1. OpenMP Tutorial

Introduction
This tutorial is aimed at introducing you to the OpenMP API, we will go over some of the basics concepts by examining a number of examples. It will be a very hands on tutorial, it has been structured so that you can explore and learn by experimenting. Some of the initial material is very basic (hello world) just to ensure everyone has the basics.

The code for this tutorial is all written in c++ and fortran, you can decide which language you would rather use. The tutorial is structured in such a way that you will only have to edit small sections of existing code. A number of applicable problems solutions have been written in serial, and the skeleton for the parallel sections has been written for you. You will have to examine these and edit them so that they can run effectively in parallel. You will be using the tools you learned during the previous lecture.

Once you have completed a section it can be run, verified and benchmarked against the serial version. These benchmarks will be run on a cluster run by CHPC.

Feel free to ask for assistance if you get stuck at any point.

For a OpenMP reference try:
http://ci-tutor.ncsa.illinois.edu/login.php (need to register, but free, and GOOD)

or

http://openmp.org/wp/

or

https://computing.llnl.gov/tutorials/openMP/

Files
There are two folders in the OpenMP_Tut folder. One called Fortran, which contains a tut for those of you who would prefer to work with fortran and one called CPP which contains the files for those of you who would prefer to work with C++.

Inside the CPP/Fortran folder you should see 4 folders

Info
Stats
Integration
Matrix

Each folder contains the code you need for each section of this tut.

Copy these to your local directory.

Part 1: Introduction (OpenMP hello world)

In this section you will write, compile and run a simple OpenMP program. It will not perform any real or useful task, other that getting you acquainted with the environment.

Open the Info folder

Inside you will see the info.cpp or info.f90 file. This file contains the skeleton for your first openMP program.

Note that you can compile these programs as normal c++ or fortran programs using the following commands.

g++ info.cpp -o info

gfortran info.f90 -o info

For both the c++ and fortran code you can run your compiled program with
./info

1.1 OpenMP Runtime Library

Your first task is to open the info.cpp/info.f90 file

The first step when planning to write an openMP program is to include the runtime library.

In c++ this is achieved by including the omp.h header file
In fortran this is achieved by using the omp_lib module

Include these libraries in your info code

In the code you will see four variables:

num_procs
num_threads
max_threads
thread_num

Each of these variables are initially set to -1.

You must assign these variables by using the appropriate OpenMP Runtime Library Routines. You can find by looking here.

1.2 Compiling with OpenMP

Once completed you will want to see the output. If you try and compile the way you did above, you will get an error saying you have used an undefined reference to your openMP routines.

To compile your code using OpenMP directives you must compile with the -fopenmp compiler option.
g++ -fopenmp info.cpp -o info
gfortran -fopenmp info.f90 -o info

After compiling with the -fopenmp flag you should be able to run your program with OpenMP functionality as you did before.

./info

After running your code you should have printed out some basic information about your OpenMP environment.

1.3 OpenMP Compiler Directives

You have OpenMP working so lets get some code running in parallel. For this section you are required to parallelize the code indicated by my comments in the code.

In c++ this can be done by adding an appropriate directive above the opening bracket.

```
#pragma omp parallel
{
        Parallel code
}
```

In Fortran you must indicate both where your parallel section starts and ends using OpenMP directives.

```
!$OMP PARALLEL
        Parallel code
!$OMP END PARALLEL
```

In both cases you must ensure that you know which variables are shared amongst threads and which variables are private to each thread.

You can explicitly set a variable to private or shared using the appropriate clauses. For more info on parallel regions look here.

If you are successful you should have output
similar to this:

```
./info
     --==== Basic OpenMP information ===--

There are  8 processors.
There are currently  1 thread(s).
The maximum number of threads is  8.
This is thread number  0.

Start of parallel execution

Hello from thread  0 of  8
I am the master thread! I am thread  0 of  8
Hello from thread  6 of  8
I am the master thread! I am thread  6 of  8
Hello from thread  4 of  8
I am the master thread! I am thread  4 of  8
Hello from thread  5 of  8
I am the master thread! I am thread  5 of  8
Hello from thread  7 of  8
I am the master thread! I am thread  7 of  8
Hello from thread  1 of  8
I am the master thread! I am thread  1 of  8
Hello from thread  2 of  8
I am the master thread! I am thread  2 of  8
Hello from thread  3 of  8
I am the master thread! I am thread  3 of  8
```

It is important to note that the threads are forked at the beginning of the section, thus all the code in the section is run by every thread. Note that the order of execution is undefined.

By default the number of threads OpenMP will run is implementation specific, but you will find it is usually as many as there are processors, in my case 8.

1.4 Control

You may want to change the number of threads being run. This can be achieved by calling the omp_set_num_threads() runtime routine. This should be called before the parallel section. You could also use the num_threads() clause when you define your parallel region. Try changing the number of threads using both methods, for example, to 11 or 4, and see how this effects the output.

Feel free to play with any other OpenMP features, before you proceed with the rest of the tutorial.

1.5 Single thread

Some times you have a task that must be performed by only one thread, for this make only thread 0 execute the code indicated by my comments in the code.

Benchmarking

Once you have gotten your code working on the login node, you can schedule your code to run on the UFS cluster. To do this navigate to the folder containing your code and run

qsub test.pbs
This will return a jobNumber

you can check the status of your run with

qstat jobNumber

To view the result of the run on the cluser simply run the command

cat test.log

This will print the result of your run to the terminal

If your code did not run correctly you can view any errors generated with the command

cat test.err

Part 2: Stats

Introduction

Reduction is a common problem, and often trivially parallelizable. Reduction includes tasks such as finding the highest or lowest value in a set or summing a list of numbers. The assumption is that this operation is independent of the rest of the values in the set. Thus in these type of scenarios each thread can perform the task on a subset of the total set and then the combine their separate results to give the final result.

In this example we will be calculating some basic statistics of a list of doubles. These are the: minimum, maximum, sum, mean and standard deviation. Remember that by its very nature a reduction will need some form of synchronization. This can be achieved in a number of different ways, these include using a critical section, the reduce clause or using the atomic construct.

Code

Go to the Stats folder and open the stats.cpp or stats.f90 file.

You can compile this code with

```
g++ -fopenmp stats.cpp main.cpp -o stats
gfortran -fopenmp stats.f90 main.f90 -o stats
```

and run with

```
./stats
```

The code should contain:
A sStats method – Serial version of code (implemented by me, do not touch this)
A pStats method – Parallel version (implemented by you)

While you are implementing your code you should comment out the call to the test method in the main section of the code. Once you are happy that you have your parallel method working you can uncomment it to verify.

When testing the parallel implementation the main program will call pStats(). You will see that I have copied the serial implementation into pStats() this way if you run the code in the state it started the answers will verify. You can test this by running your code.

When your parallel code is verified, a number of tests are run to show you how the size of the problem and the number of threads used to solve the problem affect the speed up get with your parallel code over the serial version.

As it is the code should verify, however you should notice no significant speedup as we are comparing to identical serial versions.

Parallel
Have a look at the sStats method and examine how everything fits together. Have a look at how the various loops are structured.

Remember that a reduction will require some form of synchronization of variables. There are a number of options as to how you can handle this, these include:

Using the reduce clause - Tell the compiler the operation you are reducing with and the variable to save your result in.
Using a critical section – Surround a block of code so that only one thread can enter it at a time.
Using the atomic construct - Tell the compiler when you are updating a shared variable

Pay attention to dependences and be aware of which loop(s) you parallelize, remember it can make a difference to efficiency.

Task
You are to implement and test the three methods mentioned above.

Try to see how each method affects the efficiency of your code.

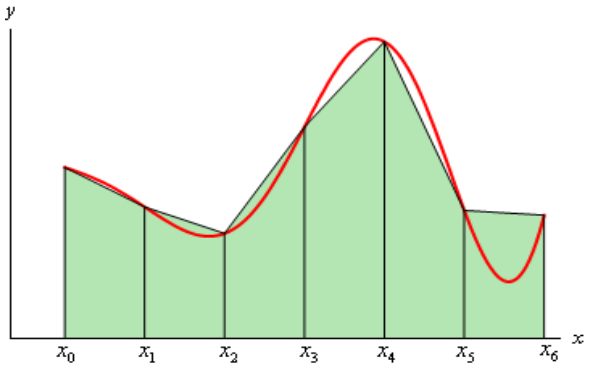
Once you have verified your code is working you can schedule it to run on the cluster as described in the Benchmarking section of this tut

Write up a paragraph or two to explain how each method works and save it to /public/ OpenMP_Tut/Writeup/part2_yourname.txt

Part 3: Numeric Integration

Introduction

Some integrals can not be solved algebraically and need to be estimated numerically. The trapezoidal rule divides a function into segments and calculates the sum of the areas of each segment



We have implemented a version of the trapezoidal rule, which states:

$$\int_a^b f(x)d(x) \approx \frac{(b-a)}{2\mathrm{N}} \sum_{k=1}^{N} (f(k)+f(k+1))$$

This is parallelizable as the elements of the sum are independent, and can be calculated separately.

Code

Go to the Integration folder and open the integration.cpp or integration.f90 file.

You can compile this code with:

g++ -fopenmp integration.cpp main.cpp -o integration
gfortran -fopenmp integration.f90 main.f90 -o integration

and run with
./integration

The code should contain
f method – The function f(x) (currently f(x) = x)
pTrapezoid method – The parallel integration method which you will be implementing.
Note that initially pTrapezoid is not implemented so the results will not validate.
You can define your own functions if you want to experiment, just alter the definition of the method f to whatever function you would like to integrate.

Task

You are to implement the contents of pTrapezoid first implement a serial version making sure the results validate. Then parallelize it and see what the best speed up you can get is.
Once you have verified your code is working you can schedule it to run on the cluster as described in the Benchmarking section of this tut.

Part 4: Matrix multiplication

Introduction

Matrix operations play a key role in many scientific applications. We will be looking at the simple case of a 2D matrix multiplication.

Code

Open the matrix directory and open the matrix.cpp or matrix.f90 file

You can compile this code with:
g++ -fopenmp matrix.cpp main.cpp -o matrix
gfortran -fopenmp matric.f90 main.f90 -o matrix

and run with :
./matrix

The code should contain
pMatMultiply method – This is the method you must implement and parallelize

Task

Your task is to implement a parallel version of matrix multiply. First implement a serial version of matrix multiply and verify your results. Once you have a serial version you should parallelize it.

Remember the order you access elements in your array can seriously affect the efficiency of your memory accesses and the speed your code will run at.

Once you have implemented and parallelized the matrix multiply method your results should validate and you will see how different numbers of threads and problem sizes affect the speed up of your algorithm.

Once you have verified your code is working you can schedule it to run on the cluster as described in the Benchmarking section of this tut

Write up how the problem size and number of threads affects your speed up in a few paragraphs and save it to

/public/OpenMP_Tut/Writeup/part4_yourname.txt