Parallel Programming using OpenMP

Oregon State University

Mike Bailey

mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

OpenMP Multithreaded Programming

• OpenMP stands for "Open Multi-Processing"

• OpenMP is a multi-vendor (see next page) standard to perform shared-memory multithreading

• OpenMP uses the fork-join model

• OpenMP is both directive- and library-based

• OpenMP threads share a single executable, global memory, and heap (malloc, new)

• Each OpenMP thread has its own stack (function arguments, function return address, local variables)

• Using OpenMP requires no dramatic code changes

• OpenMP probably gives you the biggest multithread benefit per amount of work you have to put in to using it

Much of your use of OpenMP will be accomplished by issuing C/C++ "pragmas" to tell the compiler how to build the threads into the executable

#pragma omp directive [clause]

Who is in the OpenMP Consortium?

AMD ARM CRAY FUJITSU Hewlett Packard Enterprise

IBM intel Micron NEC NVIDIA

ORACLE redhat

Argonne BSC OMP

epcc Inria Los Alamos NATIONAL LABORATORY NASA

RWTH AACHEN UNIVERSITY TACC

What OpenMP Isn't:

• OpenMP doesn't check for data dependencies, data conflicts, deadlocks, or race conditions. You are responsible for avoiding those yourself

• OpenMP doesn't check for non-conforming code sequences

• OpenMP doesn't guarantee ***identical*** behavior across vendors or hardware, or even between multiple runs on the same vendor's hardware

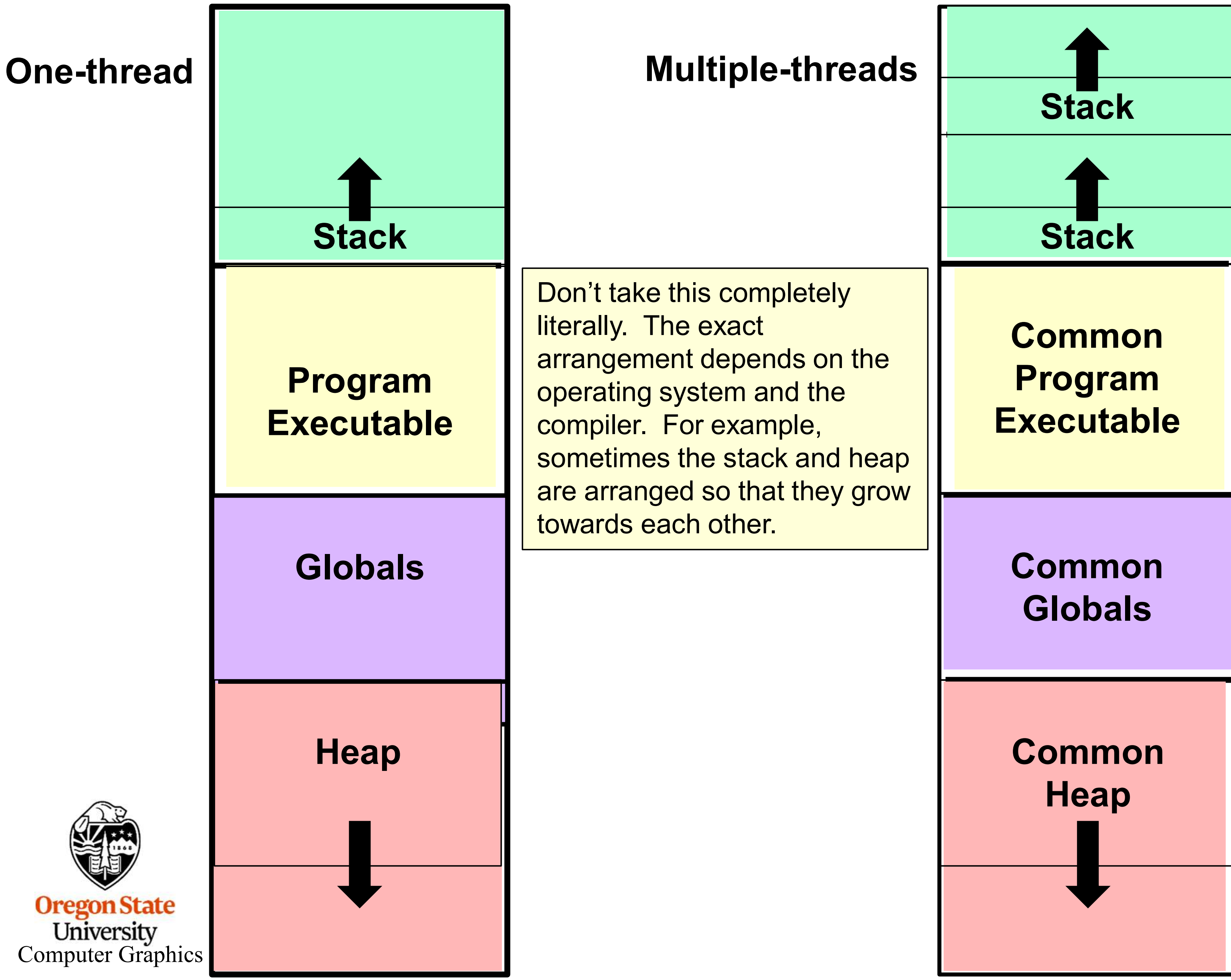
• OpenMP doesn't guarantee the ***order*** in which threads execute, just that they do execute

• OpenMP is not overhead-free

• OpenMP does not prevent you from writing code that triggers cache performance problems (such as in false-sharing), in fact, it makes it really easy

We will get to "false sharing" in the cache notes

Memory Allocation in a Multithreaded Program



Don't take this completely literally. The exact arrangement depends on the operating system and the compiler. For example, sometimes the stack and heap are arranged so that they grow towards each other.

Using OpenMP on Linux

g++ -o proj proj.cpp -lm -fopenmp

icpc -o proj proj.cpp -lm -openmp -align -qopt-report=3 -qopt-report-phase=vec

Using OpenMP in Microsoft Visual Studio

1. Go to the Project menu ⟶ Project Properties

2. Change the setting Configuration Properties ⟶ C/C++ ⟶ Language ⟶ OpenMP Support to **"Yes (/openmp)"**

Seeing if OpenMP is Supported on Your System

```
#ifndef   _OPENMP
    fprintf( stderr, "OpenMP is not supported – sorry!\n" );
    exit( 0 );
#endif
```

A Potential OpenMP/Visual Studio Problem

If you are using Visual Studio 2019 and get a compile message that looks like this:

1>c1xx: error C2338: two-phase name lookup is not supported for C++/CLI, C++/CX, or OpenMP; use /Zc:twoPhase-

then do this:

1. Go to "Project Properties"→ "C/C++" → "Command Line"
2. Add **/Zc:twoPhase-** in "Additional Options" in the bottom section
3. Press OK

No, I don't know what this means either …

Numbers of OpenMP threads

How to specify how many OpenMP threads you want to have available:

```
omp_set_num_threads( num );
```

Asking how many cores this program has access to:

```
num = omp_get_num_procs( );
```

← Actually returns the number of hyperthreads, not the number of *physical* cores

Setting the number of available threads to the exact number of cores available:

```
omp_set_num_threads(      omp_get_num_procs( )      );
```

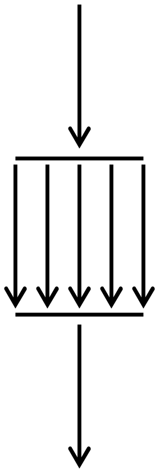
Asking how many OpenMP threads this program is using right now:

```
num = omp_get_num_threads( );
```

Asking which thread number this one is:

```
me = omp_get_thread_num( );
```

Creating an OpenMP Team of Threads



```
#pragma omp parallel default(none)
{
        . . .
}
```

This creates a team of threads

Each thread then executes all lines of code in this block.

Think of it this way:

```
#pragma omp parallel default(none)
```

Creating an OpenMP Team of Threads


```
#include <stdio.h>
#include <omp.h>
int
main( )
{
        omp_set_num_threads( 8 );
        #pragma omp parallel default(none)
        {
            printf( "Hello, World, from thread #%d ! \n" , omp_get_thread_num(  )   );
        }
        return 0;
}
```

Hint: run it several times in a row. What do you see? Why?

Uh-oh…

First Run

```
Hello, World, from thread #6 !
Hello, World, from thread #1 !
Hello, World, from thread #7 !
Hello, World, from thread #5 !
Hello, World, from thread #4 !
Hello, World, from thread #3 !
Hello, World, from thread #2 !
Hello, World, from thread #0 !
```

Second Run

```
Hello, World, from thread #0 !
Hello, World, from thread #7 !
Hello, World, from thread #4 !
Hello, World, from thread #6 !
Hello, World, from thread #1 !
Hello, World, from thread #3 !
Hello, World, from thread #5 !
Hello, World, from thread #2 !
```

Third Run

```
Hello, World, from thread #2 !
Hello, World, from thread #5 !
Hello, World, from thread #0 !
Hello, World, from thread #7 !
Hello, World, from thread #1 !
Hello, World, from thread #3 !
Hello, World, from thread #4 !
Hello, World, from thread #6 !
```

Fourth Run

```
Hello, World, from thread #1 !
Hello, World, from thread #3 !
Hello, World, from thread #5 !
Hello, World, from thread #2 !
Hello, World, from thread #4 !
Hello, World, from thread #7 !
Hello, World, from thread #6 !
Hello, World, from thread #0 !
```

There is no guarantee of thread execution order!

Creating OpenMP threads in Loops

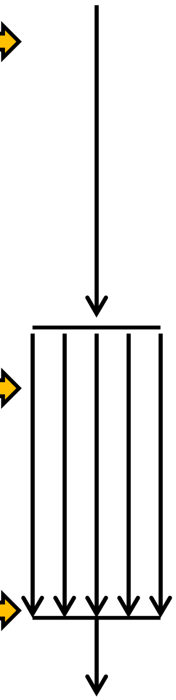
```
#include <omp.h>

. . .

omp_set_num_threads( NUMT );

. . .

#pragma omp parallel for default(none)
for( int i = 0; i < arraySize; i++ )
{
        . . .
}
```

The code starts out executing in a single thread

This sets how many threads will be in the thread pool. It doesn't create them yet, it just says how many will be used the next time you ask for them.

This creates a team of threads from the thread pool and divides the for-loop passes up among those threads

There is an **"implied barrier"** at the end where each thread waits until all threads are done, then the code continues in a single thread



This tells the compiler to parallelize the for-loop into multiple threads. Each thread automatically gets its own personal copy of the variable ***i*** because it is defined within the for-loop body.

The **default(none)** directive forces you to explicitly declare all variables declared outside the parallel region to be either private or shared while they are in the parallel region. Variables declared within the for-loop are automatically private.

OpenMP for-Loop Rules


```
#pragma  omp  parallel for  default(none), shared(…), private(…)

for( int index = start ; index terminate condition; index changed )
```

- The *index* must be an *int* or a *pointer*
- The *start* and *terminate* conditions must have compatible types
- Neither the *start* nor the *terminate* conditions can be changed during the execution of the loop
- The *index* can only be modified by the *changed* expression (i.e., not modified inside the loop itself)
- You cannot use a *break* or a *goto* to get out of the loop
- There can be no inter-loop data dependencies such as:

a[i] = a[i-1] + 1.;

a[101] = a[100] + 1.; // what if this is the ***last*** line of thread #0's work?

---

a[102] = a[101] + 1.; // what if this is the ***first*** line of thread #1's work?

OpenMP For-Loop Rules

```
for( index = start ;  index <  end   ;  index++                 )
                      index <= end      ++index
                      index >  end      index--
                      index >= end      --index
                                        index += incr
                                        index = index + incr
                                        index = incr + index
                                        index -= decr
                                        index = index - decr
```

Oregon State University Computer Graphics

What to do about Variables Declared Before the for-loop Starts?

```
float x = 0.;
#pragma omp parallel for …
for( int i = 0; i < N; i++ )
{
        x = (float) i;
        float y = x*x;
        << more code… >
}
```

i and **y** are automatically *private* because they are defined within the loop.

Good practice demands that **x** be explicitly declared to be shared or private!

private(x)
Means that each thread will get its own version of the variable

shared(x)
Means that all threads will share a common version of the variable

default(none)
I recommend that you include this in your OpenMP for-loop directive. This will force you to explicitly flag all of your externally-declared variables as *shared* or *private*. Don't make a mistake by leaving it up to the default!

Example:
#pragma omp parallel for default(none), private(x)

For-loop "Fission"

Because of the loop dependency, this whole thing is not parallelizable:

```
x[ 0 ] = 0.;
y[ 0 ] *= 2.;
for( int i = 1; i < N; i++ )
{
        x[ i ] = x[ i-1 ] + 1.;
        y[ i ] *= 2.;
}
```

But, it *can* be broken into one loop that is not parallelizable, plus one that is:

```
x[ 0 ] = 0.;
for( int i = 1; i < N; i++ )
{
        x[ i ] = x[ i-1 ] + 1.;
}

#pragma omp parallel for shared(y)
for( int i = 0; i < N; i++ )
{
        y[ i ] *= 2.;
}
```

For-loop “Collapsing”

Uh-oh, which for-loop do you put the #pragma on?

```
for( int i = 1; i < N; i++ )
{
        for( int j = 0; j < M; j++ )
        {
                . . .
        }
}
```

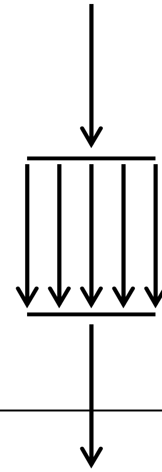
How many for-loops to collapse into one loop

Ah-ha – trick question. You put it on both!

```
#pragma omp parallel for collapse(2)
for( int i = 1; i < N; i++ )
{
        for( int j = 0; j < M; j++ )
        {
                . . .
        }
}
```

Single Program Multiple Data (SPMD) in OpenMP

```
#define NUM          1000000
float A[NUM], B[NUM], C[NUM];
. . .
total = omp_get_num_threads( );
#pragma omp parallel default(none),private(me),shared(total)
{
      me = omp_get_thread_num( );
      DoWork( me, total );
}
```



```
void  DoWork( int me, int total )
{
                int first = NUM * me / total;
                int last = NUM * (me+1)/total   -   1;
                for( int i = first; i <= last; i++ )
                {
                                C[ i ] = A[ i ] * B[ i ];
                }
}
```

OpenMP Allocation of Work to Threads

Static Threads

- All work is allocated and assigned at runtime

Dynamic Threads

- The pool is statically assigned some of the work at runtime, but not all of it
- When a thread from the pool becomes idle, it gets a new assignment
- “Round-robin assignments”

OpenMP Scheduling

schedule(static [,chunksize])
schedule(dynamic [,chunksize])
Defaults to static
chunksize defaults to 1

OpenMP Allocation of Work to Threads

```
#pragma omp parallel for default(none),schedule(static,chunksize)
for( int index = 0 ; index < 12 ; index++ )
```

Static,1

0	0,3,6,9
1	1,4,7,10
2	2,5,8,11

chunksize = 1
Each thread is assigned one iteration, then the assignments start over

Static,2

0	0,1,6,7
1	2,3,8,9
2	4,5,10,11

chunksize = 2
Each thread is assigned two iterations, then the assignments start over

Static,4

0	0,1,2,3
1	4,5,6,7
2	8,9,10,11

chunksize = 4
Each thread is assigned four iterations, then the assignments start over

Arithmetic Operations Among Threads – A Problem


```
#pragma omp parallel for private(myPartialSum),shared(sum)
 for( int i = 0; i < N; i++ )
{
        float myPartialSum = …

        sum = sum + myPartialSum;
}
```

- There is no guarantee when each thread will execute this line

- There is not even a guarantee that each thread will finish this line before some other thread interrupts it. (Remember that each line of code usually generates multiple lines of assembly.)

- This is non-deterministic !

Assembly code:

```
Load sum
Add    myPartialSum
Store sum
```

What if the scheduler decides to switch threads right here?

Conclusion: Don't do it this way!

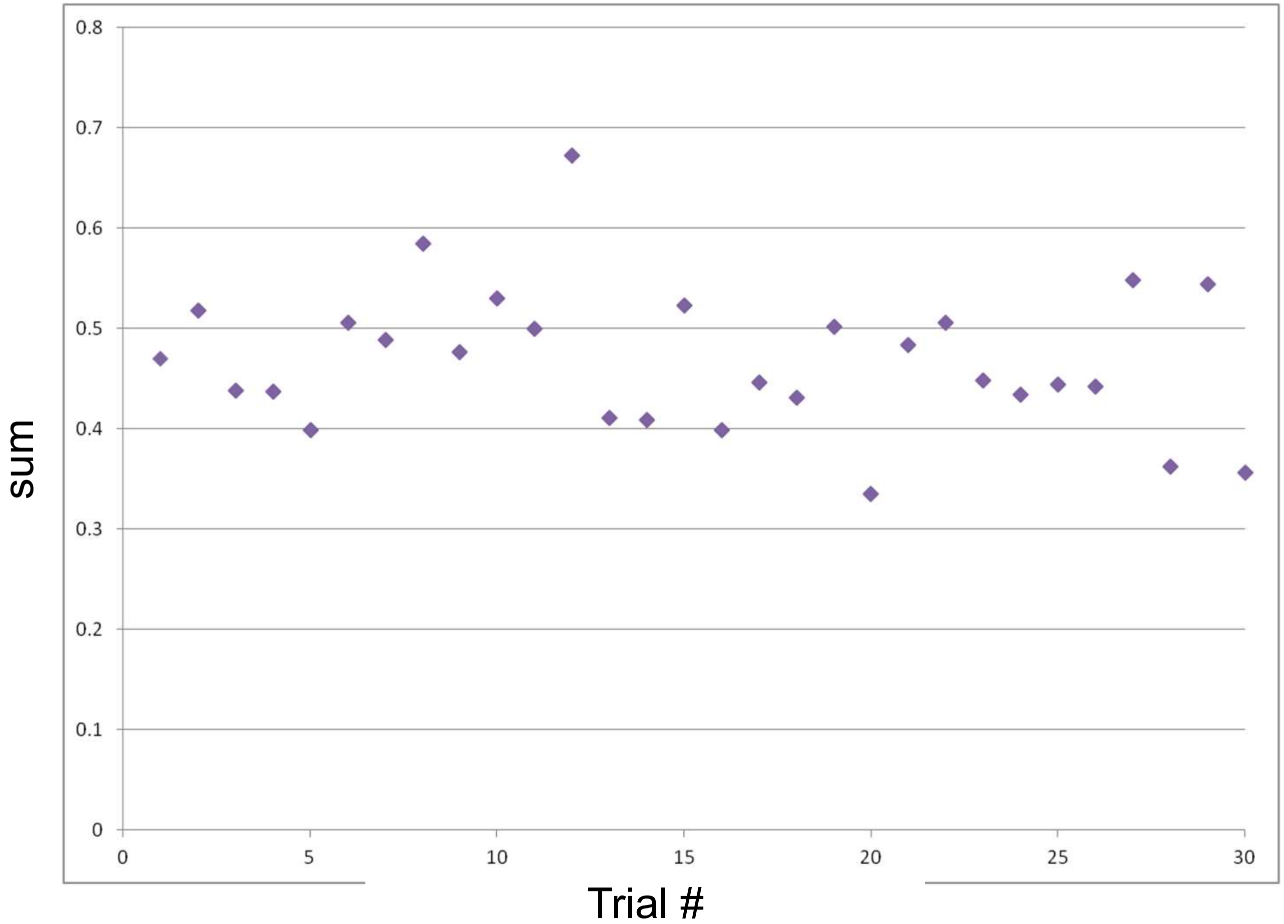
Here's a trapezoid integration example.
The partial sums are added up, as shown on the previous page.
The integration was done 30 times.
The answer is supposed to be exactly 2.
None of the 30 answers is even close.
And, not only are the answers *bad*, they are not even consistently *bad*!

0.469635	0.398893
0.517984	0.446419
0.438868	0.431204
0.437553	0.501783
0.398761	0.334996
0.506564	0.484124
0.489211	0.506362
0.584810	0.448226
0.476670	0.434737
0.530668	0.444919
0.500062	0.442432
0.672593	0.548837
0.411158	0.363092
0.408718	0.544778
0.523448	0.356299

Oregon State

Don't do it this way! We'll talk about how to it correctly in the Trapezoid Integration noteset.

Here's a trapezoid integration example.
The partial sums are added up, as shown on the previous page.
The integration was done 30 times.
The answer is supposed to be exactly 2.
None of the 30 answers is even close.
And, not only are the answers *bad*, they are not even consistently *bad*!



Oregon State

Don't do it this way! We'll talk about how to it correctly in the Trapezoid Integration noteset.

Synchronization

Mutual Exclusion Locks (Mutexes)

*omp_init_lock(omp_lock_t *);*
*omp_set_lock(omp_lock_t *);* ← Blocks if the lock is not available
Then sets it and returns when it is available
*omp_unset_lock(omp_lock_t *);*
*omp_test_lock(omp_lock_t *);* ← If the lock is not available, returns 0
If the lock is available, sets it and returns !0

(*omp_lock_t* is really an array of 4 unsigned chars)

Critical sections

#pragma omp critical
Restricts execution to one thread at a time

#pragma omp single
Restricts execution to a single thread ever

Barriers

#pragma omp barrier
Forces each thread to wait here until all threads arrive

(Note: there is an implied barrier after parallel for loops and OpenMP sections, unless the *nowait* clause is used)

Synchronization Examples

```
omp_lock_t          Sync;
. . .
omp_init_lock( &Sync );

. . .

omp_set_lock( &Sync );
          << code that needs the mutual exclusion >>
omp_unset_lock( &Sync );

. . .

while(  omp_test_lock( &Sync ) == 0  )
{
          DoSomeUsefulWork( );
}
```

Single-thread-execution Synchronization

#pragma omp single

Restricts execution to a single thread ever. This is used when an operation only makes sense for one thread to do. Reading data from a file is a good example.

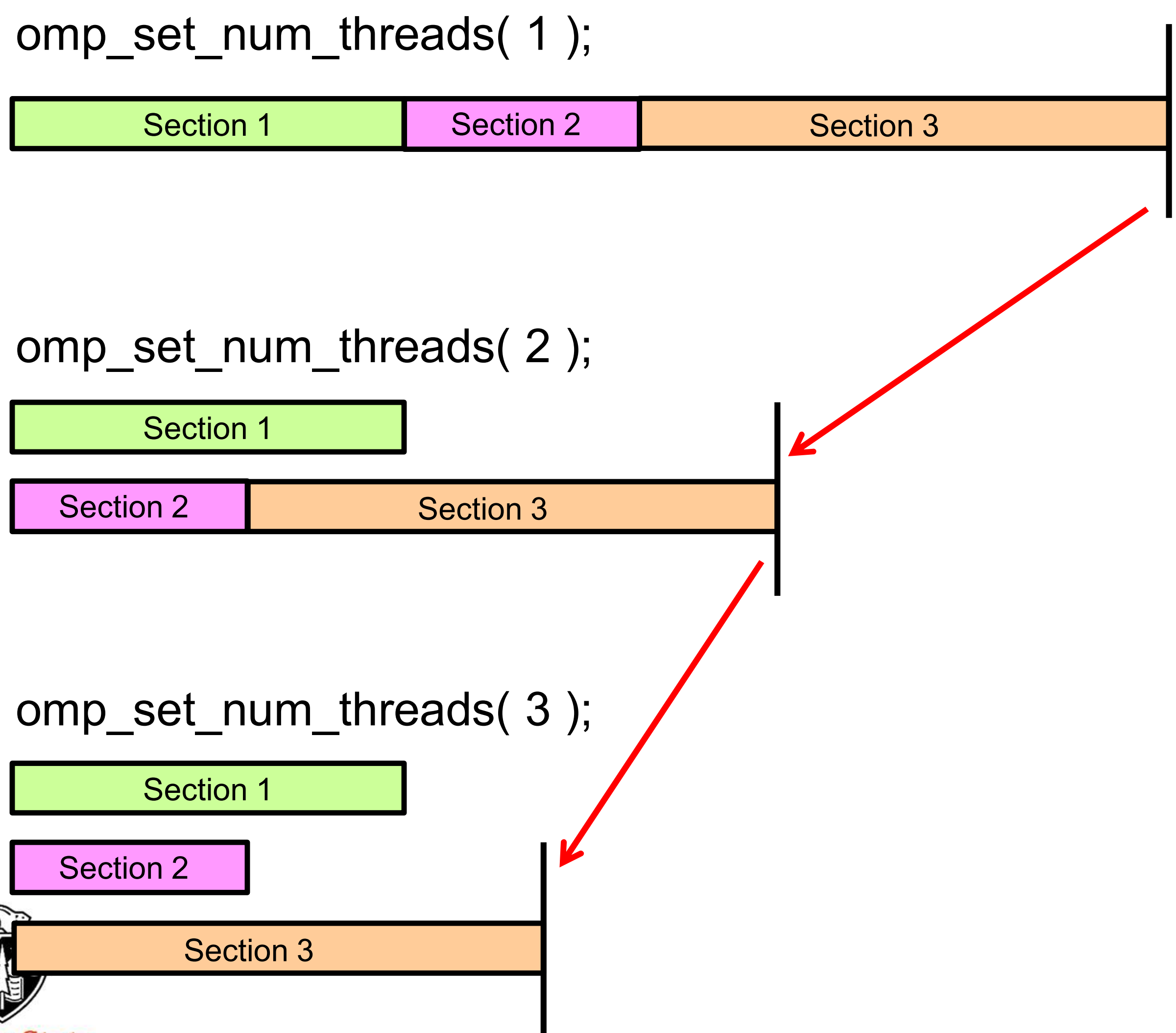
Creating Sections of OpenMP Code

Sections are independent blocks of code, able to be assigned to separate threads if they are available.

```
#pragma omp parallel sections
{
        #pragma omp section
        {
                Task 1
        }
        #pragma omp section
        {
                Task 2
        }
}
```

(Note: there is an **implied** barrier after parallel for loops and OpenMP sections, unless the *nowait* clause is used)

What do OpenMP Sections do for You?
They decrease your overall execution time.



A Functional Decomposition Sections Example

```
omp_set_num_threads( 3 );

#pragma omp parallel sections
{
        #pragma omp section
        {
                Watcher( );
        }

        #pragma omp section
        {
                Animals( );
        }

        #pragma omp section
        {
                Plants( );
        }

}   // implied barrier -- all functions must return to get past here
```