Lecture 8: Linear Algebra Algorithms (Continued)

Jack Dongarra, U of Tennessee

Review of Gaussian Elimination (GE) for solving $Ax=b$

- Add multiples of each row to later rows to make $A$ upper triangular
- Solve resulting triangular system $Ux=c$ by substitution

Refine GE Algorithm (1)

- Initial Version
  
  for each column $i$
  for each row $j$ below row $i$
  for each row $i$
  zero it out below the diagonal by adding multiples of row $i$ to later rows

- Remove computation of constant $A(j,i)/A(i,i)$ from inner loop

Refine GE Algorithm (2)

- Last version

  for each column $i$
  for each row $j$ below row $i$
  for each row $i$
  zero it out below the diagonal by adding multiples of row $i$ to later rows

  Don’t compute what we already know: zeros below diagonal in column $i$

Refine GE Algorithm (3)

- Last version

  for each column $i$
  for each row $j$ below row $i$
  for each row $i$
  zero it out below the diagonal by adding multiples of row $i$ to later rows

  Store multipliers $m$ below diagonal in zeroed entries for later use

Refine GE Algorithm (4)

- Last version

  for each column $i$
  for each row $j$ below row $i$
  for each row $i$
  zero it out below the diagonal by adding multiples of row $i$ to later rows

  Express using matrix operations (BLAS)
What GE really computes

Call the strictly lower triangular matrix of multipliers \( M \), and let \( L = I + M \).

Lemma (LU Factorization): If the above algorithm terminates (does not divide by zero) then \( A = L^*U \).

Solving \( A^*x = b \) using GE

\( \begin{align*} &\text{Factorize } A = L^*U \text{ using GE} \quad (\text{cost } \approx 2/3 n^3 \text{ flops}) \; \; \quad \text{(L)} \\ &\text{Solve } L^*y = b \text{ for } y, \text{ using substitution} \quad (\text{cost } \approx n^2 \text{ flops}) \; \; \quad \text{(U)} \\ &\text{Solve } U^*x = y \text{ for } x, \text{ using substitution} \quad (\text{cost } \approx n^2 \text{ flops}) \; \; \quad \text{(S)} \end{align*} \)

Thus \( A^*x = (L^*U)^*x = L^*(U^*x) = L^*y = b \) as desired.

Problems with basic GE algorithm

What if some \( A(i,i) \) is zero? Or very small?

• Result may not exist, or be "unstable", so need to pivot

Current computation all BLAS 1 or BLAS 2, but we know that BLAS 3 (matrix multiply) is fastest (Lecture 2)

\( \begin{align*} &\text{for } i = 1 \text{ to } n-1 \quad \text{ ... BLAS 1 (scale a vector)} \\ &\quad A(i+1:n,i) = A(i+1:n,i) / A(i,i) \quad \text{(L)} \\ &\quad A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - A(i+1:n,i) * A(i,i+1:n) \quad \text{(U)} \\ &\text{end for} \end{align*} \)

Peak

BLAS 3

BLAS 2

BLAS 1

Converting BLAS2 to BLAS3 in GEPP

• Blocking
  • Used to optimize matrix-multiplication
  • Harder here because of data dependencies in GEPP

• Delayed Updates
  • Save updates to "trailing matrix" from several consecutive BLAS2 updates
  • Apply many saved updates simultaneously in one BLAS3 operation

• Same idea works for much of dense linear algebra
  • Open questions remain
  • Need to choose a block size \( b \)
    • Algorithm will save and apply \( b \) updates
    • \( b \) must be small enough so that active submatrix consisting of \( b \) columns of \( A \) fits in cache
    • \( b \) must be large enough to make BLAS3 fast

Overview of LAPACK

• Standard library for dense/banded linear algebra
  • Linear systems: \( A^*x = b \)
  • Least squares problems: \( \min_x \| A^*x - b \|_2 \)
  • Eigenvalue problems: \( Ax = \lambda x, \quad Ax = \lambda Bx \)
  • Singular value decomposition (SVD): \( A = U^*V^T \)

• Algorithms reorganized to use BLAS3 as much as possible
  • Basis of math libraries on many computers
  • Many algorithmic innovations remain
  • Projects available
### Blocked Partitioned Algorithms

- LU Factorization
- Cholesky factorization
- Symmetric indefinite factorization
- Matrix inversion
- QR, QL, RQ, LQ factorizations
- Form $Q$ or $Q^T$

### Memory Hierarchy and LAPACK

- $ijk$ - implementations
  - Effects order in which data referenced; some better at allowing data to keep in higher levels of memory hierarchy.
  - Applies for matrix multiply, reductions to condensed form
    - May do slightly more
    - Up to 3 times faster

### LINPACK Implementation

- Here is the body of the LINPACK routine SPOFA which implements the method:

```fortran
DO 30 J = 1, N
   INFO = J
   S = 0.0E0
   JM1 = J - 1
   IF( JM1.LT.1 ) GO TO 20
   DO 10 K = 1, JM1
      T = A( K, J ) - SDOT( K-1, A( 1, K ), 1,A( 1, J ), 1 )
      T = T / A( K, K )
      A( K, J ) = T
      S = S + T*T
   10 CONTINUE
   S = A( J, J ) - S
   C    ...EXIT
   IF( S.LE.0.0E0 ) GO TO 40
   A( J, J ) = SQRT( S )
   30 CONTINUE
```

### LAPACK Implementation

- This change by itself is sufficient to significantly improve the performance on a number of machines.
- From 72 to 251 Mflop/s for a matrix of order 500 on one processor of a CRAY Y-MP.
- However on 378 Mflop/s on 8 Procs. Of a CRAY Y-MP.
- Suggest further work needed.

### Derivation of Blocked Algorithms

**Cholesky Factorization** $A = U^TU$

- Equating coefficient of second block of columns, we obtain
  
  $a_j^T U_1^T u_j$
  
  $a_j^T u_j^T a_j$

  Hence, if $U_1$ has already been computed, we can compute $a_j$ and $u_j$ from the equations:

  $U_1^T u_j = a_j$
  
  $u_j^T a_j = u_j^T a_j$

**Derivation of Blocked Algorithms**

- Orthogonal reduction to:
  - (upper) Hessenberg form
  - symmetric tridiagonal form
  - bidiagonal form

- QR iteration for nonsymmetric eigenvalue problems

- Block QR iteration for

**Linpack Implementation**

```fortran
DO 10 J = 1, N
   CALL STRSV( 'Upper', 'Transpose', 'Non -Unit', J-1, A, LDA, A( 1, J ), 1 )
   S = A( J, J ) - SDOT( J -1, A( 1, J ), 1, A( 1, J ), 1 )
   IF( S.LE.ZERO ) GO TO 20
   A( J, J ) = SQRT( S )
   10 CONTINUE
```

**Lapack Implementation**

```fortran
DO 10 J = 1, N
   CALL STRSM( 'Upper', 'Non -Unit', 'Non -Unit', J, A, LDA, A( 1, J ), 1 )
   S = A( J, J ) - SDOT( J -1, A( 1, J ), 1, A( 1, J ), 1 )
   IF( S.LE.ZERO ) GO TO 20
   A( J, J ) = SQRT( S )
   10 CONTINUE
```

**Derivation of Blocked Algorithms**

**Linpack Implementation**

```fortran
DO 10 J = 1, N
   CALL STRSV( 'Upper', 'Transpose', 'Non -Unit', J-1, A, LDA, A( 1, J ), 1 )
   S = A( J, J ) - SDOT( J -1, A( 1, J ), 1, A( 1, J ), 1 )
   IF( S.LE.ZERO ) GO TO 20
   A( J, J ) = SQRT( S )
   10 CONTINUE
```

**Lapack Implementation**

```fortran
DO 10 J = 1, N
   CALL STRSM( 'Upper', 'Non -Unit', 'Non -Unit', J, A, LDA, A( 1, J ), 1 )
   S = A( J, J ) - SDOT( J -1, A( 1, J ), 1, A( 1, J ), 1 )
   IF( S.LE.ZERO ) GO TO 20
   A( J, J ) = SQRT( S )
   10 CONTINUE
```

**Equating coefficient of second block of columns, we obtain**

$A_{12}^T U_{12}^T$

$A_{12}^T U_{12}^T A_{22}^T U_{22}^T$

Hence, if $U_{12}$ has already been computed, we can compute $U_{22}$ as the solution of the following equations by a call to the Level 3 BLAS routine STRSR:

$U_{12}^T U_{12}^T A_{22}^T U_{22}^T A_{22}^T U_{22}^T U_{22}^T$
**LAPACK Blocked Algorithms**

```fortran
DO 10 J = 1, N, NB
   CALL STRSM( 'Left', 'Upper', 'Transpose', 'Non-Unit', J-1, JB, ONE, A, LDA,
$         A( 1, J ), LDA )
   CALL SSYRK( 'Upper', 'Transpose', JB, J-1, -ONE, A( 1, J ), LDA, ONE,
$         A( J, J ), LDA )
   CALL SPOTF2( 'Upper', JB, A( J, J ), LDA, INFO )
   IF( INFO.NE.0 ) GO TO 20
10 CONTINUE
```

• On Y-MP, L3 BLAS squeezes a little more out of 1 proc, but makes a large improvement when using 8 procs.

**Cray Y-MP**

<table>
<thead>
<tr>
<th>Cholesky, n=500</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Proc</strong></td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td><strong>8 Proc</strong></td>
</tr>
<tr>
<td>72</td>
</tr>
</tbody>
</table>

**LINPACK**

<table>
<thead>
<tr>
<th>Cholesky, n=500</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Proc</strong></td>
</tr>
<tr>
<td>251</td>
</tr>
<tr>
<td><strong>8 Proc</strong></td>
</tr>
<tr>
<td>378</td>
</tr>
</tbody>
</table>

**Level 2 BLAS**

<table>
<thead>
<tr>
<th>Cholesky, n=500</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Proc</strong></td>
</tr>
<tr>
<td>287</td>
</tr>
<tr>
<td><strong>8 Proc</strong></td>
</tr>
<tr>
<td>1225</td>
</tr>
</tbody>
</table>

**Level 3 BLAS**

<table>
<thead>
<tr>
<th>Cholesky, n=500</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Proc</strong></td>
</tr>
<tr>
<td>290</td>
</tr>
<tr>
<td><strong>8 Proc</strong></td>
</tr>
<tr>
<td>1414</td>
</tr>
</tbody>
</table>

**LAPACK Contents**

° Combines algorithms from LINPACK and EISPACK into a single package. User interface similar to LINPACK.

° Built on the L 1, 2 and 3 BLAS, for high performance (manufacturers optimize BLAS)

° LAPACK does not provide routines for structured problems or general sparse matrices (i.e. sparse storage formats such as compressed-row, -column, -diagonal, skyline ...).

**LAPACK Ongoing Work**

° Add functionality
  - updating/downdating, divide and conquer least squares, bidiagonal bisection, bidiagonal inverse iteration, band SVD, Jacobi methods, ...

° Move to new generation of high performance machines
  - IBM SPs, CRAY T3E, SGI Origin, clusters of workstations

° New challenges
  - New languages: FORTRAN 90, HP FORTRAN, ...
  - (CMMD, MPL, NX ...) - many flavors of message passing, need standard (PVM, MPI): BLACS

° Highly varying ratio
  - Computational speed
  - Communication speed

° Many ways to layout data.

° Fastest parallel algorithm sometimes less stable numerically.

**History of Block Partitioned Algorithms**

° Early algorithms involved use of small main memory using tapes as secondary storage.

° Recent work centers on use of vector registers, level 1 and 2 cache, main memory, and "out of core" memory.

**Blocked Partitioned Algorithms**

° LU Factorization
° Cholesky factorization
° Symmetric indefinite factorization
° Matrix inversion
° QR, QL, RQ, LQ factorizations
° Form Q or QT
° Orthogonal reduction to:
  - upper Hessenberg form
  - symmetric tridiagonal form
  - bidiagonal form
° Block QR iteration for nonsymmetric eigenvalue problems

**Parallelizing Gaussian Elimination**

parallelization steps

° Decomposition: identify enough parallel work, but not too much
° Assignment: load balance work among threads
° Orchestrating: communication and synchronization
° Mapping: which processors execute which threads
° Decomposition

In BLAS 2 algorithm nearly each flop in inner loop can be done in parallel, so with n² processors, need 3n² parallel steps

for i = 1 to n-1
  BLAS 1 (scale a vector)
  BLAS 2 (rank-1 update)

° This is too fine-grained, prefer calls to local matmuls instead
Assignment of parallel work in GE

° Think of assigning submatrices to threads, where each thread responsible for updating submatrix it owns
  • “owner computes” rule natural because of locality
° What should submatrices look like to achieve load balance?

Different Data Layouts for Parallel GE (on 4 procs)

Bad load balance: P0 idle after first n/4 steps

Load balanced, but can’t easily use BLAS2 or BLAS3

Can trade load balance and BLAS2/3 performance by choosing b, but factorization of block column is a bottleneck

Review: Row and Column Block Cyclic Layout

Processors and matrix blocks are distributed in a 2d array

pcol-fold parallelism in any column, and calls to the BLAS2 and BLAS3 on matrices of size brow-by-bcol

Distributed GE with a 2D Block Cyclic Layout

block size b in the algorithm and the block sizes brow and bcol in the layout satisfy b=brow=bcol.

need not be symmetric in rows and columns

Distributed Gaussian Elimination with a 2D Block Cyclic Layout

for k = 1 to n-1 step 1
  \text{for } i = b to end
    (1) broadcast row k, column broadcast
    (2) swap rows k and i in block column, broadcast row k
      \text{for } j = 1 to end
        (3) A(j,i-m,b) := A(j,i-m,b) / A(j,i,m)
          (4) A(j+i,m,b) := A(j+i,m,b) * A(j+i,m-1)
      \text{end for}
    (5) broadcast all swap instruction right and left
    (6) apply all row swaps to other columns
PDGESV = ScaLAPACK parallel LU routine

Since it runs no faster than its inner loop (PDGEMM), we measure:
Efficiency = Speed(PDGESV) / Speed(PDGEMM)

Observations:
- Efficiency well above 50% for large enough problems
- For fixed N, as P increases, efficiency decreases (just as for PDGEMM)
- For fixed P, as N increases, efficiency increases (just as for PDGEMM)

From bottom table, cost of solving \( Ax = b \) about half of matrix multiply for large enough matrices.

From the flop counts we would expect it to be \( \frac{2n^3}{2/3n^3} = 3 \) times faster, but communication makes it a little slower.

Out-of-core means
- Matrix lives on disk; too big for main mem
- Much harder to hide latency of disk
- QR much easier than LU because no pivoting needed for QR

Moral: use QR to solve \( Ax = b \)

Projects available
- Perhaps very hard...

LAPACK and ScaLAPACK

LAPACK ScaLAPACK

<table>
<thead>
<tr>
<th>Machines</th>
<th>Workstations, Vector, SMP</th>
<th>Distributed Memory, DSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Based on</td>
<td>BLAS, BLACS</td>
<td>ScaLAPACK</td>
</tr>
</tbody>
</table>

Functionality
- Linear Systems
- Least Squares
- Eigenproblems
- Linear Systems
- Least Squares
- Eigenproblems (less than LAPACK)

Matrix Types
- Dense, band
- Dense, band, out-of-core

Error Bounds
- Complete

Languages
- F77 or C
- F77 and C

Interface to
- C++, Fortran
- HPF

Manual?
- Yes

Where?
- www.netlib.org/
- lapack
- www.netlib.org/
- scalapack

Ax = b

- Many physical problems lead to sparse, not dense systems of equations.
- How can we reduce the time to solution?

Graph and “stencil”

2D Poisson’s equation

- Similar to the 1D case, but the matrix \( T \) is now

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

* 3D is analogous
### Jacobi's Method

- To derive Jacobi's method, write Poisson as: 
  \[ u(i,j) = \frac{(u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) + b(i,j))}{4} \]
- Let \( u(i,j,m) \) be approximation for \( u(i,j) \) after \( m \) steps 
  \[ u(i,j,m+1) = \frac{(u(i-1,j,m) + u(i+1,j,m) + u(i,j-1,m) + u(i,j+1,m) + b(i,j))}{4} \]
- 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, \( N = n^2 \)
- Therefore, serial complexity is \( O(N^2) \)

### Parallelizing Jacobi's Method

- Reduces to sparse-matrix-vector multiply by (nearly) \( T \) 
  \[ U(m+1) = \frac{T}{4} U(m) + B/4 \]
- 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 \( m/p \) elements to update
  - amount of data communicated, \( m/p \) per neighbor, is relatively small
    if \( n > p \)

---

### Gauss-Seidel

- 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 
  - \( u(i,j,m+1) = (u(i-1,j,m) + u(i+1,j,m) + u(i,j-1,m) + u(i,j+1,m) + b(i,j)) / 4 \)
    some of the values are for \( m+1 \) are already available 
    - \( u(i,j,m+1) = (u(i-1,j,latest) + u(i+1,j,latest) + u(i,j-1,latest) + u(i,j+1,latest)) \)
    where latest is either \( m \) or \( m+1 \)

### 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: 
  \[ u(i,j,m+1) = u(i,j,m) + \text{correction}(i,j,m) \]
- If "correction" is a good direction to move, then one should move even further in that direction by some factor \( w \geq 1 \) 
  \[ u(i,j,m+1) = u(i,j,m) + w \times \text{correction}(i,j,m) \]
- Called successive overrelaxation (SOR)
- Parallels like Jacobi (Still sparse-matrix-vector multiply...)
- Can prove \( w = 2/(1 + \sin(\pi/n)) \) for best convergence
  - Number of steps to converge = parallel complexity \( \propto O(n) \), instead of \( O(n^2) \) for Jacobi
  - Serial complexity \( O(n^3) \) \( \leq \) \( O(n^2 log n) \), 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 \( x \)
    - a Cholesky factorization, \( A = L L^T \) exists
- Algorithm maintains 3 vectors
  - \( x \) is the approximate solution, improved after each iteration
  - \( r \) is the residual, \( r = A x - b \)
  - \( p \) is 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 \( O(n) \) steps, like SOR
  - Serial complexity \( \leq O(n^{3/2}) \)
  - Parallel complexity \( \leq O(n^{3/2}log n) \), \( \leq O(N) \) from dot-products

---

### Parallelizing Jacobi's Method

- Reduces to sparse-matrix-vector multiply by (nearly) \( T \) 
  \[ U(m+1) = \frac{T}{4} U(m) + B/4 \]
- 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 \( m/p \) elements to update
  - amount of data communicated, \( m/p \) per neighbor, is relatively small
    if \( n > p \)

---

### Gauss-Seidel

- Updating left-to-right row-wise order, we get the Gauss-Seidel algorithm 
  for \( i = 1 \) to \( n \) 
    for \( j = 1 \) to \( n \) 
      \[ u(i,j,m+1) = \frac{(u(i-1,j,m+1) + u(i+1,j,m) + u(i,j-1,m+1) + u(i,j+1,m) + b(i,j))}{4} \]
  - Cannot be parallelized, because of dependencies, so
    instead we use a "red-black" order
  - for all black points \( u(i,j) \) 
    \[ u(i,j,m+1) = (u(i-1,j,m) + ... \]
  - for all 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 \( \Rightarrow \) 2 colorable (red and black)
    - Nodes for each color can be updated simultaneously
  - Still Sparse-matrix-vector multiply, using submatrices

### 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 \( x \)
    - a Cholesky factorization, \( A = L L^T \) exists
- Algorithm maintains 3 vectors
  - \( x \) is the approximate solution, improved after each iteration
  - \( r \) is the residual, \( r = A x - b \)
  - \( p \) is 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 \( O(n) \) steps, like SOR
  - Serial complexity \( \leq O(n^{3/2}) \)
  - Parallel complexity \( \leq O(n^{3/2}log n) \), \( \leq O(N) \) 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

Multigrid Motivation

- Recall that Jacobi, SOR, CG, or any other sparse-matrix-vector-multiply-based algorithm can only move information one grid call at a time
- Can show that decreasing error by fixed factor \( c < 1 \) takes \( \Omega(\log n) \) steps
- Convergence to fixed error \( < 1 \) takes \( \Omega(\log n) \) steps
- Therefore, converging in \( O(1) \) steps requires moving information across grid faster than to one neighboring grid cell per step

Multigrid Motivation

- 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 Overview

- Same Big Idea used before
  - Replace fine problem by coarse approximation, recursively
- Multilevel Graph Partitioning (METIS):
  - Replace graph to be partitioned by a coarser graph, obtained via Maximal Independent Set
  - Given partitioning of coarse grid, refine using Kernighan-Lin
- Barnes-Hut (and Fast Multipole Method):
  - Approximate leaf node of QuadTree containing many particles by Center-of-Mass and Total-Mass (or multipole expansion)
  - Approximate internal nodes of QuadTree by combining expansions of children
  - Force in any particular node approximated by combining expansions of a small set of other nodes
- All examples depend on coarse approximation being accurate enough (at least if we are far enough away)

Multigrid Sketch (1D and 2D)

- Consider a \( 2^m+1 \) grid in 1D (\( 2^m+1 \) by \( 2^m+1 \) grid in 2D) for simplicity
- Let \( P^{(i)} \) be the problem of solving the discrete Poisson equation on a \( 2^i+1 \) grid in 1D (\( 2^i+1 \) by \( 2^i+1 \) grid in 2D)
- Write linear system as \( T^{(i)} x^{(i)} = b^{(i)} \)
- \( p_i^{(i)} , p_{i-1}^{(i)} , \ldots , p_1^{(i)} \) is sequence of problems from finest to coarsest
Multigrid Operators

- For problem \( P(i) \):
  - \( b(i) \) is the RHS and
  - \( x(i) \) is the current estimated solution
- All the following operators just average values on neighboring grid points
  - Neighboring grid points on coarse problems are far away in fine problems, so information moves quickly on coarse problems
- The restriction operator \( R(i) \) maps \( P(i) \) to \( P(i-1) \)
  - Restricts problem on fine grid \( P(i) \) to coarse grid \( P(i-1) \) by sampling or averaging
  - \( b(i-1) = R(i)(b(i)) \)
- The interpolation operator \( In(i-1) \) maps an approximate 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 computes an improved solution \( x(i) \) on same grid
  - Uses "weighted" Jacobi or SOR
  - \( x_{new} = S(i)(b(i), x(i)) \)
- Details of these operators after describing overall algorithm

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
  - \( x(i) = S(i)(b(i), x(i)) \) 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
  - return \( x(i) \)

Why is this 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 \( O(\text{the number of unknowns}) \)
  - Cost of Level \( i \) is \( (2^i-1)2^{2i} = O(4^i) \)
  - If finest grid level is \( m \), total time is:
    - \( \sum_{i=1}^{m} O(4^i) = O(4^m) = O(\# \text{ unknowns}) \)

- On a 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 \( O(m) = O(\log \# \text{ unknowns}) \)

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 \( i \) from 2 to \( m \)
  - \( x(1) = MGV( b(1), In(1)(x(1-1)) ) \)

- 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:
  - \( \sum_{i=1}^{m} O(4^i) = O(4^m) = O(\# \text{ unknowns}) \)
- PRAM time:
  - \( O(\log \# \text{ unknowns}) \)
Complexity of Solving Poisson’s Equation

- Theorem: error after one FMG call <= c* error before, where c < 1/2, independent of # unknowns

- 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

- Total complexity just proportional to cost of one FMG call