Lecture 5

Higher dimensional iterative methods
Performance modeling
Announcements

- A2 due today - submit hard copy and electronic copy
- A3 is posted
- Project proposals due Tuesday
- Connected component labeling
Recapping from last time

- Solve the ODE
  \[-u''(x) = f(x), \quad x \in [0, 1]\]

- Define \( u_i = u(i \times h) \) at points
  \[ x = i \times h, \quad h = 1/(N-1) \]

- Approximate the derivatives
  \[ u'' \approx \frac{(u(x+h) - 2u(x) + u(x-h))/h^2}{h} \]

- Obtain the system of equations
  \[ \frac{(u_{i-1} - 2u_i + u_{i+1})}{h^2} = f_i, \quad i \in 1..n-2 \]
Iterative solution

• Rewrite the system of equations
  \[-u_{i-1} + 2u_i - u_{i+1})/h^2 = f_i , i \in 1..n-1\]

• It can be shown that the following Gauss-Seidel algorithm will arrive at the solution …

• …. assuming an initial guess for the \( u_i \)

Repeat until the result is satisfactory
  
  for i = 1 : N-1
    \[ u_i = (u_{i+1} + u_{i-1} +h^2 f_i)/2 \]
  end for

end Repeat
Parallel implementation

• The first step is to partition the data
• We partition the data into intervals, assigning each to a unique processor
• Many to one mappings may also be useful
Data dependences

• Each interval depends on two endpoint values found on neighboring processes
• We add “overlap” or “ghost” cells to hold a copy of the off-processor value

![Diagram showing data dependences between processes P0, P1, P2, and P3 with arrows indicating dependencies.]
Dependences

• Our attempt to parallelize the algorithm fails since there are loop carried dependences

• The value of $u[i]$ computed in iteration $i$ depends on $u[i]$ computed in iteration $i-1$

```latex
\begin{align*}
\text{for } i = 1 : N-1 \\
\quad u[i] &= (u[i-1]+u[i+1] +h*h*f[i])/2 \\
\text{end for}
\end{align*}
```
Parallel implementation

- Renaming the LHS of the assignment eliminates the dependences
- Rename LHS as `unew`

```plaintext
for i = 1 : N-1
    unew[i] = (u[i-1] + u[i+1] + h*h*f[i])/2
end for

Swap u and unew
```
Tradeoffs

• We can now parallelize the algorithm
• However, we have decreased the convergence rate by about a factor of two
• Doubles the amount of computational work
• This kind of tradeoff is common, but we can do better
Convergence check

• Each process computes the error for its assigned part of the problem
• We need a global error so that we compute a result that is consistent with the single processor implementation
• This requires collective communication: a reduction
• In MPI we use Allreduce, which leaves a copy the result in the memory of all processes
Some Definitions for Solving the Poisson Equation

• We call the numerical operator that sweeps over the solution array a **stencil operator**
• In 1D we have functions of one variable
• In n dimensions we have n variables
• In 2D:
  \[
  \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \text{ within a square box, } x,y \in [0,1]
  \]
  \[
  u(x,y) = \sin(x)\sin(y) \text{ on } \partial \Omega, \text{ perimeter of the box}
  \]
  Define \( u_{i,j} = u(x_i, y_j) \) at points \( x_i = i \times h, \quad y_j = j \times h, \quad h = 1/(N-1) \)
• **Approximate the derivatives**
  \[
  u_{xx} \approx (u(x_{i+1},y_j) + u(x_{i-1},y_j) + u(x_i,y_{j+1}) + u(x_i,y_{j-1}) - 4u(x_i,y_j))/h^2
  \]
Jacobi’s Method in 2D

- The update formula

\[
\text{for } (i,j) \text{ in } 0: \text{N}-1 \times 0: \text{N}-1 \\
u'[i][j] = (u[i-1] + u[i+1][j] + u[i][j-1] + u[i][j+1] - h^2 f[i,j]) / 4 ; \\
u = u' 
\]
Partitioning

- Splitting up the data over processors
- We express the different partitions in terms of a \textit{processor geometry}
- For P processors the geometries are of the form $p_0 \times p_1$, where $P = p_0 \times p_1$
- For P=4, 3 possible geometries
Ghost cells

• Each partition needs values found on neighboring processors
• Inefficient to communicate individual values
• Non-contiguous data
Periodically refresh the ghost cells
Some results in 3D on an IBM SP2 with 16 processors

<table>
<thead>
<tr>
<th>Geometry</th>
<th>MF/s</th>
<th>Time</th>
<th>Comm</th>
<th>MF/s</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>16x 1x1</td>
<td>510</td>
<td>15.80</td>
<td>2.41</td>
<td>601</td>
<td>13.39</td>
</tr>
<tr>
<td>1x16x 1</td>
<td>655</td>
<td>12.29</td>
<td>2.43</td>
<td>816</td>
<td>9.87</td>
</tr>
<tr>
<td>1x 1x 16</td>
<td>547</td>
<td>14.72</td>
<td>8.33</td>
<td>1260</td>
<td>6.39</td>
</tr>
<tr>
<td>4x 4x 1</td>
<td>611</td>
<td>13.18</td>
<td>0.90</td>
<td>656</td>
<td>12.27</td>
</tr>
<tr>
<td>4x 1x 4</td>
<td>674</td>
<td>11.94</td>
<td>2.71</td>
<td>872</td>
<td>9.23</td>
</tr>
<tr>
<td>1x 4x 4</td>
<td>621</td>
<td>12.96</td>
<td>2.77</td>
<td>790</td>
<td>10.19</td>
</tr>
<tr>
<td>8x 2x 1</td>
<td>554</td>
<td>14.54</td>
<td>1.43</td>
<td>614</td>
<td>13.11</td>
</tr>
<tr>
<td>8x 1x 2</td>
<td>619</td>
<td>13.00</td>
<td>1.21</td>
<td>683</td>
<td>11.79</td>
</tr>
<tr>
<td>4x 2x 2</td>
<td>629</td>
<td>12.81</td>
<td>1.10</td>
<td>688</td>
<td>11.71</td>
</tr>
<tr>
<td>2x 4x 2</td>
<td>629</td>
<td>12.81</td>
<td>1.07</td>
<td>686</td>
<td>11.74</td>
</tr>
<tr>
<td>2x 2x 4</td>
<td>671</td>
<td>12.00</td>
<td>2.48</td>
<td>846</td>
<td>9.52</td>
</tr>
<tr>
<td>2x 8x 1</td>
<td>669</td>
<td>12.03</td>
<td>1.39</td>
<td>757</td>
<td>10.64</td>
</tr>
<tr>
<td>2x 1x 8</td>
<td>617</td>
<td>13.06</td>
<td>4.40</td>
<td>931</td>
<td>8.65</td>
</tr>
<tr>
<td>1x 2x 8</td>
<td>560</td>
<td>14.37</td>
<td>4.43</td>
<td>810</td>
<td>9.95</td>
</tr>
<tr>
<td>1x 8x 2</td>
<td>693</td>
<td>11.63</td>
<td>1.84</td>
<td>823</td>
<td>9.79</td>
</tr>
</tbody>
</table>

57% communication
6.8% communication

15.8% communication
What’s wrong with MFLOP rates?

• Different algorithms employ different numbers of floating point operations, e.g. Strassen’s matrix multiply algorithm
• Different library implementations can execute different numbers of FLOPs, e.g. log()
• Precision affects timing
• Floating point operations take different times
  – Divide is much slower than multiply or add
  – Some machines have a fused multiply-add
• MFLOP rates ignore memory access time
Modeling Parallel Performance

- The model has two parts
  - Local computation
  - Communication

- We may ignore the convergence test (check infrequently)
- Communication overheads are due to ghost cell updates
- Let’s start with the 1D ODE solver and then move to higher dimensional spaces
Model assumptions and definitions: 1D case

- $T(1,n) =$ running time of the **best serial algorithm** on a problem of size $n$
- $T(P,n) =$ running time on $P$ processors
- $T_\gamma(P,n) =$ **grind time**
  - Time to perform a single mesh update
  - Helps us normalize with respect to problem size
  - $T_\gamma(P,n0) = T(P,n)/(n \cdot Niter)$
Local Computation time

• $T(1,n) = n \, T_\gamma \, N_{\text{Iter}}$, where $T_\gamma$ is the cost of performing an update
  
  $$u_i = \frac{(u_{i+1} + u_{i-1} + h^2 \, f_i)}{2}$$

• On Valkyrie
  
  $$T_\gamma \approx 20\beta$$

• Datum are 8-byte double precision numbers, message passing time = $\alpha + 8\beta N$
More on the Performance model

• Make the naïve assumption that $T_\gamma(1,n)$ is independent of $n$
• $T_{comm} =$ local communication for ghost cells
• $T(P,N) = T(1,N/p) + T_{\text{local}}$
• $= 16N \beta + 2(\alpha+8\beta)$
  $\approx 16N \beta$
The curse of dimensionality

• In higher dimensional spaces
  – Many more possible processor geometries
  – Communication involves higher dimensional arrays
  – Fraction of communication increases for a fixed number of unknowns
• In 1D
  – There is only one possible processor geometry
  – Each process communicates at most 2 points
• In 2D
  – There are 1D and 2D geometries
  – Each process communicate a set of 1D arrays
• In D dimensions
  – D different sets of geometries
  – Each process communicates several \((D-1)\) dimensional arrays
Model assumptions and definitions: 2D case

- $T(1, (m,n)) =$ running time of the best serial algorithm on a problem of size $m \times n$
- $T(P, (m,n)) =$ running time on $P$ processors
- $T_\gamma(P, (m,n)) =$ grind time on $P$ processors
  - $T_\gamma(P, (m,n)) = T(P, (m,n))/(m \cdot n \cdot Niter)$
  - Ideally $T_\gamma$ is independent of $m$, $n$, and $P$
- Following analysis applies to Laplace’s equation, a special case of Poisson’s Equation with $f=0$
Performance

• 2 components
  – Local computation
  – Ghost cell communication
    – (Convergence check, including global communication)
Local Computation time

- $T(1,(m,n)) = m \times n \times T_\gamma$ where
  $T_\gamma = \text{grind time on one processor}$

- For the Blue Horizon IBM SP3 system at the San Diego Supercomputer Center (with Power3 CPUs)
  $T_\gamma \approx 16 \beta$

\[
u_{i,j} = (u_{i+1,j}+u_{i-1,j}u_{i,j+1}+u_{j,i-1})/4\]
Completing the Performance model

- Message passing time \( T(N) = \alpha + 8\beta N \)
  (8-byte double precision numbers)
- Processor geometry is \( p \times q \)
  - Strips or box–like partitions
- \( T(P,(N,N)) = T(1,(m,n)) + T_{local}^{comm} \)
  \[ m = N/p, \ n = N/q \]
Communication performance for 1D

- $P$ divides $N$ evenly
- $N/P > 2$
- For horizontal strips, data are contiguous
  $T_{comm} = 2(\alpha + 8\beta N)$
2D Processor geometry

- Assume $\sqrt{P}$ divides $N$ evenly and $N/\sqrt{P} > 2$
- Ignore the cost of packing message buffers
- $T_{comm} = 4(\alpha + 8\beta N/\sqrt{P})$
Summing up the performance models

- 1-D decomposition

\[ (16N^2 \beta/P) + 2(\alpha+8\beta N) \]

- 2-D decomposition

\[ (16N^2 \beta/P) + 4(\alpha+8\beta N/\sqrt{P}) \]
Comparative performance

- Strip decomposition will outperform box decomposition—resulting in lower communication times—when \(2(\alpha+8\beta N) < 4(\alpha+8\beta N/\sqrt{P})\)

- Assuming \(P \geq 2\): \(N < (\sqrt{P}/(\sqrt{P} – 2))(\alpha/8\beta)\)

- On SDSC’s IBM SP3 system “Blue Horizon”
  \(\alpha = 24\text{ us}\)
  \(\beta = 1/(390\text{ MB/sec})\)

- \(N < 1170 (\sqrt{P}/(\sqrt{P} – 2))\)

- For \(P = 16\), strips are preferable when \(N < 2340\),
Parallel speedup and efficiency

• 1-D decomposition

\[ S_P = \frac{T_1}{T_P} = \frac{16N^2\beta}{(16N^2\beta/P + 2(\alpha+8\beta N))} \]
\[ E_P = \frac{S_P}{P} = \frac{16N^2\beta}{(16N^2\beta + 2P(\alpha+8\beta N))} \]
\[ = \frac{1}{1 + (\alpha+8\beta N)P/(8N^2\beta)} \]

• 2-D decomposition

\[ S_P = \frac{T_1}{T_P} = \frac{16N^2\beta/(16N^2\beta/P+4(\alpha+8\beta N/\sqrt{P}))}{(16N^2\beta + 4(\alpha P + 8\beta N\sqrt{P}))} \]
\[ = \frac{1}{1 + (\alpha P + 8\beta N\sqrt{P})/(4N^2\beta)} \]
Putting these formulas to work

• 1-D decomposition
• Let’s plot $E_P$ as a function of $N$, varying $P$ as a parameter
  $$E_P = \frac{1}{1 + (\alpha + 8\beta N)P / (8N^2\beta)}$$
• Let’s also plot the fraction of time spent communicating
Parallel speedup and efficiency

\[ N = 1024 \]

\[ N = 128 \]

Graph showing parallel speedup and efficiency with different values of \( N \).
Communication fraction

\[ N = 128 \]

\[ N = 1024 \]
Surface to volume ratio affects performance

• The *surface to volume ratio* of a geometry is the maximum number of points on the surface (perimeter) over all partitions divided by the volume

• As we increase N while leaving P fixed, we decrease the surface to volume ratio, which gives us a measure of the relative cost of communication

• As volume increases, S/V drops
Surface to volume ratio

1 unit of work
4 units of communication

16 units of work
16 units of communication
The curse of dimensionality

• As we move to higher dimensional spaces, communication becomes relatively more costly
  ▶ In 2D: \( \frac{4N}{N^2} = \frac{4}{N} \)
  ▶ In 3D: \( \frac{6N^2}{N^3} = \frac{6}{N} \)
Refinements to the performance model

- Transmitting non-contiguous data is more expensive than transmitting contiguous data, strides larger in 3D than 2D
- The grind time is sensitive to the aspect ratio of the local grid
Assignment #3

• Connected component labeling
• AKA forest fire simulation
• Given a binary image, identify the connected components consisting of ‘1’ bits that share a neighbor link
Scalability

• Earlier we talked about the isoefficiency function…
• This function tells us how quickly serial work $W$ must grow as we increase $P$… so that the efficiency will remain constant
• We now consider scalability in greater detail
Overhead

• Ideally, $T_P \equiv W/P$, or $W \equiv P T_P$
• In practice, $T_P > W/P$. Why is this?
• We define $T_o \equiv PT_P - W$ as the total overhead or the overhead function
• We call $PT_P$ the cost, or the processor-time product
• Note that $E_P = W / (P T_P) = W / \text{cost}$
Cost optimality

• If system is cost-optimal if cost of the parallel computation has the same asymptotic growth as fastest serial algorithm

• The cost should grow at the same rate as $W$, i.e. $PT_P = \Theta(W)$

• Thus, efficiency = ?
An example

• Consider a serial algorithm running in time $n \log n$
• Let $T_p = (\log n)^2$ on $P=n$ processors:
  \[ \text{cost} = n(\log n)^2 \]
• The system solving this problem is \textit{not} cost optimal
• If it were, then the cost would be $n \log n$
  but is not far off
Why does efficiency decrease with $P$?

- Recall that efficiency $E_P \equiv W/(PT_P)$
- Plugging this into the overhead equation $T_o \equiv PT_P - W$ we have $E_P \equiv 1/(1 + T_o/W)$
- If $W$ remains fixed
  - Overhead $(PT_P)$ often increases with $P$
  - Efficiency must therefore decrease
A non-cost optimal example

• Summing N numbers on N processors
  – \( W = N-1 \)
  – \( T_N = \log(N) \)

• Cost = \( N \log N \)

• Efficiency = \( E_P \equiv 1/(1 + T_o/W) \)
  \[ = 1/(1 + (\text{Cost} - W)/W) \]
  \[ = 1/(1 + (N \log N-N)/N) \]
  \[ = 1/(1 + \log N) \]
  \[ = 1 / (\log N) \ll 1 \]
A cost optimal variant

• Summing $N$ numbers on $P < N$ processors
  – $W = N - 1$
  – $T_P = (N/P) + \lg(P)$
• Cost $= N + P \lg(P)$
• If we maintain $N = \Omega(P \log P)$, then system is cost optimal
Scalability

• We say that a system is **scalable** if we can maintain a (nearly) constant running time as we increase the work with the number of processors.

• Equivalently, a system if scalable if we can maintain a constant level of parallel efficiency.

• When we think about scalability we ask: “how quickly must the computational work grow with P?”
Scalability and cost-optimality

- Recall that a cost optimal system has an efficiency of $\Theta(1)$
- A scalable system can be made cost-optimal if we grow the problem size with $N$
- For example, with summation, we can maintain cost-optimality if we grow $N$ as $\Theta(P \lg P)$
- The *isoefficiency function* tells us the required growth rate, and how scalable the system is
- It vary among different systems
Isoefficiency function

• How quickly must the workload grow, as a function of $P$, in order to maintain a constant level of efficiency

• Consider the ODE solver
  – $N =$ Problem size
  – $P =$ Number of processors
  – Computational work $= W = 3N \ (= T_1 )$
How do we come up with the isoefficiency function?

- \( E_P = \frac{W}{PT_P} = \frac{W}{W + T_0(W,P)} \)
  
  \[ = \frac{1}{1 + \frac{T_0(W,P)}{W}} \]

- Solving for \( W \)
  
  \[ W = \frac{E_P}{1-E_P} T_0(W,P) = K T_0(W,P) \]

- This is the isoefficiency function, the required growth of \( W \) as a function of \( P \)

- In order for a system to remain cost-optimal as it is scaled up, we require that \( W = \Omega(f(P)) \), where \( f(P) \) is the isoefficiency function
Isoefficiency function for the ODE solver

- Let a floating point operation take unit time
- Normalized message start time = $\alpha$
- Parallel running time $T_P$
  - Perfect parallelization of $W + \text{overheads}$
    - $W/P + 2\alpha$
- Parallel efficiency
  - $E_P = T_1 / (PT_P) = 3N/(2\alpha P + 3N)$
- Rewriting to obtain expression for $N$ in terms of $P$
  - $N = (2/3) \alpha P (E_P/(1 - E_P)) = O(P)$
- So long as we can grow $N$ with $P$, then the system is scalable
Re-examining scaled speedup

• We define the speedup as

\[ \frac{W}{T_p(W,P)} \]

• The scaled speedup (linear scaling) is

\[ \frac{PW}{T_p(PW,P)} \]

• If we are scaling a problem according to the isoefficiency function \( \Theta(P \ lg \ P) \), what is the scaled speedup?

• \( \frac{PW \ lg \ P}{T_p(PW \ lg \ P, P)} \)
Scaling (Problem 5.7, p.230)

- Plot the efficiency for the problem of adding n numbers on p processors
  - $t_{\text{add}} = 10$, time to communicate = 1
  - $p = 1, 4, 16, 64, 256$

- Fixed workload
  - Let $n=256$, $W = 255$
  - Speedup = $W / T_p(W,P)$

- Scaled workload, base case $n=256$, $p=1$
  - $W = \Theta(p)$
  - Scaled speedup = $PW / T_p(PW,P)$

- “Isoefficient scaled workload”
  - $W = \Theta(p \log p)$
  - Isoefficient scaled speedup = $PW \log p / T_p(PW \log p,P)$
Results

• Running time to sum n numbers on p processors:
  \[
  \frac{n}{p} - 1 + 11 \log p
  \]

• **Fixed workload speedup**
  – Let \( n=256 \) \( W = 255 \)
  – Speedup = \( \frac{W}{T_P(W,P)} \)

**Scaled workload speedup**

• base case: \( n=256, p=1 \)
• \( W = \Theta(P) \)
• **Scaled speedup** = \( \frac{PW}{T_P(PW,P)} \)
Time constrained scaling

- For the summation problem of adding up \( N \) numbers of \( P < N \) processors….
- Determine the largest problem that can be solved in time \( T = 512 \) time units, where \( \alpha = 10 \) time units, and addition costs one unit of time, for \( P = 1, 16, 256, \) and \( 4096 \).