Jonathan Frame
Spring 2019
Pantarhei usr: sp19hpc_jmframe

CS 491/591: High Performance Computing Project 3
Parallel Sieve of Eratosthenes for Finding All Prime Numbers within 10^{10}
Due date: March 22nd, 2019

Use your program to find all prime numbers within 10^{10}. **Output the total number of prime numbers within 10^{10} and the program execution time (i.e., maximum time of all processes used in the MPI program).** Benchmark your program on PantaRhei with 32 (1 node), 64 (2 nodes), 128 (4 nodes) processes to see whether your execution time is reduced by half or not when double the number of computing cores. Compare the execution time of each version of your program to see how different designs affect the execution time of your program. Note that, in syllabus, we emphasize for **ALL** homework assignments: "Please make sure that your programs are properly documented and indented. Provide instructions on how to run your programs, give example runs, **and analyze your results.**"

Summary: Below are brief descriptions of the changes made to the sieving algorithm for this homework. Part 1 required expanding the memory for the integers in the program. Part 2 required making changes to the size and bounds of the sieving values in each process to account for the lack of even numbers. Part 3 required calculating the sieving primes for each process, to make up for the lack of broadcasting. I was not able to complete Part 4. Figure 1 shows the results these changes to the algorithm made in terms of performance. Table 1 shows that Parts 2 and 3 both improved the performance of the algorithm significantly. With 128 processors (using 4 cores) the execution run time of the Part 3 algorithm is only 14% of Part 1.

The code for this project has been uploaded to Blackboard, and should be Compiled on Pantarhei as follows:

```
mpicc -lrt sieve1.c -o program_sieve1 -lm
mpicc -lrt sieve2.c -o program_sieve2 -lm
mpicc -lrt sieve3.c -o program_sieve3 -lm
```

Example slurm job script:

```
#!/bin/bash -l
#BATCH -q defq
#SBATCH -N 4
#SBATCH -n 128
#SBATCH -t 00:20:00
#SBATCH --exclude=gpu01
#SBATCH -J jmframe_sp19hpc_hw3_part3
#SBATCH -o results/OUT_sieve3_p128.%j.out
#SBATCH -e results/ERR_sieve3_p128.%j.error

mpirun ./program_sieve3 10000000000
```

Part 1 (20 points): Simply modify the parallel Sieve of Eratosthenes program in class so that the program can find all prime members within 10^{10} (hint: 10^{10} is larger than value range of int type).

This part of the assignment simply required changing the data types for many of the values used in the algorithm to reach 10^10. We need to use long long integers, which max out at $18x10^{18}$, well above the 10^{10} that we need. There were a few other similar changes to the algorithm to get to correct result.

Part 2 (40 points): Modify the parallel Sieve of Eratosthenes program in Part 1 so that the program does NOT set aside memory for even integers.

The most important change for this part of the project were the size, low_value and high_value numbers for all processes. With these changes many of the program functions had to be modified to account for the shorter size of integer values. In addition, a lot of the calculations had to be adjusted slightly to account for the first prime value, 2, not being included in the sieve.

Part 3 (30 points): Modify the parallel Sieve of Eratosthenes program in **Part 2** so that each process of the program finds its own sieving primes via local computations instead of broadcasts.

In the original version of the algorithm the sieving prime numbers were calculated only by Process 0 and broadcast to the other processes. Without the broadcast function, which takes more time than some number of calculations, each process then needs to calculate their own sieving primes. This requires an additional marking array, to keep track of the local sieving values for each process.

Analysis:

Figure 1 show that for Parts 1 & 2 using 64 processors is actually slower than 32 processors. For Part 1 using 128 processors is fastest, but for Part 2 using 32 processors is fastest. For Part 3 the performance is improved when more processors are used, at least up to 128. This is because there is much less communication time, so the additional processors can fulfill their role of speeding up computation and directly improve performance.

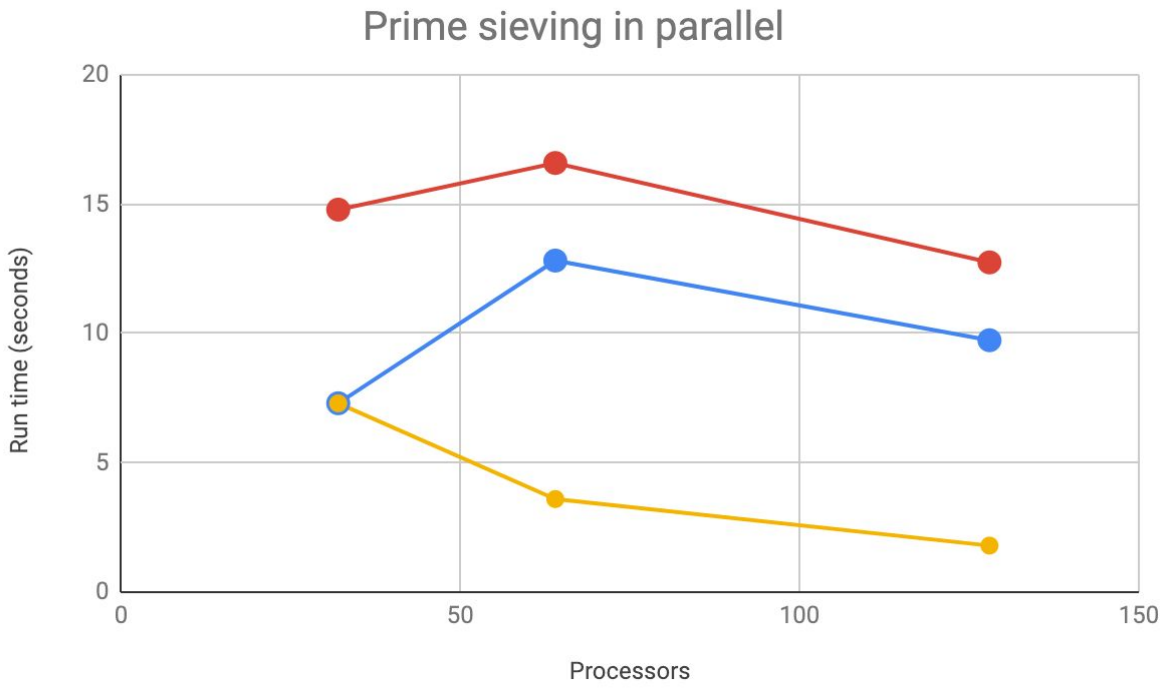

Figure 1. Performance of sieving algorithms. Number of processors vs. program execution time.

Table 1 shows that the algorithm in Part 2 is faster than Part 1, and Part 3 is faster than both. Part 3 however, is significantly faster improving performance with speeds as low as 14% of the baseline (Part 1) execution time.

Table 1. Execution time performance of the sieving algorithms.

Processors	32	64	128
Part 1	100%, baseline	100%, baseline	100%, baseline
Part 2	49%	77%	76%
Part 3	49%	22%	14%

Table 2 shows that with 32 processors as the baseline the algorithms in Parts 1 and 2 do not necessarily improve performance with additional processors. The Part 3 algorithm though, which make more efficient use of parallel calculations, does see improvements as processors are increased.

Table 2. Execution time performance of the sieving algorithms.

Processors	32	64	128
Part 1	100%, baseline	112%	86%
Part 2	100%, baseline	176%	133%
Part 3	100%, baseline	49%	25%