Hands On Session 4: Shared Memory Programming With OpenMP

John Burkardt
Information Technology Department
Virginia Tech
....
HPPC-2008
High Performance Parallel Computing Bootcamp

31 July 2008

This hands on session introduces OpenMP, which can be used to write parallel programs on shared memory systems.

We will use VT's SGI system; in particular, we will log in to the node **charon3.arc.vt.edu**.

We will compile programs interactively on **charon3**, but (with one exception!), we will not run programs interactively. We will use a script to submit a request to a batch system, so our jobs will run on some set of *compute nodes* instead.

The exercises in this hands on session will have you:

1. access **charon3**, compile a HELLO program and submit a script to run it;
2. put together a simple OpenMP program called QUAD to estimate an integral
3. make an OpenMP version of the MD program, and determine the dependence of the wall clock time on the problem size, and on the number of processors
4. make an OpenMP version of the FFT program, which will report a MegaFLOPS rate for various problem sizes;
5. make an OpenMP version of the HEATED_PLATE program and run it;

For most of the exercises, there is a source code program that you can use. The source code is generally available in a C, C++, FORTRAN77 and FORTRAN90 version, so you can stick with your favorite language.

Start the hands on session by using SSH to connect to the SGI system:

```
ssh charon3.arc.vt.edu
```

1 Hello, OpenMP World!

Get a copy of the **hello** program

```
cp /home/BOOTCAMP/day4/hello/hello.c .
```

This program is more complicated than it should be. I added some extra sample OpenMP calls for you to look at, including:

- How you get the "include" file.
- How you measure wall clock time.

- How you find out how many processors are available.
- How you find out which thread you are in a parallel section.
- How you find out how many threads are available in a parallel section.
- How you set the number of threads.

1.1 Compile "hello"

Compile the program. The SGI version of the Gnu compilers does not include OpenMP, so all our work today will be with the Intel compilers only! Sample compilation statements include:

```
icc -openmp hello.c
icpc -openmp hello.cc
ifort -openmp -fpp hello.f
ifort -openmp -fpp hello.f90
```

Rename your executable program to **hello**. Especially when you use a batch system to run programs, it's important to distinguish your programs by name.

```
mv a.out hello
```

You could also do this renaming as part of the compilation statement, by including the **-o** *name* switch:

```
icc -openmp -o hello hello.c
```

We should **not** run programs interactively on the SGI compile node. However, just this once, go ahead and run the executable program. Before an iteractive run, you need to define the number of threads, or it will be stuck at 1.

```
export OMP_NUM_THREADS=2      <=== NO SPACES around the = sign!
hello
```

1.2 Run "hello" using the Queueing System

From now on, all our programs will be run using the batch system. That means that we compile the program, give it a name, maybe put the program and its data in a special directory, and then send a request to the batch system. Then we wait until our request is processed.

Copy the script that submits the hello program:

```
cp /home/BOOTCAMP/day4/hello/hello.sh .
```

Take a look at the script. There is almost nothing you have to change. The interesting features include

- something that controls the total time allowed;
- two items that request a number of cpus, and declare (the same number) of OpenMP threads;
- a statement that actually runs your program;

Submit the script

```
qsub hello.sh
```

You can issue the command

```
showq
```

to see the status of your job (unless it's already run.)

Congratulations! Now we can try the harder stuff.

2 Estimate an Integral

In this exercise, you will be asked to finish a program, which is partially sketched out, to approximate the integral

$$\int_a^b \frac{50}{\pi(2500x^2+1)}\,dx$$

The method involves defining an array **x** of **N** equally spaced numbers between **a** and **b**, then estimating the integral as the sum of the function values at these points, multiplied by **b - a** and divided by **N**.

The correct answer is about 0.49936338107645674464...

Get a copy of one of the **quad** programs. For instance:

```
cp /home/BOOTCAMP/day4/quad/quad.cc .
```

2.1 Make an OpenMP version

The program is not ready to run. For one thing, you need to make sure the appropriate **include** file is invoked.

You also need to write the loop that adds up the function values at the **N** entries of the **x** array. Once you have done that, you should count the number of floating point operations in your formula, and use that in the statement that defines the value of **flops**.

Finally, you need to insert the appropriate OpenMP directive just before the loop. You should try to make sure that you classify every variable used in the loop to be **private** or **shared** or **reduction**. And if you are using FORTRAN, you must make sure you add a matching “end” directive.

Once you’ve got that all done, try to compile the program, name the executable **quad**, copy the **quad.sh** script file, and submit it to the queueing system!

2.2 Don’t Give Up (Unless You Tried)!

Please try to fill in the blanks in the **quad** version. It’s the only way you’ll start to pick up the techniques you need. But if you can’t get it to work, you can get a “complete” version of the program with the missing parts filled in.

For instance:

```
cp /home/BOOTCAMP/day4/quad/quad_complete.cc .
```

If you can’t get your own code to work, and you want to move on, you can work from one of these complete versions.

2.3 Measure MegaFLOPS against P and N

Investigate the behavior of the MegaFLOPS rate as a function of the number of integration points **N**. Since 1,000,000 is pretty large, do your comparison by halving **N** a few times.

You can also investigate the dependence of the MegaFLOPS rate as a function of the number of processors. The only thing you have to change in that case is the values of NCPUS and **OMP_NUM_THREADS** in your job script. Do you get a reasonable speedup?

3 The MD program for Molecular Dynamics

This exercise asks you to return to the MD molecular dynamics program that we considered on the first day. At that time, we noted that the **update** function used a large part of the computational time. We are going to try to make some simple modifications to this program so that it uses OpenMP and runs faster.

3.1 Make an OpenMP version of the MD program

Before you make any "improvements" to the MD program, you need to reference the "'include file".

The timing that we carry out will only involve the big loop in the main program, the one that calls **compute** and **update**. Replace the CPU timings by calls to **omp_get_wtime()** .

Now we are ready to parallelize **one** loop, namely, the loop in the **update** routine. This is actually a *nested* loop. Our directives will go just before the first of the two loop statements, so the parallelization will be with respect to the **J** index.

If you're using C/C++, your directive has the form:

```
# pragma omp parallel for private ( list) shared ( list)
```

Try to explicitly place every loop variable into one list or the other. (There are no reduction variables in this loop!)

Compile the program, copy the **md.sh** script and submit the job to be run through the queueing system.

3.2 Interchange I and J

It turns out that the nested loop in **update** could be also be written with the loops interchanged. In the FORTRAN77 version, this would mean we could rewrite this code as:

```
do i = 1, nd
  do j = 1, np
```

This will make no difference in the sequential version, and the parallel version will also run correctly - but will it run *efficiently*? Our job script is asking for 4 processors, and the value of **ND** is 3. Parallelization only takes place with respect to the outer loop index. What do you think will happen, especially if we increase the number of processors we ask for?

3.3 The Compute Routine in MD

The **compute** routine also has a big loop in it, which forms almost the entire function. Although this function does not use up so much time, we will consider making an parallel version of it. This loop has more variables, and it has reduction variables, so you will need some time to set it up. If you're using FORTRAN77, your directive should have the form:

```
c$omp parallel do private ( list) shared ( list) reduction(+: list)
```

and don't forget the

```
c$omp end parallel do
```

at the end.

Compare the times of your program before and after the improvements maked to the **compute** routine. Unfortunately, one big cost in this routine is the many calls to the **distance** function. There are ways to improve the distance calculation, but we won't consider them today.

4 Make an OpenMP version of the FFT program

This exercise asks you to return to the FFT Fast Fourier Transform program that we considered on the first day. We are going to make an OpenMP version of this program. By now you should be familiar with the initial things you have to do, before we can even think about inserting any directives.

Make an OpenMP version of the program now!

That is, insert a reference to the "include" file, replace calls to the CPU timer by calls to **omp_get_wtime**, get the script file **fft.sh**.

Since we haven't set up any parallel loops yet, be nice to the system, and only ask for one thread. Do this by changing the **fft.sh** script so that NCPUS and **OMP_NUM_THREADS** are both set to 1. (We'll change them back to 4 in a short time.)

Submit the script. When the program output returns, you have a nice sampling of the MegaFLOPS rate on this machine for a variety of problem sizes **N** and just one processor **P**.

4.1 The STEP Function in FFT

Indications from the GPROF program suggest that a significant amount of time is spent in the **step** function. We will try to parallelize the FFT program by working on the main loop in this function, whose loop index is **J**.

Just before the **J** loop, insert your OpenMP directive. Note that you must be careful in this example. There are a number of temporary variables which you must declare correctly. Your directive will mainly be a list of which variables you think are shared or private. There are no reduction variables in this loop.

When you think you have set up the correct OpenMP directive, rerun the program on 4 processors and see if you are getting a reasonable speedup compared to your 1 processor run.

5 Make an OpenMP version of the HEATED_PLATE program

The "heated plate" program is designed to solve for the steady state temperature of a rectangular metal plate which has three sides held at 100 degrees, and one side at 0 degrees.

A mesh of **M** by **N** points is defined. Points on the boundary are set to the prescribed values. Points in the interior are given an initial value, but we then want to try to make their values reflect the way heat behaves in real systems.

Our simplified algorithm is an iteration. Each step of the iteration updates the temperature at an interior point by replacing it by the average of its north, south, east and west neighbors.

We continue the iteration until the largest change at any point is less than some user-specified tolerance **epsilon**.

At that time, the program writes a *rather big* text file (whose name is stored as the variable **output_file**), containing the solution information, in case anyone wants to plot it.

Note that this program expects to read the values of **epsilon** and **output_file** from the command line. Thus, if you were going to run the program interactively, you might type something like

```
heated_plate 0.1 plot.txt
```

Make the usual changes so that the program is ready to be run as an OpenMP program. Your first run will be using one processor.

Get a copy of the script **heated_plate.sh**. Change it so the NCPUS and **OMP_NUM_THREADS** are set to 1.

How do we get the commandline input into the program? It's easy! The script file includes a line that is what we would type to run the program interactively. The line in the script file already includes a dummy value of epsilon and for the output file name. You can change either of these values if you want.

Compile your program, submit the job script (for one processor), and when you get the output back, note how long your program takes to run on one processor.

5.1 Parallelize the temperature update loop

After you get your basic timing run, insert OpenMP directives on the loops inside the big loop. We are talking about the double loops that save a copy of **W**, update **W** and compute the maximum change **DIFF**.

Each of these loops can be done in parallel, although you have to treat the variable **DIFF** with a little care. In FORTRAN, you are allowed to declare **DIFF** as a reduction variable, as in

```
c$omp parallel do shared ( i, j, N, u, w ) reduction ( max: diff )
```

However C and C++ are not able to work with a reduction variable that is a maximum or minimum, so the computation of **DIFF** "poisons" the whole loop. If you are working on the C or C++ code, you will have to break up the loop that updates W and computes **DIFF** into two separate loops. Then the update loop will parallelize, and the **DIFF** loop will not.

When you get a parallel version you are satisfied with, it should compute exactly the same values as the sequential version. (Don't forget to check this!) But it should run much faster. Record the CPU time, using the same value of **epsilon**, for 4 threads.