**CSC 338 Parallel and Distributed Computing**

**Exercise No. 15, November 30, 2017**

**More OpenMP**

**Goal**

Learn how to use the parallel for directive in OpenMP

**Background**

OpenMP allows parallel control of for loops with some simple directives

**Procedure**

1. A program examined in the previous exercise *(omp\_trap2b)*, used a reduction to control access to a variable used in a global sum. *omp\_trap3.c* carries this idea further by using the *parallel for* directive to parallelize a for loop. Find this directive in *omp\_trap3.c.* What line is it?

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

reduction(+: approx)

Note the use of a reduction variable in the loop. Adding values to approx constitutes a critical section that can only be accessed by one thread at a time.

But the point is parallelizing the for loop. The parallel for directive forks a team of threads to carry out the loop. The split is system dependent but most implementations, by default, use block partitioning, in which the first n/thread\_count iterations are assigned to thread 0, the next n/thread\_count iterations are assigned to thread 1, and so forth.

The variables in a parallel for directive are shared with one exception: the loop variable (*i,* in this case) must be private to each thread. Think about it—if thread 1 modifies the loop variable used by thread 0, we'll be in a real pickle.

There is one caveat about parallelizing for loops: the parallel for directive will only work if the number of iterations can be determined from the form of the loop. While loops and do-while loops cannot be parallelized, nor can a for loop with a break inside its body. In fact, the for loop must be in *canonical form—*it must be in the form

for (index = start; index [< | <= | >= | >] end; index[++ | -- | += incr | -= incr])

The loop variable cannot be modified in the body of the loop. Can you modify the parallelized for loop in *omp\_trap.c* so the compilation fails? What message do you get?

error: break statement used with OpenMP for loop

break;

1. One problem that can occur with the parallel for directive is a *loop-carried dependency.* Consider the program *omp\_fibo.c.* This program prints Fibonacci numbers calculated in parallel by some number of threads. Compile and execute this program with two or four threads; calculate 8 or 10 numbers. Are the results correct? If they are, try using more threads. The insidious aspect of this program is that it will compile successfully and will sometimes, but not always, produce correct results.

NO!

Consider the calculation inside the parallel loop: fibo[i] = fibo[i-1] + fibo[i-2]. The problem is that we are calculating a new result from results calculated in previous iterations—but remember, the “previous” iteration may have been assigned to a different thread and that result may not have been calculated yet.

We can find loop-carried dependencies by looking for variables in the loop body that are calculated in one iteration and updated or used in updating a variable in another iteration. These loops cannot be successfully parallelized by OpenMP.

1. Normally, the default partitioning (block partitions, in most implementations) is suitable. But consider a loop in which the computation time is relative to the loop variable—in other words, when *i* is 2 the computation takes twice as long as when *i* is 1, when *i* is 3 the computation takes 3 times as long as when *i* is 1, etc. In that case a cyclic partition is better—for example, with 3 threads, thread 0 performs the first iteration, thread 1 performs the second iteration, thread 2 performs the third iteration, thread 0 performs the fourth iteration, and so forth.

The program *omp\_sin\_sum.c* performs such a calculation—*f(i)* calls the sin function *i* times, so *f(2)* will take twice as long as *f(1), f(3)* will take 3 times as long as *f(1),* etc.

To control the loop partitioning, we can use the *schedule* clause. There are several different types of schedule. For a *static* schedule, the system assigns chunks of iterations to threads in a round-robin fashion. The size of the chunk depends on the chunksize argument. For example, assuming 3 threads, schedule(static, 1) assigns iterations as follows. Thread 0: 0, 3, 6, 9; thread 1: 1, 4, 7, 10; thread 2: 2, 5, 8, 11, etc. This is a cyclic partiion. shedule(2) assigns thread 0: 0, 1, 6, 7; thread 1: 2, 3, 8, 9; thread 2: 4, 5, 10, 11, etc. To get a pure block partition, use schedule(n/thread\_count), where n is the number of iterations.

Other schedule types are *dynamic* and *guided,* in which threads complete a chunk then ask for another chunk, and *runtime,* which uses an environment variable (OMP\_SCHEDULE) to control partitioning.

Compile *omp\_sin\_sum.c* and run it with one thread and 10,000 iterations. Then compile another version that uses block partitioning, schedule(static, n/thread\_count) (give it a different name—say, *omp\_sin\_sum2).* Compile yet another version, using another name, that uses cyclic partitioning, schedule(static, 1). Try the latter two versions using 2 and 4 threads and 10,000 or more iterations. Can you see a difference in the time taken by the different partitions?

YES!