**ACS2\_OPENMP\_2016   
Computer Lab Exercises**

**Tomasz Plewa, Ryan Learn**

**Modified from an assignment by John Burkardt**

**Department of Scientific Computing  
Florida State University**

Applied Computational Science II, Computer Lab Instructions for Tuesday, 30 August 2016, 3:30-6:00pm, room 152 Dirac Science Library.

**Index**

* [Introduction](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/acs2_openmp_2013.html#introduction)
* [hello](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/acs2_openmp_2013.html#hello)
* [quad](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/acs2_openmp_2013.html#quad)
* [md](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/acs2_openmp_2013.html#md)
* [jacobi](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/acs2_openmp_2013.html#jacobi)
* [heated\_plate](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/acs2_openmp_2013.html#heated_plate)

**Introduction**

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

The lab machines are dual-processors, with each processor having 4 cores. It turns out this means we can get all eight cores to cooperate on a parallel program, so we can do OpenMP experiments directly on the lab machines. This will require copying certain files into your directory, using the editor to make some changes, invoking the correct compiler with the appropriate switches, running the executable program, and comparing the execution times for different numbers of parallel processors.

The OpenMP skills you learn can also be used on FSU's Research Computing Center (RCC) cluster, which includes nodes with 48 processors. To do so, you would need to request an account on the RCC system, and learn a little about how to use the non-interactive batch system to execute jobs. OpenMP is a widely used system, so the skills you practice today can be used on your laptop, desktop, or any research cluster that uses C, C++, or FORTRAN.

The exercises involve the following programs:

* *hello* will introduce you to OpenMP;
* *quad* will estimate an integral;
* *md* does molecular dynamics; we'll investigate problem size and number of processors;
* *jacobi* is a program you will work on as your assignment.
* *heated\_plate* is a serial program which you'll convert to OpenMP;

For each exercise, there is a source code program that you can use. The source code is available in a C, C++, FORTRAN77 and FORTRAN90 version, so you can stick with your favorite language. (If you are a fan of PYTHON, the only way I know to use OpenMP requires you to write a PYTHON program for which the parallel loops are actually written in C. You are welcome to try such an approach.) You can copy each code, in a language that suits you, by using a browser.

You may want to refer to the lecture notes on OpenMP in the Class Blackboard.

**Hello, OpenMP World!**

For our first exercise, we're just going to try to run a program that uses OpenMP. Pick a version of the *hello* program:

* [hello.c](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/hello/hello.c)
* [hello.cpp](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/hello/hello.cpp)
* [hello.f](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/hello/hello.f)
* [hello.f90](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/hello/hello.f90)

You don't need to look at the text of the program, but if you do, you'll see it's a little more complicated than the average "Hello, world!" program. I added some extra sample OpenMP calls:

* 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.

**Compile the program**. The lab machines have the Gnu compilers. Sample compilation statements include:

* gcc hello.c -fopenmp
* g++ hello.cpp -fopenmp
* gfortran hello.f -fopenmp
* gfortran hello.f90 -fopenmp

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

mv a.out hello

**Run your compiled program**, using the command

./hello

Although the program is set up with OpenMP, you probably haven't defined the number of threads to use, so the program will use the default, which *might* be the default value of 8.

Explicitly **set the number of threads to 2**, run the program again, and note the difference.

export OMP\_NUM\_THREADS=2 <=== NO SPACES around the = sign!

./hello

Notice that the program itself did not change at all, only the environment, the value of **OMP\_NUM\_THREADS**. You can experiment with other values of this quantity. On some systems, your thread request can't exceed the number of cores available. On others, the thread request can be as high as you like.

If you wish to save a copy of the output file, simply use the **output redirection** command:

./hello > hello\_output.txt

Now the file *hello\_output.txt* contains the information that the program would otherwise have printed to the screen.

**A Quadrature Code**

For this exercise, pick a version of the *quad* program:

* [quad.c](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/quad/quad.c)
* [quad.cpp](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/quad/quad.cpp)
* [quad.f](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/quad/quad.f)
* [quad.f90](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/quad/quad.f90)

The *quad* program approximates the integral of

f(x) = 50 / pi / (2500x^2+1)

from 0 to 10, by evaluating the function at **n** equally spaced points, and multiplying the sum by (10-0)/n.

Your program should be ready to compile and run. Try to do it. I suggest you rename your compiled program *quad*. The program prints out a "wallclock time" measurement, but this is currently zero, because the program is not calling the necessary OpenMP function to measure this number. In fact, it isn't using OpenMP at all!

Your task is to **make 4 modifications to the program** so that it can take advantage of OpenMP. These changes will involve:

1. Adding an "include" or "use" statement to get the OpenMP include file.
2. Calling the OpenMP timer function *omp\_get\_wtime()* to initialize the timer measurement *wtime*.
3. Inserting two OpenMP directives just before the loop. The first directive lists private and shared variables. The second will tell OpenMP that the *total* variable is a reduction variable. (This change is the the hardest, and most important one, to do correctly!)
4. Calling the OpenMP timer function *omp\_get\_wtime()* to update the timer measurement *wtime*.

Once you've made your changes, **compile the program**. Your compile statement must now include a switch indicating that OpenMP is in use. For instance, the C program would now be compiled by

gcc -fopenmp quad.c

You set the number of threads with a command like

export OMP\_THREAD\_NUM=4

Run the program 4 times, **setting the number of threads** to 1, 2, 4 and 8, and recording the value of wall clock time. Do you see a pattern?

**A Molecular Dynamics Program**

For this exercise, pick a version of the *md* program:

* [md.c](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/md/md.c)
* [md.cpp](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/md/md.cpp)
* [md.f](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/md/md.f)
* [md.f90](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/md/md.f90)

The *md* program is a simple example of a molecular dynamics code. It randomly 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 *compute()* 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 in parallel, and faster.

The first change you must make to the program is to add a reference to the OpenMP ``include file''.

The second change will allow us to report the time taken by the big loop in the main program. Just before the loop, call *omp\_get\_wtime()* and save the value as **wtime**. Just after the loop, call*omp\_get\_wtime()* again, and update the value of **wtime**, and print its value.

Our third change is to parallelize the loop in the *compute()* routine. This is actually a nested loop. Our OpenMP directives will go just before the first of the two loop statements.

If you're using C or C++, your parallel directive should have the form:

# pragma omp parallel private (...) shared (...)

where you must place every loop variable into one list or the other (except for any reduction variables...and we will have two of those!)

If you're using C or C++, your "for" directive should have the form:

# pragma omp for reduction ( + : pot, kin )

because both **pot** and **kin** are reduction variables.

Remember that, in the "private" and "shared" lists, OpenMP only wants to see true variables. The compiler will complain if it sees the names of FORTRAN parameters or C/C++ "defined" quantities, because these are not actually variables. If you have a quantity called **pi2** showing up in this loop, make sure you know whether it is really a variable or not.

Once you have made the changes, then

1. compile your program and rename the executable to **md**;
2. set the number of threads to 1, and record the program's time.
3. set the number of threads to 2, and record the program's time.
4. set the number of threads to 4, and record the program's time.

You should see the program's execution time decreasing. If not, you might have used an incorrect OpenMP directive, or forgotten to compile your program with the OpenMP option.

If you are interested in this topic, here are two more things you can look into, some other time:

* the nested loop in *update()* could 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
* ...

We can still parallelize this set of loops, but can you see why it might not be a good idea, assuming the value of **nd** is 3?

* the *compute()* routine probably does not take as much time as the *update()* routine. (How could you verify this.) It might still be worth parallelizing the loop in that routine. However, this loop includes a reduction variable, called **dist**. Working on this loop would give you some more practice.

**Jacobi Iteration for A\*x=b**

For this exercise, pick a version of the *jacobi* program:

* [jacobi.c](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/jacobi/jacobi.c)
* [jacobi.cpp](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/jacobi/jacobi.cpp)
* [jacobi.f](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/jacobi/jacobi.f)
* [jacobi.f90](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/jacobi/jacobi.f90)

The *jacobi* program solves a linear system using the Jacobi iteration. The program already includes some OpenMP information, such as the include statement and the timing calls, but more changes are needed if the program is to run in parallel.

The problem size is set in the main program, in the variable **n**, and you can change the value of this variable to make the problem be small for debugging (n=10) or big as a better test of the timing (n=500).

We are going to modify the routine called **jacobi** which solves the linear system. Inside this routine there is an iteration, and the iteration involves a number of loops or vector operations. We can't make the iteration loop itself parallel. But inside that loop, there are several smaller loops that we can work with:

iteration loop begins

copy x to x\_old loop

compute new value of x loop

compute ||x-x\_old|| loop

iteration loop ends

We can use a single OpenMP parallel statement to apply to all three inner loops. In FORTRAN the **end parallel** directive makes it clear where we are stopping. In C and C++, this will only work if we also use a pair of curly brackets to enclose the three loops:

iteration loop begins

parallel directive private(...) shared(...)

{

copy x to x\_old loop

compute new value of x loop

compute ||x-x\_old|| loop

}

iteration loop ends

The loops are still sequential until we apply the appropriate OpenMP directive to request that they be executed in parallel. Mark each of the three loops. There is a reduction variable in the third loop. There is NOT a reduction variable in the second loop!

When you think your code is correct, compile and run it with 1, 2 and 4 threads. You should expect to see a significant improvement in time.

**ASSIGNMENT: The Heated Plate**

For the assignment, pick a version of the *heated\_plate* program:

* [heated\_plate.c](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/heated_plate/heated_plate.c)
* [heated\_plate.cpp](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/heated_plate/heated_plate.cpp)
* [heated\_plate.f](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/heated_plate/heated_plate.f)
* [heated\_plate.f90](http://people.sc.fsu.edu/~jburkardt/classes/acs2_openmp_2013/heated_plate/heated_plate.f90)

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 average change between old and new values is less than some user-specified tolerance **epsilon**.

This program expects to read the value of **epsilon** from the command line. Thus, if you were going to run the program interactively, you might type something like

./heated\_plate 0.001

As in the *jacobi* program, there is an iteration loop that we cannot parallelize, but this loop contains three computational loops, each of which can be made parallel.

*Your assignment is to modify the program to take advantage of OpenMP, and to produce a table showing the execution time required to run your modified program on 1, 2 and 4 threads.*

To get credit for this assignment, you must submit the following information to the lab instructor:

* a copy of the revised heated\_plate program;
* a table of the wall clock times for 1, 2 and 4 threads.

**This information must be received by 11:59pm, October 15th.**

You can go up one level to [the CLASSES page.](http://people.sc.fsu.edu/~jburkardt/classes/classes.html)

*Last revised on 09 October 2013.*