Lecture 5

Multithreaded iterative stencil methods
Caches: coherence and consistency
Announcements
Iterative mesh methods
Mesh based methods

- Many physical problems are simulated on a uniform \textit{mesh} in 1, 2 or 3 dimensions
- \textit{Field variables} defined on a discrete set of points
- A \textit{mapping} from ordered pairs to \textit{physical observables} like temperature and pressure
- One application: differential equations
Differential equations

- A **differential equation** is a set of equations involving derivatives of a function (or functions), and specifies a solution to be determined under certain constraints
- Constraints often specify **boundary conditions** or **initial values** that the solution must satisfy
- When the functions have multiple variables we have a **Partial Differential Equation** (PDE)
  \[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \]  within a square box, \( x, y \in [0,1] \)
  \[ u(x,y) = \sin(x) \ast \sin(y) \]  on \( \partial \Omega \), perimeter of the box
- When the functions have a single variable we have an **Ordinary Differential Equation** (ODE)
  \[-u''(x) = f(x), \; x \in [0,1], \; u(0) = a, \; u(1) = b\]
Solving an ODE with a discrete approximation

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

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

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

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

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

• …. assuming an initial guess for the $u_i$

Repeat until the result is satisfactory
for $i = 1 : N-1$

\begin{align*}
  u_i &= \frac{(u_{i+1} + u_{i-1} + h^2 f_i)}{2}
\end{align*}

end for

end Repeat
Convergence

• Convergence is slow
• We reach the desired precision in $O(N^2)$ iterations
• The “better” the initial guess, the sooner we converge
Estimating the error

• How do we know when the answer is “good enough?”
  • The computed solution has reached a reasonable approximation to the exact solution
  • We validate the computed solution in the field, i.e. wet lab experimentation
• But we often don’t know the exact solution, and must estimate the error
• We can estimate the error by taking the difference between the present and the previous iteration, and computing the maximum change over all points
• There are other measures, too
Parallel implementation

- We partition the data into intervals, assigning each to a unique thread
- Each thread sweeps over a reduced problem, using as boundary conditions values updated by other threads

\[ P0 \quad P1 \quad P2 \quad P3 \]
Dependences

• Our attempt to parallelize the algorithm fails: there are loop carried dependences
• The value of $u[i]$ computed in iteration $i$ depends on $u[i]$ computed in iteration $i-1$

for $i = 1 : N-1$

\[ u[i] = (u[i-1]+u[i+1] +h*h*f[i])/2 \]
end for
Jacobi’s Method

- Renaming the LHS of the assignment eliminates the dependences
- Two arrays $u$ and $u_{\text{new}}$
- This is *Jacobi’s method*

```plaintext
for i = 1 : N-1
    \[ u_{\text{new}}[i] = \frac{(u[i-1] + u[i+1] + h^2 f[i])}{2} \]
    error = Max(error, absval($u_{\text{new}}[i] - u[i]$))
end for
Swap u and $u_{\text{new}}$
```
Odd/Even or Red/Black ordering

• Another way to order the computation is to number the points as even and odd
• We alternate between sweeping over red and black points
• This algorithm parallelizes since there are no loop carried dependences
• All the red points are decoupled
• Have we sacrificed some aspect of performance?

\[ u_{i-1} \quad u_i \quad u_{i+1} \]
Performance

• In early computer designs, arithmetic was much more expensive than memory accesses
• So the running time could be expressed in terms of “flops” = “floating point operations”
• But today, memory accesses are a more realistic measure
  – Gauss-Seidel?
  – Jacobi’s
  – Odd/Even?

\[ u_{i-1} \quad u_i \quad u_{i+1} \]
Tradeoffs

• We can now parallelize the algorithm, since we have eliminated the loop carried dependencies
• Jacobi’s method reduced the convergence rate by about a factor of two
• Odd/Even ordering doubles the number of cache misses
• Doubles the amount of work needed to solve the problem
• This kind of tradeoff is common
• Which algorithm should be used in the “fastest serial” implementation?
The odd/even algorithm

for i = 1 : N-1 by 2
    \[ u[i] = \frac{(u[i-1]+u[i+1]+h^2f[i])}{2} \]
end for

for i = 2 : N-1 by 2
    \[ u[i] = \frac{(u[i-1]+u[i+1]+h^2f[i])}{2} \]
end for
Convergence check

• Each thread 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
• We form a global sum of the local contributions
Stencils and molecules

• We call the numerical operator that sweeps over the solution mesh a **stencil operator**

• In n dimensions, we have functions of n variables

• In 2D:

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \Delta u = f(x,y) \quad \text{within a square box, } x,y \in [0,1]
\]

\[
u(x,y) = \sin(x) \times \sin(y) \quad \text{on } \partial \Omega, \quad \text{perimeter of the box}
\]

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

• We call the numerical operator that sweeps over the solution mesh a **stencil operator**

• We approximate the derivatives

\[ u_{xx} \approx (u(x_{i+1},y_j) + u(x_{i-1},y_j) + u(x_i,y_{j+1}) + u(x_i,y_{j-1}) - 4u(x_i,y_j)/h^2 \]

• The update formula

for \((i,j)\) in \(0:N-1 \times 0:N-1\)

\[ u'[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1] - h^2f[i,j])/4 \]
Partitioning

- Splits up the data over threads
- Different partitionings according to the processor geometry

For $P$ processors geometries are of the form $p_0 \times p_1$, where $P = p_0 \cdot p_1$

- For $P=4$, 3 possible geometries

\[
\begin{array}{c|c|c|c|c|}
\hline
0 & 1 & 2 & 3 \\
\hline
\end{array}
\quad
\begin{array}{c|c}
0 & \\
\hline
1 & \\
\hline
2 & \\
\hline
3 & \\
\hline
\end{array}
\quad
\begin{array}{c|c}
0 & 1 \\
\hline
2 & 3 \\
\hline
\end{array}
\]
Data access

- Off processor values surround each local subproblem
- Non-contiguous data
- Inefficient to access values on certain faces/edges
Multithreaded Solve()

Local mymin = 1 + ($TID * n/$nprocs), mymax = mymin + n/$nprocs -1;

Global error, u[:,:], u\text{new}[:,:];
done = FALSE;

\textbf{while} (!done) \textbf{do}
\hspace{1em} myerr = error = 0;
\hspace{2em} \textbf{Barrier()}
\hspace{2em} \textbf{for} i = mymin \textbf{to} mymax \textbf{do}
\hspace{3em} \textbf{for} j = 1 \textbf{to} n \textbf{do}
\hspace{4em} u\text{new}[i,j] = …
\hspace{4em} myerr = \text{Max}(myerr, \text{absval}(u\text{new}[i,j]-u[i,j]))
\hspace{3em} \textbf{end for}
\hspace{2em} \textbf{end for}
\hspace{1em} \textbf{Critical Section}
\hspace{2em} \{\text{error} = \text{Max}(error,myerr); \}
\hspace{2em} \textbf{Barrier()}
\hspace{2em} \textbf{if} (error / (n*n) < \text{Tolerance}) done = TRUE;
\hspace{2em} u[mymin:mymax,:) = u\text{new}[mymin:mymax,:]
\hspace{2em} \textbf{Barrier()}
\hspace{1em} \textbf{end while}

- Don’t read locations updated by other processes in the previous iteration until they have been produced (true dependence)
- Don’t overwrite values used by other processes in the current iteration until they have been consumed (anti-dependence)
Caches
Coherency,
Consistency,
False Sharing
Cache Coherence

• A central design issue in shared memory architectures
• Processors may read and write the same cached memory location
• If one processor writes to the location, all others must eventually see the write

```
X:=1    Memory
```
Cache Coherence

- P1 & P2 load X from main memory into cache
- P1 stores 2 into X
- The memory system doesn’t have a coherent value for X
Cache Coherence Protocols

• Ensure that all processors *eventually* see the same value

• Two policies
  – Update-on-write (implies a write-through cache)
  – Invalidate-on-write
SMP architectures

- Employ a *snooping protocol* to ensure coherence
- Processors listen to bus activity

---

*Parallel Computer Architecture*, Culler, Singh, Gupta
Memory consistency and correctness

- Cache coherence tells us that memory will eventually be consistent
- The memory consistency policy tells us when this will happen
- Even if memory is consistent, changes don’t propagate instantaneously
- These give rise to correctness issues involving program behavior
Memory consistency

• A memory system is consistent if the following 3 conditions hold
  – Program order
  – Definition of a coherent view of memory
  – Serialization of writes
Program order

- If a processor writes and then reads the same location $X$, and there are no other intervening writes by other processors to $X$, then the read will always return the value previously written.
Definition of a coherent view of memory

- If a processor P reads from location X that was previously written by a processor Q, then the read will return the value previously written, if a sufficient amount of time has elapsed between the read and the write.
Serialization of writes

• If two processors write to the same location X, then other processors reading X will observe the same the sequence of values in the order written

• If 10 and then 20 is written into X, then no processor can read 20 and then 10
Memory consistency model

• The memory consistency model determines when a written value will be seen by a reader

• **Sequential Consistency** maintains a linear execution on a parallel architecture that is consistent with the sequential execution of some interleaved arrangement of the separate concurrent instruction streams

• Expensive to implement

• **Relaxed consistency**
  – Enforce consistency only at well defined times
  – Useful in handling false sharing
False sharing

• Consider two processors that write to different locations mapping to different parts of the same cache line
False sharing

• P0 writes a location
• Assuming we have a write-through cache, memory is updated
False sharing

- P1 reads the location written by P0
- P1 then writes a different location in the same block of memory
False sharing

- P1’s write updates main memory
- Snooping protocol invalidates the corresponding block in P0’s cache
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache
Eliminating false sharing

• Cleanly separate locations updated by different processors
  – Manually assign scalars to a preallocated region of memory using pointers
  – With a block partitioned array, we want partition boundaries to coincide with a cache line boundary

• Compilers can perform some of these optimizations
How do cache misses arise?

• The 3 C’s
• Cold Start
• Capacity
• Conflict
False sharing in higher dimension arrays

• Compare with distributed memory solution
Reducing conflict misses

- Pad the array with unused cells to change the memory access patterns
- Rivera & Tseng [Sigplan, 1998]
- Any other ways?