Introduction

The aims of this coursework are to study a parallel technologies, mainly focus on OpenCL and OpenMPI with C programming language also look into the performances, advantages, and disadvantages of both technologies and compare them with each other and traditional sequential programme. The task will be perform by a pair. One person will develop a code for OpenCL and another will develop the code for OpenMPI. Then the pair will tune all the codes (Both Parallel and Sequential) together.

The OpenCL and Sequential programmes will run on a single Beowulf 05 with fixed 1000 workgroup. The OpenMPI will run on Beowulf Cluster from 1 core to 256 cores. All the result will be collect and discuss in this report.

Sequential Performances Measurements

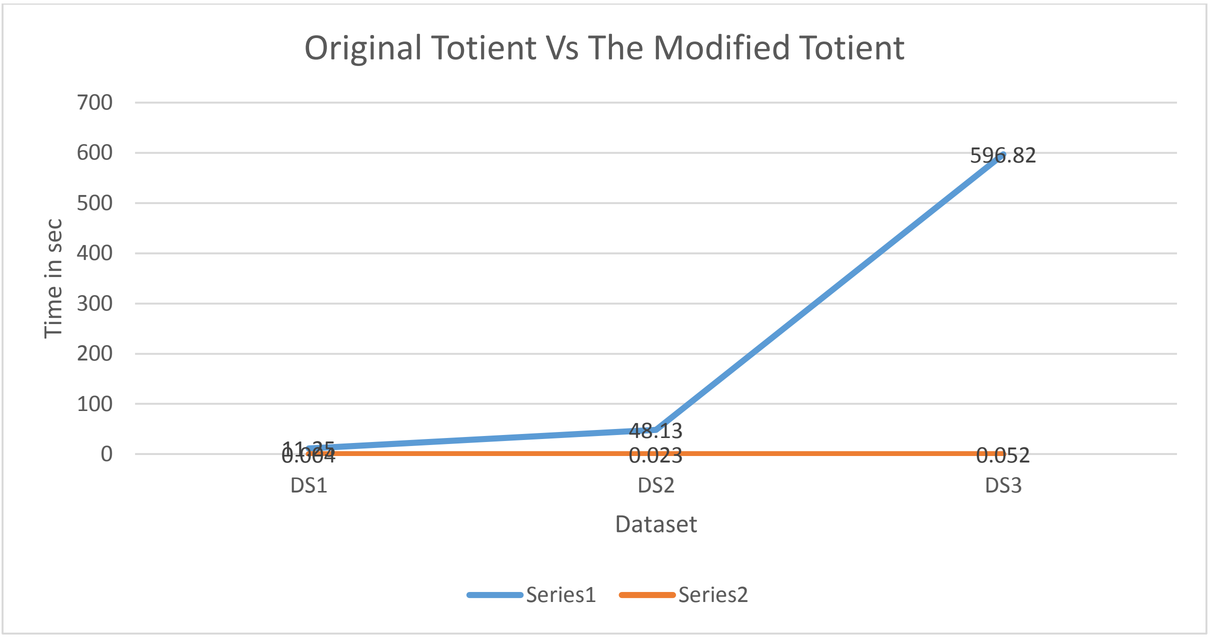

Figure1: Runtime Graphs of sequential programme with an original vs modified Totient function.

We improved the original version of the sequential program. We calculated the sum of Totient between 1 and 15000, 1 and 20000, 1 and 100000 using both original version and the improved version. The run-times are shown in Figure 1 above.

The hotspot of the original version of the sequential program is the function which is used to calculate the Euler Totient computations for a number. In order to calculate the Euler Totient computations of N numbers, the original sequential program calculates and compares each number from 1 to N with N. So with the size of N getting larger and larger, the times that the programme required is also dramatically increased.

The improved version of sequential program calculates the Euler Totient computations in a more efficient way. When calculate the Euler Totient computations for a number N, the improved programme used the Factoring method on N, start with 2(prime number) and then the programme begin its factoring loop from 3 to a square root of N, each loop was increased by 2 since we already factored 2 out of N so the loop will try to factoring only with odd numbers. This saves massive amount of processing times required to run the tasks.

Comparative Parallel Performance Measurements

Runtimes Graphs

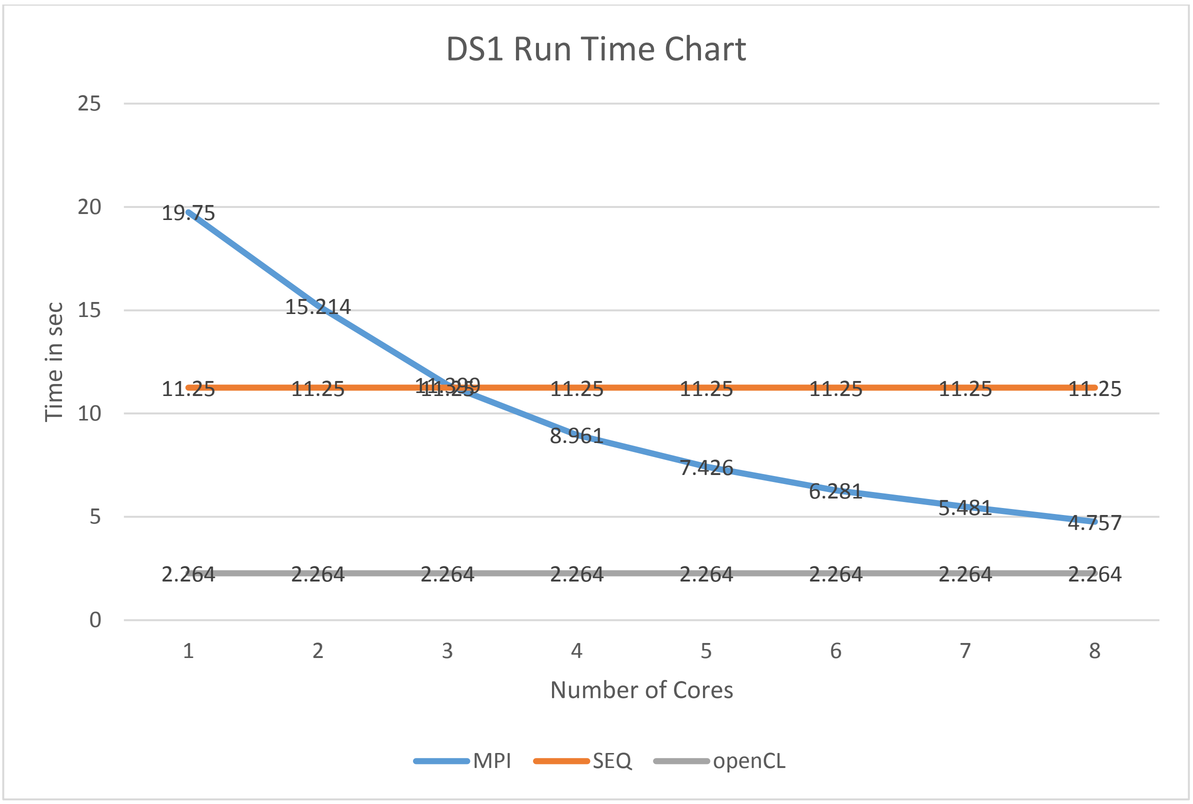

Figure2: Runtime Graphs of DS1 parallel-sequential programmes with an original Totient function.

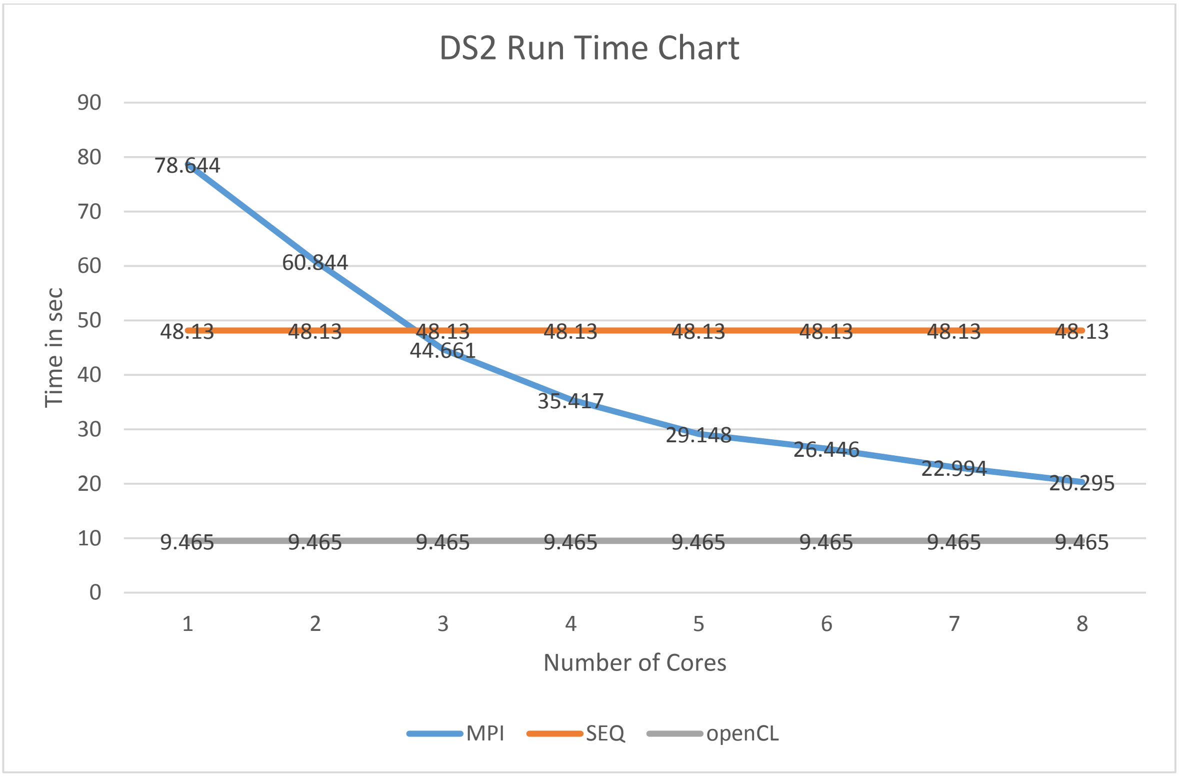

Figure3: Runtime Graphs of DS2 parallel-sequential programmes with an original Totient function.

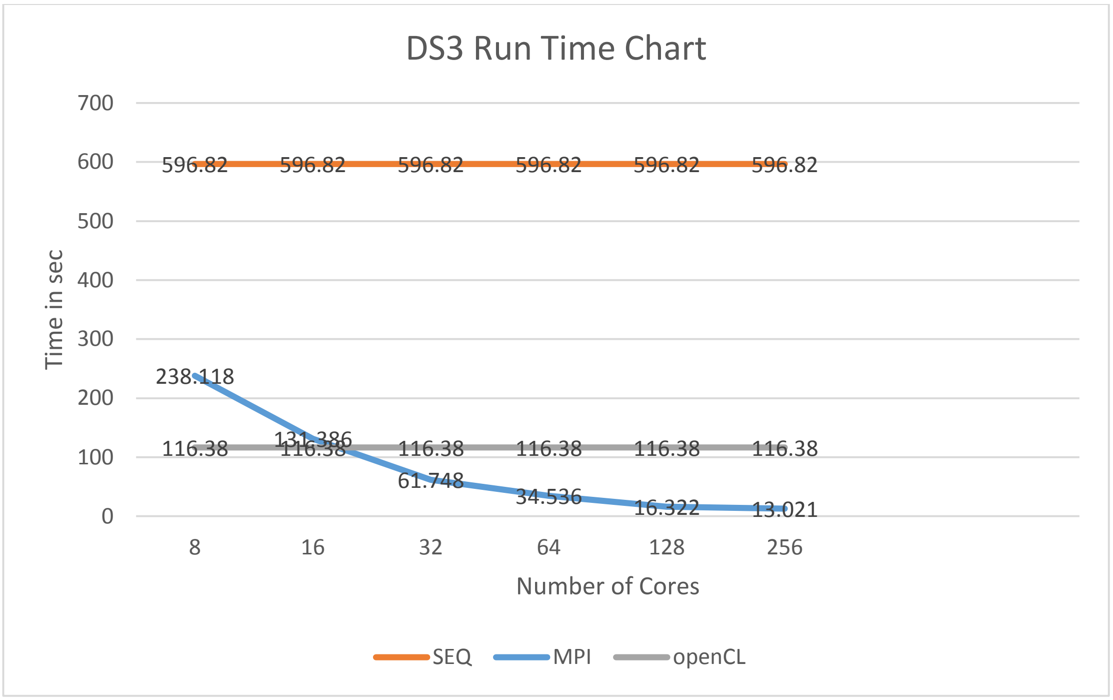

Figure4: Runtime Graphs of DS3 parallel-sequential programmes with an original Totient function.

Runtimes Graphs (Using the modified Totient)

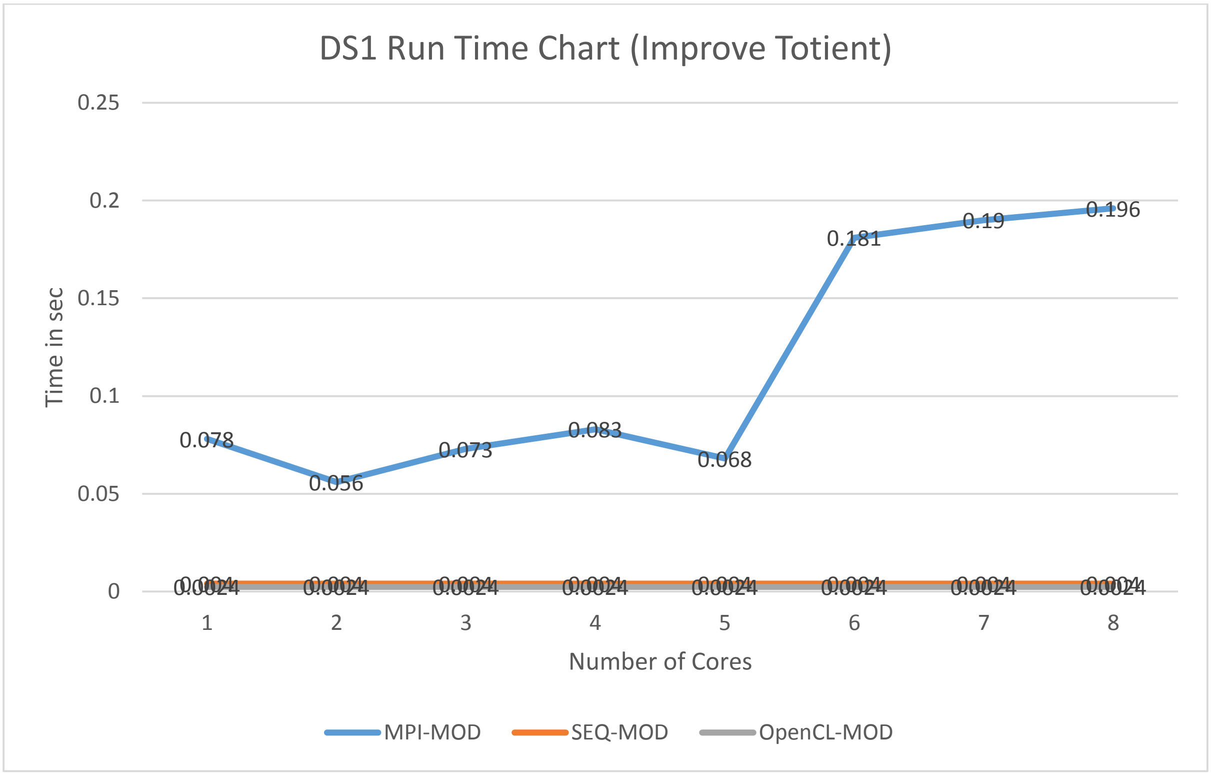

Figure5: Runtime Graphs of DS1 parallel-sequential programmes with a modified Totient function.

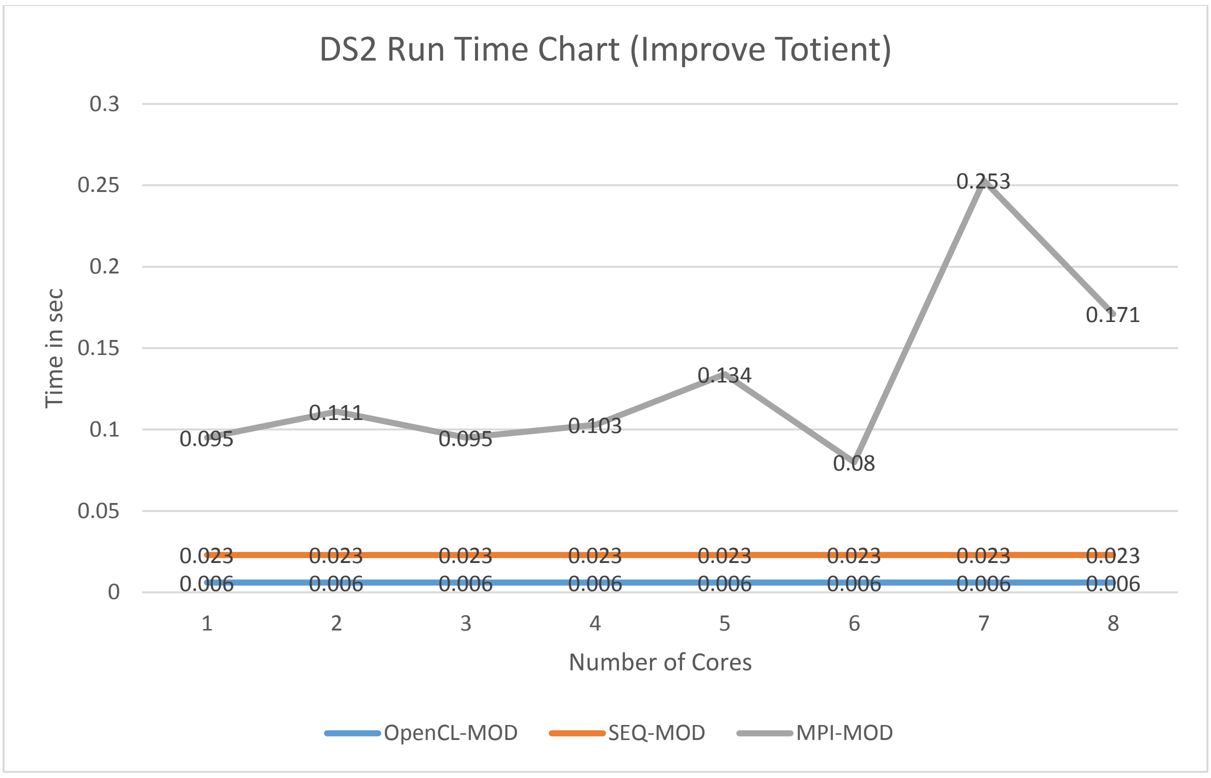

Figure6: Runtime Graphs of DS2 parallel-sequential programmes with a modified Totient function.

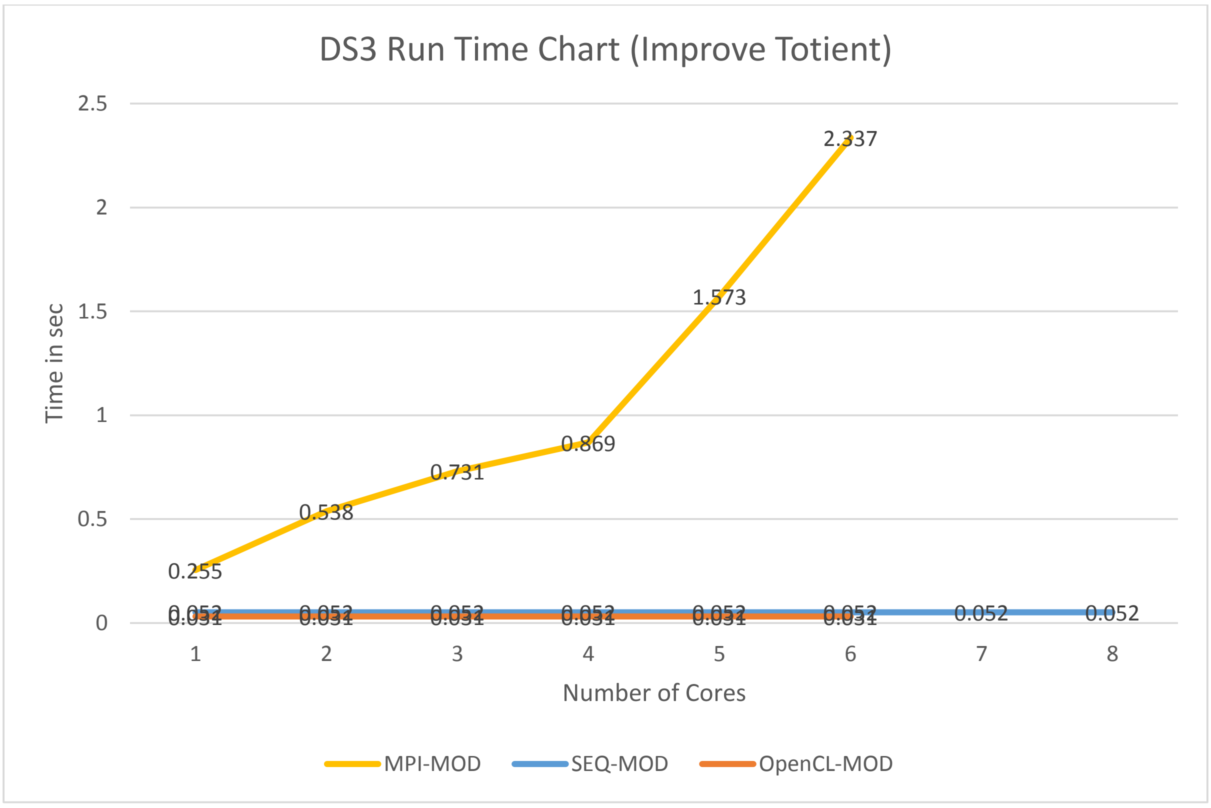

Figure7: Runtime Graphs of DS3 parallel-sequential programmes with a modified Totient function.

Speedups Graphs

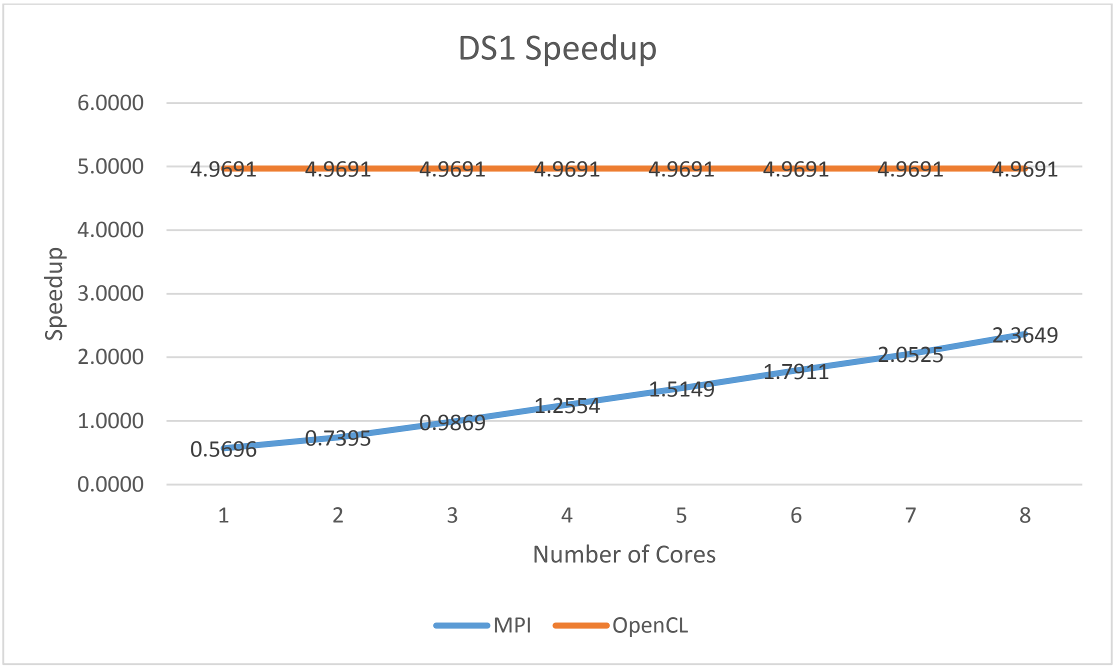

Figure8: Speedup graph of MPI and OpenCL against the Sequential programme on DS1.

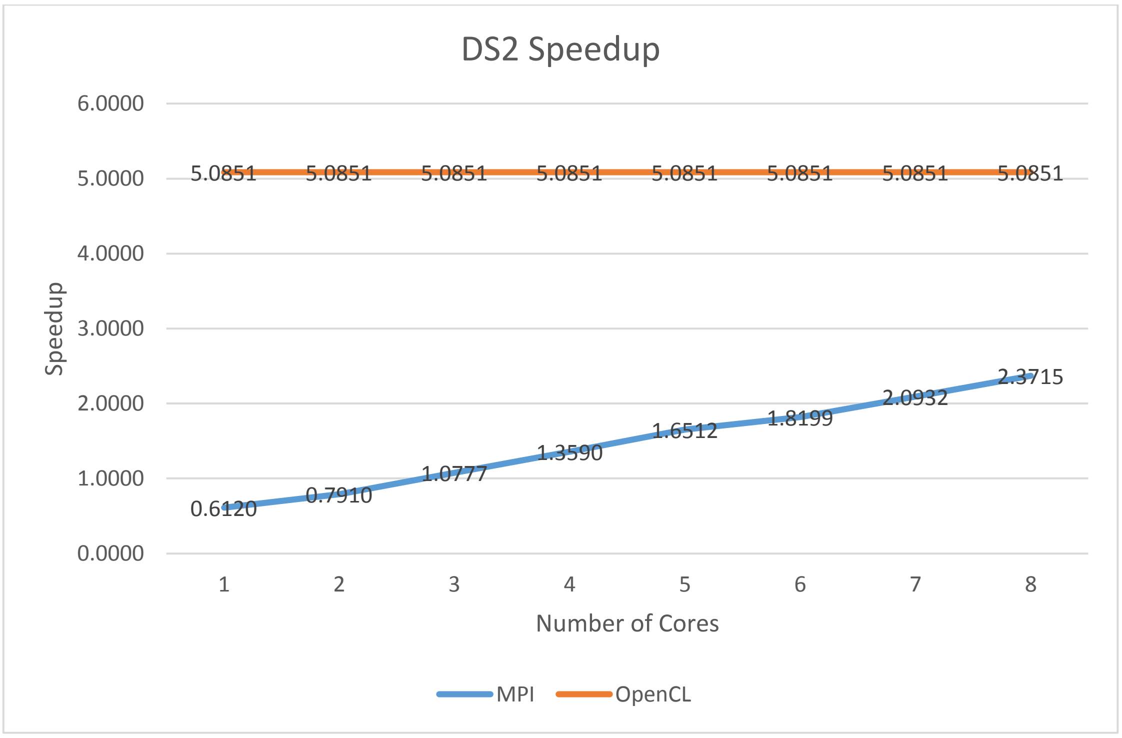

Figure9: Speedup graph of MPI and OpenCL against the Sequential programme on DS2.

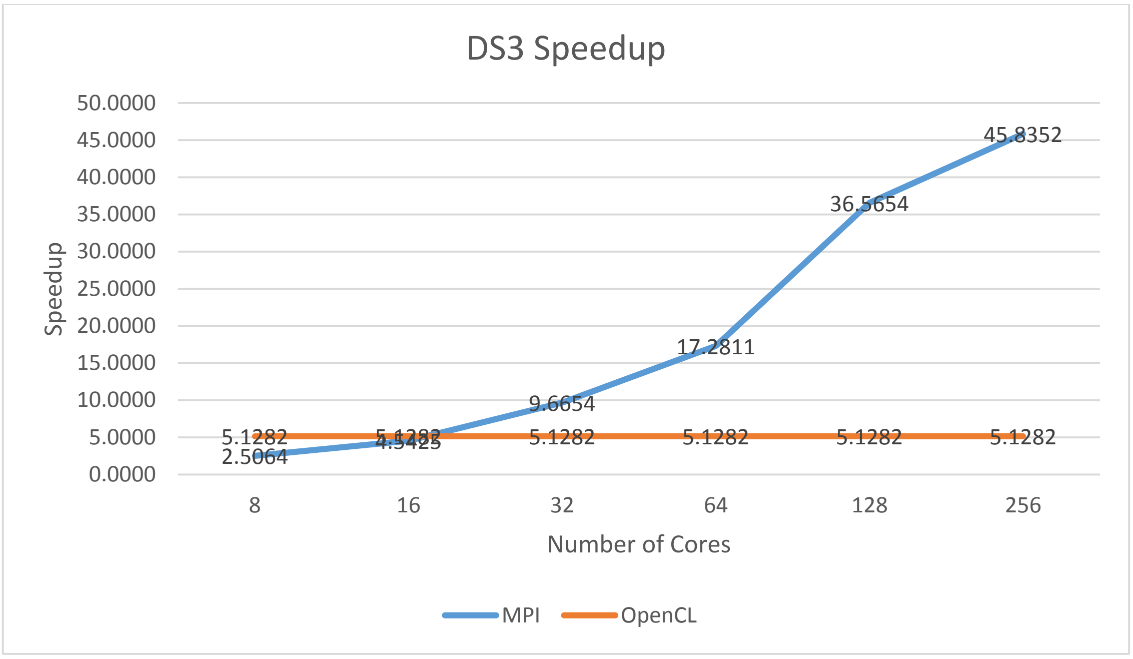

Figure10: Speedup graph of MPI and OpenCL against the Sequential programme on DS3.

Speedups Graphs (Using the modified Totient)

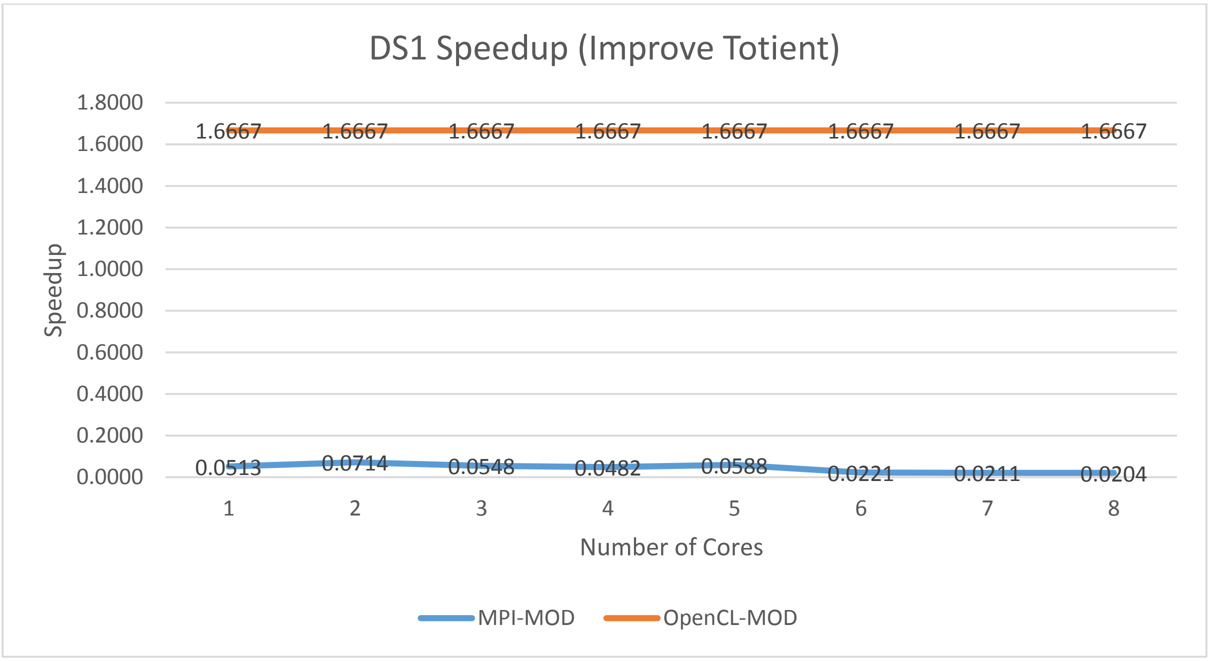

Figure11: Speedup graph using modified Totient in both parallel and sequential programmes on DS1.

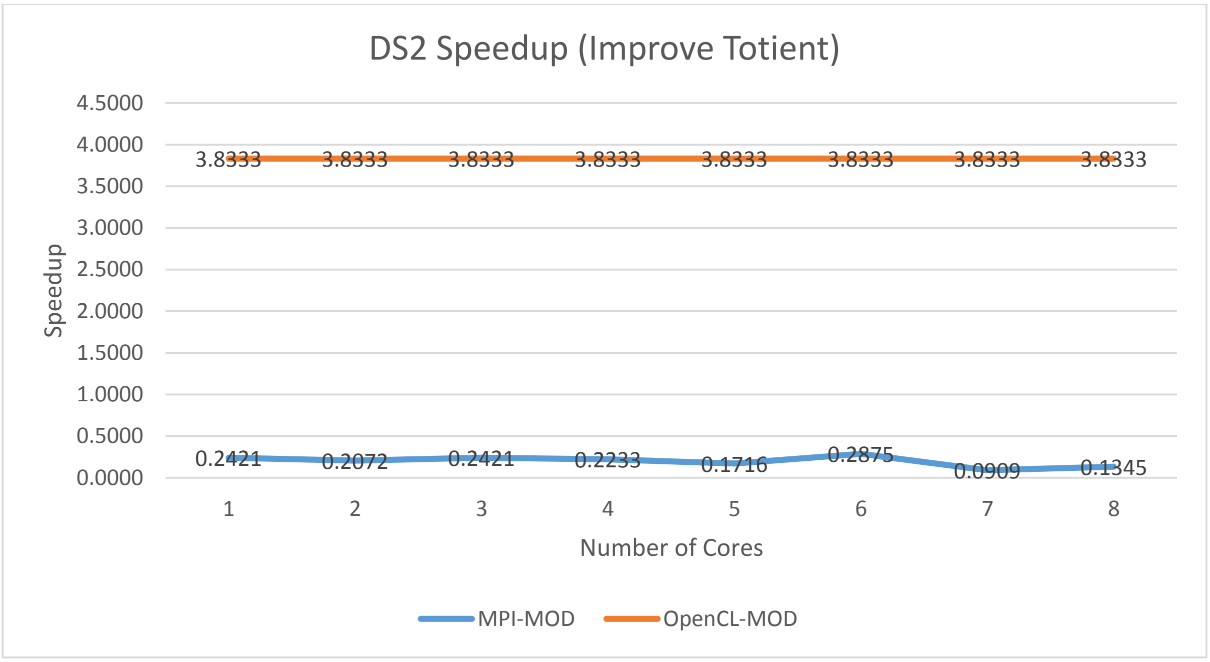

Figure12: Speedup graph using modified Totient in both parallel and sequential programmes on DS2.

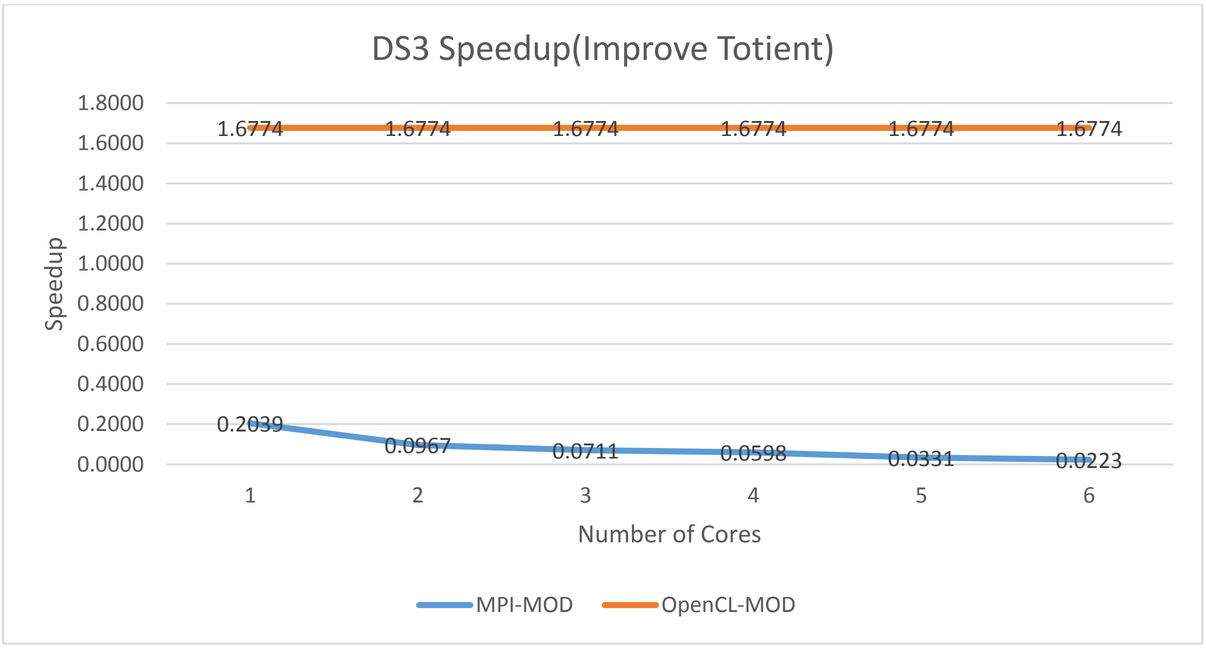

Figure13: Speedup graph using modified Totient in both parallel and sequential programmes on DS3.

Performances Table

	MPI	MPI-MOD	OpenCL	OpenCL-Mod	SEQ	SEQ-MOD
DS1	4.757	0.067	2.264	0.0024	11.25	0.004
DS2	20.295	0.078	9.465	0.006	48.13	0.023
DS3	12.134	0.221	116.38	0.031	596.82	0.052

Table1: Table contain the best speed on from every cores of all programmes.

Comparison

The original Totient function is required a lot of processing time as you can see from the Figure1. So the performances are greatly improved when you applied it with parallel programing. Figure 2 and 3 showed that OpenCL reduced the run time of the programme from the beginning (Around 40 seconds faster than the sequential programme) and the performances of OpenCL is came from a single machine and mainly from GPUs of that machine so the line on the chart stay the same at all core.

OpenMPI also outperform the sequential programme when the number of cores is greater than 3 which mean at below 3 cores, the total times that OpenMPI required to do massaging between the core and processing dataset is more expensive than the sequential programme. OpenMPI performances are increasing as the cores go up but it still 8 cores CPU cannot keep up with the performance of a GPU from OpenCL. In Figure 4, it clearly showed the advantage of the OpenMPI with a large dataset as dataset 3 also an expensive tasks as original Totient function it out perform other programming model. The main advantage is that OpenMPI have no limit on hardware compare to the OpenCL. The cutting point of the OpenMPI in this coursework is around 16 CPU cores as in Figure10 (The speed up chart).

In this report we also collect data with Modified Totient functions on both OpenCL and OpenMPI. The results are very interesting and it's clearly showed advantages and disadvantages from both parallel models. With a modified Totient functions, the tasks required only a few milliseconds on the sequential programme. With all dataset, the performances of OpenMPI are very poor because the task is much cheaper than the messaging time between the CPU cores over the network. You can see it on the all speed up chart (Figure 11, Figure 12 and Figure 13), as the number of cores are increasing the speed is going down because it's required more message to communicate between the cores across the network. On another hand, OpenCL still be able to slightly improve the performance in all dataset because the there is no limit on the communication between the core as in OpenMPI. Furthermore, GPUs are design to perform a parallel tasks.

Programming Model

The OpenMPI development code in this coursework is quite simple (When you understand it). OpenMPI is working base on the communication between multiple machines with a massages. The concept of OpenMPI is simple, you sending a small messages with data required in order to process the specific task that you want other machines on the cluster to help process that task but when you try writing the code you have to considered many thing and carefully write your senders and receivers or else everything will not work. There a lot of problem that we encounter during the coding process as follows.

- First time when we start to write the code, we were confused about how actually the sender and receiver work. The first run of the programme, the programme just waiting infinitely for message to send and receive. We have to swap thing around before we understand how the messaging work in OpenMPI
- Next, after we are able to make the programme run correctly, we found that the programme was running on very low speed and we know that the problem is came from the number of the message that we send out to other machines inside the cluster because we actually send 1 message for each number in the entire range of datasets. So we have to modify the code to send the message equal to the number of processing cores that we want the code top run on them by splitting the data set into multiple chunks by dividing the range of dataset with

the number of cores. To do that you also have to modify your sender and receiver code and we mess thing up lot of time with a minor bug such as missing chunks of data set.

- Printing the results you want, sometime create more problems if you put the print code in the wrong place then it start flooding your entire console with message and you can't capture your result properly so you have to run it again and wasting precise times.

Overall the OpenMPI can run very well if you get the thing right and you don't have to worry about the limitation of the hardware because you have the entire cluster to use with this coursework.

By writing and performing program in OpenCL, we found that OpenCL is powerful with clear structure and functions. So we actually did not have much problem when writing the code. However we still encountered several challenges.

When constructing program, we encountered following problems:

- The problem of choosing suitable group size. At the beginning we try to find a best group size to best performance for every ranges. As bigger group size makes the program faster, we tried to make the group size as big as we can. We also tried to make the group size the same as the size of dataset which need to be parallel. However we found that we cannot make the group size as big as we want because different hardware has its limit on the size of group. As the group size cannot be as big as we want, we try to find a constant suitable group size which is suitable for each dataset. Also, we found that the group size must be able divided the size of the dataset that need to be parallel with no fraction. Finally, we decided to make the group size changeable. We let the user to type in the group size. At the end the group size that we use to construct the report is 1000 for all datasets.
- The problem of determine the type of variables. As the sum of the small range is small and for large range is much larger, we need to decide a suitable variable type to get the result. We firstly use integer type but it is too small for the large range. Finally we have to change the code to use long double to get the result.
- The problem of collecting experiment data. We found that the run-times and the maximum group sizes will be different because of different performances of different hardware (First we use local machine in Linux lab). For example: the maximum group size is 1000 for the bwlf 05 server, but the maximum group size of the computers in the lab is just 100. Finally we collect all the experiment data from the same bwlf 05 server in order to make the comparison.

We think the OpenCL suitable on both original Totient and modified Totient functions. The performance of each programmes is much better than the Sequential programmes (not much for modified sequential programme). The real problem is a limitation of hardware on single computer.

Reflection on Programming Model

The OpenMPI programming model is easy and straightforward to write the code also give you an ability to control the tasks. To be able to control the task you have to create the senders to distribute the task across the network and you also have to write receivers correspond to the senders. Also you can control the list of servers and cores that you want your task to run on, sometime if your programme required a lot of send and receive massage between multiple machines and multiple points of sender and receiver, your code might become really messy.

The OpenMPI performance are depended on many factors, the first thing is a computer architecture that you run your code on. OpenMPI can only run on CPU, when compare it with OpenCL on the single machine OpenCL run much faster than the OpenMPI because of the GPU is designed to run the programme in parallel mode in the first place. But OpenMPI can send the tasks to another machines using massages which mean the OpenMPI have ability to scaling up the number of CPU cores to works on the tasks. If you run the OpenMPI on a computer cluster, your code performances will greatly improve. Another factor that clearly impact performance of the OpenMPI is the tasks itself, if your tasks is expensive to process on the single machine and you really can't improve the code anymore, in this case the OpenMPI will clearly improve the performance of your code. But if the tasks is very cheap to process, the performance may be worse than running on the single machine because the OpenMPI have to split the task and send messages across the network which is time consuming. Also, in Cluster architecture there might be some people working on it as well as your code so the performance can be different depending on the load on the cluster.

The OpenCL programming model are much easier to write the code and make it run on your machine. There are no need to change anything in your code, just wrap it up with a normal host program and you can immediately run it.

The OpenCL can process the task on both GPU and CPU but you don't have an ability to communicate with other machines which mean you can only run your code on the single machine. The performances of the parallel programme is much faster on the single machine compare to the OpenMPI. But since it can run only on the second machine the performances will depend on the number and speed of your CPU and GPU. The performances on the expensive tasks is really high, it might take 3 or more computers with 8 cores CPU in the cluster to gain better performance than a very cheap GPU in the lab. The limit of OpenCL is actually the number of your PCI slots on your computer. Furthermore everyone can buy a personal computer but not everyone could be able to get a cluster.

Appendix A

A.1 Original and Modified Totient function on MPI in the same files.

```
/*
 ================================================================================
 Name       : MPITotient.c
 Author     : Wanchana Ekakkharanon
 Version    :
 Copyright  : Your copyright notice
 Description: The code below contained both the modified and original Totient functions
(Need to uncomment to use). The MPI code is straightforward with equally splitting the rage of
dataset by the number of processes at the beginning of the main method. Then distribute the
chunks of data to Beowulf Cluster and then the main method will create receiver to wait for all
results from all cores to come back and then sum each chucks together and display the results.
================================================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int isdigit(int c);
long phi(long n);
//double sqrt(double x);

long phi(long n) {
  long length, i, t;

  length = 0;
  for (i = 1; i <= n; i++) {

    long x = n;
    long y = i;

    while(y != 0) {
     t = x % y;
     x = y;
     y = t;
    }

    if(x == 1) {
     length++;
    }

  }//END of for loop
  return length;
}
/*
```

```
 * Modify phi method
 * Uncomment for performance testing
 */
//long phi(long n) {
//	long length = 0;
//	long i = 0;
//	// skip number 1 (phi(1)== 1)
//	if (n == 1) {
//		return 1;
//	}
//	length = n;
//	//  next factoring out 2
//	if (n % 2 == 0) {
//		length = length / 2;
//		while (n % 2 == 0) {
//			n = n / 2;
//		}
//	}
//	//then factoring out 3 to sqrt(n)
//	for (i = 3; i <= (int) sqrt(n); i += 2) {
//		if (n % i == 0) {
//			length = length - (length / i);
//			while (n % i == 0) {
//				n = n / i;
//			}
//		}
//	}
//	//if n > 1, it is the last factor
//	if (n > 1) {
//		length = length - length / n;
//	}
//	return length;
//}

long sumTotient(long lower, long upper) {
	long sum = 0;
	long i;

	for (i = lower; i <= upper; i++)
		sum = sum + phi(i);
	return sum;
}

int main(int argc, char * argv[]) {
	long lower, upper, localSum;
	int p, id;
	long sum = 0;
	//double start, end;
	/* Red tape */
	MPI_Status status;
	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	//MPI_Barrier(MPI_COMM_WORLD);
	//start = MPI_Wtime();
	/* Begin Parallel Application Block */

	/*The first process id will split the range of number into multiple chucks
	 *the size of chuck is depended on the size of process
	 */
	if (id == 0) {
		//Checking simple conditions for arguments before distributing the chucks
		if (argc != 3) {
			printf("not 2 arguments\n");
			return 1;
		}
		sscanf(argv[1], "%ld", &lower);
		sscanf(argv[2], "%ld", &upper);

		if (isdigit(lower) && isdigit(upper)) {
			printf("Arguments must be numbers\n");
			return 1;
		}
		if (lower > upper) {
			printf("The first argument must less than the second argument\n");
			return 1;
		}
		/*
		 * Begin splitting the range into chunks the file
		 */
		int fullRange = upper - lower + 1;
		int chunkRange = fullRange / p;
		//sending message to other process id
		long localUpper = upper;
		long localLower = lower;
		for (int i = 0; i < p; i++) {
			localUpper = localLower + chunkRange - 1;
			if (i == 0) {
				sum = sumTotient(localLower, localUpper);
				localLower = localUpper + 1;
			}else{
				if(i==p-1){
					localUpper = upper;
				}
//				printf("Lower is %ld\n", localLower);
//				printf("Upper is %ld\n", localUpper);
```

```
                                        MPI_Send(&localLower, 1, MPI_LONG, i, 1, MPI_COMM_WORLD);
                                        MPI_Send(&localUpper, 1, MPI_LONG, i, 1, MPI_COMM_WORLD);
                                        localLower = localUpper + 1;
                                }
                        }

                        //Creating the waiting point for all process
                        for (int i = 1; i < p; i++) {
                                MPI_Recv(&localSum, 1, MPI_LONG, MPI_ANY_SOURCE, 0,
MPI_COMM_WORLD,
                                                &status);
                                sum = sum + localSum;
                        }
                        printf("C: Sum of Totients  between [%ld..%ld] is %ld\n", lower, upper,
                                        sum);
                        //end = MPI_Wtime();
                        //printf("Total execution time is %f\n", end);
                        /*Else other parallel process will receive the chunks
                         * and find the
                         */
                } else {
                        //get lower and upper
                        MPI_Recv(&lower, 1, MPI_LONG, 0, 1, MPI_COMM_WORLD, &status);
                        MPI_Recv(&upper, 1, MPI_LONG, 0, 1, MPI_COMM_WORLD, &status);

                        //Calling sumTotient()
                        localSum = sumTotient(lower, upper);
//                      printf("C: Sum of Totients  between [%ld..%ld] is %ld\n", lower, upper,
//                                      localSum);
                        //Now send back to process 0
                        MPI_Send(&localSum, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD);
                }
                /* End  Parallel Application Block */
                /* Red tape */

                MPI_Finalize();
                return 0;
}
```

Appendix B

B.1Original Totient function with OpenCL

```
/*
 ============================================================================
 Name       : original.c
 Author     : yuting zhu
 Version     :
 Copyright   : Your copyright notice
 Description: This program is paralleling the original version of sequence program.  The functions in
the kernel are the functions that calculate the Euler totient computations.   These functions calculate
This program put the numbers in the range into the kernel as an array and put Euler totient
computations of each number in the result array.  After running the kernel, the program gets the
result array and adds all the value in the result array.
 ============================================================================
 */


#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <CL/cl.h>
#include "simple.h"

#define DATA_SIZE 10240000

const char *KernelSource =                              "\n"
"long hcf(long x, long y)                              \n"
"{                                      \n"
"  long t;                                  \n"
"                                       \n"
"  while (y != 0) {                            \n"
"    t = x % y;                              \n"
"    x = y;                                \n"
"    y = t;                                \n"
"  }                                    \n"
"  return x;                                 \n"
"}                                      \n"
"                                       \n"
"int relprime(long x, long y)                          \n"
"{                                      \n"
"  return hcf(x, y) == 1;                         \n"
"}                                      \n"
"                                       \n"
" unsigned long euler(long n)                                \n"
"{                                      \n"
"  unsigned long length, i;                             \n"
"                                       \n"
"  length = 0;                               \n"
"  for (i = 1; i <= n; i++)                        \n"
"    if (relprime(n, i))                         \n"
"      length++;                             \n"
"                                       \n"
"  return length;                              \n"
"}                                      \n"
```

```
" __kernel void sumTotient(                              \n"
"   __global float* input,                              \n"
"   __global float* output,                             \n"
"   const unsigned int count)                           \n"
"{                                                \n"
"   int i = get_global_id(0);                           \n"
"     output[i] = euler(input[i]);                      \n"
"}                                                \n"
"\n";


 struct timespec start, stop;

void printTimeElapsed( char *text)
{
  double elapsed = (stop.tv_sec -start.tv_sec)*1000.0
                  + (double)(stop.tv_nsec -start.tv_nsec)/1000000.0;
  printf( "%s: %f msec\n", text, elapsed);
}

void timeDirectImplementation( int count, float* data, float* results)
{
  clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &start);
  for (int i = 0; i < count; i++)
    results[i] = data[i] * data[i];
  clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &stop);
  printTimeElapsed( "kernel equivalent on host");
}

int main (int argc, char * argv[])
{
  cl_int err;
  cl_kernel kernel;
  size_t global[1];
  size_t local[1];

  long lower, upper, num_procs;
  sscanf(argv[1], "%ld", &lower);
  sscanf(argv[2], "%ld", &upper);
  sscanf(argv[3], "%ld", &num_procs);

  if( argc <2) {
    local[0] = 32;
  } else {
    local[0] = atoi(argv[3]);
  }

  printf( "work group size: %d\n", (int)local[0]);
```

```
/* Create data for the run.  */
float *data = NULL;               /* Original data set given to device.  */
float *results = NULL;            /* Results returned from device.  */
long double correct, sum_t;                   /* Number of correct results returned.  */



//int count = DATA_SIZE;
long count = (upper - lower) + 1;
global[0] = count;


data = (float *) malloc (count * sizeof (float));
results = (float *) malloc (count * sizeof (float));

/* Fill the vector with random float values.  */
/*for (int i = 0; i < count; i++)
  data[i] = rand () / (float) RAND_MAX;*/
  for(int i = 0; i < count; i++) {
          data[i] = lower + i;
  }


clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &start);
err = initGPU();


if( err == CL_SUCCESS) {
  kernel = setupKernel( KernelSource, "sumTotient", 3, FloatArr, count, data,
                                       FloatArr, count, results,
                                       IntConst, count);

  runKernel( kernel, 1, global, local);

 clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &stop);

 printKernelTime();
 printTimeElapsed( "CPU time spent");

  /* Validate our results.  */
  /*correct = 0;
  for (int i = 0; i < count; i++)
    if (results[i] == data[i] * data[i])
      correct++;*/
   sum_t = 0;
```

```
    for(long i = 0; i <= count; i++) {
            sum_t = sum_t + results[i];
    }

   /* Print a brief summary detailing the results.  */
  /*printf ("Computed %d/%d %2.0f%% correct values\n", correct, count,
        (float)count/correct*100.f);*/
   printf("The sum is %Lf\n", sum_t);

   err = clReleaseKernel (kernel);
   err = freeDevice();

   timeDirectImplementation( count, data, results);

  }


  return 0;
}
```

B.2 Modify Totient function with OpenCL

```
/*
 ==============================================================================
 Name        : improve.c
 Author      : Yuting zhu
 Version     :
 Copyright   : Your copyright notice
 Description: This program is paralleling the improve version of sequence program.  The functions in
the kernel are the functions that calculate the Euler totient computations.   These functions calculate
the Euler totient computations of N by divided N by 2 until N cannot be divided and then divided by
3+i*2 where 3+i*2<square n, until N cannot be divided.   This program put the numbers in the range
into the kernel as an array and put Euler totient computations of each number in the result array.
After running the kernel, the program gets the result array and adds all the value in the result array.
 ==============================================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <CL/cl.h>
#include "simple.h"

#define DATA_SIZE 10240000

const char *KernelSource =                 "\n"
            "long euler (unsigned long n) {                                            \n"
  "         long length =0;                                                   \n"
  "   long i = 0;                                    \n"
  "       if(n==1) {                                                                           \n"
  "                 return 1;                                                                  \n"
          "}
                  \n"
          "       length = n;
\n"
  "       if(n%2 == 0) {                                                       \n"
          "       length = length / 2;                \n"
          "       while(n%2 == 0) {                                                           \n"
          "                  n = n / 2;
\n"
          "       }
                  \n"
          "}
                  \n"
          "for (i = 3; i*i <= n; i+=2) {                           \n"
          "       if(n%i == 0) {                                                              \n"
          "                  length = length - (length / i);    \n"
          "       }
                  \n"
          "       while(n%i == 0) {                                                           \n"
          "                  n = n / i;
\n"
          "                  }
                  \n"
  "}
          \n"
  "if(n > 1) {                                                                        \n"
          "       length = length - length / n;                 \n"
          "}
                  \n"
          "return length;                                                             \n"
  "}
          \n"
  "__kernel void sumTotient(           \n"
  "   __global float* input,            \n"
  "   __global float* output,           \n"
  "   const unsigned int count)         \n"
  "{                                   \n"
  "   long tid = get_global_id(0);        \n"
  "     output[tid] += euler(input[tid]);  \n"
  "}                                   \n"
  "\n";

 struct timespec start, stop;

void printTimeElapsed( char *text)
{
  double elapsed = (stop.tv_sec -start.tv_sec)*1000.0
                + (double)(stop.tv_nsec -start.tv_nsec)/1000000.0;
  printf( "%s: %f msec\n", text, elapsed);
}

/*void timeDirectImplementation( int count, float* data, float* results)
{
  clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &start);
  for (int i = 0; i < count; i++)
    results[i] = data[i] * data[i];
  clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &stop);
  printTimeElapsed( "kernel equivalent on host");
}*/

int main (int argc, char * argv[])
{
  cl_int err;
  cl_kernel kernel;
  size_t global[1];
  size_t local[1];

  long lower, upper, num_procs;
  sscanf(argv[1], "%ld", &lower);
  sscanf(argv[2], "%ld", &upper);
  sscanf(argv[3], "%ld", &num_procs);

  if( argc <2) {
    local[0] = 32;
  } else {
    local[0] = atoi(argv[3]);
  }

  printf( "work group size: %d\n", (int)local[0]);



  /* Create data for the run.  */
  float *data = NULL;               /* Original data set given to device.  */
  float *results = NULL;            /* Results returned from device.  */
  long  sum_t;                    /* Number of correct results returned.  */



  //int count = DATA_SIZE;
  long count = (upper - lower) + 1;
  global[0] = count;
  data = (float *) malloc (count * sizeof (float));
  results = (float *) malloc (count * sizeof (float));

  /* Fill the vector with random float values.  */
  /*for (int i = 0; i < count; i++)
    data[i] = rand () / (float) RAND_MAX;*/
    for(int i = 0; i < count; i++) {
            data[i] = lower + i;
    }

  clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &start);
  err = initGPU();


  if( err == CL_SUCCESS) {
    kernel = setupKernel( KernelSource, "sumTotient", 3, FloatArr, count, data,
                                               FloatArr, count, results,
                                               IntConst, count);

    runKernel( kernel, 1, global, local);

   clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &stop);

   printKernelTime();
   printTimeElapsed( "GPU time spent");

    /* Validate our results.  */
    /*correct = 0;
    for (int i = 0; i < count; i++)
      if (results[i] == data[i] * data[i])
        correct++;*/
     sum_t = 0;
     for(long i = 0; i <= count; i++) {
            sum_t = sum_t + results[i];
     }

    /* Print a brief summary detailing the results.  */
   /*printf ("Computed %d/%d %2.0f%% correct values\n", correct, count,
           (float)count/correct*100.f);*/
    printf("The sum is %ld\n", sum_t);

    err = clReleaseKernel (kernel);
    err = freeDevice();

   // timeDirectImplementation( count, data, results);
  }
  return 0;
}
```