Group Project Design Document

CP-631-OC1 Pawel Pomorski

WiLFRID LAURIER UNIVERSITY

Chunxiang Zhang & Vishnu Sukumaran

  Group 2

Contents

[**1: Introduction** 2](#_Toc14972292)

[**2: Problem** 3](#_Toc14972293)

[**3: Overview of the Design** 3](#_Toc14972294)

[**4: The Scope** 5](#_Toc14972295)

[**5: User Interface format** 5](#_Toc14972296)

[**6. Output** 6](#_Toc14972297)

[**7: Performance Testing** 7](#_Toc14972298)

[**8: Deliverables** 8](#_Toc14972299)

Version Control History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Changes |
| 0.1 | 17/07/2019 | V Sukumaran  / C Zhang | An Initial draft of skeleton and Design definition – Updates on Serial Program |
| 0.2 | 25/07/2019 | V Sukumaran  / C Zhang | Updates and results of MPI program |

## **1: Introduction**

The purpose of this document is to address the steps taken to find the best possible solution to a real-world problem through parallel computing programming frameworks like Message Passing Interface (MPI), Compute Unified Device Architecture (CUDA) and also OpenMP.

Message Passing Interface is a language-independent communications protocol. It is portable, platform-independent, and is the existing standard for parallel computing on distributed memory systems. MPI is not a new programming language. It is a collection of functions and macros or a library that can be used in C programs. Most MPI programs are based on the SPMD model - Single Program Multiple Data. This means that the same executable runs in several processes, but the input data makes each copy compute different things.

OpenMP offers a more restricted set of options for thread operations, controlled by compiler pragmas.  In this way, the compiler itself generates threaded code in the executable following the relatively simple pragma directive issued by the programmer. This offers less flexibility than working with threads directly, but in exchange, it makes programming easier and increases program reliability.

Nvidia released the CUDA API for programming its Graphics Processing Units (GPU). It allows the programmer to treat the GPU as a general computing device, without any explicit reference to graphics. CUDA enables developers to speed up compute-intensive applications by harnessing the power of GPUs for the parallelizable part of the computation.

By implementing a solution on parallel computing, the project team is trying to make programs more efficient and better in performance to achieve its goal. Advantages of Parallel Computing over Serial Computing are as follows:

* It saves time and money as many resources working together will reduce the time and cut potential costs.
* It can be impractical to solve larger problems on Serial Computing.
* It can take advantage of non-local resources when the local resources are finite.
* Parallel Computing makes better work of the hardware.

## **2: Problem**

*A prime number (or a prime) is a natural number greater than 1 that cannot be formed by multiplying two smaller natural numbers.*

The project intends to find the five biggest distance between consecutive prime numbers in the chosen range [0, 1000000000].

For example, say the gap between consecutive prime numbers 17 and 19 is equal to 2. The gap between consecutive prime numbers 37 and 41 is equal to 4. Find the best algorithm and approach to determine the largest 5 gaps between a pair of consecutive prime numbers, up to 10^9 (1,000,000,000) using parallel computing methods.

## **3: Overview of the Design**

Upon research, Sieve of Eratosthenes algorithm is a simple, ancient algorithm for finding all prime numbers up to any given limit. It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2.

Given a number n, print all primes smaller than or equal to n. It is also given that n is a small number. Following is the algorithm to find all the prime numbers less than or equal to a given integer n by Eratosthenes’ method:

* Create a list of consecutive integers from 2 to n: (2, 3, 4 till n).
* Initially, let p equal 2, the first prime number.
* Starting from p2, count up in increments of p and mark each of these numbers greater than or equal to p2 itself in the list. These numbers will be p (p+1), p (p+2), p (p+3), etc.
* Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this number (which is the next prime), and repeat from step 3.

When the algorithm terminates, all the numbers in the list that are not marked are prime.

Below is sample pseudocode for the algorithm

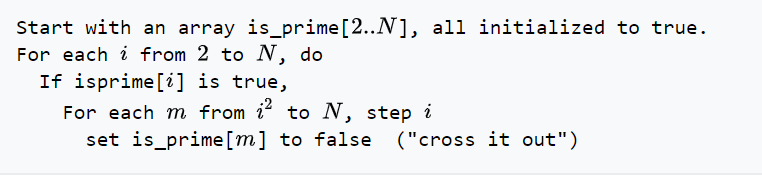


Figure 1: Eratosthenes Algorithm

In Serial Implementation, the above algorithm is used to find the prime numbers and the program keeps on saving the highest distance between prime numbers as it goes through each loop. The execution of the serial program should take more time as the limit of for loop is higher and it is executed by only one processor.

In MPI implementation, Process 0 prints out the information once it receives final values from other processes. Except for first and last process, the computation is mainly divided into two parts where each process will calculate prime numbers for the range [0, 32000] and even computes the operation for the allotted process area (number range). Each process is given a process area by calculating the number range that a process must handle with the help of the assigned number of processors. Then all processes except for the first process sends the first prime number to the previous process to find the distances between the ranges of different processes. By using collective communication, the program finds the distance between recent prime numbers with the largest distance and broadcasts it to other processes to compute the top five biggest distances between any prime numbers.

## **4: The Scope**

The scope of this group project mainly consists of five main aspects

1. Implement the serial algorithm to solve the problem
2. Write an MPI parallel program for the same serial algorithm
3. Write a program which combines both MPI and OpenMP
4. Implement the above programs by using CUDA on GPU.
5. Analyze the above programs and performance with serial program to decide the efficient method to execute the compiled algorithm.

## **5: User Interface format**

|  |
| --- |
| **The largest gap is XXXX1 between prime number XXXXa and prime number XXXXb.**  **The 2nd largest gap is XXXX2 between prime number XXXXc and prime number XXXXd.**  **The 3rd largest gap is XXXX3 between prime number XXXXe and prime number XXXXf.**  **The 4th largest gap is XXXX4 between prime number XXXXg and prime number XXXXh.**  **The 5th largest gap is XXXX5 between prime number XXXXi and prime number XXXXj.** |

Since most of the output is captured in command prompt screen, it is designed to look like the below format.

Note: All the prime numbers (XXXXa, XXXXb, XXXXc, XXXXd, XXXXe, XXXXf, XXXXg, XXXXh, XXXXi, XXXXj) should be in the range [1:10^9].

## **6. Output**

1. Below is the output screen from the Serial Implementation

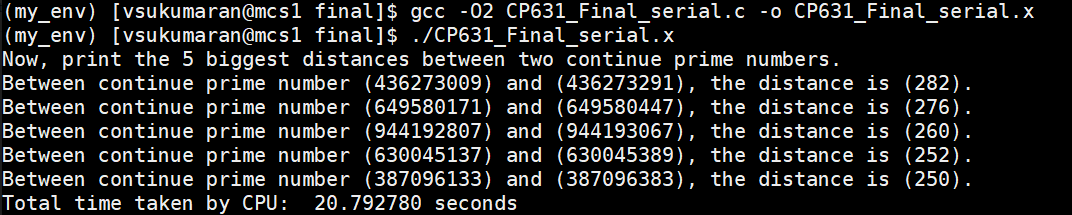


Figure 2: Serial Output

1. Below is the output screen from the MPI Implementation

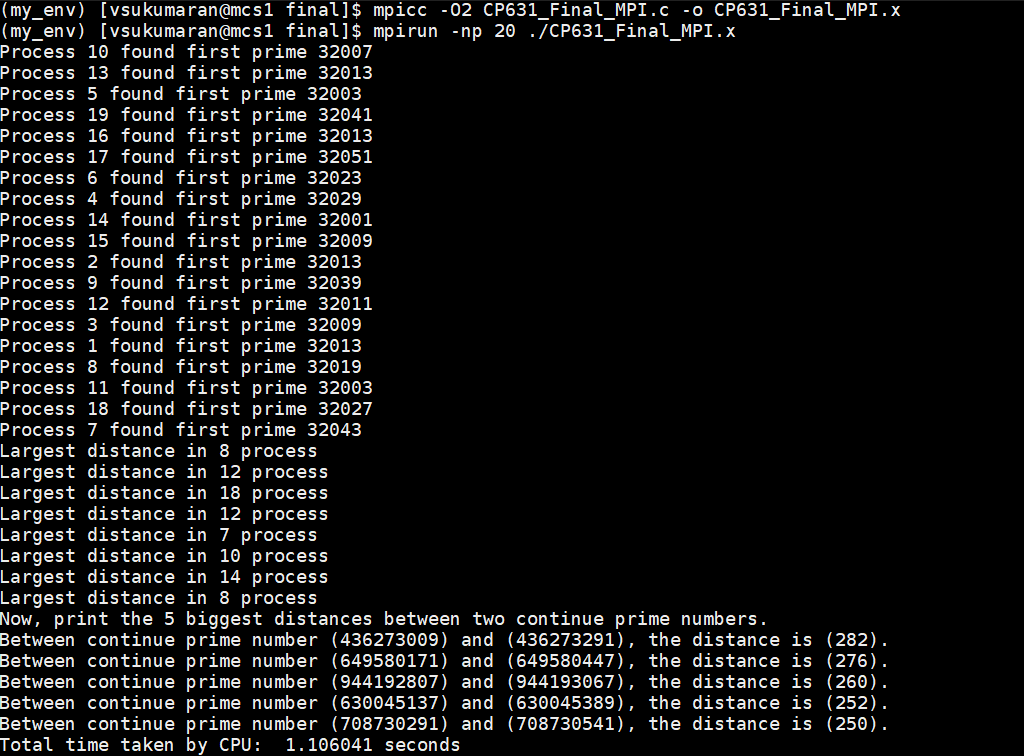


Figure 3: MPI Output

## **7: Performance Testing**

Performance is measured by recording time attribute and thereby calculating Parallel overheard, speed up and efficiency.

**Parallel overhead T o** = pTp – Ts, a perfect parallel program would have a runtime that would decrease as an inverse of the number of processors. In such a program parallel overhead would be zero. In real life programs, it is nonzero and we want to minimize it in our parallel programs.

**Parallel speedup S** = Ts/Tp if the problem is well parallelizable, then S≈ p.

**Parallel efficiency E=** S/p where S is parallel speedup defined above and p is the number of processors. If a problem is well parallelizable, then E ≈ 1.

|  |
| --- |
| Terms and definitions |
| Tp - Time for parallel algorithm  Ts - Time for (best) serial algorithm  p – Number of parallel processes |

|  |  |  |
| --- | --- | --- |
| File Name | Time Taken | Parallel Performance Metrics (Compared with Serial program) |
| CP631\_Final\_serial.c | 20.792780 | Ts = 20.792780 |
| CP631\_Final\_MPI.c | 1.106041 | Tp= 1.106041  P = 20  Parallel overhead To= pTp – Ts = 1.32804  Parallel speed up S=Ts/Tp=18.79928  Parallel efficiency E=S/p=0.9399 |

Table 1: Performance Metrics

## **8: Deliverables**

Please follow the below format to understand the code structure and also the execution format to verify the results.

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Deliverable File Name | File Type | Steps to Execute |
| D1 | CP631\_Final\_serial.c | .c | C: gcc -O2 CP631\_Final\_serial.c -o CP631\_Final\_serial.x  R: ./CP631\_Final\_serial.x |
| D2 | CP631\_Final\_MPI.c | .c | C: mpicc -O2 CP631\_Final\_serial.c -o CP631\_Final\_MPI.x  R: mpirun –np 20 ./CP631\_Final\_MPI.x |

Table 1: Deliverables