Simple OpenMP

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University

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

AMD
ARM
CRAY
FUJITSU
Hewlett Packard Enterprise
IBM
intel
Micron
NEC
NVIDIA
ORACLE
redhat
Argonne
BSC
OMPunity
epcc
Inria
Los Alamos NATIONAL LABORATORY
NASA
RWTH AACHEN UNIVERSITY
TACC

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.

Doing parallel processing ***well*** is harder. That’s where you come in.

Using OpenMP in 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)"**

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 OpenMP threads your system can use:

```
int  numthreads  =  omp_get_num_threads( );
```

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

```
omp_set_num_threads( num );
```

Creating OpenMP threads for a for loop

```
#include <omp.h>

. . .

omp_set_num_threads( NUMT );

. . .

#pragma omp parallel for
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

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