Report on Project 1

for SWE 594, Multicore Programming, instructed by Can Öztüran;
project by Ali Kutlu Durşen and Taner Eşme

1. Assignment

For this project, we were to implement an OpenMP program that found prime numbers in an interval defined by input value M. We were specifically supplied with an algorithm to use: each number is to be compared against prime numbers found so far, until squareroot of the number is passed.

2. Problems With Parallelization and Ways To Overcome

Primary problem is the algorithm itself: this process cannot be parallelized for unbounded numbers (at least, not without some manipulation). That is because deciding if a candidate is prime or not depends on previous candidates, therefore if we cannot guarantee all previous candidates to be classified correctly, then we cannot properly classify current candidate. This is, essentially, a sequential algorithm.

Fortunately, number theory allows a shortcut: every composite number has a factor up to square root of itself. Since our problem description bounds the numbers to be checked, we can then suggest a guaranteed list of primes to compare against. Once we have that, we can then check the remaining numbers in parallel. (This can also be used to parallelize the unbounded numbers problem: simply square the previous bound ad infinitum.) Of course, as described above, this process of finding primes up to square root needs to be sequential.

Having solved that, a more practical problem arose: when recording the newfound primes, if threads cannot act in concert, then various exceptions occur: multiple write commands at same memory location. There are ways to solve it: we can have threads each write into an array slot that is equal to their thread ID, mod total number of threads. Another option, and the one we went with, is each thread having a dedicated data container that only they write into, which results in having total number of threads and one (‘and one’, because we have a data container containing the primes found in the first phase of running sequantially up to square root.) Then it becomes a routine process of combining and sorting numbers, which we will do but not measure in our timing.

3. Program Input and Output

Program will simply accept an integer as upper bound of the interval and find primes less than it 30 times: we’re running each configuration of thread scheduling and chunk size 5 times: on 1, 2, 4, 8 and 12 threads. We’re running over a set of 6 configurations: dynamic, static and guided scheduling and chunk sizes 50 and 100 for each. Various other configurations could be had, but we believe 6 is a reasonable amount for demonstration purposes.

Program’s output will be a .csv file named “`results.csv`”, located in the same directory with the program. It’s a list of 7 rows: 1 title row and 6 row for each configuration. Its fields list the upper bound (which was input); scheduling type;

chunk size; times spent to find primes in 1, 2, 4, 8, 12 threads respectively; speedup for 2, 4, 8 threads respectively. Note that, for this particular implementation, speedup is calculated in comparison to single thread time and not the best-sequential-algorithm time.

Prime numbers themselves are not part of the output: that is mainly because each time we run the program we generate the same list 30 times. Printing it either on screen or to a file would be programatically simple but a bad user experience. Still, there is one line of code in source file that is commented out, which, if desired, can be uncommented so that our program will print the primes on screen. Those tests are advised to be held with small inputs to make viewing managable.

4. Performance and Results

Interestingly, for small inputs multi-threaded runs do not offer any speedup. That may be due to small overheads becoming the dominant factor in a very quick process. For example, primes less than 121000 are found in around 0.05 seconds so we can suggest inputs around 100 or 1000 are almost instantaneous and any overhead due to thread synchronization and scheduling is the main time cost.

A more curious fact is that, our test of input 121000 offered significant speedup *but* that speedup remained more or less the same for different thread counts; as opposed to higher speedup for greater thread count as expected. Reasons why that may be the case elude us, but one can suggest if higher input turned slowdown into speedup, then maybe even greater inputs can cause higher speedup for more threads used. An even more optimistic suggestion is that our test machine has 2 real cores, explaining why speedup is bound by 2. If that is the case, a higher-end systems can possibly not even experience this.

Overall, it is safe to conclude that this project was a success on several fonts, first and foremost being hands-on implementing and troubleshooting an OpenMP program.