**3.2 Documents**

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 g

For the first part, we divide the numbers to be tested as the factor of p-1 to each process such that each process has almost equal range of numbers to be tested. 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 compute by the other processes (using MPI\_Allgather) so that each process now know all the factors of p-1. For optimization, we also 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\_val1] [starval1…. endval1] [starvaln… endvaln]

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

**Step 2**: Each process run through the numbers and calculate 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 factor find by process x)

**Step 3**: MPI\_Allgather for each process to send and receive the elements they compute to all process:

Process 0 Process 1 … Process n

[f1\_0,f2\_0,…] [f1\_1, f2\_1, ….] … [fn\_1,fn\_2,…]

MPI\_Allgather

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| f1\_0 | f2\_0 | … | f1\_1 | f2\_1 | … | fn\_1 | fn\_2 | …. |

All the process now have the array stored all the factors of p – 1.

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 process. 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\_val1] [starval1…. endval1] … [starvaln… endvaln]

**Step 5**: In each process: for each element g, with all factors f of p-1, if then g is not a generator. The processes compute their counters concurrently.

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

MPI\_Reduce(…., MPI\_SUM, ….)

Total number of generators

**Optimization:**

Based on our experiment, we notice that the bottleneck is to find all the generators as the range of number to be tested is quite large for large prime number. Thus, we aim to minimize the computation by only storing the prime factors of p. This greatly reduced the times we have to compute for each g as the total elements f are only the number of prime generators. After that, we calculate the sum of all 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 stored the factors dynamically computed based on the prime p. The unused slot will have the value of 0. After the processes have received the array that stored all factors, we use quick sort to get 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 balance across the processes in our program.