Lab Sessions for OpenMP Workshop
Ontario HPC Summer School 2016 East
University of Ottawa, August 9, 2016

General Considerations:

All lab sample codes are in `~/OpenMP`. There are sub-directories that contain `C` and `F90` sample code. Different sample programs are further contained in directories below `C` and `F90`. For instance, the `hello` example resides in `~/OpenMP/C/hello` for C and in `~/OpenMP/ F90/hello` for Fortran.

Most example directories contain the following files (using *name* as an example):

- The source code `name.c` for C or `name.f90` for Fortran.
- A `makefile` with instructions for compilation and a sample run.
- In some cases, an input file `name.in` with sample input for the code.
- Sometimes there is a `solution` directory as well.

The code is supposed to be examined and in some cases modified. The `makefile`'s are there as a reminder on how to compile and execute the code. The idea is, of course, to issue the commands in the `makefile` *manually* as a practice.

Compilations:

C compiler: `gcc`
Fortran compiler: `gfortran`
C compiler (MPI macro, Intel): `mpiicc`
Fortran compiler (MPI macro, Intel): `mpiifort`
Pre-processor: `cpp`
OpenMP flag: `-fopenmp`
Optimization flag: `-O3`
Renaming executable: `-o name.exe`
Compilation only: `-c`
Pre-processor flag for keyword KEY: `-DKEY`
Use math library: `-lm`

For instance:

C code `name.c` using OpenMP, compiled into `name.exe`, using math lib:
```
gcc -o name.exe -fopenmp -O3 name.c -lm
```
Fortran code `name.f90` using OpenMP, compiled into `name.exe`
```
gfortran -o name.exe -fopenmp -O3 name.f90
```

Displaying/Editing code:

Editors on the system:

- `vi` Standard Unix editor, requires practice, only useful to people familiar with Unix
- `nano` simple text editor, almost self-explanatory, ideal for new users

Executing OpenMP code from command line:

From a bash login shell, simply set `OMP_NUM_THREADS` environment variable and type name of the executable preceded by "`./`":

```
export OMP_NUM_THREADS=4
./name.exe
```

or

```
OMP_NUM_THREADS=4 ./name.exe
```

Input can be "redirected" from the keyboard to a file by the "`<`" operator, output to the screen by the "`>`" operator, for instance

```
OMP_NUM_THREADS=4 ./name.exe <name.in >name.out
```

Executing Hybrid (MPI/OpenMP) code from command line:

Combine the above instructions for OpenMP and MPI
Due to thread-safety issues with Gnu compilers/OpenMPI, it's best to use the Intel compilers:

```
use icsmpi
mpiicc –o name.exe –openmp –O3 name.c –lm
(mpiifort –o name.exe –openmp –O3 name.f90)
OMP_NUM_THREADS=3 mpirun -np 2 ./name.exe
```

for 2 processes with 3 threads each.

<u>Individual Labs:</u>

OpenMP Session 1:

1. **hello**: Simple hello-world example for multiple threads. Compile with and without -openmp flag and run with different OMP_NUM_THREADS settings.

OpenMP Session 2:

1. **default**: Used to demonstrate the default shared/private status of variables. Try out what happens if you re-declare some of them, for instance the loop indices.
2. **testomp**: Simple OpenMP test program. Insert a `schedule(runtime)` clause and see what happens when you reset the `OMP_SCHEDULE` variable.
3. **rootsum**: Sum-of-square-roots example. Run with various numbers of processes. Use "`time -p`" command to obtain runtimes. Compute speedup and efficiency over a range of process numbers and make a scaling plot.
4. **workload**: Similar to "`testomp`" but uses varying workloads for the iterations. Try to optimize the execution time through manipulation of `OMP_SCHEDULE`
5. **depend**: example of a data dependency that destroys the loop parallelization. Try to find the cause for the dependency and fix it. The solution to the "depend" problem is in the "solution" directory, but try to find it on your own. Modify and re-run until the result is correct, i.e. agrees with result from compilation without -openmp or serial run.

OpenMP Session 3:

1. **sqroots**: awkward implementation of sum-of-square-roots example through arrays. Run with various numbers of processors and determine scaling. Use scaling results to estimate serial fraction of code based on Amdahl's law.
2. **minimum**: finding the minimum of a simple function on a mesh. Uses locks to protect minimal value. Compile with or without lock, use keyword LOCK to do so (see makefile). Try to trigger the resulting race condition when the lock is not used. May or may not happen.
3. **slaves**: simple "all-slaves" model with simulated workload. Determine scaling behaviour for various random execution times. For instance, try running it with execution times evenly between 1-5 seconds, then with "either 1 or 5 seconds".
4. **mandel**: runs a scan through the the top (*Im(c)* positive) half of the Mandelbrot set on a grid using three parallel approaches: static parallel loop, dynamic parallel loop and all-slaves model. The dynamic approaches are more efficient because they address the inherent load imbalance.
5. **erf**: write a program for the computation of the error function from scratch. Use rectangle rule for numeric integration. Once you got it, see how it scales. The solution is in the code provided, but don't peek. Use "template" as a starter
6. **race**: an example for a race condition between data update and referencing that requires the use of locks to be fixed.