CIS 372 INTRODUCTION TO PARALLEL PROGRAMMING
Spring 2018
OpenMP, Lab 5

Objectives

- Compile, execute, and add basic OpenMP commands to data parallel programs
- Understand basic concurrency problems in shared memory programming
- Apply correct pragmas to address concurrency for common patterns
- Create an OpenMP data parallel version of sequential programs

Instructions:

Review the in class exercise for getting started with OpenMP and chapter 5 of your textbook.
Clone lab 5 from your github classroom account.
Complete each part. Submit a writeup that contains timing output and answers to prompts in green. **Submit all code produced**.
Additional resources about OpenMP can be found at: http://openmp.org/wp/resources/

Task

You will convert 4 different sequential programs to parallel programs using OpenMP. Each program requires you to add correct # directives and each include a sequential check to verify correctness. We have also included timing around the solving portion of the program and provided estimated timing information so that you know when you have done a good enough job of parallelization.

Note: The flags for the compute devices on our system are a little bit non-intuitive. SLURM gives an option for number of cores, -c. When used on our system these map to hyperthreaded cores on the Intel devices, so -c 16 actually gives us only 8 physical cores. Because most of the tasks we are running are compute intensive, the hyperthreading doesn't benefit us (i.e. running -c 16 with 8 threads gives about the same performance as -c 16 with 16 threads because there are only 8 physical cores). The tests use a 2:1 ratio of cores to threads so that each physical core is reserved for a single thread.

Part 1: using parallel for

Convert the matrix-vector multiplication program into an OpenMP parallel version. Included is a sequential version, **vmatrixmult.c**, that has the correct include for OpenMP and timing/correctness testing built-in. You only need to modify the code in the **`matrix_vector_multiply_by_row`** function and if done correctly can be accomplished using parallel for in one #pragma.

Once you have completed your implementation, test it for correctness (run it with n < 100 and it automatically checks for you). Now test the timing to ensure that your parallel version performs as expected:

```
make vmatrixmult-run-part1
```

```
GOMP_CPU_AFFINITY=1-16 srun -c 2 vmatrixmult.out 20000 1
Computed matrix-vector multiplication result in 1.345159 seconds
GOMP_CPU_AFFINITY=1-16 srun -c 4 vmatrixmult.out 20000 2
Computed matrix-vector multiplication result in 0.944280 seconds
GOMP_CPU_AFFINITY=1-16 srun -c 8 vmatrixmult.out 20000 4
Computed matrix-vector multiplication result in 0.533415 seconds
GOMP_CPU_AFFINITY=1-16 srun -c 16 vmatrixmult.out 20000 8
Computed matrix-vector multiplication result in 0.282446 seconds
```

Describe how the each thread works to contribute to the solution in the parallel version. Compute and describe the speedup achieved (using the single thread 2 core timing as the baseline "sequential" time).

Part 2: using a private local variable and manual reduction

Convert the histogram computation program into an OpenMP parallel version. Included is a sequential version, **histogram.c**, that generates a random number array and then computes a histogram by counting how many of the random numbers fall into different bins based on their range. You will need to use a parallel for in addition to private local variables and manually combining results (a manual reduction) into the array.

Once you have completed your implementation, test it for correctness (run it with n < 100 and it automatically checks for you). Now test the timing to ensure that your parallel version performs as expected:

```
make histogram-run-part2
GOMP_CPU_AFFINITY=1-16 srun -c 2 histogram.out 1000000000 10 1
Computed histogram result in 4.735366 seconds
GOMP_CPU_AFFINITY=1-16 srun -c 4 histogram.out 1000000000 10 2
Computed histogram result in 2.738467 seconds
GOMP_CPU_AFFINITY=1-16 srun -c 8 histogram.out 1000000000 10 4
Computed histogram result in 1.540053 seconds
GOMP_CPU_AFFINITY=1-16 srun -c 16 histogram.out 1000000000 10 8
Computed histogram result in 0.843695 seconds
```

Describe how the each thread works to contribute to the solution in the parallel version. Compute and describe the speedup achieved (using the single thread 2 core timing as the baseline "sequential" time).

Part 3: using a thread local variable (threadprivate)

Convert the sequential merge sort program, **sort.c**, by parallelizing the merge_all function. This is the same merge sort algorithm from earlier this semester. Be careful -- because each thread uses R as a global variable, you must tag it as threadprivate (so that each thread has its own copy) and then make sure to allocate and deallocate from inside of a block with #pragma omp parallel. Make sure you only parallelize the loop in merge_all -- don't parallelize the recursive step.

NOTE: Your program only needs to work for data arrays of length that are powers of 2.

Once you have completed your implementation, test it for correctness (run it with n < 1000 and it automatically checks for you). Now test the timing to ensure that your parallel version performs as expected:

```
make sort-run-part3
GOMP_CPU_AFFINITY=1-16 srun -c 2 sort.out 16777216 1
Total time to solve with 1 OpenMP threads was 3.957640
GOMP_CPU_AFFINITY=1-16 srun -c 4 sort.out 16777216 2
Total time to solve with 2 OpenMP threads was 2.691709
GOMP_CPU_AFFINITY=1-16 srun -c 8 sort.out 16777216 4
Total time to solve with 4 OpenMP threads was 1.522621
GOMP_CPU_AFFINITY=1-16 srun -c 16 sort.out 16777216 8
Total time to solve with 8 OpenMP threads was 1.006932
```

Describe how the each thread works to contribute to the solution in the parallel version. Compute and describe the speedup achieved (using the single thread 2 core timing as the baseline "sequential" time).

<u>Part 4: using scheduling options</u>

Convert the sequential **isprime.c** program by parallelizing the count_primes loop using a parallell for. Once you have completed your implementation, test it for correctness (run it with n < 1000 and it automatically checks for you). If done correctly for the default schedule (without extra flags) you should see approximately the following times:

```
make isprime-run-part4
GOMP_CPU_AFFINITY=1-16 srun -c 2 isprime.out 1 10000000 1
Found 664579 primes between 1 and 10000000 in 3.333582 seconds
GOMP_CPU_AFFINITY=1-16 srun -c 4 isprime.out 1 10000000 2
Found 664579 primes between 1 and 10000000 in 2.213532 seconds
GOMP_CPU_AFFINITY=1-16 srun -c 8 isprime.out 1 10000000 4
Found 664579 primes between 1 and 10000000 in 1.285369 seconds
GOMP_CPU_AFFINITY=1-16 srun -c 16 isprime.out 1 10000000 8
Found 664579 primes between 1 and 10000000 in 0.732583 seconds
```

Now try to add different schedule flags (see section 5.7 of your textbook for static, dynamic, and guided options). Experiment until you find a setting which will compute 1 to 10,000,000 with 8 threads (16 cores) in under 0.5 seconds. Submit the code for your version that is under 0.5 seconds.

Describe how the each thread works to contribute to the solution in the parallel version. Compute and describe the speedup achieved (using the single thread 2 core timing as the baseline "sequential" time).

<u>**Grading**</u>

4 parts, each 25 points for a total of 100 points:
(15 points each) = Implementation of OpenMP pragmas that meet correctness check and performance check
(5 points each) = Writeup description of how each thread works
(5 points each) = Computation and writeup description of speedup achieved