Lecture 6

Floating Point Arithmetic
Stencil Methods
Introduction to OpenMP
Announcements

• Section and Lecture will be switched next week
  • Thursday: section and Q2
  • Friday: Lecture
Today’s lecture

• Floating point arithmetic
• Stencil methods
• Introduction to OpenMP
Floating Point Arithmetic
What is floating point?

- A representation
  - $\pm 2.5732\ldots \times 10^{22}$
  - NaN ∞
  - Single, double, extended precision

- A set of operations
  - + = * / √ rem
  - Comparison < ≤ = ≠ ≥
  - Conversions between different formats, binary to decimal
  - Exception handling

- IEEE Floating point standard P754
  - Universally accepted
  - W. Kahan received the Turing Award in 1989 for design of IEEE Floating Point Standard
  - Revision in 2008
IEEE Floating point standard P754

- Normalized representation \( \pm 1.d\cdots d \times 2^{\text{exp}} \)
  - \( \text{Macheps} = \text{Machine epsilon} = \varepsilon = 2^{-\#\text{significand bits}} \)
  - relative error in each operation
  - \( \text{OV} = \text{overflow threshold} = \text{largest number} \)
  - \( \text{UN} = \text{underflow threshold} = \text{smallest number} \)
- \( \pm \text{Zero}: \pm \text{significand and exponent} = 0 \)

<table>
<thead>
<tr>
<th>Format</th>
<th># bits</th>
<th>#significand bits</th>
<th>macheps</th>
<th>#exponent bits</th>
<th>exponent range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>32</td>
<td>23+1</td>
<td>2^{-24} (~10^{-7})</td>
<td>8</td>
<td>2^{-126} - 2^{127} (~10^{±38})</td>
</tr>
<tr>
<td>Double</td>
<td>64</td>
<td>52+1</td>
<td>2^{-53} (~10^{-16})</td>
<td>11</td>
<td>2^{-1022} - 2^{1023} (~10^{±308})</td>
</tr>
<tr>
<td>Double</td>
<td>≥80</td>
<td>≥64</td>
<td>≤2^{-64} (~10^{-19})</td>
<td>≥15</td>
<td>2^{-16382} - 2^{16383} (~10^{±4932})</td>
</tr>
</tbody>
</table>

© 2010 Scott B. Baden / CSE 160 / Winter 2011
Roundoff

• 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
What happens in a floating point operation?

- Round to the nearest representable floating point number that corresponds to the exact value (correct rounding)
- Round to nearest value with the lowest order bit = 0 (rounding toward nearest even)
- Others are possible
- We don’t need the exact value to work this out!
- Applies to \(+ = * / \sqrt{\text{rem}}\)
  
  Error formula: \(\text{fl}(a \text{ op } b) = (a \text{ op } b)*(1 + \delta)\) where
  - op one of \(+ , - , * , /\)
  - \(|\delta| \leq \varepsilon\)
  - assuming no overflow, underflow, or divide by zero

- Addition example
  - \(\text{fl}(\sum x_i) = \sum_{i=1:n} x_i*(1+e_i)\)
  - \(|e_i| \sim < (n-1)\varepsilon\)
Exception Handling

• An exception occurs when the result of a floating point operation is not representable as a normalized floating point number
  ‣ 1/0, √-1

• P754 standardizes how we handle exceptions
  ‣ **Overflow**: exact result > OV, too large to represent
  ‣ **Underflow**: exact result nonzero and < UN, too small to represent
  ‣ **Divide-by-zero**: nonzero/0
  ‣ **Invalid**: 0/0, √-1, log(0), etc.
  ‣ **Inexact**: there was a rounding error (common)

• Two possible responses
  ‣ Stop the program, given an error message
  ‣ Tolerate the exception
An example

• Graph the function

\[ f(x) = \frac{\sin(x)}{x} \]

• But we get a singularity @ x=0: \( \frac{1}{x} = \infty \)
• This is an “accident” in how we represent the function (W. Kahan)
• \( f(0) = 1 \)
• We *catch* the exception (divide by 0)
• *Substitute* the value \( f(0) = 1 \)
NaN (Not a Number)

• Invalid exception
  ‣ Exact result is not a well-defined real number
  ‣ 0/0, √-1

• We can have a quiet NaN or an sNan
  ‣ Quiet –does not raise an exception, but propagates a distinguished value
    • E.g. missing data: max(3,NAN) = 3
  ‣ Signaling - generate an exception when accessed
    • Detect uninitialized data
When compiler optimizations alter precision

- Let’s say we support 79+ bit extended format in registers.
- When we store values into memory, values are converted to the lower precision format.
- Compilers can keep things in registers and we may lose referential transparency.

```c
float x, y, z;
int j;

....

x = y + z;
if (x >= j) replace x by something smaller than j
y = x;
```

- With optimization turned on, x is computed to extra precision; it is not a float.
- If x < j in a register, there is no guarantee the condition will be preserved when x is stored in y, i.e. y >= j.
Today’s lecture

• Floating point arithmetic
• Stencil methods
• Introduction to OpenMP
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, \( x, y \in [0, 1] \)

\[ u(x, y) = \sin(x) \ast \sin(y) \] 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

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

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

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

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

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

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

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

\[ u = u' \]
Today’s lecture

• Floating point arithmetic
• Stencil methods
• Introduction to OpenMP
OpenMP programming

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

```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 to spawn work groups of multiple threads
- For-join can be logical; a clever compiler can use barriers instead
Sections

#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
For 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");
```

© 2010 Scott B. Baden / CSE 160 / Winter 2011
Partitioning

• How we split up the data over processors
• Different partitioning according to the *processor geometry*
• For $P$ processors geometries are of the form $p_0 \times p_1$, where $P = p_0 \times p_1$
• For $P=4$: 3 possible geometries
• We’ll use 1D; not all openmp implementations support more

The geometries are illustrated as follows:

- **1 x 4**
  - 0
  - 1
  - 2
  - 3

- **4 x 1**
  - 0
  - 1
  - 2
  - 3

- **2 x 2**
  - 0
  - 1
  - 2
  - 3
Workload decomposition in Jacboi’s Method

• We use static assignment, since \( n \) is known
• We parallelize the outer loop index
• Not all implementations can parallelize inner loops
• Dynamic assignment for irregular problems (later on)

```c
#pragma omp parallel private(i) shared(n)
{
    while (not converged) {
        #pragma omp parallel private(i) shared(n)
            #pragma omp for
        for(i=0; i < n; i++)
            for(j=0; j < n; j++) {
                \[ u_{\text{new}}[i,j] = \frac{(u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1] - h^2f[i,j])}{4} \]
            }
    }
    swap u and \( u_{\text{new}} \)
}
```
Generated code

```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
    }
```

- Compiler generates an implicit barrier after each loop

```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
```

© 2010 Scott B. Baden / CSE 160 / Winter 2011
Fin
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

```c
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));
}
```
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
Race Conditions

#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
    {x = x + 1;}

    #pragma omp section    // Another unit
    {x = x + 1;}

    } // Wait until both units complete

} // End of Parallel Construct; disband team

// continue serial execution
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
        #pragma omp barrier // Wait for all members to arrive
    } // Wait until both units of work complete
}
```

© 2010 Scott B. Baden / CSE 160 / Winter 2011
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];
```

```c
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*
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?
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;
    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

Does this code use minimal synchronization?

\begin{align*}
\text{for } i &= \text{mymin to mymax do} \\
\text{   for } j &= 1 \text{ to n do} \\
\text{      } U^\text{new}[i,j] &= \ldots \\
\text{      myresid} += \ldots \\
\text{   end for} \\
\text{end for}
\end{align*}