Assignment 3

David Warren
University of Alabama
dlwarren2@crimson.ua.edu
CWID: 11703777

I. Introduction

This report reflects the observations and analyses of *Parallel Sieve of Eratosthenes for Finding All Prime Numbers within* 10^{10}.

II. Ease Of Use

A. CPU and Compiler Information

CPU used was Intel Xeon Gold 4180 CPUs on normal nodes to gather performance metrics. Compiler used was mpicc 3.2.1. Optimization flag was set to default -O0 least optimized. Reduces compilation time and make debugging produce the expected results.

B. Using the Makefile

- "Make" to compile *sieve_part1.c, sieve_part2.c, sieve_part3.c*
- "Make clean" to remove binary and executable files
- "Make submit" to all jobs 32, 64, and 128 processes for all parts

III. Part I

A. Parallel Sieve of Eratosthenes within 10^{10}

This is a basic implementation of the parallel sieve of Eratosthenes given that allows for *n* values larger than integer values. The changes were simple in that all instances of integers (apart from process and id variables) were changed to long long integers. Relevant type casting and type dependent functions were also changed in the code. Below are results from differing process values in the job implementation (32,64,128). 32 processes averaged around 15.1s, 64 around 9.3s, and 128 around 5.09s.

Figure (1) Part 1 implementation average times in seconds over 10 runs

32 processes	64 processes	128 processes
15.1042289	9.2998309	5.0929717

IV. Part II

A. Excluding Even Integers

This part included slightly more complex changes. First, low_value is changed to start at 3. *n* will be halved since we are excluding even numbers so high_value and low_value will be changed to reflect that. In size, however we had to multiply by 2 in high_value low_value to get the correct values, so now since the size is halved, all we have to do is simply divide size by 2. In the actual parallel algorithm, we must adapt the code to detect even or odd numbers. But first, we must start at 3. The previous else clause in Part 1 will now check for even or odd numbers, and treat them the same if odd (except we must divide by 2 as we have excluded even numbers in marked[]) and if even will treat them the same except add the prime value again to find the new first (and of course halving as before). The next loop going over marked indices until we hit a 0 now increases prime by the same formula as low_value *3+2k*. As expected, this improvement about halved our computation time (a little more as we save some on *n/2* broadcasts) with 32 processes. I ran into trouble when increasing the number of processes past 1 node, however. 64 processes over 2 nodes actually increased my speed when I was recording results. 128 processes over 4 nodes decreased execution time from 64 processes, but not more than 32 processes on 1 node. I am not sure if this is from a poor algorithm I have written for excluding even numbers, or if I was encountering MPI issues for broadcasting at the time. The rest of my results for Part 3 ran as expected, but Part 2 64 & 128 processes were slower than expected. The results were still correct however, as the number of primes was as expected. Only strange thing was the broadcasting. Part 3 (using the same algorithm to exclude evens) running at 128 processes averaged 1.78s. While Part 2 at 128 processes averaged at 9.9s. Not exactly sure what happened here.

Figure (2) Part 2 implementation average times in seconds over 10 runs

32 processes	64 processes	128 processes
7.3176614	12.8660945	9.7682332

V: Part III

A. *Eliminating Broadcasts*

To eliminate broadcasts, each process must know the values of the primes to mark, as we cannot rely on receiving information from other tasks. Thus, we must use a sequential implementation of the Sieve of Eratosthenes and complete it for each task. Then we may explore each id's block without interfering with another's. To do this, we must expand upon the even numbers' elimination, and complete this sieve for each task from 3 to $\sqrt{n}$. Each task now has its own knowledge of relevant primes. The tasks may now sieve through their own portion of the marked[] array without any broadcasts. The changes for this task are relatively simple once the sequential sieve is implemented. The sequential sieve is done at the start of each task and its count recorded along with the values that were counted. We once again clear our marked[] variable and dive into the same parallel algorithm, but with the small change of looping through the sequential sieve's primes instead, setting the variable "prime" to each prime found by the sequential algorithm. We then proceed as normal, not forgetting to add the "2" back that we have excluded by eliminating even numbers.

Figure (3) Part 3 implementation average times in seconds over 10 runs

32 processes	64 processes	128 processes
7.3104033	3.6263243	1.7795857

VI: Part IV

I was not able to correctly implement part 4.

VI: Source Code And Results

Source code is attached in tar.gz. Sample output is included in /output/console/[algorithm]. You may also find the error logs on /output/error/[algorithm]. The results for Parts 1 and 3 were as expected. Part 2 with 32 processes was as expected (execution time about halved) but then slowed down when adding more processes. Not sure why this happened. I will test again later to see if it is due to MPI Broadcasting errors, or if I need to improve my code to be more efficient in excluding even numbers while handling larger amounts of processes.