Tutorials | Exercises | Abstracts | LC Workshops | Comments | Search | Privacy & Legal Notice

OpenMP Exercise

Exercise 1

1. **Login to the workshop machine**

   Workshops differ in how this is done. The instructor will go over this beforehand.

2. **Copy the example files**

   1. In your home directory, create a subdirectory for the example codes and then **`cd`** to it.

      ```
      mkdir openMP
      cd  openMP
      ```

   2. Then, copy either the Fortran or the C version of the parallel OpenMP exercise files to your openMP subdirectory:

      ```
      C:        cp  /usr/global/docs/training/blaise/openMP/C/*  ~/openMP
      Fortran:  cp  /usr/global/docs/training/blaise/openMP/Fortran/*  ~/openMP
      ```

3. **List the contents of your openMP subdirectory**

   You should notice the following files. ***Note:*** Most of these are simple example files. Their primary purpose is to demonstrate the basics of how to parallelize a code with OpenMP. Most execute in a second or two.

C Files	Fortran Files	Description
omp_hello.c	omp_hello.f	Hello world
omp_workshare1.c	omp_workshare1.f	Loop work-sharing
omp_workshare2.c	omp_workshare2.f	Sections work-sharing
omp_reduction.c	omp_reduction.f	Combined parallel loop reduction
omp_orphan.c	omp_orphan.f	Orphaned parallel loop reduction
omp_mm.c	omp_mm.f	Matrix multiply
omp_getEnvInfo.c	omp_getEnvInfo.f	Get and print environment information
omp_bug1.c omp_bug1fix.c omp_bug2.c omp_bug3.c omp_bug4.c omp_bug4fix omp_bug5.c omp_bug5fix.c omp_bug6.c	omp_bug1.f omp_bug1fix.f omp_bug2.f omp_bug3.f omp_bug4.f omp_bug4fix omp_bug5.f omp_bug5fix.f omp_bug6.f	Programs with bugs

4. **Compilers - What's Available?**

   1. Visit the Compilers Currently Installed on LC Platforms webpage.

   2. Look for one of LC's Linux clusters, such as cab, zin or sierra, in the section near the top of the page. Then, click on a specific cluster name/link to see additional detail for that cluster. Note that this page shows the default compilers only. Most systems have several versions of each.

   3. Now, try the **`use -l compilers`** command to display available compilers on the cluster you're logged into. You should see GNU, Intel, and PGI compilers - several versions of each. Try any/all of these commands to narrow your search to a specific compiler:

      ```
      use -l compilers/gcc
      use -l compilers/icc
      use -l compilers/pgi
      ```

4. You can also see the Compiling OpenMP Programs section of the tutorial.

5. **Create, compile and run an OpenMP "Hello world" program**

    1. Using your favorite text editor (vi/vim, emacs, nedit, gedit, nano...) open a new file - call it whatever you'd like.
    2. Create a simple OpenMP program that does the following:
        - Creates a parallel region
        - Has each thread in the parallel region obtain its thread id
        - Has each thread print "Hello World" along with its unique thread id
        - Has the master thread only, obtain and then print the total number of threads

        If you need help, see the provided omp_hello.c or omp_hello.f file.
    3. Using your choice of compiler (see above section 4), compile your hello world OpenMP program. This may take several attempts if there are any code errors. For example:

        C:
        ```
        icc -openmp omp_hello.c -o hello
        pgcc -mp omp_hello.c -o hello
        gcc -fopenmp omp_hello.c -o hello
        ```
        Fortran:
        ```
        ifort -openmp omp_hello.f -o hello
        pgf90 -mp omp_hello.f -o hello
        gfortran -fopenmp omp_hello.f -o hello
        ```

        When you get a clean compile, proceed.
    4. Run your **hello** executable and notice its output.
        - Is it what you expected? As a comparison, you can compile and run the provided **omp_hello.c** or **omp_hello.f** example program.
        - How many threads were created? By default, the Intel and GNU compilers will create 1 thread for each core. The PGI compiler will create only 1 thread total.
    5. Notes:
        - For the remainder of this exercise, you can use the compiler command of your choice unless indicated otherwise.
        - Compilers will differ in which warnings they issue, but all can be ignored for this exercise. Errors are different, of course.

6. **Vary the number of threads and re-run Hello World**

    1. Explicitly set the number of threads to use by means of the OMP_NUM_THREADS environment variable:

        ```
        setenv OMP_NUM_THREADS 8
        ```
    2. Your output should look something like below.

        ```
        Hello World from thread = 0
        Hello World from thread = 3
        Hello World from thread = 2
        Number of threads = 8
        Hello World from thread = 6
        Hello World from thread = 1
        Hello World from thread = 4
        Hello World from thread = 7
        Hello World from thread = 5
        ```
    3. Run your program several times and observe the order of print statements. Notice that the order of output is more or less random.

This completes Exercise 1

Exercise 2

1. **Still logged into the workshop cluster?**

    1. If so, then continue to the next step. If not, then login as you did previously for Exercise 1.

2. **Review / Compile / Run the workshare1 example code**

This example demonstrates use of the OpenMP loop work-sharing construct. Notice that it specifies dynamic scheduling of threads and assigns a specific number of iterations to be done by each thread.

1. First, set the number of threads to 4:

   ```
   setenv OMP_NUM_THREADS 4
   ```

2. After reviewing the source code, use your preferred compiler to compile and run the executable. For example:

   C:
   ```
   icc -openmp omp_workshare1.c -o workshare1
   workshare1 | sort
   ```
   Fortran:
   ```
   ifort -openmp omp_workshare1.f -o workshare1
   workshare1 | sort
   ```

3. Review the output. Note that it is piped through the sort utility. This will make it easier to view how loop iterations were actually scheduled across the team of threads.
4. Run the program a couple more times and review the output. What do you see? Typically, dynamic scheduling is not deterministic. Everytime you run the program, different threads can run different chunks of work. It is even possible that a thread might not do any work because another thread is quicker and takes more work. In fact, it might be possible for one thread to do all of the work.
5. Edit the workshare1 source file and change the **dynamic scheduling** to **static scheduling**.
6. Recompile and run the modified program. Notice the difference in output compared to dynamic scheduling. Specifically, notice that thread 0 gets the first chunk, thread 1 the second chunk, and so on.
7. Run the program a couple more times. Does the output change? With static scheduling, the allocation of work is deterministic and should not change between runs, and every thread gets work to do.
8. Reflect on possible performance differences between dynamic and static scheduling.

3. **Review / Compile / Run the matrix multiply example code**

   This example performs a matrix multiple by distributing the iterations of the operation between available threads.

   1. After reviewing the source code, compile and run the program. For example:

      C:
      ```
      icc -openmp omp_mm.c -o matmult
      matmult
      ```
      Fortran:
      ```
      ifort -openmp omp_mm.f -o matmult
      matmult
      ```

   2. Review the output. It shows which thread did each iteration and the final result matrix.
   3. Run the program again, however this time sort the output to clearly see which threads execute which iterations:

      ```
      matmult | sort | grep Thread
      ```

      Do the loop iterations match the SCHEDULE(STATIC,CHUNK) directive for the matrix multiple loop in the code?

4. **Review / Compile / Run the workshare2 example code**

   This example demonstrates use of the OpenMP SECTIONS work-sharing construct Note how the PARALLEL region is divided into separate sections, each of which will be executed by one thread.

   1. As before, compile and execute the program after reviewing it. For example:

      C:
      ```
      icc -openmp omp_workshare2.c -o workshare2
      workshare2
      ```
      Fortran:
      ```
      ifort -openmp omp_workshare2.f -o workshare2
      workshare2
      ```

   2. Run the program several times and observe any differences in output. Because there are only two sections, you should notice that some threads do not do any work. You may/may not notice that the threads doing work can vary. For example, the first time thread 0 and thread 1 may do the work, and the next time it may be thread 0 and thread 3. It is even possible for one thread to do all of the work. Which thread does work is non-deterministic in this case.

This completes Exercise 2

Exercise 3

1. **Still logged into the workshop cluster?**

    1. If so, then continue to the next step. If not, then login as you did previously for Exercise 1.

2. **Review / Compile / Run the orphan example code**

    This example computes a dot product in parallel, however it differs from previous examples because the parallel loop construct is orphaned - it is contained in a subroutine outside the lexical extent of the main program's parallel region.

    1. After reviewing the source code, compile and run the program. For example:

        C:
        ```
        icc -openmp omp_orphan.c -o orphan
        orphan | sort
        ```
        Fortran:
        ```
        ifort -openmp omp_orphan.f -o orphan
        orphan | sort
        ```

    2. Note the result...and the fact that this example will come back to haunt as **omp_bug6** later.

3. **Get environment information**

    1. Starting from scratch, write a simple program that obtains information about your openMP environment. Alternately, you can modify the "hello" program to do this.

    2. Using the appropriate openMP routines/functions, have the master thread query and print the following:
        - The number of processors available
        - The number of threads being used
        - The maximum number of threads available
        - If you are in a parallel region
        - If dynamic threads are enabled
        - If nested parallelism is enabled

        NOTE: Some compilers (IBM, GNU) implement some of the necessary Fortran functions as integer instead of logical as the standard specifies.

    3. If you need help, you can consult the omp_getEnvInfo.c or omp_getEnvInfo.f example file.

4. **When things go wrong...**

    There are many things that can go wrong when developing OpenMP programs. The **omp_bugX.X** series of programs demonstrate just a few. See if you can figure out what the problem is with each case and then fix it.

    The buggy behavior will differ for each example. Some hints are provided below.

    Note: Please use the Intel (icc, ifort) compile commands for these exercises.

Code	Behavior	Hints/Notes
omp_bug1 **omp_bug1fix**	Fails compilation. Solution provided - must compile solution file.	Explanation
omp_bug2	Thread identifiers are wrong. Wrong answers.	Explanation
omp_bug3	Run-time error, hang.	Explanation
omp_bug4 **omp_bug4fix**	Causes a segmentation fault. Solution provided - note that it is a script and will need to be "sourced". For example: "source omp_bug4fix". Be sure to examine the solution file to see what's going on - especially the last line, where you may need to change the name of the executable to match yours.	Explanation
omp_bug5 **omp_bug5fix**	Program hangs. Solution provided - must compile solution file.	Explanation
omp_bug6	Failed compilation	Explanation

This completes the exercise.

Evaluation Form

Please complete the online evaluation form if you have not already done so for this tutorial.

Where would you like to go now?

- Agenda
- Back to the tutorial

---

file:///Users/jacek/Downloads/OpenMP%20Exercise.html
Last Modified: 07/27/2016 10:53:00 blaiseb@llnl.gov
UCRL-MI-133316

---

https://computing.llnl.gov/tutorials/openMP/exercise.html
Last Modified: 06/30/2016 16:32:18 blaiseb@llnl.gov
UCRL-MI-133316