Lecture 3

Stencil methods
Multithreaded programming with OpenMP
Announcements

• Office hours
  ✷ Scott Baden: Mondays@4.30pm, Thursdays@4pm
  ✷ Edmund Wong: Mondays/Wednesdays 10am to 11am
  ✷ See http://cseweb.ucsd.edu/classes/fa12/cse260-b/kontakt.html

• Partners

• Blas Performance
Stencil Methods
Stencil methods

- Many physical problems are simulated on a uniform mesh in 1, 2 or 3 dimensions
- Field variables defined on a discrete set of points
- A mapping from ordered pairs to 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)
  \[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \]  
  within a square box, \( x,y \in [0,1] \)
  \( u(x,y) = \sin(x) \cdot \sin(y) \) 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

• 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 \textit{Gauss-Seidel} algorithm will arrive at the solution …

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

Repeat until the result is satisfactory

for \(i = 1 : N-1\)

\[ u_i = (u_{i+1} + u_{i-1} + h^2 f_i)/2 \]

end for

end Repeat
Convergence

- Convergence is slow
- We reach the desired precision in $O(N^2)$ iterations
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
Using the residual to estimate the error

• Recall the equations
  \[ (-u_{i-1} + 2u_i - u_{i+1})/h^2 = f_i , \ i \in 1..n-1 \ [Au = f] \]
  \[ Au = f \]

• Define the residual \( r_i \):
  \[ r_i = (-u_{i-1} + 2u_i - u_{i+1})/h^2 - f_i , \ i \in 1..n-1 \]

• Thus, our computed solution is correct when \( r_i = 0 \)

• We can obtain a good estimate of the error by finding the maximum \( r_i \ \forall i \)

• Another possibility is to take the root mean square (L2 norm)
  \[ \sqrt{\frac{\sum r_i^2}{n}} \]
Stencil operations in higher dimensions

- 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:
  \[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \Delta u = f(x,y) \]
  within a square box, $x, y \in [0,1]$
  \[ u(x,y) = \sin(x) \cdot \sin(y) \] on $\partial \Omega$, perimeter of the box

Define $u_{ij} = u(x_i, y_j)$ at points $x_i = i \times h$, $y_j = j \times h$, $h = 1/(N-1)$

- 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

Until converged:
   for \((i,j)\) in \(0:N-1 \times 0:N-1\)
   \[ u'[i,j] = \frac{(u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1] - h^2f[i,j])}{4} \]

\[ u = u' \]
Roundoff

• You may notice slight variations in the residual in a parallel computation
• Machine arithmetic is finite
• Consider 4-digit decimal arithmetic
• Compute $10^4 - (10^4 - 1) = 1$
  - $10^4 - 1 = 1.000E4 - 1.000E0 = 9.999E3$
    - Normalize 1.000E0 to 0.0001E4
    - But with only 4 digits we truncate 0.0001E4 to 0.000E4
    - Result: $10^4$
  - $10^4 - 10^4 = 0$, not 1; what if we had to divide?
• Machine arithmetic is neither associative nor commutative
Multicore programming
Shared memory programming with threads

- Program executes a collection of independent instruction streams, called *threads*
  - We usually specify the number of threads, $NT$, at program launch time
- Each thread
  - is usually assigned to a distinct physical processor
  - has a *thread index*, a unique integer in $[0: NT-1]$
  - executes the same program, but at its own rate
  - communicates with other threads via shared memory
- A thread is similar to a procedure call with notable differences
  - A new storage class: shared data
  - A procedure call is “synchronous:” a return indicates completion
  - A spawned thread executes asynchronously until it completes
  - Shares global storage with spawner
  - Synchronization may be required for *thread safety*
Why threads?

- Processes are “heavy weight” objects scheduled by the OS
  - Protected address space, open files, and other state
- A thread AKA a lightweight process (LWP)
  - Threads share the address space and open files of the parent, but have their own stack
  - Reduced management overheads, e.g. thread creation
  - Kernel scheduler multiplexes threads
Multithreaded programming in practice

- One approach is to use program annotations via openMP
- A lower level interface is the POSIX Threads “standard” (pthreads): IEEE POSIX 1003.1c-1995
  - Beware of non-standard features
- Another is Java threads, but not used in HPC
OpenMP programming

• Simpler interface than explicit threads
• Parallelization handled via annotations
• See http://www.openmp.org
• Parallel loop:

```c
#pragma omp parallel private(i) shared(n)
{
    #pragma omp for
    for(i=0; i < n; i++)
        work(i);
}
```

```c
i0 = $TID*n/$nthreads;
i1 = i0 + n/$nthreads;
for (i=i0; i < i1; i++)
    work(i);
```
OpenMP Fork-Join Model

- A program begins life as a single thread
- Parallel regions spawn work groups of multiple threads
- Threads may or may not execute on different processors, and might be interleaved
## Sections

```c
#pragma omp parallel // Begin a parallel construct
{ // form a team
    // Each team member executes the same code
#pragma omp sections // Begin work sharing
    {
        #pragma omp section // A unit of work
        {functionA(..);}

        #pragma omp section // Another unit
        {functionB(..);}
    } // Wait until both units complete

} // End of Parallel Construct; disband team

// continue serial execution
```

©2012 Scott B. Baden /CSE 260/ Fall 2012
Fork join model with loops

```c
printf("Start\n");
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

printf("Finish\n");
```

Seung-Jai Min
Workload decomposition

• Can by static or dynamic
• Dynamic assignment for irregular problems (later on)
• Translator automatically generates appropriate local loop bounds

```c
#pragma omp parallel private(i) shared(n)
{
    #pragma omp for
    for(i=0; i < n; i++)
        work(i);
}
```

```
0 1 2 3
0
1
2
3
```
Parallelizing a nested loop with OpenMP

- We parallelize the outer loop index
- Not all implementations can parallelize inner loops

```c
#pragma omp parallel private(i) shared(n)
#pragma omp for
for(i=0; i < n; i++)
  for(j=0; j < n; j++) {
    u_{new}[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1] - h^2 f[i,j]) / 4
  }
```

- We have an implicit barrier after each loop
- Generated code

```c
mymin = 1 + ($TID * n/nprocs),
mymax = mymin + n/nprocs - 1
for(i=mymin; i < mymax; i++)
  for(j=0; j < n; j++)
    u_{new}[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1] - h^2 f[i,j]) / 4
```
OpenMP is also an API

```c
#ifdef _OPENMP
#include <omp.h>

int nthreads = 1;
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of openMP threads: %d\n", nthreads);
    }
}
#endif
```
Reductions in OpenMP

- OpenMP uses a local accumulator, which it then accumulates globally after the loop finishes.

```c
#pragma omp parallel reduction(+:sum)
for (int i=0; i < N-1; i++)
    sum += x[i];

i0 = $TID*n$/nthreads, i1 = i0 + n/$nthreads;
for (i=i0, localSum=0; i < i1; i++)
    localSum += x[i];

All threads accumulate localSum into Global Sum
```
Reduction - Computing the Residual

double c = 1 / 6.0, err=0;
#pragma omp parallel shared(U,fB,c)
#pragma omp for reduction(+:err)
for (int i=1; i<=nx; i++)
    for (int j=1; j<=ny; j++)
        for (int k=1; k<=nz; k++){
            double du = c * (U[i-1][j][k] + U[i+1][j][k] + U[i][j-1][k] +
                             U[i][j+1][k] + U[i][j][k-1] + U[i][j][k+1] - 6.0*f[i-1][j-1][k-1]);
            double = f[i-1][j-1][k-1] - du;
            err = err + r*r;
        }
double resid = sqrt(err)/(double)((nx+2)*(ny+2)*(nz+2));
Critical Sections

- Only one thread at a time may run the code in a critical section
- Uses mutual exclusion to implement critical sections

```c
#pragma omp parallel // Begin a parallel construct
{
  #pragma omp sections // Begin worksharing
  {
    #pragma omp critical // Critical section
    {x = x + 1}
    #pragma omp critical // Another critical section
    {x = x + 1}
    ... // More Replicated Code
  } // Wait until both units of work complete
} // Wait for all members to arrive
```
Race conditions

• A Race condition arises when the timing of accesses to shared memory can affect the outcome.

• Consider this statement, assume \( x == 0 \)
  \[
  x = x + 1;
  \]

• Generated code
  - \( r1 \leftarrow (x) \)
  - \( r1 \leftarrow r1 + #1 \)
  - \( r1 \rightarrow (x) \)

• Possible interleaving with two threads

  \[\begin{align*}
  & P1 \\
  & r1 \leftarrow x \\
  & r1 \leftarrow r1 + #1 \\
  & x \leftarrow r1
  \end{align*}\]

  \[\begin{align*}
  & P2 \\
  & r1 \rightarrow x \\
  & r1 \leftarrow r1 + #1 \\
  & x \leftarrow r1
  \end{align*}\]

  - \( r1(P1) \) gets 0
  - \( r2(P2) \) also gets 0
  - \( r1(P1) \) set to 1
  - \( r1(P1) \) set to 1
  - \( P1 \) writes its \( R1 \)
  - \( P2 \) writes its \( R1 \)
Consequences of race conditions

• We say we have a *non-deterministic* computation
• Usually we want to avoid non-determinism
• If we compute with the same inputs we want to obtain the same results
• Not necessarily true for operations that have side effects (global variables, I/O and random number generators)
• We need to take steps to avoid race conditions through appropriate program synchronization
  ✓ Critical sections
  ✓ Barriers
  ✓ Atomic functions
Critical Sections

• Each thread sums into the shared variable $x$, to which it has momentary, exclusive access
• Threads take turns executing a \textit{critical section}
• A critical section is non-parallelizing computation and we can implement it with mutual exclusion

\begin{verbatim}
Begin Critical Section
    global_sum += x[i] ;
End Critical Section
\end{verbatim}
Implementation issues

• Primitive atomic operations can be more efficient

• Hardware support
  ♦ Test and set: atomically test a memory location and then set it
  ♦ Cache coherence protocol provides synchronization

• Scheduling issues
  ♦ Busy waiting or spinning
  ♦ Yield process
  ♦ Pre-emption by scheduler
Multithreaded Solve()

Local mymin = 1 + ($TID * n/$nprocs),
     mymax = mymin + n/$nprocs -1;
Global resid, U[::,], U_{new}[::,]
Local  done = FALSE;
while (!done) do
    Local myResid = 0;
    resid = 0;
    update $U_{new}$ and myResid
    resid += myResid;
    if (resid < Tolerance) done = TRUE;
U[mymin:mymax,:) = $U_{new}[mymin:mymax,:];
end while

for i = mymin to mymax do
    for j = 1 to n do
        $U_{new}[i,j] = ...$
        myresid += ...
    end for
end for

Is this code correct?
Correctness

Local mymin = 1 + ($TID * n / nprocs),
     mymax = mymin + n / nprocs - 1;
Global resid, U[:, :], U^{new}[:, :]
Local done = FALSE;
while (!done) do
    Local myResid = 0;
    BARRIER
    Only on thread 0: resid = 0;
    BARRIER
    update U^{new} and myResid
    CRITICAL SEC: resid += myResid
    BARRIER
    if (resid < Tolerance) done = TRUE;
    Only on thread 0: U[mymin:mymax, :] = U^{new}[mymin:mymax, :];
end while

for i = mymin to mymax do
    for j = 1 to n do
        U^{new}[i, j] = ...
        myresid += ...
    end for
end for

Does this code use minimal synchronization?

10/8/12  ©2012 Scott B. Baden /CSE 260/ Fall 2012
Dealing with Data Dependences
Loop carried dependences

• OpenMP will dutifully parallelize a loop when you tell it to, even if doing so “breaks” the correctness of the code

```cpp
int* fib = new int[N];
    fib[0] = fib[1] = 1;
#pragma omp parallel for num_threads(2)
for (i=2; i<N; i++)
    fib[i] = fib[i-1] + fib[i-2];
```

• Sometimes we can restructure an algorithm, as in odd/even sorting
Parallel Sorting

• Sorting is fundamental algorithm in data processing
  ◆ Given an unordered set of keys $x_0, x_1, \ldots, x_{N-1}$
  ◆ Return the keys in sorted order
• The keys may be character strings, floating point numbers, integers, or any object for which the relations $>$, $<$, and $==$ hold
• We’ll assume integers here
• Will talk about other algorithms later on
Compare and exchange sorts

- Simplest sort, based on the bubble sort algorithm
- The fundamental operation is compare-exchange
- **Compare-exchange(a[j], a[j+1])**
  - swaps its arguments if they are in decreasing order
  - satisfies the post-condition that $a[j] \leq a[j+1]$
  - Returns FALSE if a swap was made

```plaintext
for i = N-1 to 1 by -1 do 
  done = TRUE;
  for j = 0 to i-1 do // Compare-exchange(a[j], a[j+1])
    if (a[i] < a[j]) { a[i] ↔ a[j];
      done = FALSE; }
  end do
  if (done) break;
end do
```
Loop carried dependencies

- We cannot parallelize bubble sort owing to the *loop carried dependence* in the inner loop.
- The value of $a[j]$ computed in iteration $j$ depends on the $a[i]$ computed in iterations $0, 1, \ldots, j-1$.

```plaintext
for i = N-1 to 1 by -1 do
    done = TRUE;
    for j = 0 to i-1 do
        done = Compare-exchange(a[j], a[j+1])
    end do
    if (done) break;
end do
```
Odd/Even sort

• If we re-order the comparisons we can parallelize the algorithm
  ♦ number the points as even and odd
  ♦ alternate between sorting the odd and even points
• This algorithm parallelizes since there are no loop carried dependences
• All the odd (even) points are decoupled
The algorithm

done = false;

for i = 0 to n-1 do

  for j = 0 to n-1 by 2 do  // Even
    done &= Compare-exchange(a[j], a[j+1]);
  end do

  for j = 1 to n-1 by 2 do  // Odd
    done &= Compare-exchange(a[j], a[j+1]);
  end do
  if (done) break;

end do
Odd/Even Sort Code

int OE = lo % 2;
for (s = 0; s < MaxIter; s++) {
    int done = Sweep(Keys, OE, lo, hi); /* Odd phase */

    done &= Sweep(Keys, 1-OE, lo, hi); /* Even phase */

    if (done){
        s++;
        break;
    }
}

} /* End For */
Inside Sweep

```c
int Sweep(int *Keys, int OE, int lo, int hi){
    int done = 1;
    #pragma omp parallel for shared(done) private(i) reduction(*:done)
    for (int i = OE+lo; i <= Hi; i+=2) {
        if (Keys[i] > Keys[i+1]){  
            Keys[i] ← Keys[i+1];
            done *= 0;
        }
    }
    return done;
}
```
Memory system behavior

- Off processor values surround each local subproblem
- Non-contiguous data
- Inefficient to access values on certain faces/edges; poor utilization of cache
- False sharing of boundary data
OpenMP under the hood

- A program begins life as a single “boss” thread
- When the boss encounters a parallel construct, it creates a team of worker threads
- The lexically enclosed program statements execute in parallel by all team members
- When we reach the end of the scope…
  - The team of threads synchronize and are dissolved; they enter a wait state
  - Only the master thread continues,
- Thread teams can be created and dissolved many times during program execution
  
  www.ncsa.uiuc.edu/UserInfo/Resources/Software/Intel/Compilers/10.0/main_cls/mergedProjects/optaps_cls/whskin_homepage.htm
- A clever compiler can avoid so many thread creations and joins