**CSC 338 Parallel and Distributed Computing**

**Exercise No. 14, November 30, 2017**

**Introduction to OpenMP**

**Goal**

Learn thread programming with OpenMP

**Background**

Pthreads provides a low-level API for thread programming; OpenMP (Open MultiProcessing) is a higher level API for writing multiprocessing programs using threads.

**Procedure**

1. *omp\_hello.c,* in the exercise folder, is a simple program that starts a number of threads and prints a message from each one. Note that the mechanism for creating threads is a *pragma.* A pragma is a preprocessor directive in C or C++. OpenMP pragmas start with #pragma omp. The parallel directive specifies that the following structured block is executed by multiple threads.

Prior to the parallel directive, the program executes a single thread—the process stared by the operating system. The parallel directive causes an additional thread\_count – 1 threads to begin execution. The original thread is the *master;* the new threads are *slaves.* Together, the threads are known as a *team.*

When the block of code is completed (in this case, the call to Hello()), there is an implicit barrier—the threads all wait until they have all returned from the call to Hello(), then the slaves terminate. The master goes on to execute the return statement. Note that we don't have to explicitly start the threads or terminate them with a join command. Nor do we have to explicitly set up barriers.

Each thread has its own stack. The local variables in the Hello() function are allocated from the stack; they are said to have *private* scope. This would also be true of any variable defined in the parallel block. In OpenMP programming, the variables defined outside the parallel block, in main(), are accessible to all threads—they have *shared* scope.

How does each thread get its rank? How does each thread get the number of threads?

Omp\_get\_thread\_rank() and Omp\_get\_thread\_count()

Compile and execute omp\_hello.c with varying numbers of threads.

1. Now look at *omp\_trap2a.c.* This program uses the trapezoidal rule to calculate the area under a curve (in other words, it sums the areas of some number of trapezoids imagined to be under the curve). Note the structured block after the parallel directive. Find the critical section and note how easy it is to specify a critical section in OpenMP. What is the directive for specifying a critical section?

# pragma omp critical

1. Now look at omp\_trap2b.c. This program is functionally the same as omp\_trap2a but it uses a reduction variable. A reduction is a computation (such as addition or multiplication) in which the operator is repeatedly applied to a sequence of operands to obtain a single result (in this case, it's the sum of all the trapezoid computations—essentially a global sum). Furthermore, all the intermediate results are stored in the same variable—the reduction variable. The point is that the reduction clause causes OpenMP to treat the reduction operation as an atomic action—only one thread at a time can add to the reduction variable. In other words, it's essentially a critical section. How is the reduction variable specified?

# pragma omp parallel num\_threads(thread\_count) \

reduction(+: global\_result)