$$



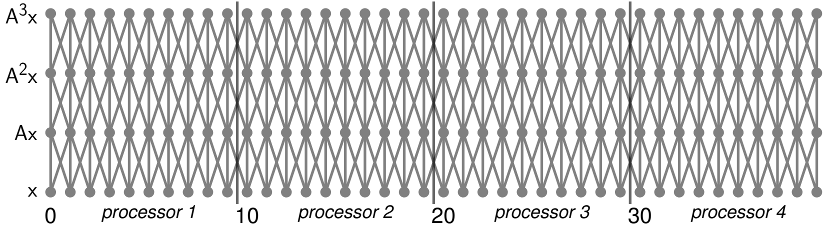
Naïve parallel algorithm

Example: tridiagonal matrix, $k = 3$, 4 processors



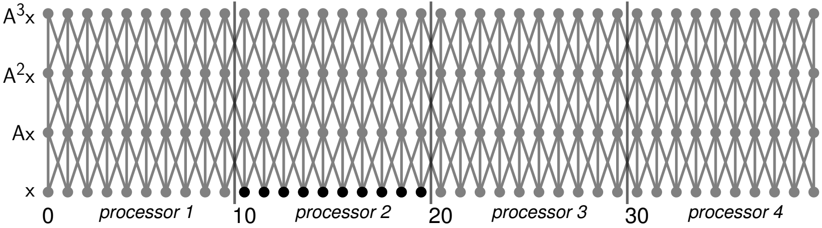
Naïve parallel algorithm

Example: tridiagonal matrix, $k = 3$, 4 processors



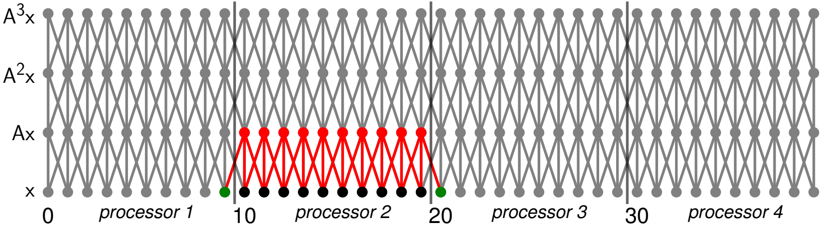
Naïve parallel algorithm

Example: tridiagonal matrix, $k = 3$, 4 processors



Naïve parallel algorithm

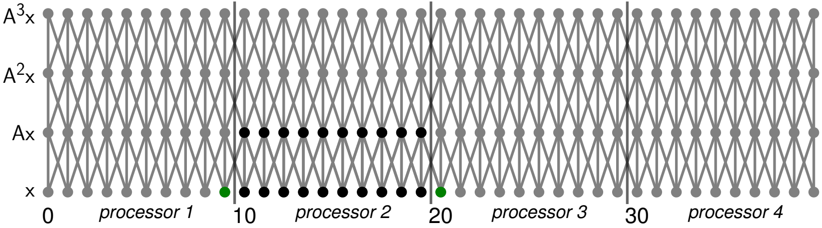
Example: tridiagonal matrix, $k = 3$, 4 processors



1. Fetch green entries of x: 1 message/neighbor

Naïve parallel algorithm

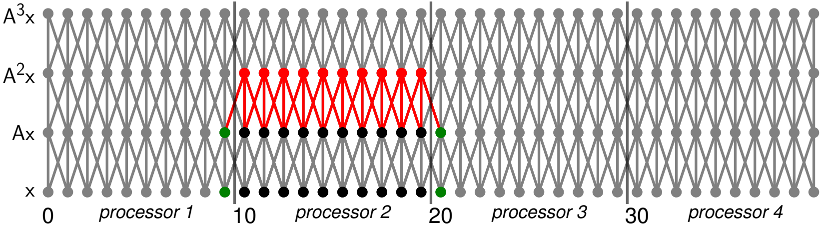
Example: tridiagonal matrix, $k = 3$, 4 processors



1. Fetch green entries of x: 1 message/neighbor
2. Compute local entries of Ax

Naïve parallel algorithm

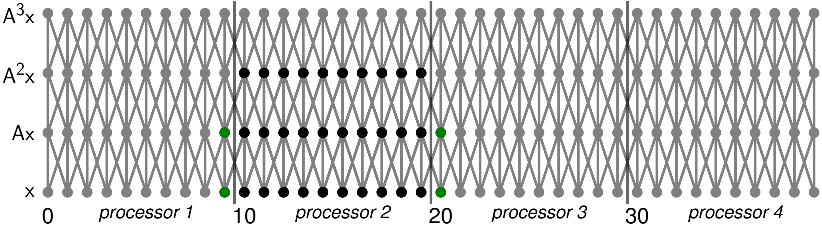
Example: tridiagonal matrix, $k = 3$, 4 processors



1. Fetch green entries of x: 1 message/neighbor
2. Compute local entries of Ax
3. Fetch green entries of Ax: 1 message/neighbor

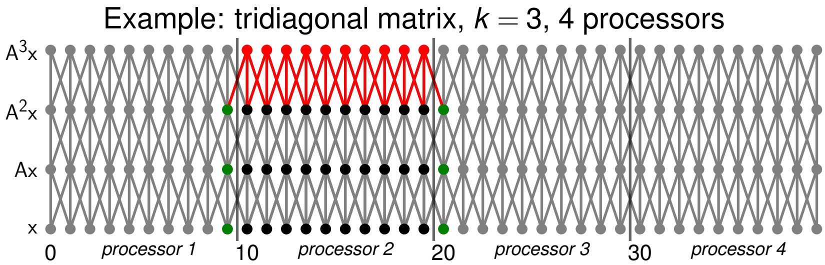
Naïve parallel algorithm

Example: tridiagonal matrix, $k = 3$, 4 processors



1. Fetch green entries of x: 1 message/neighbor
2. Compute local entries of Ax
3. Fetch green entries of Ax: 1 message/neighbor
4. Compute local entries of A^2x

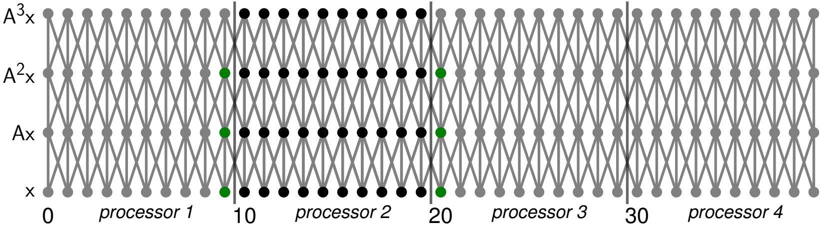
Naïve parallel algorithm



1. Fetch green entries of x: 1 message/neighbor
2. Compute local entries of Ax
3. Fetch green entries of Ax: 1 message/neighbor
4. Compute local entries of A^2x
5. Fetch green entries of A^2x: 1 message/neighbor

Naïve parallel algorithm

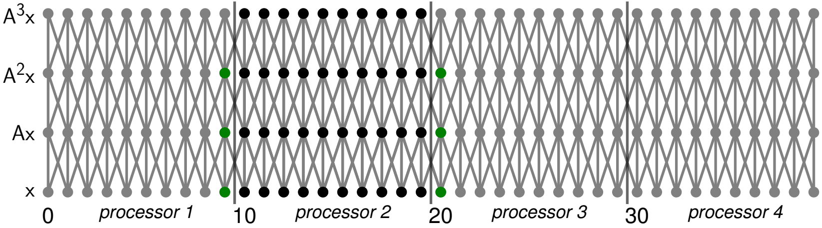
Example: tridiagonal matrix, $k = 3$, 4 processors



1. Fetch green entries of x: 1 message/neighbor
2. Compute local entries of Ax
3. Fetch green entries of Ax: 1 message/neighbor
4. Compute local entries of A^2x
5. Fetch green entries of A^2x: 1 message/neighbor
6. Compute local entries of A^3x

Naïve parallel algorithm

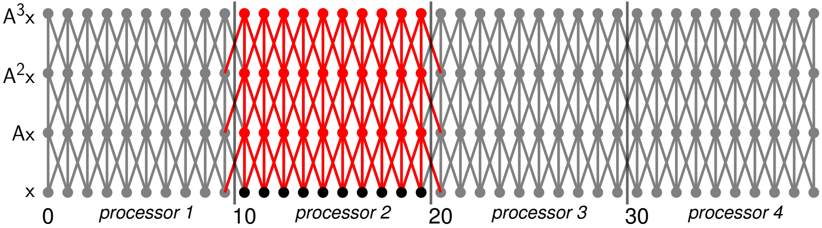
Example: tridiagonal matrix, $k = 3$, 4 processors



- 3 messages/neighbor
- *k* messages/neighbor in general
  - *k* times min. latency cost

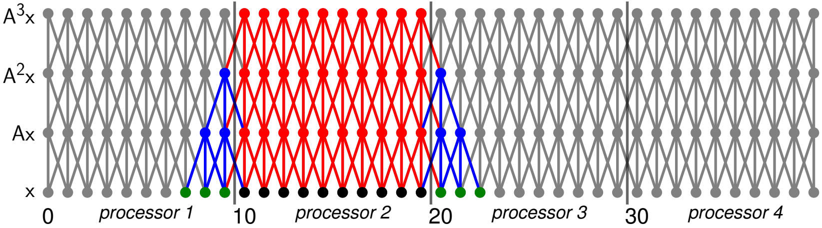
A better parallel algorithm for matrix powers

Example: Tridiagonal matrix, $k = 3$, 4 processors



A better parallel algorithm for matrix powers

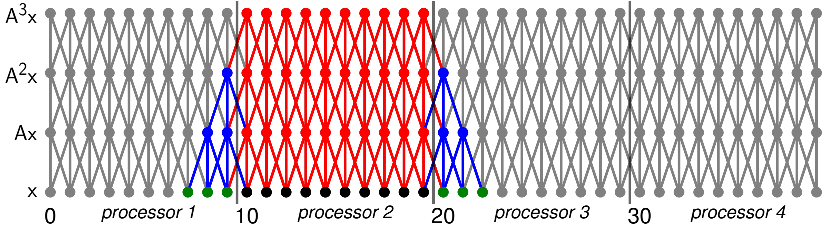
Example: Tridiagonal matrix, $k = 3$, 4 processors



- Green+black entries of x sufficient to compute all the local entries
- Blue entries represent redundant computation

A better parallel algorithm for matrix powers

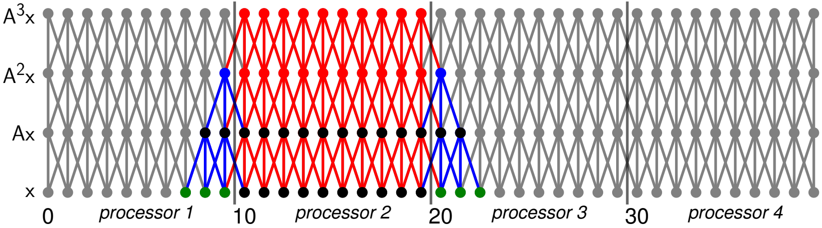
Example: Tridiagonal matrix, $k = 3$, 4 processors



1. Fetch ‘ghost’ entries (green) from other processors
   - *1 message per neighbor*

A better parallel algorithm for matrix powers

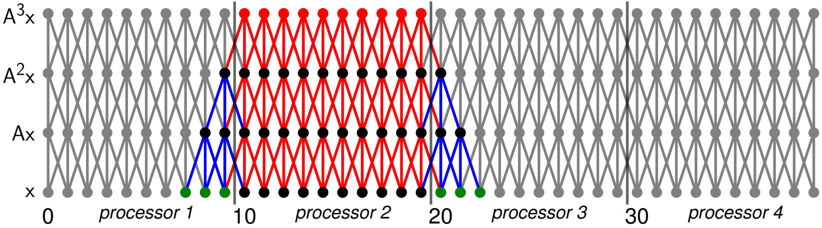
Example: Tridiagonal matrix, $k = 3$, 4 processors



1. Fetch 'ghost' entries (green) from other processors
   - *1 message per neighbor*
2. Compute required entries of Ax

A better parallel algorithm for matrix powers

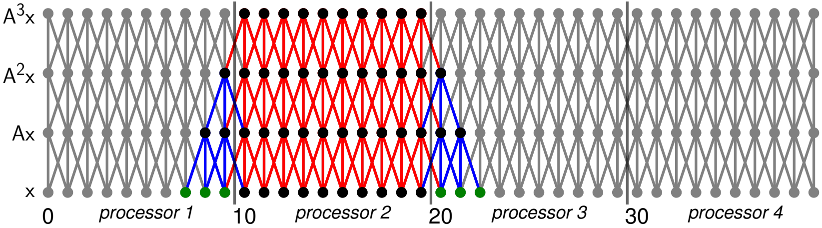
Example: Tridiagonal matrix, $k = 3$, 4 processors



1. Fetch ‘ghost’ entries (green) from other processors
   - *1 message per neighbor*
2. Compute required entries of Ax
3. Compute required entries of A^2x

A better parallel algorithm for matrix powers

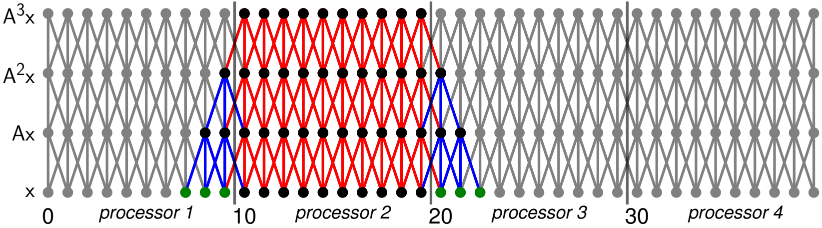
Example: Tridiagonal matrix, $k = 3$, 4 processors



1. Fetch 'ghost' entries (green) from other processors
   - *1 message per neighbor*
2. Compute required entries of Ax
3. Compute required entries of A^2x
4. Compute required entries of A^3x

A better parallel algorithm for matrix powers

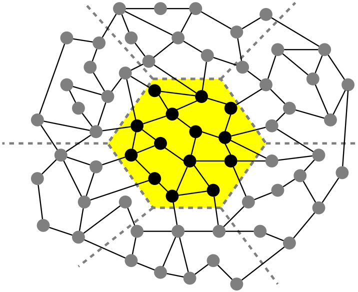
Example: Tridiagonal matrix, $k = 3$, 4 processors



- 1 message/neighbor ($O(k)$ improvement)
- Redundant computation $\Rightarrow$ want it to be small
- Can order local+ghost entries to reuse tuned SpMV

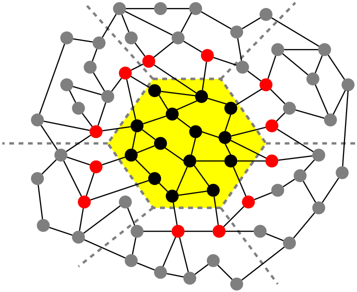
General matrix/graph example

- Our algorithms work for general matrices
- Performance improvement best when the surface-to-volume ratio is small



General matrix/graph example

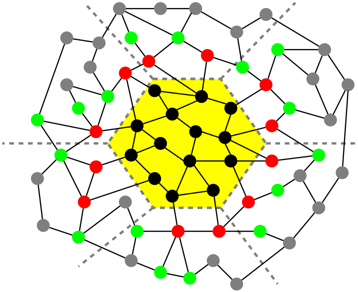
- Our algorithms work for general matrices
- Performance improvement best when the surface-to-volume ratio is small



Red entries of x needed when $k = 1$

General matrix/graph example

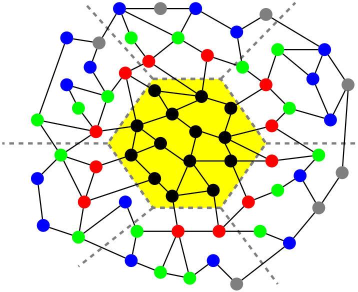
- Our algorithms work for general matrices
- Performance improvement best when the surface-to-volume ratio is small



Red+green entries of x needed when $k = 2$

General matrix/graph example

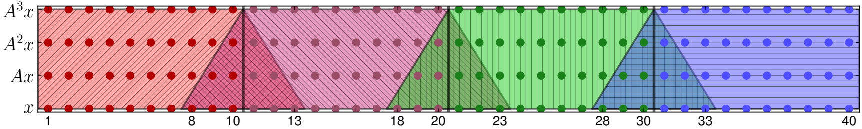
- Our algorithms work for general matrices
- Performance improvement best when the surface-to-volume ratio is small



Red+green+blue entries of x needed when $k = 3$

Sequential algorithms: Explicitly blocked algorithm

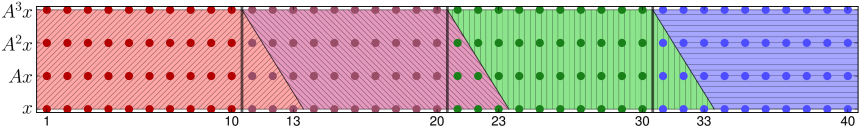
Example: 40×40 tridiagonal matrix, $k = 3$



- Simulate parallel algorithm on 1 processor
- Each block should be small enough to fit in cache
- Redundant flops performed
- Read the matrix once per k iterations ($O(k)$ improvement) $\Rightarrow$ bandwidth savings

Sequential algorithms: Implicitly blocked algorithm

Example: 40×40 tridiagonal matrix, $k = 3$



- Improve upon the explicit algorithm
  - Eliminate redundant computation
- No redundant flops
- Implicit blocking by reordering computations
- Bookkeeping overhead for computation schedule
- Computation inside blocks depends on block order
  $\Rightarrow$ need to solve Traveling Salesman problems

Hybrid algorithm for multicores

- Multicore $\Rightarrow$ 2 kinds of communication:
  - Inter-core on-chip
  - DRAM Off-chip

Hybrid algorithm for multicores

- Multicore $\Rightarrow$ 2 kinds of communication:
  - Inter-core on-chip
  - DRAM Off-chip
- Parallel algorithm minimizes inter-core on-chip communication
- Sequential algorithm minimizes off-chip communication

Hybrid algorithm for multicores

- Multicore $\Rightarrow$ 2 kinds of communication:
  - Inter-core on-chip
  - DRAM Off-chip
- Parallel algorithm minimizes inter-core on-chip communication
- Sequential algorithm minimizes off-chip communication
- Hierarchical blocking of the matrix and vectors
  - Minimize inter-block communication: reordering may occur
  - Cache blocks small enough to hold the matrix and vector entries in cache

Hybrid algorithm for multicores

- Multicore $\Rightarrow$ 2 kinds of communication:
  - Inter-core on-chip
  - DRAM Off-chip
- Parallel algorithm minimizes inter-core on-chip communication
- Sequential algorithm minimizes off-chip communication
- Hierarchical blocking of the matrix and vectors
  - Minimize inter-block communication: reordering may occur
  - Cache blocks small enough to hold the matrix and vector entries in cache
- Redundant work due to parallelization (+explicit sequential algorithm)

Tuning the matrix powers kernel

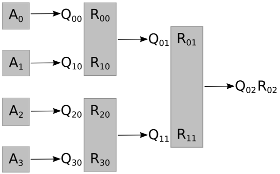
- Tuning parameters and choices:
  - Sequential algorithm: explicit/implicit
  - Explicit: using cyclic buffers or not
  - Partitioning strategy: reorder or not, # partitions
  - Solving the ordering problems
  - SpMV tuning parameters: register tile size, SW prefetch distance
- Autotuning
  - Choice of parameter values dependent on matrix structure

Outline

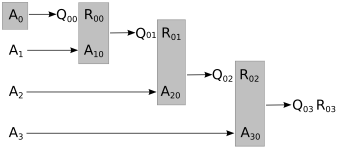
Tall skinny QR factorization

Compute the QR factorization of an $n \times (k+1)$ matrix

- "Tall skinny" matrix ($n \gg k$)
- `MPI_Reduce` with QR as the reduction operator $\Rightarrow$ only one reduction



Reduction tree for 4 processors



Reduction tree for 4 cache blocks

- Implementation uses a hybrid approach
  - Sequential reduction inside a parallel reduction

Outline

Block GRAM-Schmidt Orthogonalization

- Original MGS: orthogonalize a vector against a block of n orthogonal vectors
  - BLAS level 1 operations: dot-products
- Orthogonalize a block of k vectors against a block of n orthogonal vectors
  - BLAS level 3 operations: matrix-matrix multiplies $\Rightarrow$ better cache reuse $\Rightarrow$ better performance

Outline

CA-GMRES: Putting the pieces together

Conventional GMRES (solve for $Ax = b$)

1. **for** $i = 1$ to r
2. $\quad w = Av_{i-1}$ */* SpMV */*
3. $\quad$ Orthogonalize w against $\{v_0, \ldots, v_{i-1}\}$ */* MGS */*
4. $\quad$ Update vector v_i, matrix H
5. Use H, $\{v_0, \ldots, v_r\}$ to construct the solution

CA-GMRES: Putting the pieces together

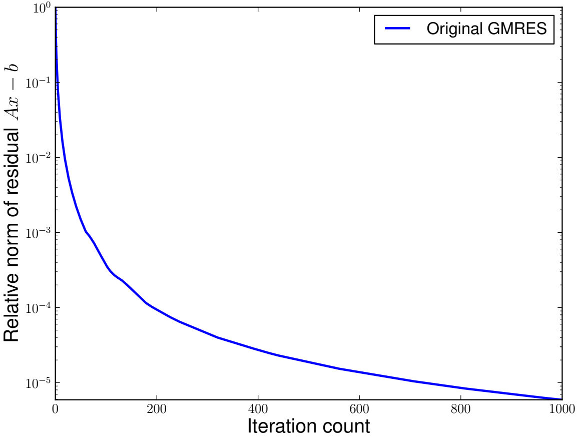
Conventional GMRES (solve for $Ax = b$)

1. **for** $i = 1$ to r
2. $\quad w = Av_{i-1}$ */* SpMV */*
3. $\quad$ Orthogonalize w against $\{v_0, \ldots, v_{i-1}\}$ */* MGS */*
4. $\quad$ Update vector v_i, matrix H
5. Use H, $\{v_0, \ldots, v_r\}$ to construct the solution

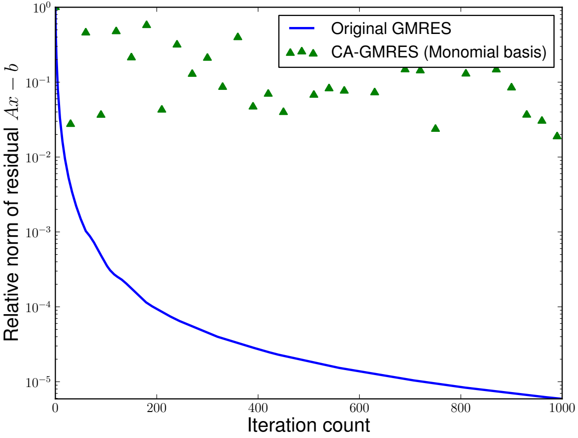
CA-GMRES (Communication-Avoiding GMRES)

1. **for** $i = 0, k, 2k, \ldots, k(t-1)$ */* Outer iterations: $t = r/k$ */*
2. $\quad W = \{Av_i, A^2 v_i, \ldots, A^k v_i\}$ */* Matrix powers */*
3. $\quad$ Make W orthogonal against $\{v_0, \ldots, v_i\}$ */* Block GS */*
4. $\quad$ Make W orthogonal */* TSQR */*
5. $\quad$ Update $\{v_{i+1}, \ldots, v_{i+k}\}$, H
6. Use H, $\{v_0, v_1, \ldots, v_{kt}\}$ to construct the solution

Does CA-GMRES converge?

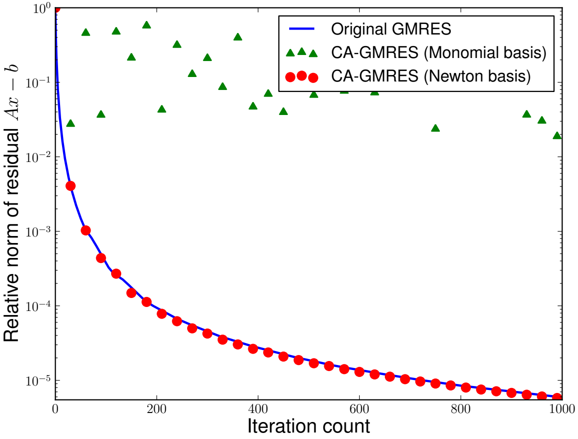


Does CA-GMRES converge?



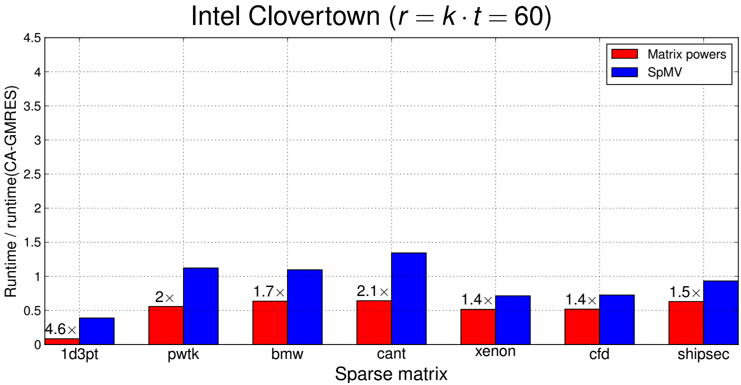
- Monomial basis: matrix powers kernel computes $[Ax, A^2x, \ldots, A^kx]$

Does CA-GMRES converge?



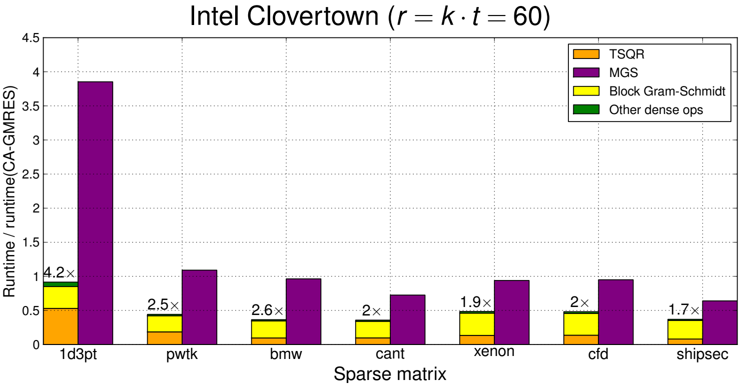
- Monomial basis: matrix powers kernel computes $[Ax, A^2x, \ldots, A^kx]$
- Newton basis: matrix powers kernel computes $[(A - \lambda_1 I)x, (A - \lambda_2 I)(A - \lambda_1 I)x, \ldots, (A - \lambda_k I) \cdots (A - \lambda_1 I)x]$

Speedups over conventional GMRES: Sparse kernel



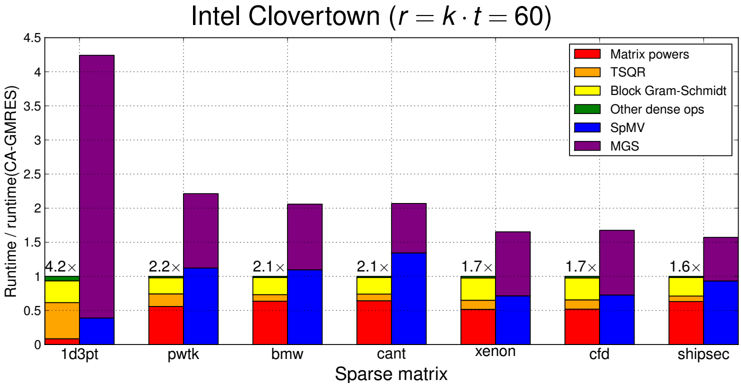
- Sparse: median speedup of 1.7×

Speedups over conventional GMRES: Dense kernels



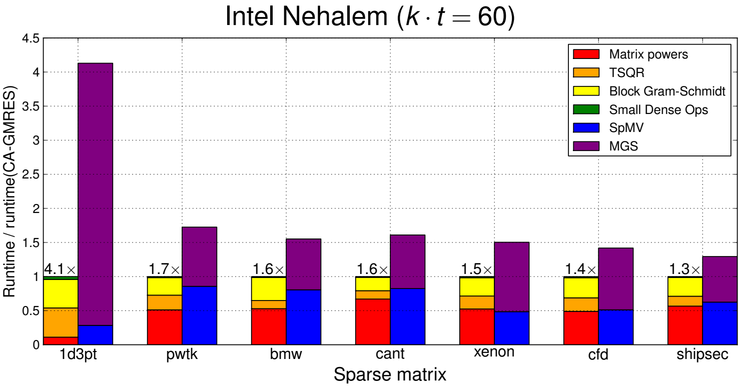
- Dense: median speedup of 2×

Overall speedups over conventional GMRES



- Overall: medial speedup of 2.1×

Overall speedups over conventional GMRES



- Median speedup of 1.6×
- More available bandwidth ⇒ speedups lower

Outline

Conclusions/Future work

- Implemented a communication-avoiding solver using three new kernels
  - Amortized reading matrix over multiple iterations
  - Built on prior work, introduced new algorithms for modern multicores, auto-tuned implementation
  - Achieve 2.1$\times$ median speedup on Intel Clovertown and 1.6$\times$ median speedup on Intel Nehalem
- Implication for HW design: communication-avoiding $\Rightarrow$ lower bandwidth $\Rightarrow$ lower cost

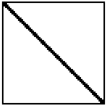
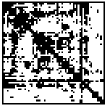
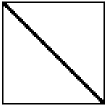
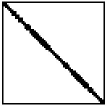
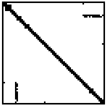
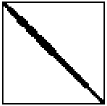
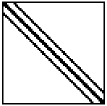
Conclusions/Future work

- Implemented a communication-avoiding solver using three new kernels
  - Amortized reading matrix over multiple iterations
  - Built on prior work, introduced new algorithms for modern multicores, auto-tuned implementation
  - Achieve 2.1$\times$ median speedup on Intel Clovertown and 1.6$\times$ median speedup on Intel Nehalem
- Implication for HW design: communication-avoiding $\Rightarrow$ lower bandwidth $\Rightarrow$ lower cost
- Future work:
  - Extending to distributed memory implementations
  - Extensions to other iterative solvers
  - Add preconditioning
  - Incorporate TSP solver to solve the ordering problems
  - Autotuning compositions of kernels

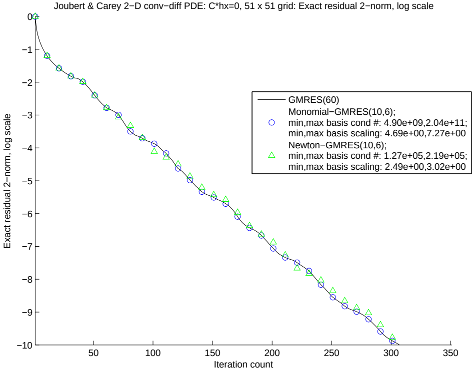
Contributions

- High performance implementations and co-tuning of all relevant kernels on multicore
  - Simultaneous optimizations to reduce parallel and sequential communication
- New algorithm allows independent choice of restart length r and kernel size k
  - Prior work required $r = k$, but want $k \ll r$ in most cases
- Showed how to incorporate preconditioning
  - Still need to implement
- See paper for lots of references on prior work
- **Questions?**

Sparse Matrices

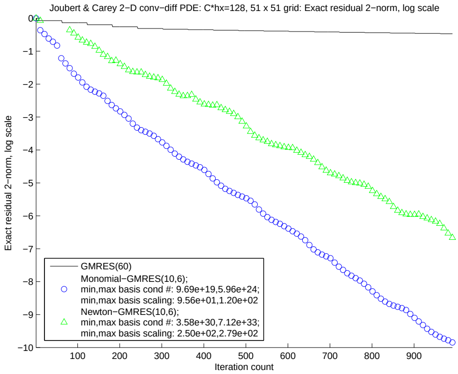
 **1d3pt** Tridiagonal matrix (1M, 3M, 3)	 **bmw** Stiffness matrix (141K, 7.3M, 51)	 **cant** FEM cantilever (62K, 4M, 65)	 **cfd** Pressure matrix (123K, 3.1M, 25)
 **pwtk** Pressurized wind tunnel stiffness matrix (218K, 12M, 55)	 **shipsec** FEM ship section/detail (141K, 7.8M, 55)	 xenon Complex zeolite, sodalite crystals (157K, 3.9M, 25)	

Example 1: CA-GMRES same as standard GMRES



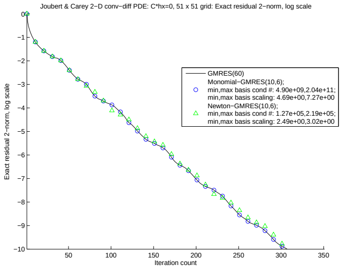
- Discretized $-\Delta u = f$ in $[0,1]^2$
- CA-GMRES w/ any basis converges as fast as standard (restarted) GMRES, but. . .

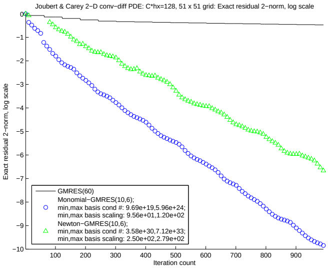
Example 2: CA-GMRES beats standard GMRES



- Added a Cu_x convection term to the PDE
- CA-GMRES beats standard restarted GMRES!

CA-GMRES may be better than GMRES





- Previous metric for success: CA-GMRES = GMRES
- For some problems, CA-GMRES converges faster
- Future work: investigate and control this phenomenon