OpenMP Exercises

These exercises will introduce you to using OpenMP for parallel programming. There are four exercises:

1. OMP Hello World
2. Worksharing Loop
3. OMP Functions
4. Hand-coding vs. MKL

To begin, log onto an interactive node of Stampede using the SSH Secure Shell client or ssh:

SSH Client:

All Programs | ClassFiles | SSH Secure Shell | Secure Shell Client
Host Name: **stampede.tacc.utexas.edu**

ssh:

```
ssh  <user-name>@stampede.tacc.utexas.edu
```

Untar the openmp_lab.tar file (in ~tg459572) into your current directory:

```
$ tar xvf ~tg459572/LABS/lab_openmp.tar
```

Change your working directory to the new **lab_openmp** directory:

```
$ cd lab_openmp
```

OMP Hello world

The *Hello world* example is very short, so for convenience we will run it on the interactive node where you're logged in. There are other sections of this lab that will run longer and will involve measuring performance; those will be done on dedicated nodes through the batch system.

Look at the code in *hello.c* and/or *hello.f90*. This code simply reports OpenMP thread IDs in a parallel region.

Compile hello.c or hello.f90 using the makefiles provided:

```
$ make hello_c
- or -
$ make hello_f90
```

Specify 3 threads:

```
$ export OMP_NUM_THREADS=3
```

Run:

```
$ ./hello_c
- or -
$ ./hello_f90
```

Use the makefile to run with 2 to 16 threads:

```
$ make run_hello_c
- or -
$ make run_hello_f90
```

Worksharing Loop

Look at the code in file daxpy.f90. The nested loop repeats a simple DAXPY type of operation (double-precision ax+y, scalar times vector plus vector). It is repeated ten times in order to gather statistics on performance. Parameter N determines the size of the vector: N=48*1024*1024 is the default.

Compile and run daxpy.f90:

```
$ make daxpy
$ export OMP_NUM_THREADS=3
$ ./daxpy
```

A more detailed comparison will be done in a later section.

OMP Functions

Look at the code in work.f90. Threads perform some work in a subroutine called pwork. The timer returns wall-clock time.

Compile work.f90 and run it with one set of threads to verify that it built properly:

```
$ make work
$ export OMP_NUM_THREADS=1
$ ./work
wall-clock time = 
```

Now look at work_serial.f90. We no longer use omp_lib, and numeric values are substituted for the calls to OMP_ functions. The OpenMP directives are ignored because the code is not compiled with OpenMP.

Compile and run work_serial.f90:

```
$ make work_serial
$ ./work_serial
wall-clock time = 
```

As expected, this code runs with nearly the same speed as the work.f90 code with 1 thread. The overhead due to OpenMP is minimal in this case, because all threads are forked at the beginning and the parallel region contains all the work.

Is the overhead worth it when running on multiple threads? Run work again, using three threads:

```
$ export OMP_NUM_THREADS=3
$ ./work
wall-clock time = 
```

Ideally, you saw that the serial time was a little faster than parallel with one thread, and that the run with three threads had overhead, but overall showed good speedup in spite of the overhead. Seeing good results is unlikely on a shared login node. Run these again on a dedicated node using the batch system (as shown in the Environment talk) to see meaningful results. (Optional)

Hand-coding vs. MKL

Look at the code in file daxpy2.f90. The nested loop performs a DAXPY operation for each outer loop. The DAXPY routine comes from the Intel MKL library, which is already parallelized with OpenMP (!). All you have to do is set the value of OMP_NUM_THREADS. Compare the performance to what you saw in the earlier exercise using the hand-coded OpenMP version of DAXPY.

Compile and run daxpy2.f90, using 3 threads:

```
$ make daxpy2
$ export OMP_NUM_THREADS=3
$ ./daxpy2
```

Next, prepare to run a batch job. Edit the file *job* to put the account number, *TG-TRA120006*, after the -A flag:

```
$ vi job
```

Submit the batch job; this job makes more detailed comparisons on a dedicated node:

```
$ sbatch job
```

Check on your job status:

```
$ showq -u
```

Examine your output file:

```
$ cat myOMP.o[Job ID]
```

Note 1: the number of OpenMP threads can exceed the number of physical cores.

Note 2: the makefile has additional various interactive run_ and plot_ options; the run options can be done in batch, and the plot_ options can be done interactively.