Lecture 3

Stencil methods
Multithreaded programming with OpenMP
Announcements

• Office hours in EBU3B Room 3244
  ◆ Mondays 2.30 to 3.30pm; Thurs 4:30pm-3pm

• Partners

• Lilliput accounts

• Programming Lab #1
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)
  \[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \]
  within a square box, \( x,y \in [0,1] \)
  \[ u(x,y) = \sin(x)\sin(y) \]
  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
  \[-u''(x) = f(x), \quad x \in [0,1]\]

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

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

• 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 Gauss-Seidel algorithm will arrive at the solution …

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

\[
\text{Repeat until the result is satisfactory for } i = 1 : N-1 \\
\quad u_i = (u_{i+1} + u_{i-1} + h^2 f_i)/2 \\
\text{end for} \\
\text{end Repeat}
\]
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
  \[
  \frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2} = f_i, \ i \in 1..n-1
  \]
  \[Au = f\]

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

• 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)
  \[
  \sqrt{\sum_i r_i^2}
  \]
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:
  \[
  \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \Delta u = f(x, y) \text{ within a square box, } x, y \in [0, 1] \\
  u(x, y) = \sin(x) \sin(y) \text{ on } \partial \Omega, \text{ perimeter of the box}
  \]
  Define $u_{ij} = u(x_i, y_j)$ 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 \(0:N-1 \times 0:N-1\)
\[u'[i,j] = \frac{u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1] - h^2 f[i,j]}{4}\]

\[u = u'\]
Roundoff

• 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
Programming with threads
SPMD execution model

• Most parallel programming is implemented under the Same Program Multiple Data programming model = SPMD
  ✷ Threads
  ✷ Message passing
  ✷ Other names: “loosely synchronous” or “bulk synchronous”

• Programs execute as a set of P processes or threads
  ✷ We specify P when we run the program
  ✷ Each process/thread is usually assigned to a different physical processor

• Each process or thread
  ✷ is initialized with the same code
  ✷ has an associated index or rank, a unique integer in the range 0:P-1
  ✷ executes instructions at its own rate

• Processes communicate via messages, threads through shared memory
Shared memory programming with threads

- Program executes a collection of independent instruction streams, called *threads*
- Each thread
  - is usually assigned to a distinct physical processor
  - has a *thread index*, a unique integer in \([0:NT-1]\)
- 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
  - Both share global storage with caller
  - Synchronization may be needed when updating shared state (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
Programming model

- Start with a single root thread
- Fork-join parallelism to create concurrently executing threads
- Threads may or may not execute on different processors, and might be interleaved
- Scheduling behavior specified separately
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);
}
```

```
i0 = $TID*n/$nthreads;
i1 = i0 + n/$nthreads;
for (i=i0; i < i1; i++)
    work(i);
```
```
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");
```
Multithreaded Jacobi’s method

- Off processor values surround each local subproblem
- Non-contiguous data
- Inefficient to access values on certain faces/edges; poor utilization of cache
Partitioning

- Splits up the data over processors
- 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

![Partitioning Diagram](image)
Workload decomposition in OpenMP

- We use static assignment here, since n is known
- Dynamic assignment for irregular problems (later on)
- Translator automatically generates appropriate local loop bounds

```
#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
```
3D Jacobi in OpenMP

```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
```
Computational loop

FLOAT c = 1/ 6.0, h = 1.0, c2 = h * h;

for (it= 0; it<nIters; it++) {
#pragma omp parallel shared(U,Un,b,nx,ny,nz,c2,c) private(i,j,k)
#pragma omp for schedule(static,bi)
    for (int i=1; i<\nx; i++)
        for (int j=1; j<ny; j++)
            for (int k=1; k<\nz+1; k++)
                Un[i][j][k] = 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] - c2*b[i-1][j-1][k-1]);

    Grid3D tmp = U;
    U = Un;
    Un = tmp
}
Computing the residual

FLOAT resid7(Grid3D U, Grid3D B, const int nx, const int ny, const int nz){
    double c = 1 / 6.0, err=0;
    #pragma omp parallel shared(U,B,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++){
                FLOAT 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* B[i-1][j-1][k-1]);
                FLOAT r = B[i-1][j-1][k-1] - du;
                err = err + r*r;
            }
    return sqrt(err)/(float)((nx+2)*(ny+2)*(nz+2));
}
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
```