EC527: High Performance Programming with Multicore and GPUs
Programming Assignment 6

Objective

To learn about and practice using OpenMP. At the end of this assignment you should have a good understanding of the limits of Shared Address Space parallelization going in two directions: (i) what kind of performance you can expect using OpenMP, the most direct method of parallelizing your code, and (ii) the kinds of codes that are likely to be limited by basic overhead inherent in OpenMP.

Prerequisites (to be covered in class or through examples in on-line documentation)

Programming – Basics of OpenMP and PThreads.

Eng-Grid note: It is important to work on a machine that is not busy with another person's work. See notes at the end of "`eng-grid.txt`" in Assignment 5. (That file was updated for this assignment).

Assignment

Part 1: OpenMP basics

1a. "Hello World!" (or "`ell HWoo!dlr`")

Reference code: **`test_omp.c`**

The reference code prints "Hello World!" by having each character printed by a different thread. The `omp parallel` and `omp sections` pragmas are used.

Task: Write a similar function, but this time use **Parallel For**. Have each iteration print a different character.

Deliverable: Your code and your output.

1b. Parallel For

Reference code: **`test_omp_for.c`**

The reference code has three different functions with two different versions each: serial baseline and OpenMP. The three functions are, respectively, compute bound, memory bound, and neither (overhead bound).

Note that this program prints out times *in* ***seconds****, not cycles.* (Measuring cycles is fairly meaningless when a program spends part of its time in a parallel section, on modern machines where multiple cores are running *at different clock speeds* and accumulating "cycles" at different rates.)

Task: (Analogous to the *"find pthreads overhead"* part of the pthreads tutorial assignment) Find the intrinsic OpenMP overhead. That is, how long does it take to do operations necessary to setup and run the parallel loop. When using OpenMP you do not explicitly pass parameters to threads, and there is no barrier at the end of the loop, but these operations are actually happening! Also as in the pthreads tutorial, you need to explore array sizes small enough to see the break-even point.

OpenMP knows abut an environment variable OMP_NUM_THREADS. You can run your program with a different number of threads by prefixing it with an environment variable settings, like this:

```
OMP_NUM_THREADS=4 ./test_omp_for
```

Compare OpenMP with pthreads (which has worse overhead and by how much?)

Deliverable: Give the results of your calculations and a brief explanation.

1c. MMM, 2 loop versions (serial vs OpenMP)

Reference code: **`test_mmm_inter_omp.c`**

The reference code has two 2 loop functions for MMM: `ijk` and `kij`. Each has two versions, a serial baseline and an OpenMP version.

As before, this program prints out times *in **seconds**, not cycles.*

We will experiment with two things, "shared" lists and `pragma` placement

Start by generating results for a variety of matrix sizes. As always, the sizes you test should span a wide range: at least from fairly small (all 3 arrays fit in L2 cache) to really large (all 3 arrays do not fit even in L3 cache).

Loop indices are supposed to be private variables (by default). Test this out by moving the loop indices out of the "`private(...)`" list and into the "`shared(...)`" list, and rerunning the code.

It may seem obvious that the best place to put the pragma "`parallel for`" is around the outermost loop. Test this assumption by moving the pragma: first to the middle loop, and then the inner loop. For each, rerun the code.

Deliverable: Results and brief explanation.

Part 2: OpenMP on real programs

For both of these, the goal is to take a naïve approach to using OpenMP and see what happens (e.g., compared with the approach in Part 1.3 and the threaded approach). If you have time see if you can optimize the OpenMP directives.

2a. SOR

Task: Use OpenMP to parallelize test_SOR from Assignment 6 part 2.

2b. MMM

Task: Parallelize your best previous MMM code using OpenMP.

Deliverables: your code, results, and a brief explanation.