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

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

Today's lecture

- Floating point arithmetic
- Stencil methods
- Introduction to OpenMP

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

Floating Point Arithmetic

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

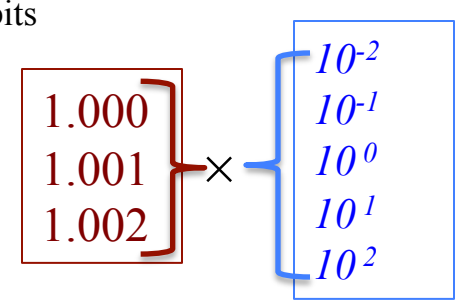
What is floating point?

- A representation
  - $\pm 2.5732\ldots \times 10^{22}$
  - NaN ∞
  - Single, double, extended precision
- A set of operations
  - + = * / $\sqrt{}$ rem
  - Comparison $< \leq = \neq \geq >$
  - 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

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

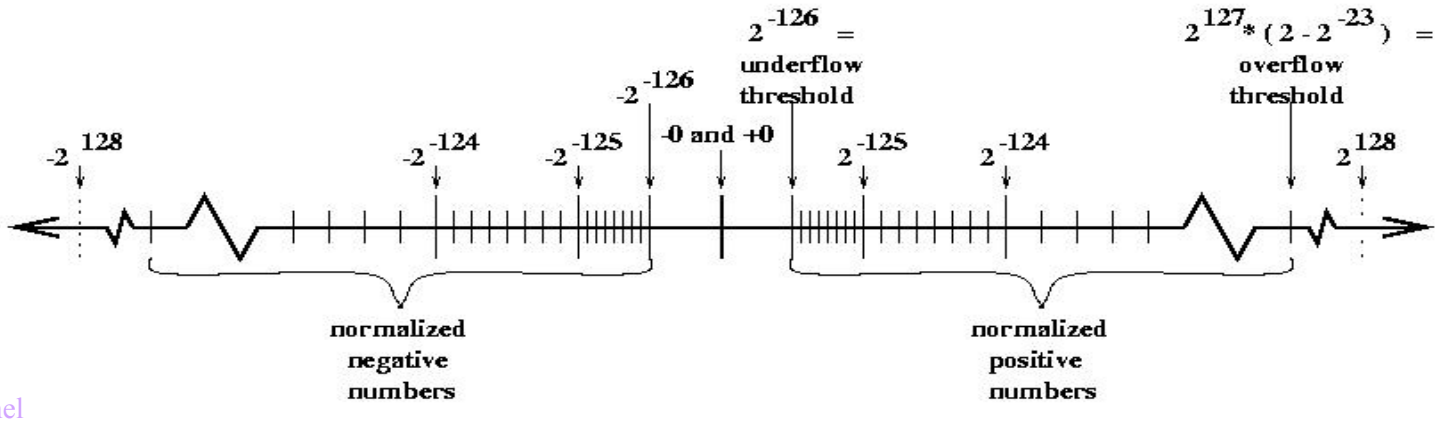
IEEE Floating point standard P754

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



0.01000, 0.01001 100.0, 100.1

Format	# bits	#significand bits	macheps	#exponent bits	exponent range
Single	32	23+1	2^{-24} (~10^{-7})	8	2^{-126} - 2^{127} (~$10^{\pm 38}$)
Double	64	52+1	2^{-53} (~10^{-16})	11	2^{-1022} - 2^{1023} (~$10^{\pm 308}$)
Double	≥80	≥64	≤2^{-64}(~10^{-19})	≥15	2^{-16382} - 2^{16383} (~$10^{\pm 4932}$)



Jim Demmel

© 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

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

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 + = * / √ rem
- Error formula: fl(a op b) = (a op b)*(1 + δ) where
  - op one of + , - , * , /
  - | δ | ≤ ε
  - assuming no overflow, underflow, or divide by zero
- Addition example
  - $fl(\sum x_i) = \sum_{i=1:n} x_i*(1+e_i)$
  - $|e_i| \tilde{<} (n-1)\varepsilon$

9.99
0.00~~710~~
9.99

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

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

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

An example

- Graph the function

$$f(x) = \sin(x) / x$$

- But we get a singularity @ x=0: $1/x = \infty$
- 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

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

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

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

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*

```
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

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

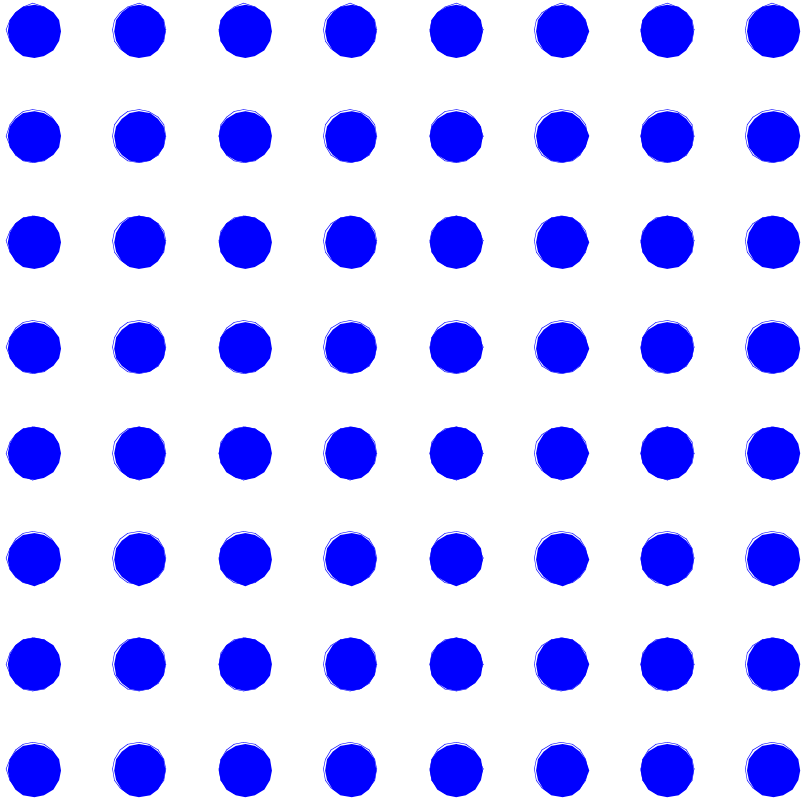
Today's lecture

- Floating point arithmetic
- Stencil methods
- Introduction to OpenMP

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

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



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

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)

  $\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 = 0$ 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)

$$-u''(x) = f(x),\ x \in [0,1],\ u(0) = a,\ u(1) = b$$

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

Solving an ODE with a discrete approximation

- Solve the ODE

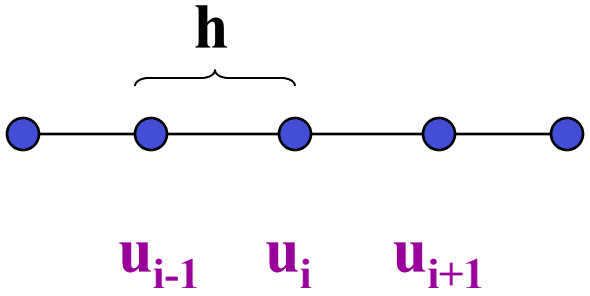
  $-u''(x) = f(x),\ x \in [0,1]$

- 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

  $(u_{i-1} - 2u_i + u_{i+1})/h^2 = f_i \quad i \in 1..n-2$

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

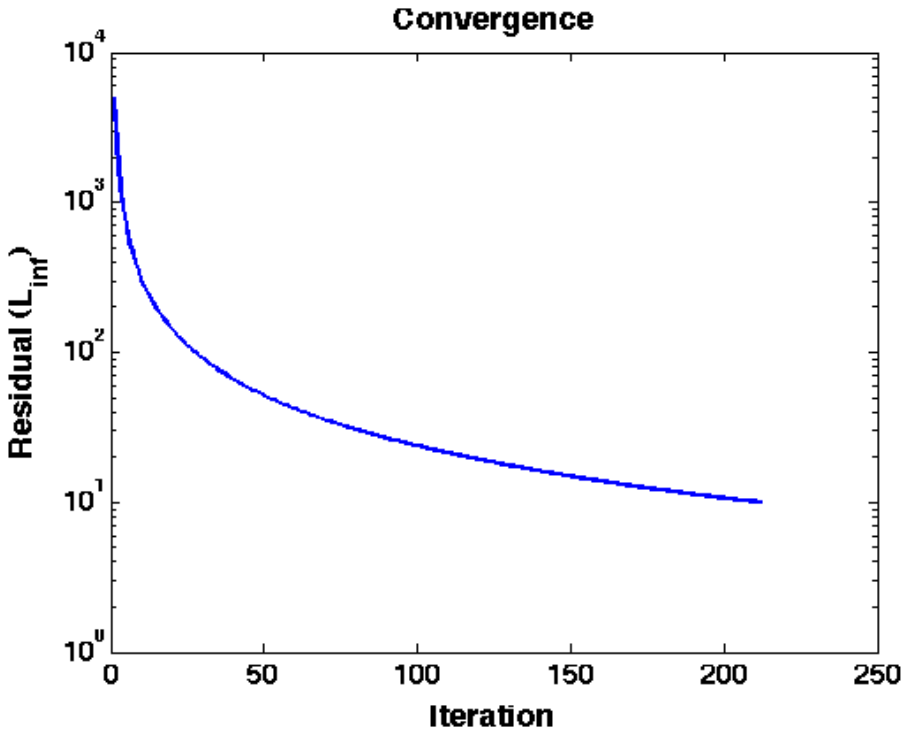
Iterative solution

- 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-2
      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



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

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

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

Using the residual to estimate the error

- Recall the equations

  $(-u_{i-1} + 2u_i - u_{i+1})/h^2 = f_i \text{ , } i \in 1..n\text{-}1 \quad [Au = f]$

- Define the ***residual*** r_i: $[r = Au\text{-}f]$

  $r_i = (-u_{i-1} + 2u_i - u_{i+1})/h^2 - f_i \text{ , } i \in 1..n\text{-}1$

- 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)

$$\sqrt{\sum_i r_i^2}$$

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

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:

  $\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 = \Delta u = f(x,y)$ within a square box, $x,y \in [0,1]$

  $u(x,y) = \sin(x)*\sin(y)$ on $\partial\Omega$, perimeter of the box

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

- Approximate the derivatives

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

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

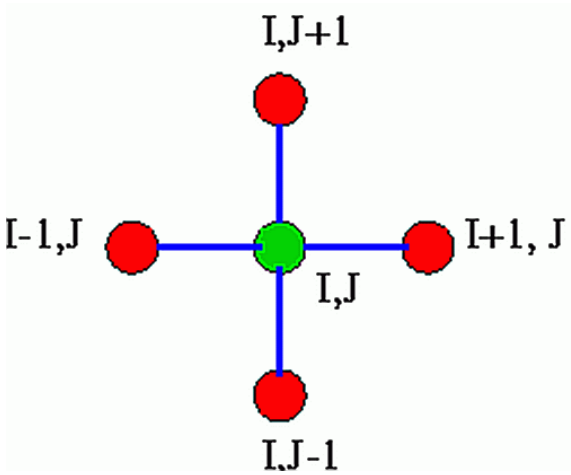
Jacobi's Method in 2D

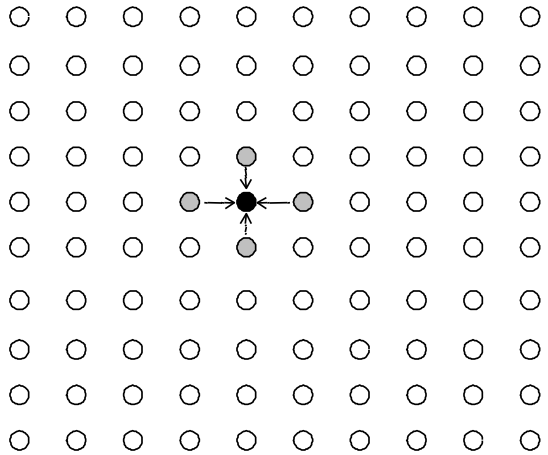
- The update formula

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

  u =u'

$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$





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

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

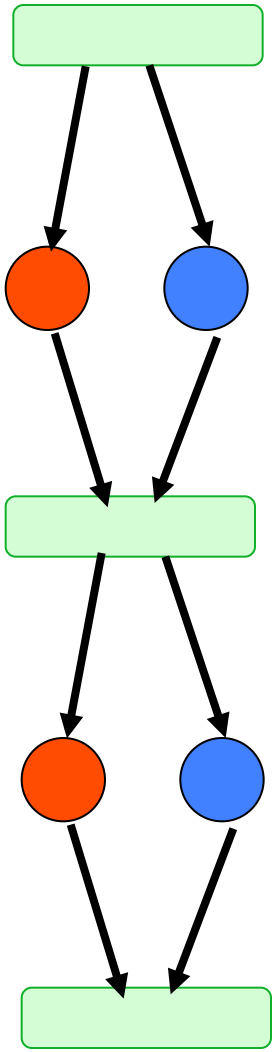
```
#pragma omp parallel private(i) shared(n)
{
#pragma omp for
for(i=0; i < n; i++)
    work(i);
}
```

```
i0 = $TID*n/$nthreads;
i1 = i0 + n/$nthreads;
for (i=i0; i< i1; i++)
   work(i);
```

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

OpenMP Fork-Join Model

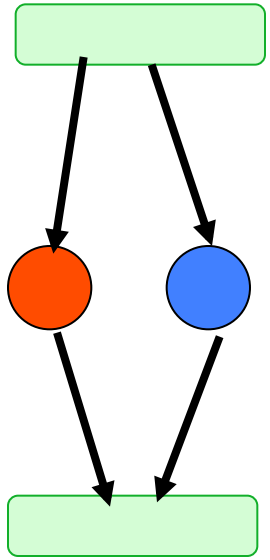
- A program begins life as a single thread
- Parallel regions to spawn work groups of multiple threads
- For-join can be logical; a clever compiler can use barriers instead



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

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
```

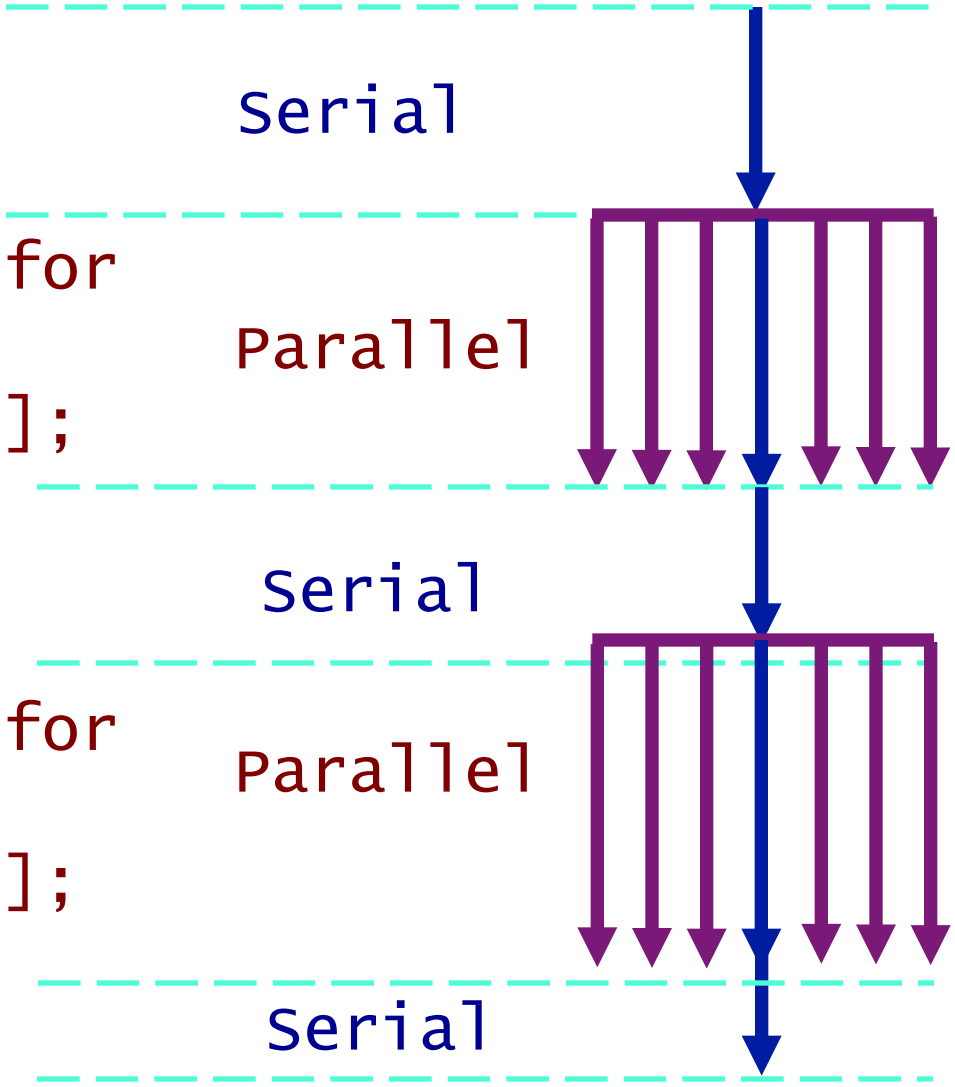


© 2010 Scott B. Baden / CSE 160 / Winter 2011

For loops

Seung-Jai Min

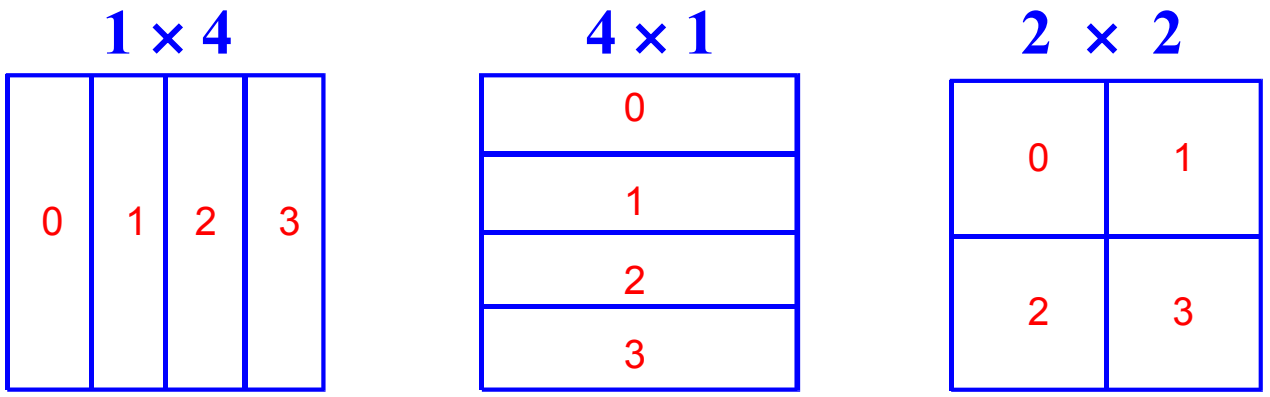
```
printf("Start\n");
N = 1000;

#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 $P = p_0\, p_1$
- For P=4: 3 possible geometries
- We'll use 1D; not all openmp implementations support more



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

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)

```
#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^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
      }
    }
  swap u and u^new
}
```

Generated code

```
#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^2f[i,j])/4
   }
```

- Compiler generates an implicit barrier after each loop

```
   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^2f[i,j])/4
   }
```

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

OpenMP is also an API

```
#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
}
```

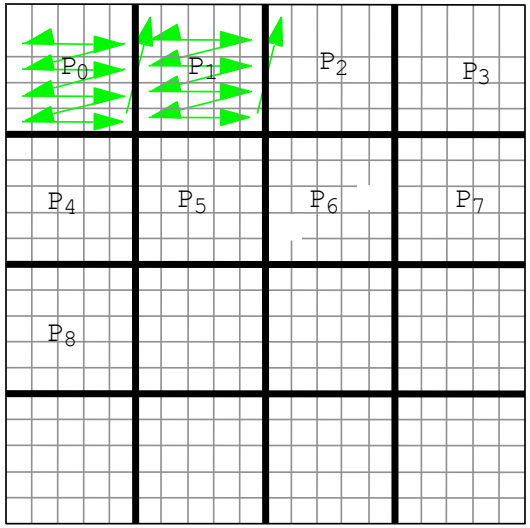
Computing the residual

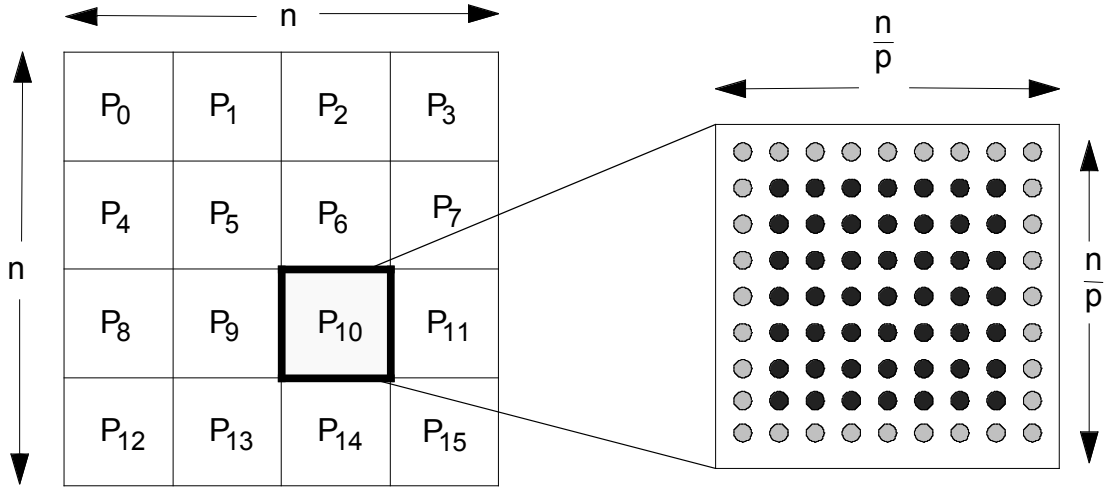
```
FLOAT  resid7(Grid3D U, Grid3D B, const int nx, const int ny, const int nz){
double c = 1 / 6.0, err=0;
#pragma omp parallel shared(U,B,c)
#pragma omp for reduction(+:err)
for (int i=1; i<=nx; i++)
  for (int j=1; j<=ny; j++)
    for (int k=1; k<=nz; k++){
      FLOAT du = c * (U[i-1][j][k] + U[i+1][j][k]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

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

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

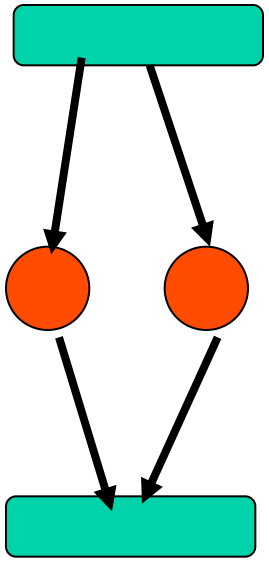




© 2010 Scott B. Baden / CSE 160 / Winter 2011

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
```

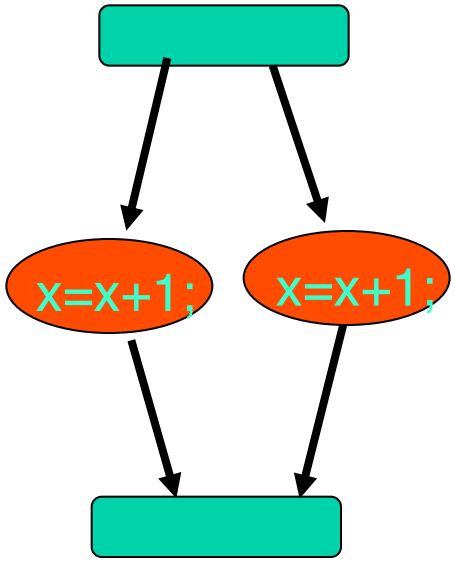


© 2010 Scott B. Baden / CSE 160 / Winter 2011

Critical Sections

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

```
#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

```
#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];
```

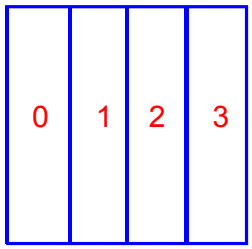
All threads accumulate localSum into Global Sum

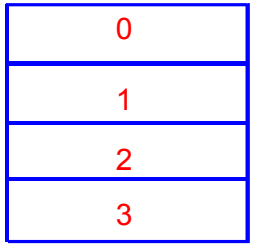
© 2010 Scott B. Baden / CSE 160 / Winter 2011

Multithreaded Solve()

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

```
for i = mymin to mymax do
      for j = 1 to n do
          Unew[i,j] = …
          myresid+= …
      end for
end for
```





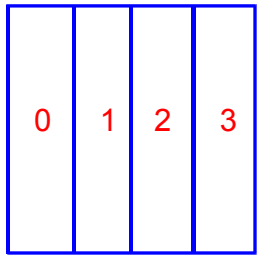
Is this code correct?

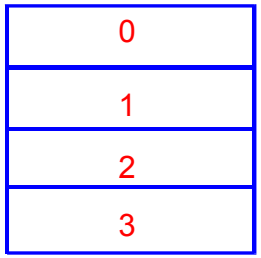
© 2010 Scott B. Baden / CSE 160 / Winter 2011

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
```

```
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?

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