

ASSIGNMENT 1

Group #3



Contents

[Solution 1](#_Toc21087635)

[Source Code 3](#_Toc21087636)

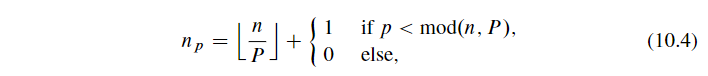
[Benchmark 6](#_Toc21087637)

[Conclusion: 7](#_Toc21087638)

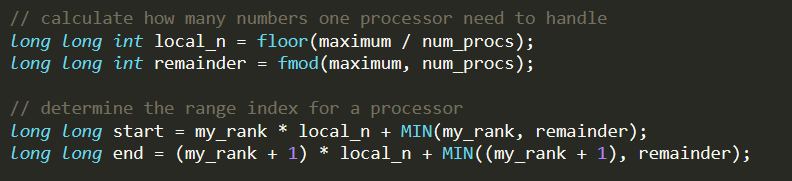
[Results 7](#_Toc21087639)

# Solution

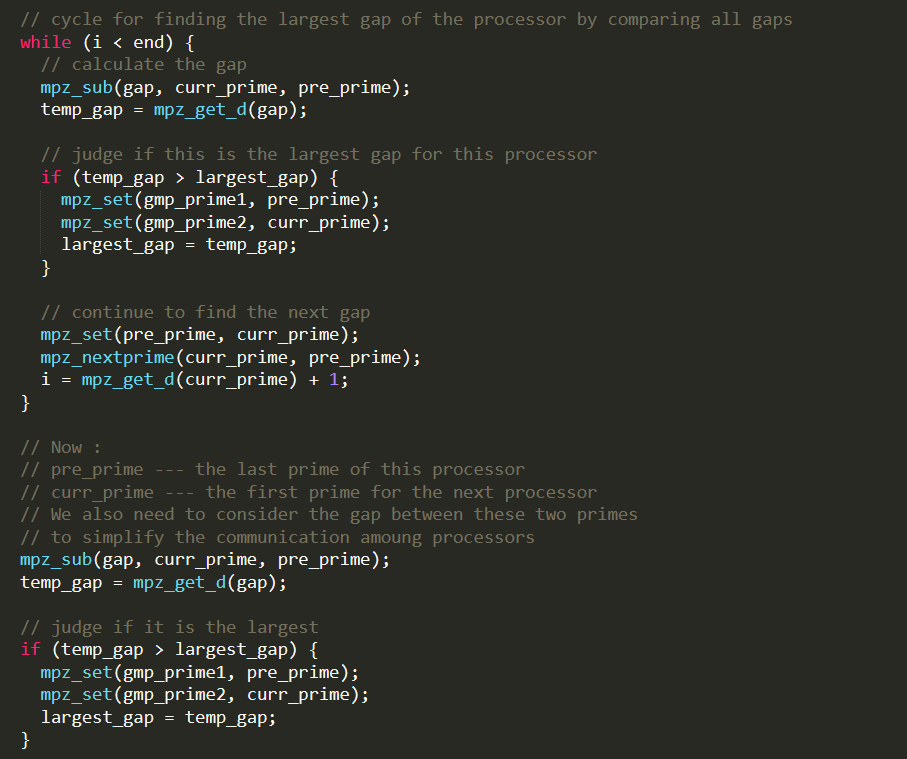
Divide the numbers of the given range according to the following equation:



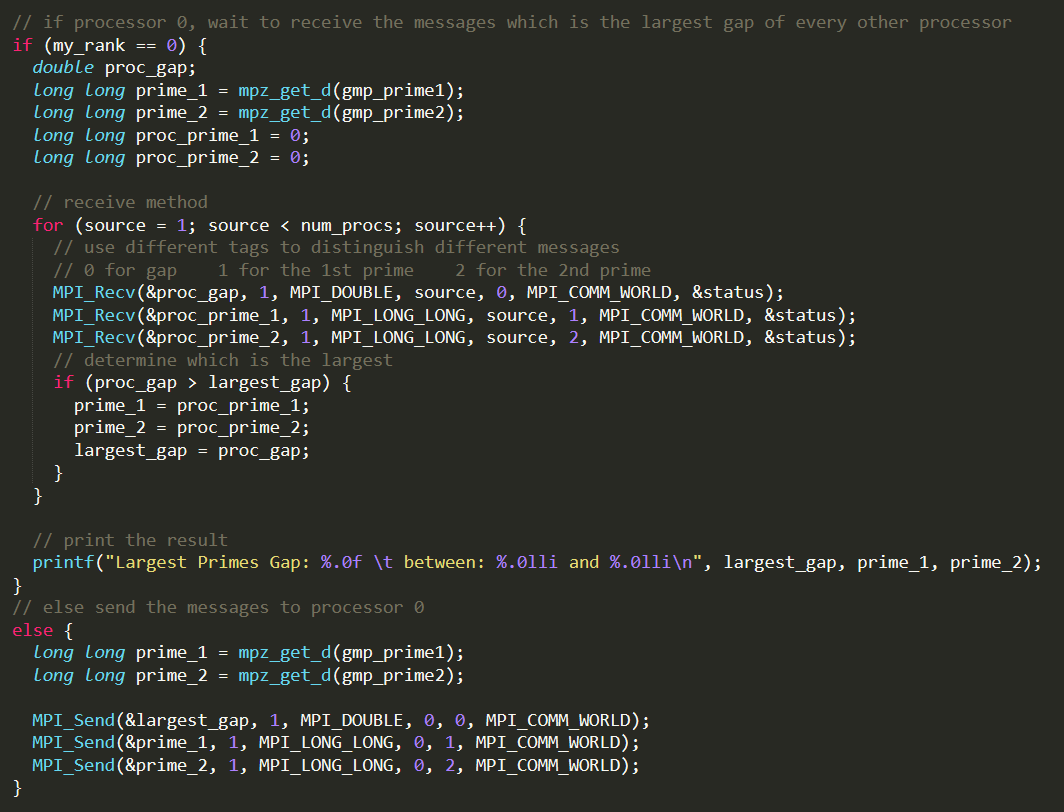
Then we can get the index of the range for every processor.



And next we use the index to make a cycle for every processor to calculate their own largest prime gap. Don’t forget to get the gap between the last prime of the current processor and the first prime of the next processor.



Finally, every other processors send the results to processor 0 who will find the largest gap by comparing every largest gap of every processor.



# Source Code

**Dear professor, please be careful when copying the code to run on your own computer. It may have some syntax errors because of the copying operation. But it will usually work I think.**

#include <stdio.h>

#include <math.h>

#include <gmp.h>

#include "mpi.h"

#define MIN(a,b) (a < b ? a : b)

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

int my\_rank;

int num\_procs;

int source;

int dest;

long long maximum = 1000000000;

double largest\_gap = 0;

double temp\_gap = 0;

MPI\_Status status;

// initialize MPI

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs);

MPI\_Barrier(MPI\_COMM\_WORLD);

double elapsed\_time = -MPI\_Wtime();

// calculate how many numbers one processor need to handle

long long int local\_n = floor(maximum / num\_procs);

long long int remainder = fmod(maximum, num\_procs);

// determine the range index for a processor

long long start = my\_rank \* local\_n + MIN(my\_rank, remainder);

long long end = (my\_rank + 1) \* local\_n + MIN((my\_rank + 1), remainder);

// initialize mpz types to store "the two primes" in every processor

mpz\_t gmp\_prime1;

mpz\_init\_set\_d(gmp\_prime1, 2);

mpz\_t gmp\_prime2;

mpz\_init\_set\_d(gmp\_prime2, 2);

// cast the long long type to mpz\_t type for better use of GMP

mpz\_t gmp\_index;

mpz\_init\_set\_d(gmp\_index, start);

mpz\_t gap;

mpz\_init(gap);

mpz\_t pre\_prime;

mpz\_init(pre\_prime);

mpz\_t curr\_prime;

mpz\_init(curr\_prime);

// find the first prime for the processor

mpz\_nextprime(pre\_prime, gmp\_index);

mpz\_set(curr\_prime, pre\_prime);

long long i;

i = mpz\_get\_d(pre\_prime) + 1;

// cycle for finding the largest gap of the processor by comparing all gaps

while (i < end) {

// calculate the gap

mpz\_sub(gap, curr\_prime, pre\_prime);

temp\_gap = mpz\_get\_d(gap);

// judge if this is the largest gap for this processor

if (temp\_gap > largest\_gap) {

mpz\_set(gmp\_prime1, pre\_prime);

mpz\_set(gmp\_prime2, curr\_prime);

largest\_gap = temp\_gap;

}

// continue to find the next gap

mpz\_set(pre\_prime, curr\_prime);

mpz\_nextprime(curr\_prime, pre\_prime);

i = mpz\_get\_d(curr\_prime) + 1;

}

// Now :

// pre\_prime --- the last prime of this processor

// curr\_prime --- the first prime for the next processor

// We also need to consider the gap between these two primes

// to simplify the communication amoung processors

mpz\_sub(gap, curr\_prime, pre\_prime);

temp\_gap = mpz\_get\_d(gap);

// judge if it is the largest

if (temp\_gap > largest\_gap) {

mpz\_set(gmp\_prime1, pre\_prime);

mpz\_set(gmp\_prime2, curr\_prime);

largest\_gap = temp\_gap;

}

// if processor 0, wait to receive the messages which is the largest gap of every other processor

if (my\_rank == 0) {

double proc\_gap;

long long prime\_1 = mpz\_get\_d(gmp\_prime1);

long long prime\_2 = mpz\_get\_d(gmp\_prime2);

long long proc\_prime\_1 = 0;

long long proc\_prime\_2 = 0;

// receive method

for (source = 1; source < num\_procs; source++) {

// use different tags to distinguish different messages

// 0 for gap 1 for the 1st prime 2 for the 2nd prime

MPI\_Recv(&proc\_gap, 1, MPI\_DOUBLE, source, 0, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&proc\_prime\_1, 1, MPI\_LONG\_LONG, source, 1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&proc\_prime\_2, 1, MPI\_LONG\_LONG, source, 2, MPI\_COMM\_WORLD, &status);

// determine which is the largest

if (proc\_gap > largest\_gap) {

prime\_1 = proc\_prime\_1;

prime\_2 = proc\_prime\_2;

largest\_gap = proc\_gap;

}

}

// print the result

printf("Largest Primes Gap: %.0f \t between: %.0lli and %.0lli\n", largest\_gap, prime\_1, prime\_2);

}

// else send the messages to processor 0

else {

long long prime\_1 = mpz\_get\_d(gmp\_prime1);

long long prime\_2 = mpz\_get\_d(gmp\_prime2);

MPI\_Send(&largest\_gap, 1, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

MPI\_Send(&prime\_1, 1, MPI\_LONG\_LONG, 0, 1, MPI\_COMM\_WORLD);

MPI\_Send(&prime\_2, 1, MPI\_LONG\_LONG, 0, 2, MPI\_COMM\_WORLD);

}

// get the elapsed time for benchmarking

elapsed\_time += MPI\_Wtime();

printf("The elapsed time is: %lf\n", elapsed\_time);

MPI\_Finalize();

}

# Benchmark

These values are derived on Oct 4, 2019. (About 12:00am) Sometimes, the values change according to different time when we run the program. But the trend would not change.

|  |  |  |
| --- | --- | --- |
| #Processors | Real Time (s) | Theoretical Time (s) |
| 2 | 323.92 | 323.92 |
| 3 | 228.03 | 215.95 |
| 4 | 167.35 | 161.96 |
| 5 | 133.88 | 129.57 |
| 6 | 114.06 | 107.97 |
| 7 | 99.37 | 92.55 |
| 8 | 87.67 | 80.98 |

## Conclusion:

As the graph shows, the execution time decreases as we add processors. And it’s a “perfect” speed improvement theoretically. But in reality, due to the reduction operations, the theoretical and real execution time are different and the difference grows with adding processors.

# Results

For range [1, 10^9], the largest gap is 282 between 436273009 and 436273291



For range [1, 10^12], the largest gap is：

Unfortunately, It took too long to find it.