CS3211 Parallel and Concurrent Programming

Project 2 Report

Group 5

Ong Jit Sheng, Jonathan	A0073924B
Tran Minh Hoang	A0088452X
Le Vu Nguyen Chuong	A0088552W

I. Documentation

Our algorithm:

We divide the job to two parts:

1/ Find all the factors of p-1

2/ Try out all numbers as candidates for a generator

Finding the factors of p-1

For the first part, we test a range of numbers to see if they are factors of p-1. We split the range of numbers to be tested among the processes such that each process has an almost equal range of numbers to test. All the processes will calculate the factors and put them into its local array. After all have finished, the processes will send and receive all the other factors computed by the other processes (using MPI_Allgather) so that each process now knows all the factors of p-1. For optimization, we only find the prime factors (will be elaborated later).

Step 1: Divide the range of numbers almost equally among each process. The maximum number is up to sqrt(p-1)

Process 0	Process 1	….	Process n
[2… end_val0]	[starval1…. endval1]		[starvaln… endvaln]

**Note*: Assuming we have n+1 processes

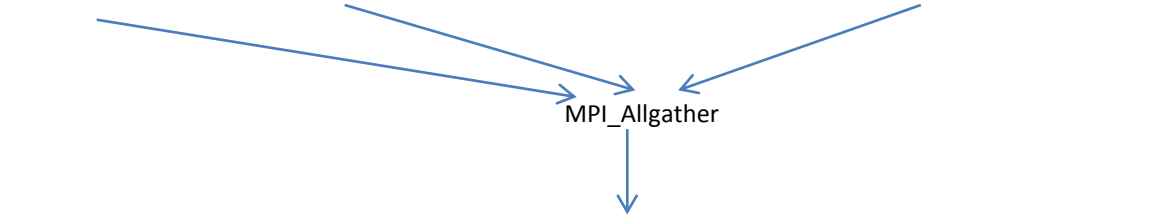
Step 2: Each process runs through the numbers and calculates the factors simultaneously:

Process 0	Process 1	…	Process n
[f1_0, f2_0,…]	[f1_1, f2_1,….]	…	[fn_1, fn_2,…]

(fi_x is the i^{th} factor found by process x)

Step 3: MPI_Allgather for each process to send and receive the elements they computed to all processes:

Process 0	Process 1	…	Process n
[f1_0, f2_0, …]	[f1_1, f2_1, ….]	…	[fn_1, fn_2,…]



f1_0	f2_0	…	f1_1	f2_1	…	fn_1	fn_2	….

All the processes now have an array storing all the factors of p – 1.

Trying out all numbers as candidates for a generator

For the second part, we will do nearly the same: test all the numbers from 2 to p-1 and see if it is the generator. We will again divide the range of numbers evenly among all the processes. Each process will increment its counter when it sees a generator. At the end, we collect the total number of generators to the root process using MPI_Reduce.

Step 4: Divide the range of numbers almost equally among each process. The maximum number is up to p -1

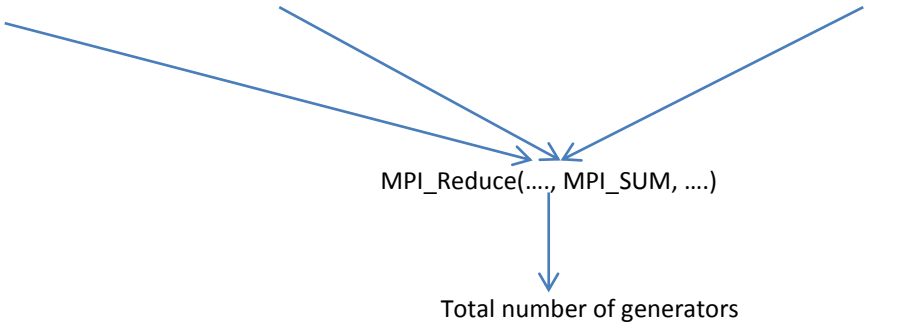
Process 0	Process 1	…	Process n
[2… end_val0]	[starval1…. endval1]	…	[starvaln… endvaln]

Step 5: In each process: for each element g, with all factors f of p-1, if $g^{(p-1)/f} \ mod \ p = 1$ then g is not a generator. The processes compute their counters in parallel.

Process 0	Process 1	…	Process n
Count0	Count1	…	Countn

Step 6: After all processes have finished, we take the sum of all the counters and get the result

Process 0	Process 1	…	Process n
Count0	Count1	…	Countn



Optimization:

Based on our experiment, we notice that the bottleneck is to find all the generators as the range of numbers to be tested is quite large for large prime numbers. Thus, we aim to minimize the computation by only storing the prime factors of p-1. This greatly reduced the times we have to compute $g^{(p-1)/f} \ mod \ p$ for each g as the total number of factors of p-1 to test with are only the number of prime factors of p-1. After that, we calculate the sum of all generator counts from each process using MPI_Reduce with MPI_Sum to the root process. Thus, we have the result in the root process.

Each process will have their arrays to store the factors dynamically computed based on the prime p. The unused slots will have the value of 0. After the processes have received the array that stores all factors, we use quick sort to move all the 0s to the back (which are the unused slots).

Load Balance:

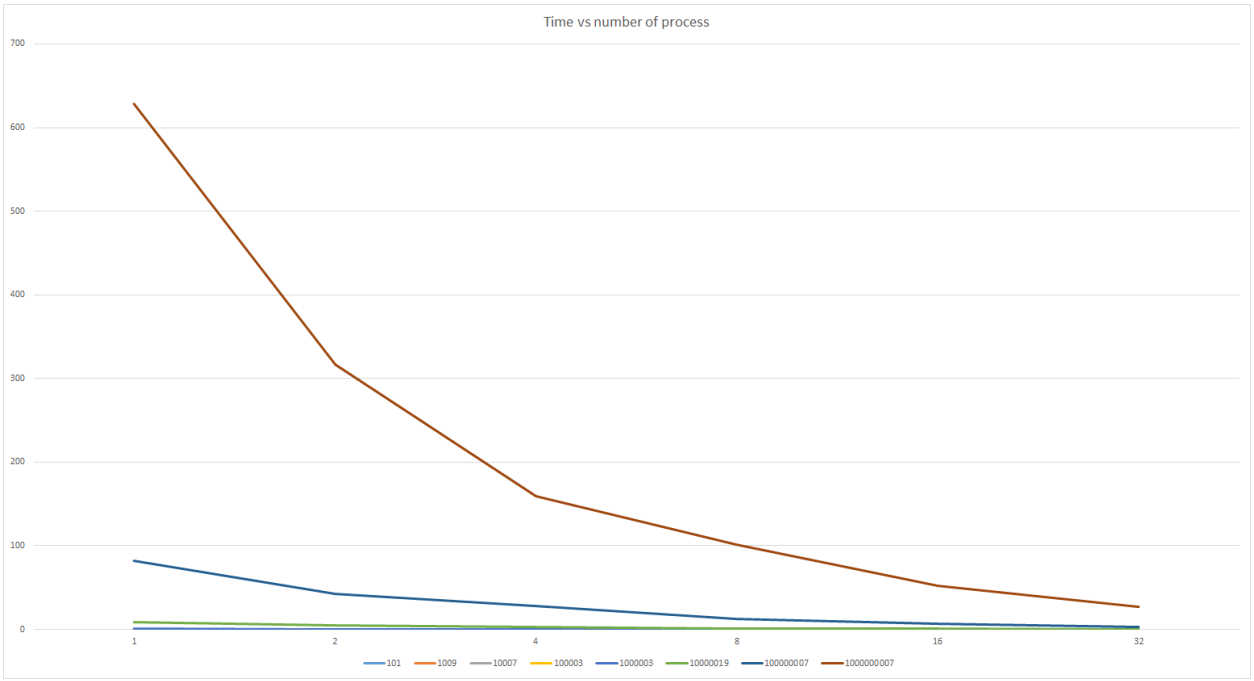
Based on our timing, we see that the run time across all the processes are roughly the same. Thus, we conclude that the load is balanced across the processes in our program.

II. Evaluation

1. **Experiments and testing:**
   The code to make calls to the "time" system call is added to get the detailed information of runtime of each part of the process. Then a script file is written to run the test with number of process in (1, 2, 4, 8, 16, 32) and prime numbers in (101, 1009, 10007, 100003, 1000003, 10000019, 100000007, 1000000007). The results are presented below.
2. **Timing**:
   Below is the table of runtime in seconds.

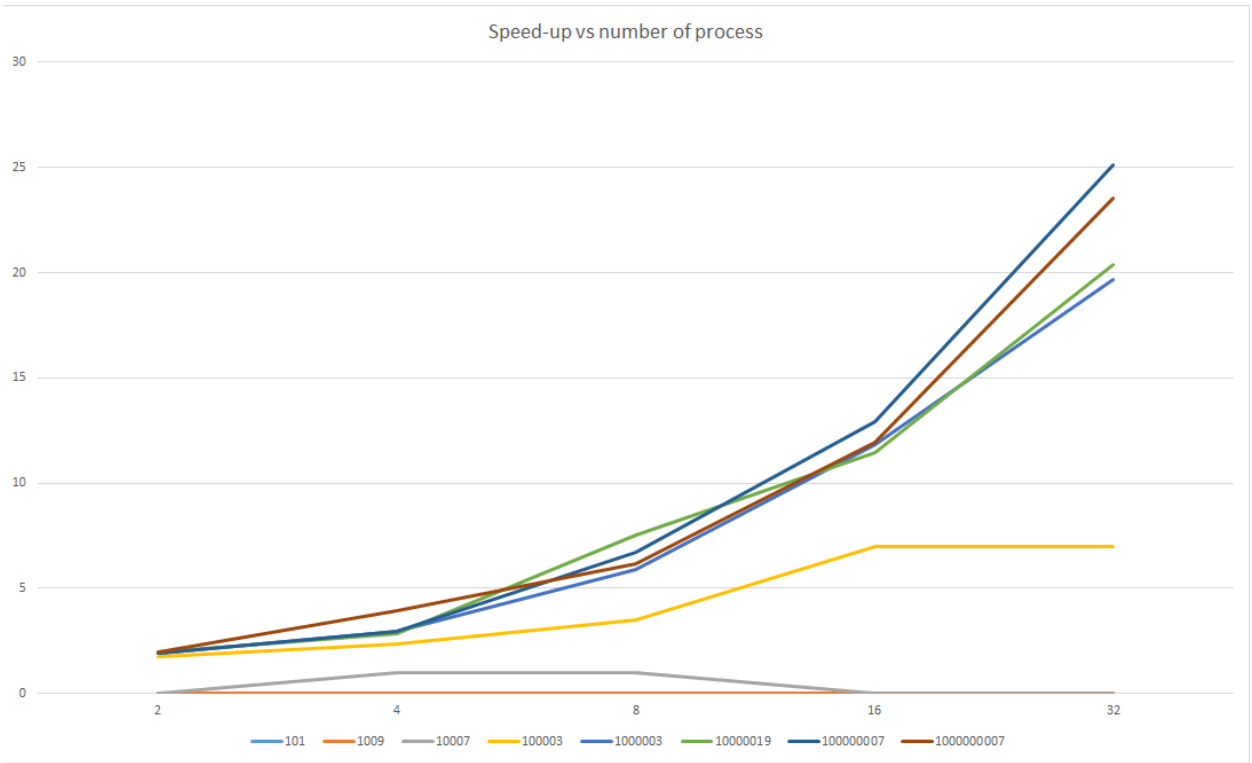
Prime number \ No. of processes	1	2	4	8	16	32
101	0	0	0	0.01	0	0
1009	0	0	0	0.01	0	0
10007	0.01	0	0.01	0.01	0	0
100003	0.07	0.04	0.03	0.02	0.01	0.01
1000003	0.59	0.31	0.2	0.1	0.05	0.03
10000019	9.17	4.7	3.21	1.22	0.8	0.45
100000007	82.06	42.7	28.11	12.25	6.36	3.27
1000000007	628.4	316.7	159.64	101.83	52.59	26.72

3. **Graphs:**



Runtimes vs number of processes graph

From the graph of runtimes vs number of processes, we can see that the runtime is significantly improved when the prime number is big (6 to 10 digits). When the prime number is small, the communication time is comparable to the actual compute time so the parallel runtimes are not improved as expected.



Speed-up vs number of process graph

The speed-up graph shows a similar conclusion:

- Four biggest prime number lines (1000003, 10000019, 100000007 and 1000000007) show a good speed up with about 20 to 25 times with 32 processes.
- The 6 digit prime number 100003 shows a good speed up until 16 processes, then shows no improvement with 32 processes, suggesting that with more than 16 processes, the communication time overhead will actually defeat the speed-up of running in parallel.
- With 3 to 5 digit prime numbers, the speed-up of running in parallel is quickly overcome by the communication time overhead, reaching peak speed up at 4 processes then becomes lower as the number of processes increases.