Simple OpenMP

Oregon State University

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University
Computer Graphics

OpenMP Multithreaded Programming

- OpenMP stands for "Open Multi-Processing"
- It is run by a consortium of companies, labs, and universities
- OpenMP (IMHO) gives you the biggest multithread benefit per amount of work you have to put into 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]

That's it! That's where the compiler comes in.

But, as you are about to find out, doing parallel processing ***at all*** is not difficult.

The trick is doing parallel processing ***well***.
That's where *you* come in.

Using OpenMP on Linux

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

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)"**

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

Seeing if OpenMP is Supported on Your System

```
#ifdef   _OPENMP
      fprintf( stderr, "OpenMP version %d is supported here\n", _OPENMP );

#else
      fprintf( stderr, “OpenMP is not supported here – sorry!\n” );
      exit( 0 );

#endif
```

This gives you a year and month of the OpenMP you are using

To get an OpenMP version number:

OpenMP 5.0 – November 2018

OpenMP 4.5 – November 2015

OpenMP 4.0 – July 2013

OpenMP 3.1 – July 2011

- By default, flip is using g++ 11.4.1, which uses OpenMP version 4.5
- Looks like Visual Studio 2019's is even older (?)

Oregon State University
Computer Graphics

Threads

We will get into more detail pretty soon, but for now, know that a thread is an independent execution path for your code to take.

Threads are at their very best when each one can run on a separate hardware core.

Seeing if OpenMP is Supported on Your System:

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

How to find out how many cores your system has:

```
int  numprocs  =  omp_get_num_procs( );
```

How to specify how many OpenMP threads you want to reserve starting now:

```
omp_set_num_threads( num );
```

How to use one thread per core:

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

Creating OpenMP threads for a for loop

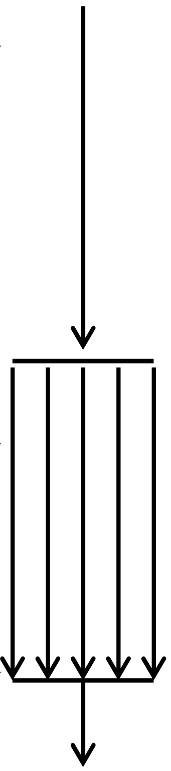
```
#include <omp.h>

. . .

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 statement are automatically private