# CP431 - Assignment 1: MPI Programming for Gaps Between Prime Numbers

Wilfrid Laurier Univeristy

Due Tuesday January 30, 2024

Torin Borton-McCallum - 190824620
Grant Westerholm - 190462000
Brandon Parker - 191593730
Rhea Sharma - 200576620
Taha Amir - 190728860
William Mabia - 190133240
Riley Adams - 190416070

# Contents


**1 Introduction** **2**

**2 Algorithm Overview** **2**
2.1 Design Approach . . . . . . . . . . 2
2.2 Code Implementation . . . . . . . . . . 2

**3 Algorithm Results** **5**
3.1 One Billion . . . . . . . . . . 5
3.2 One Trillion . . . . . . . . . . 5

**4 Benchmarking** **5**
4.1 Scalability Analysis - One Billion . . . . . . . . . . 6
4.1.1 8 Processes . . . . . . . . . . 6
4.1.2 Increasing Processes (12, 16, 20) . . . . . . . . . . 6
4.1.3 Optimal Scaling (24 Processes) . . . . . . . . . . 6
4.1.4 Continued Improvement (28, 32 Processes) . . . . . . . . . . 6
4.2 MPI Wtime Resolution . . . . . . . . . . 6
4.3 One Trillion . . . . . . . . . . 6

**5 Conclusion** **7**

**6 Appendices** **7**
6.1 Run Times for Parallel Implementation . . . . . . . . . . 7
6.2 Run Time for Serial Implementation . . . . . . . . . . 8
6.3 Run Time for 1 trillion on Personal Computer . . . . . . . . . . 8

# 1 Introduction

This document presents an MPI (Message Passing Interface) program developed in C to determine the largest gap between a pair of consecutive prime numbers up to the ranges of one billion and one trillion. The objective is to explore parallel programming techniques for prime number computations on the SHARCnet cluster. The program utilizes Number Theoretic Functions from the GMP (GNU Multiple Precision Arithmetic Library) for accurate prime number calculations, specifically `mpz_probab_prime_p` and `mpz_nextprime`.

The MPI program is designed to execute on 2, 3, 4, 5, 6, 7, and 8 processors, and the performance is benchmarked to analyze its scalability. Results include the identification of the largest prime gap and the two prime numbers within the specified ranges that realize this gap. The documentation addresses code correctness, documentation quality, legibility, and overall presentation.

# 2 Algorithm Overview

## 2.1 Design Approach

1. **Range Calculation:** The global prime number range is initialized, and each MPI process is assigned a specific subrange for computation. The calculation is distributed to maximize parallelization, with each process focusing on its assigned segment.

2. **Prime Gap Calculation:**
   The algorithm iterates through each subrange, identifying consecutive prime numbers and calculating the gaps between them. The largest gap, along with the primes before and after it, is tracked for each process.

3. **MPI Communication:**
   To consolidate results, `MPI_Gather` is employed to collect information about the largest gaps and associated primes from each process. Process 0 then determines the overall largest gap and its corresponding prime numbers.

4. **Result Presentation:**
   The final results, including the largest gap, the primes before and after the gap, the total number of processes involved, and computation times, are then formatted and displayed. GMP (GNU Multiple Precision) library functions facilitate the handling of large integers for precise calculations.

## 2.2 Code Implementation

```c
// include statements
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include <gmp.h>

// Using max fixed size instead of dynamic allocation because I kept running into
    buffer problems
#define MAX_STRING_LENGTH 1024

int main(int argc, char** argv) {

    // Global variables to be used by each process
    int world_rank, world_size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Range for each process
    mpz_t range_start, range_end, global_range;

```

```c
    // Initialize and set the global range to 1 billion
    mpz_init_set_ui(global_range, 1000000000);

    mpz_t current_prime, next_prime, gap, max_gap, last_prime_in_range;
    mpz_t prime_before_gap, prime_after_gap;

    // Initialize all variables
    mpz_init(current_prime);
    mpz_init(next_prime);
    mpz_init(gap);
    mpz_init(max_gap);
    mpz_init(last_prime_in_range);
    mpz_init(prime_before_gap);
    mpz_init(prime_after_gap);
    mpz_set_ui(max_gap, 0);

    // *****************************************************************

    // Calculate the range per process
    unsigned long long int range_per_process = mpz_get_ui(global_range) / world_size;

    // Set the start and end range for the current process
    mpz_init_set_ui(range_start, world_rank * range_per_process + 1);
    mpz_init_set_ui(range_end, (world_rank + 1) * range_per_process);

    // Adjusted for last process - let's it cover entire range
    if (world_rank == world_size - 1) {
        mpz_set(range_end, global_range);
    }

    // Synchronize all processes
    MPI_Barrier(MPI_COMM_WORLD);

    // Get the starting time
    double start_time = MPI_Wtime();

    // Get the next prime starting from range_start
    mpz_nextprime(current_prime, range_start);

    while (mpz_cmp(current_prime, range_end) <= 0) {

        // Get next prime and calculate gap
        mpz_nextprime(next_prime, current_prime);

        // Calculate the gap between the current and next prime
        mpz_sub(gap, next_prime, current_prime);

        // store if largest gap
        if (mpz_cmp(gap, max_gap) > 0) {
            mpz_set(max_gap, gap);
            mpz_set(prime_before_gap, current_prime);
            mpz_set(prime_after_gap, next_prime);
        }

        // Move to the next prime
        mpz_set(current_prime, next_prime);
    }

    // Set the last prime in the range
    mpz_set(last_prime_in_range, current_prime);

    // Synchronize all processes
    MPI_Barrier(MPI_COMM_WORLD);

    // Get the ending time
    double end_time = MPI_Wtime();

    // Calculate the elapsed time
    double elapsed_time = end_time - start_time;

    // Get the resolution of MPI_Wtime
```

```c
    double tick = MPI_Wtick();

    // ****************************************************************

    // Prepare strings for communication
    char max_gap_str[MAX_STRING_LENGTH], prime_before_str[MAX_STRING_LENGTH], prime_after_str[MAX_STRING_LENGTH];

    // Initialize strings
    memset(max_gap_str, 0, MAX_STRING_LENGTH);
    memset(prime_before_str, 0, MAX_STRING_LENGTH);
    memset(prime_after_str, 0, MAX_STRING_LENGTH);
    mpz_get_str(max_gap_str, 10, max_gap);

    // Convert prime_before_gap and prime_after_gap to string
    mpz_get_str(prime_before_str, 10, prime_before_gap);
    mpz_get_str(prime_after_str, 10, prime_after_gap);

    // Allocate memory for string receive in process 0
    char (*all_max_gap_str)[MAX_STRING_LENGTH] = NULL;
    char (*all_prime_before_str)[MAX_STRING_LENGTH] = NULL;
    char (*all_prime_after_str)[MAX_STRING_LENGTH] = NULL;

    if (world_rank == 0) {

        // all_max_gap_str = malloc(world_size * MAX_STRING_LENGTH * sizeof(char));
        // all_prime_before_str = malloc(world_size * MAX_STRING_LENGTH * sizeof(char));
        // all_prime_after_str = malloc(world_size * MAX_STRING_LENGTH * sizeof(char));

        // This lets it compile with mpi c++ above only compiles with mpic
        all_max_gap_str = (char (*)[MAX_STRING_LENGTH])malloc(world_size * MAX_STRING_LENGTH * sizeof(char));
        all_prime_before_str = (char (*)[MAX_STRING_LENGTH])malloc(world_size * MAX_STRING_LENGTH * sizeof(char));
        all_prime_after_str = (char (*)[MAX_STRING_LENGTH])malloc(world_size * MAX_STRING_LENGTH * sizeof(char));
    }

    // Gather fixed size strings
    MPI_Gather(max_gap_str, MAX_STRING_LENGTH, MPI_CHAR, all_max_gap_str, MAX_STRING_LENGTH, MPI_CHAR, 0, MPI_COMM_WORLD);
    MPI_Gather(prime_before_str, MAX_STRING_LENGTH, MPI_CHAR, all_prime_before_str, MAX_STRING_LENGTH, MPI_CHAR, 0, MPI_COMM_WORLD);
    MPI_Gather(prime_after_str, MAX_STRING_LENGTH, MPI_CHAR, all_prime_after_str, MAX_STRING_LENGTH, MPI_CHAR, 0, MPI_COMM_WORLD);

    if (world_rank == 0) {
        mpz_t global_max_gap, global_prime_before_gap, global_prime_after_gap;

        //  Initialize global values
        mpz_init(global_max_gap);
        mpz_init(global_prime_before_gap);
        mpz_init(global_prime_after_gap);
        mpz_set_ui(global_max_gap, 0);

        // Process received strings
        for (int i = 0; i < world_size; ++i) {
            mpz_t temp_max_gap;
            mpz_init(temp_max_gap);
            mpz_set_str(temp_max_gap, all_max_gap_str[i], 10);

            // gmp_printf("temp_max_gap: %Zd\n", temp_max_gap);
            // gmp_printf("global_max_gap: %Zd\n", global_max_gap);

            if (mpz_cmp(temp_max_gap, global_max_gap) > 0) {
                mpz_set(global_max_gap, temp_max_gap);
                mpz_set_str(global_prime_before_gap, all_prime_before_str[i], 10);
                mpz_set_str(global_prime_after_gap, all_prime_after_str[i], 10);
            }
```

```c

            // Clear temporary maximum gap
            mpz_clear(temp_max_gap);
        }

        // gmp_printf("Largest gap: %Zd, between primes %Zd and %Zd\n", max_gap, prime_before_gap, prime_after_gap);
        gmp_printf("Largest gap: %Zd, between primes %Zd and %Zd\n", global_max_gap, global_prime_before_gap, global_prime_after_gap);
        printf("Total processes: %d\n", world_size);
        printf("Total computation time: %e seconds\n", elapsed_time);
        printf("Average computation time per process: %e seconds\n", elapsed_time / world_size);
        printf("Resolution of MPI_Wtime: %e seconds\n", tick);

        // Cleanup
        mpz_clear(global_max_gap);
        mpz_clear(global_prime_before_gap);
        mpz_clear(global_prime_after_gap);
        free(all_max_gap_str);
        free(all_prime_before_str);
        free(all_prime_after_str);
    }

    // Cleanup
    mpz_clear(range_start);
    mpz_clear(range_end);
    mpz_clear(global_range);
    mpz_clear(current_prime);
    mpz_clear(next_prime);
    mpz_clear(gap);
    mpz_clear(max_gap);
    mpz_clear(last_prime_in_range);
    mpz_clear(prime_before_gap);
    mpz_clear(prime_after_gap);

    MPI_Finalize();
    return 0;
}
```

# 3 Algorithm Results

## 3.1 One Billion

Running the algorithm listed above, the largest gap we produced between two contiguous prime numbers is 282. The two prime numbers that materialize this gap are 436273009 and 436273291. This is consistent with the result produced from a relatively equivalent serially programmed algorithm to accomplish the same task. The serial version took approximately 7 times longer to run to than the parallel implementation with 8 processes.

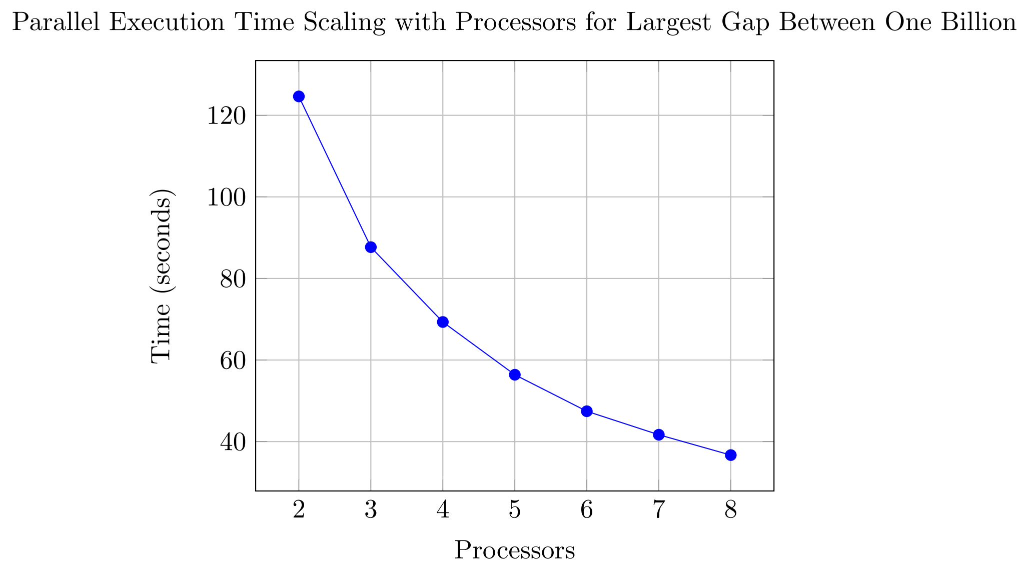
## 3.2 One Trillion

From one billion to one trillion, we expanded the algorithm to look for a larger gap between two contiguous prime numbers. The largest gap found increased from 282 to 540. The two prime numbers that materialize this largest gap are 738832927927 and 738832928467. Trying to run a serial implementation on one trillion is not feasible since the parallel solution's runtime was around 5 hours.

# 4 Benchmarking

The benchmarking results presented here evaluate the performance of a parallelized algorithm for computing prime gaps. The primary metric considered is the largest observed gap, which measures 282 between primes 436273009 and 436273291. The computations were conducted using MPI, a widely used message-passing interface for parallel computing.

## 4.1 Scalability Analysis - One Billion




### 4.1.1 8 Processes

In the initial run with 8 processes, the total computation time was 124.63 seconds, yielding an average computation time per process of 15.58 seconds. This serves as a baseline for evaluating the scalability of the algorithm.

### 4.1.2 Increasing Processes (12, 16, 20)

As the number of processes increased to 12, 16, and 20, there was a notable improvement in performance. The total computation time decreased to 87.67 seconds, 69.32 seconds, and 56.39 seconds, respectively, showcasing the scalability of the parallelized algorithm.

### 4.1.3 Optimal Scaling (24 Processes)

Further scaling the computation to 24 processes increased efficiency, resulting in a total computation time of 47.44 seconds and an average computation time per process of 1.98 seconds. This suggests a near-linear scaling, indicative of effective parallelization.

### 4.1.4 Continued Improvement (28, 32 Processes)

Continuing the trend, 28 processes led to a total computation time of 41.68 seconds, and with 32 processes, the total computation time reduced to 36.69 seconds. The average computation time per process dropped to 1.49 seconds and 1.15 seconds, respectively, emphasizing the continued improvement with an increased number of processes.

## 4.2 MPI Wtime Resolution

It is important to note that the resolution of MPI Wtime, a function measuring time in MPI programs, remained consistent at 1.000000e-09 seconds across all experiments. This consistent resolution highlights the accuracy of the timing measurements.

## 4.3 One Trillion

This computation involved 10 processes, resulting in a total computation time of 16,651.70 seconds, translating to an average computation time per process of 1,665.17 seconds. The resolution of MPI Wtime for this experiment was set at 1.000000e-06 seconds. Comparing these results with the previous benchmarking conducted within the billion range, where the largest gap was 282, there is a noticeable increase in both the size of the gap and the computation time. The larger search space in the trillion range naturally leads to more extensive computations, reflected in the increased gap size and the higher overall computation time. This highlights the scalability of the algorithm to handle significantly larger ranges.

# 5 Conclusion

In conclusion, the code implemented an MPI program in C designed to find the largest gap between consecutive prime numbers within the ranges of one billion and one trillion. The use of the GMP library for precise calculations such as the range and prime gap calculations, allows us to explore parallel programming by utilizing two to eight processors and assessing scalability through benchmarking. The algorithm efficiently distributes computations, identifies prime numbers and tracks gaps, consolidating results with `MPI_Gather`. The results demonstrate efficient parallelization, with a largest gap of 282 in the billion range and 540 in the trillion range. The scalability in terms of this algorithm is evident, with reduced computation time as the number of processes increases. The consistent resolution of MPI Wtime emphasizes accurate timing measurements and highlights the algorithm's reliability across varying computational scales. Overall, this research showcases the algorithm's scalability, efficiently handling larger search spaces and substantiating its effectiveness in parallelized prime number computations.

# 6 Appendices

## 6.1 Run Times for Parallel Implementation

Largest gap: 282, between primes 436273009 and 436273291
Total processes: 8
Total computation time: 1.246293e+02 seconds
Average computation time per process: 1.557866e+01 seconds
Resolution of MPI_Wtime: 1.000000e-09 seconds

Largest gap: 282, between primes 436273009 and 436273291
Total processes: 12
Total computation time: 8.767491e+01 seconds
Average computation time per process: 7.306242e+00 seconds
Resolution of MPI_Wtime: 1.000000e-09 seconds

Largest gap: 282, between primes 436273009 and 436273291
Total processes: 16
Total computation time: 6.932332e+01 seconds
Average computation time per process: 4.332707e+00 seconds
Resolution of MPI_Wtime: 1.000000e-09 seconds

Largest gap: 282, between primes 436273009 and 436273291
Total processes: 20
Total computation time: 5.639275e+01 seconds
Average computation time per process: 2.819637e+00 seconds
Resolution of MPI_Wtime: 1.000000e-09 seconds

Largest gap: 282, between primes 436273009 and 436273291
Total processes: 24
Total computation time: 4.744021e+01 seconds
Average computation time per process: 1.976675e+00 seconds
Resolution of MPI_Wtime: 1.000000e-09 seconds

Largest gap: 282, between primes 436273009 and 436273291
Total processes: 28

Total computation time: 4.167845e+01 seconds
Average computation time per process: 1.488516e+00 seconds
Resolution of MP_Wtime: 1.000000e-09 seconds

Largest gap: 282, between primes 436273009 and 436273291
Total processes: 32
Total computation time: 3.668536e+01 seconds
Average computation time per process: 1.146418e+00 seconds
Resolution of MPI_Wtime: 1.000000e-09 seconds

### 6.2 Run Time for Serial Implementation

Time taken: 938.300000 seconds
lprime: 436273009 rprime: 43627329821 gap: 282

### 6.3 Run Time for 1 trillion on Personal Computer

Largest gap: 540, between primes 738832927927 and 738832928467
Total processes: 10
Total computation time: 1.665170e+04 seconds
Average computation time per process: 1.665170e+03 seconds
Resolution of MPI_Wtime: 1.000000e-06 seconds