CS 267 Applications of Parallel Computers

Lecture 24:

Solving Linear Systems arising from PDEs - I

James Demmel

http://www.cs.berkeley.edu/~demmel/cs267_Spr99

Outline

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

(Jacobi's method, Red-Black SOR method, Conjugate Gradients: Reduce to sparse-matrix-vector multiply. Need them to understand Multigrid)

- Multigrid (next lecture)

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)

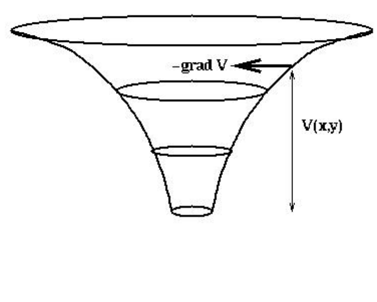
Relation of Poisson's equation to Gravity, Electrostatics

- Force on particle at (x,y,z) due to particle at 0 is

  -(x,y,z)/r^3, where r = sqrt(x^2+y^2 +z^2)
- Force is also gradient of potential V = -1/r

  = -(d/dx V, d/dy V, d/dz V) = -grad V
- V satisfies Poisson's equation (try it!)

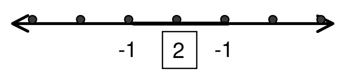
Relationship of Potential V and Force –grad V in 2D



Poisson’s equation in 1D

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

Graph and “stencil”

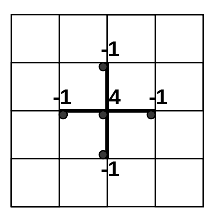


2D Poisson’s equation

° 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

Algorithms for 2D Poisson Equation with N unknowns

Algorithm	Serial	PRAM	Memory	#Procs
° Dense LU	N^3	N	N^2	N^2
° Band LU	N^2	N	$N^{3/2}$	N
° Jacobi	N^2	N	N	N
° Explicit Inv.	N^2	log N	N^2	N^2
° Conj.Grad.	$N^{3/2}$	$N^{1/2}$ *log N	N	N
° RB SOR	$N^{3/2}$	$N^{1/2}$	N	N
° Sparse LU	$N^{3/2}$	$N^{1/2}$	N*log 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

Short explanations of algorithms on previous slide

- 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
- Band LU: exploit fact that T is nonzero only on sqrt(N) diagonals nearest main diagonal, so faster
- Jacobi: essentially does matrix-vector multiply by T in inner loop of iterative algorithm
- Explicit Inverse: assume we want to solve many systems with T, so we can precompute and store inv(T) "for free", and just multiply by it
  - It's still expensive!
- Conjugate Gradients: uses matrix-vector multiplication, like Jacobi, but exploits mathematical properies of T that Jacobi does not
- Red-Black SOR (Successive Overrelaxation): Variation of Jacobi that exploits yet different mathematical properties of T
  - Used in Multigrid
- Sparse LU: Gaussian elimination exploiting particular zero structure of T
- 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); parallel (time to combine N inputs)
- Details in class notes and www.cs.berkeley.edu/~demmel/ma221

Comments on practical meshes

° **Regular 1D, 2D, 3D meshes**

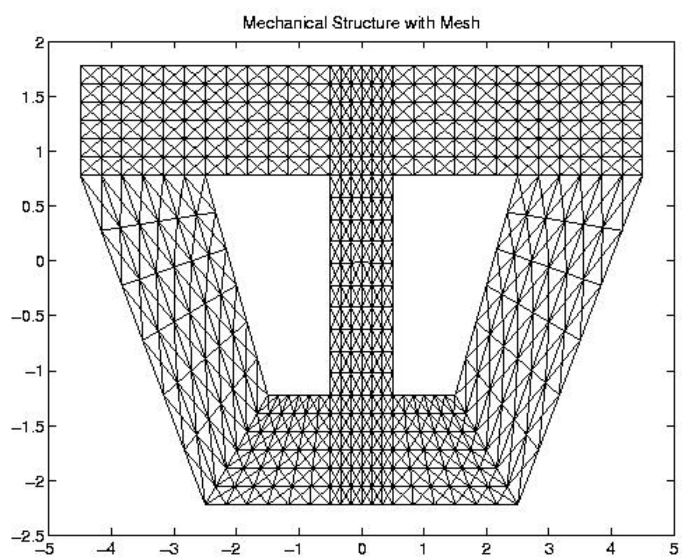
- **Important as building blocks for more complicated meshes**
- **We will discuss these first**

° **Practical meshes are often irregular**

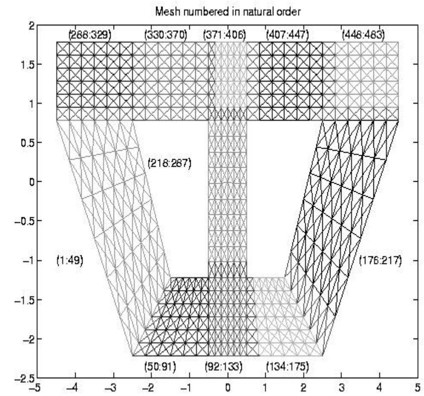
- **Composite meshes, consisting of multiple "bent" regular meshes joined at edges**
- **Unstructured meshes, with arbitrary mesh points and connectivities**
- **Adaptive meshes, which change resolution during solution process to put computational effort where needed**

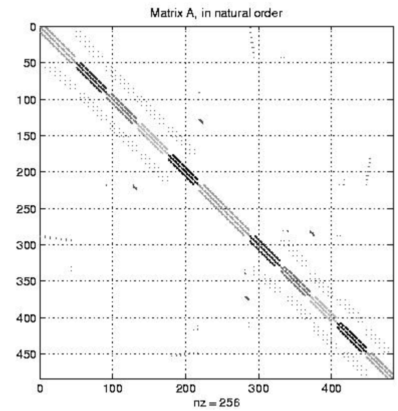
° **In later lectures we will talk about some methods on unstructured meshes; lots of open problems**

Composite mesh from a mechanical structure

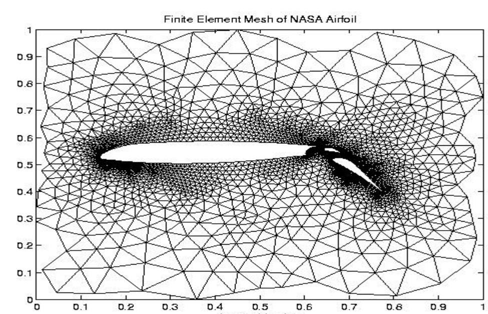


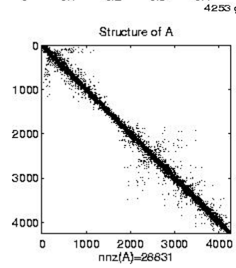
Converting the mesh to a matrix

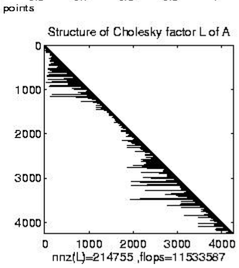




Irregular mesh: NASA Airfoil in 2D (direct solution)







Irregular mesh: Tapered Tube (multigrid)

Example of Prometheus meshes

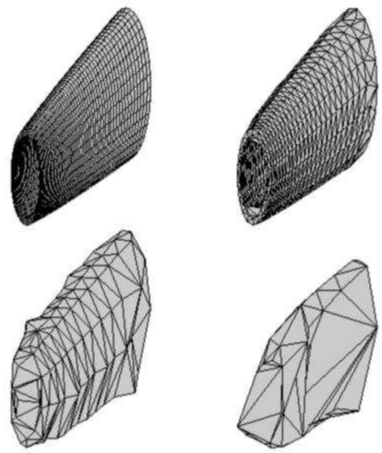
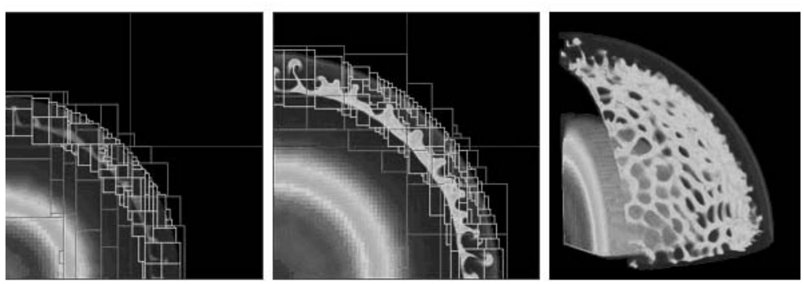


Figure 6 Sample input grid and coarse grids

Adaptive Mesh Refinement (AMR)



- Adaptive mesh around an explosion
- John Bell and Phil Colella at LBL (see class web page for URL)
- Goal of Titanium is to make these algorithms easier to implement in parallel

Jacobi’s Method

° To derive Jacobi’s method, write Poisson 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(i,j,m) be approximation for u(i,j) after m steps

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

° I.e., u(i,j,m+1) is a weighted average of neighbors

° Motivation: u(i,j,m+1) chosen to exactly satisfy equation at (i,j)

° Convergence is proportional to problem size, $N=n^2$

- See http://www.cs.berkeley.edu/~demmel/lecture24 for details

° Therefore, serial complexity is $O(N^2)$

Parallelizing Jacobi’s Method

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

U(m+1) = (T/4 - I) * U(m) + B/4

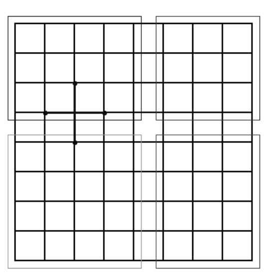
° Each value of U(m+1) may be updated independently

- keep 2 copies for timesteps m and m+1

° Requires that boundary values be communicated

- if each processor owns n^2/p elements to update
- amount of data communicated, n/p per neighbor, is relatively small if n>>p

Partitioning of the 2D Heat Equation



Successive Overrelaxation (SOR)

- Similar to Jacobi: u(i,j,m+1) is computed as a linear combination of neighbors
- Numeric coefficients and update order are different
- Based on 2 improvements over Jacobi
  - Use "most recent values" of u that are available, since these are probably more accurate
  - Update value of u(m+1) "more aggressively" at each step

- First, note that while evaluating *sequentially*
  - u(i,j,m+1) = (u(i-1,j,m) + u(i+1,j,m) …

  some of the values are for m+1 are already available
  - u(i,j,m+1) = (u(i-1,j,latest) + u(i+1,j,latest) …

  where latest is either m or m+1

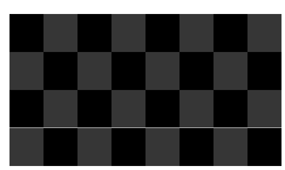
Gauss-Seidel

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

```
for i = 1 to n
 for j = 1 to n
   u(i,j,m+1) = (u(i-1,j,m+1) + u(i+1,j,m) + u(i,j-1,m+1) + u(i,j+1,m)
                   + b(i,j)) / 4
```

- Cannot be parallelized, because of dependencies, so instead we use a "red-black" order

```
forall black points u(i,j)
  u(i,j,m+1) = (u(i-1,j,m) + …
forall red points u(i,j)
  u(i,j,m+1) = (u(i-1,j,m+1) + …
```



- 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, so the same in practice

° To motivate next improvement, write basic step in algorithm as:

u(i,j,m+1) = u(i,j,m) + correction(i,j,m)

° If "correction" is a good direction to move, then one should move even further in that direction by some factor w>1

u(i,j,m+1) = u(i,j,m) + w * correction(i,j,m)

° Called successive overrelaxation (SOR)

° Parallelizes like Jacobi (Still sparse-matrix-vector multiply…)

° Can prove $w = 2/(1+\sin(\pi/(n+1)))$ 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

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 = A*x - b
- p = search direction, also called the conjugate gradient

° One iteration costs

- Sparse-matrix-vector multiply by A (major cost)
- 3 dot products, 3 saxpys (scale*vector + vector)

° Converges in $O(n) = O(N^{1/2})$ steps, like SOR

- Serial complexity = $O(N^{3/2})$
- Parallel complexity = $O(N^{1/2} \log N)$, log N factor from dot-products

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
- It takes $n = N^{1/2}$ steps for information to travel across an n-by-n grid
- Since solution on one side of grid depends on data on other side of grid faster methods require faster ways to move information
  - FFT
  - Multigrid

Solving the Poisson equation with the FFT

- Motivation: express continuous solution as Fourier series
  - $u(x,y) = \Sigma_i \Sigma_k\ u_{ik} \sin(\pi\ ix) \sin(\pi\ ky)$
  - u_{ik} called Fourier coefficient of $u(x,y)$
- Poisson's equation $\delta^2 u/\delta x^2 + \delta^2 u/\delta y^2 = b$ becomes

  $\Sigma_i \Sigma_k\ (-\pi i^2 - \pi k^2)\ u_{ik} \sin(\pi\ ix) \sin(\pi\ ky)$

  $= \Sigma_i \Sigma_k\ b_{ik} \sin(\pi\ ix) \sin(\pi\ ky)$
  - where b_{ik} are Fourier coefficients of $b(x,y)$
- By uniqueness of Fourier series, $u_{ik} = b_{ik} / (-\pi i^2 - \pi k^2)$
- Continuous Algorithm (Discrete Algorithm)
  - Compute Fourier coefficient b_{ik} of right hand side
    - Apply 2D FFT to values of b(i,k) on grid
  - Compute Fourier coefficients u_{ik} of solution
    - Divide each transformed b(i,k) by function(i,k)
  - Compute solution u(x,y) from Fourier coefficients
    - Apply 2D inverse FFT to values of b(i,k)

Serial FFT

- Let i=sqrt(-1) and index matrices and vectors from 0.
- The Discrete Fourier Transform of an m-element vector v is:

F*v

Where F is the m*m matrix defined as:

$$F[j,k] = \varpi^{(j*k)}$$

Where ϖ is:

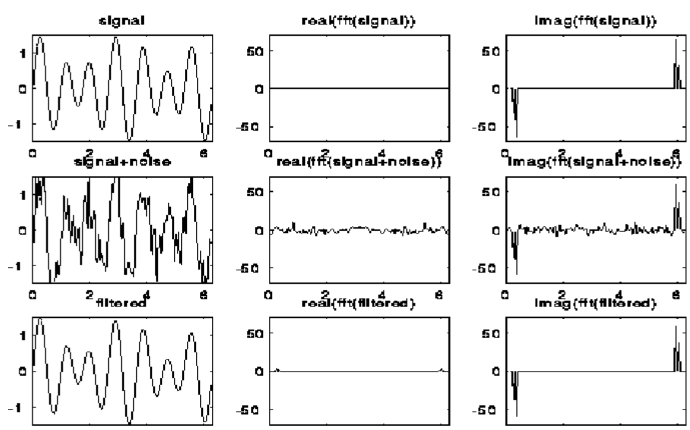
$$\varpi = e^{(2\pi i/m)} = \cos(2\pi/m) + i*\sin(2\pi/m)$$

- This is a complex number with whose m^{th} power is 1 and is therefore called the m^{th} root of unity
- E.g., for m = 4:

$$\varpi = 0+1*i,\ \varpi^2 = -1+0*i,\ \varpi^3 = 0-1*i,\ \varpi^4 = 1+0*i,$$

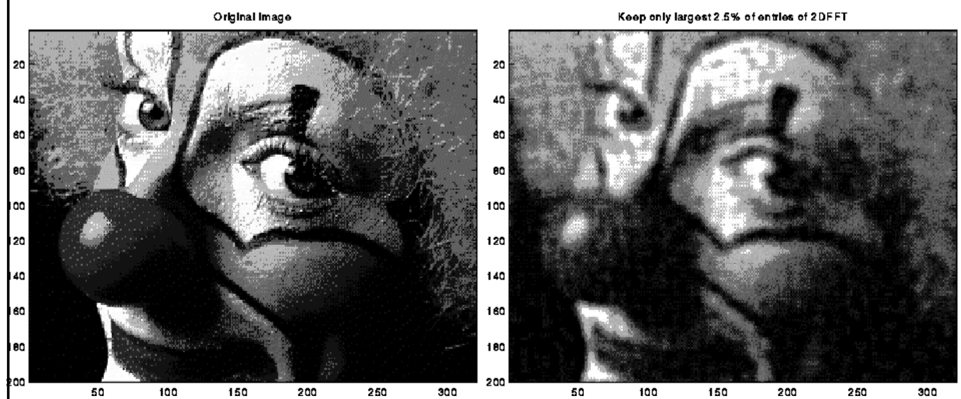
Using the 1D FFT for filtering

- Signal = sin(7t) + .5 sin(5t) at 128 points
- Noise = random number bounded by .75
- Filter by zeroing out FFT components < .25



Using the 2D FFT for image compression

- Image = 200x320 matrix of values
- Compress by keeping largest 2.5% of FFT components



Related Transforms

- Most applications require multiplication by both F and inverse(F).
- Multiplying by F and inverse(F) are essentially the same. (inverse(F) is the complex conjugate of F divided by n.)
- For solving the Poisson equation and various other applications, we use variations on the FFT
  - The sin transform -- imaginary part of F
  - The cos transform -- real part of F
- Algorithms are similar, so we will focus on the forward FFT.

Serial Algorithm for the FFT

° Compute the FFT of an m-element vector v, F*v

$$(F*v)[j] = \sum_{k=0}^{m-1} F(j,k)*v(k)$$
$$= \sum_{k=0}^{m-1} \varpi^{(j*k)} * v(k)$$
$$= \sum_{k=0}^{m-1} (\varpi^{j})^{k} * v(k)$$
$$= V(\varpi^{j})$$

° Where V is defined as the polynomial

$$V(x) = \sum_{k=0}^{m-1} x^k * v(k)$$

Divide and Conquer FFT

° V can be evaluated using divide-and-conquer

$$V(x) = \sum_{k=0}^{m-1} (x)^k * v(k)$$
$$= v[0] + x^2*v[2] + x^4*v[4] + \dots$$
$$+ x*(v[1] + x^2*v[3] + x^4*v[5] + \dots)$$
$$= V_{even}(x^2) + x*V_{odd}(x^2)$$

° V has degree m, so V_{even} and V_{odd} are polynomials of degree m/2-1

° We evaluate these at points $(\varpi^{j})^2$ for 0<=j<=m-1

° But this is really just m/2 different points, since

$$(\varpi^{(j+m/2)})^2 = (\varpi^{j} * \varpi^{m/2})^2 = (\varpi^{2j} * \varpi) = (\varpi^{j})^2$$

Divide-and-Conquer FFT

FFT(v, v, m)

if m = 1 return v[0]

else

v_{even} = FFT(v[0:2:m-2], ϖ^2, m/2)

v_{odd} = FFT(v[1:2:m-1], ϖ^2, m/2)

ϖ-vec = [ϖ^0, ϖ^1, … $\varpi^{(m/2-1)}$] precomputed

return [v_{even} + (ϖ-vec .* v_{odd}),

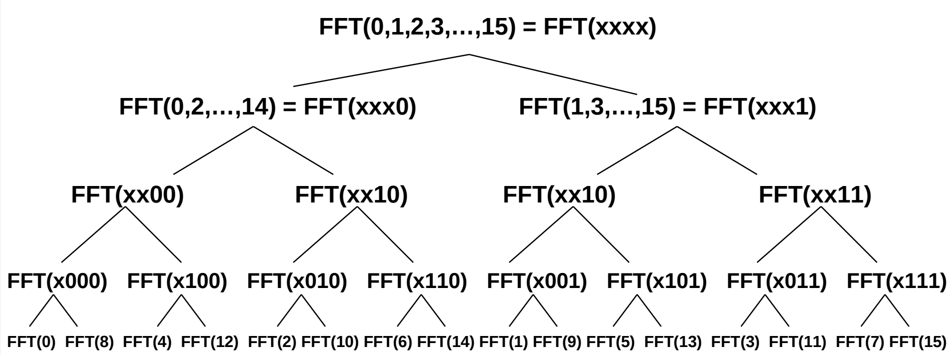
v_{even} - (ϖ-vec .* v_{odd})]

- **The .* above is component-wise multiply.**
- **The […,…] is construction an m-element vector from 2 m/2 element vectors**

This results in an O(m log m) algorithm.

An Iterative Algorithm

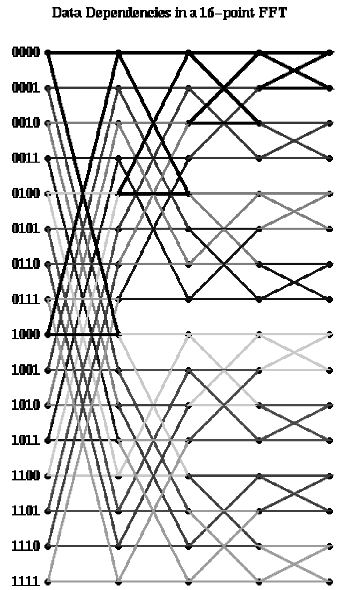
- **The call tree of the d&c FFT algorithm is a complete binary tree of log m levels**



- **Practical algorithms are iterative, going across each level in the tree starting at the bottom**
- **Algorithm overwrites v[i] by (F*v)[bitreverse(i)]**

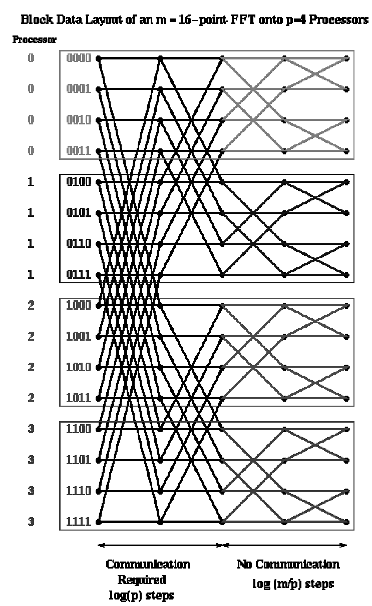
Parallel 1D FFT

- Data dependencies in 1D FFT
  - Butterfly pattern
- A PRAM algorithm takes O(log m) time
  - each step to right is parallel
  - there are log m steps
- What about communication cost?
- See LogP paper for details



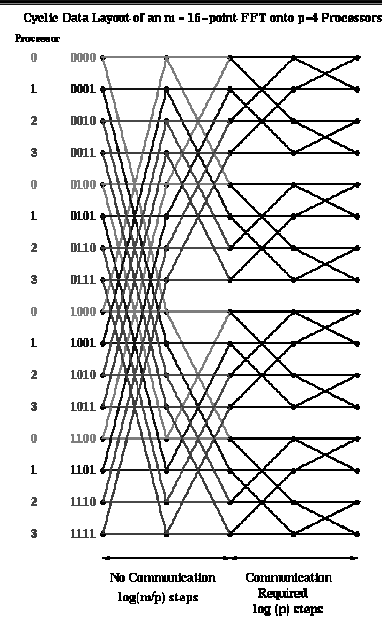
Block Layout of 1D FFT

- Using a block layout (m/p contiguous elts per processor)
- No communication in last log m/p steps
- Each step requires fine-grained communication in first log p steps



Cyclic Layout of 1D FFT

- Cyclic layout (only 1 element per processor, wrapped)
- No communication in first log(m/p) steps
- Communication in last log(p) steps



Parallel Complexity

- m = vector size, p = number of processors
- f = time per flop = 1
- α = startup for message (in f units)
- β = time per word in a message (in f units)

- Time(blockFFT) = Time(cyclicFFT) =

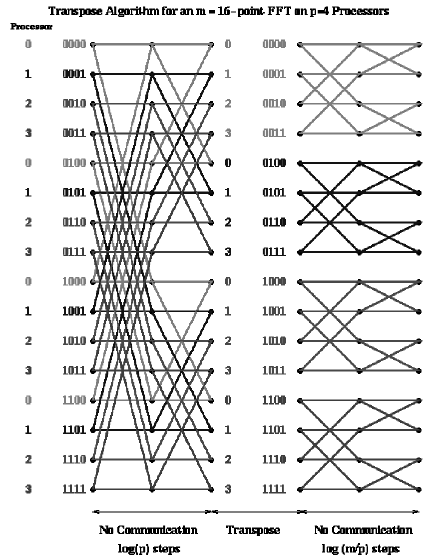
   2*m*log(m)/p

   + log(p) * α

   + m*log(p)/p * β

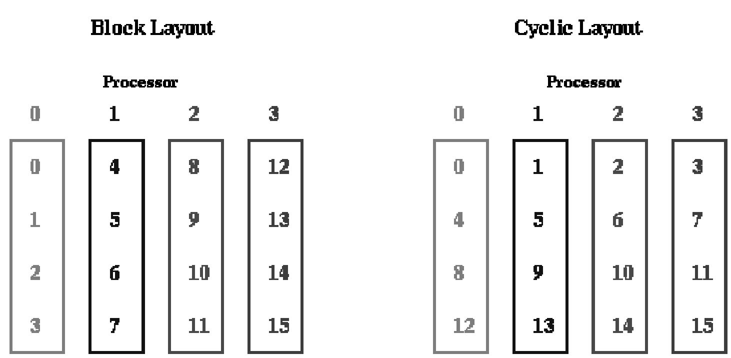
FFT With “Transpose”

- If we start with a cyclic layout for first log(p) steps, there is no communication
- Then transpose the vector for last log(m/p) steps
- All communication is in the transpose



Why is the Communication Step Called a Transpose?

- Analogous to transposing an array
- View as a 2D array of n/p by p
- Note: same idea is useful for uniprocessor caches



Complexity of the FFT with Transpose

° If communication is not overlapped

° Time(transposeFFT) =

2*m*log(m)/p	same as before
+ (p-1) * α	was log(p) * α
+ m*(p-1)/p^2 * β	was m* log(p)/p $*$ β

° Transpose version sends less data, but more messages

° If communication is overlapped, so we do not pay for p-1 messages, the second term becomes simply α, rather than (p-1)α.

° This is close to optimal. See LogP paper for details.

Comment on the 1D Parallel FFT

° The above algorithm leaves data in bit-reversed order

- Some applications can use it this way, like Poisson
- Others require another transpose-like operation
- Is the computation location-dependent?

° Other parallel algorithms also exist

- A very different 1D FFT is due to Edelman (see http://www-math.mit.edu/~edelman)
- Based on the Fast Multipole algorithm
- Less communication for non-bit-reversed algorithm

Higher Dimension FFTs

° **FFTs on 2 or 3 dimensions are define as 1D FFTs on vectors in all dimensions.**

° **E.g., a 2D FFT does 1D FFTs on all rows and then all columns**

° **There are 3 obvious possibilities for the 2D FFT:**

- **(1) 2D blocked layout for matrix, using 1D algorithms for each row and column**
- **(2) Block row layout for matrix, using serial 1D FFTs on rows, followed by a transpose, then more serial 1D FFTs**
- **(3) Block row layout for matrix, using serial 1D FFTs on rows, followed by parallel 1D FFTs on columns**
- **Option 1 is best**

° **For a 3D FFT the options are similar**

- **2 phases done with serial FFTs, followed by a transpose for 3rd**
- **can overlap communication with 2nd phase in practice**