Group 4
Brian Tiner
Adam Gumieniak
Matt Buchanan
Duc Trung Nguyen

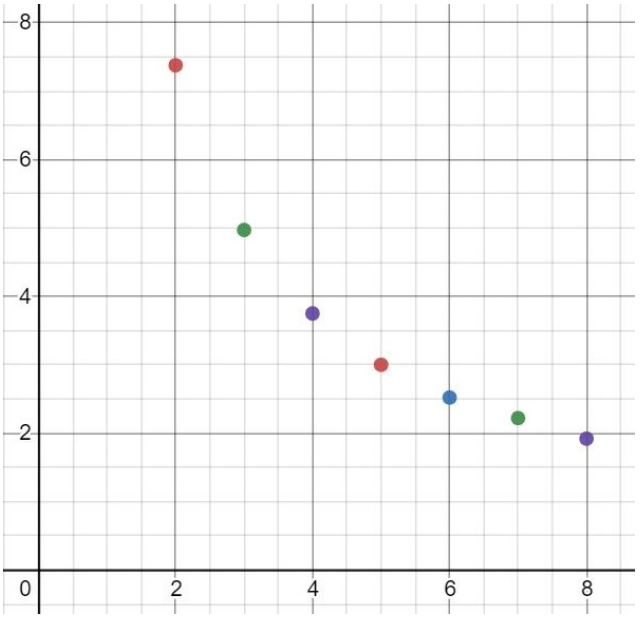
CP431 A1 Prime Gaps

We accomplished finding the largest prime gap from 1 to 1 billion with an MPI program. We first wrote a serial program to compute the largest prime gap then built an MPI program based on it. Our program utilizes the GMP library to find the prime numbers accurately. We first created a delta variable to calculate the work load distributed to each processor accurately. We then run a while loop that loops for the amount of numbers that processor has been given. The while loop finds the next prime number and compares the gap between prime numbers to the current gap already found. If it finds a new largest gap it updates the variables accordingly. It then splits into two if statements. We use the slave master method meaning process 0 is the “slave master” and all the other processes are the “slaves” which compute the output and returns it to process 0. If the process is not the 0 process then it sends all its information through MPI_Send to the 0 process. If the process is the 0 process then it assigns its information to arrays then receives all the information from the other processes using MPI_Recv and assigns them to the same arrays. It then runs through the arrays finding and finds the largest gap and outputs it to the user along with the boundary prime numbers of the gap.

In our output we concluded that increasing the cores did not give is perfect increases. By that we mean it is expected to have halved time when we compare the outputs from cores 2 and 4 however this is not the case. Also, increasing the amount of cores gives less of a time decrease which is evident in our findings below.

Output:

Processors	Job Wall Clock Time (MM:SS)
2	7:23
3	4:58
4	3:45
5	3:00
6	2:31
7	2:13
8	1:55



Gap Results:

Prime Gap Size: 282
Prime Gap High Boundary: 436273291
Prime Gap Low Boundary: 436273009

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <gmp.h>
#include <mpi.h>

int main(int argc, char **argv){
	int p, rank;
	//Range start and end are boundaries, first prime and last prime are highest and lowest prime number in range, gap high and low are the boundary numbers of the prime gap
	long long n = 1000000000, range_start, range_end, last_prime, first_prime, delta, size, gap_boundary_low, gap_boundary_high;
	long long *first_prime_array, *last_prime_array, *size_array, *gap_max_array, *gap_min_array;
	mpz_t gap_low, gap_high, current_low, current_high, max_gap_size, current_gap_size, mpz_range_end;
	MPI_Status status;

	//initializing MPI
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &p);

	//setting arrays
	if (rank == 0){
		first_prime_array = malloc(p * sizeof(long long));
		last_prime_array = malloc(p * sizeof(long long));
		size_array = malloc(p * sizeof(long long));
		gap_max_array = malloc(p * sizeof(long long));
		gap_min_array = malloc(p * sizeof(long long));
	}

	//getting delta size
	delta = n/p;

	//setting range for processor
	range_start = delta * rank;
```

```
range_end = delta * rank + delta;
if (rank < n%delta){
	range_end++;
}

//initializations
mpz_init(current_low);
mpz_init(current_high);
mpz_init(gap_low);
mpz_init(gap_high);
mpz_init(max_gap_size);
mpz_init(mpz_range_end);
mpz_init(current_gap_size);

//setting first low value
if (rank == 0){
	mpz_set_si(current_low, range_start);
}
else{
	mpz_set_si(current_low, range_start - 1);
}

//setting current low and high to first prime numbers
mpz_nextprime(current_low, current_low);
mpz_nextprime(current_high, current_low);

//converting back to normal data type
first_prime = mpz_get_si(current_low);

//setting gap low and high to current low and high
mpz_set_si(gap_low, 0);
mpz_set_si(gap_high, 0);
mpz_set_si(max_gap_size, 0);
//setting duplicate variable
mpz_set_si(mpz_range_end, range_end);

while (mpz_cmp(current_high, mpz_range_end) <= 0){
	//getting gap size
	mpz_sub(current_gap_size, current_high, current_low);
	//setting gap size
	if (mpz_cmp(current_gap_size, max_gap_size) > 0){
		mpz_set(max_gap_size, current_gap_size);
		mpz_set(gap_low, current_low);
		mpz_set(gap_high, current_high);
	}
	//setting current low to current high
```

```
                mpz_set(current_low, current_high);
                //getting next prime
                mpz_nextprime(current_high, current_high);
        }
        //converting values back to normal data types
        last_prime = mpz_get_si(current_low);
        size = mpz_get_si(max_gap_size);
        gap_boundary_low = mpz_get_si(gap_low);
        gap_boundary_high = mpz_get_si(gap_high);

        //zero process
        if (rank == 0){
                //assigning first values in arrays
                first_prime_array[0] = first_prime;
                last_prime_array[0] = last_prime;
                size_array[0] = size;
                gap_max_array[0] = gap_boundary_high;
                gap_min_array[0] = gap_boundary_low;

                //filling rest of arrays
                for (int i = 1; i < p; i++){
                        MPI_Recv(&(first_prime_array[i]), 1, MPI_LONG_LONG, i, 0,
MPI_COMM_WORLD, &status);
                        MPI_Recv(&(last_prime_array[i]), 1, MPI_LONG_LONG, i, 1,
MPI_COMM_WORLD, &status);
                        MPI_Recv(&(size_array[i]), 1, MPI_LONG_LONG, i, 2,
MPI_COMM_WORLD, &status);
                        MPI_Recv(&(gap_max_array[i]), 1, MPI_LONG_LONG, i, 3,
MPI_COMM_WORLD, &status);
                        MPI_Recv(&(gap_min_array[i]), 1, MPI_LONG_LONG, i, 4,
MPI_COMM_WORLD, &status);
                }
                //finding largest gap
                for(int i = 0; i < p; i++){
                        if (size_array[i] > size){
                                size = size_array[i];
                                gap_boundary_low = gap_min_array[i];
                                gap_boundary_high = gap_max_array[i];
                        }
                }
                //comparing largest gap to edge cases
                for(int i = 0; i < p - 1; i++){
                        if (first_prime_array[i + 1] - last_prime_array[i] > size){
                                size = first_prime_array[i + 1] - last_prime_array[i];
                                gap_boundary_low = last_prime_array[i];
                                gap_boundary_high = first_prime_array[i + 1];
```

```
                        }
                }
                printf("Prime Gap Size: %lld Prime Gap High Boundary: %lld Prime Gap Low
Boundary: %lld", size, gap_boundary_high, gap_boundary_low);
        }
        //for other processes
        else{
                //sending information
                MPI_Send(&first_prime, 1, MPI_LONG_LONG, 0, 0, MPI_COMM_WORLD);
                MPI_Send(&last_prime, 1, MPI_LONG_LONG, 0, 1, MPI_COMM_WORLD);
                MPI_Send(&size, 1, MPI_LONG_LONG, 0, 2, MPI_COMM_WORLD);
                MPI_Send(&gap_boundary_high, 1, MPI_LONG_LONG, 0, 3,
MPI_COMM_WORLD);
                MPI_Send(&gap_boundary_low, 1, MPI_LONG_LONG, 0, 4,
MPI_COMM_WORLD);
        }

        MPI_Finalize();

        return 0;
}
```

Bonus:

Prime Gap Size: 540
Prime Gap High Boundary: 738832928467
Prime Gap Low Boundary: 738832927927
Job Wall-clock time: 04:42:11

From viewing the data above, we have completed the bonus question. Meaning our code is scalable for numbers greater than 1 billion.