CS 267: Applications of Parallel Computers

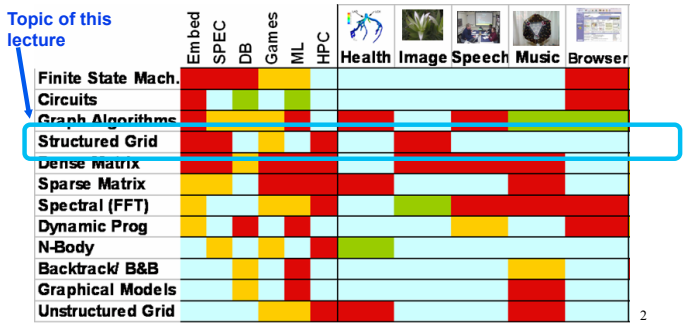
Lecture 17 - Structured Grids

James Demmel

www.cs.berkeley.edu/~demmel/cs267_Spr16

Motifs

The Motifs (formerly "Dwarfs") from "The Berkeley View" (Asanovic et al.)
Motifs form key computational patterns



Outline

- Review of Poisson Equation
- Jacobi's method
- Red-Black SOR method
- Conjugate Gradient (topic of Lecture 15)
- Multigrid
- (Later lecture: FFTs)

Solving PDEs

- Hyperbolic problems (waves):
  - Sound wave(position, time)
  - Use explicit time-stepping
  - Solution at each point depends on neighbors at previous time
- Elliptic (steady state) problems:
  - Electrostatic Potential (position)
  - Everything depends on everything else
  - This means locality is harder to find than in hyperbolic problems
- Parabolic (time-dependent) problems:
  - Temperature(position,time)
  - Involves an elliptic solve at each time-step
- Focus on elliptic problems
  - Canonical example is the Poisson equation

$$\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 + \partial^2 u/\partial z^2 = f(x,y,z)$$

Explicit Solution of PDEs

° Often used for hyperbolic PDEs

° Stability limits size of time-step

° Computation corresponds to
- Matrix vector multiply
- Combine nearest neighbors on grid

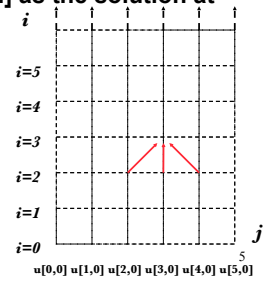
° Use finite differences with u[j,i] as the solution at
- time t= i*δ (i = 0,1,2,…) and
- position x = j*h (j=0,1,…,N=1/h)
- initial conditions on u[j,0]
- boundary conditions on u[N,i] and u[N,i]

° At each timestep i = 0,1,2,...

For j=1 to N-1

u[j,i+1]= z*u[j-1,i]+ (1-2*z)*u[j,i] + z*u[j+1,i]

where z =C*δ/h^2



Matrix View of Explicit Method for Heat Equation

- u[j,i+1]= z*u[j-1,i]+ (1-2*z)*u[j,i] + z*u[j+1,i]
- u[:, i+1] = T * u[:, i] where T is tridiagonal:

$$T = \begin{pmatrix} 1-2z & z & & & \\ z & 1-2z & z & & \\ & z & 1-2z & z & \\ & & z & 1-2z & z \\ & & & z & 1-2z \end{pmatrix} = I - z*L, \quad L = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

Graph and "3 point stencil"

- L called Laplacian (in 1D)
- For a 2D mesh (5 point stencil) the Laplacian has 5 diagonals
- For a 3D mesh there are 7 diagonals

Poisson's equation in 1D: $\partial^2 u/\partial x^2 = f(x)$

Solve Tu = f for u where

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

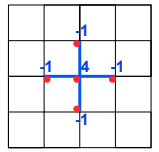
Graph and "stencil"

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 (3D) Poisson Equation (N = n^2 (n^3) vars)

Algorithm	Serial	PRAM	Memory	#Procs
° Dense LU	N^3	N	N^2	N^2
° Band LU	N^2 ($N^{7/3}$)	N	$N^{3/2}$ ($N^{5/3}$)	N ($N^{4/3}$)
° Jacobi	N^2 ($N^{5/3}$)	N ($N^{2/3}$)	N	N
° Explicit Inv.	N^2	log N	N^2	N^2
° Conj.Gradients	$N^{3/2}$ ($N^{4/3}$)	$N^{1/2}$($1/3$) *log N	N	N
° Red/Black SOR	$N^{3/2}$ ($N^{4/3}$)	$N^{1/2}$ ($N^{1/3}$)	N	N
° Sparse LU	$N^{3/2}$ (N^2)	$N^{1/2}$	N*log N ($N^{4/3}$)	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

Reference: James Demmel, Applied Numerical Linear Algebra, SIAM, 1997.

Algorithms for 2D (3D) Poisson Equation (N = n^2 (n^3) vars)

Algorithm	Serial	PRAM	Memory	#Procs
° Dense LU	N^3	N	N^2	N^2
° Band LU	N^2 ($N^{7/3}$)	N	$N^{3/2}$ ($N^{5/3}$)	N ($N^{4/3}$)
° Jacobi	N^2 ($N^{5/3}$)	N ($N^{2/3}$)	N	N
° Explicit Inv.	N^2	log N	N^2	N^2
° Conj.Gradients	$N^{3/2}$ ($N^{4/3}$)	$N^{1/2}$($1/3$) *log N	N	N
° Red/Black SOR	$N^{3/2}$ ($N^{4/3}$)	$N^{1/2}$ ($N^{1/3}$)	N	N
° Sparse LU	$N^{3/2}$ (N^2)	$N^{1/2}$	N*log N ($N^{4/3}$)	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

Reference: James Demmel, Applied Numerical Linear Algebra, SIAM, 1997.

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)

° Steps to converge proportional to problem size, $N=n^2$

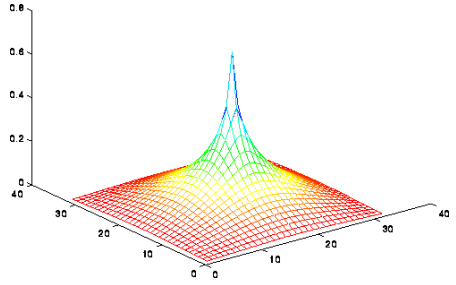
° Therefore, serial complexity is $O(N^2)$

Convergence of Nearest Neighbor Methods

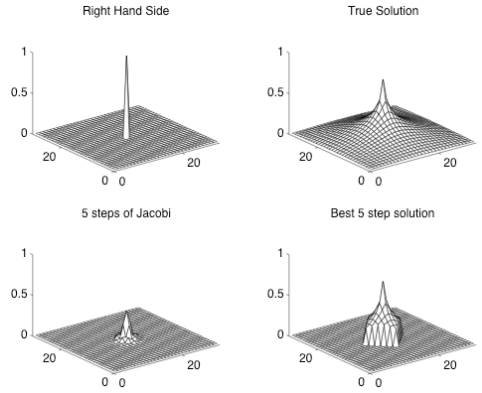
° Jacobi's method involves nearest neighbor computation on nxn grid ($N = n^2$)
- So it takes O(n) = O(sqrt(N)) iterations for information to propagate

° E.g., consider a rhs (b) that is 0, except the center is 1

° The exact solution looks like:



Convergence of Nearest Neighbor Methods



Takes O(n) steps to propagate information across an nxn grid

Parallelizing Jacobi's Method

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

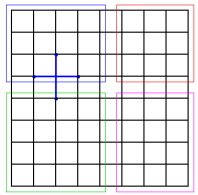
U(m+1) = (T/4 - I) * U(m) + B/4

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

- **keep 2 copies for iterations m and m+1**

° **Requires that boundary values be communicated**

- **each processor owns n^2/p elements to update**
- **amount of data communicated, $n/p^{1/2}$ per neighbor, relatively small if n>>p**



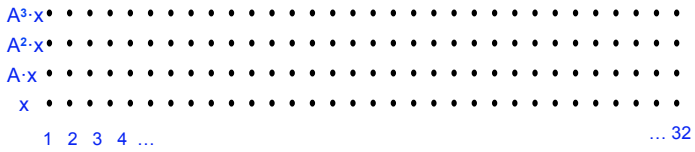
Want to take s>>1 iterations

All the communication-avoiding techniques for Matrix-powers kernel (i.e. repeated SpMVs) from Lecture 15 may be used

Reduce communication cost of s iterations to 1 iteration

Communication Avoiding Jacobi:

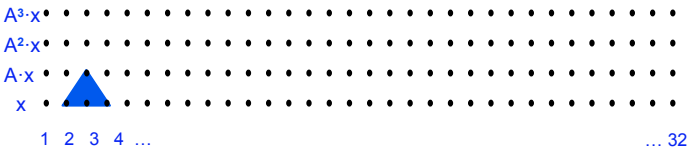
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Example: A tridiagonal, n=32, k=3
- Like Matrix-Powers Kernel, but simpler:
  - Don't need to store A explicitly (it's Jacobi)
  - Only need to save A^kx

Communication Avoiding Jacobi:

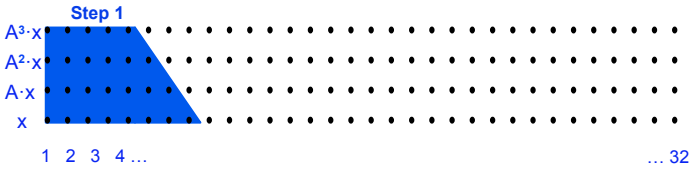
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Example: A tridiagonal, n=32, k=3
- Like Matrix-Powers Kernel, but simpler:
  - Don't need to store A explicitly (it's Jacobi)
  - Only need to save A^kx

Communication Avoiding Jacobi:

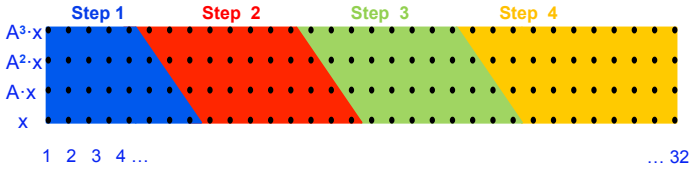
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]
- Sequential Algorithm



- Example: A tridiagonal, n=32, k=3
- Like Matrix-Powers Kernel, but simpler:
  - Don't need to store A explicitly (it's Jacobi)
  - Only need to save A^kx

Communication Avoiding Jacobi:

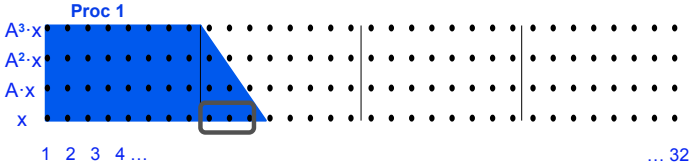
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]
- Sequential Algorithm



- Example: A tridiagonal, n=32, k=3
- Like Matrix-Powers Kernel, but simpler:
  - Don't need to store A explicitly (it's Jacobi)
  - Only need to save A^kx – move O(n) words instead of O(kn)

Communication Avoiding Jacobi:

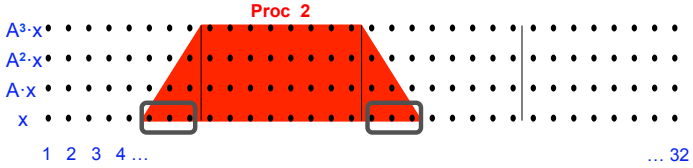
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]
- Parallel Algorithm



- Example: A tridiagonal, n=32, k=3
- Like Matrix-Powers Kernel, but simpler:
  - Don't need to store A explicitly (it's Jacobi)
  - Only need to save A^kx

Communication Avoiding Jacobi:

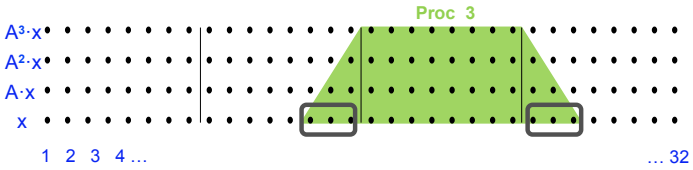
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]
- Parallel Algorithm



- Example: A tridiagonal, n=32, k=3
- Like Matrix-Powers Kernel, but simpler:
  - Don't need to store A explicitly (it's Jacobi)
  - Only need to save A^kx

Communication Avoiding Jacobi:

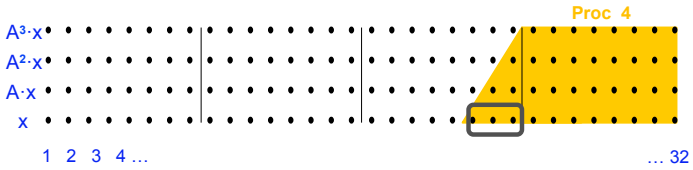
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]
- Parallel Algorithm



- Example: A tridiagonal, n=32, k=3
- Like Matrix-Powers Kernel, but simpler:
  - Don't need to store A explicitly (it's Jacobi)
  - Only need to save A^kx

Communication Avoiding Jacobi:

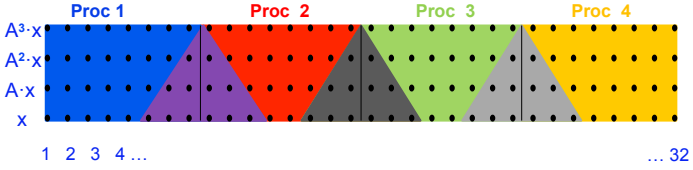
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]
- Parallel Algorithm



- Example: A tridiagonal, n=32, k=3
- Like Matrix-Powers Kernel, but simpler:
  - Don't need to store A explicitly (it's Jacobi)
  - Only need to save A^kx

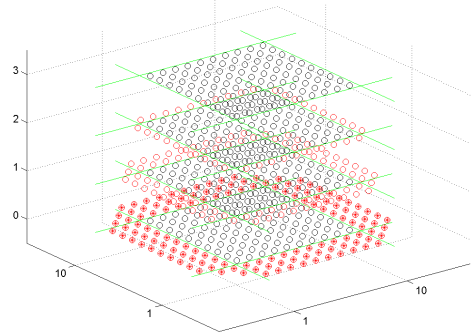
Communication Avoiding Jacobi:

- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]
- Parallel Algorithm



- Example: A tridiagonal, n=32, k=3
- 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)

° 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

° SEJITS – sejits.org (Armando Fox et al @ UCB)
  "Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization"

° Pochoir – stencil compiler (Charles Leiserson @ MIT)
  people.csail.mit.edu/yuantang/

° Autotuning stencils and multigrid (Mary Hall @ Utah)
  super-scidac.org/

° Polyhedral tiling (Michelle Strout @ Colorado)
  www.cs.colostate.edu/~mstrout/Papers/pubs-poly.php

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

Improvements to Jacobi

° Similar to Jacobi: u(i,j,m+1) will be computed as a linear combination of neighbors
  - Numeric coefficients and update order are different

° 2 improvements
  - 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) …

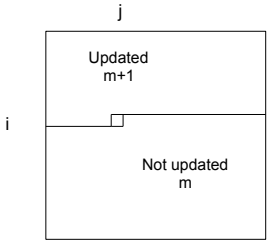
  some of the values for m+1 are already available
  - u(i,j,m+1) = (u(i-1,j,latest) + u(i+1,j,latest) …

  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
```



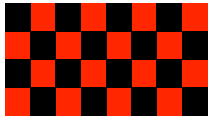
° Cannot be parallelized, because of dependencies

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

° For general graph, use "graph coloring"
- °Can use repeated Maximal Independent Sets to color
- ◦ Graph(T) is bipartite => 2 colorable (red and black)
- ◦ Nodes for each color can be updated simultaneously
- ° Same optimizations, 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

° Can prove w = 2/(1+sin(π/(n+1))) for best convergence for Poisson
- 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 Algorithm for Solving Ax=b

° Initial guess x

° r = b – A*x, j=1

° Repeat
- rho = r^T*r … dot product
- If j=1, p = r, else beta = rho/old_rho, p = r + beta*p, endif … saxpy
- q = A*p … sparse matrix vector multiply, or stencil
- alpha = rho / p^T * q … dot product
- x = x + alpha * p … saxpy
- r = r – alpha * q … saxpy
- old_rho = rho; j=j+1

° Until rho small enough

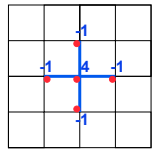
• Converges in $O(n) = O(N^{1/2})$ steps, like SOR, but more general
• Can be reorganized to use matrix powers kernel [Ax,A^2x,…,A^kx]
- "Communication Avoiding Krylov Subspace Methods," M. Hoemmen, UCB PhD Thesis, bebop.cs.berkeley.edu, 2010

2D Poisson's equation

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

$$T = \begin{pmatrix} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{pmatrix}$$

Graph and "stencil"



° 3D is analogous

Algorithms for 2D (3D) Poisson Equation (N = n^2 (n^3) vars)

Algorithm	Serial	PRAM	Memory	#Procs
° Dense LU	N^3	N	N^2	N^2
° Band LU	N^2 ($N^{7/3}$)	N	$N^{3/2}$ ($N^{5/3}$)	N ($N^{4/3}$)
° Jacobi	N^2 ($N^{5/3}$)	N ($N^{2/3}$)	N	N
° Explicit Inv.	N^2	log N	N^2	N^2
° Conj.Gradients	$N^{3/2}$ ($N^{4/3}$)	$N^{1/2}$(1/3) *log N	N	N
° Red/Black SOR	$N^{3/2}$ ($N^{4/3}$)	$N^{1/2}$ ($N^{1/3}$)	N	N
° Sparse LU	$N^{3/2}$ (N^2)	$N^{1/2}$	N*log N ($N^{4/3}$)	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

Reference: James Demmel, Applied Numerical Linear Algebra, SIAM, 1997.

Algorithms for 2D (3D) Poisson Equation (N = n^2 (n^3) vars)

Algorithm	Serial	PRAM	Memory	#Procs
° Dense LU	N^3	N	N^2	N^2
° Band LU	N^2 ($N^{7/3}$)	N	$N^{3/2}$ ($N^{5/3}$)	N ($N^{4/3}$)
° Jacobi	N^2 ($N^{5/3}$)	N ($N^{2/3}$)	N	N
° Explicit Inv.	N^2	log N	N^2	N^2
° Conj.Gradients	$N^{3/2}$ ($N^{4/3}$)	$N^{1/2}$(1/3) *log N	N	N
° Red/Black SOR	$N^{3/2}$ ($N^{4/3}$)	$N^{1/2}$ ($N^{1/3}$)	N	N
° Sparse LU	$N^{3/2}$ (N^2)	$N^{1/2}$	N*log N ($N^{4/3}$)	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

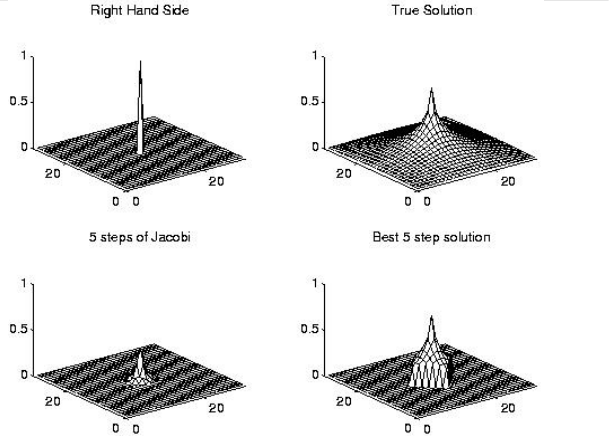
Reference: James Demmel, Applied Numerical Linear Algebra, SIAM, 1997.

Multigrid Motivation

° Recall that Jacobi, SOR, CG, or any other sparse-matrix-vector-multiply-based algorithm can only move information one grid cell at a time
- Take at least n steps to move information across n x n grid

° Therefore, converging in O(1) steps requires moving information across grid faster than to one neighboring grid cell per step
- One step can't just do sparse-matrix-vector-multiply

Multigrid Motivation



Takes O(n) steps to propagate information across an nxn grid

Big Idea used in multigrid and elsewhere

° If you are far away, problem looks simpler
  • For gravity: approximate earth, distant galaxies, … by point masses

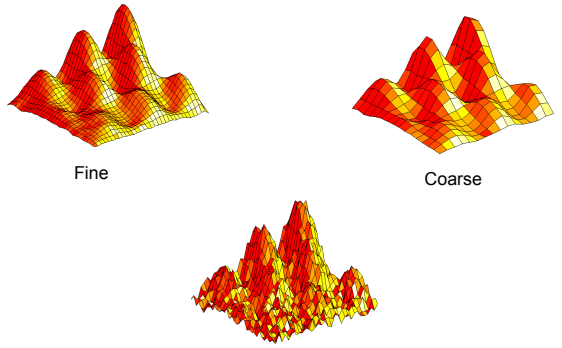
° Can solve such a coarse approximation to get an approximate solution, iterating if necessary
  • Solve coarse approximation problem by using an even coarser approximation of it, and so on recursively

° Ex: Graph Partitioning (used to parallelize SpMV)
  • Replace graph to be partitioned by a coarser graph

° Ex: Multigrid for solving PDE in O(n) time
  • Use coarser mesh to get approximate solution of Poisson's Eq.

° Ex: Fast Multipole Method, Barnes-Hut for computing gravitational forces on n particles in O(n log n) time:
  • Approximate particles in box by total mass, center of gravity

Fine and Coarse Approximations



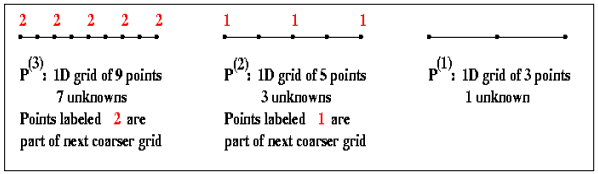
Multigrid Overview

° Basic Algorithm:
  • Replace problem on fine grid by an approximation on a coarser grid
  • Solve the coarse grid problem approximately, and use the solution as a starting guess for the fine-grid problem, which is then iteratively updated
  • Solve the coarse grid problem recursively, i.e. by using a still coarser grid approximation, etc.

° Success depends on coarse grid solution being a good approximation to the fine grid

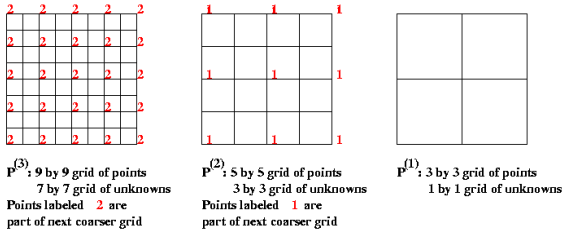
Multigrid Sketch in 1D

• Consider a 2^m+1 grid in 1D for simplicity
• Let $P^{(i)}$ be the problem of solving the discrete Poisson equation on a 2^i+1 grid in 1D (2^i-1 unknowns plus 2 boundaries)
  • Write linear system as T(i) * x(i) = b(i)
• $P^{(m)}$, $P^{(m-1)}$, … , $P^{(1)}$ is sequence of problems from finest to coarsest



Multigrid Sketch in 2D

- Consider a 2^m+1 by 2^m+1 grid in 2D
- Let $P^{(i)}$ be the problem of solving the discrete Poisson equation on a 2^i+1 by 2^i+1 grid in 2D
  - Write linear system as T(i) * x(i) = b(i)
- $P^{(m)}$, $P^{(m-1)}$, … , $P^{(1)}$ is sequence of problems from finest to coarsest



Multigrid Operators

- For problem $P^{(i)}$ at varying coarsening levels (i, grid size grows with i):
  - b(i) is the Right Hand Side (RHS) and
  - x(i) is the current estimated solution

  } both live on grids of size 2^i-1
- All the following operators just average values on neighboring grid points (so information moves fast on coarse grids)
- The restriction operator R(i) maps $P^{(i)}$ to $P^{(i-1)}$
  - Restricts problem on fine grid $P^{(i)}$ to coarse grid $P^{(i-1)}$
  - Uses sampling or averaging
  - b(i-1)= R(i) (b(i))
- The interpolation operator In(i-1) maps approx. solution x(i-1) to x(i)
  - Interpolates solution on coarse grid $P^{(i-1)}$ to fine grid $P^{(i)}$
  - x(i) = In(i-1)(x(i-1))
- The solution operator S(i) takes $P^{(i)}$ and improves solution x(i)
  - Uses "weighted" Jacobi or SOR on single level of grid
  - $x_{improved}$ (i) = S(i) (b(i), x(i))
- Overall algorithm, then details of operators

Multigrid Operators

- For problem $P^{(i)}$ at varying coarsening levels (i, grid size grows with i):
  - b(i) is the Right Hand Side (RHS) and
  - x(i) is the current estimated solution

  } both live on grids of size 2^i-1
- All the following operators just average values on neighboring grid points (so information moves fast on coarse grids)
- The restriction operator R(i) maps $P^{(i)}$ to $P^{(i-1)}$
  - Restricts problem on fine grid $P^{(i)}$ to coarse grid $P^{(i-1)}$
  - Uses sampling or averaging
  - b(i-1)= R(i) (b(i))
- The interpolation operator In(i-1) maps approx. solution x(i-1) to x(i)
  - Interpolates solution on coarse grid $P^{(i-1)}$ to fine grid $P^{(i)}$
  - x(i) = In(i-1)(x(i-1))
- The solution operator S(i) takes $P^{(i)}$ and improves solution x(i)
  - Uses "weighted" Jacobi or SOR on single level of grid
  - $x_{improved}$ (i) = S(i) (b(i), x(i))
- Overall algorithm, then details of operators

Multigrid Operators

- For problem $P^{(i)}$ at varying coarsening levels (i, grid size grows with i):
  - b(i) is the Right Hand Side (RHS) and
  - x(i) is the current estimated solution

  } both live on grids of size 2^i-1
- All the following operators just average values on neighboring grid points (so information moves fast on coarse grids)
- The restriction operator R(i) maps $P^{(i)}$ to $P^{(i-1)}$
  - Restricts problem on fine grid $P^{(i)}$ to coarse grid $P^{(i-1)}$
  - Uses sampling or averaging
  - b(i-1)= R(i) (b(i))
- The interpolation operator In(i-1) maps approx. solution x(i-1) to x(i)
  - Interpolates solution on coarse grid $P^{(i-1)}$ to fine grid $P^{(i)}$
  - x(i) = In(i-1)(x(i-1))
- The solution operator S(i) takes $P^{(i)}$ and improves solution x(i)
  - Uses "weighted" Jacobi or SOR on single level of grid
  - $x_{improved}$ (i) = S(i) (b(i), x(i))
- Overall algorithm, then details of operators

Multigrid Operators

- For problem $P^{(i)}$ at varying coarsening levels (i, grid size grows with i):
  - b(i) is the Right Hand Side (RHS) and
  - x(i) is the current estimated solution

  } both live on grids of size 2^i-1
- All the following operators just average values on neighboring grid points (so information moves fast on coarse grids)
- The restriction operator R(i) maps $P^{(i)}$ to $P^{(i-1)}$
  - Restricts problem on fine grid $P^{(i)}$ to coarse grid $P^{(i-1)}$
  - Uses sampling or averaging
  - b(i-1)= R(i) (b(i))
- The interpolation operator In(i-1) maps approx. solution x(i-1) to x(i)
  - Interpolates solution on coarse grid $P^{(i-1)}$ to fine grid $P^{(i)}$
  - x(i) = In(i-1)(x(i-1))
- The solution operator S(i) takes $P^{(i)}$ and improves solution x(i)
  - Uses "weighted" Jacobi or SOR on single level of grid
  - $x_{improved}$ (i) = S(i) (b(i), x(i))
- Overall algorithm, then details of operators

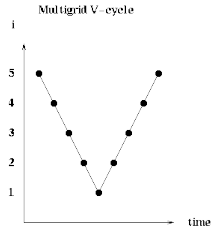
Multigrid V-Cycle Algorithm

```
Function MGV ( b(i), x(i) )
  … Solve T(i)*x(i) = b(i) given b(i) and an initial  guess for x(i)
  … return an improved x(i)
  if (i = 1)
     compute exact solution x(1) of P(1)        only 1 unknown
     return x(1)
  else                                          solve recursively
     x(i) = S(i) (b(i), x(i))                   improve solution by damping
                                                    high frequency error
     r(i)  = T(i)*x(i) - b(i)                   compute residual
     d(i) = In(i-1) ( MGV( R(i) ( r(i) ), 0 ) ) solve T(i)*d(i) = r(i) recursively
     x(i) = x(i) - d(i)                         correct fine grid solution
     x(i) = S(i) ( b(i), x(i) )                 improve solution again
     return x(i)
```


This is called a V-Cycle

° Just a picture of the call graph

° In time a V-cycle looks like the following



Complexity of a V-Cycle

° On a serial machine
- Work at each point in a V-cycle is O(the number of unknowns)
- Cost of Level i is $(2^i\text{-}1)^2 = O(4^i)$ for a 2D grid
- If finest grid level is m, total time is:

$$\sum_{i=1}^{m} O(4^i) = O(4^m) \quad \text{for a 2D grid}$$

$$= O(\#\text{ unknowns}) \text{ in general}$$

° On an ideal parallel machine (PRAM)
- with one processor per grid point and free communication, each step in the V-cycle takes constant time
- Total V-cycle time is O(m) = O(log #unknowns)

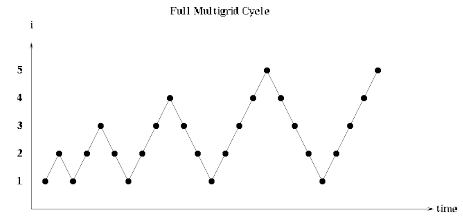
Full Multigrid (FMG)

° Intuition:
- improve solution by doing multiple V-cycles
- avoid expensive fine-grid (high frequency) cycles
- analysis of why this works is beyond the scope of this class

```
Function FMG (b(m), x(m))
    … return improved x(m) given initial guess
    compute the exact solution x(1) of P(1)
    for i=2 to m    … from coarse to fine mesh
        x(i) = MGV ( b(i), In (i-1) (x(i-1) ) )
```

° In other words:
- Solve the problem with 1 unknown
- Given a solution to the coarser problem, $P^{(i-1)}$, map it to starting guess for $P^{(i)}$
- Solve the finer problem using the Multigrid V-cycle

Full Multigrid Cost Analysis



° One V for each call to FMG
- people also use Ws and other compositions

° Serial time: $\Sigma_{i=1}^{m} O(4^i) = O(4^m) = O(\#\text{ unknowns})$

° PRAM time: $\Sigma_{i=1}^{m} O(i) = O(m^2) = O(\log^2 \#\text{ unknowns})$

Complexity of Solving Poisson's Equation

- Theorem: error after one call to multigrid
  - error_after ≤ .5 * error_before
  - independent of # unknowns
  - ➔At least 1 bit each time
- Corollary: We can make the error < any fixed tolerance in a fixed number of steps, independent of size of finest grid
- This is the most important convergence property of MG, distinguishing it from all other methods, which converge more slowly for large grids

Complexity of Solving Poisson's Equation

- Theorem: error after one FMG call
  - error_after ≤ .5 * error_before
  - independent of # unknowns
  - ➔At least 1 bit each time
- Corollary: We can make the error < any fixed tolerance in a fixed number of steps, independent of size of finest grid
- This is the most important convergence property of MG, distinguishing it from all other methods, which converge more slowly for large grids

The Solution Operator S(i) - Details

° The solution operator, S(i), is a weighted Jacobi

° Consider the 1D problem

° At level i, pure Jacobi replaces:

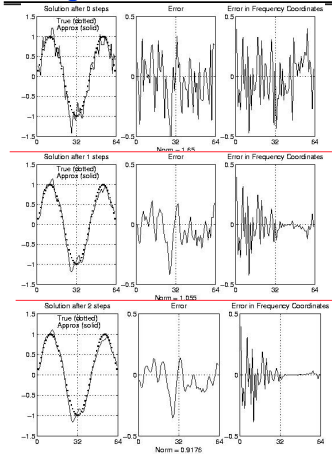
x(j) := 1/2 (x(j-1) + x(j+1) + b(j))

° Weighted Jacobi uses:

x(j) := 1/3 (x(j-1) + x(j) + x(j+1) + b(j))

° In 2D, similar average of nearest neighbors

Weighted Jacobi chosen to damp high frequency error



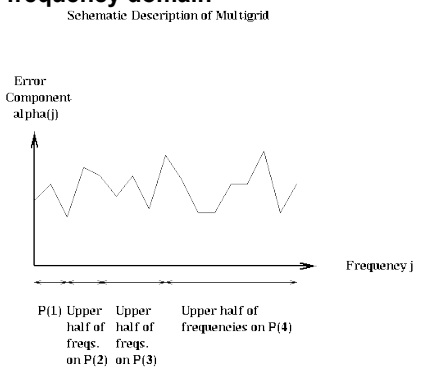
Initial error
"Rough"
Lots of high frequency components
Norm = 1.65

Error after 1 weighted Jacobi step
"Smoother"
Less high frequency component
Norm = 1.06

Error after 2 weighted Jacobi steps
"Smooth"
Little high frequency component
Norm = .92,
won't decrease much more

Multigrid as Divide and Conquer Algorithm

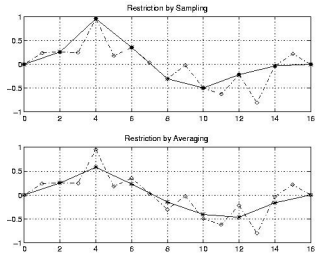
° Each level in a V-Cycle reduces the error in one part of the frequency domain



The Restriction Operator R(i) - Details

° The restriction operator, R(i), takes
- a problem $P^{(i)}$ with RHS b(i) and
- maps it to a coarser problem $P^{(i-1)}$ with RHS b(i-1)

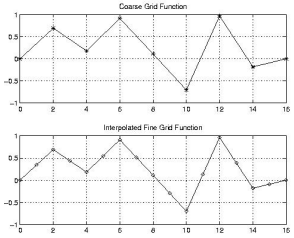
° In 1D, average values of neighbors
- $x_{coarse}(i) = 1/4 * x_{fine}(i-1) + 1/2 * x_{fine}(i) + 1/4 * x_{fine}(i+1)$



° In 2D, average with all 8 neighbors (N,S,E,W,NE,NW,SE,SW)

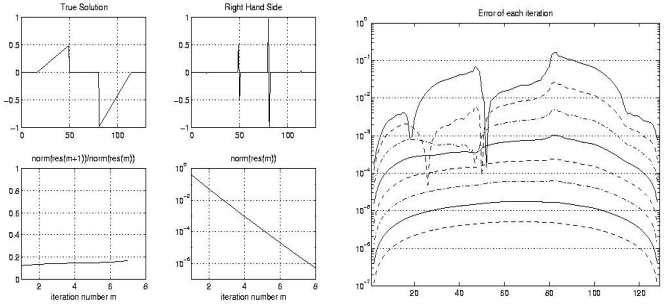
Interpolation Operator In(i-1): details

- The interpolation operator In(i-1), takes a function on a coarse grid $P^{(i-1)}$, and produces a function on a fine grid $P^{(i)}$
- In 1D, linearly interpolate nearest coarse neighbors
  - $x_{fine}(i) = x_{coarse}(i)$ if the fine grid point i is also a coarse one, else
  - $x_{fine}(i) = 1/2 * x_{coarse}(\text{left of i}) + 1/2 * x_{coarse}(\text{right of i})$

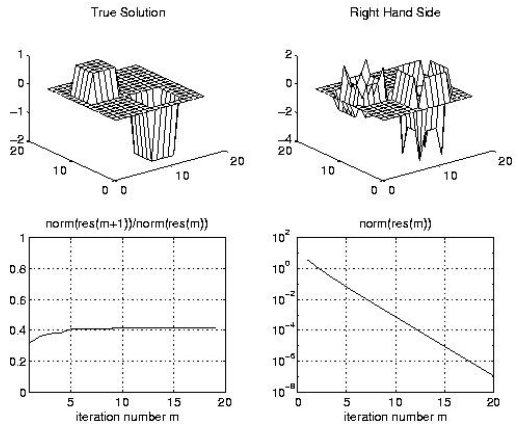


- In 2D, interpolation requires averaging with 4 nearest neighbors (NW,SW,NE,SE)

Convergence Picture of Multigrid in 1D

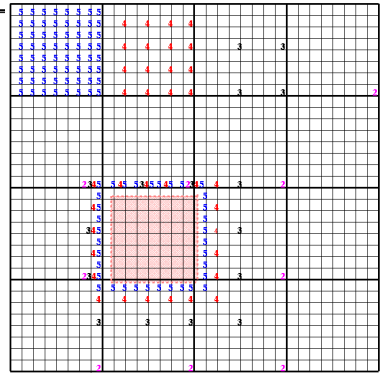


Convergence Picture of Multigrid in 2D



Parallel 2D Multigrid

- Multigrid on 2D requires nearest neighbor (up to 8) computation at each level of the grid
- Start with $n=2^m+1$ by 2^m+1 grid (here m=5)
- Use an s by s processor grid (here s=4)



Communication pattern for Multigrid on 33 by 33 mesh with 4 by 4 processor grid
In top processor row, grid points labeled m are updated in problem P(m) of multigrid
Pink processor owns grid points inside pink box
In lower half of graph, grid points labeled m need to be communicated to pink processor in problem P(m) of multigrid

Performance Model of parallel 2D Multigrid (V-cycle)

° Assume 2^m+1 by 2^m+1 grid of points, n= 2^m-1, N=n^2

° Assume p = 4^k processors, arranged in 2^k by 2^k grid
- Processors start with 2^{m-k} by 2^{m-k} subgrid of unknowns

° Consider V-cycle starting at level m
- At levels m through k of V-cycle, each processor does some work
- At levels k-1 through 1, some processors are idle, because a 2^{k-1} by 2^{k-1} grid of unknowns cannot occupy each processor

° Cost of one level in V-cycle
- If level j >= k, then cost =
  - $O(4^{j-k})$ .... Flops, proportional to the number of grid points/processor
  - + $O(1)\ \alpha$ .... Send a constant # messages to neighbors
  - + $O(2^{j-k})\ \beta$ .... Number of words sent
- If level j < k, then cost =
  - $O(1)$ .... Flops, proportional to the number of grid points/processor
  - + $O(1)\ \alpha$ .... Send a constant # messages to neighbors
  - + $O(1)\ \beta$ .... Number of words sent

° Sum over all levels in all V-cycles to get complexity

Comparison of Methods (in O(.) sense)

	# Flops	# Messages	# Words sent
MG	N/p + log p * log N	$(\log N)^2$	$(N/p)^{1/2}$ + log p * log N
FFT	N log N / p	$p^{1/2}$	N/p
SOR	$N^{3/2}$ /p	$N^{1/2}$	N/p

° SOR is slower than others on all counts

° Flops for MG depends on accuracy of MG

° MG communicates less total data (bandwidth)

° Total messages (latency) depends …
- This coarse analysis can't say whether MG or FFT is better when $\alpha >> \beta$

Practicalities

° In practice, we don't go all the way to $P^{(1)}$

° In sequential code, the coarsest grids are negligibly cheap, but on a parallel machine they are not.
- Consider 1000 points per processor, so flops = O(1000)
- In 2D, the surface to communicate is 4 x $1000^{1/2}$ ~= 128, or 13%
- In 3D, the surface is 1000-8^3 ~= 500, or 50%

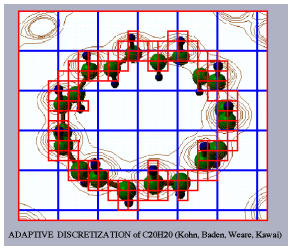
° See Tuminaro and Womble, SIAM J. Sci. Comp., v14, n5, 1993 for analysis of MG on 1024 nCUBE2
- on 64x64 grid of unknowns, only 4 per processor
  - efficiency of 1 V-cycle was .02, and on FMG .008
- on 1024x1024 grid
  - efficiencies were .7 (MG V-cycle) and .42 (FMG)
  - although worse parallel efficiency, FMG is 2.6 times faster than V-cycles alone
- nCUBE had fast communication, slow processors

° Today: Same problem in Chombo @ LBL
- Communication of coarsest meshes

Multigrid on an Adaptive Mesh

° For problems with very large dynamic range, another level of refinement is needed

° Build adaptive mesh and solve multigrid (typically) at each level

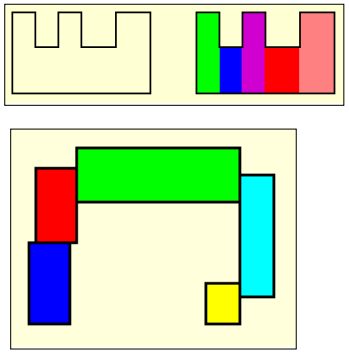


° Can't afford to use finest mesh everywhere

Multiblock Applications

° Solve system of equations on a union of rectangles
  • subproblem of AMR

° E.g.,

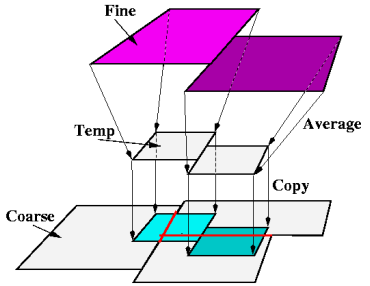


Adaptive Mesh Refinement

° Data structures in AMR

° Usual parallelism is to assign grids on each level to processors

° Load balancing is a problem

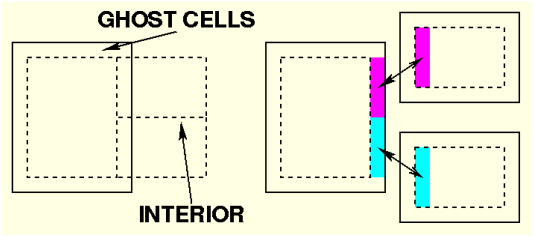


Support for AMR

° Domains in Titanium designed for this problem

° Kelp, Boxlib, and AMR++ are libraries for this

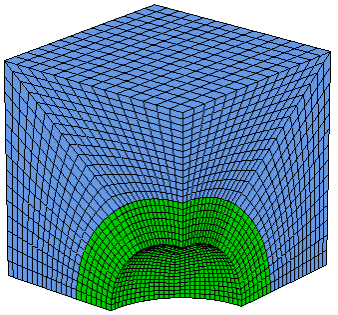
° Primitives to help with boundary value updates, etc.



Multigrid on an Unstructured Mesh

° Another approach to variable activity is to use an unstructured mesh that is more refined in areas of interest

° Adaptive rectangular or unstructured?
  • Numerics easier on rectangular
  • Supposedly easier to implement (arrays without indirection) but boundary cases tend to dominate code



Up to 39M unknowns on 960 processors,
With 50% efficiency (Source: M. Adams)

Multigrid on an Unstructured Mesh

° Need to partition graph for parallelism

° What does it mean to do Multigrid anyway?

° Need to be able to coarsen grid (hard problem)
- Can't just pick "every other grid point" anymore
- Use "maximal independent sets" again
- How to make coarse graph approximate fine one

° Need to define R() and In()
- How do we convert from coarse to fine mesh and back?

° Need to define S()
- How do we define coarse matrix (no longer formula, like Poisson)

° Dealing with coarse meshes efficiently
- Should we switch to using fewer processors on coarse meshes?
- Should we switch to another solver on coarse meshes?

Irregular mesh: Tapered Tube (multigrid)

Example of Prometheus meshes

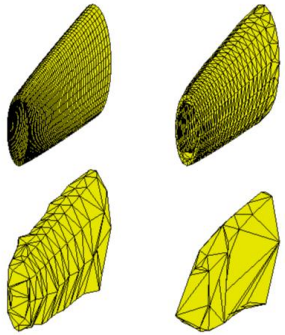


Figure 6 Sample input grid and coarse grids

Source of Unstructured Finite Element Mesh: Vertebra

Study failure modes of trabecular bone under stress

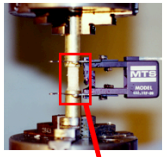
Source: M. Adams, H. Bayraktar, T. Keaveny, P. Papadopoulos, A. Gupta

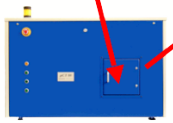
Multigrid for nonlinear elastic analysis of bone

Gordon Bell Prize, 2004

Source: M. Adams et al

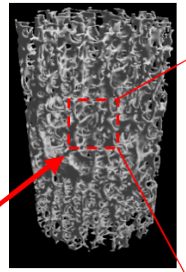
Mechanical testing for material properties



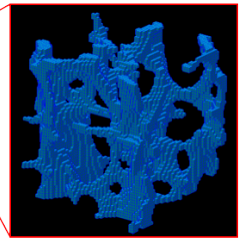


Micro Computed Tomography @ 22 μm resolution

3D image



μFE mesh
2.5 mm^3
44 μm elements



Up to
537M unknowns
4088 Processors (ASCI White)
70% parallel efficiency