Iowa High Performance Computing Summer School 2011
OpenMP Programming Exercise Set

Thursday, June 2, 2011

These exercises may be completed in C or Fortran.

1. **Hello World in OpenMP**

   Write an OpenMP version of the parallel "Hello World" program you wrote yesterday and run it on Helium to observe its output. Set the environment variable `OMP_NUM_THREADS=8` to use 8 threads. You will need to define a parallel region and call OpenMP library functions to get the current thread ID and the total number of threads. Have your program produce output like this: "Hello World. I am thread 2 of a team of 8 threads."

   Reminder:
   To compile with OpenMP, use `-fopenmp` (GNU compilers) or `-openmp` (Intel compilers).

   (a) Study the order with which the output appears on the screen. Is it reproducible / predictable?

   (b) Modify your code to produce ordered output. *Hints*: You won't need any additional OpenMP directives to achieve this, but may need to introduce private and shared variables. Come up with a way of keeping track of which thread has already said hello and make sure that thread with ID n speaks up only thread with ID $n-1$ has done so. Thread 0 starts.

2. **Computing π**

   π can be computed (perhaps not in the most efficient way) via the Gregory-Leibniz series:

$$\pi = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} \; .$$

   (a) Implement the Gregory-Leibniz series in a serial code. Make sure it works (i.e., with 500 iterations you should get something not too far off from the real value of π. Add code that computes the absolute and relative error of your result to $\pi = \mathrm{acos}(-1.0)$.

   (b) Parallelize your code using OpenMP. Run it with `OMP_NUM_THREADS=1` (serial) and `OMP_NUM_THREADS=8` and measure how long it takes to run, using `time YourProgram` on the command line. Is there any speed up? What if you instead terminate the series at $n = 1000000000$. How good is your estimate of π now?

3. **Multiplying Matrices**

   Implement a program that multiplies two $n \times n$ matrices A_{ij} and B_{ij} according to

$$C_{ij} = (AB)_{ij} = \sum_k A_{ik} B_{kj} \; .$$

   For this, set up three $n \times n$ arrays. To get some data, set $A_{ij} = \sin(ij)$ and $B_{ij} = \cos(ij)$, then implement the multiplication loop.

(a) First implement this as serial code and run this with $n = 4000$. Make sure to compile with the `-O2` compiler flag to activate compiler optimization. Use `time YourProgram` on the command line to measure the run time of your program. How long does it take? Now look up how to use timing routines in the language your code is in (`cpu_time` in Fortran, use `clock()` in `time.h` in C) to find out which part of the code requires most of the time. Explain the result. How does the time needed by the different parts of your code increase with n?

(b) Now parallelize your code with OpenMP. Focus on the part of the code that requires most of the CPU time. Use $n = 4000$ and run your code with 1 to 8 threads on Helium and measure (using `time` on the command line) the time it takes for each thread count. Repeat this exercise for $n = 3000$ and $n = 5000$ Make a table of your performance measurements. What trends do you observe?

4. **Advanced: OpenMP enhancement of** `HYDRO`.

The parallel scaling of MPI programs like `HYDRO` typically breaks down when run on a large numbers of cores (with one MPI process per core), since communication eventually takes more time than computation. The exact number of cores at which this happens depends on the code and on the computer hardware it is run on.

Modern supercomputers like `helium` have many cores per physical node that have access to shared memory. **Hybrid Parallelism** is an approach that combines OpenMP and MPI to reduce the number of MPI processes that communicate. For example, on `helium`, it would be possible to have only one MPI process per node, but use OpenMP to execute the code with 8 OpenMP threads on the memory owned by the single MPI process. This will instantly cut down the number of MPI processes and the MPI communication overhead by a factor of 8!

(a) Analyze the example code `HYDRO` to see in which parts/loops of the code most of the work is done. OpenMP-parallelize these loops. Run the code with a small problem once using 2 MPI processes with one thread each and once using 1 MPI process and 2 OpenMP threads. Make sure that the output agrees to machine precision.

(b) Perform a strong scaling test for a large problem on `helium`. Make sure to pick a problem size that completes within $\sim$5 minutes using 8 cores and fits into a single node's memory. First do this with pure MPI. Run the code on 8, 16, 32, 64, 128, 256, and 512 cores and record the time it takes to run on each of these core counts.

What is the optimal number of cores for this problem?

(c) Now use 2 OpenMP threads per MPI process. This translates to only 4 MPI processes per physical node (and a total of 8 threads running on the node). For this, you will need to modify the standard way in which you launch parallel jobs on `helium`. Either look this up on the internet or ask the instructors. Again run the code on 8, 16, 32, 64, 128, 256, and 512 cores (remember, 512 cores now corresponds to only 256 MPI processes!). Record the time it takes to run on each of these core counts.

Repeat this with 4 OpenMP threads per MPI process (2 MPI processes per node) and with 8 OpenMP threads per MPI process (1 MPI process per node).

Compare your new scaling results with each other and with your pure MPI results. What is your conclusion?