Group Project Design Document

CP-631-OC1 Pawel Pomorski

WiLFRID LAURIER UNIVERSITY

Chunxiang Zhang & Vishnu Sukumaran

  Group 2

Contents

[**1: Introduction** 2](#_Toc15463879)

[**2: Problem** 3](#_Toc15463880)

[**3: Overview of the Design** 3](#_Toc15463881)

[**4. Optimization** 4](#_Toc15463882)

[**5: Scope** 6](#_Toc15463883)

[**6: User Interface format** 7](#_Toc15463884)

[**7. Output Screens** 7](#_Toc15463885)

[**8: Performance Metrics** 9](#_Toc15463886)

[**9: Deliverables** 10](#_Toc15463887)

[**10. Conclusion** 12](#_Toc15463888)

[**References** 13](#_Toc15463889)

**Version 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 |
| 1.0 | 30/07/2019 | V Sukumaran  / C Zhang | Final draft and output recording for performance testing |
| 1.1 | 31/07/2019 | V Sukumaran  / C Zhang | Final version |

## **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 frameworks like Message Passing Interface (MPI), Compute Unified Device Architecture (CUDA) and 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. 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 (Heller, 2018). By implementing a solution to parallel computing, the ultimate goal for the team creates programs more efficient and better in performance compared to a serial program. Advantages of Parallel Computing over Serial Computing are as follows (Introduction to Parallel Computing):

* 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**

*Definition: A prime number (or a prime) is a natural number greater than 1 that cannot be formed by multiplying two smaller natural numbers. (Ferrie, 2018)*

The project intends to find the five biggest distances between consecutive prime numbers in the given range between [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. The intended goal is to 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**

After study, the algorithm is a simple and old for finding all prime numbers for a chosen range. It does so by iteratively marking as composite (i.e., not prime) which is the multiples of each prime number, starting with the first prime number, 2.

Given a number n, print all primes smaller than or equal to n (assuming 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 (Priyadarshi)

* 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

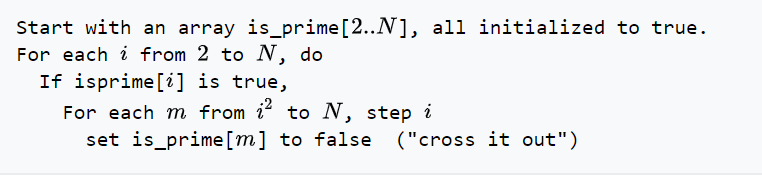


Figure 1: Eratosthenes Algorithm

## **4. Optimization**

There are two major problems with the Sieve algorithm approach

1. It's not very cache-efficient,
2. Tough to parallelize on multiple threads.

The code will be inefficient if the maximum value is larger and the bit array will not fit inside of the outermost processor cache. The main reason for the problem identified is because the algorithm traverses the entire array from beginning to end continuously. This would result in constant cache misses and also the basic algorithm above is very hard to parallelize to multiple threads in an efficient manner. The outer loop cannot be distributed among threads because the inner loop modifies the same data dynamically. The different threads would either come to deadlock or there are high chances of failure in producing the correct results.

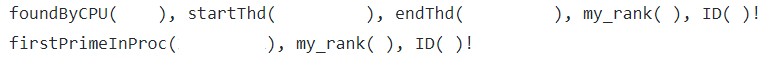
We start by noticing that there's nothing in the algorithm that requires all the multiples of a given prime to be marked on the entire array at once. This means that we can take a smaller section of the array, and mark all the multiples of all the primes on it.

In other words, below steps have to be performed for better efficiency

* Run the basic algorithm up to the square root (maximum value say 32000).
* Divide the rest of the array into approximately CPU cache sized sections (or preferably a bit less), and run the algorithm for each such section.

Since the square root of the maximum value is very small, the first step is usually very quick and doesn't need optimization. The modified algorithm is almost three times faster than the original basic algorithm, even though it does basically the exact same amount of work. So, in serial Implementation, the above technique is used to find the largest gap between prime numbers where the sieve algorithm is used for the range [0, 32000] and the rest with a commonly used method to find prime numbers.

In MPI influenced implementation, Process 0 prints out the results 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 in the range [0, 32000] and even computes the operation for the allotted process area (number range). Each process is given a process area which is defined by their allotted number range calculated by considering the total number of processes. 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. All the CPU calculations are printed in the format foundByCPU and rest of the parallel computation can be verified on the output screen by the line firstPrimeInProc (See below)



OpenMP coding follows the same pattern, but instead of different processors, the handling is distributed over threads. A very similar fashion is followed in CUDA program as well, but as far as the kernel execution is concerned, it runs on a 512-block size.

## **5: Scope**

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

1. Implement a serial algorithm
2. Write an MPI program
3. Write an OpenMP program
4. Write a program which combines both MPI and OpenMP
5. Implement the same algorithm by using CUDA on GPU.
6. Analyze the above programs and record its performances in comparison with the serial code.

## **6: User Interface format**

Since most of the output is captured in command prompt screen, it is designed to look like the below format (Figure 2)

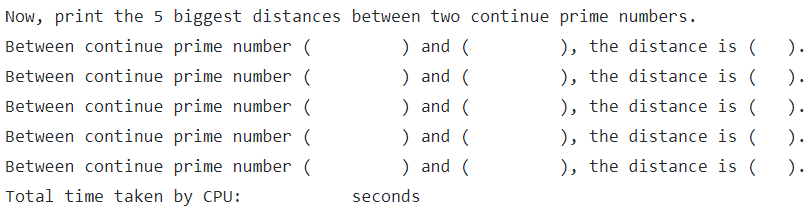


Figure 2: Output Format

Note: All the prime numbers should be in the range [1:10^9].

## **7. Output Screens**

Below is the Output screens of various approaches and results

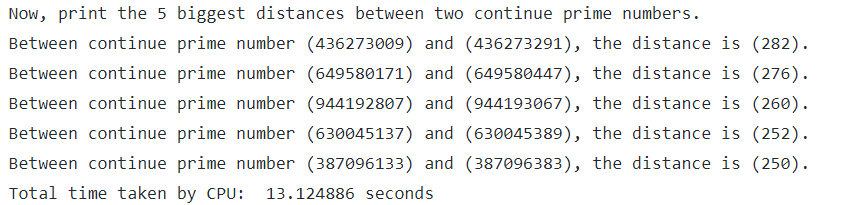


Figure 2: Serial Output

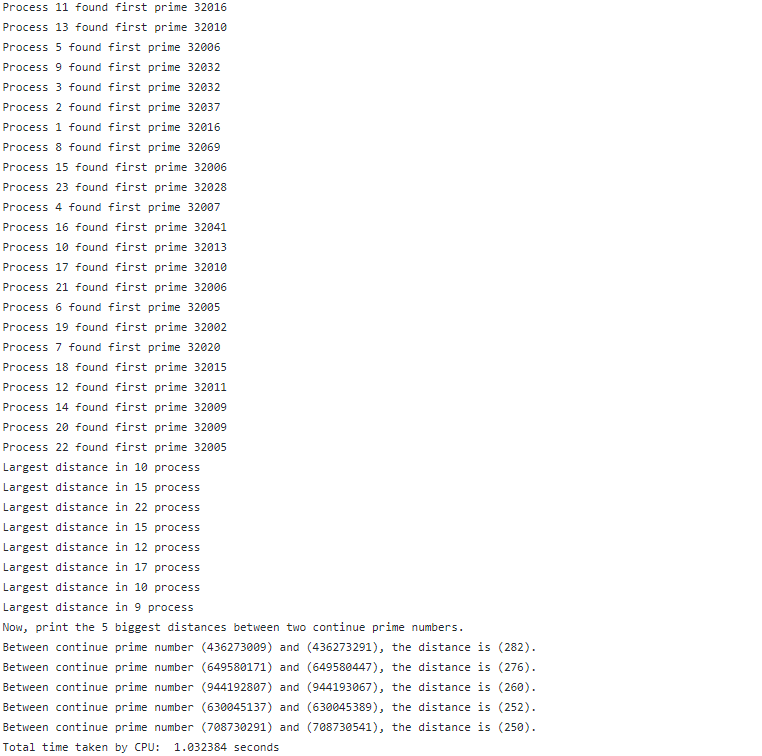


Figure 3: MPI Output

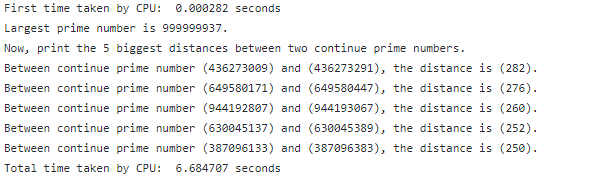


Figure 4: CUDA Output

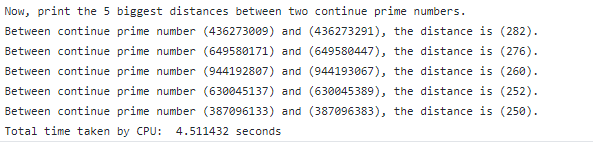


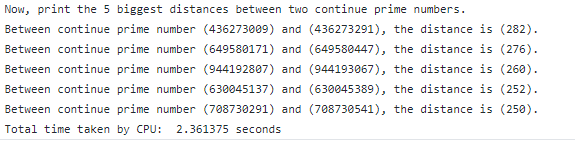
Figure 5: OpenMP Output 

Figure 6: Combined MPI and OpenMP Output

In the range [0, 10^9], 4 distances between consecutive prime numbers have the same distance 250: (387096133, 387096383), (428045491, 428045741), (516540163, 516540413) and (708730291, 708730541). Different programs use different methods to sort the same value distance, then the fifth output must be one in the group and they are different. The definition of NEEDED\_PRIME\_NUM can be changed to larger and more details can be obtained. To verify, the output proves the context when the NEEDED\_PRIME\_NUM is 10.

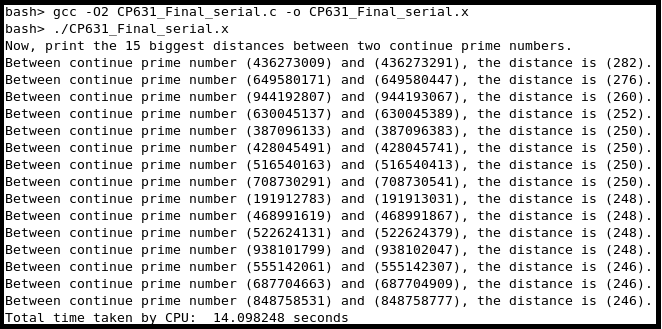


Figure 7: More information about distance 250 between consecutive prime numbers

## **8: Performance Metrics**

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

|  |
| --- |
| 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 (P=24) |
| CP631\_Final\_serial.c | 13.124886 | Ts = 13.124886 |
| CP631\_Final\_MPI.c | 1.032384 | Tp= 1.032384  P = 24  Parallel overhead To= pTp – Ts = 11.625233  Parallel speed up S=Ts/Tp=12.713182  Parallel efficiency E=S/p=0.5297 |
| CP631\_Final\_OpenMP.c | 4.511432 | Tp = 4.511432  P=24  Parallel overhead To= pTp – Ts = 95.14  Parallel speed up S=Ts/Tp=2.909  Parallel efficiency E=S/p=0.1212 |
| CP631\_Final\_MPI\_OpenMP.c | 2.361375 | Tp=2.361375  P = 24 (4 threads and 6 processes)  Parallel overhead To= pTp – Ts = 43.54811  Parallel speed up S=Ts/Tp=5.55815  Parallel efficiency E=S/p=0.2315 |

Table 1: Performance Metrics

While testing the performance of each code based on a few selected numbers of processes, (say for P =20, 24, 28), below is a chart depicting their response times. Figure 7 clearly states that as the number increases, the time for running the process decreases gradually.

Figure 8: Performance Chart

## **9: Deliverables**

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

|  |  |  |
| --- | --- | --- |
| ID | Deliverable File Name | Steps to Execute |
| 1 | CP631\_Final\_serial.c | C: gcc -O2 CP631\_Final\_serial.c -o CP631\_Final\_serial.x  R: ./CP631\_Final\_serial.x |
| 2 | CP631\_Final\_MPI.c | C: mpicc -O2 CP631\_Final\_serial.c -o CP631\_Final\_MPI.x  R: mpirun –np 20 ./CP631\_Final\_MPI.x |
| 3 | CP631\_Final\_cuda.cu | C: nvcc -arch=sm\_60 CP631\_Final\_cuda.cu -o CP631\_Final\_cuda.x  R: ./CP631\_Final\_cuda.x |
| 4 | CP631\_Final\_OpenMP.c | C: gcc -fopenmp -O2 CP631\_Final\_OpenMP.c -o CP631\_Final\_OpenMP.x  R: OMP\_NUM\_THREADS=24 ./CP631\_Final\_OpenMP.x |
| 5 | CP631\_Final\_MPI\_OpenMP.c | C: mpicc -fopenmp -O2 CP631\_Final\_MPI\_OpenMP.c -o CP631\_Final\_MPI\_OpenMP.x  R: OMP\_NUM\_THREADS=4 OMP\_SCHEDULE=guided OMP\_PROC\_BIND=true mpirun -np 5 ./CP631\_Final\_MPI\_OpenMP.x |

Table 2: Deliverables

## **10. Conclusion**

There is clearly no one answer to the problem because the approach is solely dependent on the application of the program and the hardware where it is prepared to run. Out of the computing completed with this assignment, MPI must be chosen if the code is run on a distributed system, e.g. a cluster. If the same program is run on a single computer MPI is not worth the overhead. Most importantly, there is one point to remember before any implementation, each MPI process allocates the same amount of memory, whereas with OpenMP uses a shared memory model. In the case of OpenMP, it is always easy to get started because only a few directives are needed to parallelize for loops. But once we get to know the full OpenMP standard, it is very hard to get the best performance out of the same.

If the prepared simulations can take advantage of the massive parallelism of the GPU, the program can achieve better speedups. Another advantage of GPUs is the improved memory bandwidth and to take advantage of that requires

(1) Problems that mostly fit into the limited GPU memory

(2) Avoiding frequent data transfer from/to GPU

(3) Reasonably regular access patterns.

## **References**

Ferrie, C. (2018, June 12). One is the loneliest prime number. Retrieved from <https://csferrie.com/2018/06/12/one-is-the-loneliest-prime-number/>

Heller, M. (2018, August 30). What is CUDA? Parallel programming for GPUs. Retrieved from <https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html>

Introduction to Parallel Computing. Retrieved from https://www.geeksforgeeks.org/introduction-to-parallel-computing/

Priyadarshi, A. Sieve of Eratosthenes. Retrieved from <https://www.geeksforgeeks.org/sieve-of-eratosthenes/>