Exercise 3a - Make it parallel, and a schedule

- add ‘adaptive integration’: where needed, the program refines the step size. This means that the iterations no longer take a predictable amount of time.

```
for (i=0; i<nsteps; i++) {
      double x = i*h,x2 = (i+1)*h,

      y = sqrt(1-x*x),y2 = sqrt(1-x2*x2),

      slope = (y-y2)/h;

      if (slope>15) slope = 15;

      int samples = 1+(int)slope,

      is;
```

```
      for (is=0; is<samples; is++)
      {
            double hs = h/samples,

            xs = x+ is*hs,

            ys = sqrt(1-xs*xs);

            quarterpi += hs*ys;

            nsamples++;
      }
}

pi = 4*quarterpi;
```

Exercise 3a - Make it parallel, and add the schedule clause. Findings.

- Use the omp parallel for construct to parallelize the loop. As in the previous lab, you may at first see an incorrect result. Use the reduction clause to fix this.

- Your code should now see a decent speedup, using up to 8 cores. However, it is possible to get completely linear speedup. For this you need to adjust the schedule.

- Start by using schedule(static,n). Experiment with values for n. When can you get a better speedup? Explain this.

- Since this code is somewhat dynamic, try schedule(dynamic). This will actually give a fairly bad result. Why? Use schedule(dynamic,n) instead, and experiment with values for n.

- Finally, use schedule(guided), where OpenMP uses a heuristic. What results does that give?

Observations?

- With regarding shared memory:
    - What have you seen?
    - What have you done to work through these issues?
    - What are some red flags? Are there any?

Controlling Thread Data, private/shared revisited

- Shared memory makes life easy for the programmer.
- no explicit data traffic between the processor is needed.
- But multiple processes/processors can also write to the same variable
- This is a source of potential problems.

Controlling Thread Data, private/shared revisited

```
Processor 1

            I = I + 2
```

```
Processor 2

            I = I + 3
```

Controlling Thread Data, private/shared revisited

scenario 1.		scenario 2.		scenario 3.	
$I = 0$					
read $I = 0$	read $I = 0$	read $I = 0$	read $I = 0$	read $I = 0$	
compute $I = 2$	compute $I = 3$	compute $I = 2$	compute $I = 3$	compute $I = 2$	
write $I = 2$			write $I = 3$	write $I = 2$	
	write $I = 3$	write $I = 2$			read $I = 2$
					compute $I = 5$
					write $I = 5$
$I = 3$		$I = 2$		$I = 5$	

Controlling Thread Data, shared

Any data declared outside a parallel region will be shared.

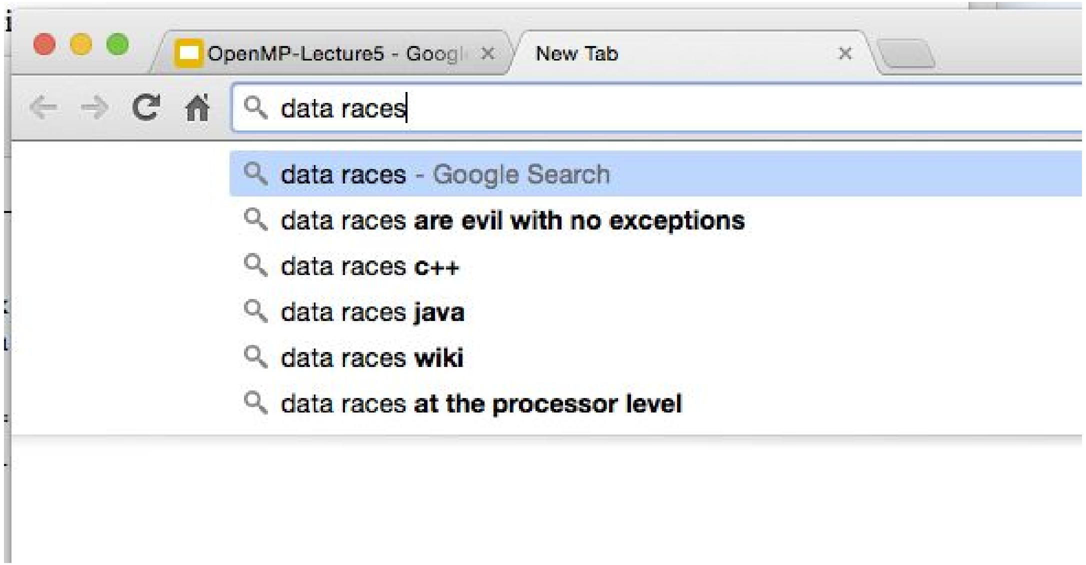
```
int x = 5;
#pragma omp parallel
{
   x = x+1;
   printf("shared: x is %d\n",x);
}

printf("final: x is %d\n", x);
```

Any thread using variable *x* will access the same memory location associated with that variable.

Controlling Thread Data, Gotcha's

Data Races



Controlling Thread Data, Gotchas

Data Races - Defined

A data race occurs when:

- two or more threads in a **single process** access the same memory location concurrently, and
- at least one of the accesses is for writing, and
- the threads are not using any exclusive locks to control their accesses to that memory.

When these three conditions hold, the order of accesses is non-deterministic, and the computation may give different results from run to run depending on that order. Some data-races may be benign (for example, when the memory access is used for a busy-wait), but many data-races are bugs in the program.

Controlling Thread Data, Gotchas

Data Races - Work Arounds

- declare updates of a shared variable in a *critical section*.
  - the instructions in the *critical section* (‘read sum from memory, update it, write back to memory’) can be executed by only one thread at a time.
- set a temporary lock on certain memory areas.
  - one process entering a critical section would prevent any other process from writing to the data

Controlling Thread Data, private

Any variable declared in a block following an OpenMP directive will be local to the executing thread.

```
int x = 5;
#pragma omp parallel
{
   int x; x = 3;
   printf("local x: x is %d\n",x);
}

printf("original: x is %d\n", x);
```

Note: Fortran does not have this level of scope

Controlling Thread Data, private

The private directive declares data to have a separate copy in the memory of each thread.

```
int x = 5;
#pragma omp parallel private(x)
{
   //dangerous
   x = x + 1;
   printf("private x: x is %d\n",x);
}
//also dangerous
printf("final: x is %d\n", x);
```

Note: Any computed value goes away at the end of the parallel region. But, run the example above, and see what happens. **Do not rely on any initial value, or on the value of the outer variable after the region.**

Controlling Thread Data, Setting Defaults

most data in a parallel section is shared. This default behaviour can be controlled by adding a default clause:

```
#pragma omp parallel default(shared) private(x)

        { ... }

#pragma omp parallel default(private) shared(matrix)

        { ... }
```

play it safe:

```
#pragma omp parallel default(none) private(x) shared(matrix)

        { ... }
```

Note: Setting `default(none)` is useful for debugging. If your code behaves differently in parallel vs sequential there is probably a data race. Specifying the status of every variable is a good way to debug this.

Controlling Thread Data, firstprivate

creates a variable that will act as if it's private inside the parallel region but is initialized outside the parallel region

```
int t=2;
#pragma omp parallel firstprivate(t)
{
   t += omp_get_thread_num();
   printf("Thread number: %d     t:%d\n",omp_get_thread_num(),t);
}
```

The variable t behaves like a private variable, except that it is initialized to the outside value.

Controlling Thread Data, lastprivate

preserves a private variable from the last iteration of a parallel region.

```
int x = 42;
#pragma omp parallel for private(x)
for(i=0;i<=10;i++)
{
   x=i;
   printf("Thread number: %d     x: %d\n",omp_get_thread_num(),x);
}

printf("x is %d\n", x);
```

```
int x = 42;
#pragma omp parallel for lastprivate(x)
for(i=0;i<=10;i++)
{
   x=i;
   printf("Thread number: %d     x: %d\n",omp_get_thread_num(),x);
}

printf("x is %d\n", x);
```

Compare the output of the above two programs. In the second example, the final value of *x* is preserved.

Controlling Thread Data, Persistency

Most data in OpenMP parallel regions is either inherited from the master thread and shared, or temporary within the scope of the region and fully private.

Controlling Thread Data, Persistency

But, what about keeping data persistency across a thread?
The threadprivate pragma is used to declare that each thread is to have a private copy of a variable:

```
#pragma omp threadprivate(var)
```

`var`'s lifetime is not limited to one parallel region.

Controlling Thread Data, Persistent Data

```
#include <stdio.h>
#include <omp.h>

int a;
double x;

#pragma omp threadprivate(a,x)
int main()
{
   int b = 0, tid;
   omp_set_dynamic(0);
   printf("1st Parallel Region:\n");
   #pragma omp parallel private(b,tid)
   {
      tid = omp_get_thread_num();
      a = tid;
      b = tid;
      x = 1.1 * tid +1.0;
```

```
      printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
   }  /* end of parallel section */

   printf("************************************\n");
   printf("Master thread doing serial work here\n");
   printf("************************************\n");

   printf("2nd Parallel Region:\n");
   #pragma omp parallel private(tid)
   {
      tid = omp_get_thread_num();
      printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
   }  /* end of parallel section */
}
```

Exercise 4 - Homework, Matrix Multiplication Revisited

Write a program that performs matrix multiplication.

- Create two double dimensioned arrays, populate them with random numbers using a single thread
- Apply what we've learned about Data Races, and keep an eye on "red flags"
- Create a set of nested loops that multiplies the two arrays (your matrices) together in parallel and using a simple reduction clause (we will be covering reduction in depth)
- Add a worksharing clause, you may need to experiment with this.
- add a time function, and record start time and end time of your loop and how long the loop took to process.
- run this for a 10x10, 100x100, and 1000x1000

How are your running times now compared to when we first did this assignment (Exercise 2)?