ECE 432-532 Lab exercise - 11

Fall 2017

Part 1:

When threads start accessing shared variables, it is all too easy to create race conditions. As a first example, download and run *private.c*, and follow the directions at the beginning of the source file. As you work through them, find the answers to these questions:

1. What shared variable is the source of the conflict?

2. Is it a write-write conflict, a read-write conflict, or both?

3. How does OpenMP's `private` clause resolve the conflict?

For race conditions where the code contains a critical section, the *Mutual Exclusion* pattern can be used. OpenMP provides two different mechanisms for this pattern: the `atomic` mechanism and the `critical` mechanism.

You can explore the `atomic` mechanism by downloading and running *atomic.c*. Follow the directions at the beginning of the file, and map the behavior you observe to the source code producing it.

You can explore the `critical` mechanism by downloading and running *critical.c* and *critical2.c*. Follow the directions at the beginning of the file, and figure out how the source code is producing the behavior you observe.

If you compare the *Mutual Exclusion* pattern in OpenMP against the same pattern in pthreads, it should be evident that OpenMP's mechanisms are much simpler and easier to use than those of pthreads. This simplicity and comparative ease of use are what have made OpenMP so popular.

Part 2:

The next pattern is the *Parallel Task* pattern, which OpenMP implements using its `sections` directive. To see how it works, download and run *sections.c*. Note that each section can be performing a different function, so if a problem can be decomposed into tasks that can run in parallel, the `sections` directive provides a way to solve that problem.