

# Exercises 3: OpenMP Tutorial

Thanks to Larry Meadows, Mark Bull, Clay Breshears, Tim Mattson (Intel)

# Introduction

---

- This set of slides supports a collection of exercises to be used when learning OpenMP.
- scp the ex3.tar file downloaded from Moodle to your chosen directory on the cluster, and extract the files from the archive

# OpenMP Exercises



<b>Topic</b>	<b>Exercise</b>	<b>concepts</b>
<b>I. OMP Intro</b>	<b>hello_world</b>	<b>Parallel regions</b>
<b>II. Creating threads</b>	<b>Pi_spmc_simple</b>	<b>Parallel, default data environment, runtime library calls</b>
<b>III. Synchronization</b>	<b>Pi_spmc_final</b>	<b>False sharing, critical, atomic</b>
<b>IV. Parallel loops</b>	<b>Pi_loop, Matmul</b>	<b>For, schedule, reduction,</b>
<b>V. ThreadPrivate</b>	<b>Monte Carlo pi</b>	<b>Thread safe libraries</b>
<b>VI. Data Environment</b>	<b>Mandelbrot set area</b>	<b>Data environment details, software optimization</b>

# Compiler notes: Linux and OSX

---

- Linux and OS X with gcc:
  - > gcc -fopenmp foo.c
  - > export OMP\_NUM\_THREADS=4

# OpenMP constructs used in these exercises

---

- `#pragma omp parallel`
- `#pragma omp for`
- `#pragma omp critical`
- `#pragma omp atomic`
  
- Data environment clauses
  - `private (variable_list)`
  - `firstprivate (variable_list)`
  - `lastprivate (variable_list)`
  - `reduction(+:variable_list)`

# Exercise 1: Hello world

## Verify that your OpenMP environment works

- Copy the file `hello.c` to file `hello_par.c`
- Modify `hello_par.c` to do the following:
  - Write a multithreaded program that prints “hello world” with thread numbers.
    - e.g., "Hello World from Thread 0"
- Remember you can change number of threads by, e.g.,  
`export OMP_NUM_THREADS=16`
- You can get the thread number with `omp_get_thread_num()`
- Compile via

```
gcc -fopenmp -o hello_par hello_par.c
```


# Solution

# Exercise 1: Hello world

```
#include <stdio.h>
#include <omp.h>
int main(){
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("Hello World from thread %d\n", ID);
    }
    return 0;
}
```

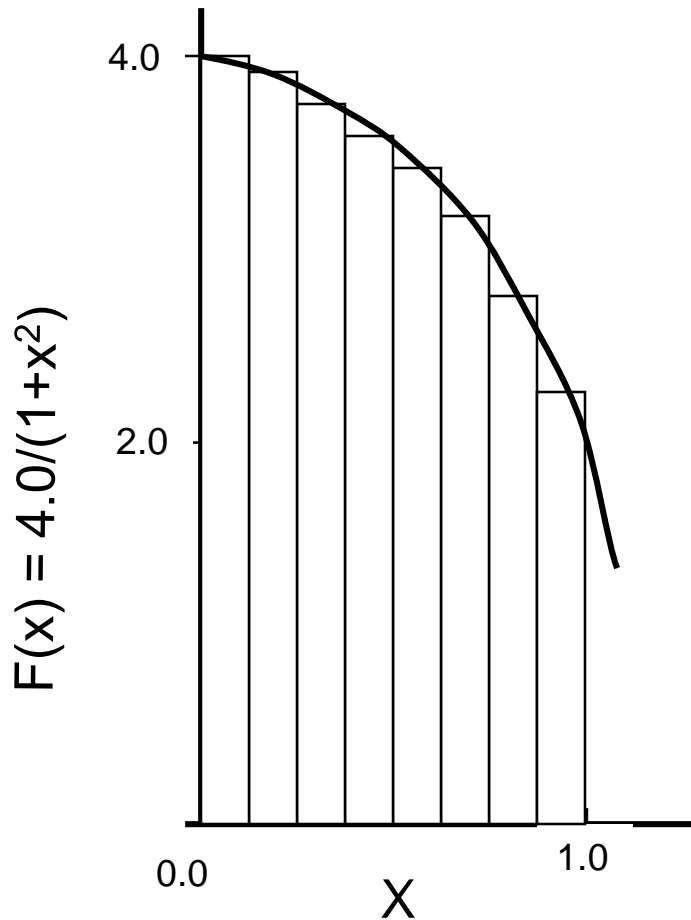
```
#include <stdio.h>
#include <omp.h>
int main(){
    #pragma omp parallel
    printf("Hello World from thread %d\n", omp_get_thread_num());
    return 0;
}
```

# OpenMP Exercises



<b>Topic</b>	<b>Exercise</b>	<b>concepts</b>
<b>I. OMP Intro</b>	hello_world	Parallel regions
<b>II. Creating threads</b>	Pi_spmc_simple	Parallel, default data environment, runtime library calls
<b>III. Synchronization</b>	Pi_spmc_final	False sharing, critical, atomic
<b>IV. Parallel loops</b>	Pi_loop, Matmul	For, schedule, reduction,
<b>V. ThreadPrivate</b>	Monte Carlo pi	Thread safe libraries
<b>VI. Data Environment</b>	Mandelbrot set area	Data environment details, software optimization

# Exercises 2 to 4: Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Where each rectangle has width  $\Delta x$  and height  $F(x_i)$  at the middle of interval  $i$ .

# Exercises 2 to 4: Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{
    int i; double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for ( i=0; i< num_steps; i++)
    {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

See the file pi.c

# The SPMD pattern

---

- The most common approach for parallel algorithms is the SPMD or Single Program Multiple Data pattern.
- Each thread runs the same program (Single Program), but using the thread ID, they operate on different data (Multiple Data) or take slightly different paths through the code.
- In OpenMP this means:
  - A parallel region “near the top of the code”.
  - Get thread ID and num\_threads.
  - Use them to split up loops and select different blocks of data to work on.

# Exercise 2

- We look at a parallel version of the pi program that uses a parallel construct in a file called `pi_spmd_simple.c`
- Pay close attention to shared versus private variables.
- In addition to a parallel construct, the program uses the runtime library routines

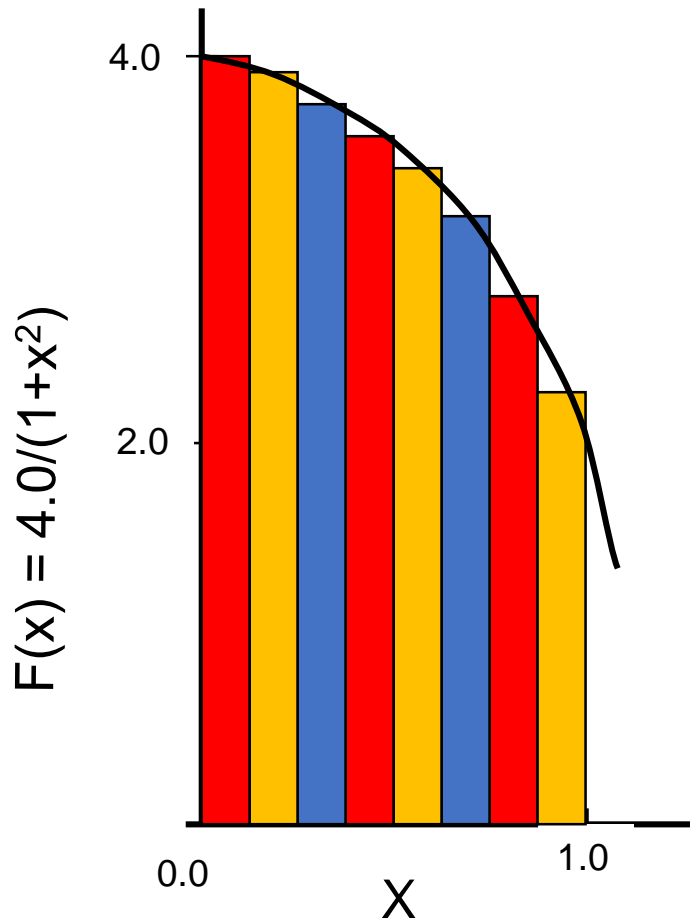
- `int omp_get_num_threads();`
- `int omp_get_thread_num();`
- `double omp_get_wtime();`
- `omp_set_num_threads(int);`

Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

# SPMD Example



- We will first only use the "`#pragma omp parallel`" construct
  - (pretend we don't know about "`parallel for`" construct yet)
- We will first manually define the number of threads we use, e.g.,

```
#define NUM_THREADS=2
```

(not good practice in general)
- We will manually split up the loop based on how many threads there are
  - One way is to do a cyclic distribution of the steps

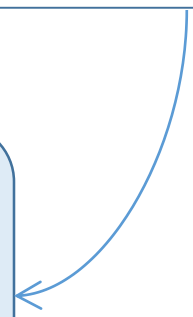
# Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{
    int i; double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for ( i=0; i< num_steps; i++)
    {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

We want to parallelize this loop, so that num\_steps is distributed amongst threads in a cyclic distribution



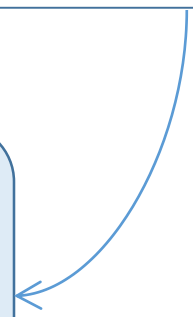
# Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{
    int i; double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for ( i=0; i< num_steps; i++)
    {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

We want to parallelize this loop, so that num\_steps is distributed amongst threads in a cyclic distribution



Can you see any potential race conditions?  
(if every thread is using the same "x" and "sum", what could happen?)

# Serial PI Program

```
for ( i=0; i< num_steps; i++)  
{  
    x = (i+0.5)*step;  
    sum = sum + 4.0/(1.0+x*x);  
}  
pi = step * sum;
```

x= 0.0  
sum = 0.0

Thread 1: (i = 0, 2)

Thread 2: (i = 1, 3)

# Serial PI Program

```
for ( i=0; i< num_steps; i++)  
{  
    x = (i+0.5)*step;  
    sum = sum + 4.0/(1.0+x*x);  
}  
pi = step * sum;
```

**x= .125**  
sum = 0.0

Thread 1: (i = 0, 2)  
 $x = (0+0.5) \cdot (.25) = .125$

Thread 2: (i = 1, 3)

# Serial PI Program

```
for ( i=0; i< num_steps; i++)  
{  
    x = (i+0.5)*step;  
    sum = sum + 4.0/(1.0+x*x);  
}  
pi = step * sum;
```

$x = .375$   
 $\text{sum} = 0.0$

Thread 1: (i = 0, 2)

$x = (0+0.5) \cdot (.25) = .125$

Thread 2: (i = 1, 3)

$x = (1+0.5) \cdot (.25) = .375$

# Serial PI Program

```
for ( i=0; i< num_steps; i++)  
{  
    x = (i+0.5)*step;  
    sum = sum + 4.0/(1.0+x*x);  
}  
pi = step * sum;
```

x= .375  
sum = 0.0

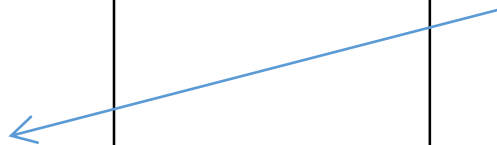
Thread 1: (i = 0, 2)

**x = (0+.5)\*(.25)=.125**

sum+4.0/(1.0+x\*x)  
= 0 + 4.0/(1.0+.375\*.375)=3.507

Thread 2: (i = 1, 3)

x = (1+.5)(.25) = .375



# Serial PI Program

```
for ( i=0; i< num_steps; i++)  
{  
    x = (i+0.5)*step;  
    sum = sum + 4.0/(1.0+x*x);  
}  
pi = step * sum;
```

x= .375  
sum = 0.0

Thread 1: (i = 0, 2)  
 $x = (0+0.5) \cdot (.25) = .125$

$\text{sum} + 4.0 / (1.0 + x \cdot x)$   
 $= 0 + 4.0 / (1.0 + .125 \cdot .125) = 3.507$

Thread 2: (i = 1, 3)

$x = (1+0.5) \cdot (.25) = .375$

$\text{sum} + 4.0 / (1.0 + x \cdot x)$   
 $= 0 + 4.0 / (1.0 + .375 \cdot .375) = 3.507$

# Serial PI Program

```
for ( i=0; i< num_steps; i++)  
{  
    x = (i+0.5)*step;  
    sum = sum + 4.0/(1.0+x*x);  
}  
pi = step * sum;
```

x= .375

sum = 3.507

Thread 1: (i = 0, 2)

$x = (0+0.5) \cdot (.25) = .125$

$\text{sum} + 4.0 / (1.0 + x \cdot x)$   
 $= 0 + 4.0 / (1.0 + .375 \cdot .375) = 3.507$

sum = 3.507

Thread 2: (i = 1, 3)

$x = (1+0.5) \cdot (.25) = .375$

$\text{sum} + 4.0 / (1.0 + x \cdot x)$   
 $= 0 + 4.0 / (1.0 + .375 \cdot .375) = 3.507$

# Serial PI Program

```
for ( i=0; i< num_steps; i++)  
{  
    x = (i+0.5)*step;  
    sum = sum + 4.0/(1.0+x*x);  
}  
pi = step * sum;
```

x= .375  
sum = 3.507

Thread 1: (i = 0, 2)  
 $x = (0+0.5) \cdot (.25) = .125$

$\text{sum} + 4.0 / (1.0 + x \cdot x)$   
 $= 0 + 4.0 / (1.0 + .375 \cdot .375) = 3.507$

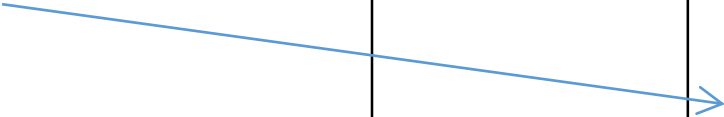
sum = 3.507

Thread 2: (i = 1, 3)

$x = (1+0.5) \cdot (.25) = .375$

$\text{sum} + 4.0 / (1.0 + x \cdot x)$   
 $= 0 + 4.0 / (1.0 + .375 \cdot .375) = 3.507$

sum = 3.507



# Exercise 2: A simple SPMD pi program

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
    int nthreads; double pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum[id]=0.0; i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(int j=0, pi=0.0; j<nthreads; j++){
        pi += sum[j] * step;
    }
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.

x is local to each thread

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations


# Exercise 2: A simple SPMD pi program

---

- Compile and run this program a number of times
- What do you observe?

```
carson@r3d3:[~/HPC_W22/ex3] ./pi_spmd_simple
Number of threads: 2
pi with 100000000 steps is 3.141593 in 1.227657 seconds
carson@r3d3:[~/HPC_W22/ex3] ./pi_spmd_simple
Number of threads: 2
pi with 100000000 steps is 3.141593 in 1.384369 seconds
carson@r3d3:[~/HPC_W22/ex3] ./pi_spmd_simple
Number of threads: 2
pi with 100000000 steps is 3.141593 in 0.660179 seconds
carson@r3d3:[~/HPC_W22/ex3] ./pi_spmd_simple
Number of threads: 2
pi with 100000000 steps is 3.141593 in 1.242808 seconds
```

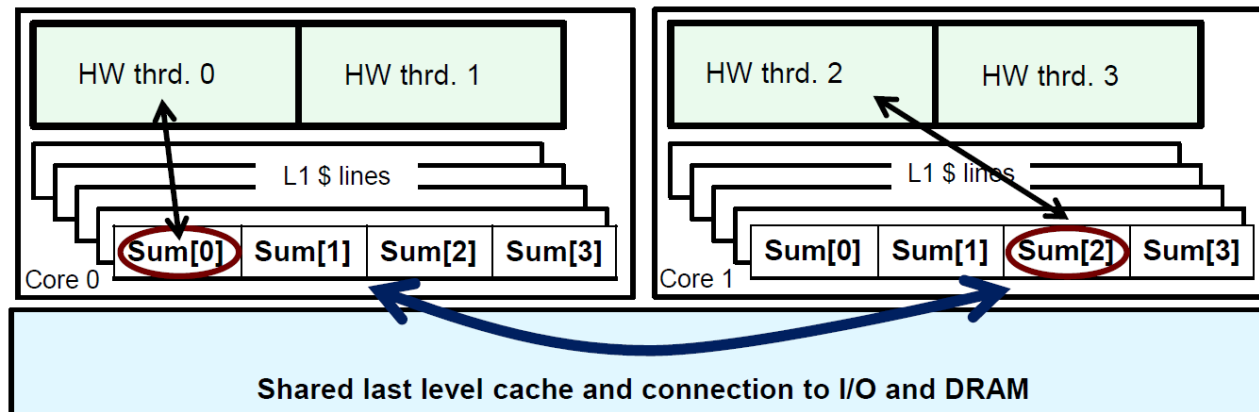
# OpenMP Exercises



<b>Topic</b>	<b>Exercise</b>	<b>concepts</b>
<b>I. OMP Intro</b>	hello_world	Parallel regions
<b>II. Creating threads</b>	Pi_spmc_simple	Parallel, default data environment, runtime library calls
<b>III. Synchronization</b>	Pi_spmc_final	False sharing, critical, atomic
<b>IV. Parallel loops</b>	Pi_loop, Matmul	For, schedule, reduction,
<b>V. ThreadPrivate</b>	Monte Carlo pi	Thread safe libraries
<b>VI. Data Environment</b>	Mandelbrot set area	Data environment details, software optimization

# False sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads.
  - This is called **“false sharing”**.
- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines.
  - Result ... **poor scalability**



- Solution:
  - When updates to an item are frequent, work with local copies of data instead of an array indexed by the thread ID.
  - Or pad arrays so elements you use are on distinct cache lines.

# Example: Array Padding

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
#define PAD 8
void main ()
{
    int nthreads; double pi, sum[NUM_THREADS][PAD];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum[id][0]=0.0; i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id][0] += 4.0/(1.0+x*x);
        }
    }
    for(int j=0, pi=0.0; j<nthreads; j++){
        pi += sum[j][0] * step;
    }
}
```

Assumes 64 byte  
cache line size  
(x86/x64  
architectures)  
-> Need to know  
cache line size!

...must be a  
better way

# Synchronization Constructs

---

- Most common:
  - Critical
  - Atomic
  - Barrier

# Synchronization: Barrier

- Barrier: Each thread waits until all threads arrive.

```
#pragma omp parallel
{
    int id=omp_get_thread_num();
    A[id] = big_calc1(id);

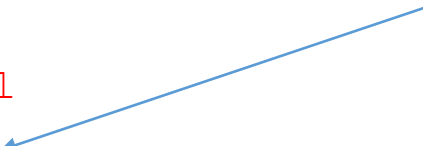
    #pragma omp barrier
    B[id] = big_calc2(id, A);
}
```

# Synchronization: Critical

- Mutual exclusion: Only one thread at a time can enter a critical region.

```
float res;  
#pragma omp parallel  
{  
    float B; int i, id, nthrds;  
    id = omp_get_thread_num();  
    nthrds = omp_get_num_threads();  
    for(i=id; i<niters; i+=nthrds)  
    {  
        B = big_job();  
        #pragma omp critical  
        res += consume(B);  
    }  
}
```

Threads wait their  
turn – only one at  
a time calls  
consume()



# Synchronization: Atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel
{
    double tmp, B, X;
    B = do_it();
    tmp = something(B);

    #pragma omp atomic
    X += tmp;
}
```

The statements that  
can go inside the  
atomic are limited,  
e.g.,  
X++  
X- -  
X+=

See OpenMP  
documentation for  
complete list

# Exercise 3

- Copy `pi_spmc_simple.c` to a new file called `pi_spmc_final.c`
- Now, parallelize the code in `pi_spmc_final.c`, the file to avoid false sharing due to the sum array.
  - `sum` will now be a variable local to each thread rather than a global array
  - Use synchronization to accumulate the local sums into shared `pi` variable
- We should no longer need to hardcode `NUM_THREADS` (which we needed to set the size of the sum array before).

# Exercise 3: SPMD Pi without false sharing

```
#include <omp.h>
static long num_steps = 100000; double step;
void main ()
{
    double pi; step = 1.0/(double) num_steps;

    #pragma omp parallel
    {
        int i, id, nthrds; double x, sum;
        sum = 0.0;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        for (i=id; i< num_steps; i=i+nthrds){
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
        pi += sum * step;
    }
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes “out of scope” beyond the parallel region ... so you must sum it in here. Must protect summation into pi in a critical region so updates don’t conflict


# Be careful where you put a critical section!

```
#include <omp.h>
static long num_steps = 100000; double step;
void main ()
{
    double pi; step = 1.0/(double) num_steps;

    #pragma omp parallel
    {
        int i, id, nthrds; double x, sum;
        sum = 0.0;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        for (i=id; i< num_steps; i=i+nthrds){
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);

            #pragma omp critical
            pi += sum * step;

        }
    }
}
```



What would happen if you put the critical section inside the loop?

# Critical vs. Atomic


---

- An OpenMP critical section is completely general - it can surround any arbitrary block of code
  - The cost is that there is a significant overhead every time a thread enters and exits the critical section
    - Requires OS calls
- Atomic operations have much lower overhead
  - Can take advantage of atomic hardware instructions!

# Exercise 3: SPMD Pi without false sharing - Improved

```
#include <omp.h>
static long num_steps = 100000; double step;
void main ()
{
    double pi; step = 1.0/(double) num_steps;

    #pragma omp parallel
    {
        int i, id, nthrds; double x, sum;
        sum = 0.0;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        for (i=id; i< num_steps; i=i+nthrds){
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
        sum = sum * step;
        #pragma omp atomic
        pi += sum;
    }
}
```



Change the update to just use a += operation.  
Now we can use atomic!

# OpenMP Exercises

<b>Topic</b>	<b>Exercise</b>	<b>concepts</b>
<b>I. OMP Intro</b>	hello_world	Parallel regions
<b>II. Creating threads</b>	Pi_spmc_simple	Parallel, default data environment, runtime library calls
<b>III. Synchronization</b>	Pi_spmc_final	False sharing, critical, atomic
<b>IV. Parallel loops</b>	Pi_loop, Matmul	For, schedule, reduction,
<b>V. ThreadPrivate</b>	Monte Carlo pi	Thread safe libraries
<b>VI. Data Environment</b>	Mandelbrot set area	Data environment details, software optimization



# Exercise 4: Pi with loops

- Our solution is `pi_spmf_final.c` works well, but there is an even simpler way...
- Try it yourself:
  - Copy the serial `pi.c` program to a file called `pi_loop.c`
  - Edit this file to parallelize it with a **loop construct**
  - Your goal is to minimize the number of changes made to the serial program.
    - Look at your OpenMP cheat sheets

# Solution

# Exercise 4: solution

```
#include <omp.h>
static long num_steps = 100000; double step;
void main()
{
    int i;
    double pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double x;
        #pragma omp for reduction(+:sum)
        for (i=0; i< num_steps; i++) {
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    pi = step * sum;
}
```


# Exercise 4: better solution

Using data environment clauses so parallelization only requires changes to the pragma

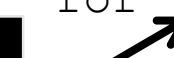
```
#include <omp.h>
static long num_steps = 100000; double step;

int main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=0; i< num_steps; i++) {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

For good OpenMP implementations, reduction is more scalable than critical.



i private by default



Note: we created a parallel program without changing any code and by adding 1 simple lines of text!

# Exercise 4b: Optimizing loops

---

- Copy the sequential file `matmul.c` into `matmul_par.c`
- Parallelize the matrix multiplication program by adding a single OpenMP pragma

# Solution

# Matrix multiplication

```
#pragma omp parallel for private(tmp, j, k)
for (i=0; i<Ndim; i++){
    for (j=0; j<Mdim; j++){
        tmp = 0.0;
        for (k=0; k<Pdim; k++) {
            /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
            tmp += *(A+(i*Ndim+k)) * *(B+(k*Pdim+j));
        }
        *(C+(i*Ndim+j)) = tmp;
    }
}
```

You must declare tmp, j, and k private! (i is private by default). Otherwise you will see errors in your matrix multiplication!

# Matrix Multiplication

---

- Is it better to parallelize the outermost loop (over  $i$ )? Or one of the other loops?

# OpenMP Exercises

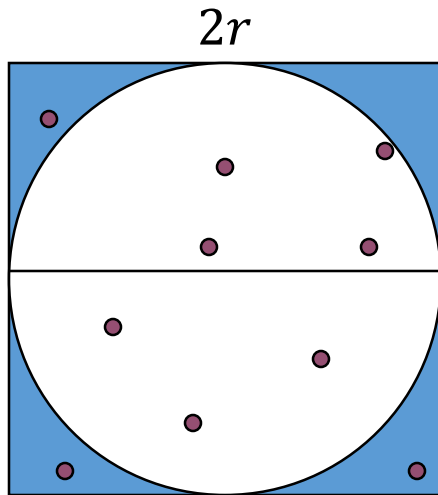
<b>Topic</b>	<b>Exercise</b>	<b>concepts</b>
<b>I. OMP Intro</b>	hello_world	Parallel regions
<b>II. Creating threads</b>	Pi_spmc_simple	Parallel, default data environment, runtime library calls
<b>III. Synchronization</b>	Pi_spmc_final	False sharing, critical, atomic
<b>IV. Parallel loops</b>	Pi_loop, Matmul	For, schedule, reduction,
<b>V. ThreadPrivate</b>	Monte Carlo pi	Thread safe libraries
<b>VI. Data Environment</b>	Mandelbrot set area	Data environment details, software optimization



# Exercise 5: Monte Carlo Calculations

## Using Random numbers to solve tough problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing  $\pi$  with a digital dart board:



$N = 10$	$\pi = 2.8$
$N = 100$	$\pi = 3.16$
$N = 1000$	$\pi = 3.148$

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:
$$A_c = r^2\pi$$
$$A_s = (2r)^2 = 4r^2$$
$$P = \frac{A_c}{A_s} = \frac{\pi}{4}$$
- Compute  $\pi$  by randomly choosing points, count the fraction that falls in the circle, compute pi.

# Exercise 5

- There are three files for this exercise
  - `pi_mc.c`: the monte carlo method pi program
  - `random.c`: a simple random number generator
  - `random.h`: include file for random number generator
- Create a parallel version of the `pi_mc.c` program called `pi_mc_par.c`

compiling:

```
gcc -fopenmp -c pi_mc_par.c
```

```
gcc -fopenmp -c random.c
```

```
gcc -fopenmp -lm -o pi_mc_par random.o pi_mc_par.o
```

# Solution

# Parallel Programmers love Monte Carlo algorithms

```
#include <omp.h>
static long num_trials = 10000;
int main ()
{
    long i; long Ncirc = 0; double pi, x, y;
    double r = 1.0;    // radius of circle. Side of square is 2*r
    seed(-r, r);    // The circle and square are centered at the origin
    #pragma omp parallel for private (x, y) reduction (+:Ncirc)
    for(i=0;i<num_trials; i++)
    {
        x = random(); y = random();
        if ( x*x + y*y) <= r*r)    Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc/((double)num_trials);
    printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```

*Embarrassingly parallel:* the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

# OpenMP Exercises

<b>Topic</b>	<b>Exercise</b>	<b>concepts</b>
<b>I. OMP Intro</b>	hello_world	Parallel regions
<b>II. Creating threads</b>	Pi_spmc_simple	Parallel, default data environment, runtime library calls
<b>III. Synchronization</b>	Pi_spmc_final	False sharing, critical, atomic
<b>IV. Parallel loops</b>	Pi_loop, Matmul	For, schedule, reduction,
<b>V. ThreadPrivate</b>	Monte Carlo pi	Thread safe libraries
<b>VI. Data Environment</b>	Mandelbrot set area	Data environment details, software optimization



# Exercise 6: Mandelbrot set area

- The supplied program (mandel.c) computes the area of a Mandelbrot set.
  - See, e.g., <http://mathworld.wolfram.com/MandelbrotSet.html>
- The program has been parallelized with OpenMP, but we were lazy and didn't do it right.
  - **Try to run the program - get a different incorrect answer each time ... there is a race condition!!!!**
- Find and fix the errors

# Solution

# Area of a Mandelbrot set

- A solution is in the file `sols/mandel_par.c`
- Errors you could have found:
  - `eps` is private but uninitialized. Two solutions:
    - It's read-only so you can make it shared.
    - Make it `firstprivate`
  - The loop index variable `j` is shared by default. Make it private.
  - Updates to `"numoutside"` are a race. Protect with an atomic.

# Debugging parallel programs

- Find tools that work with your environment and learn to use them. A good parallel debugger can make a huge difference.
- But parallel debuggers are not portable and you will assuredly need to debug “by hand” at some point.
- There are tricks to help you. The most important is to use the `default(none)` pragma

```
#pragma omp parallel for default(none) private(c, eps)
  for (i=0; i<NPOINTS; i++) {
    for (j=0; j<NPOINTS; j++) {
      c.r = -2.0+2.5*(double) (i) / (double) (NPOINTS)+eps;
      c.i = 1.125*(double) (j) / (double) (NPOINTS)+eps;
      testpoint();
    }
  }
}
```

Using `default(none)` generates a compiler error that `j` is unspecified.