

Lecture 4

Multithreading
OpenMP
Parallel Performance

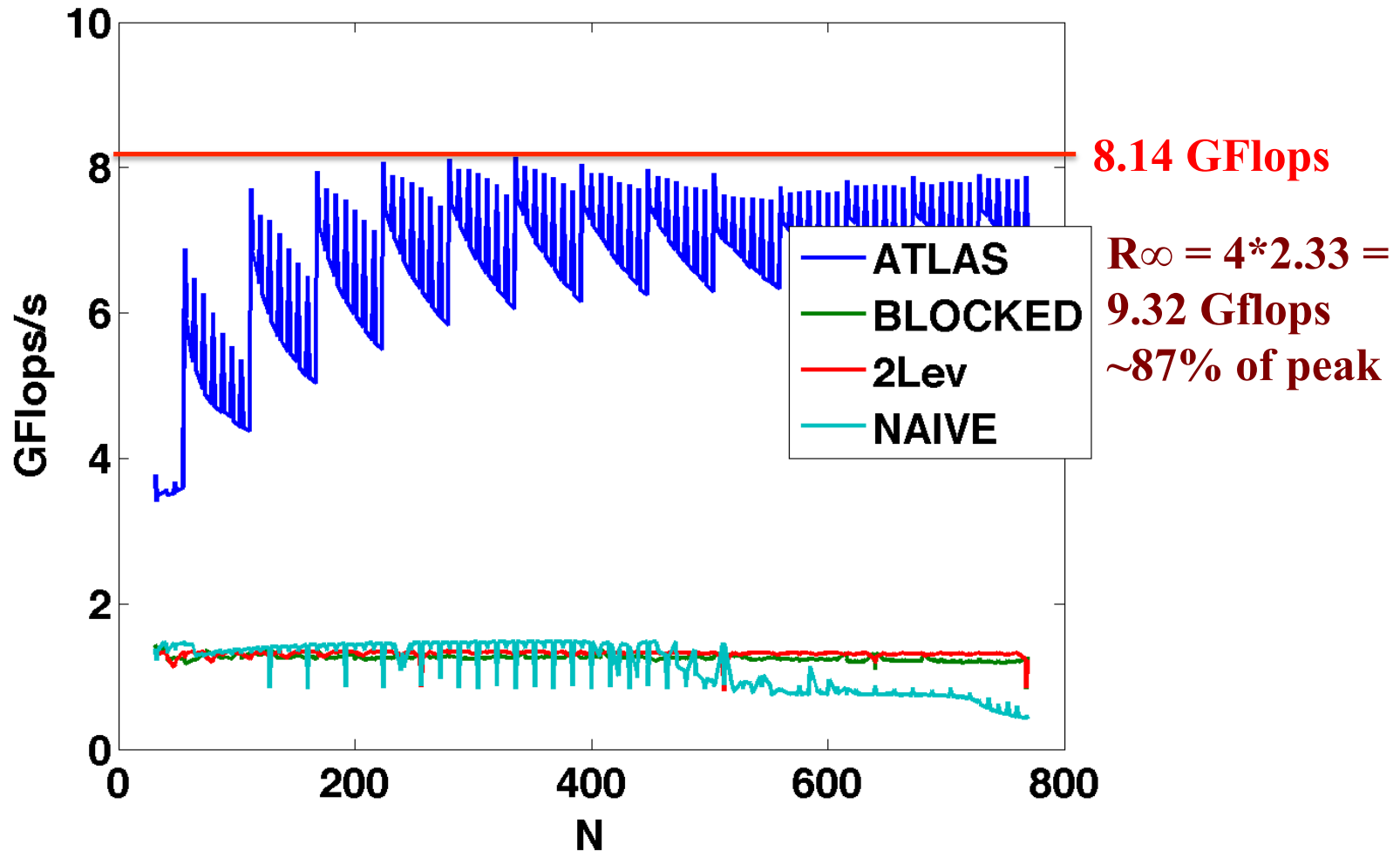
Announcements

Today's Lecture

- Multithreading
- OpenMP
- Parallel Performance Metrics

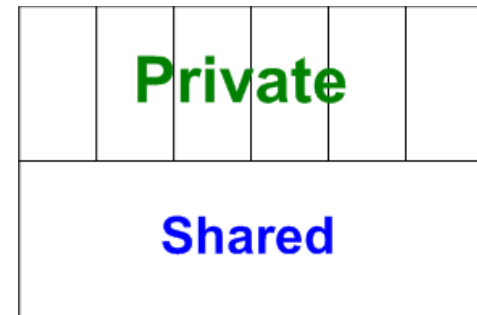
Assignment #1

- Blocking for cache will boost performance but a lot more is needed to approach ATLAS' performance



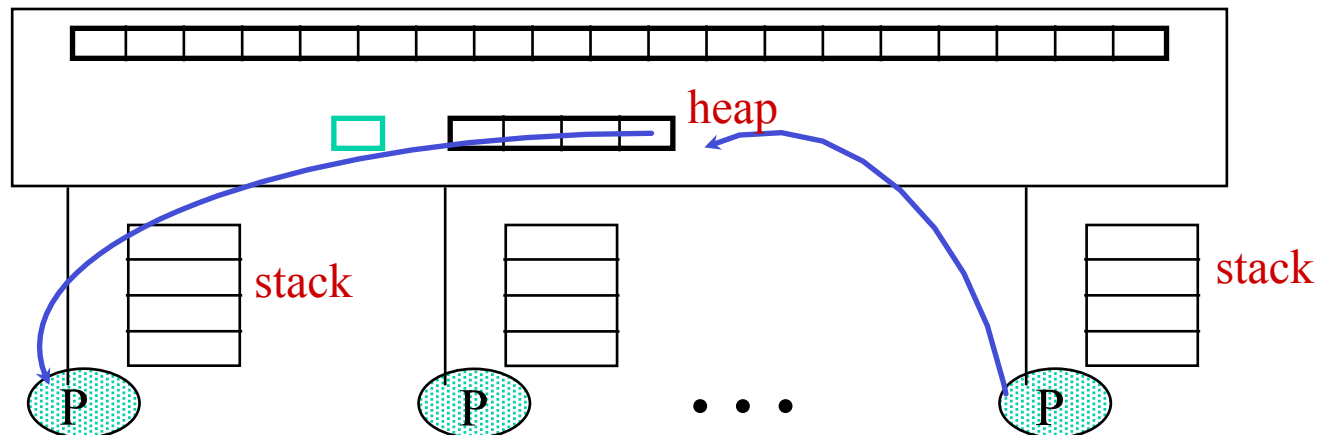
Threads programming model

- Extends traditional von Neumann execution model to run multiple instructions stream in parallel
 - Program executes a collection of independent instruction streams, called *threads*
 - 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)
 - Usually implemented under SPMD programming model
- Same Program Multiple Data**
- ♦ Each thread is initialized with the same code
 - ♦ Threads are distinguished by their rank
 - ♦ We will generally specify number of threads at program’s start



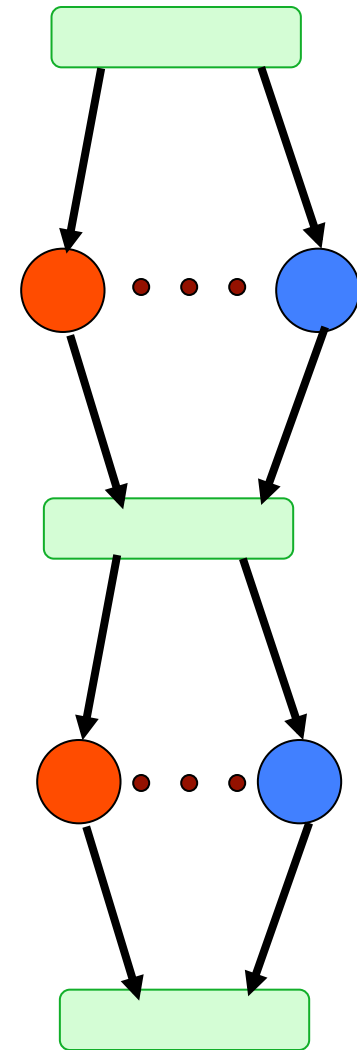
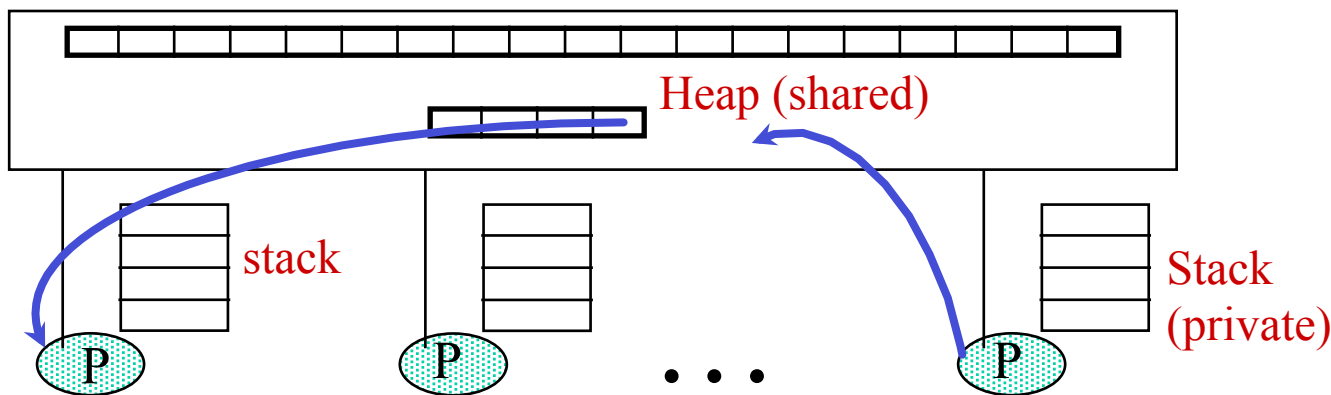
Why threads?

- 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



Threads Programming model

- Start with a single root thread
- Fork-join parallelism to create concurrently executing threads
- Threads communicate via shared memory
- A spawned thread executes asynchronously until it completes
- Threads may or may not execute on different processors



Multithreading in Practice

- C++11
- POSIX Threads “standard” (pthreads):
IEEE POSIX 1003.1c-1995
 - ♦ Low level interface
 - ♦ Beware of non-standard features
- Java threads not used in high performance computation
- OpenMP – program annotations
- Parallel programming languages
 - ♦ Co-array FORTRAN
 - ♦ UPC

C++11 Threads

- Via `<thread>`, C++ supports a threading interface similar to pthreads, though a bit more user friendly
- Async is a higher level interface suitable for certain kinds of applications
- New memory model
- Atomic template

Hello world with <Threads>

```
#include <thread>
void Hello(int TID) {
    cout << "Hello from thread " << TID << endl;
}

int main(int argc, char *argv[ ]){
    thread *thrds = new thread[NT];

    // Spawn threads
    for(int t=0;t<NT;t++){
        thrds[t] = thread(Hello, t );
    }

    // Join threads
    for(int t=0;t<NT;t++){
        thrds[t].join();
    }
}
```

```
$ ./hello_th 3
```

```
Hello from thread 0
```

```
Hello from thread 1
```

```
Hello from thread 2
```

```
$ ./hello_th 3
```

```
Hello from thread 1
```

```
Hello from thread 0
```

```
Hello from thread 2
```

```
$ ./hello_th 4
```

```
Running with 4 threads
```

```
Hello from thread 0
```

```
Hello from thread 3
```

```
Hello from thread Hello from  
thread 21
```

\$PUB/Examples/Threads/Hello-Th

PUB = /share/class/public/cse260-wi14

Today's Lecture

- Multithreading
- **OpenMP**
- Parallel Performance Metrics

OpenMP

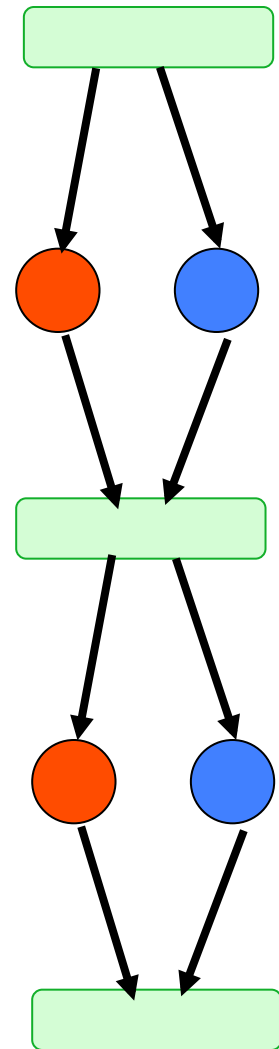
- A higher level interface for threads programming
- Parallelization handled via source code annotations
- See <http://www.openmp.org>
- Compare with explicit threads programming
- Set the number of threads via the environment
`export OMP_NUM_THREADS=4`

```
#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);
```

OpenMP's Fork-Join Model

- A program begins life as a single thread
- Parallel regions spawn work groups of multiple 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 at a *barrier* and are disbanded; they enter a wait state
 - Only the initial thread continues
- Thread teams can be created and disbanded many times during program execution, but this can be costly
- A clever compiler can avoid so many thread creations and joins



Fork join model with loops

Seung-Jai Min

```
cout << "Serial\n";  
N = 1000;
```

Serial

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

Parallel

```
M = 500;
```

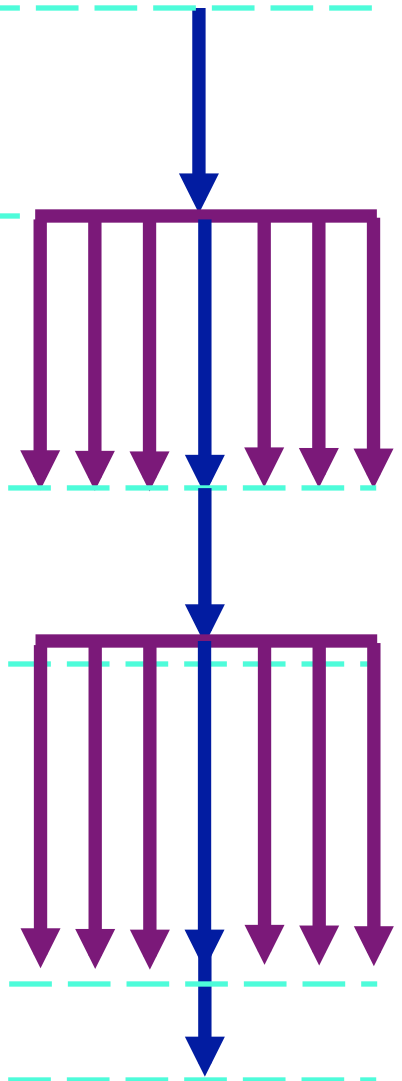
Serial

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

Parallel

```
Cout << "Finish\n";
```

Serial



Workload decomposition

- Translator automatically generates local loop bounds
- Also inserts any needed barriers
- We use private/shared pragmas to distinguish thread private from global variable
- Decomposition can be static or dynamic
- Dynamic assignment for irregular problems

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

Parallelizing a nested loop with OpenMP

- We parallelize the outer loop index, indicating shared and private (local) variables
- Some implementations can parallelize inner loops

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

0
1
2
3

- Generated code
 - ◆ An implicit barrier after the loop

```
mymin = 1 + ($TID * n/nprocs), mymax = mymin + n/nprocs -
for(i=mymin; i < mymax; i++)
  for(j=0; j < n; j++)
     $u^{new}[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1]) / 4$ 
  Barrier();
```

0	1	2	3
---	---	---	---

Variable scoping

- Any variables declared outside a parallel region are shared by all threads
- Variables declared inside the region are private
- Used **shared** and **private** declarations to override the defaults

```
double c = 1 / 6.0, h = 1.0, c2 = h * h;
double ***c = ...;
for (it= 0; it<nlters; it++) {
#pragma omp parallel shared(U,Un,b,nx,ny,nz,c2,c) private(i,j,k)
#pragma omp for
    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]);
    Swap U ↔ Un;
}
```

An application: Matrix Vector Multiplication

a_{00}	a_{01}	$\cdots$	$a_{0,n-1}$
a_{10}	a_{11}	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
a_{i0}	a_{i1}	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

x_0
x_1
$\vdots$
x_{n-1}

 $=$

y_0
y_1
$\vdots$
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
$\vdots$
y_{m-1}

Initialization

- Allocate and initialize storage outside a parallel region
- In some apps, we initialize in parallel

```
double **A;  
A = new double [N*N];  
assert(A);  
  
for ( i=0; i<N; i++ )  
    for ( j=0; j<N; j++ )  
        A[i*N+j] = 1.0 / (double) (i+j-1);
```

Computation

```
double *A, *x, *y;                                // GLOBAL
// Start timer
    double t0 = -getTime();

#pragma omp parallel shared(A,x,N)
    for (int k = 0; k<reps; k++)
#pragma omp for
        for (i=0; i<N; i++){
            y[i] = 0.0;
            for (j=0; j<N; j++)
                y[i] += A[i*n+j] * x[j];
        }

// Take time
t0 += getTime();
```

Reductions in OpenMP

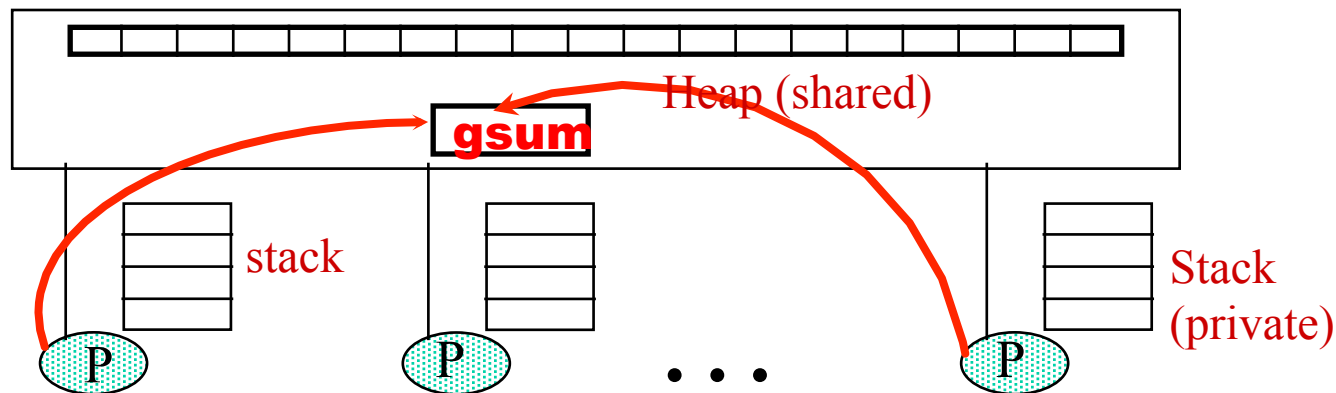
- What if we need to reduce a set of values down to a single value?
- We need to tell OpenMP to ensure atomicity when updating sum, to avoid the *race condition*

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

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

Race conditions

- The program usually runs correctly
Failing to employ synchronization in multithreaded code doesn't guarantee safety violations, it just allows them!
- But sometimes results are incorrect: *non-deterministic*
- There was a conflict when updating a shared quantity:
a data race
 - ◆ Arises when there is at least one writer on shared data
 - ◆ The timing of accesses to shared data can affect the outcome



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

P1
 $r1 \leftarrow x$

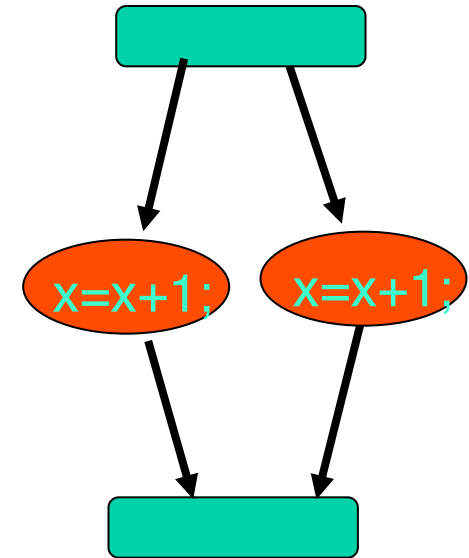
$r1 \leftarrow r1 + \#1$

$x \leftarrow r1$

P2
 $r1 \leftarrow x$

$r1 \leftarrow r1 + \#1$

$x \leftarrow r1$



$r1(P1)$ 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

Avoiding race conditions

- 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
 - ♦ Atomic variables
 - ♦ Critical sections
 - ♦ Barriers
- Very difficult to detect atomicity errors

Barriers in OpenMP

- Even if we avoid the race condition through atomicity, we still need the barrier
- Why?

```
#pragma omp parallel reduction(+:sum)
```

```
    for (int i=0; i< N-1; i++)
```

```
        gsum += f(x[i]);
```

```
Print gsum
```

```
i0 = $TID*n/$nthreads, i1 = i0 + n/$nthreads;
```

```
for (i=i0, localSum=0; i < i1; i++)
```

```
    localSum += f(x[i]);
```

```
gsum.atomicAdd(localSum);
```

```
Print gsum
```

Dealing with loop carried dependences

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

```
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 we saw in odd/even sorting
- OpenMP may warn you when it is doing something unsafe, but not always

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
```

In class exercises – for OpenMP

Questions

1. Iteration to thread mapping
2. Removing data dependencies
3. Dependence analysis
4. Tree Summation

1. Iteration to thread mapping

```
#pragma omp parallel shared(N, iters) private(i)
```

```
#pragma omp for
```

```
for (i = 0; i < N; i++)
```

```
    iters[i] = omp_get_thread_num();
```

N = 9, # of openMP threads = 3

0 0 0 1 1 1 2 2 2

N = 16, # of openMP threads = 4, schedule(static,2)

0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3

N=9: 0 0 1 1 2 2 0 0 1

N = 16, # of openMP threads = 4, schedule(dynamic,2)

3 3 0 0 1 1 2 2 3 3 3 3 3 3 3 3

2 2 3 3 0 0 1 1 2 2 2 2 2 2 2 2

In \$PUB/Examples/OpenMP/Assign

2. Removing data dependencies

- B initially: 0 1 2 3 4 5 6 7
- B on 1 thread: 7 7 7 7 11 12 13 14
- How can we split into 2 loops so that each loop parallelizes, the result is correct?

#pragma omp for shared (N,B)

for i = 0 to N-1

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

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

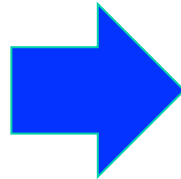
B[3] += B[4], B[4] += B[3], B[5] += B[2]

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

Splitting a loop

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



for $i = 0$ to $N/2-1$
 $B[i] += B[N-1-i];$
for $i = N/2+1$ to $N-1$
 $B[i] += B[N-1-i];$

In \$PUB/Examples/OpenMP/Assign
Compile with `omp=1` on “make” line

3. Loop Dependence Analysis

- Which loop(s) can we correctly parallelize with OpenMP?

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

2. for $i = 0$ to $N-2$
 $A[i+1] = A[i] + 1;$

3. for $i = 1$ to $N-1$ step 2
 $A[i] = A[i-1] + A[i];$

4. for $i = 0$ to $N-2$ {
 $A[i] = B[i];$
 $C[i] = A[i] + B[i];$
 $E[i] = C[i+1];$
}

Fin