Solving Poisson's Equation

Outline

- Review Poisson equation
- Overview of Methods for Poisson Equation
- Jacobi's method
- Red-Black SOR method
- Conjugate Gradients

Reduce to sparse-matrix-vector multiply
Need them to understand Multigrid

Next:

- Multigrid method
- FFT

Review: Poisson's equation arises in many models

- **Heat flow:** Temperature(position, time)
- **Diffusion:** Concentration(position, time)
- **Electrostatic or Gravitational Potential:** Potential(position)
- **Fluid flow:** Velocity,Pressure,Density(position,time)
- **Quantum mechanics:** Wave-function(position,time)
- **Elasticity:** Stress,Strain(position,time)

$$1-D: \frac{d^2u}{dx^2} = f(x)$$

$$2-\mathrm{D}: \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$$

$$3-\mathrm{D}: \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = f(x,y,z)$$

Review: Poisson's equation in 1D

- ***1-D Poisson Equation:*** $d^2U/d^2x = b(x)$
- ***Difference equation:*** **-u(i-1)+2*u(i)-u(i+1) =b(i), discrete: i for x_i**
- ***T*z= *b*, *T* contains "geometrical" coefficients, if the grid is regular. *b*=-*b*(x_i)/h^2 and combines forcing terms and boundary conditions.**

$$T = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

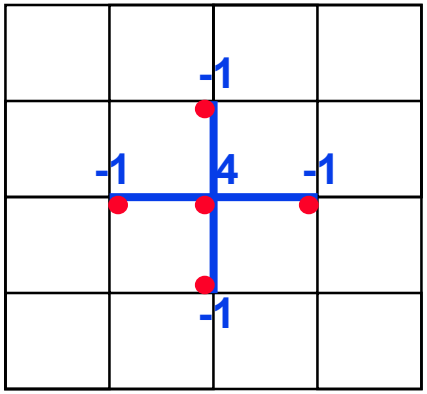
Graph and "stencil"

Review: 2D Poisson's equation

Similar to the 1D case, but the matrix T is now

$$T = \left(\begin{array}{ccc|ccc|ccc}
4 & -1 & & -1 & & & & & \\
-1 & 4 & -1 & & -1 & & & & \\
 & -1 & 4 & & & -1 & & & \\
\hline
-1 & & & 4 & -1 & & -1 & & \\
 & -1 & & -1 & 4 & -1 & & -1 & \\
 & & -1 & & -1 & 4 & & & -1 \\
\hline
 & & & -1 & & & 4 & -1 & \\
 & & & & -1 & & -1 & 4 & -1 \\
 & & & & & -1 & & -1 & 4
\end{array}\right)$$

Graph and "stencil"



- **3D is analogous**

2D Poisson's equation

$$\frac{1}{h^2}(4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}) = f_{ij}$$

$$z = (u_{11}, u_{21}, \ldots, u_{N1}, u_{12}, u_{22}, \ldots, u_{N2}, \ldots\ldots, u_{1N}, u_{2N}, \ldots, u_{NN}) \ .$$

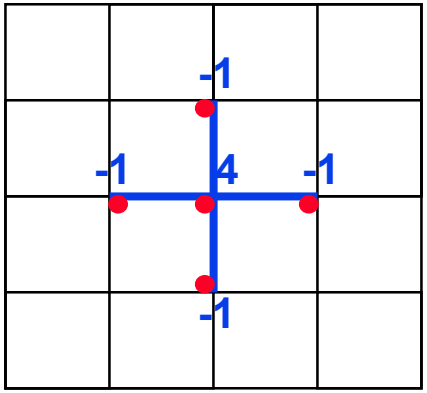
$$z_k := u_{ij} \text{ with } k = i + (j-1)N \text{ for } i, j = 1, \ldots, N \ .$$

$$\frac{1}{h^2}(4z_k - z_{k+1} - z_{k-1} - z_{k+N} - z_{k-N}) = d_k \ \text{ with } \ d_{i+(j-1)N} = f_{ij}$$

Similar to the 1D case, but the matrix T is now

$$T = \left(\begin{array}{ccc|ccc|ccc}
4 & -1 & & -1 & & & & & \\
-1 & 4 & -1 & & -1 & & & & \\
 & -1 & 4 & & & -1 & & & \\
\hline
-1 & & & 4 & -1 & & -1 & & \\
 & -1 & & -1 & 4 & -1 & & -1 & \\
 & & -1 & & -1 & 4 & & & -1 \\
\hline
 & & & -1 & & & 4 & -1 & \\
 & & & & -1 & & -1 & 4 & -1 \\
 & & & & & -1 & & -1 & 4
\end{array}\right)$$

Graph and "stencil"



Various algorithms to solve Poisson's problem

- **Solution methods:**
  - **Direct methods**
  - **Iterative methods**

- **Sorted in two orders (roughly):**
  - **from slowest to fastest on sequential machines**
  - **from most general (works on any matrix) to most specialized (works on matrices "like" Poisson)**

- **Dense LU: Gaussian elimination; works on any N-by-N matrix**
- **Jacobi: essentially does matrix-vector multiply by T in inner loop of an iterative algorithm**

- **Red-Black SOR (Successive Overrelaxation): Variation of Jacobi that exploits yet different mathematical properties of T**
  - **Used in Multigrid**
  - **Red-Black only is related to Gauss-Seidel**

Various algorithms (continued)

- **Conjugate Gradients: uses matrix-vector multiplication, like Jacobi, but exploits mathematical properties of T that Jacobi does not**
- **FFT (Fast Fourier Transform): works only on matrices *very* like T**
- **Multigrid: also works on matrices like T, that come from elliptic PDEs**

- **Lower Bound:**
  - **serial (time to print answer): O(*N*) with *N*=number of unknowns;**
  - **parallel (time to combine N inputs): O(log(*N*))**

Complexity of Algorithms; 2D Poisson with *N* unknowns

Algorithm	Serial	PRAM	Memory	#Procs
° Dense LU	N^3	N	N^2	N^2
°				
° Jacobi	N^2	N	N	N
° RB SOR	$N^{3/2}$	$N^{1/2}$	N	N
° Conj.Grad.	$N^{3/2}$	$N^{1/2}$ *log N	N	N
° FFT	N*log N	log N	N	N
° Multigrid	N	$\log^2 N$	N	N
° Lower bound	N	log N	N	

PRAM is an idealized parallel model with zero cost communication

Jacobi's Method

- To derive Jacobi's method, write Poisson (2D) as:

$$u(i,j) = (u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) + b(i,j))/4$$

- Let $u^m(i,j)$ be approximation for $u(i,j)$ after m steps

$$u^{m+1}(i,j) = (u^m(i-1,j) + u^m(i+1,j) + u^m(i,j-1) + u^m(i,j+1) + b(i,j)) / 4$$

- I.e., $u^{m+1}(i,j)$ is a weighted average of its neighbors
- Motivation: $u^{m+1}(i,j)$ chosen to exactly satisfy the discretized equation at (i,j)
- Convergence is proportional to problem size, $N=n^2$
- Therefore: serial complexity, number of iterations times work per iteration, is $O(N^2)$.

Parallelizing Jacobi's Method

- Reduces to sparse-matrix-vector multiply by (nearly) T

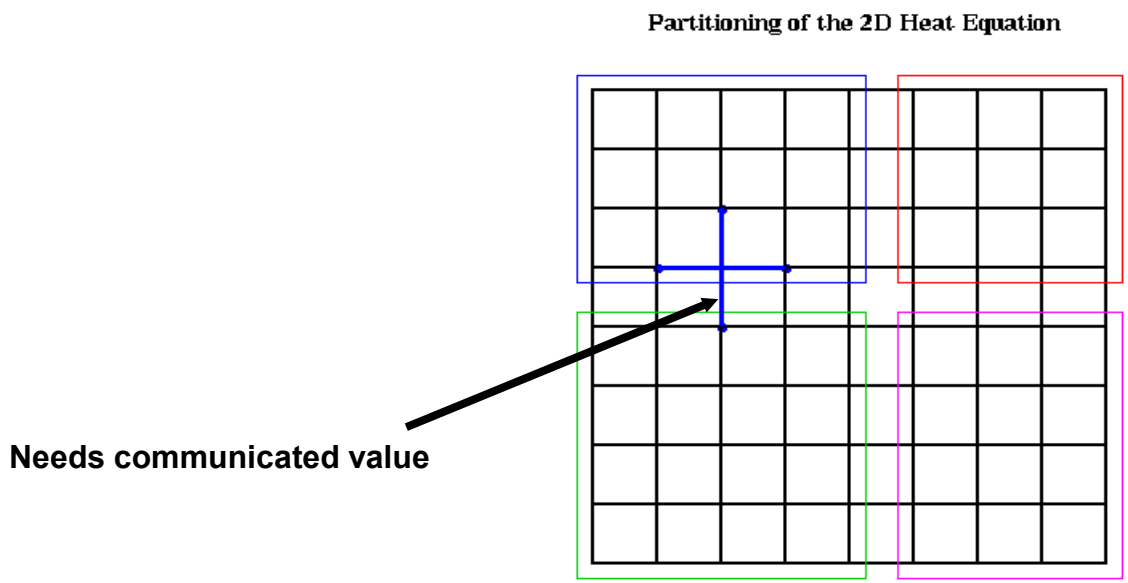
$$\underline{u}^{(m+1)} = (I - T/4) * \underline{u}^{(m)} + \underline{b}/4$$

where I is the unit matrix. Work can be best directly distributed by partition the grid!

- Each value of $\underline{u}^{(m+1)}$ may be updated independently
  - keep 2 copies for time steps m and m+1
- Requires that neighboring boundary values be communicated

Parallelizing Jacobi's Method (continued)

- **Requires that boundary values be communicated**
  - **if each processor owns n^2/p elements to update**
  - **amount of data communicated, $n/\sqrt{p}$ per neighbor, whereas the amount of data owned by each processor is $O(n^2/p)$. So the data locality ratio $=T_{comp}/T_{comm} = O(n/\sqrt{p})$, *can be* relatively large if $n>>p$.**
  - **other partitions are possible as well, see exercise computer lab**



Important performance issues: data locality ratio, large startup time in communication, …

Gauss-Seidel/SOR

- Similar to Jacobi: $u^{m+1}(i,j)$ is computed as a linear combination of neighbors
- Based on 2 improvements over Jacobi
  - Use "most recent values" of u that are available, since these are probably more accurate
  - Update value of $\underline{u}^{(m+1)}$ "more aggressively" at each step
- First, note that while evaluating *sequentially*
  - $u^{m+1}(i,j) = (u^{m}(i-1,j) + u^{m}(i+1,j) \ldots$

some of the values for m+1 are already available

  - $u^{m+1}(i,j) = (u^{latest}(i-1,j) + u^{latest}(i+1,j) \ldots$

where 'latest' is either m or m+1

Gauss-Seidel

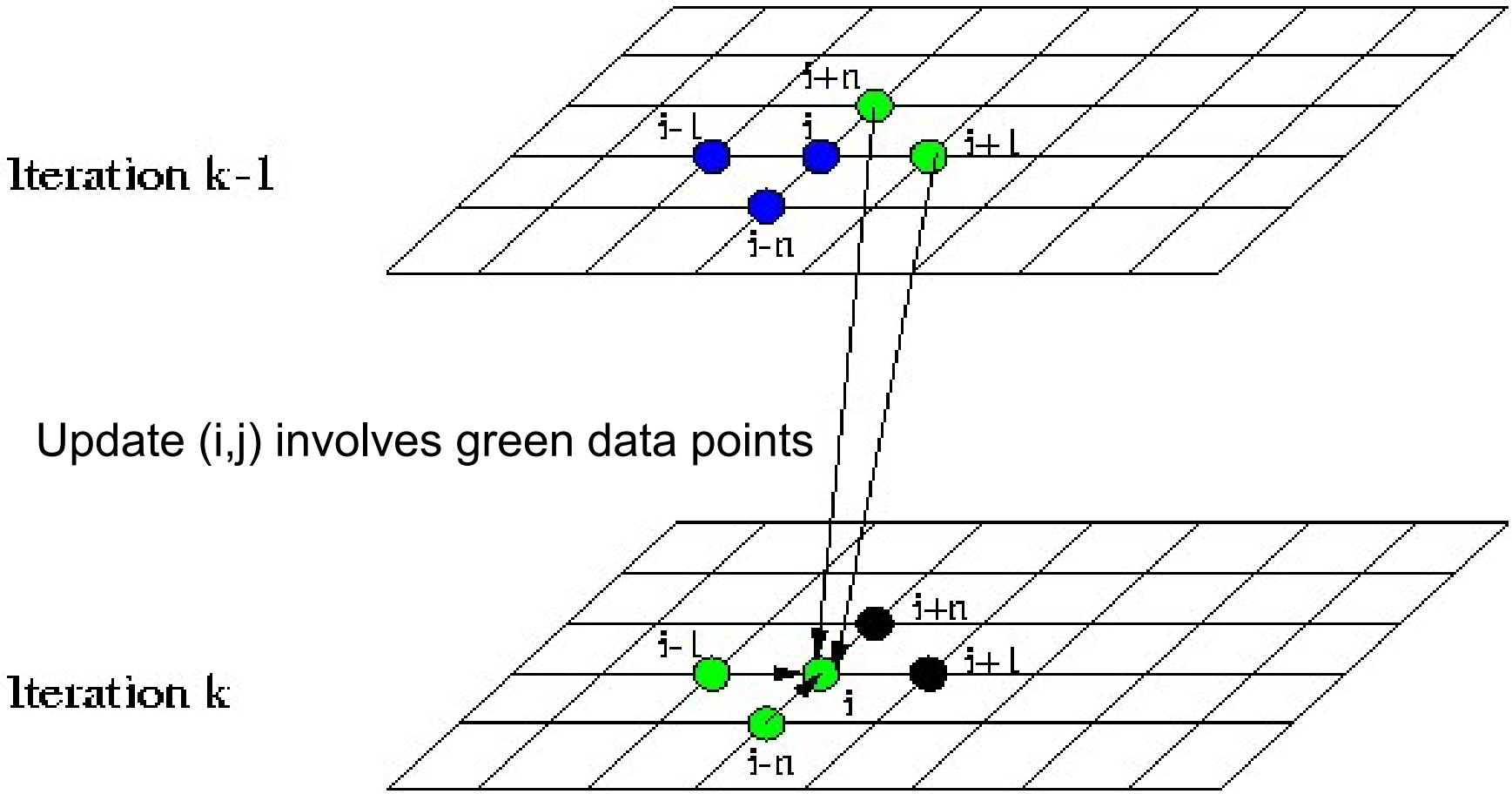
- Updating immediately left-to-right row-wise order, we get the Gauss-Seidel algorithm

```
for i = 1 to n
  for j = 1 to n
```

$$u^{m+1}(i,j) = (u^{m+1}(i-1,j) + u^{m}(i+1,j) + u^{m+1}(i,j-1) + u^{m}(i,j+1) + b(i,j)) / 4$$

- Cannot be parallelized easily, because of dependencies, so we have to think of something else ….
- Alternative: Visit points in a different sequence

Data dependences of GS on a grid



Red-Black Gauss-Seidel

Instead of default ordering we now use a "red-black" order

forall black points u(i,j)

$u^{m+1}(i,j) = (u^{m}(i-1,j) + \ldots$

forall red points u(i,j)

$u^{m+1}(i,j) =(u^{m+1}(i-1,j)+ \ldots$ (black points)

1	17	2	18	3	19	4	20
21	5	22	6	23	7	24	8
9	25	10	26	11	27	12	28
29	13	30	14	31	15	32	16

- Parallelization is now very easy
  - Communication before or after the foralls
- For general graph, use graph coloring
  - Graph(T) is bipartite => 2 colorable (red and black)
  - Nodes for each color can be updated simultaneously
  - Still Sparse-matrix-vector multiply, using submatrices

Successive Overrelaxation (SOR)

- Red-black Gauss-Seidel converges twice as fast as Jacobi, but there are twice as many parallel steps
- To motivate next improvement, rewrite basic step in algorithm :

$$u^{m+1}(i,j) = u^{m}(i,j) + \text{correction}^{(m)}(i,j)$$

- then one should move even further in that direction If "correction" is a good direction to move, i.e., w>1

$$u^{m+1}(i,j) = u^{m}(i,j) + w * \text{correction}^{(m)}(i,j)$$

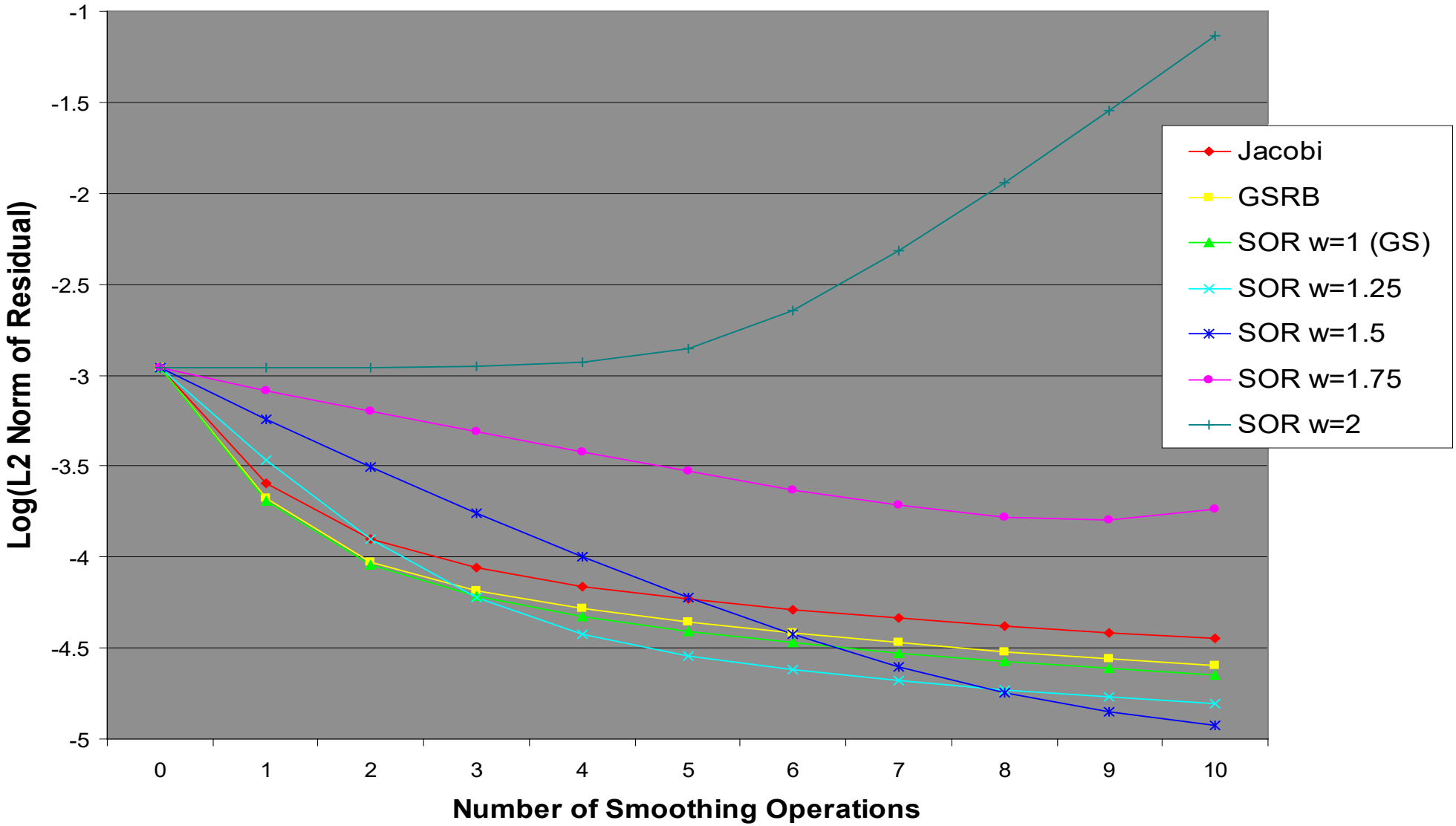
- It's called successive overrelaxation (SOR)
  - Successive: always use latest information like in ordinary Gauss-Seidel
  - Overrelaxtion: $w > 1$
  - But $w < 2$ is necessary for convergence

Red-Black SOR

- Parallelizes like Jacobi
  - Still sparse-matrix-vector multiply…
- Can be proved: $w = 2/(1+\sin(\pi/(n+1)))$ for best convergence
  - Number of steps to converge = parallel complexity = $O(n)$, instead of $O(n^2)$ for Jacobi
  - Serial complexity $O(n^3) = O(N^{3/2})$, instead of $O(n^4) = O(N^2)$ for Jacobi.
- In general w_{Opt} behaves as $2 - O(1/n)$.
- For general matrices T, w_{Opt} should be determined empirically.

Parallel time complexity: minimum parallel execution time of an algorithm on a PRAM computer (see extra slides at the end) with infinite many processors and zero communication cost.
(So it only considers parallelism (parallel operations) but ignores communication overhead).

Convergence Rates for Various Smoothers



Conjugate Gradient; the algorithm

- **Algorithm maintains 3 vectors**
  - $\underline{x}$ = the approximate solution, improved after each iteration
  - $\underline{r}$ = the residual, $\underline{r} = \underline{b} - A*\underline{x}$
  - $\underline{p}$ = search direction, also called the conjugate gradient
- **Start with**
  - $\underline{x}=\underline{0}$, $r=\underline{b}$, $\underline{p}=\underline{b}$
- **Iterate until $\underline{r}\cdot\underline{r}$ is small enough**
  - $\underline{v} = A \cdot \underline{p}$
  - $a = (\underline{r} \cdot \underline{r}) / (\underline{p} \cdot \underline{v})$
  - $\underline{x} = \underline{x} + a * \underline{p}$ new approximate solution
  - $\underline{r}_{old} = \underline{r}$
  - $\underline{r} = \underline{r} - a * \underline{v}$ new residual
  - $\underline{p} = \underline{r} + (\underline{r} \cdot \underline{r}) / \underline{r}_{old} \cdot \underline{r}_{old}) \underline{p}$ new search direction

Conjugate Gradient (CG) for solving $A*\underline{x} = \underline{b}$

- **This method can be used when the matrix A is**
  - **symmetric, i.e., $A = A^T$**
  - **positive definite, defined equivalently as:**
    - **all eigenvalues are positive**
    - **$\underline{x}^T * A * \underline{x} > 0$ for all nonzero vectors s**
    - **a Cholesky factorization, $A = L*L^T$ exists**
- **Algorithm maintains 3 vectors**
  - **$\underline{x}$ = the approximate solution, improved after each iteration**
  - **$\underline{r}$ = the residual, $\underline{r} = \underline{b} - A*\underline{x}$**
  - **$\underline{p}$ = search direction, also called the conjugate gradient**

There are a number of different computations in a CG-iteration, where should we begin with the parallelization?
➔ Rule of thump: start with the most compute-intensive/communication-intensive step.

Conjugate Gradient; computation/operations

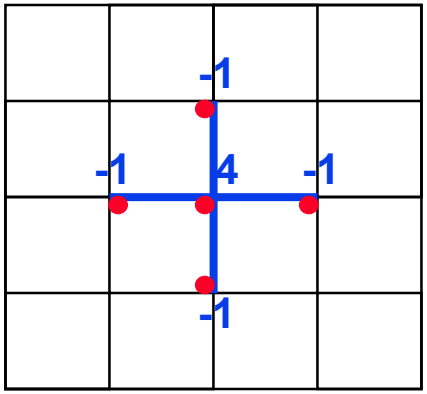
- **Algorithm maintains 3 vectors**
  - $\underline{x}$ = the approximate solution, improved after each iteration
  - $\underline{r}$ = the residual, $\underline{r} = \underline{b} - A*\underline{x}$
  - $\underline{p}$ = search direction, also called the conjugate gradient
- **Start with**
  - $\underline{x}=\underline{0}$, $r=\underline{b}$, $\underline{p}=\underline{b}$
- **Iterate until $\underline{r}\cdot\underline{r}$ is small enough**
  - $\underline{v} = A \cdot \underline{p}$ Matrix-vector multiplication
  - $a = (\underline{r} \cdot \underline{r}) / (\underline{p} \cdot \underline{v})$ Inner product (1x)
  - $\underline{x} = \underline{x} + a * \underline{p}$ vector +scalar*vector
  - $\underline{r}_{old} = \underline{r}$ copy
  - $\underline{r} = \underline{r} - a * \underline{v}$ vector - scalar*vector
  - $\underline{p} = \underline{r} + (\underline{r} \cdot \underline{r}) / \underline{r}_{old} \cdot \underline{r}_{old}) \underline{p}$ vector + scalar*vector

Matrix-vector Multiplication

° Multiply a row-vector with a column-vector: sum of neighboring values (stencil)

$$A = \left(\begin{array}{ccc|ccc|ccc}
4 & -1 & & -1 & & & & & \\
-1 & 4 & -1 & & -1 & & & & \\
 & -1 & 4 & & & -1 & & & \\
\hline
-1 & & & 4 & -1 & & -1 & & \\
 & -1 & & -1 & 4 & -1 & & -1 & \\
 & & -1 & & -1 & 4 & & & -1 \\
\hline
 & & & -1 & & & 4 & -1 & \\
 & & & & -1 & & -1 & 4 & -1 \\
 & & & & & -1 & & -1 & 4
\end{array}\right)$$

Graph and "stencil"



° 3D is analogous

Complexity of Conjugate Gradient (CG)

- **One iteration costs**
  - **Sparse-matrix-vector multiply by A (major cost)**
  - **3 dot products, 3 *saxpys* (scalar*vector + vector)**

- **Converges in $O(n) = O(N^{1/2})$ steps, like SOR**
  - **Serial complexity $= O(N^{3/2})$**
  - **Parallel complexity $= O(N^{1/2} \log N)$,**
  - **The log N factor is from dot-products. Global sum needs to be done. This can be obtained by adding the N (single) products in log N phases.**

- **Implementation on a real parallel computer, computing inner products can be the dominant communication overhead. Why?**

Local (neighbour) communication versus global communication

2D Poisson's equation and matrix equation

$$\frac{1}{h^2}(4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}) = f_{ij}$$

$$z = (u_{11}, u_{21}, \ldots, u_{N1}, u_{12}, u_{22}, \ldots, u_{N2}, \ldots\ldots, u_{1N}, u_{2N}, \ldots, u_{NN}) \ .$$

$$z_k := u_{ij} \text{ with } k = i + (j-1)N \text{ for } i, j = 1, \ldots, N \ .$$

$$\frac{1}{h^2}(4z_k - z_{k+1} - z_{k-1} - z_{k+N} - z_{k-N}) = d_k \text{ with } d_{i+(j-1)N} = f_{ij}$$

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48

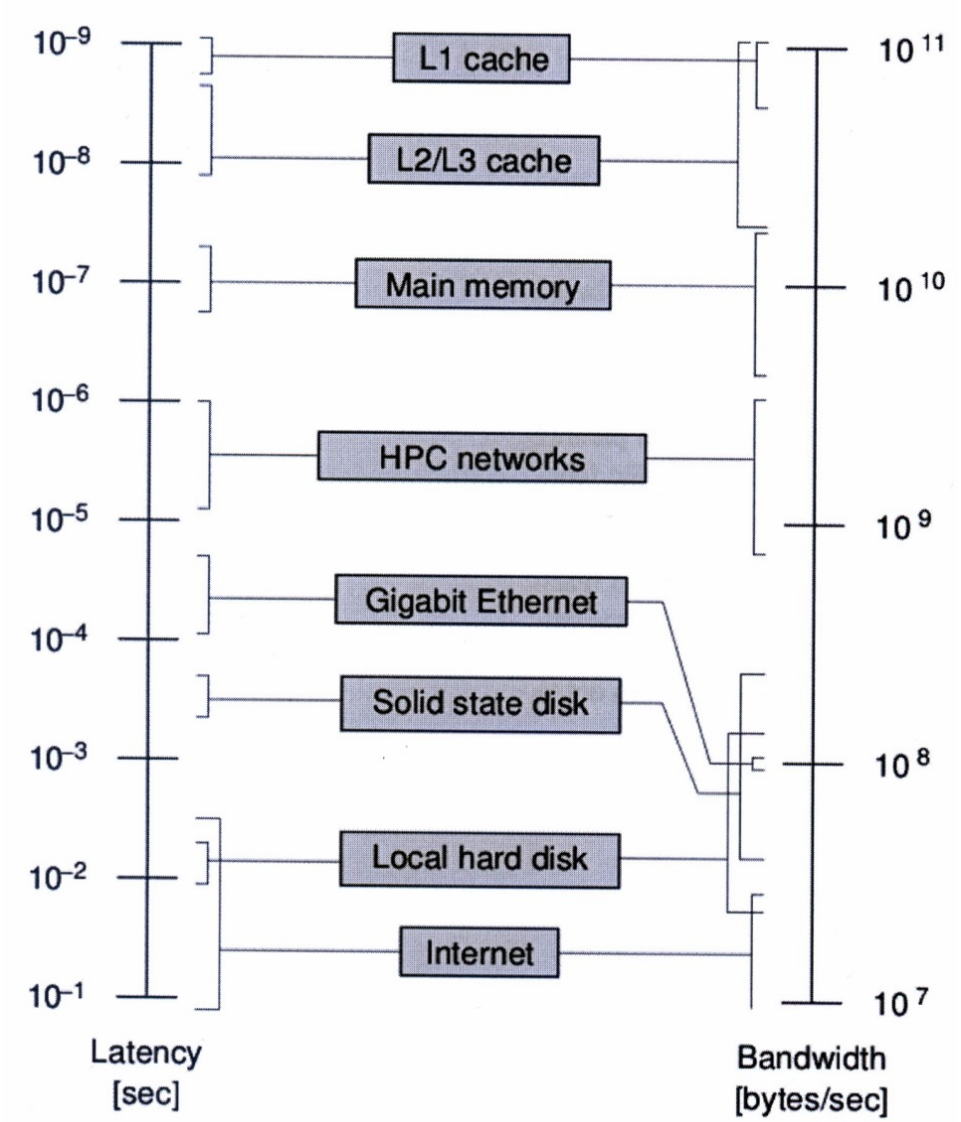
Y

X

$$\begin{bmatrix}
4 & -1 & & & & -1 & & & & & & & & & & \\
-1 & 4 & -1 & & & & -1 & & & & & & & & & \\
 & -1 & 4 & -1 & & & & -1 & & & & & & & & \\
 & & -1 & 4 & 0 & & & & -1 & & & & & & & \\
-1 & & & 0 & 4 & -1 & & & & -1 & & & & & & \\
 & & & & & & & & & & & & & & & \\
 & -1 & & & -1 & 4 & -1 & & & & -1 & & & & & \\
 & & -1 & & & -1 & 4 & -1 & & & & -1 & & & & \\
 & & & -1 & & & -1 & 4 & 0 & & & & -1 & & & \\
 & & & & -1 & & & 0 & 4 & -1 & & & & -1 & & \\
 & & & & & -1 & & & -1 & 4 & -1 & & & & -1 & \\
 & & & & & & & & & & & & & & & \\
 & & & & & & & & & & -1 & -1 & 4 & -1 & & \\
 & & & & & & & & & & -1 & & & -1 & 4 &
\end{bmatrix}$$

Partitioning the grid or the matrix?

Speed difference between memories and networks



Communication time:

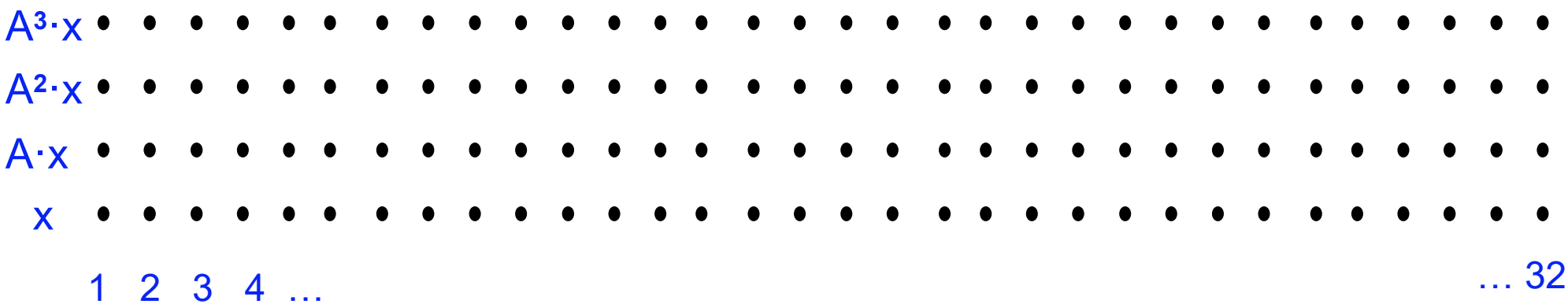
$T_{comm}(L)$=Startup+ L/B,

L=message length (#bytes)
Latency=Startup
B=Bandwidth

Large latency → Frequent comm. of small messages should be avoided!

Communication Avoiding Jacobi:

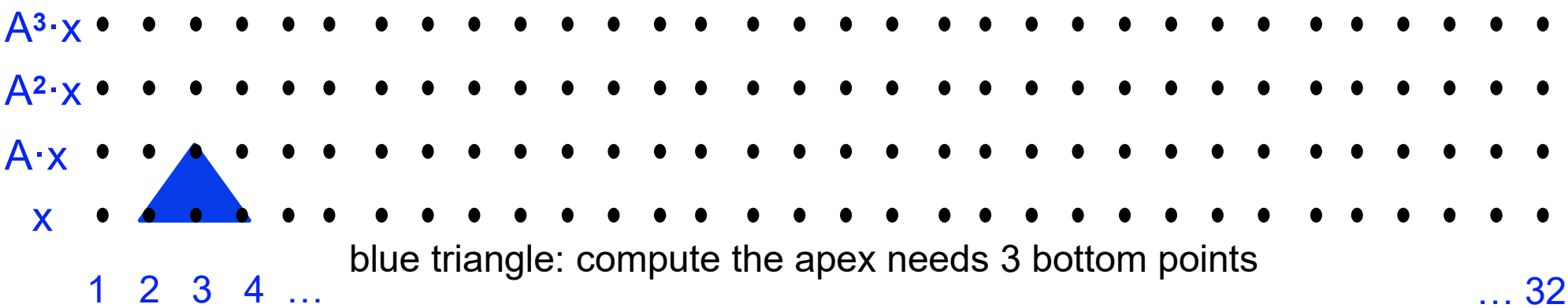
- Iterative methods require exchange of data of neighbor grid points after every iteration. How to reduce the communication overhead? The following is one way of optimization.
- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$



- Example: A tridiagonal, n=32, k=3 (1-D problem). Simplified here: compared to slide 4, $A=(T/2-I)$ and $x^{(m+1)}= Ax^{(m)}+b/2$.
- Like computing the powers of the matrix, but simpler:
  - Don't need to store A explicitly (it's Jacobi)
  - Only need to save vectors A^ix for $i=1,\ldots,k$

Communication Avoiding Jacobi:

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

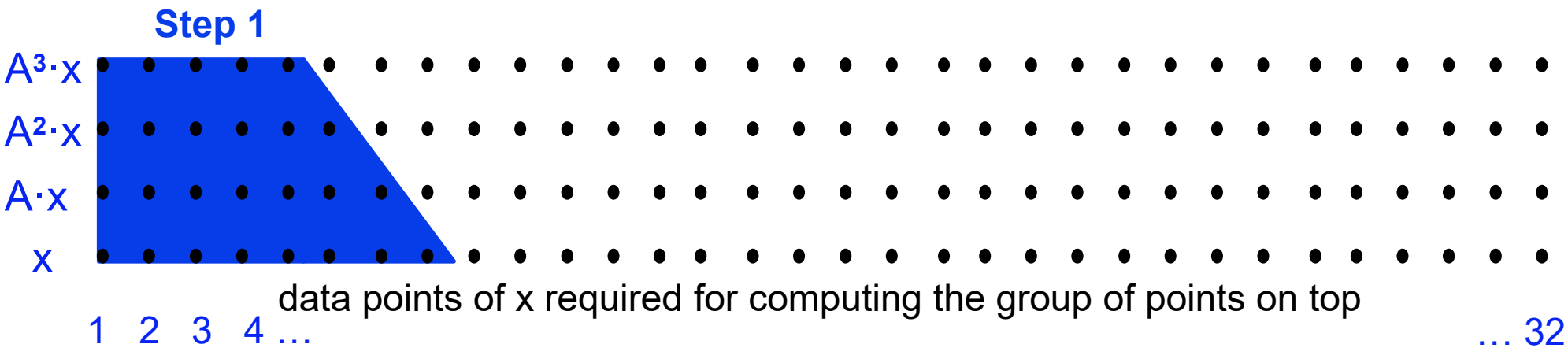


blue triangle: compute the apex needs 3 bottom points

- Example: A tridiagonal, n=32, k=3
- Like computing the powers of the matrix, but simpler :
  - Don't need to store A explicitly (it's Jacobi)
  - Only need to save vectors A^ix for i=1,…,k

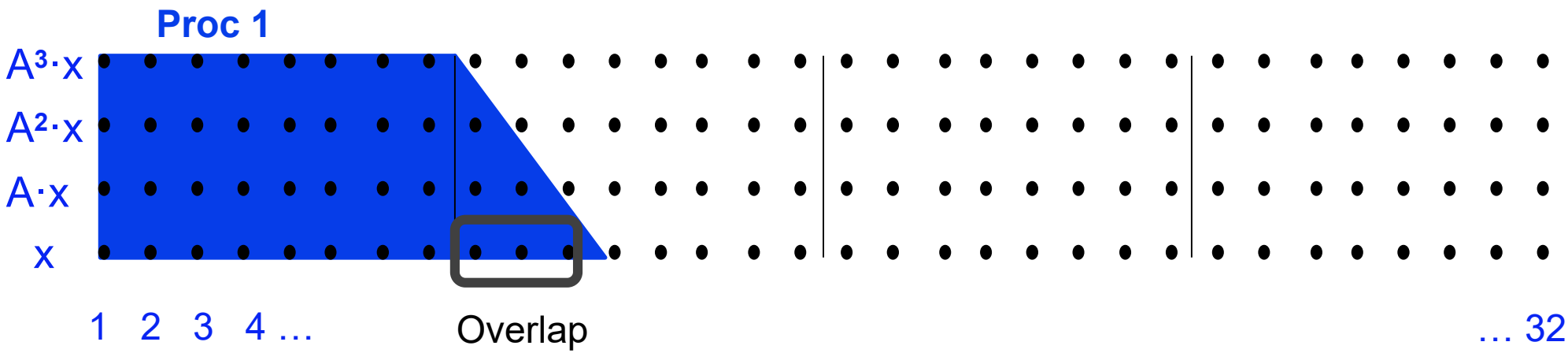
Communication Avoiding Jacobi:

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



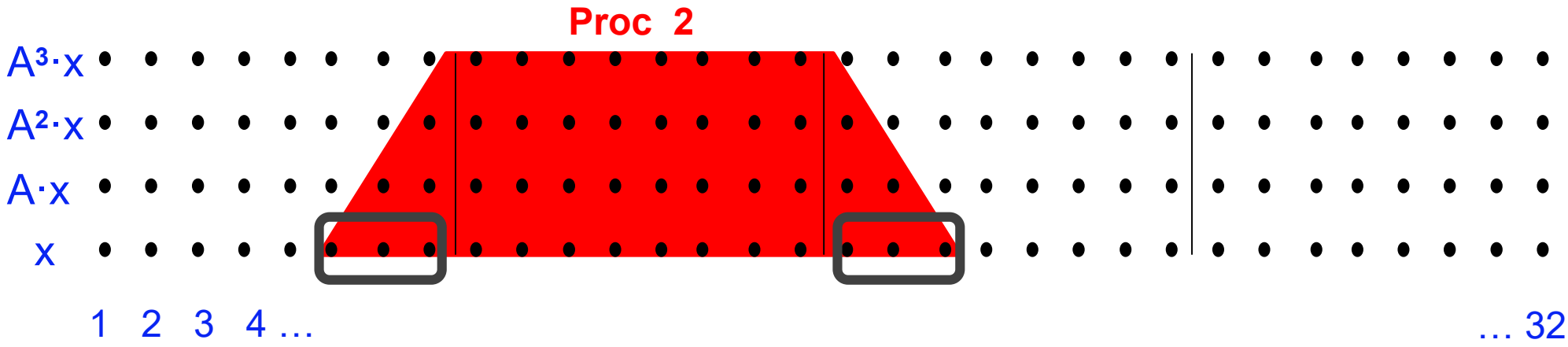
Communication Avoiding Jacobi:

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$
- Parallel Algorithm: each processor computes a sub-vector (sub-matrix-vector product)



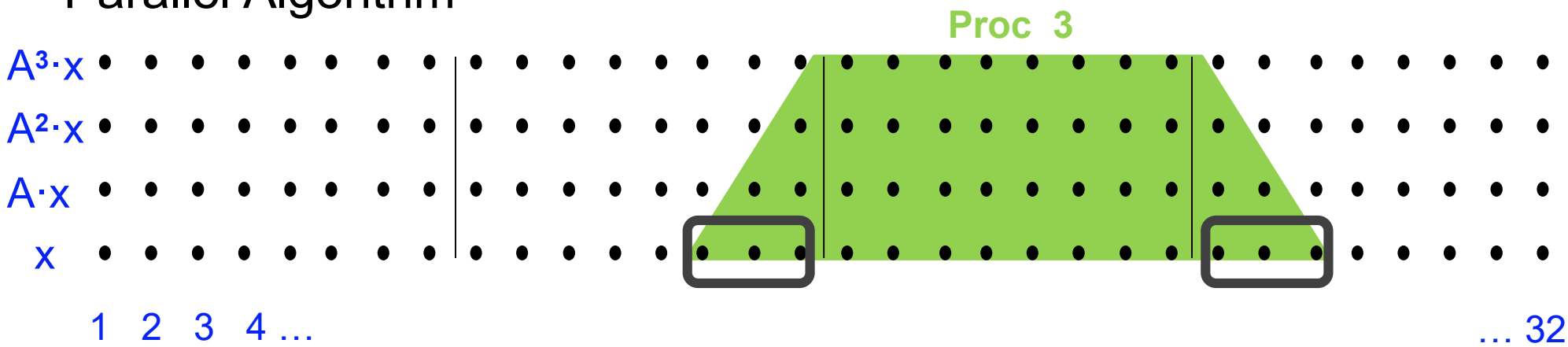
Communication Avoiding Jacobi:

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



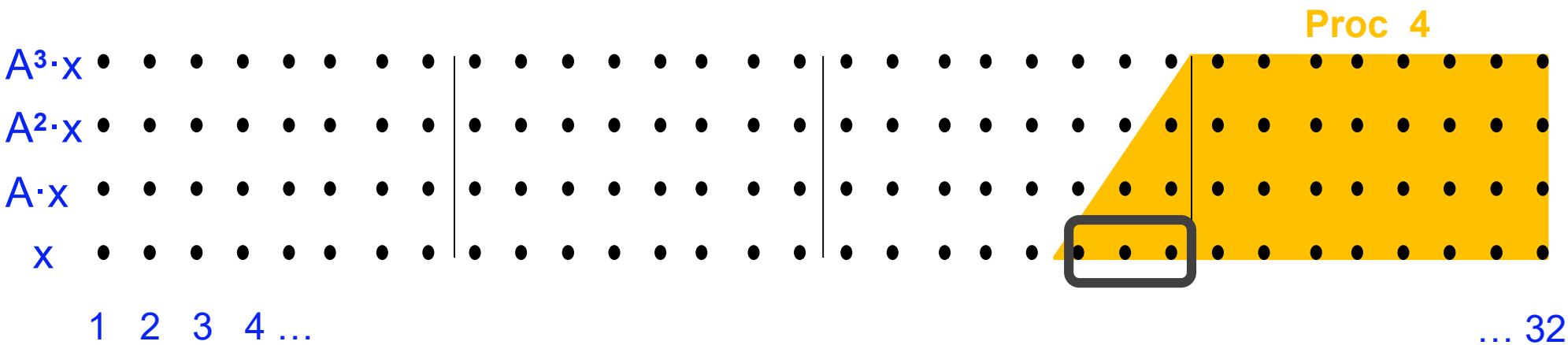
Communication Avoiding Jacobi:

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



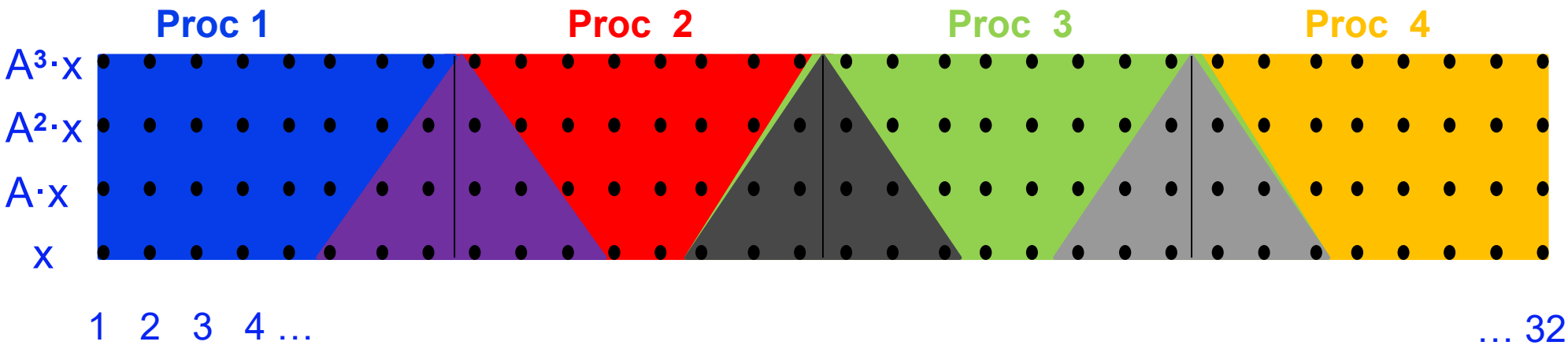
Communication Avoiding Jacobi:

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



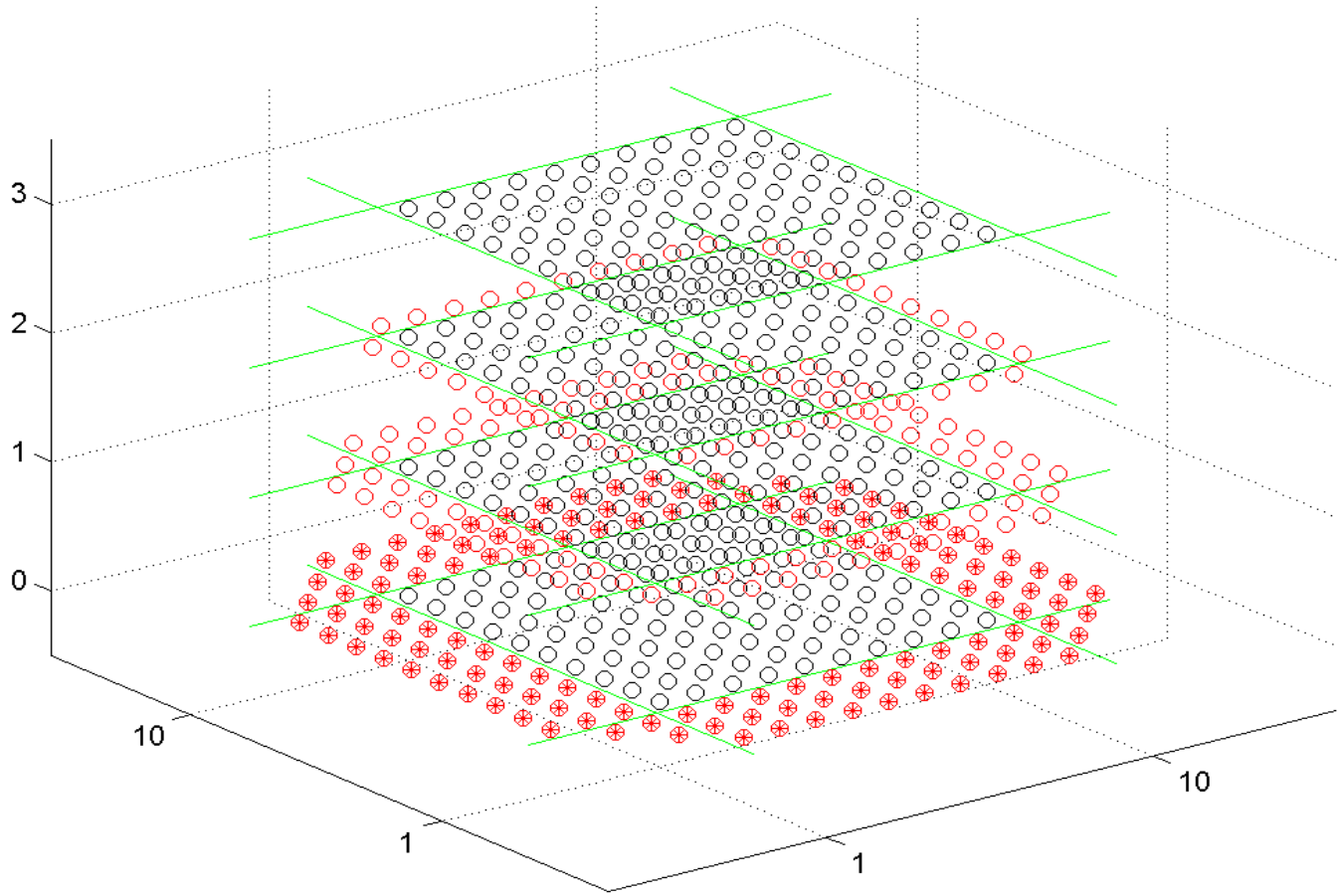
Communication Avoiding Jacobi:

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$
- Parallel Algorithm: perform k iterations at once before the next communication step



- Example: A tri-diagonal, n=32, k=3
- Trade-off:
  - Entries in overlapping regions (triangles) computed redundantly
  - Send O(1) messages instead of O(k)

Remotely Dependent Entries for [x,Ax,A^2x,A^3x], 2D Laplacian



References for Optimizing Stencils (1/2)

- **References at Bebop.cs.berkeley.edu**
  - "Autotuning Stencil Codes for Cache-Based Multicore Platforms", K. Datta, UCB PhD thesis, 2009,
  - "Avoiding Communication in Computing Krylov Subspaces," J. Demmel, M. Hoemmen, M. Mohiyuddin, K. Yelick, 2007
  - "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors", K. Datta, S. Kamil, S. Williams, L. Oliker, J.Shalf, K. Yelick, SIAM Review, 2008

Extra info:

- **SEJITS – sejits.org (Armando Fox et al @ UCB)**
  "Bringing parallel performance to python with domain- specific selective embedded just-in-time specialization"
- **Autotuning stencils and multigrid (Mary Hall @ Utah)**
  super-scidac.org/

References for Optimizing Stencils (2/2)

- Ian Foster et al, on grids (SC2001)
- "Efficient out-of-core algorithms for linear relaxation using blocking covers," C. Leiserson, S. Rao, S. Toledo, FOCS, 1993
- "Data flow and storage allocation for the PDQ-5 program on the Philco-2000," C. Pfeifer, CACM, 1963

Summary of Jacobi, SOR and CG

- Jacobi, SOR, and CG all perform sparse-matrix-vector multiply
- For Poisson, this means nearest neighbor communication on an n-by-n grid ($N=n^2$)
- Parallelization with Red-Black ordering: decoupling dependence.
- Optimization of communication with performing multiple iterations at once (comm. avoidance)
- Limitations of Jacobi, SOR and CG methods:
  - It takes $n = N^{1/2}$ steps for information to travel across an n-by-n grid.
  - Since the solution on one side of grid depends on data on other side of grid faster methods require faster ways to move information
    - Multigrid (next lecture)
    - FFT

Reference model: RAM and PRAM (optional)

Algorithmic Reference Models

Serve as an idealized model for analyzing algorithms

RAM (Random Access Machine).

Uses a simplified abstraction of the hardware

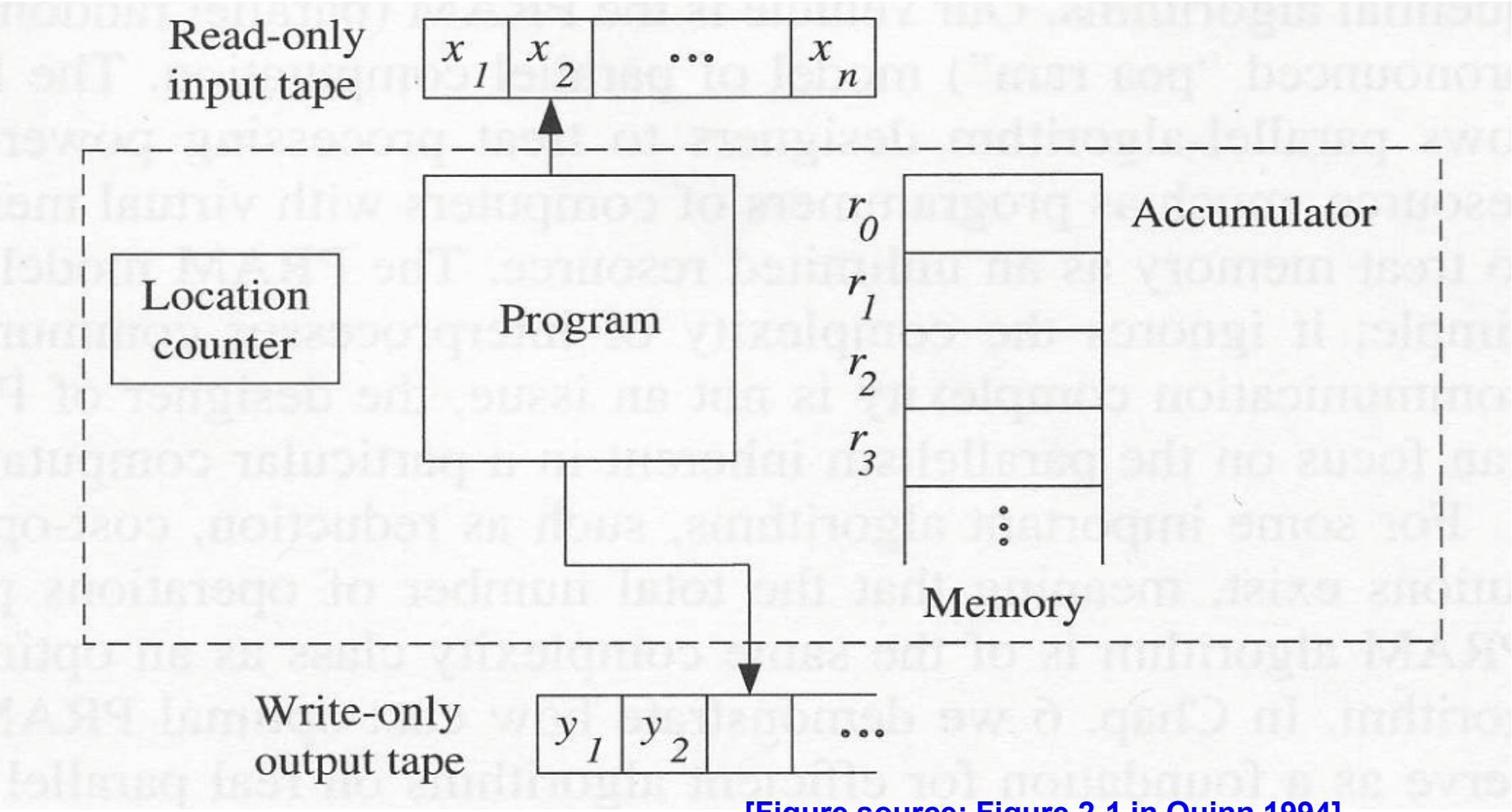
- **Implement an algorithm on a RAM**
- **Execution of a real machine follows the trend using Complexity Analysis**

PRAM (Parallel RAM, pronounced as "P RAM")

Uses a simplified abstraction of the parallel hardware

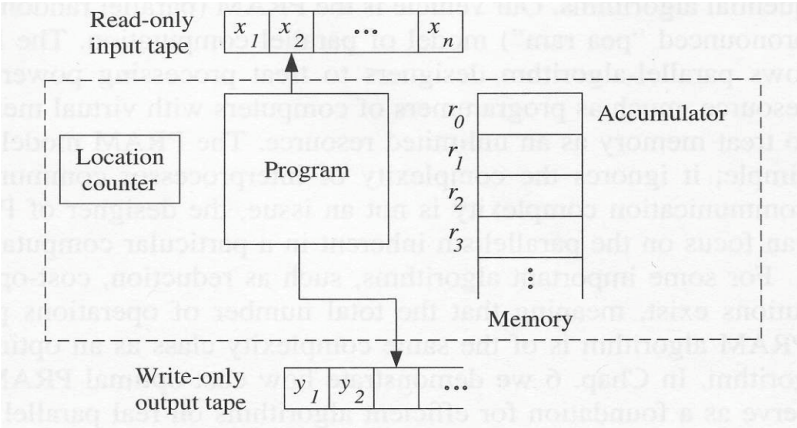
- **The gap is larger than in the sequential case**

RAM model



[Figure source: Figure 2-1 in Quinn 1994]

Analysis using the RAM Model



- Read input of size n
- Write output
- Constant time instructions
- Randomly accessible memory of unbounded size
- Time complexity = # of constant-time instructions
  - Symbolically as function of n
  - Can be often simplified to the # of arithmetic operations (e.g. nxn matrix-vector multiply = $\Theta(n^2)$)
- Space complexity = max # of constant-size memory locations used
  - Symbolically as function of n

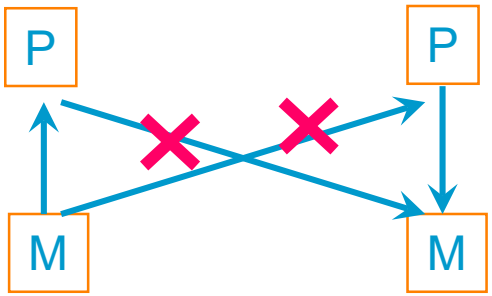
Analysis using the PRAM Model

- p processors
- A global memory of unbounded size that is uniformly accessible (i.e. equal and constant-time access times)
- Processor share the same address space
- Processors have a common clock, but may execute different constant-time instructions, one per clock cycle
- Resource contention is absent
- Time complexity is the # of constant-time instructions by any processor
  - e.g. nxn matrix-vector multiply = $\Omega(\log(n))$ ➔ Why?

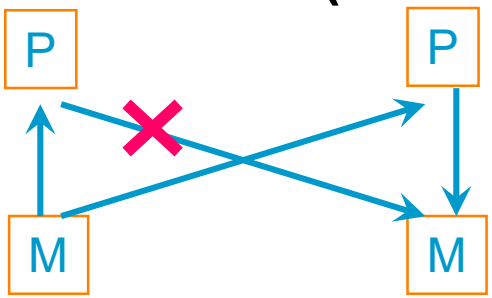
Problem: simultaneous write operations !

PRAM variants (1)

- **Exclusive-read, Exclusive-write (EREW)**

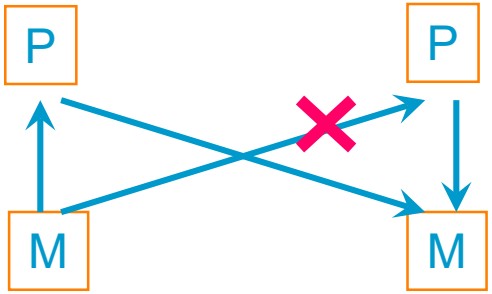


- **Concurrent-read, Exclusive-write (CREW)**

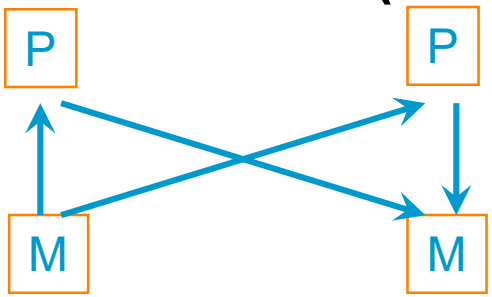


PRAM variants (2)

- **Exclusive-read, Concurrent-write (ERCW)**



- **Concurrent-read, Concurrent-write (CRCW)**



Concurrent-write protocols

- Common (writes allowed if values are identical)
- Arbitrary
- Priority
- Sum (more general: associative)