Fast Implementations of the Akx Kernel

Marghoob Mohiyuddin Mark Hoemmen James Demmel
Katherine Yelick

Department of Electrical Engineering and Computer Science
University of California at Berkeley

SIAM Conference on Parallel Processing for Scientific Computing, 2008

Outline

Takeaways

- Sequential and parallel algorithms for the Akx kernel with minimum communication.
- Optimal speedups possible.
- Measured speedups of 3x for the sequential algorithm.

Communication is Costly, Computation is Cheap

- Gap between computational capability and communication cost increasing.
- Applications need to be designed with this gap in mind
  - Communication hiding not enough.
    - Latency can be dealt with by overlap, but limited by the amount of computation
- Communication avoiding:
  - Trade off communication with computation.

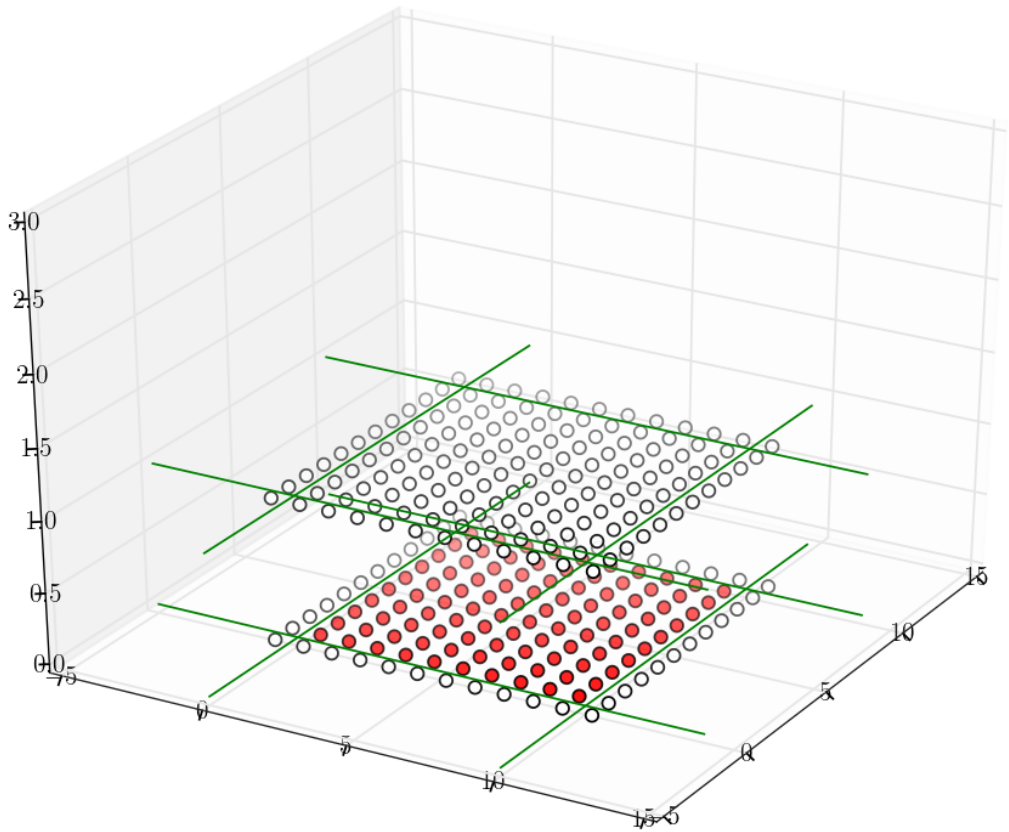
The Akx Kernel

- Given $n \times n$ sparse matrix A, vector x, integer $k > 0$,
- Compute the k vectors $Ax, A^2x, \ldots, A^kx$ efficiently.
  - Parallel and sequential algorithms.
- Arises in Krylov Subspace Methods.
  - Unpreconditioned KSMs need to look at the Krylov subspace spanned by $[x, Ax, A^2x, \ldots, A^kx]$.

Setup

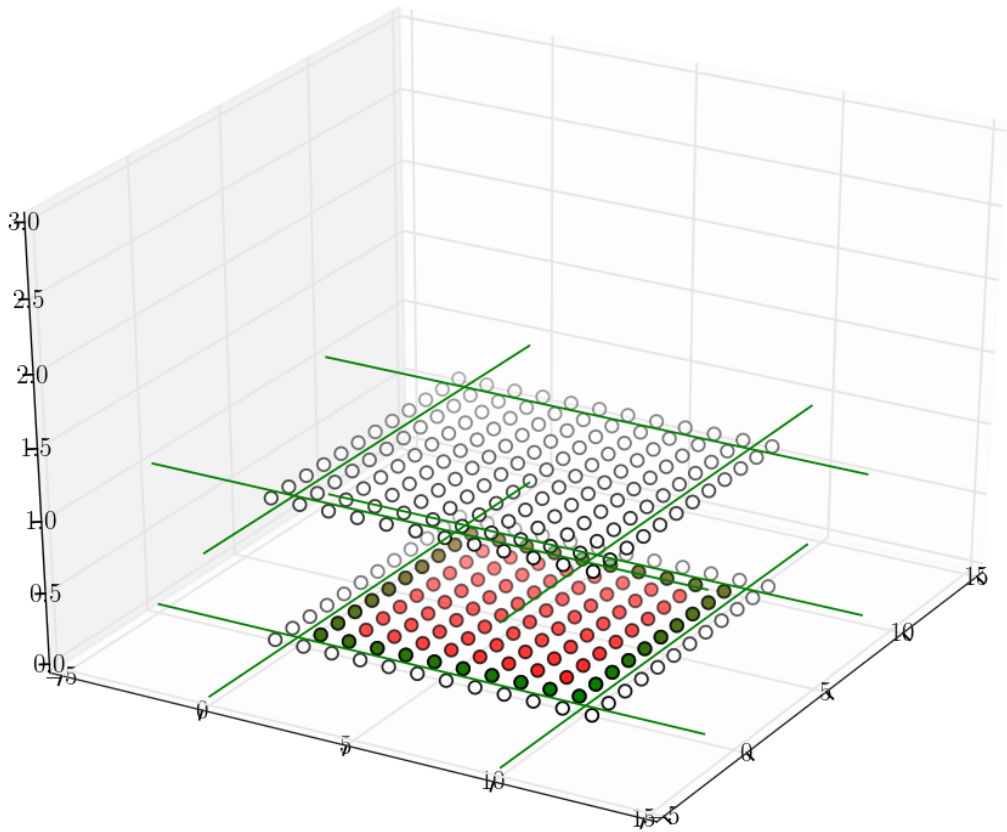
- Matrix A and vector x divided in to p blocks.
- Parallel machine:
  - Each proc. operates on a separate block.
  - Interproc. communication for remote dependencies.
- Sequential machine:
  - Each block stored contiguously in slow memory.
  - Algorithm operates on a block-by-block basis.
- Output vectors computed on a per block basis.

Naïve Algorithm: 9-pt Operator for 3 iterations



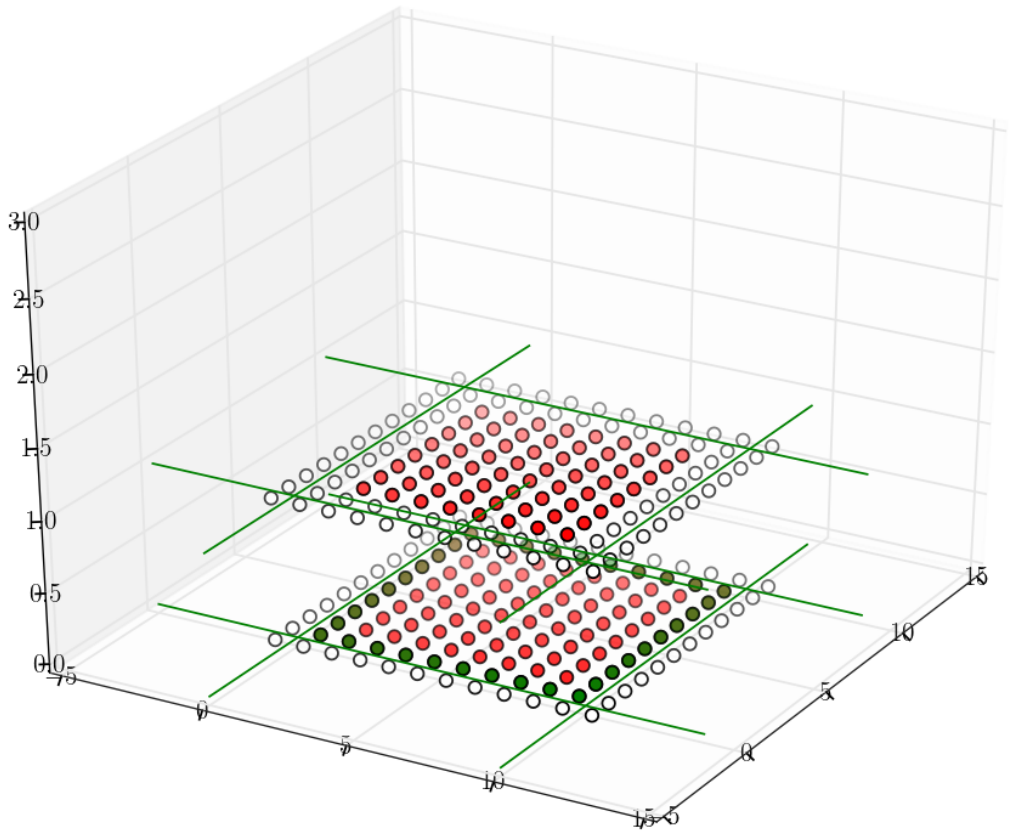
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



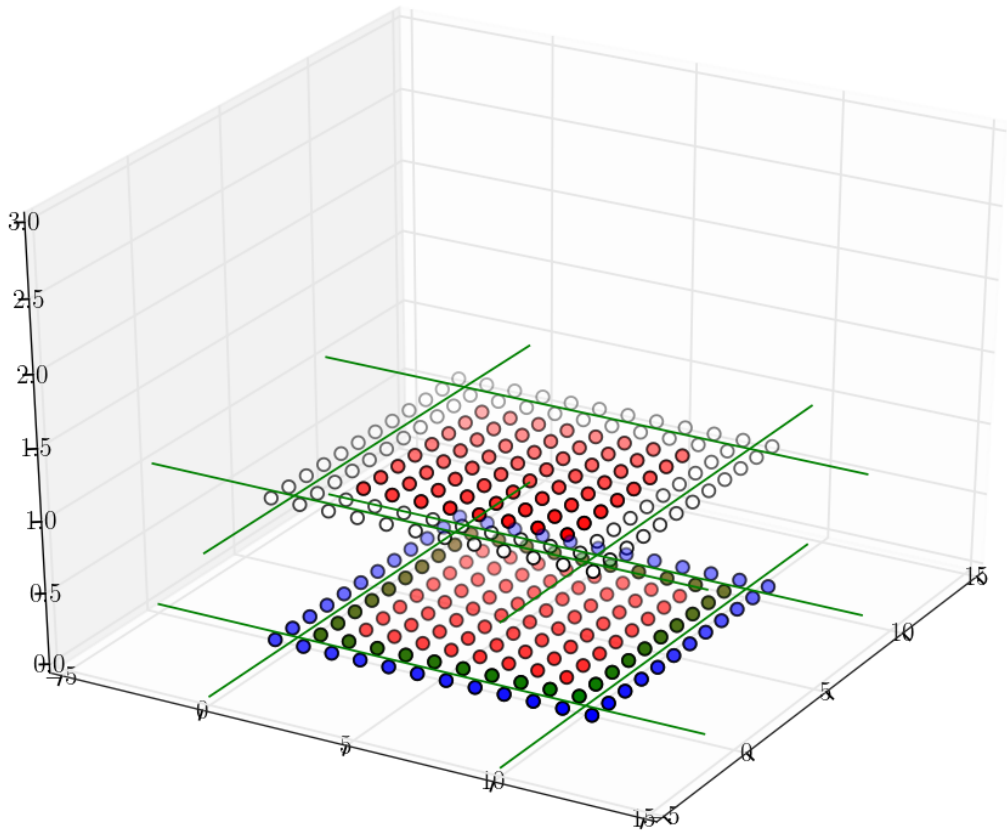
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



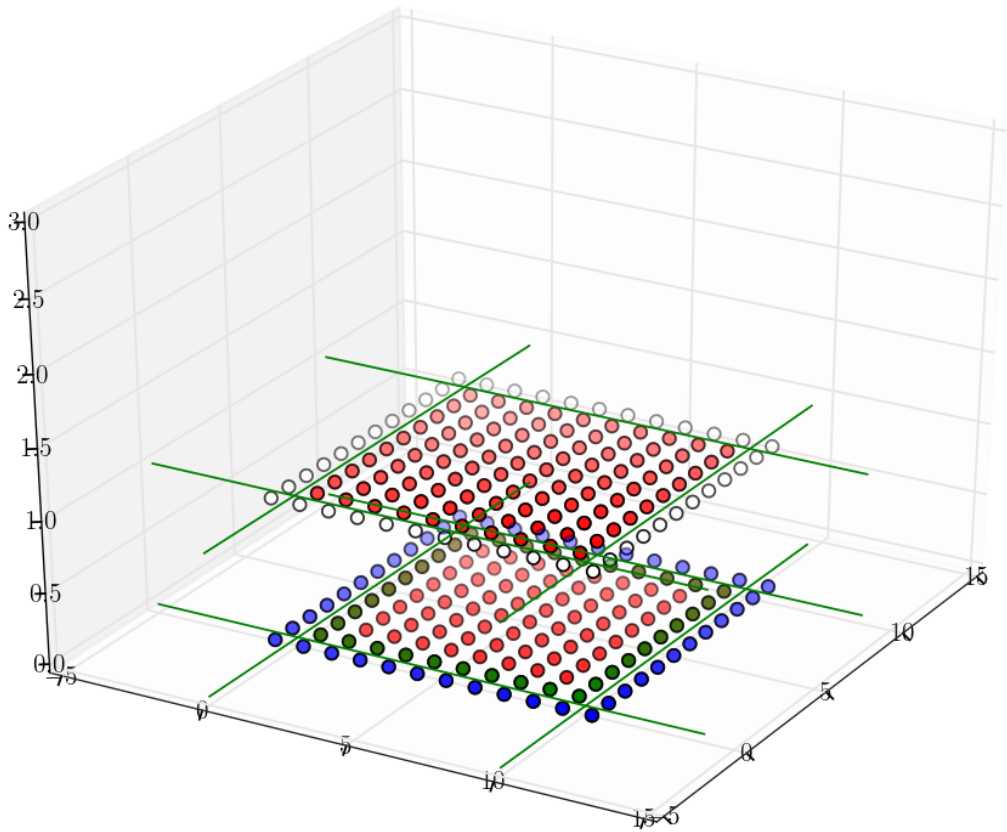
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



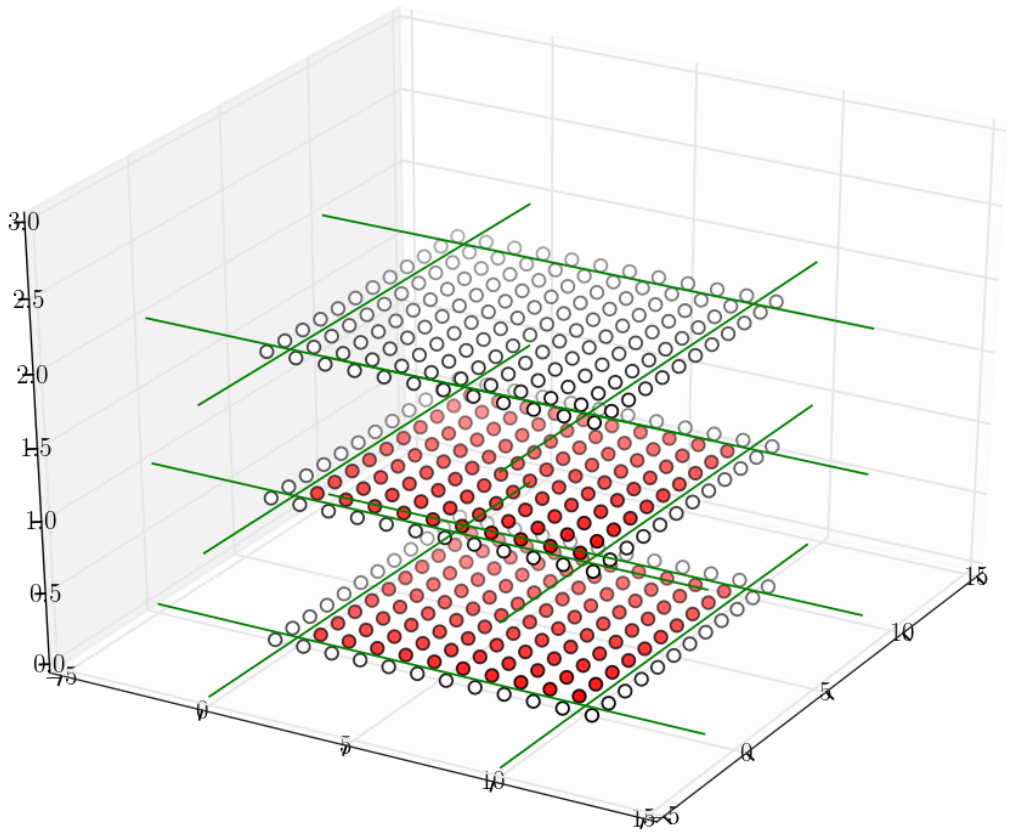
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



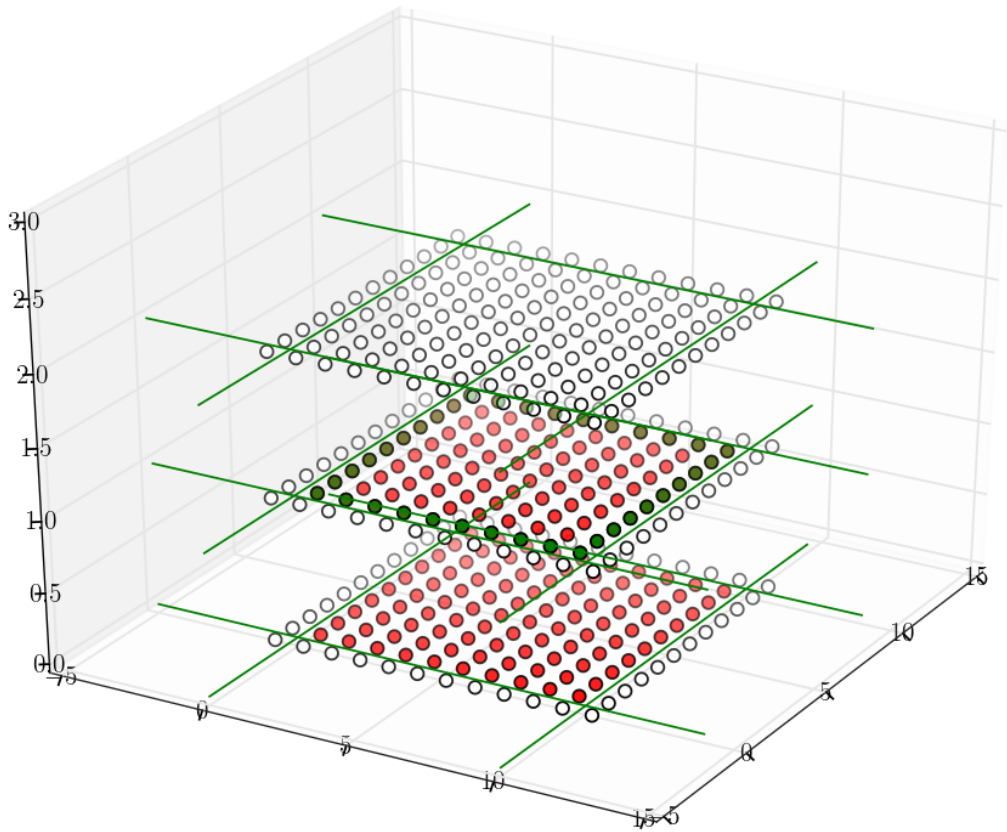
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



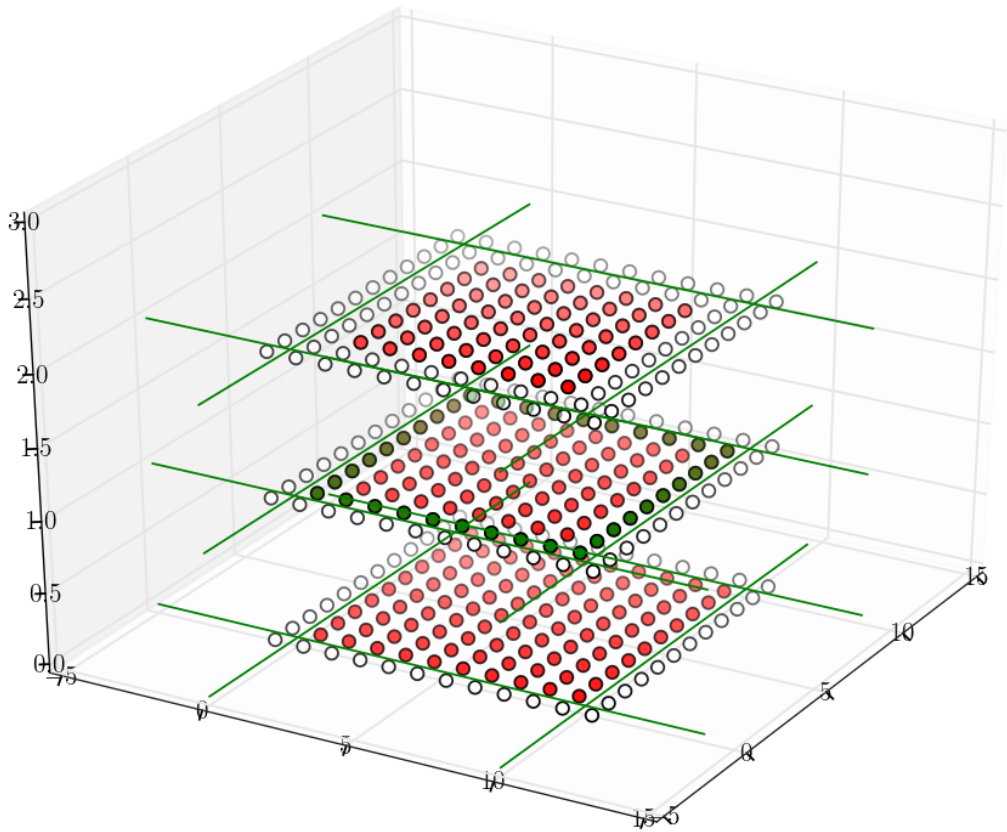
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



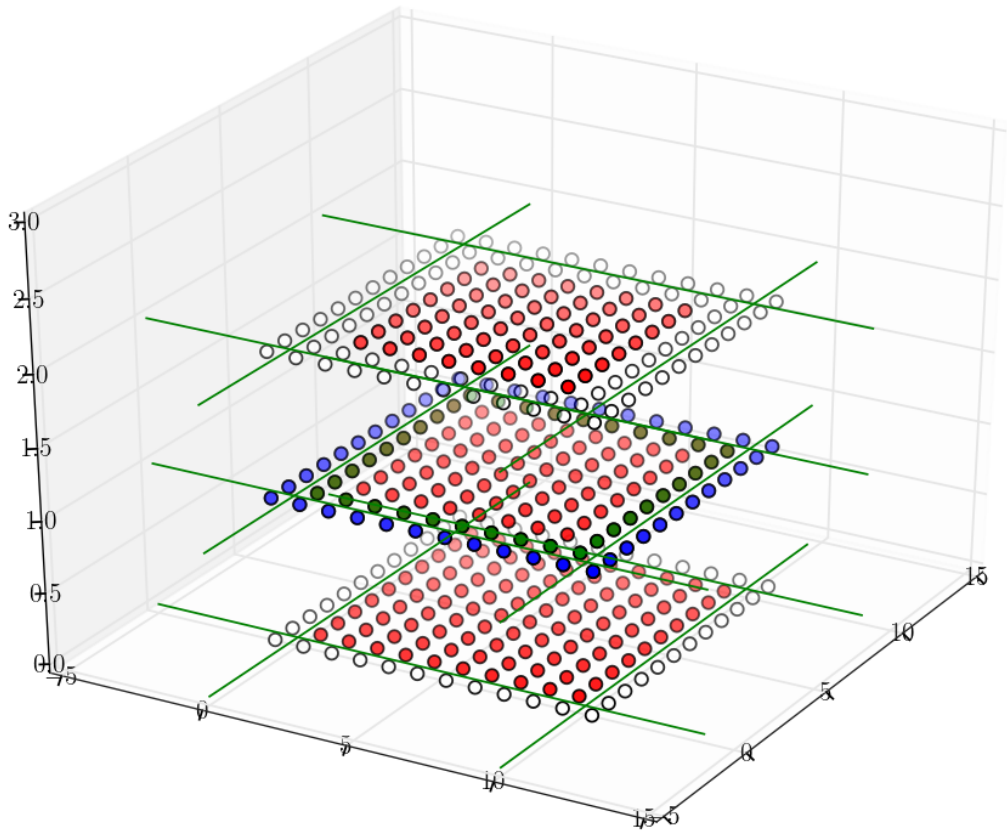
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



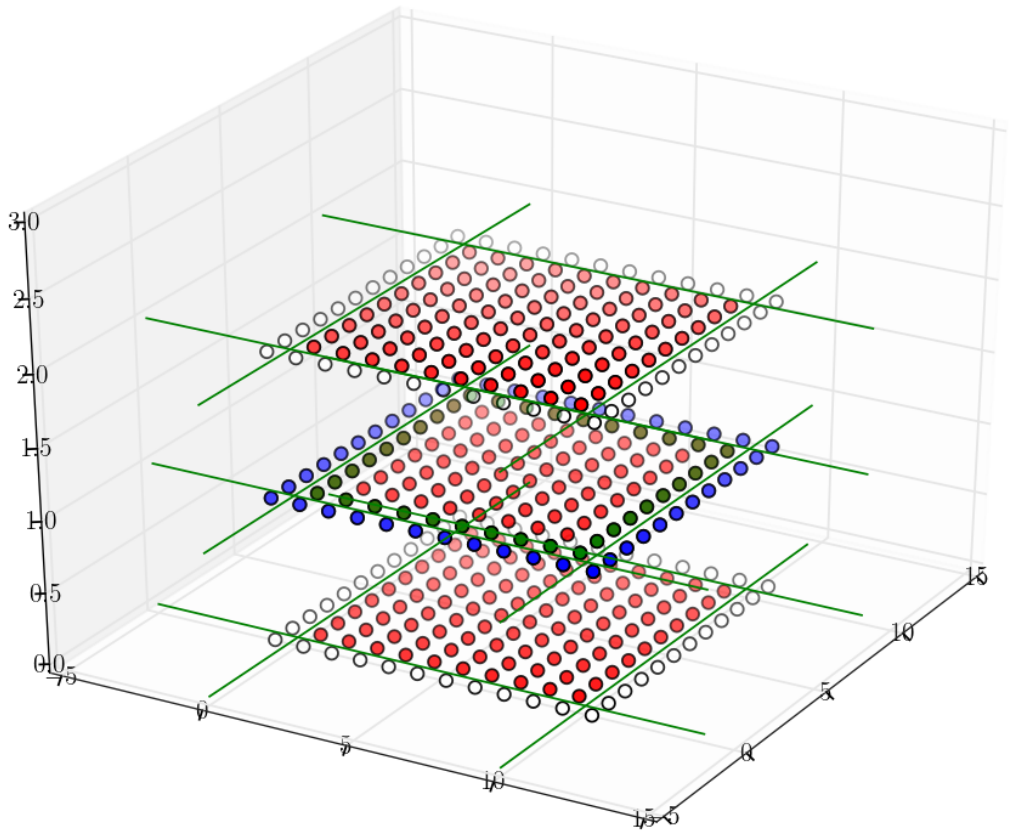
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



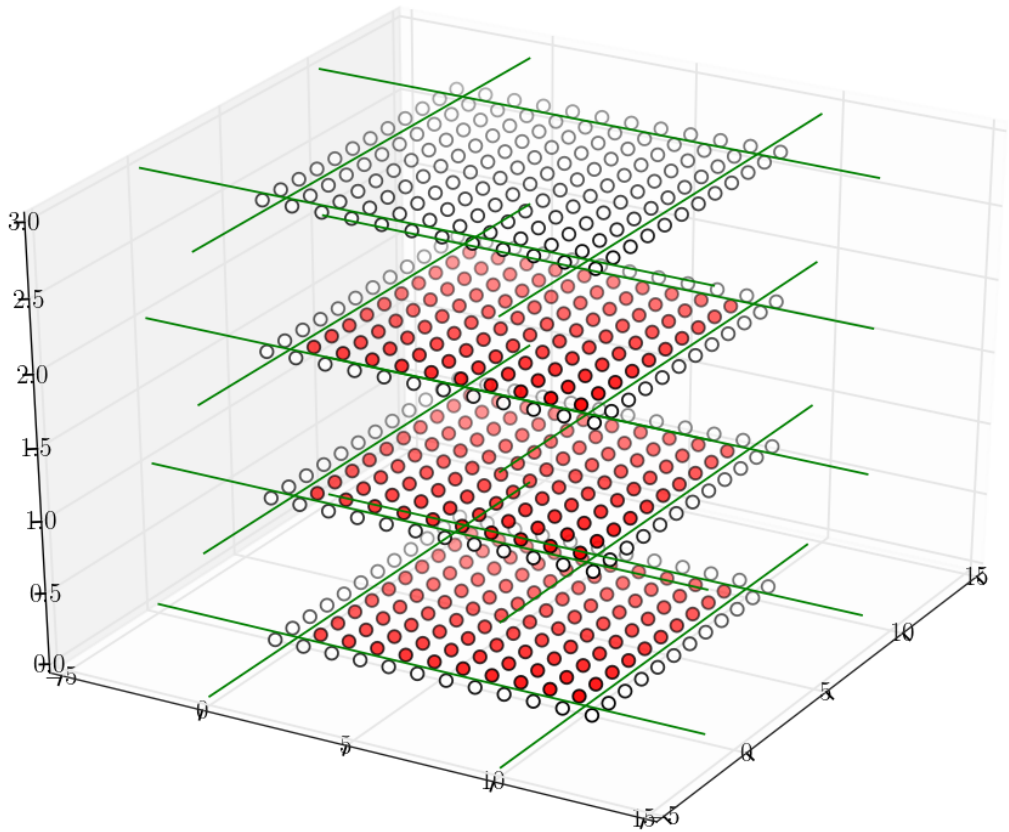
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



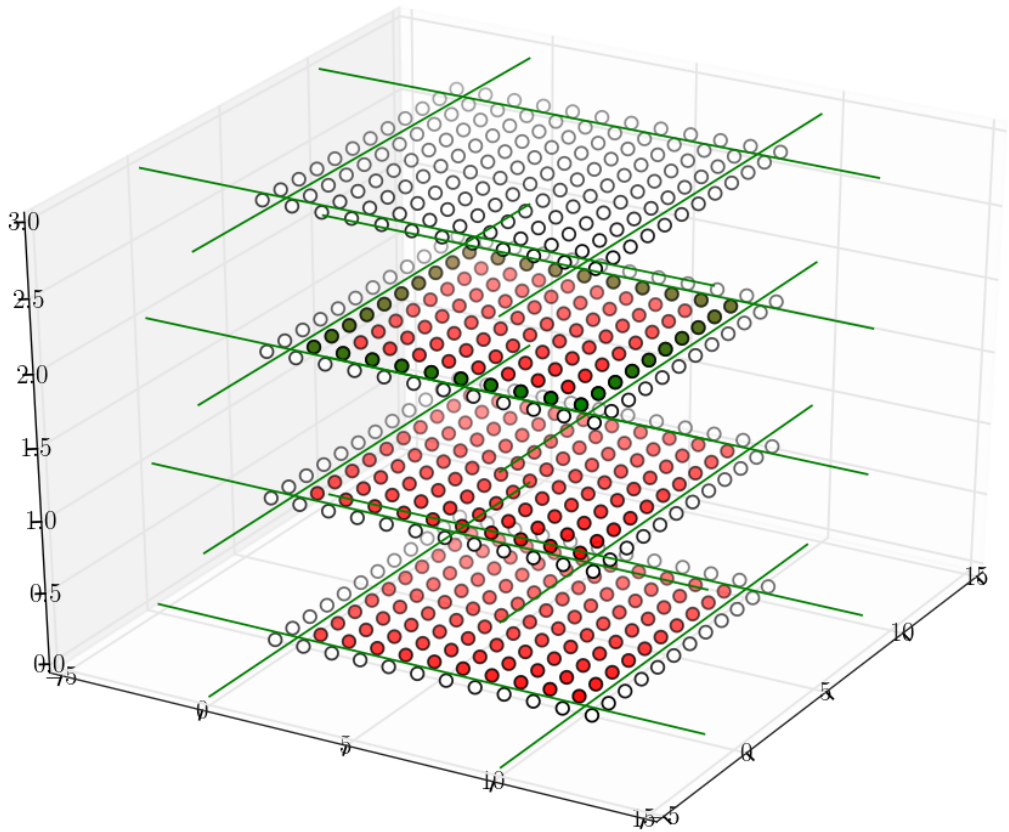
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



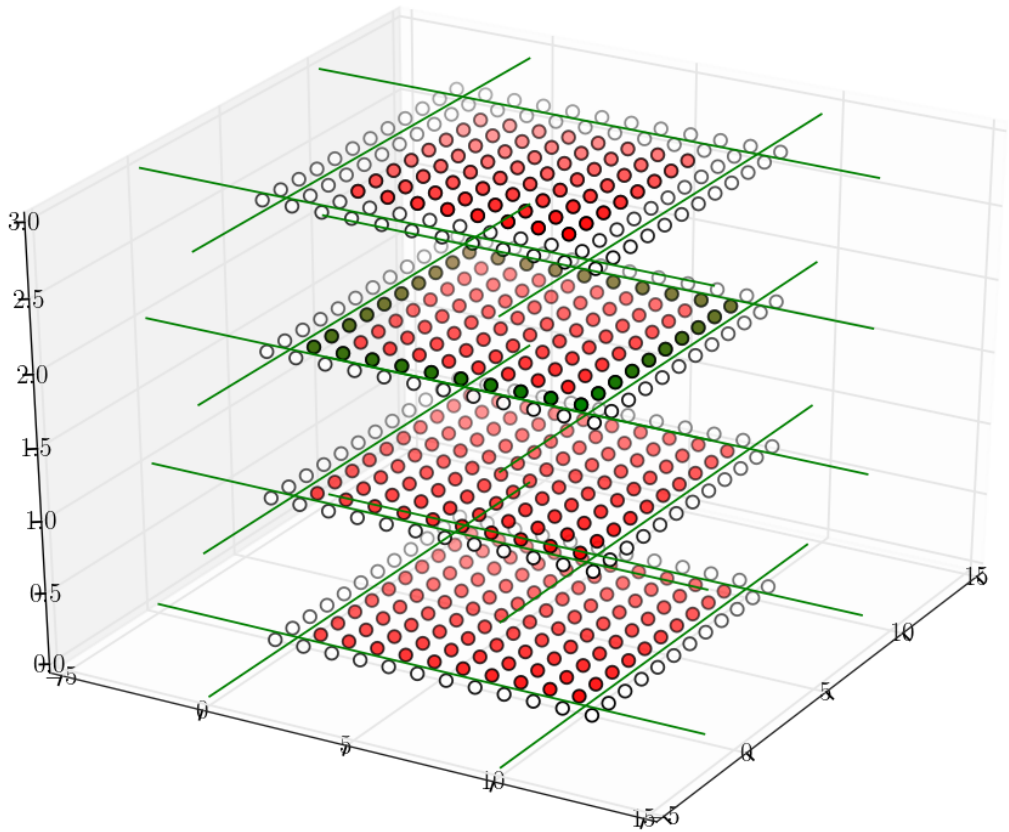
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



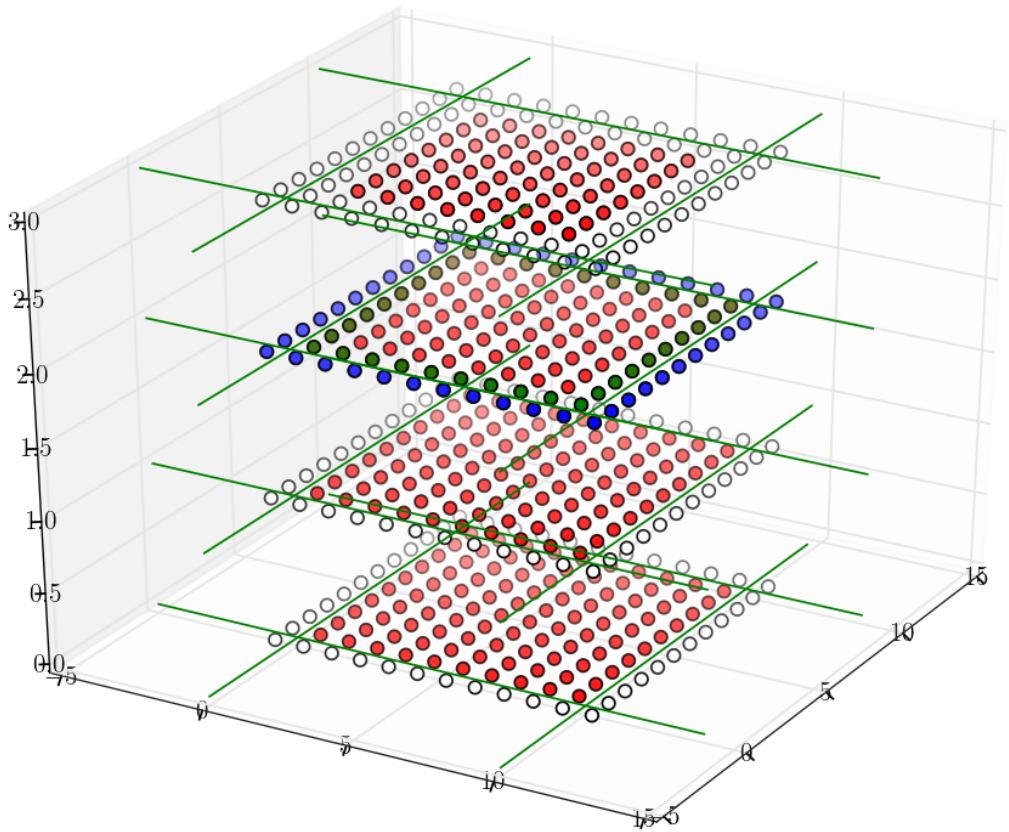
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



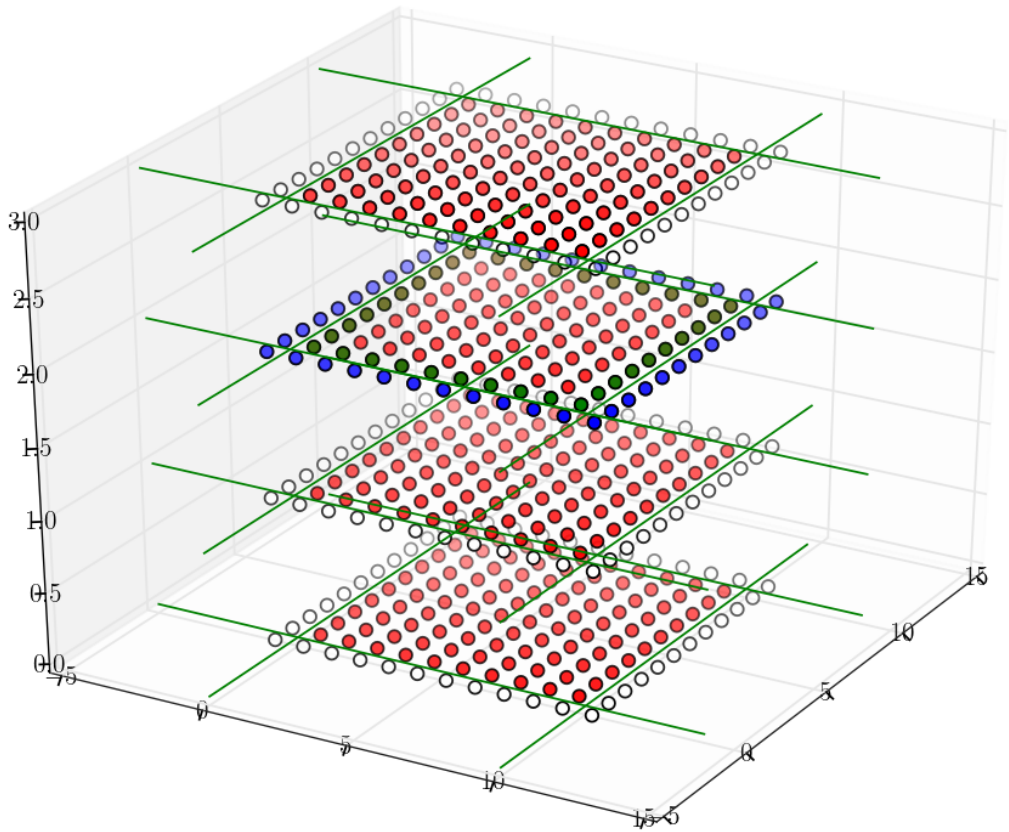
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



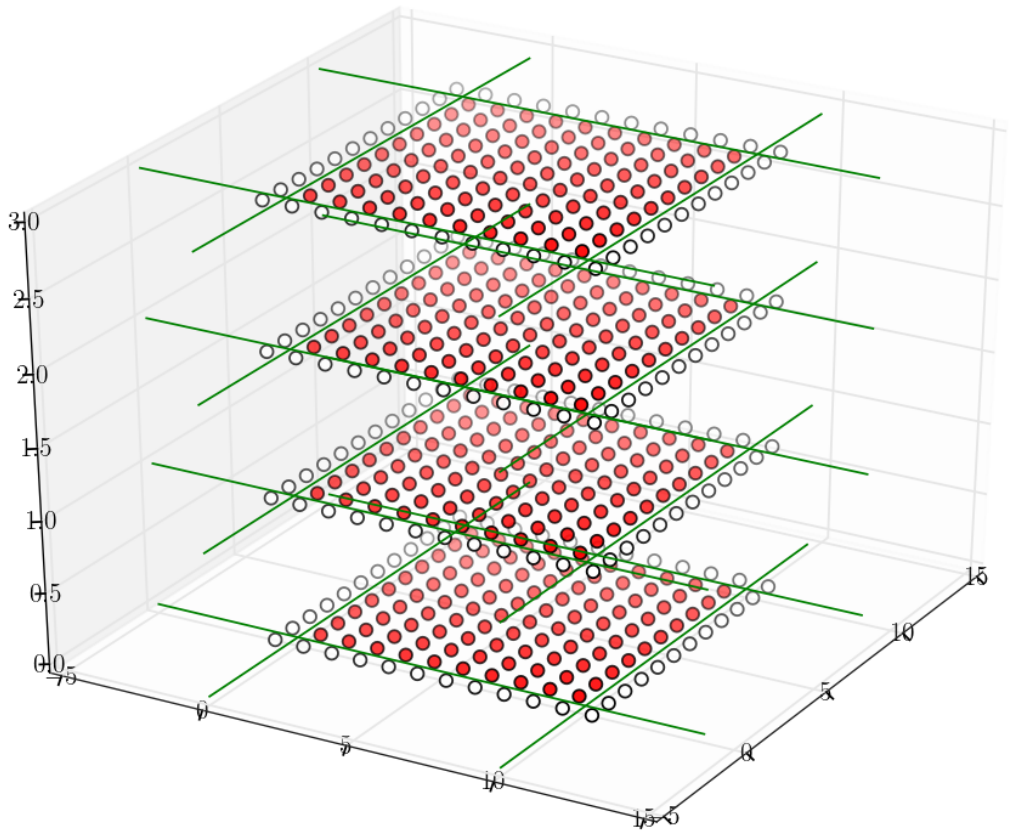
1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm: 9-pt Operator for 3 iterations



1. Entries available marked red.
2. Computing Ax: Send entries needed by other procs (green).
3. Computing Ax: Compute locally dependent entries.
4. Computing Ax: Receive entries from other procs (blue).
5. Computing Ax: Compute remaining entries of Ax.
6. Compute A^2x as $A(Ax)$.
7. Compute A^3x as $A(A^2x)$.

Naïve Algorithm

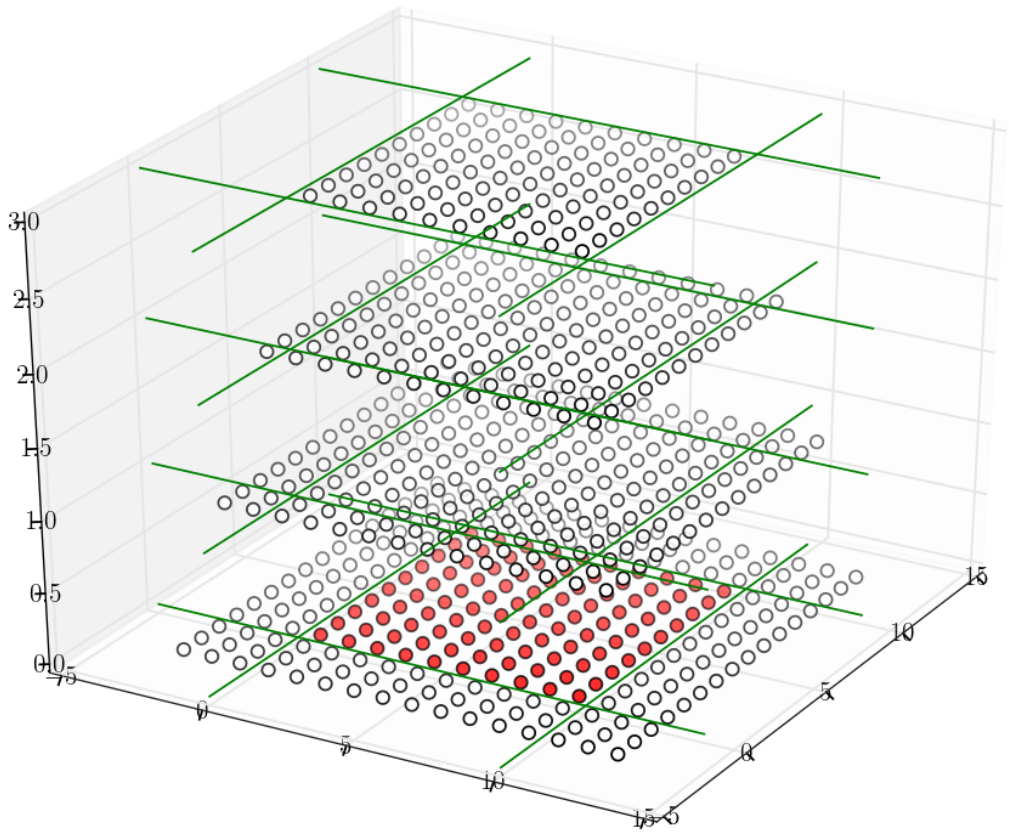
For $i = 1, \ldots, k$,

Compute $A^i x$ using the product of A and $A^{i-1}x$

- Fetch ghost zones of x from other procs/blocks.

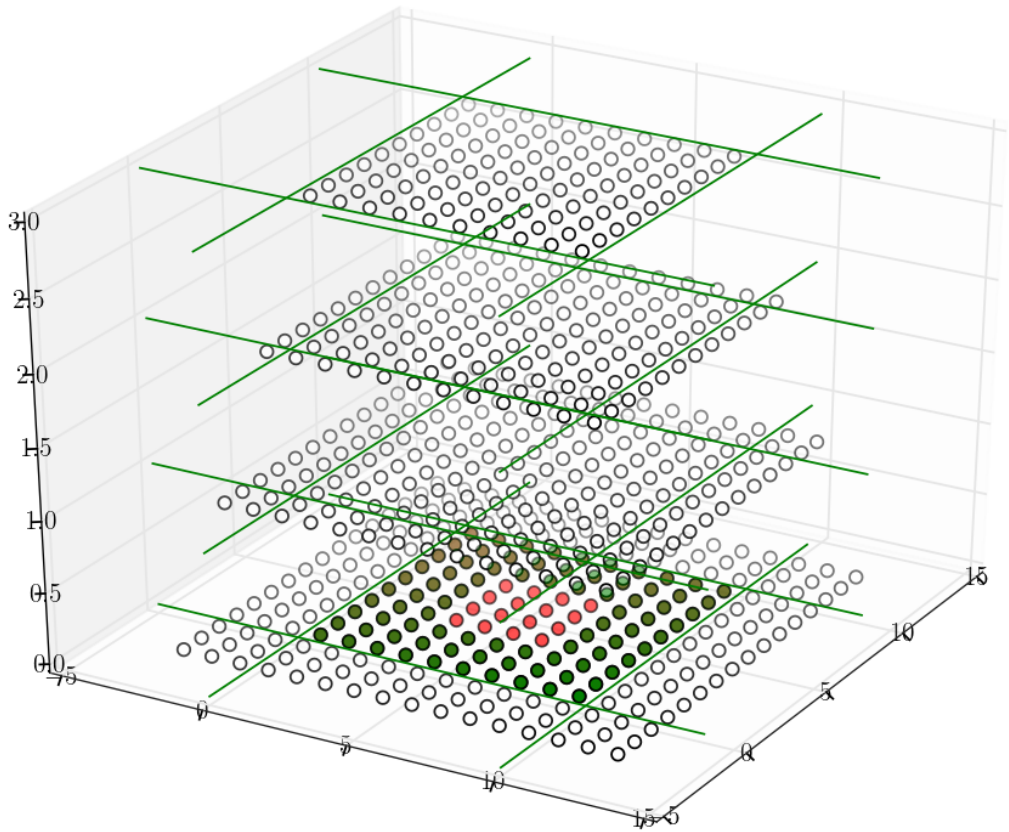
- $O(k)$ messages between any two procs/blocks.
  - Latency cost is k times the minimum.
  - Objective: O(1) messages between any 2 blocks/procs.
- Sequential machine: A and x read k times from slow to fast memory.
  - Bandwidth cost is k times the minimum.
  - Objective: Read A and x at most once.
- Minimum number of floating-point operations performed.
  - Objective: Minimize number of extra flops.

Improving Naïve Algorithm: 9-point operator for 3 iterations



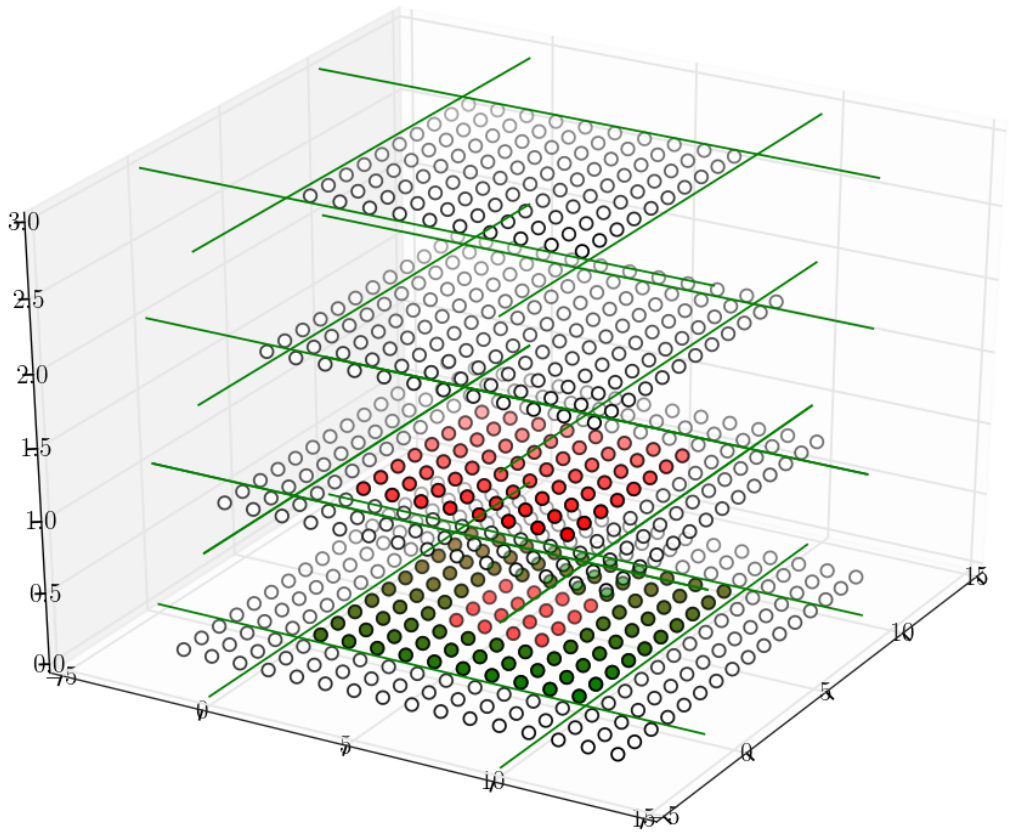
1. Entries available marked red.
2. Send entries needed by other procs.
3. Compute locally computable entries.
4. Receive entries from other procs (blue).
5. Compute remotely dependent entries.

Improving Naïve Algorithm: 9-point operator for 3 iterations



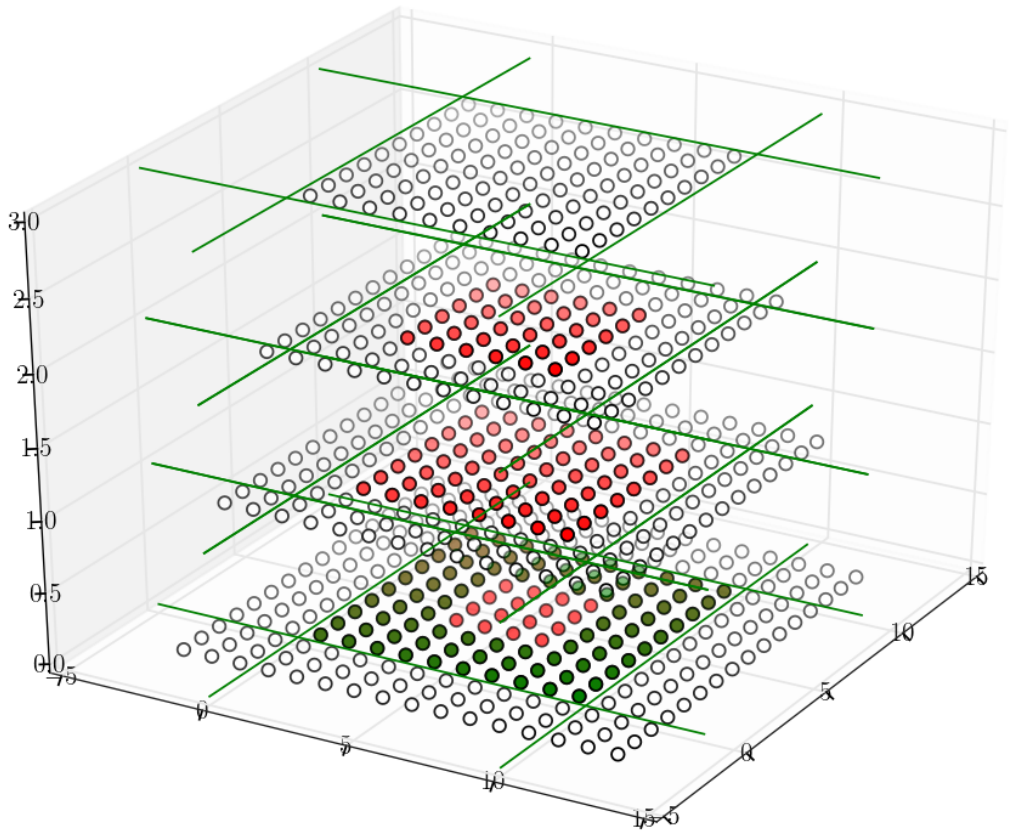
1. Entries available marked red.
2. Send entries needed by other procs.
3. Compute locally computable entries.
4. Receive entries from other procs (blue).
5. Compute remotely dependent entries.

Improving Naïve Algorithm: 9-point operator for 3 iterations



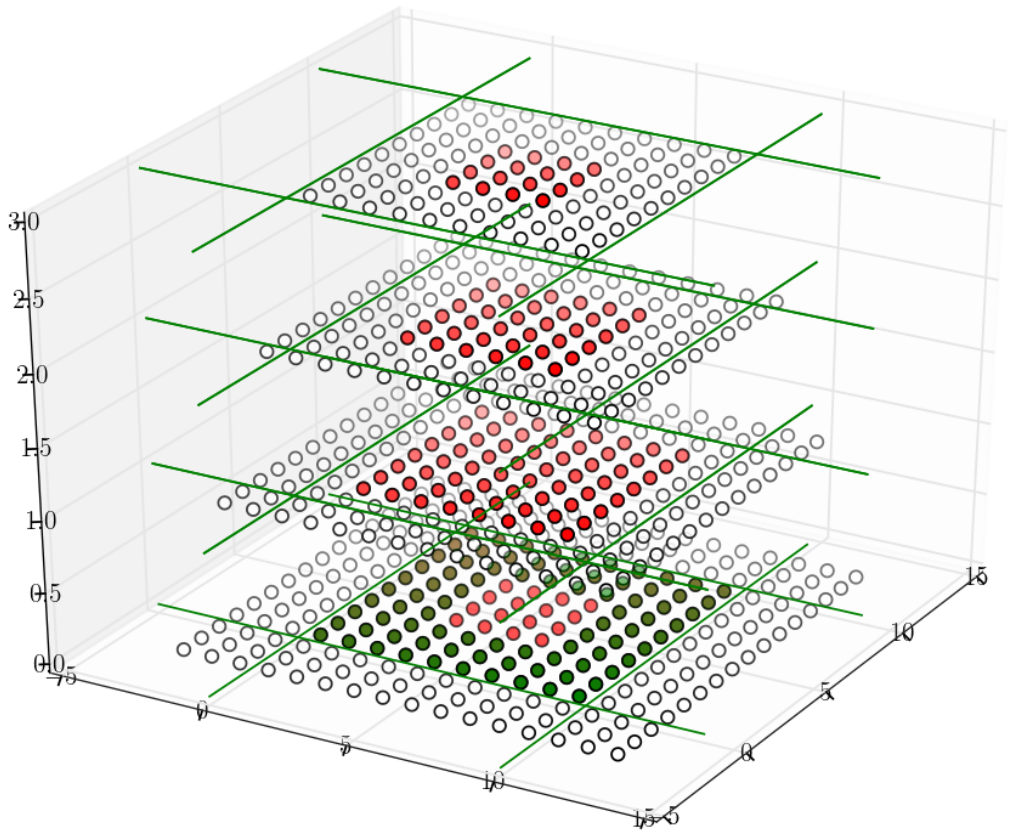
1. Entries available marked red.
2. Send entries needed by other procs.
3. Compute locally computable entries.
4. Receive entries from other procs (blue).
5. Compute remotely dependent entries.

Improving Naïve Algorithm: 9-point operator for 3 iterations



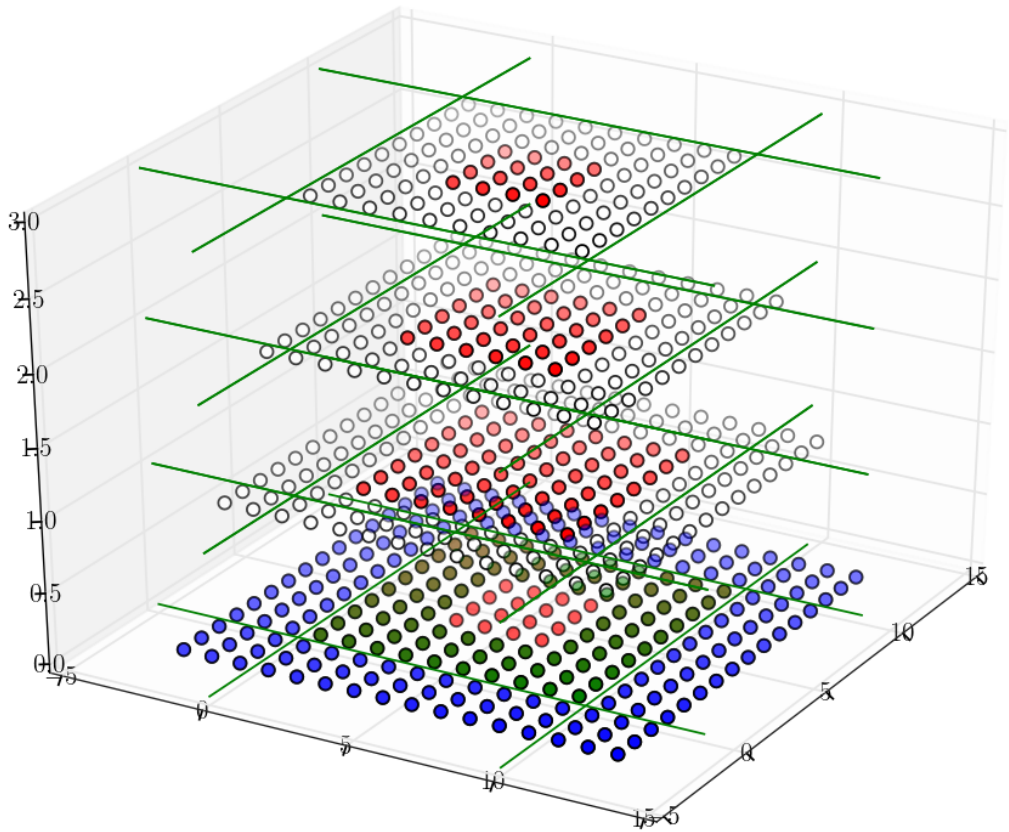
1. Entries available marked red.
2. Send entries needed by other procs.
3. Compute locally computable entries.
4. Receive entries from other procs (blue).
5. Compute remotely dependent entries.

Improving Naïve Algorithm: 9-point operator for 3 iterations



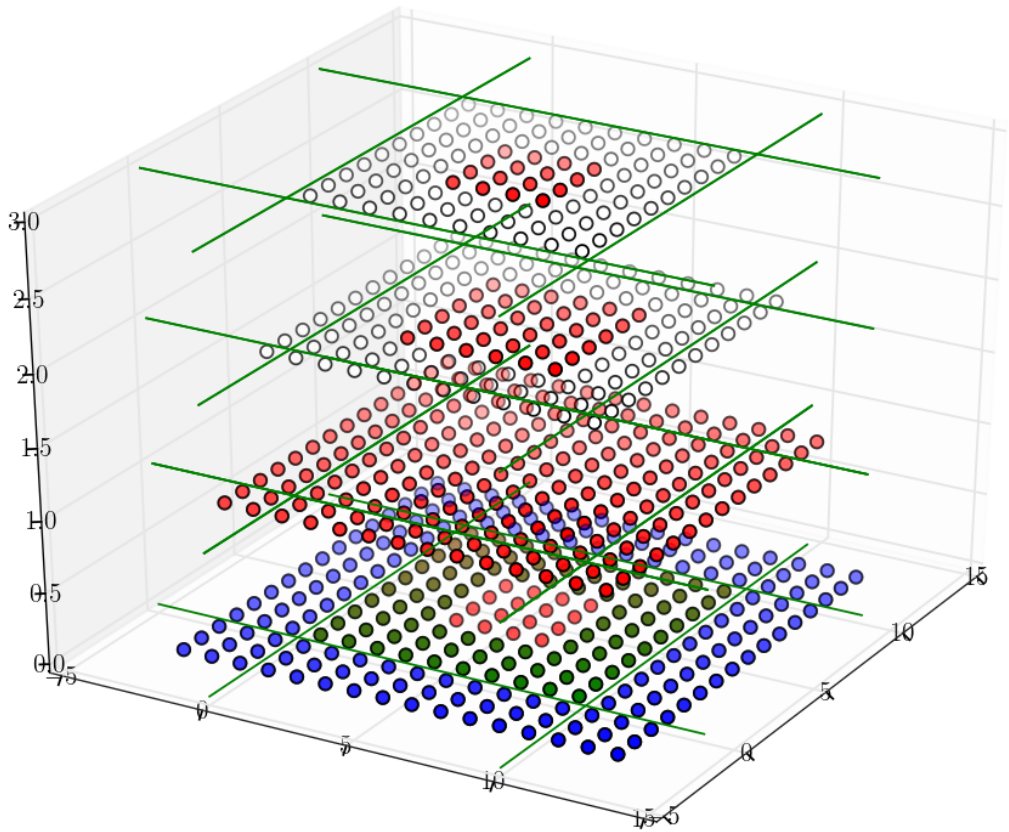
1. Entries available marked red.
2. Send entries needed by other procs.
3. Compute locally computable entries.
4. Receive entries from other procs (blue).
5. Compute remotely dependent entries.

Improving Naïve Algorithm: 9-point operator for 3 iterations



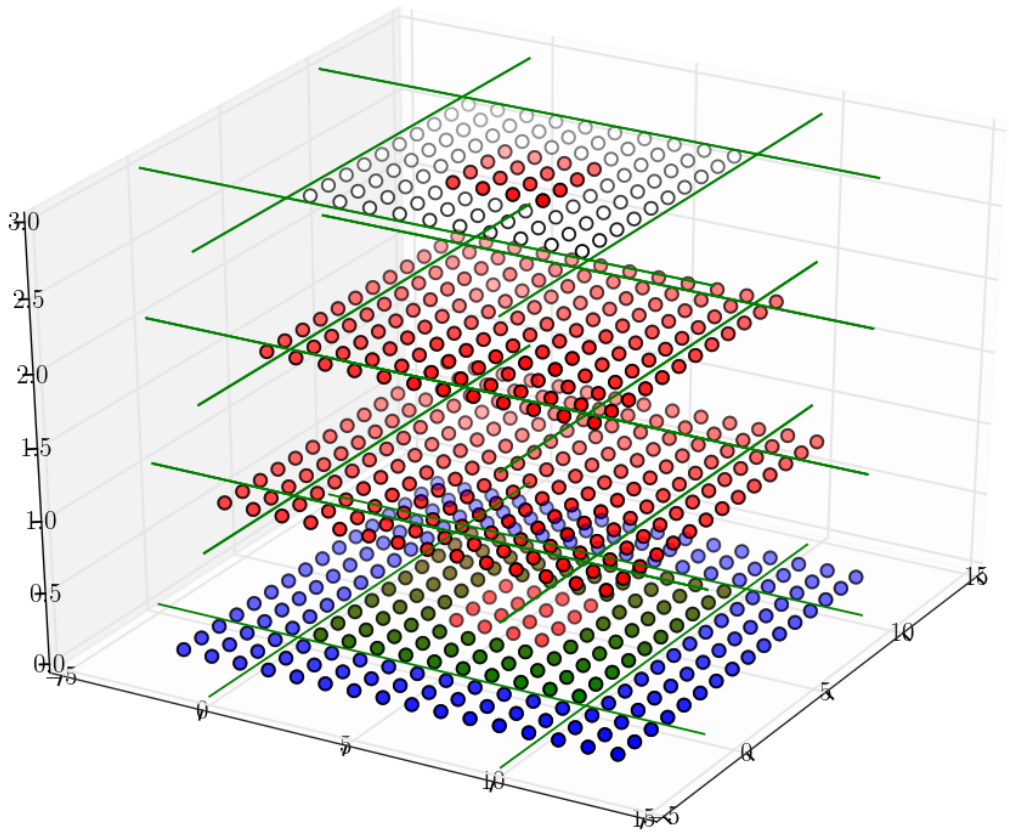
1. Entries available marked red.
2. Send entries needed by other procs.
3. Compute locally computable entries.
4. Receive entries from other procs (blue).
5. Compute remotely dependent entries.

Improving Naïve Algorithm: 9-point operator for 3 iterations



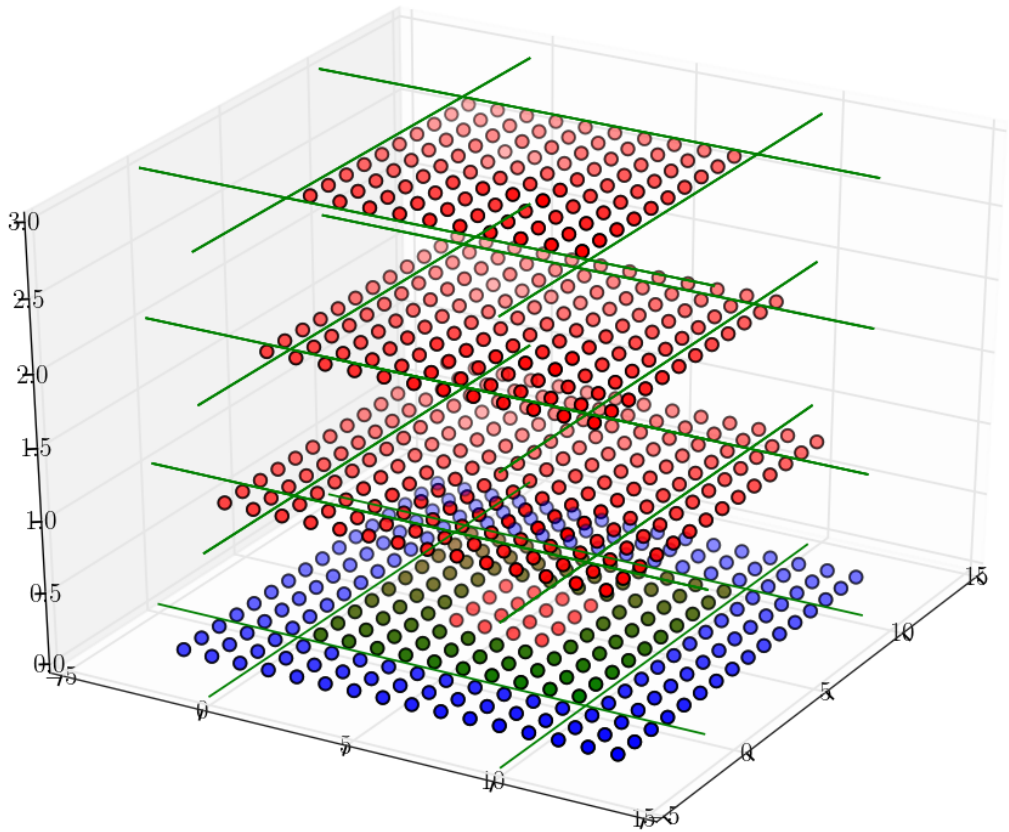
1. Entries available marked red.
2. Send entries needed by other procs.
3. Compute locally computable entries.
4. Receive entries from other procs (blue).
5. Compute remotely dependent entries.

Improving Naïve Algorithm: 9-point operator for 3 iterations



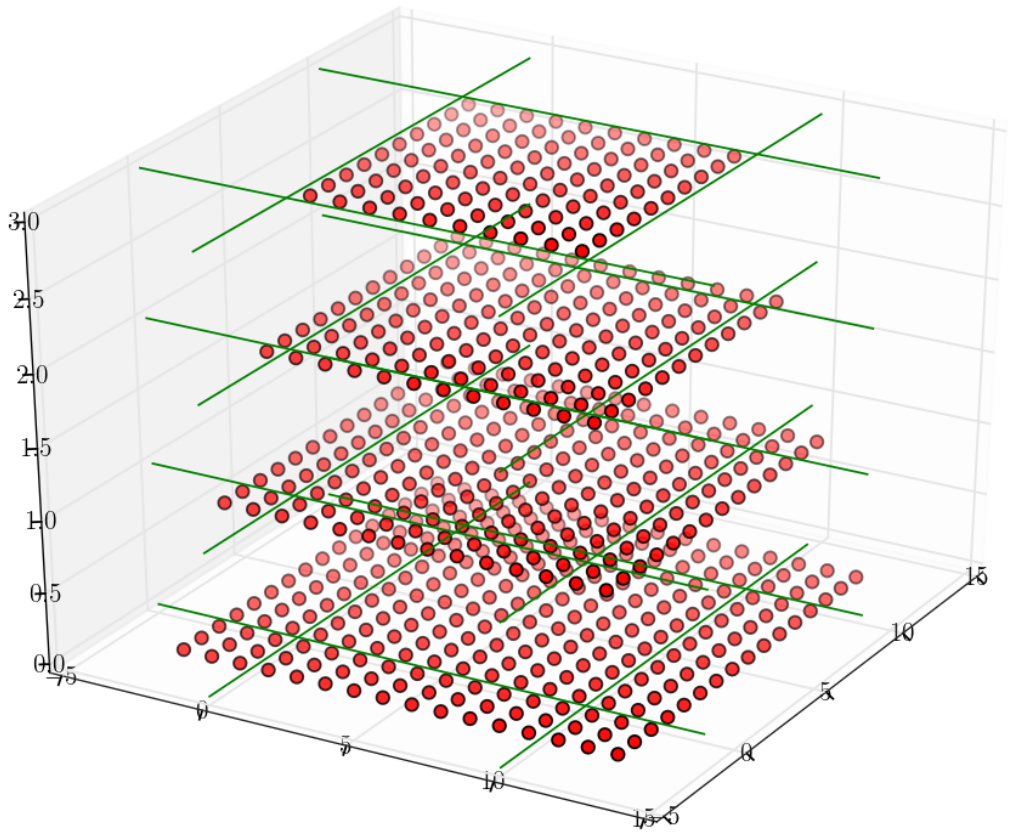
1. Entries available marked red.
2. Send entries needed by other procs.
3. Compute locally computable entries.
4. Receive entries from other procs (blue).
5. Compute remotely dependent entries.

Improving Naïve Algorithm: 9-point operator for 3 iterations



1. Entries available marked red.
2. Send entries needed by other procs.
3. Compute locally computable entries.
4. Receive entries from other procs (blue).
5. Compute remotely dependent entries.

Improving Naïve Algorithm: 9-point operator for 3 iterations



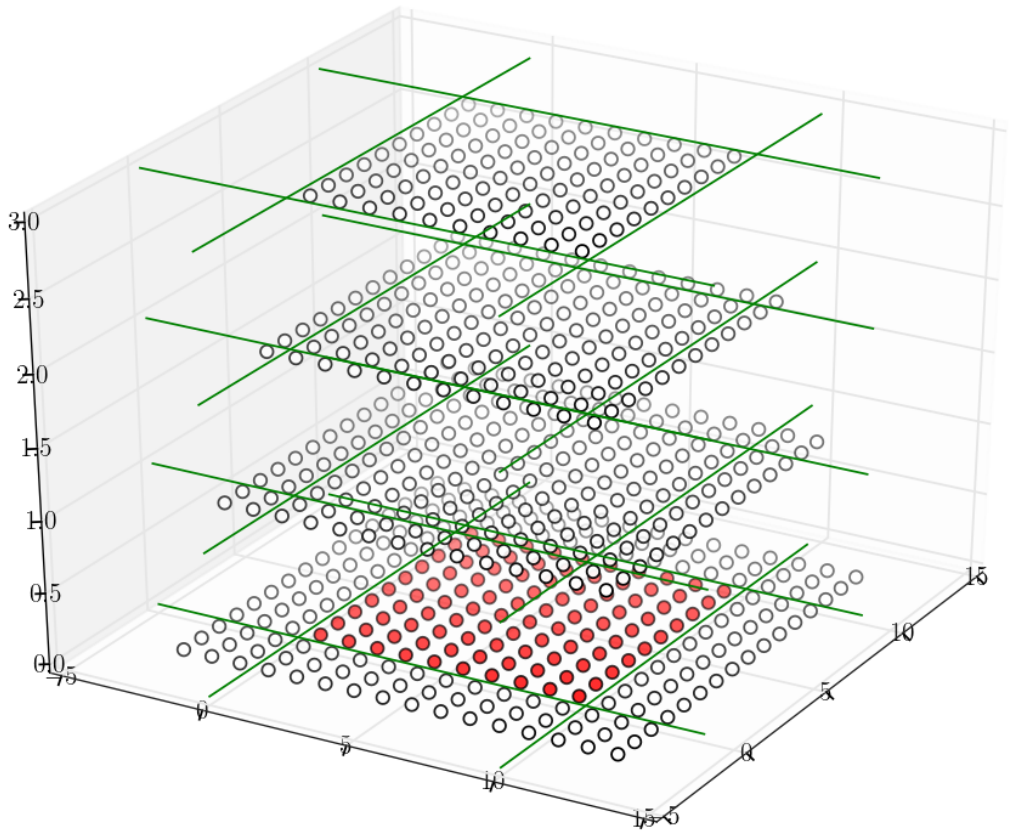
1. Entries available marked red.
2. Send entries needed by other procs.
3. Compute locally computable entries.
4. Receive entries from other procs (blue).
5. Compute remotely dependent entries.

PA1 (code for proc. i)

1. The required entries of x are computed as dependencies on x of entries of $A^k x$ in block i.
2. Send the entries of x needed by other procs.
3. Compute the locally computable entries of $A^j x$ for $1 \leq j \leq k$.
4. Receive the entries of x needed from other procs.
5. Compute the remaining entries of $A^j x$ for $1 \leq j \leq k$.

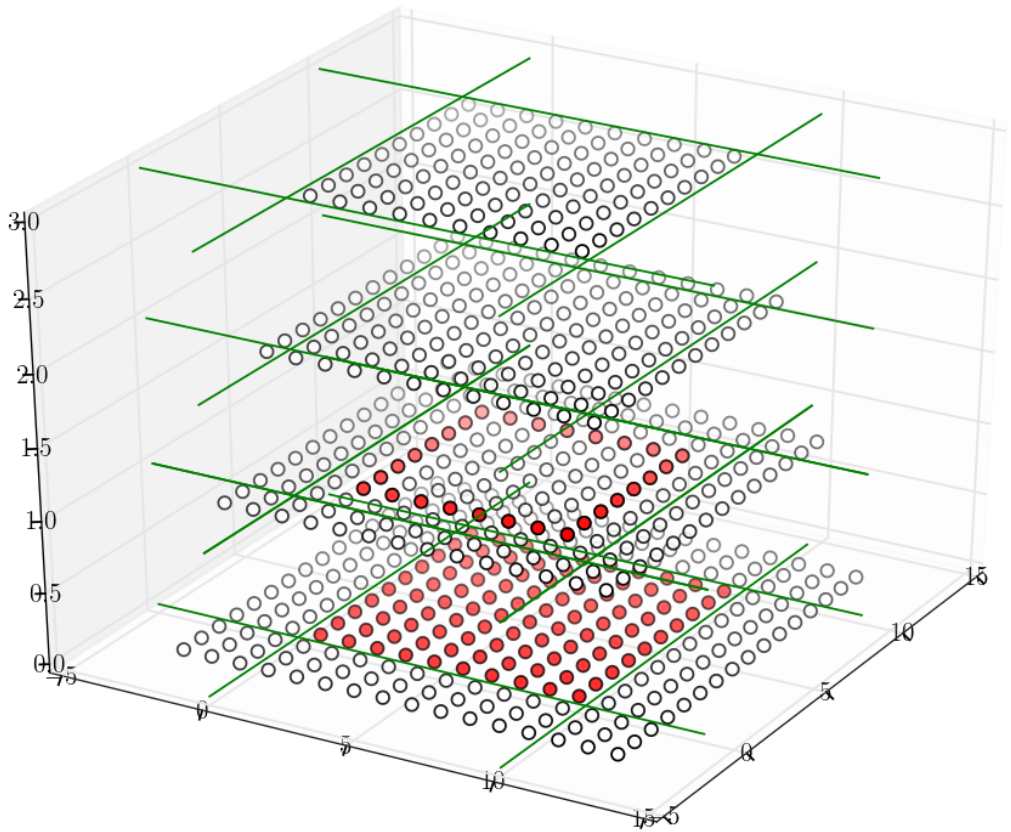
- $O(1)$ messages between any 2 procs.
- Redundant computations.

Improving PA1: 9-pt operator for 3 iterations



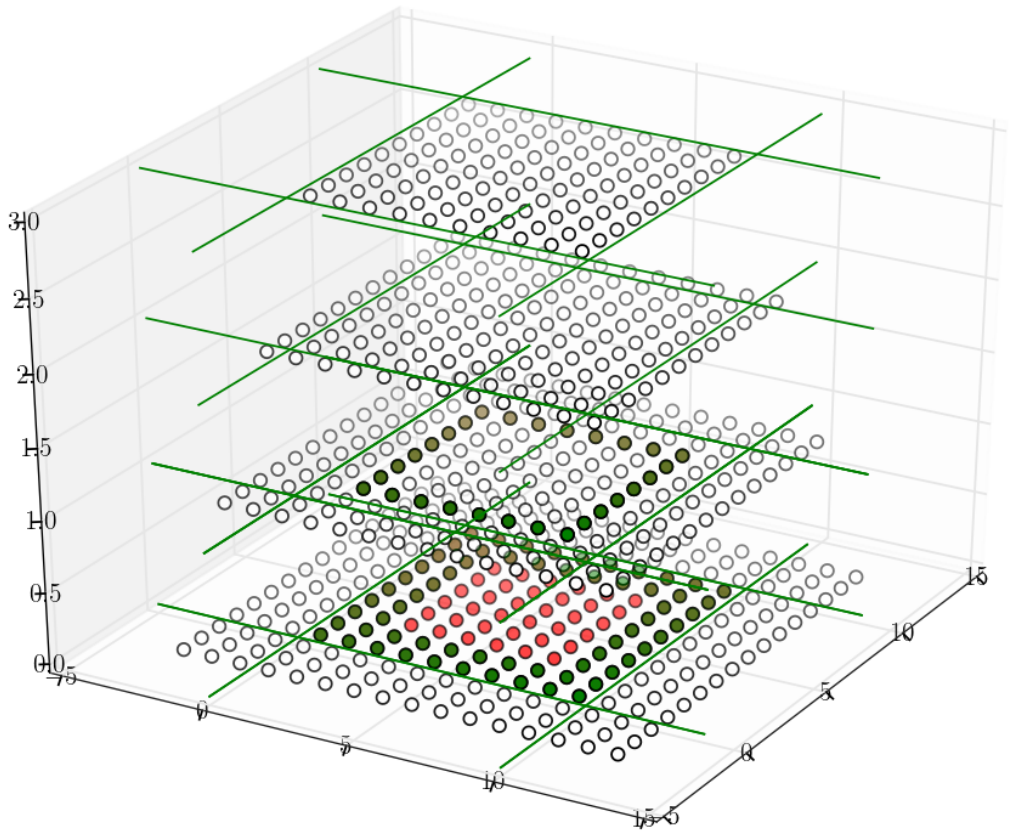
1. Entries available marked red.
2. Compute entries needed by other procs.
3. Send entries to other procs (green).
4. Compute remaining locally computable.
5. Receive entries from other procs (blue).
6. Compute remotely dependent entries.

Improving PA1: 9-pt operator for 3 iterations



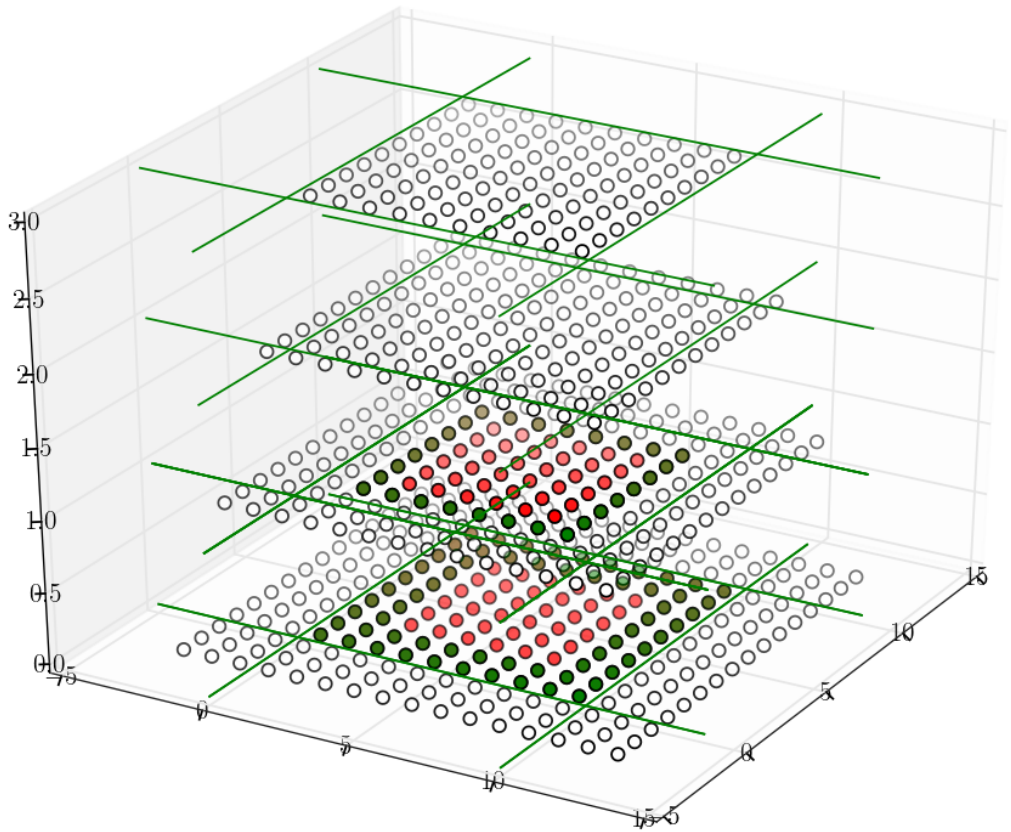
1. Entries available marked red.
2. Compute entries needed by other procs.
3. Send entries to other procs (green).
4. Compute remaining locally computable.
5. Receive entries from other procs (blue).
6. Compute remotely dependent entries.

Improving PA1: 9-pt operator for 3 iterations



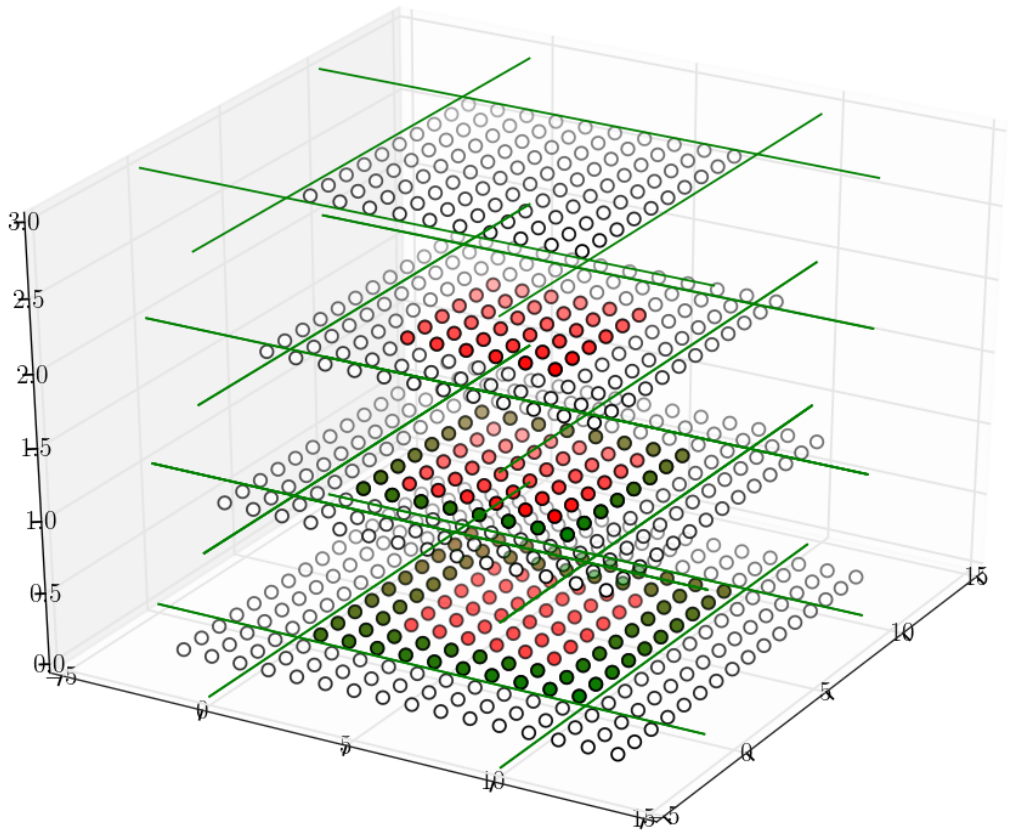
1. Entries available marked red.
2. Compute entries needed by other procs.
3. Send entries to other procs (green).
4. Compute remaining locally computable.
5. Receive entries from other procs (blue).
6. Compute remotely dependent entries.

Improving PA1: 9-pt operator for 3 iterations



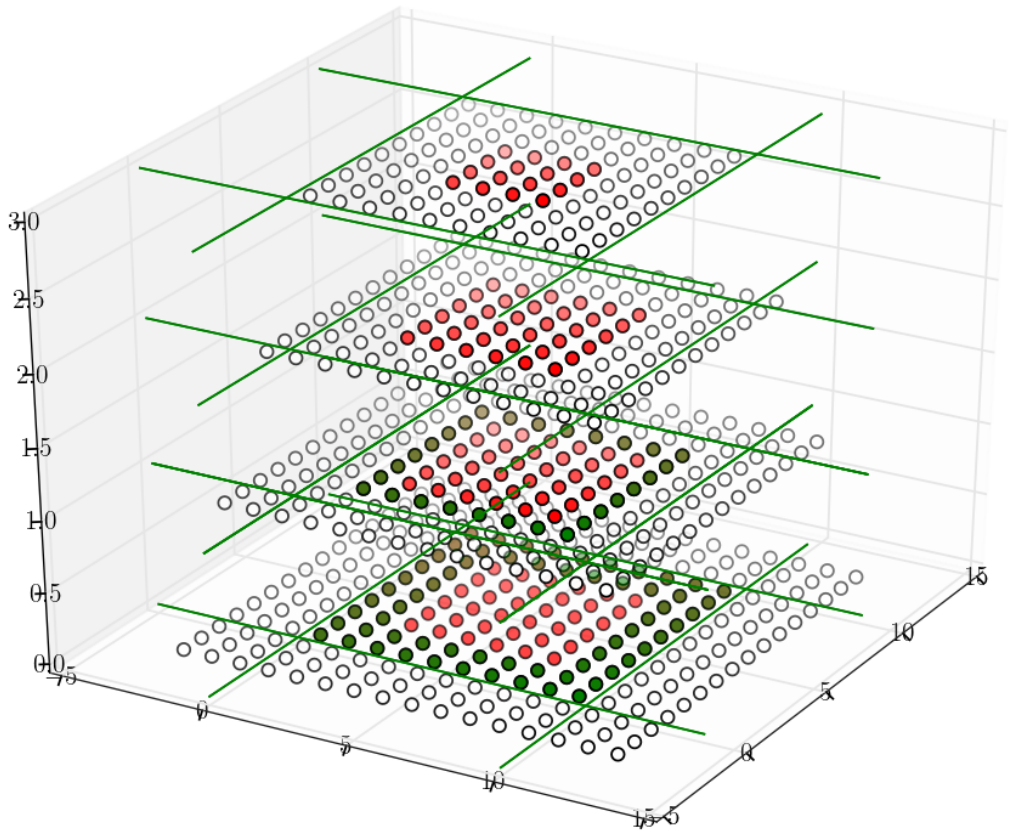
1. Entries available marked red.
2. Compute entries needed by other procs.
3. Send entries to other procs (green).
4. Compute remaining locally computable.
5. Receive entries from other procs (blue).
6. Compute remotely dependent entries.

Improving PA1: 9-pt operator for 3 iterations



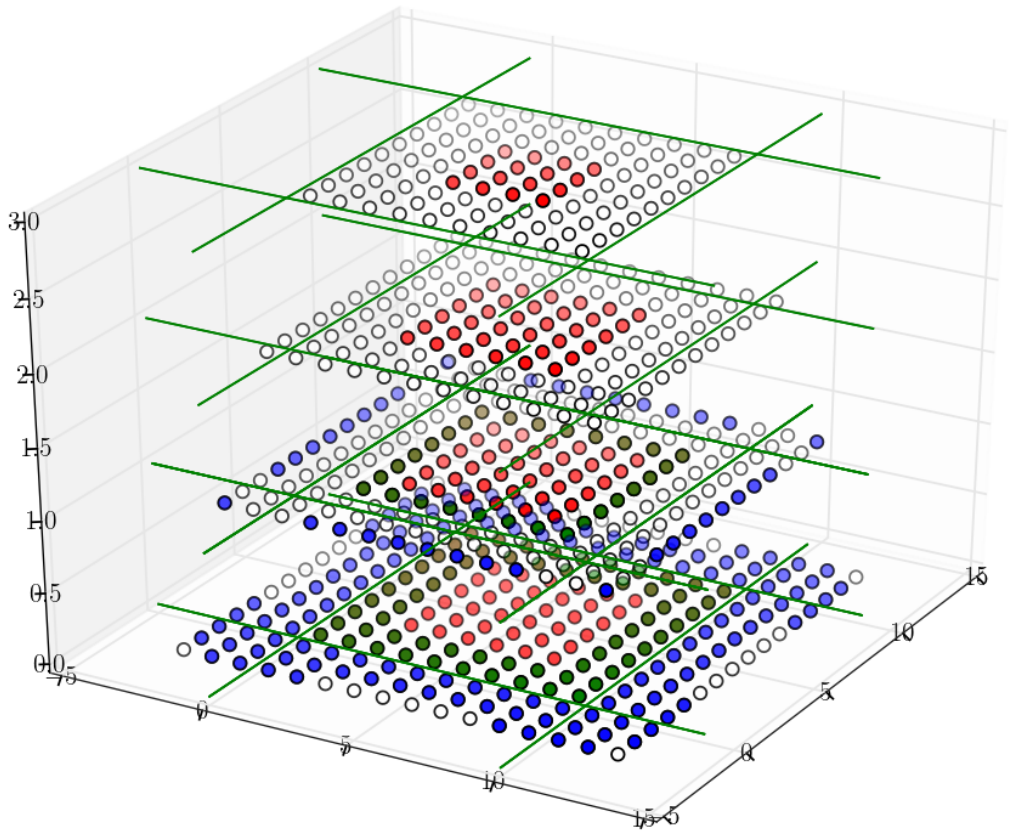
1. Entries available marked red.
2. Compute entries needed by other procs.
3. Send entries to other procs (green).
4. Compute remaining locally computable.
5. Receive entries from other procs (blue).
6. Compute remotely dependent entries.

Improving PA1: 9-pt operator for 3 iterations



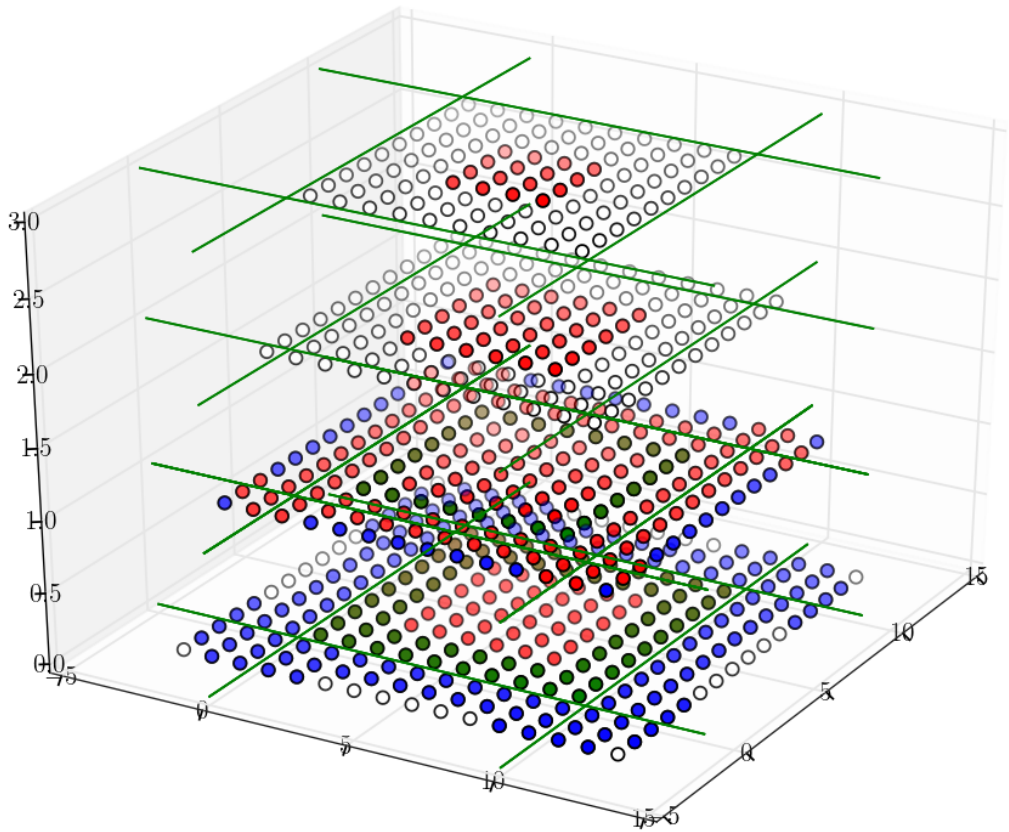
1. Entries available marked red.
2. Compute entries needed by other procs.
3. Send entries to other procs (green).
4. Compute remaining locally computable.
5. Receive entries from other procs (blue).
6. Compute remotely dependent entries.

Improving PA1: 9-pt operator for 3 iterations



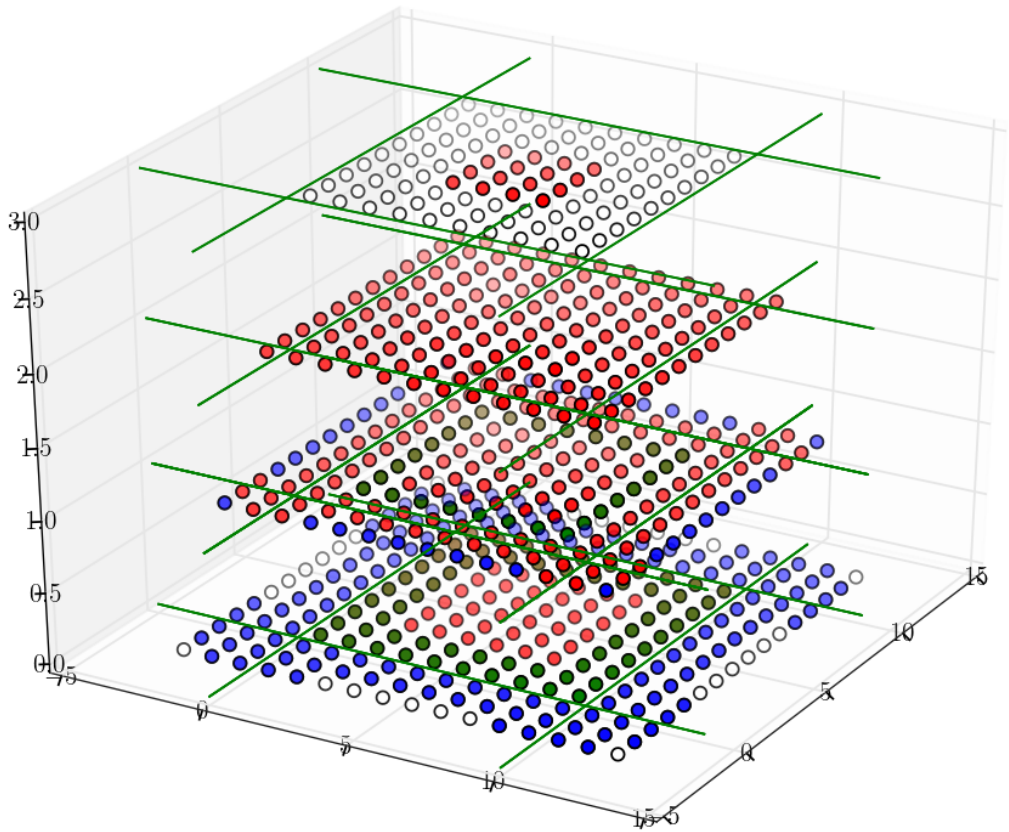
1. Entries available marked red.
2. Compute entries needed by other procs.
3. Send entries to other procs (green).
4. Compute remaining locally computable.
5. Receive entries from other procs (blue).
6. Compute remotely dependent entries.

Improving PA1: 9-pt operator for 3 iterations



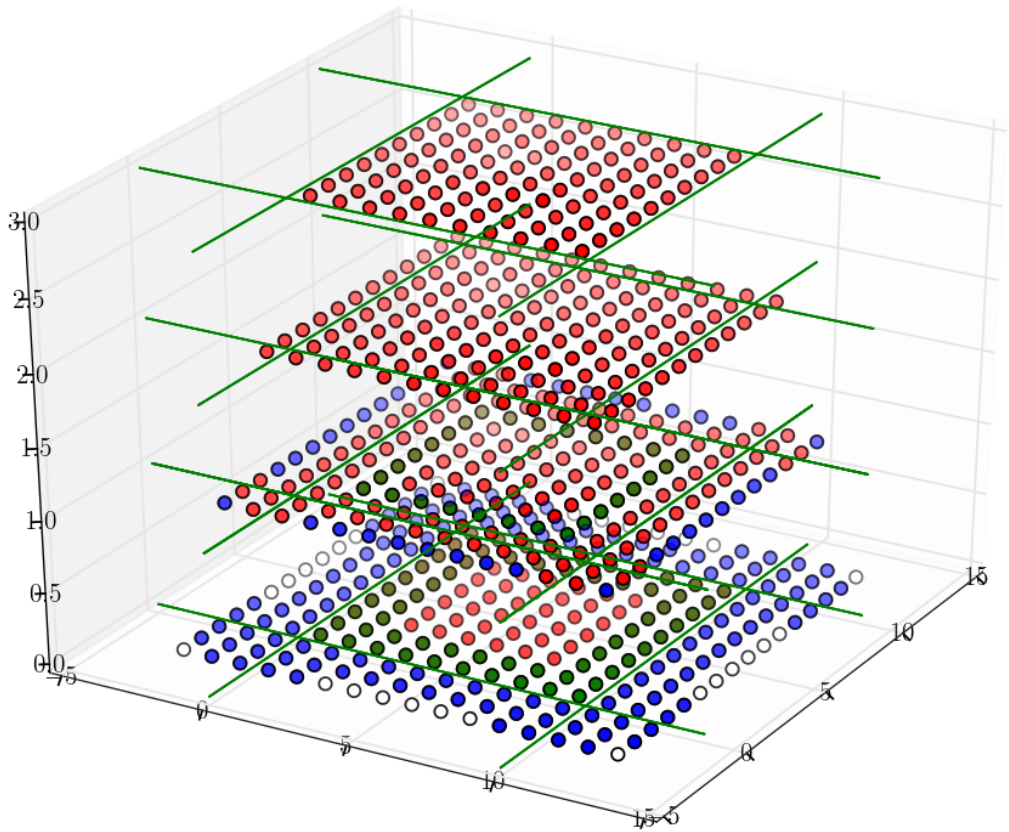
1. Entries available marked red.
2. Compute entries needed by other procs.
3. Send entries to other procs (green).
4. Compute remaining locally computable.
5. Receive entries from other procs (blue).
6. Compute remotely dependent entries.

Improving PA1: 9-pt operator for 3 iterations



1. Entries available marked red.
2. Compute entries needed by other procs.
3. Send entries to other procs (green).
4. Compute remaining locally computable.
5. Receive entries from other procs (blue).
6. Compute remotely dependent entries.

Improving PA1: 9-pt operator for 3 iterations



1. Entries available marked red.
2. Compute entries needed by other procs.
3. Send entries to other procs (green).
4. Compute remaining locally computable.
5. Receive entries from other procs (blue).
6. Compute remotely dependent entries.

PA2 (code for proc. i)

1. Compute the locally computable entries of $x, Ax, \ldots, A^k x$ needed by other procs. These are the entries on the boundary of locally computable and remotely dependent entries.
2. Send the entries of $x, Ax, \ldots, A^k x$ needed by other procs.
3. Compute remaining locally computable entries of $[Ax, \ldots, A^k x]$.
4. Receive the entries of $x, Ax, \ldots, A^k x$ needed from other procs.
5. Compute the remaining entries of $A^j x$ for $1 \leq j \leq k$ using the already computed entries and the fetched entries.

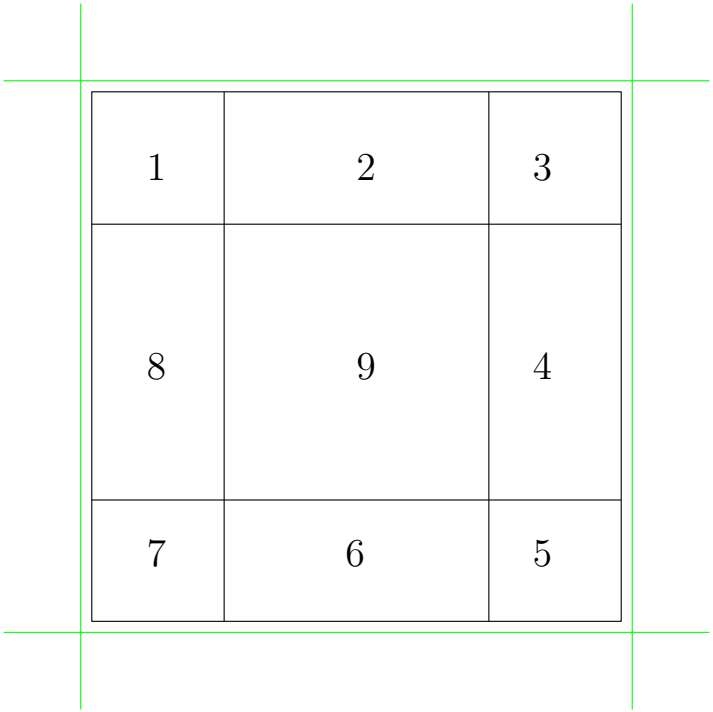
- $O(1)$ messages between any 2 procs..
- Fewer flops than PA1.
  - Locally computable entries computed on only their host proc.

Sequential Algorithm

1. For $i = 1, \ldots, p$, (p= number of blocks of A and x)
2. Load block i from slow memory to fast memory.
3. Load parts of x needed from other blocks in to fast memory.
4. Compute the local entries of $Ax, \ldots, A^k x$.
5. Store the computed entries in to slow memory.

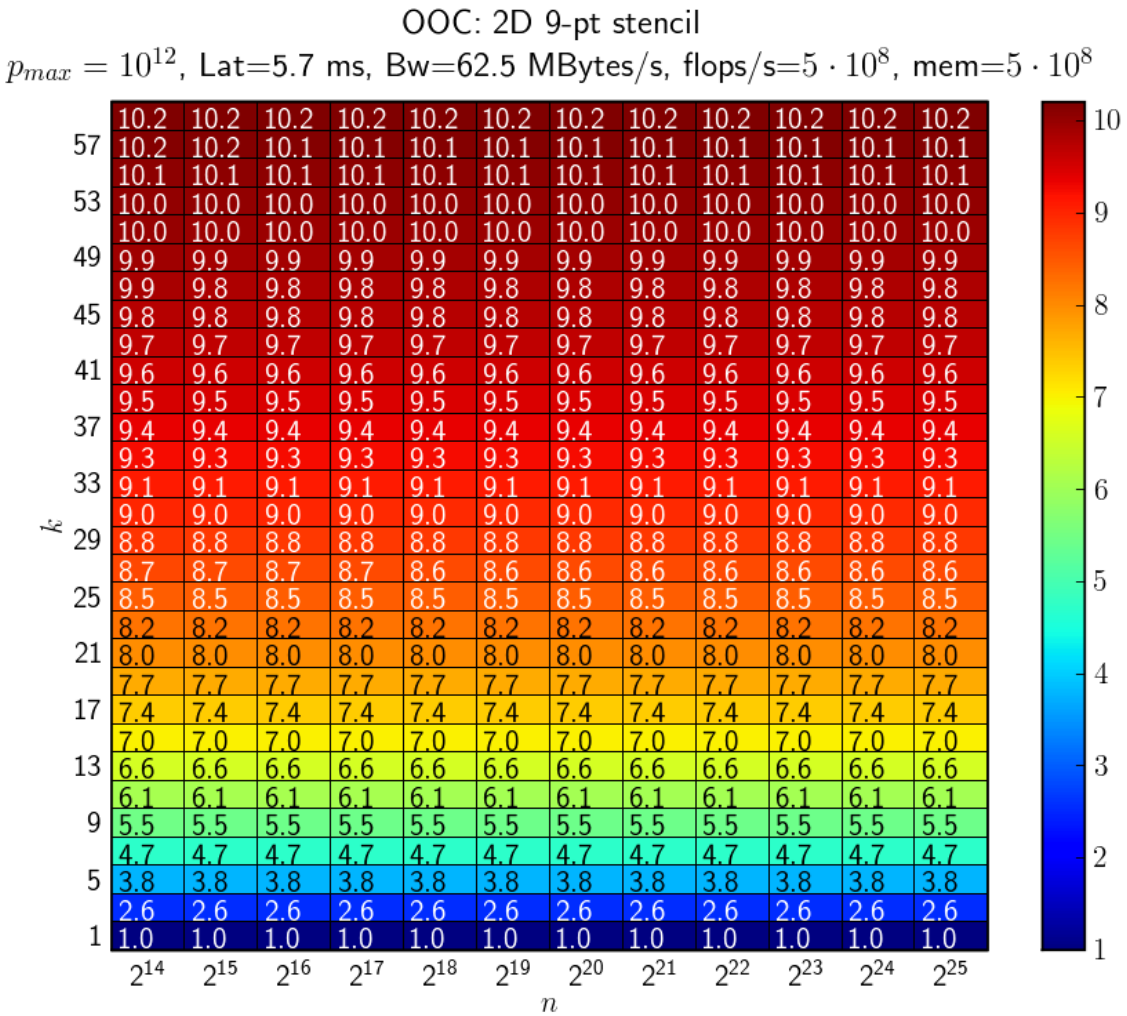
- Entries of x and A may be reordered to minimize the cost of accessing slow memory.
  - Minimizing number of slow memory accesses.
    - 2 kinds of Travelling Salesman Problems: one for the neighbors of a block, and other for the number of blocks.
  - Minimizing number of entries fetched from slow memory:
    - Extra entries fetched to reduce the number of slow memory accesses.

Sequential Algorithm Ordering Example



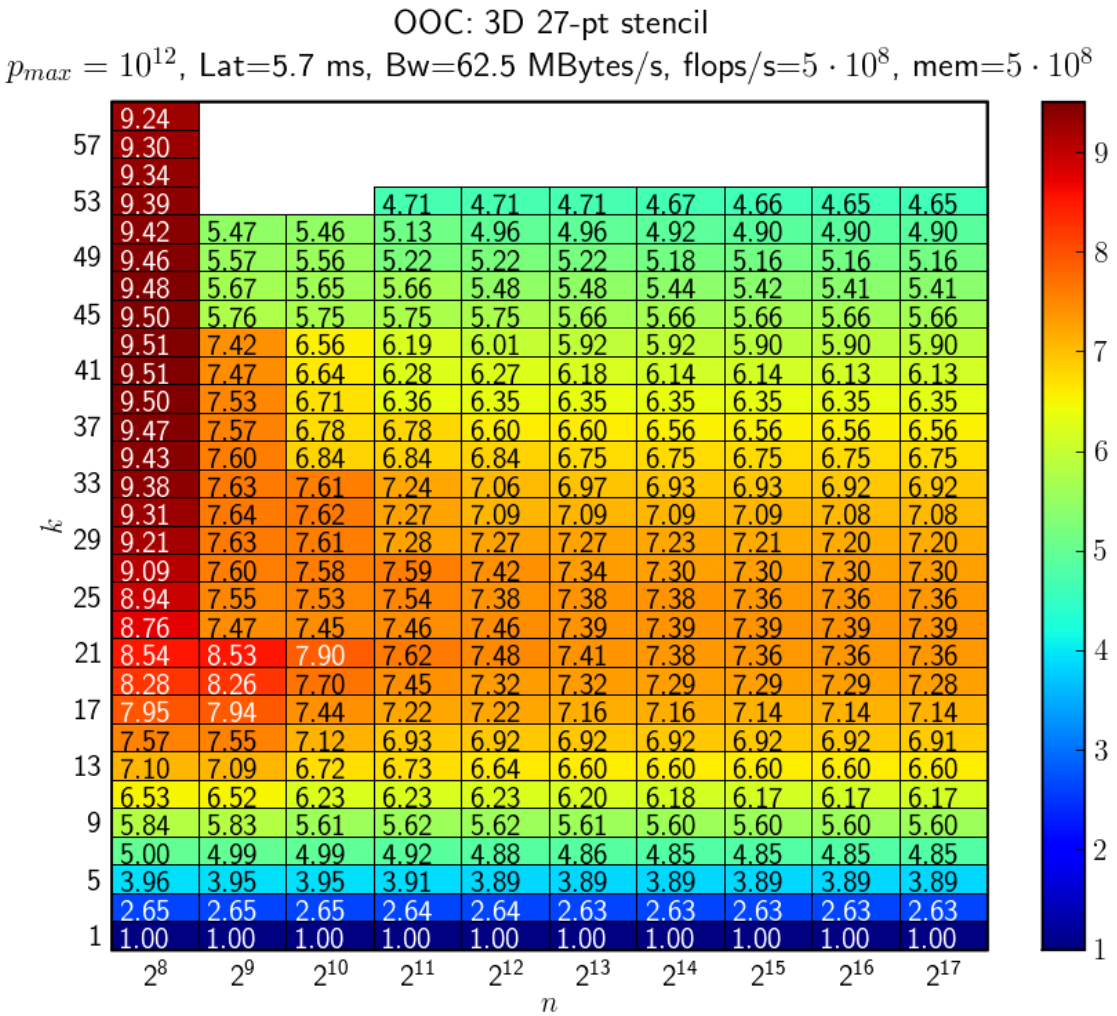
- Left block needs 2 accesses to fetch the entries in 1, 7, 8.
- Other blocks need 1 access to fetch their needed entries.

Performance Model: Sequential Out-of-Core Algorithm on 9-pt Operator



- 500 MFlops/s, mem = 4 GBytes, lat = 5.7 ms, bw = 62.5 MBytes/s.
- No. of blocks p ($1 \leq p \leq p_{max}$) chosen for best perf.
- Speedups across whole range of problem sizes (at least 10x)
  - Reading A and x always costs bw. $\Rightarrow$ speedups always possible.

Performance Model: Sequential Out-of-Core Algorithm on 27-point Operator



- Speedups across whole range of problem sizes (at least 7x).

Performance Model: Parallel Algorithm (PA2)

- 9-pt. operator on $n \times n$ mesh, 27-pt. operator on $n \times n \times n$ mesh.

Peta: No. procs. = 8100, proc. performance = 50 GFlops/s, memory=500 GBytes, lat=10 μs, bw=4 GBytes/s

Matrix	Range of n	Max Modeled Speedup
9-pt	2^{10} to 2^{22}	6.9
27-pt	2^{9} to 2^{14}	1.02

- Speedups for small problem sizes (for 9-pt operator, $2^{10} \leq n \leq 2^{13}$).
- Other problem sizes computation bound, so not limited by communication (hidden by overlap with communication).

Performance Model: Parallel Algorithms (PA2)

Grid: No. procs. = 125, proc. performance = 1 TFlops/s, memory=10 TBytes, lat=100 ms, bw=320 MBytes/s

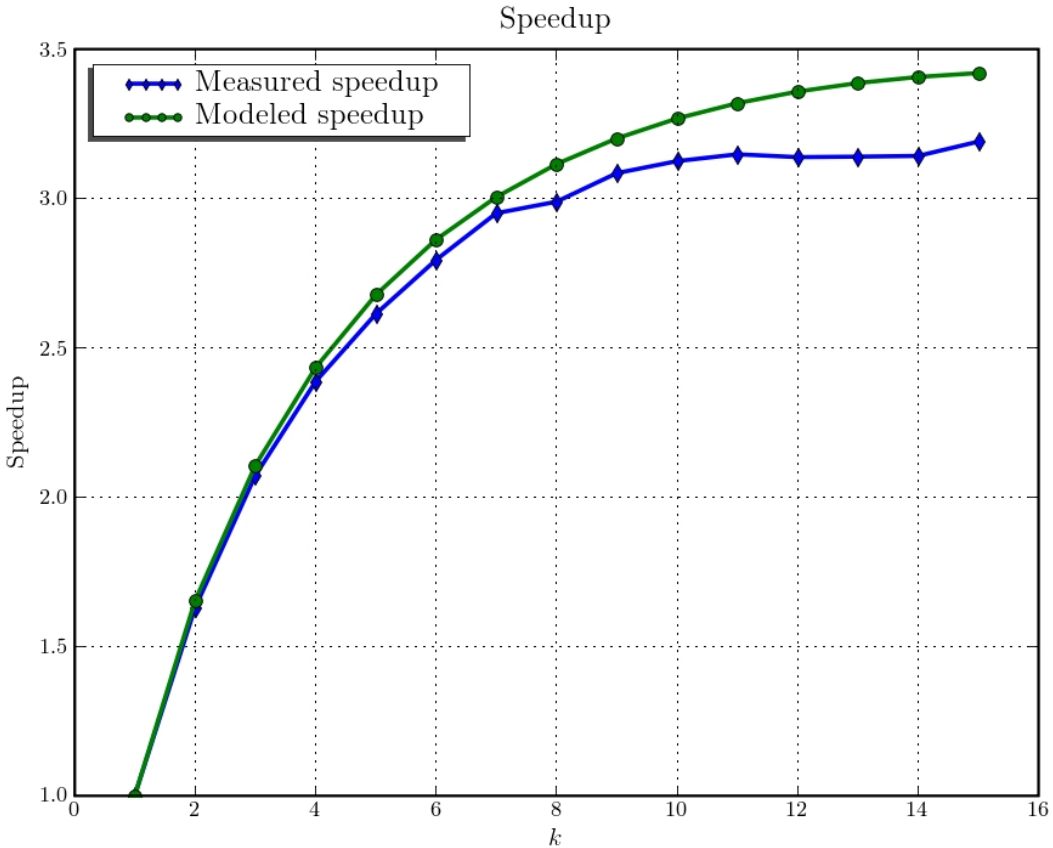
Matrix	Range of n	Max Modeled Speedup
9-pt operator	2^{10} to 2^{22}	22.22
27-pt operator	2^{9} to 2^{14}	4.41

- Small problem sizes run on 1 proc.
- Speedups for moderate problem sizes (for 9-pt operator, $2^{15} \leq n \leq 2^{19}$).
- Large problem sizes computation bound.

Implementation

- Parallel algorithm implemented in UPC (Unified Parallel C).
  - Works for general sparse matrices.
  - For PA1, entries of A and x reordered to make local computations as k invocations of Sparse Matrix Vector multiplication.
- Sequential (out-of-core) algorithm implemented in C.
  - Slow memory assumed to be disk.
  - Reordering done to minimize bandwidth cost for disk access using a randomized heuristic.

Results: Sequential Out-of-Core Algorithm



- Itanium II node with 5.2 GFlops peak flop rate.
- 27-point operator with 368^3 points partitioned in to $4^3 = 64$ blocks.
- Performance 6x slower than ideal machine (∞ DRAM).

Summary

- Sequential and parallel communication avoiding algorithms for the Akx kernel.
  - Almost linear speedups possible.
  - Minimum latency cost.
  - Minimum bandwidth cost for the sequential algorithm.
- Performance modeling of the algorithms.
  - Parallel implementation expected to achieve speedups for moderate problem sizes.
  - Sequential implementation expected to achieve speedups across the whole range of problem sizes.
- Sequential implementation demonstrates speedup of 3x for a 27-point operator.