

Lecture 12

Stencil methods

Atomics

Announcements

- Midterm scores have been posted to Moodle
Mean/Median: 63/64 ~79% (B-/C+)
- The midterm will be returned in section on Friday

Today's lecture

- OpenMP – some finer points on language law
- Stencil Methods
- Programming Assignment #4
- Atomics

What does the fine print of a specification tell us?

- OpenMP 3.1 spec@ [gcc 4.7 & 4.8]

<http://www.openmp.org/mp-documents/OpenMP3.1.pdf>

A compliant implementation of the static schedule must ensure that the same assignment of logical iteration numbers to threads will be used in two loop regions if the following conditions are satisfied for both loop regions:

- 1) have the same number of loop iterations,*
- 2) have the same value of chunk_size specified, or have no chunk_size specified,*
- 3) bind to the same parallel region. A data dependence between the **same logical iterations** in two such loops is guaranteed to be satisfied allowing safe use of the nowait clause*

```
#pragma omp parallel
{
# pragma omp for schedule static nowait
    for (int i=1; i< N-1; i++)
        a[i] = i;
#pragma omp schedule static for
    for (int i=1; i<N-1; i++)
        b[i] = (a[i+1] - a[i-1])/2h
}
```

Will this code run correctly?

- A. Yes, they have the same number of iterations
- B. Yes, they bind to the same parallel region
- C. Yes, there is no data dependence between the *same logical iterations* in the two such loops
- D. All of A, B and C
- E. No, one or more of A, B, C doesn't hold

```
#pragma omp parallel
{
# pragma omp for schedule
  static nowait
    for (int i=1; i< N-1; i++)
      a[i] = i;
#pragma omp schedule static for
  for (int i=1; i<N-1; i++)
    b[i] = (a[i+1] - a[i-1])/2h
}
```

Will this code run correctly?

- A. Yes, they have the same number of iterations
- B. Yes, they bind to the same parallel region
- C. Yes, there is no data dependence between the *same logical iterations* in the two such loops
- D. All of A, B and C
- E. No, one or more of A, B, C doesn't hold

```
#pragma omp parallel
{
# pragma omp for schedule
static nowait
    for (int i=1; i< N-1; i++)
        a[i] = i;
#pragma omp schedule static for
    for (int i=1; i<N-1; i++)
        b[i] = (a[i+1] - a[i-1])/2h
}
```

```
$ ./assign 8
N = 8
# of openMP threads: 4
A: 0 1 2 3 4 5 6 7
B: 0 1 0.5 3 1.5 2 6 0
$ ./assign 8
A: 0 1 2 3 4 5 6 7
B: 0 1 0.5 3 4 0 6 0
```

Will this code run correctly?

- A. Yes, they have the same number of iterations
- B. Yes, they bind to the same parallel region
- C. Yes, there is no data dependence between the *same logical iterations* in the two such loops
- D. All of A, B and C
- E. No, one or more of A, B, C doesn't hold

```
#pragma omp parallel
{
  #pragma omp for schedule static nowait
    for (int i=1; i< N-1; i++)
      a[i] = i;
  #pragma omp schedule static for
    for (int i=1; i< N-1; i++)
      b[i] = a[i];
}
```

What does the fine print of a specification tell us?

- OpenMP 3.1 spec@ [gcc 4.7 & 4.8]

<http://www.openmp.org/mp-documents/OpenMP3.1.pdf>

A compliant implementation of the static schedule must ensure that the same assignment of logical iteration numbers to threads will be used in two loop regions if the following conditions are satisfied for both loop regions:

- 1) have the same number of loop iterations,*
- 2) have the same value of chunk_size specified, or have no chunk_size specified,*
- 3) bind to the same parallel region. A data dependence between the **same logical iterations** in two such loops is guaranteed to be satisfied allowing safe use of the nowait clause*

```
#pragma omp parallel
{
  #pragma omp for schedule static nowait
    for (int i=1; i< N-1; i++)
      a[i] = i;
  #pragma omp schedule static for
    for (int i=1; i< N-1; i++)
      b[i] = a[i];
}
```

```
$ ./assign 8
N = 8
# of openMP threads: 4
A: 0 1 2 3 4 5 6 7
B: 0 1 2 3 4 5 6 7
$ ./assign 8
A: 0 1 2 3 4 5 6 7
B: 0 1 2 3 4 5 6 7
```

The nowait clause with static scheduling

- If we specify a static schedule, the `nowait` clause will preserve correctness unless there are data dependencies across the loops
- The left code block will fail, the right will succeed



```
#pragma omp parallel
{
# pragma omp for schedule static nowait
    for (int i=1; i< N-1; i++)
        a[i] = i;
#pragma omp schedule static for
    for (int i=1; i<N-1; i++)
        b[i] = (a[i+1] - a[i-1])/2h
}
```

```
#pragma omp parallel
{
#pragma omp for schedule static nowait
    for (int i=1; i< N-1; i++)
        a[i] = i;
#pragma omp schedule static for
    for (int i=1; i< N-1; i++)
        b[i] = a[i];
}
```

Implementation dependent details

- Set up when the compiler is built
- For example, what schedule do we get if we don't specify it?



According to the specification for OpenMP3.1

2.5.1.1. When execution encounters a loop directive, the schedule clause (if any) on the directive, and the *run-sched-var* and *def-sched-var* ICVs are used to determine how loop iterations are assigned to threads. See Section 2.3 on page 28 for details of how the values of the ICVs are determined. If the loop directive does not have a schedule clause then the current value of the *def-sched-var* ICV determines the schedule

2.3. An OpenMP *implementation must act as if there were internal control variables* (ICVs) that *control the behavior of an OpenMP program*. These ICVs store information such as the number of threads to use for future parallel regions, *the schedule to use for worksharing loops* and whether nested parallelism is enabled or not. The ICVs are given values at various times (described below) during the execution of the program. They are initialized by the implementation itself and *may be given values through OpenMP environment variables* and through calls to OpenMP API routines. The program can retrieve the values of these ICVs only through OpenMP API routines. For purposes of exposition, this document refers to the ICVs by certain names, but an implementation is not required to use these names or to offer any way to access the variables other than through the ways shown in Section 2.3.2 on page 29.

Modifying and Retrieving ICV Values

- According to the specification for OpenMP3.1



ICV	Ways to modify	Ways to retrieve	Initial value
run-sched-var	OMP_SCHEDULE omp_set_schedule()	omp_get_schedule()	Implementation defined (On Bang, g++4.8.4: Dynamic)
nest-var	OMP_NESTED omp_set_nested()	omp_get_nested()	False (On Bang: False)

Why might the results be incorrect?

- When we don't specify the schedule, the schedule is implementation dependent; it could be dynamic
- On Bamboo, gcc4.8.4 specifies dynamic, but on the Stampede system @ TACC, Intel's compiler chooses static
- But with dynamic, OpenMP doesn't define the order in which the loop iterations will execute
- The code may or may not run correctly unless we specify static!
- I could not get this code to fail on Bang with $N=1M$, $NT = 8$



2.5.1. Binding. When `schedule(dynamic, chunk_size)` is specified, the iterations are distributed to threads in the team in chunks as the threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed..

```
#pragma omp parallel
shared(N,a,b) private(i)
{
    #pragma omp for
    schedule(dynamic) nowait
        for (i=0; i < N; i++)
            a[i] = i;
    #pragma omp for
    schedule(dynamic) nowait
        for (i=0; i < N-1; i++)
            b[i] = a[i];
}
```

Testing for race conditions

- This code failed on Bang, and also on 16 cores of the Stampede cluster (located at TACC) with v15 of Intel C++ compiler: $c[i] \neq \sqrt{i}$ for at least 1 value of i
- This code “wouldn’t” fail if the middle loop followed the same order as the others, or with the 3rd loop removed



```
#pragma omp parallel shared(N,a,b) private(i)
{
  #pragma omp for schedule(dynamic) nowait
    for (i=0; i< N; i++)
      a[i] = i;
  #pragma omp for schedule(dynamic) nowait
    for (i=0; i< N-1; i++)
      b[i] = sqrt(a[i]);
  #pragma omp for schedule(dynamic) nowait
    for (N-1; i>-1; i--)
      c[i] = b[i];
}
```

Exercise: removing data dependencies

- How can we split into this loop into 2 loops so that each loop parallelizes, the result is correct?

‣ B initially: 0 1 2 3 4 5 6 7

‣ B on 1 thread: 7 7 7 7 11 12 13 14

for i = 0 to N-1

 B[i] += B[N-1-i];

B[0] += B[7], B[1] += B[6], B[2] += B[5]

B[3] += B[4], B[4] += B[3], B[5] += B[2]

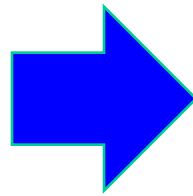
B[6] += B[1], B[7] += B[0]



Splitting a loop: attempt 1

- For iterations $i=N/2+1$ to N , $B[N-i]$ reference newly computed data
- All others reference “old” data
- B initially: 0 1 2 3 4 5 6 7
- Correct result: 7 7 7 7 11 12 13 14

for $i = 0$ to $N-1$
 $B[i] += B[N-i];$

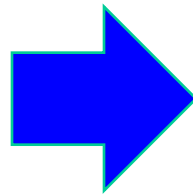


```
#pragma omp parallel  
for ... nowait  
for  $i = 0$  to  $N/2-1$   
     $B[i] += B[N-1-i];$   
for  $i = N/2+1$  to  $N-1$   
     $B[i] += B[N-1-i];$ 
```

Why will the new loops run correctly?

- A. We've eliminated the dependencies in both loops
- B. The second loop runs on 1 core
- C. Both
- D. Not sure

```
for i = 0 to N-1  
    B[i] += B[N-i];
```



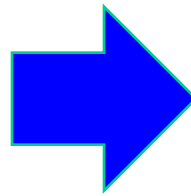
```
#pragma omp parallel  
for ... nowait  
for i = 0 to N/2-1  
    B[i] += B[N-1-i];  
for i = N/2+1 to N-1  
    B[i] += B[N-1-i];
```

Is the nowait clause required?

A. Yes

B. No

```
for i = 0 to N-1  
    B[i] += B[N-i];
```

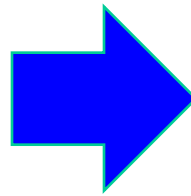


```
#pragma omp parallel  
for ... nowait  
for i = 0 to N/2-1  
    B[i] += B[N-1-i];  
for i = N/2+1 to N-1  
    B[i] += B[N-1-i];
```

Splitting a loop: attempt 2

- For iterations $i=N/2+1$ to N , $B[N-i]$ reference newly computed data
- All others reference “old” data
- B initially: 0 1 2 3 4 5 6 7
- Correct result: 7 7 7 7 11 12 13 14

for $i = 0$ to $N-1$
 $B[i] += B[N-i];$



```
#pragma omp parallel for  
for  $i = 0$  to  $N/2-1$   
     $B[i] += B[N-1-i];$   
#pragma omp parallel for  
for  $i = N/2+1$  to  $N-1$   
     $B[i] += B[N-1-i];$ 
```

Why will the 2 parallel loops run correctly?

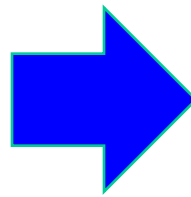
A. The barrier after loop 1 ensures correctness

B. We've eliminated all dependencies by splitting the loop and no barrier is needed

C. The loops won't run correctly

D. Not sure

```
for i = 0 to N-1  
    B[i] += B[N-i];
```



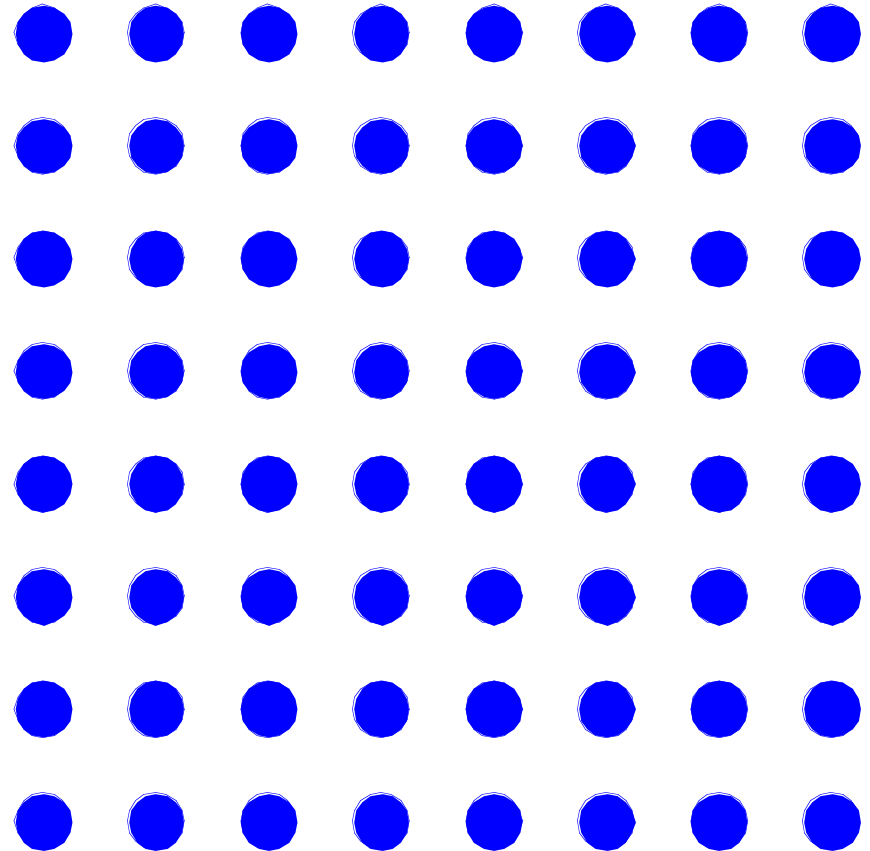
```
#pragma omp parallel for  
for i = 0 to N/2-1  
    B[i] += B[N-1-i];  
#pragma omp parallel for  
for i = N/2+1 to N-1  
    B[i] += B[N-1-i];
```

Today's lecture

- OpenMP – some finer points on language law
- Stencil Methods
- Programming Assignment #4
- Atomics

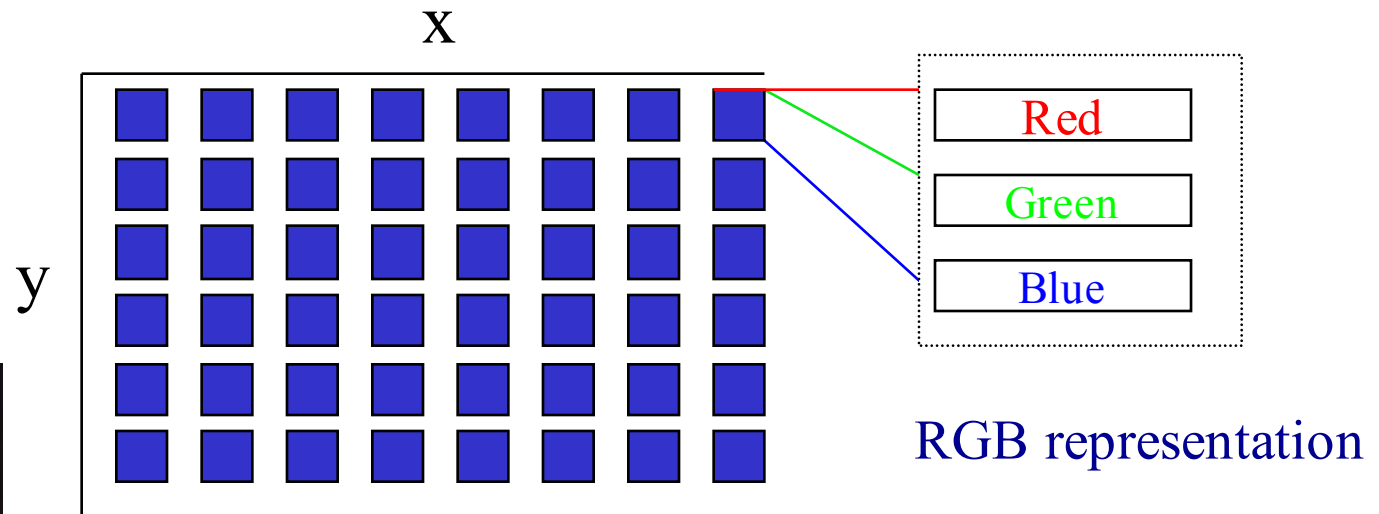
Stencil methods

- 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 mesh is a **mapping** from ordered pairs to **physical observables** like temperature
- Important applications
 - Weather forecasting
 - Image processing



Motivating Application: Digital Image Processing

- We represent the image in terms of *pixels*
- In color image, each pixel can contain 3 colors: RGB



Photos by Martin Juell

wikipedia

Ryan Cuprak

Image smoothing algorithm

- One important operations is called image smoothing
- We replace each pixel by the average of its neighbors
- Repeat as many times as necessary

while not smooth enough do:

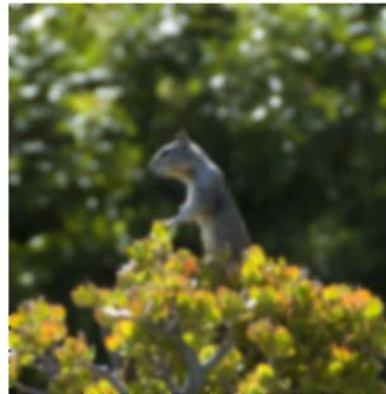
for (i,j) in 0:N-1 x 0:N-1

$I^{new}[i,j] = (I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i,j+1])/4$

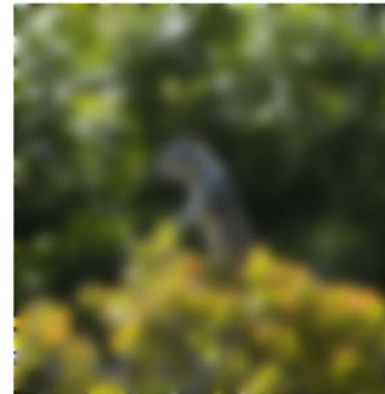
$I = I^{new}$



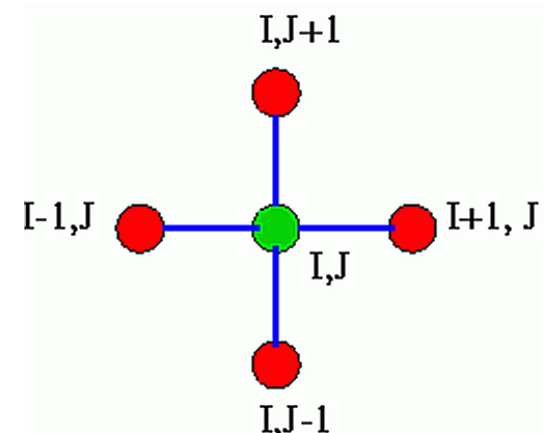
Original



100 iter

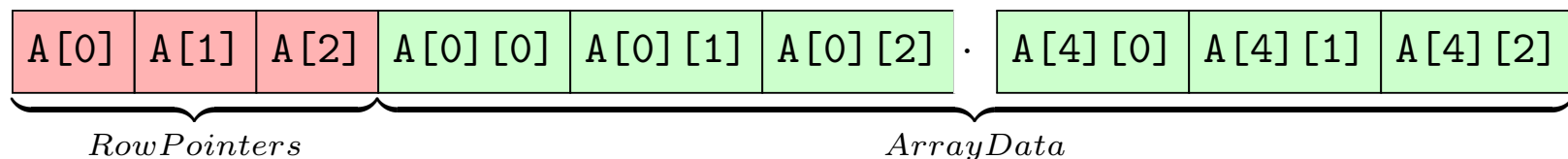
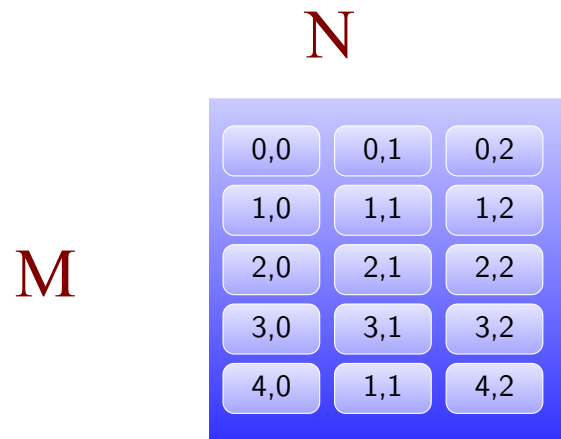


1000 iter



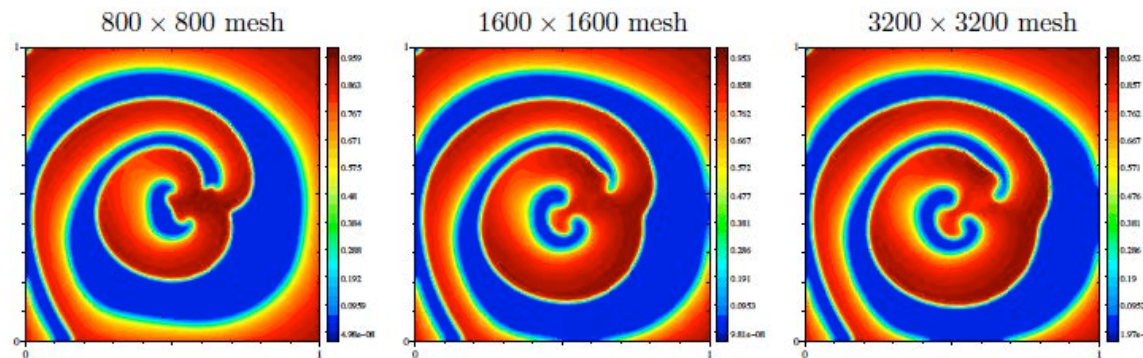
Multidimensional arrays

- Accessing an array by columns is more expensive than accessing by rows



Another motivating application: biomedical computing

- Improve our understanding of disease of the heart
- Model electrical signal propagation in cardiac tissue
 - Demonstrates complex behavior of spiral waves that are known to cause life-threatening situations
- *Reaction-diffusion system*
 - Reactions are the cellular exchanges of certain ions across the cell membrane during the cellular electrical impulse
- Our simulation model has two state variables
 - Transmembrane potential: e
 - Recovery of the tissue: r



The Aliev-Panfilov Model

- Our model has two parts
 - 2 Ordinary Differential Equations:
Kinetics of reactions occurring at every point in space
 - Partial Differential Equation: spatial diffusion of reactants
- 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

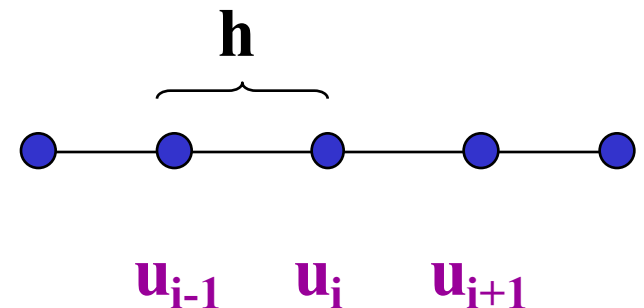
$$\begin{aligned}\frac{\partial e}{\partial t} &= \delta \nabla^2 e - ke(e-a)(e-1) - er, & \text{on } \Omega_T, \\ \frac{\partial r}{\partial t} &= - \left[\varepsilon + \frac{\mu_1 r}{\mu_2 + e} \right] [r + ke(e-b-1)], & \text{on } \Omega_T,\end{aligned}$$

$$\vec{n} \cdot \delta \nabla e = 0 \text{ on } \partial\Omega, \quad \text{and} \quad (e, r)_{t=0} = (e(\cdot, 0), r(\cdot, 0)),$$

Discrete approximations

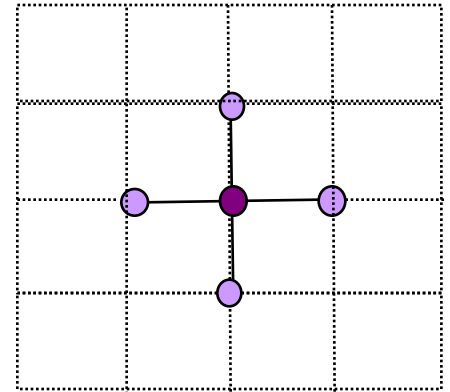
- The functions in the differential equations are continuous
- On the computer, we can represent discrete values only
- We will approximate the solution on the mesh a method known as *finite differences*
- First-order explicit numerical scheme which we will approximate on the mesh: the functions are continuous, but our representation is discrete
- For example, we represent a 2nd derivative

$$u'' \approx (u(x+h) - 2u(x) + u(x-h))/h^2$$



Computational loop of the cardiac simulator

- ODE solver:
 - No data dependency, trivially parallelizable
 - Requires a lot of registers to hold temporary variables
- PDE solver:
 - Jacobi update for the 5-point Laplacian operator.
 - Sweeps over a uniformly spaced mesh
 - Updates voltage to weighted contributions from the 4 nearest neighbors



For a specified number of iterations, using supplied initial conditions:

```
for (j=1; j<=m+1; j++){  
    double *RR = &R[j][1], *EE = &E[j][1];  
    for (i=1; i<=n+1; i++, EE++, RR++) {  
        // PDE SOLVER  
        EE[0]= E_p[j][i]+ $\alpha$ *(E_p[j][i+1]+E_p[j][i-1]-4*E_p[j][i]+E_p[j+1][i]+E_p[j-1][i]);  
        // ODE SOLVER  
        EE[0] += -dt*(kk*EE[0]*(EE[0]-a)*(EE[0]-1)+EE[0]*RR[0]);  
        RR[0] += dt*( $\epsilon$ +M1* RR[0]/( EE[0]+M2))*(-RR[0]-kk*EE[0]*(EE[0]-b-1));  
    }  
}
```

Performance of the simulator

- Use valgrind to obtain cache performance, down to the source line (see Getting Started with Bang for instructions)
`make -j N` runs parallelizes the make on N threads
- Program runs very slowly so be sure to use a qlogin node
`valgrind --tool=cachegrind ./apf -n 255 -i 2000`
`cg_annotate --auto=yes cachegrind.out.18164 > Report.txt`

For a specified number of iterations, using supplied initial conditions:

```
==18164== D  refs:      1,777,880,755 (1,382,193,805 rd  + 395,686,950 wr)
==18164== D1 misses:    67,385,322 ( 50,592,395 rd  + 16,792,927 wr)
==18164== D1 miss rate:      3.7% (      3.6%   +      4.2%   )
```

...

Where is the time spent?

- Memory accesses are costly

```

I1 cache:      32768 B, 64 B, 8-way associative
D1 cache:      32768 B, 64 B, 8-way associative
LL cache:      4194304 B, 64 B, 16-way associative
Command:       ./apf -n 255 -i 2000
Data file:     cachegrind.out.18164
               Dr      D1mr

```

```

-----
1,382,193,805 50,592,395 PROGRAM TOTALS
1,381,488,017 50,566,005 solve.cpp:solve( ...)
      .      .      .      // Fills in the TOP Ghost Cells
      10,000      1,999 for (i = 0; i < (n+3); i++)
      516,000      66,000     E_prev[i] = E_prev[i + (n+3)*2];
                          // Fills in the RIGHT Ghost Cells
      10,000          0 for i = (n+2); i < (m+3)*(n+3); i+=(m+3))
      516,000      504,003     E_prev[i] = E_prev[i-2];
                          // Solve for the excitation, a PDE
      1,064,000      8,000 for(j =innerBlkRowStartIndx;j <=innerBlkRowEndIndx; j+=(m+3)){
      1,024,000      2,000     for(i = 0; i <= n; i++) {
      721,920,001 16,630,000         E_tmp[i] = E_prev_tmp[i]+alpha*(E_prev_tmp[i+1]...)
      }
                          // Solve the ODEs
      4,000      4,000 for(j=innerBlkRowStartIndx; j <= innerBlkRowEndIndx;j+=(m+3)){
      262,144,000 33,028,000         E_tmp[i] += -dt*(kk*E_tmp[i]*(E_tmp[i]-a)..)*R_tmp[i]);
      393,216,000      4,000         R_tmp[i] += dt*(ε+M1*R_tmp[i]/(E_tmp[i]+M2))*(...);
      }

```

Where is the time spent? [Provided Code]

- Loops are unfused

I1 cache: 32768 B, 64 B, 8-way associative
 D1 cache: 32768 B, 64 B, 8-way associative
 LL cache: 20971520 B, 64 B, 20-way associative
 Command: ./apf -n 256 -i 2000

Ir	I1mr	I1mr	Dr	D1mr	DLmr	Dw	D1mw	DLmw	
4.451B	2,639	2,043	1,381,173,237	50,592,465	7,051	3957M	16,794,937	26,115	PROGRAM TOTALS

Dr	D1mr	
1,380,464,019	50,566,007	solve.cpp:solve(...)
.	.	// Fills in the TOP Ghost Cells
10,000	1,999	for (i = 0; i < n+2; i++)
516,000	66,000	E_prev[i] = E_prev[i + (n+2)*2];
		// Fills in the RIGHT Ghost Cells
10,000	0	for i = (n+1); i < (m+2)*(n+2); i+=(m+2))
516,000	504,003	E_prev[i] = E_prev[i-2];
		// Solve for the excitation, a PDE
1,064,000	8,000	for(j =innerBlkRowStartIndx;j<=innerBlkRowEndIndx; j+=(m+)) {
0	0	E_prevj = E_prev + j; E_tmp = E + j;
512,000	0	for(i = 0; i < n; i++) {
721,408,002	16,630,001	E_tmp[i] = E_prevj[i]+alpha*(E_prevj[i+1]...)
		}
		// Solve the ODEs
4,000	4,000	for(j=innerBlkRowStartIndx; j <= innerBlkRowEndIndx;j+=(m+3)){
		for(i = 0; i <= n; i++) {
262,144,000	33,028,000	E_tmp[i] += -dt*(kk*E_tmp[i]*(E_tmp[i]-a)..)*R_tmp[i]);
393,216,000	4,000	R_tmp[i] += dt*(ε+M1*R_tmp[i]/(E_tmp[i]+M2))*(...);
		}

Fusing the loops

- Slows down the simulation by 20%
- # data references drops by 35%, total number of read misses drops by 48%
- What happened?

For a specified number of iterations, using supplied initial conditions repeat

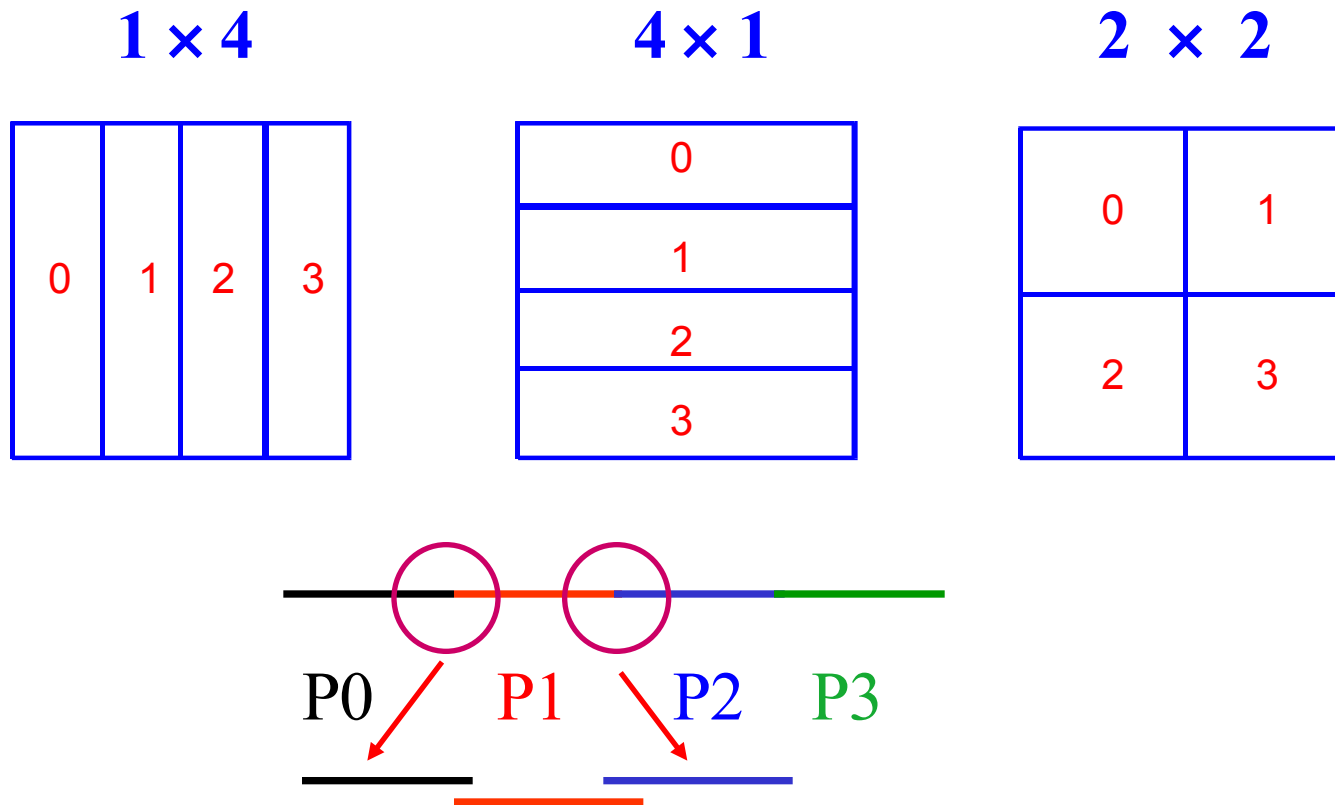
```
for (j=1; j < m+1; j++){  
  for (i=1; i < n+1; i++) {  
    // PDE SOLVER  
    E[j,i] = E_p[j,i]+ $\alpha$ *(E_p[j,i+1]+E_p[j,i-1]-4*E_p[j,i]+E_p[j+1,i]+E_p[j-1,i]);  
    // ODE SOLVER  
    E[j,i] += -dt*(kk*E[j,i]*(E[j,i]-a)*(E[j,i]-1)+E[j,i]*R[j,i]);  
    R[j,i] += dt*( $\epsilon$ +M1* R[j,i]/(E[j,i]+M2))*(-R[j,i]-kk*E[j,i]*(E[j,i]-b-1));  
  }  
}
```

swap E_p and E

End repeat

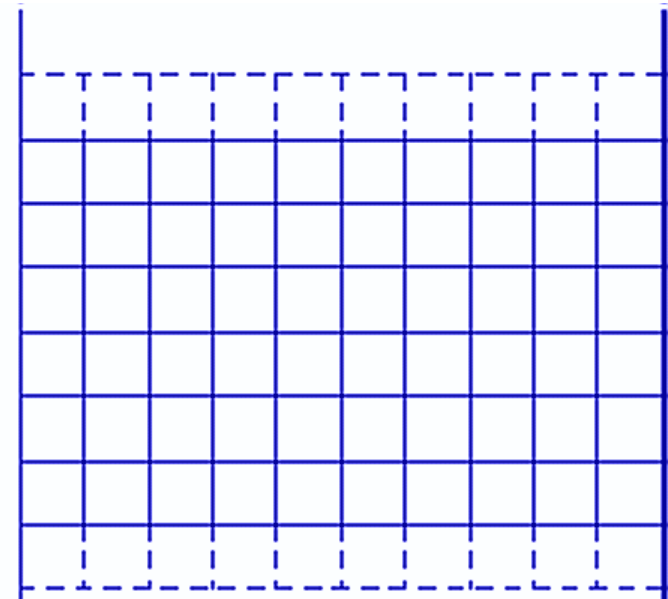
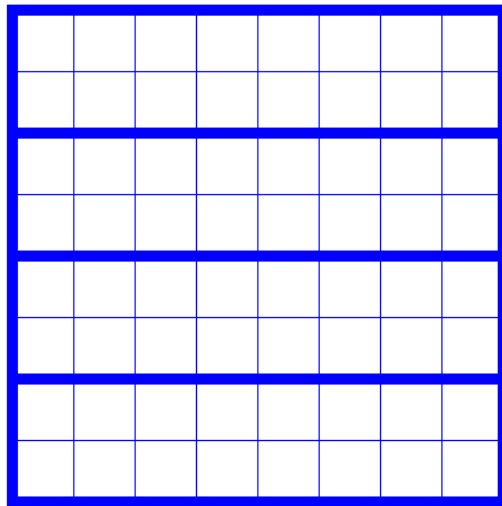
Parallel Implementation

- Partition data, assigning each partition to a unique thread
- Different partitionings according to the *processor geometry*
- Dependences on values found on neighboring threads



1D geometries

- Assumptions
 - ▶ P divides N evenly
 - ▶ $N/P > 2$
 - ▶ 1 word = double precision floating point = 8 bytes
- For horizontal strips, data are contiguous along the boundaries



Today's lecture

- OpenMP – some finer points on language law
- Stencil Methods
- Programming Assignment #4
- **Atomics**

C++ Atomic variables

- Atomic variables support *indivisible* operations that can be more efficient than mutexes
- Often more efficient than accessing variables through synchronized code, but requires care to avoid memory consistency errors

```
#include <atomic>
Class AtomicCounter {
    private: std::atomic<int> value;
    public:
        void increment(){
            ++value;
        }
        int get(){
            return value.load();
        }
};
```

What does indivisibility mean?

- A. The variable is incremented as if an indivisible operation
- B. The variable is incremented as if in a critical section
- C. The variable is incremented as if in a critical section followed by a barrier
- D. A&B
- E. A&C

```
std::atomic<int> value;  
value++;
```

The interface

- C++ provides atomic variants of the major built in types
 - Usual arithmetic operations
 - Assignment operators such as +=, ++ where appropriate
- Assignment is different
 - No copy or assignment constructors
 - Instead, there are load() and store() operations
 - We can assign or copy from or to a non-atomic type
`atomic<int> x=7;`
`int y = x;`
`atomic <int> z=x+2;`
- We will use the sequentially consistent variant (default)
`memory_order_seq_cst`

What can go wrong if the assignment returned a reference?

- A. Another thread could modify *x* before assigning its value
- B. The modification of *y* could affect *x*
- C. Both

```
atomic<int> x = 3;  
int y = x;  
y++;
```

Fin