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 $O(N^2)$.

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 :

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

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

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

- 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(π/(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).

Conjugate Gradient (CG) for solving A*x = 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**
    - **$x^T * A * x > 0$ for all nonzero vectors s**
    - **a Cholesky factorization, $A = L*L^T$ exists**
- **Algorithm maintains 3 vectors**
  - **x = the approximate solution, improved after each iteration**
  - **r = the residual, r = b- A*x**
  - **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

Complexity of Conjugate Gradient (CG)

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

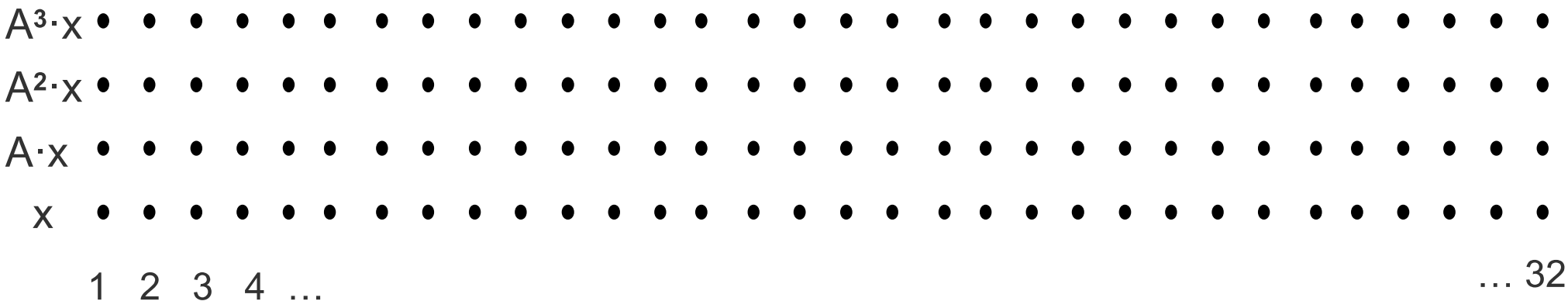
- **Converges in O(n) = O($N^{1/2}$) steps, like SOR**
  - **Serial complexity = O($N^{3/2}$)**
  - **Parallel complexity = O($N^{1/2}$ log N),**
  - **The log N factor is from dot-products. Global sum needs to be done. This can be obtained by adding the N (single) products in log N phases.**

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

Local (neighbour) communication versus global communication

Communication Avoiding Jacobi:

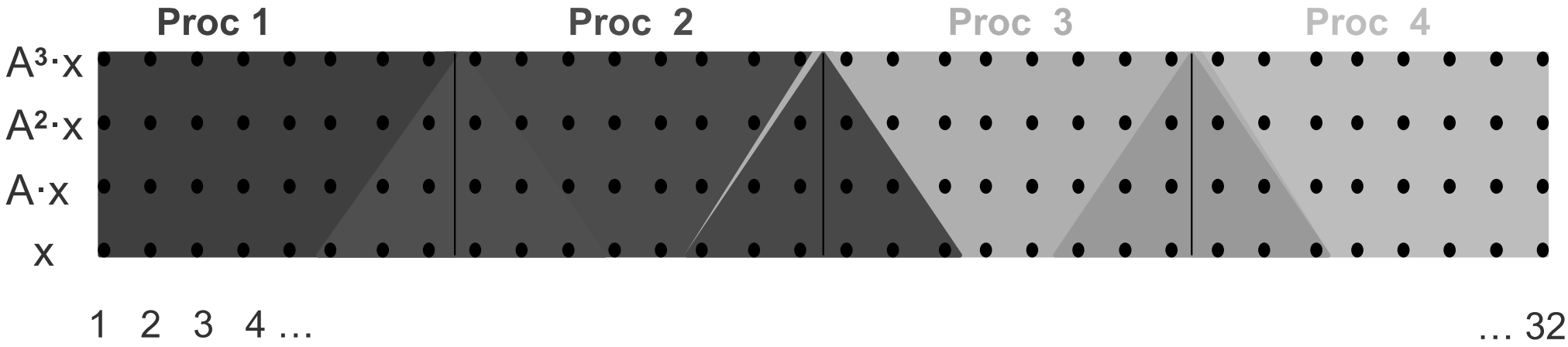
- Iterative methods require exchange of data of neighbor grid points after every iteration. How to reduce the communication overhead? The following is one way of optimization.
- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$



- 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,…,k

Communication Avoiding Jacobi:

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

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