Shared Memory Programming With OpenMP
Exercise Instructions

John Burkardt
Interdisciplinary Center for Applied Mathematics &
Information Technology Department
Virginia Tech
.....
Advanced Computational Science II
Department of Scientific Computing
Florida State University

15 September 2008

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

The lab machines are dual processors, so we can do a small amount of parallelism directly and interactively. We will also take advantage of FSU's HPC system, although that will require transferring files to another system and using a script to submit the programs to run indirectly. However, the HPC system uses nodes with 8 processors, so we will be able to do more with OpenMP there.

The exercises in this hands on session will have you:

1. 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;

At the end of these exercises, you are to work on an assignment **to be handed in by Friday**, involving the Jacobi iteration.

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. You can access the codes by using a browser. For each exercise, there is a corresponding web page that lets you get a copy of the code.

1 Hello, OpenMP World!

Get a copy of the **hello** program. One location is

`http://www.scs.fsu.edu/~burkardt/vt2/fsu_open_mp_2008/hello/hello.html`

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 and run HELLO locally

We can compile the program directly on the lab machine. The lab machines have the Gnu compilers. Sample compilation statements include:

```
gcc hello.c -fopenmp
g++ hello.cc -fopenmp
gfortran hello.f -fopenmp
gfortran hello.f90 -fopenmp
```

The compilation should create an executable program called **a.out**. Rename it to **hello**:

```
mv a.out hello
```

Now set the number of threads to 2, and run the program.

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

1.2 Run HELLO using the Queueing System

We will want to run the remaining exercises on the HPC system.

Of course, it's very convenient to have OpenMP available on the lab system. It means we can easily test and correct our codes interactively. But when we've got them running correctly, we want to do transfer to the HPC system where we can get up to 8 processors if we want.

So if start out with our files on the lab machine, we must transfer copies to the HPC machine, compile the program, send a request to the batch system, and wait until our request is processed.

Copy the script **hello.sh** that submits the hello program. One location is

```
http://www.scs.fsu.edu/~burkardt/vt2/fsu_open_mp_2008/hello/hello.html
```

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;
- an item that requests a number of nodes and processors
- a statement that sets **OMP_NUM_THREADS**
- a statement **./hello** that actually runs the program;

To run the hello program on the HPC system:

- Use **sftp** to transfer your program and script to the HPC system.
- Use **ssh** to login to the HPC system
- Issue the command **source**
- Compile your program, rename it **hello**
- Submit your script by typing **msub 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 with QUAD

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**.

We will set the problem up with **a**=0 and **b**=10, so that the correct answer is about 0.49936338107645674464...

Get a copy of one of the **quad** programs. One location is

```
http://www.scs.fsu.edu/~burkardt/vt2/fsu_open_mp_2008/quad/quad.html
```

2.1 Make an OpenMP version of QUAD

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 Compare your version and mine

Once everyone has had time to try fixing up the QUAD program, I will unprotect completed copies of the program called QUAD_COMPLETE. (Remind me to do this if I forget!)

If your code wasn't working, see if you can fix it by comparing with my version.

2.3 Measure QUAD's MegaFLOPS against P and N

Investigate the behavior of the MegaFLOPS rate as a function of the number of integration points **N**. We're not concerned with accuracy here, just computational speed. So run your program with the following values of N, and record the MegaFLOPS rate: N = 10, 100, 1000, 10,000, 100,000, and 1,000,000.

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 PPN and **OMP_NUM_THREADS** in your job script. Go back to N = 1,000,000 and compare the MegaFLOPS rates for 1, 2, 4 and 8 processors.

3 The MD program for Molecular Dynamics

The MD program is a simple example of a molecular dynamics code. Basically, it places many particles in a 3D region, gives them an initial velocity, and then tracks their movements. The particles are influenced not just by their own momentum, but by attractions to other particles, whose strength depends on distance.

The program is divided into a few large sections. One section, the **update** function, uses a large part of the computational time. We are going to try to make some simple modifications to this function so that the program runs faster.

Get a copy of one version of the MD program from

```
http://www.scs.fsu.edu/~burkardt/vt2/fsu_open_mp_2008/md/md.html
```

3.1 Make an OpenMP version of MD

The first change you must make to the program is to include a reference to the OpenMP "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 calls to **cpu_time** by calls to the OpenMP timing function **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 in MD's Update Routine

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 Parallelize MD's Compute Routine

The **compute** routine also has a big double 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 calls a function **dist** which returns some variables. You will need to decide if those variables are shared or private.

If you're using FORTRAN77, your directive should have the form:

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

and because you're using FORTRAN, you also have to "close" the loop with:

`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 FFT

This exercise asks you to work on a Fast Fourier Transform program called FFT. 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 a "basic' OpenMP version of FFT

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. Set the **fft.sh** script to run on a single processor:

```
nodes=1:ppn=1
```

and

```
export OMP_NUM_THREADS=1
```

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 Modify 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, include some temporary vectors, 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.

When you think you have set up the correct OpenMP directive, modify the script **fft.sh** to run on 2, 4 or 8 processors: (your choice).

```
nodes=1:ppn=4
```

and

```
export OMP_NUM_THREADS=4
```

See if you are getting a reasonable speedup compared to your 1 processor run.

5 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
```

5.1 Make a "basic" OpenMP version of HEATED_PLATE

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**. Start out by setting it up to run on a single processor:

```
nodes=1:ppn=1
```

and

```
export OMP_NUM_THREADS=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.

5.2 Parallelize HEATED_PLATE's 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.

6 ASSIGMENT: The JACOBI program

In the **jacobi** directory, you will find source code for a program to solve a linear system using the Jacobi iteration. The program already includes some OpenMP information, such as the include statement and the timing calls.

The program is set to run with a particular problem size N. You can change N for debugging, but the run you turn in must have N set to 500.

The program includes a routine called **jacobi** which solves the linear system, once it is specified.

Inside this routine there is an iteration, and the iteration involves a number of loops or vector operations.

Run the program on the lab machines, and make sure you understand how it works, and what the output looks like.

Your assignment: Insert OpenMP directives into the code in the **jacobi** routine, so that the resulting code is still correct, and runs in parallel.

Demonstrate that the code runs in paralell by running it on the HPC system, first using 1 processor, and then using 8. (This means you must change both **PPN** and **OMP_NUM_THREADS** in the shell script.) If your 8 processor code does not run "significantly" faster than the 1 processor version, something is probably wrong.

Save the output files of the two runs.

Turn in three files to Chris Harden by Friday, 19 September 2008:

1. your revised source code
2. the output from the HPC run on 1 processor
3. the output from the HPC run on 8 processors

Until Friday afternoon, you may ask me for advice, guidance, or help.