Evaluating Low-level Parallel Programming models

Submitted by:
Max M BAIRD (H00153951)
Boris MOCIALOV (H00180016)

March 6, 2015
Contents

1 Introduction 1

2 Sequential Performance Measurements 1

3 Comparative Parallel Performance Measurements 2
   3.1 Runtimes ...................................................... 4
   3.2 C+MPI Comparative Runtimes .............................. 4

4 Programming Model Comparison 7
   4.1 OpenCL ......................................................... 8
   4.2 C+MPI .......................................................... 8

5 Reflection on Programming Models 9

Appendix A 14
## List of Figures

1. Comparative Parallel Performance Measurements of OpenCL vs C+MPI .................................................. 3
2. Runtime comparisons Implementation1 vs Implementation2 [1-15000] dataset ............................................. 5
3. Runtime comparisons Implementation1 vs Implementation2 [1-30000] dataset ............................................. 6
4. Runtime comparisons Implementation1 vs Implementation2 [1-100000] dataset ........................................... 7
1. SIMP model ........................................................................................................................................ 12
1. SPMD model ........................................................................................................................................ 13
Listings

1  Max M Baird - kernel3.c .................................................. 15
2  Max M Baird - loader.c .................................................... 15
3  Max M Baird - loader.h .................................................... 16
4  Max M Baird - simple.c .................................................... 16
5  Max M Baird - simple.h .................................................... 22
6  Max M Baird - totient.c ................................................... 25
7  Boris Mocialov - Implementation1 ....................................... 28
8  Boris Mocialov - Implementation2 ....................................... 31
9  Boris Mocialov - Implementation3 ....................................... 36
10 Boris Mocialov - Implementation4 ....................................... 41
1 Introduction

The purpose of this report is to demonstrate the delivered requirements concerning the implementation and parallelisation of the sum of Euler totient computations over a range of integer values.

In brief, the Euler totient function computes, for a given integer, the number of integers which are relatively prime to the given integer. The sum of Euler totient computations therefore computes the summation of relatively prime numbers for a linear progressing range of input.

The provided C implementation of the described algorithm is by nature sequential with scope for optimization through parallelisation. It is worthy of noting that optimization could have also been achieved via writing an optimized implementation of the algorithm, however, if this was done improvements through parallelizing an already optimized algorithm would have been minimal. The developers did not believe the true goal of the task to be optimizing the algorithm, but rather obtaining speed up through parallel approaches.

Each developer attempted optimization through parallelization through a selected parallel technology; the technologies considered were C with MPI for explicit message passing and C with OpenCL for off-loading computations to a GP-GPU.

MPI is essentially a message passing library specification which has bindings to multiple programming languages, here, the binding to the C programming language was utilized. MPI is based on Single Program Multiple Data (SPMD) model where every processor will run the same program but will, in general, execute different instructions simultaneously.

In contrast, OpenCL promotes a heterogeneous approach which combines different kinds of hardware, compatible with the OpenCL specification, to take advantage of the combined processing power. The programming model here is Single Instruction Multiple Threads (SIMT) which promotes data parallelism. The specific device targeted in this report is the GP-GPU which is used for the offloading of computationally intense regions when calculating the summation of Euler totient computations.

Through completing this coursework, skills with respect to attaining a deeper understanding of the requisite thought processes in the realm of parallel programming are expected to be developed. Also, there will be technology and analysis skills developed.

2 Sequential Performance Measurements

Analysis of the sequential algorithm was performed in order to measure performance and identify possible areas of improvement in terms of locating code hotspots and how the program made use of CPU cache. The analysis tools used are as follows:

- GNU Profiler (gprof)
- cachegrind

The GNU Profiler was used to profile the code so that areas of the code where the majority of the execution time is spent since these areas may prove to be good
candidates for parallel optimizations. Cachegrind was used as a cache profiler to pinpoint sources of cache misses, and to count the data and memory references within the program. For both tools, the analysis was executed in triplicate; one instance for each stipulated input:

- **DS1:** Calculating the sum of totients between 1 and 15000
- **DS2:** Calculating the sum of totients between 1 and 30000
- **DS3:** Calculating the sum of totients between 1 and 100000

The analysis using gprof revealed two functions (hcf and relprime) of the sequential code which were called most often when performing computations. The percentage of time and calls spent with these functions grew consistently with each input. Throughout the execution of each input values, profiling revealed that over 90% of the computation was spent with the hcf and relprime functions. Analysis using cachegrind did not reveal any significant areas of inefficiency, the data and cache miss rates were generally quite low (less than 0.00%).

Given the above analyses, the candidate of the sequential implementation most applicable for parallelization was identified to be the operations performed by the hcf function.

### 3 Comparative Parallel Performance Measurements

A comparison of the runtime measurements for the parallel approaches is depicted in the table below, the numerical values represent the median of three executions. Best time is taken for C+MPI.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>OpenCL Time Taken (msec)</th>
<th>C+MPI Time Taken (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 15000</td>
<td>6118.067192</td>
<td>6018.833333</td>
</tr>
<tr>
<td>1 - 30000</td>
<td>24758.187099</td>
<td>25595.11111</td>
</tr>
<tr>
<td>1 - 100000</td>
<td>290418.934736</td>
<td>299966.3333</td>
</tr>
</tbody>
</table>

Table 1: Comparative Parallel Performance Measurements of OpenCL vs C+MPI

Graph representation of the comparison of the runtime measurements for the parallel approaches is presented in the diagram below, where the x-axis corresponds to the ratio:

Parallel runtime/Sequential runtime

Given "speed-up" ratio will be used throughout this report to compare runtimes of various or same implementations. It is worth noting that the higher ratio is, the slower the runtime of a particular implementation.
From the given above figure it can clearly be seen that C+MPI outperforms OpenCL using the first dataset, namely 1-15000 items. Later, for bigger datasets, OpenCL takes over and the performance is nearly the same for bigger datasets for both technologies.

The reader must keep in mind that the sequential time in the case of this comparative graph would be located on points (1-15000, 1), (1-30000, 1) and (1-100000, 1). For the sake of putting more strength on comparison between the two used technologies, the sequential runtime is omitted to keep graph focused on the runtime of both OpenCL and C+MPI technologies. But the reader can always mentally draw a horizontal line on y=1.0 to find out where the sequential runtime would be located.
3.1 Runtimes

Sequential Execution Runtimes. Measurements obtained for sequential execution are taken as a baseline for all the comparisons throughout the report.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Sequential Time Taken (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 15000</td>
<td>13027.543959</td>
</tr>
<tr>
<td>1 - 30000</td>
<td>56013.734887</td>
</tr>
<tr>
<td>1 - 100000</td>
<td>691313.435465</td>
</tr>
</tbody>
</table>

Table 2: Sequential Execution Runtimes

3.2 C+MPI Comparative Runtimes

Table is given in Appendix A, which presents runtimes for all techniques that were tried out to find the best parallelisation method for the given problem.

Three general techniques were used to achieve parallelisation using C+MPI and their corresponding performances were measured to find the best technique using ”speed-up” as a quantification method.

- Implementation 1 - Partitioning - the whole range was divided into equal partitions/chunks of data items.
- Implementation 2 - Partitioning with load balancing - approximate load balancing was utilised to get every partition/chunk to contain approximately equal sum of elements.
- Memoization - another improvement to the initial parallel version of the program included memoization applied to all processes in the system.
  Memoization was split into two versions, where
  - version 1 - implemented with load balancing (Implementation 3) and
  - version 2 - without load balancing (Implementation 4)

From the presented table it can be seen that the most efficient versions are: ‘Implementation 1’ and ‘Implementation 2’ (highlighted with light blue color).
Below are the run-time comparisons presented on graphs, where y-axis corresponds to the ”speed-up” ratio used throughout this report, while x-axis tells the number of processes the program was distributed to solve a particular task in cooperative fashion.
Figure 2: Runtime comparisons Implementation1 vs Implementation2 [1-15000] dataset
Figure 3: Runtime comparisons Implementation1 vs Implementation2 [1-30000] dataset
A general pattern can be inferred from these graphs, which tells us that load balancing works better as more and more processes come into play. This should be obvious since it is better to distribute relatively evenly all the workload over as many processes as possible as to distribute unevenly the workload over as many processes as possible (though this development is not instantly seen when the amount of processes passes number 8).

Another aspect to pay attention to is the memoization technique. The results were expected to be much better than with the load balancing, but, on the contrary, results came to be as poor as the sequential implementation of the program. This can be justified by the extra loops that are needed to find factors of a number presented to "euler" function.

The used mathematical formula in this case is: \( gcd(m, n) = 1, then \phi(mn) = \phi(m) \times \phi(n) \) which is supposed to utilise the multiplicative property of \( \phi \).

In this case, the previously calculated \( \phi \) values can be stored in global array for the future lookup by all the processes.

What may be preventing optimised performance is the distribution factor of the computation. Since the shared memory is being used, it has to be synchronised between all the other processes once the reference is updated.

4 Programming Model Comparison

Each technology places emphasis on a different programming model, for OpenCL technology this was Single Instruction Multiple Threads (SIMT) and for C MPI it was Single Program Multiple Data (SPMD). Applying different models to the same...
problem domain (in this case the totient application) can prove to have advantage or disadvantages subject to the characteristics of the algorithm attempting to be parallelized.

4.1 OpenCL

For the OpenCL implementation the obvious area for parallelization was initially taken, this meant parallelizing the call of the function which performs the Euler calculations. This appeared to be a good area for parallelization since the calculation of each value for the range of input has no dependency on the calculation of any other value. However, this proved to be quite slow as was described on the blog post made on February 28th, 2015. At this point, it is realized that the reason for this was two fold; firstly when executing the kernel with the operating system in GUI mode causes it to forcefully evict the process before it can complete its computation. The second reason is that the kernel needs to be made as simple as possible to take advantage of the SIMT model. Embedding the entire Euler calculation broke this model since the loop within the hcf function will allow some threads to complete their execution before the other.

The identification of the latter problem led to the development of the final version where the kernel was called iteratively to perform the hcf calculations for each input value. This is not the most optimal solution, but the best which could be derived in the face of time and other technical constraints.

4.2 C+MPI

As for the C+MPI, during the parallelisation process, it had been encountered (blog posts on 23rd February) that sending data in item-by-item fashion actually reduces the performance, since an extra overhead is added to the total computation by the amount of communication between processes involved in the computation. Obvious first change (optimisation) had been made on the same day, which can be traced by going through the blog, to partition the whole set of data into buckets (arrays) of corresponding data and these buckets communicated between processes. This improved the performance, but later it was found that in case when the data cannot be partitioned nicely, the rest element(s) will be added into a separate array, thus introducing additional overhead for one extra communication.

Therefore, it had been decided to expand every array by one element in case if the rest is present and only then fill the arrays with data and pass it to a process that is waiting to compute the $\phi$ for any given array. The resulting implementation is presented in 'Implementation 1' and the quantification data is given for the reference.

On 26th of February, as per blog post, potential problem was found: "Sometime during a computation, one processor will be processing the most amount of numbers, while some other process will be processing the least amount of numbers". This results in uneven distribution, that obviously will make one process run longer and another faster. This had to be solved!

The problem during the development of the data distribution was to make sure that no additional loops will be introduced into the computation, but rather appropriate indexes must be used to make the distribution more or less even. This turned to be
problematic and for a cost of one additional day plus one question on stackoverflow, which got almost no attention from serious programmers, the solution had been developer after many trials and errors (and currently is ugly and long). But it improves the runtime by a bit, which turned to be a pleasant surprise. Memoization technique had been adopted at the very end with a vision to improve performance by much more than with the even distribution. But this approach turned to be a fiasco. Additional work can be put into refactoring this approach to find possible synchronisation issues between processes.

As for the very last, an idea came to mind that maybe memoisation is as slow as it was due to the fact that the balancing had been used, which would mean that \(\phi\) of some of the most common factors (i.e. 2, 3, 5,\ldots) will not be computed first, but be postponed for a while (due to balancing). Therefore one step backwards was taken to get rid of the balancing, but rely purely on the memoization. This did not give any sensible results either.

5 Reflection on Programming Models

Given the different programming model of each technology there are trade offs worthy of consideration. This section will provide a generic evaluation of the SIMT and SPMD programming models.

With SIMT, a problem space where a large number of simple, independent computations need to occur suits this model very well. The operations need to be quite simple as complex logic involving branching and looping can break the synchronization of the threads which results in very poor performance. The programmability of the SIMT model requires a complete understanding of the core operations which can be implemented in parallel to obtain a speedup. For this task it was observed that parallelism done from too high a level does not scale well and needs to be as close to actual computation which can benefit from parallelism as much as possible.

The final aspect to be considered with this model is that of potability, with the OpenCL implementation of this model, care needs to be taken in identifying the hardware architecture and the configuration of how many threads should be allowed for local work group sizes. There is no exact guideline in determining this which therefore means some experimentation is required. Migrating to other architectures can have negative or positive impacts on performance.

With SPMD, the parallelism can be achieved on any machine with dynamic number of cores or an a cluster with many processes willing to accept an incoming task. It can be said that the whole process happens in an ad-hoc manner - as long as the program is developer, node endpoints can be added or removed from the configuration file. In my opinion, this is a very useful tool when dealing with large computations and at some point the host decides to connect more nodes to make computation even faster. This can be achieved by modifying the configuration file and attaching the specified nodes to the main process. The set up is robust - nodes can die and the tolerance can be added to cater for such behavior. In addition, since only an additional specification has to be added to an existing language specification to allow the communication between the nodes, SPMD approach becomes a natural extension to an existing language to enable distributed communication.
between processes. The general pitfall, which a user should be aware of is a shared memory model, which is used by for example C+MPI implementation. This can cause many issues when distributing all data over all available node, such as deadlocks, memory locking, starvation, racing conditions, etc. Therefore, the user (programmer) has to be aware of all the existing pitfalls and plan for which pattern he/she will employ to solve a specific problem.
References

   URL: https://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf

StackOverflow (2015), ‘Distribute elements of equivalent arrays to achieve balanced sums’.
Parallel paradigm and performance tuning SIMP

For OpenCL the parallel paradigm is that of single instruction multiple threads; the solution to the totient function was implemented to be specific to NVIDIA GeForce GT 520 architecture and is therefore tuned to use the maximum work group size supported by this platform in an effort to tune performance. Through experimentation, it was found that this work group size produced the fastest results for all datasets. The model for the SIMT paradigm is depicted in the diagram below.

Figure 1: SIMP model
Parallel paradigm and performance tuning SPMD

For C+MPI the parallelism is achieved by using a single program on multiple nodes

Figure 1: SPMD model
## C+MPI Runtimes

<table>
<thead>
<tr>
<th>Implementation</th>
<th># proc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

### Implementation 1

<table>
<thead>
<tr>
<th>[1-1500 0]</th>
<th>6074.61028</th>
<th>6670.394961</th>
<th>6095.037728</th>
<th>5067.305022</th>
<th>6149.305649</th>
<th>6170.726469</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1-3000 0]</td>
<td>28361.54509</td>
<td>26190.74533</td>
<td>25066.50106</td>
<td>26186.917</td>
<td>28211.38803</td>
<td>26270.12728</td>
</tr>
<tr>
<td>[1-1000 00]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Implementation 2

<table>
<thead>
<tr>
<th>[1-1500 0]</th>
<th>6111.888889</th>
<th>6126.777778</th>
<th>6018.833333</th>
<th>6052</th>
<th>6035</th>
<th>6119.444445</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1-3000 0]</td>
<td>26248.11111</td>
<td>26255.33333</td>
<td>25886.66667</td>
<td>26351.22222</td>
<td>25956.1111</td>
<td>25764.22222</td>
</tr>
<tr>
<td>[1-1000 00]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Implementation 3

<table>
<thead>
<tr>
<th>[1-1500 0]</th>
<th>11651.16667</th>
<th>12215.33333</th>
<th>12368</th>
<th>12185</th>
<th>12368</th>
<th>12317</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1-3000 0]</td>
<td>49241.66667</td>
<td>51840.66667</td>
<td>51748.66667</td>
<td>52334</td>
<td>52362.33333</td>
<td>52622.33333</td>
</tr>
<tr>
<td>[1-1000 00]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Implementation 4

<table>
<thead>
<tr>
<th>[1-1500 0]</th>
<th>9387.410567</th>
<th>11770.64042</th>
<th>12137.23865</th>
<th>12338.58759</th>
<th>12309.44629</th>
<th>12304.9388</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1-3000 0]</td>
<td>38157.76347</td>
<td>48098.70437</td>
<td>50005.74428</td>
<td>51198.46029</td>
<td>51506.39666</td>
<td>52515.26462</td>
</tr>
<tr>
<td>[1-1000 00]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Implementation 5

<table>
<thead>
<tr>
<th>[1-1500 0]</th>
<th>64324.3342</th>
<th>64354.3903</th>
<th>671324.8014</th>
<th>664154.3714</th>
<th>585520.0776</th>
<th>654411.3654</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1-3000 0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1-1000 00]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OpenCL

```c
long hcf(long x, long y)
{
    long t;
    while (y != 0)
    {
        t = x % y;
        x = y;
        y = t;
    }
    return x;
}

_kernellaotient(const unsigned int n, _global int *output)
{
    int i = get_global_id(0);
    output[i] = hcf(n, i+1);
}
```

Implementation 1: Max M Baird - kernel3.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 128
void ensureFileExists(FILE *fp, char *filename)
{
    if (fp == NULL)
    {
        fprintf(stderr, "%s not found! exiting...\n", filename);
        exit(-1);
    }
}

int countLines(char *filename)
{
    char ch;
    int lines = 0;

    FILE *fp = fopen(filename, "r");
    ensureFileExists(fp, filename);
    while (!feof(fp))
    {
        ch = fgetc(fp);
        if (ch == '\n')
        {
            lines++;  
        }
    }
    fclose(fp);
    return lines;
}

char *loader(char *filename)
{
```
```c
FILE *fp = NULL;
char line[SIZE];
char *str = NULL;
int count = 0;
count = countLines(filename);
str = (char *)malloc((count * SIZE) * sizeof(char));
if (str == NULL)
{
    fprintf(stderr, "Failed to allocate memory for file \"%s\"!", filename);
    exit(-1);
}
fp = fopen(filename, "r");
ensureFileExists(fp, filename);
while (fgets(line, SIZE, fp) != NULL)
{
    strcat(str, line);
}
fclose(fp);
return str;
```

Implementation 2: Max M Baird - loader.c

```c
#ifdef LOADER_H_
#define LOADER_H_

char *loader(char *filename);

#endif /* LOADER_H_ */
```

Implementation 3: Max M Baird - loader.h

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <time.h>
#include "CL/c1.h"
#include "simple.h"

typedef struct {
    clarg_type arg_t;
    cl_mem dev_buf;
    float *host_buf;
    int *int_host_buf;
    int num_elems;
    int val;
} kernel_arg;

#define MAX_ARG 10

#define die(msg, ...) do {
```
(void) printf (stderr, msg, ##_VA_ARGS__);
)
while (0)

/* global setup */
static cl_platform_id cpPlatform; /* openCL platform. */
static cl_device_id device_id; /* Compute device id. */
static cl_context context; /* Compute context. */
static cl_command_queue commands; /* Compute command queue. */
static cl_program program; /* Compute program. */
static int num_kernel_args;
static kernel_arg kernel_args[MAX_ARG];

cl_int initDevice ( int devType)
{
  cl_int err = CL_SUCCESS;
  /* Connect to a compute device. */
  err = clGetPlatformIDs (1, &cpPlatform, NULL);
  if (CL_SUCCESS != err) {
    die ("Error: Failed to find a platform!");
  } else {
    /* Get a device of the appropriate type. */
    err = clGetDeviceIDs (cpPlatform, devType, 1, &device_id, NULL);
    if (CL_SUCCESS != err) {
      die ("Error: Failed to create a device group!");
    } else {
      /* Create a compute context. */
      context = clCreateContext (0, 1, &device_id, NULL, NULL, &err);
      if ( !context || err != CL_SUCCESS) {
        die ("Error: Failed to create a compute context!");
      } else {
        /* Create a command commands. */
        commands = clCreateCommandQueue (context, device_id, 0, &err);
        if ( !commands || err != CL_SUCCESS) {
          die ("Error: Failed to create a command commands!");
        }
      }
    }
  }
  return err;
}

cl_int initCPU ()
{
  return initDevice ( CL_DEVICE_TYPE_CPU);
}

cl_int initGPU ()
{
  return initDevice ( CL_DEVICE_TYPE_GPU);
cl_kernel setupKernel(const char *kernel_source, char *kernel_name, int num_args, ...)
{
    cl_kernel kernel = NULL;
    cl_int err = CL_SUCCESS;
    va_list ap;
    int i;

    /* Create the compute program from the source buffer. */
    program = clCreateProgramWithSource(context, 1,
        (const char **) &kernel_source,
        NULL, &err);
    if (!program || err != CL_SUCCESS) {
        die("Error: Failed to create compute program!");
    }

    /* Build the program executable. */
    err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    if (err != CL_SUCCESS) {
        size_t len;
        char buffer[2048];
        clGetProgramBuildInfo (program, device_id, CL_PROGRAM_BUILD_LOG,
            sizeof(buffer), buffer, &len);
        die("Error: Failed to build program executable!\n%s", buffer);
    }

    /* Create the compute kernel in the program. */
    kernel = clCreateKernel(program, kernel_name, &err);
    if (!kernel || err != CL_SUCCESS) {
        die("Error: Failed to create compute kernel!");
        kernel = NULL;
    } else {
        num_kernel_args = num_args;
        va_start(ap, num_args);
        for (i = 0; i < num_args && (kernel != NULL); i++) {
            kernel_args[i].arg_type = va_arg(ap, clarg_type);
            switch(kernel_args[i].arg_type) {
                case FloatArr:
                    kernel_args[i].num elems = va_arg(ap, int);
                    kernel_args[i].host buf = va_arg(ap, float *);
                    /* Create the device memory vector */
                    kernel_args[i].dev buf = clCreateBuffer(context,
                        sizeof(float) *
                    kernel_args[i].num elems, NULL, NULL);
                    if (!kernel_args[i].dev buf) {
                        die("Error: Failed to allocate device memory for arg %d!",
                            i + 1);
                        kernel = NULL;
                    } else {
                    }
        }
    }
}
err = clEnqueueWriteBuffer ( commands, kernel_args[i].dev buf, CL_TRUE, 0, sizeof (float) * kernel_args[i].num elems, host_buf, 0, NULL, NULL);
if( CL_SUCCESS != err) {
    die ("Error: Failed to write to source array for arg %d!", i+1);
    kernel = NULL;
}
err = clSetKernelArg ( kernel, i, sizeof (cl_mem), & kernel_args[i].dev buf);
if( CL_SUCCESS != err) {
    die ("Error: Failed to set kernel arg %d!", i);
    kernel = NULL;
}
break;
case IntArr:
    kernel_args[i].num elems = va_arg(ap, int);
    kernel_args[i].int_host_buf = va_arg(ap, int*);
    kernel_args[i].dev buf = clCreateBuffer ( context, CL_MEM_READ_WRITE, sizeof (int) * kernel_args[i].num elems, NULL, NULL);
    if (!kernel_args[i].dev buf) {
        die ("Error: Failed to allocate device memory for arg %d!", i+1);
        kernel = NULL;
    } else {
        err = clEnqueueWriteBuffer ( commands, kernel_args[i].dev buf, CL_TRUE, 0, sizeof (int) * kernel_args[i].num elems, kernel_args[i].int host_buf, 0, NULL, NULL);
        if( CL_SUCCESS != err) {
            die ("Error: Failed to write to source array for arg %d!", i+1);
            kernel = NULL;
        }
        err = clSetKernelArg ( kernel, i, sizeof (cl_mem), & kernel_args[i].dev buf);
        if( CL_SUCCESS != err) {
            die ("Error: Failed to set kernel arg %d!", i);
            kernel = NULL;
        }
    }
    break;
case IntConst:
    kernel_args[i].val = va_arg(ap, unsigned int);
    err = clSetKernelArg ( kernel, i, sizeof (unsigned int), & kernel_args[i].val);
    if( CL_SUCCESS != err) {
        die ("Error: Failed to set kernel arg %d!", i);
        kernel = NULL;
    }
}
break;
default:
die("Error: illegal argument tag for executeKernel!");
kernels = NULL;
}
va_end(ap);
return kernel;

cl_int runKernel( cl_kernel kernel, int dim, size_t *global, size_t *local)
{
    cl_int err;

clock_gettime( CLOCK_REALTIME, &start);
if (CL_SUCCESS != clEnqueueNDRangeKernel (commands, kernel,
    dim, NULL, global, local, 0, NULL,
    NULL))
die("Error: Failed to execute kernel!");
/* Wait for all commands to complete. */
err = clFinish (commands);
clock_gettime( CLOCK_REALTIME, &stop);
for( int i=0; i< num_kernel_args; i++) {
    if (kernel_args[i].arg.t == FloatArr) {
        err = clEnqueueReadBuffer (commands, kernel_args[i].dev_buf,
            CL_TRUE, 0, sizeof (float) *
            kernel_args[i].num elems,
            kernel_args[i].host_buf, 0, NULL, NULL) ;
        if ( err != CL_SUCCESS)
            switch (err) {
            case CL_INVALID_COMMAND_QUEUE:
                printf("Invalid command queue\n");break;
            case CL_INVALID_CONTEXT:
                printf("Invalid context\n");break;
            case CL_INVALID_MEM_OBJECT:
                printf("Invalid memory object\n");break;
            case CL_INVALID_VALUE:
                printf("Invalid value\n");break;
            case CL_INVALID_EVENT_WAIT_LIST:
                printf("Invalid event wait list\n");break;
            case CL_MEM_OBJECT_ALLOCATION_FAILURE:
                printf("Memory object allocation failure\n");break;
            case CL_OUT_OF_HOST_MEMORY:
                printf("Out of host memory\n");break;
            case CL_OUT_OF_RESOURCES:
                printf("Out of resources\n");break;
            case CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST:
                printf("Status error\n");break;
        }
case CL_MISALIGNED_SUB_BUFFER_OFFSET:
    printf("Misaligned sub buffer\n"); break;
default:
    printf("No error codes found: %d\n", err);
} die( "Error: Failed to transfer back arg %d!\n", i);
}
if( kernel_args[i].arg.t == IntArr) {
    err = clEnqueueReadBuffer (commands, kernel_args[i].dev_buf,
        CL_TRUE, 0, sizeof (int) * kernel_args[i].num elems,
        kernel_args[i].int_host_buf, 0, NULL,
        NULL);
    if( err != CL_SUCCESS){
        switch (err)
        {
        case CL_INVALID_COMMAND_QUEUE:
            printf("Invalid command queue\n"); break;
        case CL_INVALID_CONTEXT:
            printf("Invalid context\n"); break;
        case CL_INVALID_MEM_OBJECT:
            printf("Invalid memory object\n"); break;
        case CL_INVALID_VALUE:
            printf("Invalid value\n"); break;
        case CL_INVALID_EVENT_WAIT_LIST:
            printf("Invalid event wait list\n"); break;
        case CL_MEM_OBJECT_ALLOCATION_FAILURE:
            printf("Memory object allocation failure\n"); break;
        case CL_OUT_OF_HOST_MEMORY:
            printf("Out of host memory\n"); break;
        case CL_OUT_OF_RESOURCES:
            printf("Out of resources\n"); break;
        case CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST:
            printf("Status error\n"); break;
        case CL_MISALIGNED_SUB_BUFFER_OFFSET:
            printf("Misaligned sub buffer\n"); break;
        default:
            printf("No error codes found: %d\n", err);
        } die( "Error: Failed to transfer back arg %d!\n", i);
    }
}
return err;

void printKernelTime()
{
    double elapsed = (stop.tv_sec - start.tv_sec)*1000.0
        + (stop.tv_nsec - start.tv_nsec)/1000000.0;
    printf("time spent on kernel: %f msec\n", elapsed);
}
cl_int freeDevice()
{ cl_int err;
    for( int i=0; i< num_kernel_args; i++) {
        if( kernel_args[i].arg_t == FloatArr)
            err = clReleaseMemObject (kernel_args[i].dev_buf);
    }
    err = clReleaseProgram (program);
    err = clReleaseCommandQueue (commands);
    err = clReleaseContext (context);
    return err;
}

Implementation 4: Max M Baird - simple.c

#define SIMPLE_H
#include <cl/cl.h>

#ifndef SIMPLE_H
#define SIMPLE_H

/*****
* initGPU : sets up the openCL environment for using a GPU.
* Note that the system may have more than one GPU in which
* the one that has been pre-configured will be chosen.
* If anything goes wrong in the course, error messages
* will be printed to stderr and the last error encountered will be
* returned.
*****

extern cl_int initGPU ();

/*****
* initCPU : sets up the openCL environment for using the host
* machine.
* If anything goes wrong in the course, error messages
* will be printed to stderr and the last error encountered will be
* returned.
* Note that this may go wrong as not all openCL implementations
* support this!
*****

extern cl_int initCPU ();

*/
#endif
setupKernel : this routine prepares a kernel for execution. It takes the following arguments:
- the kernel source as a string
- the name of the kernel function as string
- the number of arguments (must match those specified in the kernel source!)
- followed by the actual arguments. Each argument to the kernel results in two or three arguments to this function, depending on whether these are pointers to float arrays or integer values:

legal argument sets are:
- FloatArr::clarg_type, num elems::int, pointer::float *
- IntConst::clarg_type, number::int

If anything goes wrong in the course, error messages will be printed to stderr. The pointer to the fully prepared kernel will be returned.

Note that this function actually performs quite a few openCL tasks. It compiles the source, it allocates memory on the device and it copies over all float arrays. If a more sophisticated behaviour is needed you may have to fall back to using openCL directly.

```
typedef enum {
  FloatArr,
  IntArr,
  IntConst
} clarg_type;

extern cl_kernel setupKernel( const char *kernel_source, char *kernel_name, int num_args, ...);
```

runKernel : this routine executes the kernel given as first argument.
The thread-space is defined through the next two arguments:

- `<dim>` identifies the dimensionality of the thread-space and
- `<globals>` is a vector of length `<dim>` that gives the upper bounds for all axes. The argument `<local>` specifies the size of the individual warps which need to have the same dimensionality as the overall range.

If anything goes wrong in the course, error messages will be printed to stderr and the last error encountered will be returned.

Note that this function not only executes the kernel with the given range and warp-size, it also copies back *all* arguments from the kernel after the kernel’s completion. If a more sophisticated behaviour is needed you may have to fall back to using openCL directly.

---

 extern cl_int runKernel( cl_kernel kernel, int dim, size_t *global, size_t *local);

/*
 * printKernelTime : we internally measure the wallclock time that elapses during the kernel execution on the device. This routine prints the findings to stdout. Note that the measurement does not include any data transfer times for arguments or results! Note also, that the only function that influences the time values is runKernel. It does not matter how much time elapses between the last call to runKernel and the call to printKernelTime!
 *
extern void printKernelTime();
*/
freeDevice : this routine releases all acquired resources.
If anything goes wrong in the course, error messages
will be printed to stderr and the last error encountered will
be returned.

extern cl_int freeDevice();

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

/** starting point */
define DS_LOWER 1
#define DS1_UPPER 15000
#define DS2_UPPER 30000
#define DS3_UPPER 100000
#define DS1 1
#define DS2 2
#define DS3 3

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);
}

long euler(long n, int *output)
{
    long length, i;
    length = 0;

    for (i = 1; i <= n; i++)
        if (output[i-1] == 1)
            length++;
int main(int argc, char *argv[])
{
    KernelSource = loader("kernel3.c");
    cl_int err;
    cl_kernel kernel = NULL;
    size_t global[1];
    size_t local[1] = {1000.0}; /* size of work group */
    int opt = 0;
    int lower_num = 0;
    int upper_num = 0;
    if (argc == 2)
    {
        opt = atoi(argv[1]);
    }
    lower_num = DS_LOWER;
    switch (opt)
    {
        case DS1:
            upper_num = DS1_UPPER + 1;
            break;
        case DS2:
            upper_num = DS2_UPPER + 1;
            break;
        case DS3:
            upper_num = DS3_UPPER + 1;
            break;
        default:
            fprintf(stderr, "Invalid argument, valid arguments: 1, 2, 3\n");
            exit(-1);
    }
    if (lower_num > upper_num)
    {
        fprintf(stderr, "Lower number cannot be greater than upper number !\n");
        exit(-1);
    }
    //int step = 10;
    global[0] = (upper_num - 1); /* step */
    printf("global: %d\n", (int)global[0]);
    int *output = (int *)malloc(global[0] * sizeof(int));
    if (output == NULL)
    {
        }
printf(stderr, "Failed to allocate %d bytes\n", (int)local[0]);
exit(-1);
}

int i;
for(i = 0; i < (int)global[0]; i++)
{
    output[i] = 0;
}

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
err = initGPU();
long int total = 0;

if (err == CL_SUCCESS)
{
    kernel = setupKernel(KernelSource, "totient", 2, IntConst, 0,
                        IntArr, global[0], output);
    for(i = 1; i <= (int)global[0]; i++)
    {
        err = clSetKernelArg(kernel, 0, sizeof(int), &i);
        runKernel(kernel, 1, global, local);
        total += euler(i, output);
    }
}

printf("Total: %ld\n", total);

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &stop);
printKernelTime();
printTimeElapsed("CPU time spent");
err = clReleaseKernel(kernel);
err = freeDevice();
free(KernelSource);
return 0;

Implementation 6: Max M Baird - totient.c
// TotientRange.c – Sequential Euler Totient Function (C Version)
// compile: gcc -Wall -O -o TotientRange TotientRange.c
// run: ./TotientRange lower_num upper_num

// Greg Michaelson 14/10/2003
// Patrick Maier 29/01/2010 [enforced ANSI C compliance]

// This program calculates the sum of the totients between a lower
// and an upper limit using C longs. It is based on earlier work by:
// Phil Trinder, Nathan Charles, Hans-Wolfgang Loidl and Colin
// Runciman

/*
compile: mpicc -Wall -O -o ParallelTotientRangeExpandArraysForTheRest
ParallelTotientRangeExpandArraysForTheRest.c
run: mpirun -np 4 ParallelTotientRangeExpandArraysForTheRest
*/

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

struct timespec start, stop;

// hcf x 0 = x
// hcf x y = hcf y (rem x y)

long hcf(long x, long y)
{
    long t;
    while (y != 0) {
        t = x % y;
        x = y;
        y = t;
    }
    return x;
}

// relprime x y = hcf x y == 1

int relprime(long x, long y)
{
    return hcf(x, y) == 1;
}

// euler n = length (filter (relprime n) [1 .. n-1])

void euler(int pid, int payload_size, int rest)
{
    if (rest > 0) payload_size = payload_size + 1;
    while (1) {
long sum = 0;
long payload[ payload_size ];
MPI_Status status;
MPI_Recv(&payload, payload_size, MPI_LONG, 0, 0,
MPI_COMM_WORLD, &status);

int j;
for (j = 0; j < payload_size; j++){
    long length, i;
    length = 0;
    for (i = 1; i <= *(payload+j); i++)
        if (relprime(*(payload+j), i))
            length++;
    sum += length;
}
MPI_Send(&sum, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD);
}

long sumTotient( long lower, long upper, int processes, int rest )
{
    long sum, result, i, j;
    MPI_Status status;
    int payload_size;
    if(rest == 0)
        payload_size = (upper-lower)/(processes-1);
    else
        payload_size = (upper-lower)/(processes-1) + 1;
    long payload[payload_size];
    sum = 0;
    for (i = lower; i <= upper; i=i+payload_size){
        for(j = i; j<(i+payload_size); j++){
            if(j < upper){
                payload[(j-i)] = j;
            } else{
                payload[(j-i)] = '0';
            }
        }
    }
    MPI_Send(&payload /mssg*/ , payload_size /items*/ ,
MPI_LONG, ((i)/payload_size)% (processes-1)+1 /dst*/ , 0 /tag*/ ,
MPI_COMM_WORLD);
    MPI_Recv(&result /mssg*/ , 1 /items*/ , MPI_LONG, ((i)/
payload_size)% (processes-1)+1 /dst*/ , 0 /tag*/ , MPI_COMM_WORLD
, &status);
    sum = sum + result;
}
return sum;
}

int main( int argc, char ** argv )
{
    double runs_sum = 0;
long lower, upper;

int p, id;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &id);

if (argc != 3) {
  printf("not 2 arguments\n");
  return 1;
}
scanf(argv[1], "%ld", &lower);
scanf(argv[2], "%ld", &upper);

int rest = 0;
rest = ((upper - lower) % ((upper - lower) / (p - 1)));

if (id == 0) {
  int i;
  for (i = 0; i < 1; i++) {
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
    long answer = sumTotient(lower, upper, p, rest);
    if ((answer != 68394316) && (answer != 273571774) && (answer != 3039650754)) {
      printf("%ld wrong answer!!!\n", answer);
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &stop);

    double elapsed = (stop.tv_sec - start.tv_sec) * 1000.0 +
    (double)(stop.tv_nsec - start.tv_nsec) / 1000000.0;

    runs_sum += elapsed;
  }
  printf("%f\n", runs_sum / (double)3);
  MPI_Abort(MPI_COMM_WORLD, 1);
} else {
  euler(id, (upper - lower) / (p - 1), rest);
}
printf("here i am\n");
MPI_Finalize();
return 0;
}

Implementation 7: Boris Mocialov - Implementation1
Boris Mocialov - C+MPI Implementation 2

1 // TotientRange.c - Sequential Euler Totient Function (C Version)
2 // compile: gcc -Wall -O -o TotientRange TotientRange.c
3 // run: ./TotientRange lower_num upper_num
4
5 // Greg Michaelson 14/10/2003
6 // Patrick Maier 29/01/2010 [enforced ANSI C compliance]
7
8 // This program calculates the sum of the totients between a lower
9 // and an upper limit using C longs. It is based on earlier work by:
10 // Phil Trinder, Nathan Charles, Hans-Wolfgang Loidl and Colin
11 // Runciman
12
13 /*
14 compile: mpicc -Wall -O -o ParallelTotientRangeExpandArraysForTheRestWithReflection
15 ParallelTotientRangeExpandArraysForTheRestWithReflection.c
16 run: mpirun -np 4
17 ParallelTotientRangeExpandArraysForTheRestWithReflection
18 mpiexec -n 2 -f hostfile
19 ParallelTotientRangeExpandArraysForTheRestWithReflection 1 15000 */
20
21 #include <stdio.h>
22 #include <mpi.h>
23 #include <time.h>
24
25 struct timespec start, stop;
26
27 // hcf x 0 = x
28 // hcf x y = hcf y (rem x y)
29
30 long hcf(long x, long y)
31 {
32     long t;
33     while (y != 0) {
34         t = x % y;
35         x = y;
36         y = t;
37     }
38     return x;
39 }
40
41 // relprime x y = hcf x y == 1
42
43 int relprime(long x, long y)
44 {
45     return hcf(x, y) == 1;
46 }
47
48 // euler n = length (filter (relprime n) [1 .. n-1])
void euler(int pid, int payload_size)
{
    while (1) {
        long sum = 0;
        long payload[payload_size];
        MPI_Status status;
        MPI_Recv(&payload, payload_size, MPI_LONG, 0, 0, MPI_COMM_WORLD, &status);

        int j;
        for (j = 0; j < payload_size; j++) {
            long length, i;

            length = 0;
            for (i = 1; i <= *(payload+j); i++){
                if (relprime(*(payload+j), i))
                    length++;
            }
            sum += length;
            MPI_Send(&sum, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD);
        }
    }
}

// sumTotient lower upper = sum (map euler [lower, lower+1 .. upper])
// separate every loop iteration to different processes
long sumTotient(long lower, long upper, int payload_size, int processes)
{
    long sum, result, i, j;
    MPI_Status status;

    long payload[payload_size];
    int m = 0;
    int k = (payload_size/2)+(payload_size%2)+1;
    sum = 0;
    int lastAdded1 = 0;
    int lastAdded2 = 0;
    int p = 0;
    int substituted = 0;
    int allowUpdate = 1;
    int s;
    int times = 1;
    int times2 = 0;
    for (i = lower; i <= upper; i=i+payload_size) {
        for (j = i; j<(i+payload_size); j++) {
            if (j <= upper) {
                if (k != 0) {
                    if ((j-i) >= k) {
                        payload[(j-i)] = j - (m);
                        lastAdded2 = payload[(j-i)];
                    } else {
                        // further code
                    }
                } else {
                    // further code
                }
            } else {
                // further code
            }
        }
    }
}

// further code
payload[(j-i)] = upper - (p*payload_size) - (m++) + (p*payload_size);

    if(allowUpdate){
        lastAdded1 = payload[(j-i)];
        allowUpdate = 0;
    }
} else{

    int n;
    int from = lastAdded1 > lastAdded2 ? lastAdded1 :
        from = from + 1;
    int to = lastAdded1 > lastAdded2 ? lastAdded1 :
        lastAdded2;

    int tempFrom = (to-from)/payload_size + ((to-from)
        %payload_size > 0 ? 1 : 0);
    for(s = 0; s < tempFrom; s++){
        int restIndex = -1;

        for(n = from; n < from+payload_size; n++){
            restIndex = restIndex + 1;
            payload[restIndex] = '\0';
            if(n < to && n >= from){
                payload[restIndex] = n;
            } else{
                payload[restIndex] = '\0';
            }
        }

        MPI_Ssend(&payload /*mssg*/, payload_size /*items*/, MPI_LONG, ((i)/payload_size)%(processes-1)+1 /*dst*/, 0 /*tag*/, MPI_COMM_WORLD);
        MPI_Recv(&result /*mssg*/, 1 /*items*/, MPI_LONG, ((i)/payload_size)%(processes-1)+1 /*dst*/, 0 /*tag*/, MPI_COMM_WORLD, &status);
        sum = sum + result;
    }

    return sum;
} else{ payload[(j-i)] = '\0'; }
}

p++;  
k=(k/2)+(k%2)+1;  
allowUpdate = 1;

MPI_Ssend(&payload /*mssg*/, payload_size /*items*/, MPI_LONG, ((i)/payload_size)%(processes-1)+1 /*dst*/, 0 /*tag*/, ...
MPI_COMM_WORLD();
    MPI_Recv(&result /*mssg*/, 1 /*items*/, MPI_LONG, ((i)/payload_size)%((processes-1)+1 /*dst*/, 0 /*tag*/, MPI_COMM_WORLD, &status);
    sum = sum + result;
}
    return sum;

int main(int argc, char ** argv)
{
    long runs_sum = 0;
    long lower, upper;
    int p, id;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    if (argc != 3) {
        printf("not 2 arguments\n");
        return 1;
    }
    scanf(argv[1], "%ld", &lower);
    scanf(argv[2], "%ld", &upper);
    int rest = 0;
    rest = ((upper-lower)%((upper-lower)/(p-1)));
    int payload_size;
    if(rest == 0)
        payload_size = (upper-lower)/(p-1);
    else
        payload_size = (upper-lower)/(p-1) + 1;
    if(id == 0){
        int i;
        for(i=0; i<1; i++){
            clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &start);
            long answer = sumTotient(lower, upper, payload_size, p);
            if((answer != 68394316) && (answer != 273571774) && (answer != 3039650754)) {printf("%ld wrong answer!!!\n", answer);}
            else { printf("answer is right\n"); }
            clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &stop);

            double elapsed = (stop.tv_sec -start.tv_sec)*1000.0 + (double)(stop.tv_nsec -start.tv_nsec)/1000000.0;
            runs_sum += elapsed;
        }
    printf("%f\n", runs_sum/(double)3);
MPI_Abort(MPI_COMM_WORLD, 1);

} else {
  euler(id, payload_size);
}

MPI_Finalize();
return 0;
Boris Mocialov - C+MPI Implementation 3

```c
// TotientRange.c - Sequential Euler Totient Function (C Version)
// compile: gcc -Wall -O -o TotientRange TotientRange.c
// run: ./TotientRange lower_num upper_num

// Greg Michaelson 14/10/2003
// Patrick Maier  29/01/2010 [enforced ANSI C compliance]
// This program calculates the sum of the totients between a lower and an
// upper limit using C longs. It is based on earlier work by:
// Phil Trinder, Nathan Charles, Hans-Wolfgang Loidl and Colin Runciman

/*
compile: mpicc -Wall -O -o ParallelTotientRangeExpandArraysForTheRestWithReflectionWithMemoization
ParallelTotientRangeExpandArraysForTheRestWithReflectionWithMemoization.c
run: mpirun -np 5
ParallelTotientRangeExpandArraysForTheRestWithReflectionWithMemoization
*/

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

struct timespec start, stop;
long *memory = NULL;

// hcf x 0 = x
// hcf x y = hcf y (rem x y)
long hcf(long x, long y)
{
    long t;

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

// relprime x y = hcf x y == 1
int relprime(long x, long y)
{
    return hcf(x, y) == 1;
}
```
/ euler n = length ( filter (relprime n) [1 .. n-1])

void euler(int pid, int payload_size)
{
    while (1) {
        long sum = 0;
        long payload[payload_size];
        MPI_Status status;
        MPI_Recv(&payload, payload_size, MPI_LONG, 0, 0, MPI_COMM_WORLD, &status);

        int j;
        for (j = 0; j < payload_size; j++){
            int memoizing = 0;

            // are any payload[j] factors gcd == 1 & factor[0] * factor[1] == payload[j]
            int o, h;
            for (o=1; o<=payload[j]; o++) {
                if (payload[j]%o==0){
                    for (h=1; h<=payload[j]; h++) {
                        if (payload[j]%h==0){
                            if ((relprime(o, h)) & (o*h == payload[j])){
                                if (memory[o] != NULL && memory[h] != NULL){
                                    // in memory
                                    memoizing = 1;
                                    length = memory[o] * memory[h];
                                }
                            }
                        }
                    }
                }
            }

            if (memoizing == 0){
                length = 0;
                for (i = 1; i <= *(payload+j); i++){
                    if (relprime(*(payload+j), i))
                        length++;
                }

                memory[payload[j]] = length; // remember
            }

            sum += length;
        }

        MPI_Send(&sum, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD);
    }

// sumTotient lower upper = sum (map euler [lower, lower+1 .. upper])
// separate every loop iteration to different processes

long sumTotient(long lower, long upper, int payload_size, int processes) {
    long sum, result, i, j;
    MPI_Status status;

    long payload[ payload_size ];
    int m = 0;
    int k = ( payload_size / 2 ) + ( payload_size % 2 ) + 1;
    sum = 0;
    int lastAdded1 = 0;
    int lastAdded2 = 0;
    int p = 0;
    int substituted = 0;
    int allowUpdate = 1;
    int s;
    int times = 1;
    int times2 = 0;
    for ( i = lower; i <= upper; i = i + payload_size ){
        for ( j = i; j < ( i + payload_size ); j++ ){
            if ( j <= upper ){
                if ( k != 0 ){
                    if ( ( j - i ) >= k ){
                        payload[ ( j - i ) ] = j - ( m );
                        lastAdded2 = payload[ ( j - i ) ];
                    } else{
                        payload[ ( j - i ) ] = upper - ( p * payload_size ) - ( m++ ) + ( p * payload_size );
                        if ( allowUpdate ){
                            lastAdded1 = payload[ ( j - i ) ];
                            allowUpdate = 0;
                        }
                    }
                } else{
                    int n;
                    int from = lastAdded1 > lastAdded2 ? lastAdded2 : lastAdded1;
                    from = from + 1;
                    int to = lastAdded1 > lastAdded2 ? lastAdded1 : lastAdded2;

                    int tempFrom = ( to - from ) / payload_size + ( ( to - from ) % payload_size > 0 ? 1 : 0 ) ;
                    for ( s = 0; s < tempFrom; s++ ){
                        int restIndex = -1;
                        for ( n = from; n < from + payload_size; n++ ){
                            restIndex = restIndex + 1;
                            payload[ restIndex ] = '0';
                            if ( n < to && n >= from ){
                                payload[ restIndex ] = n;
                            } else{
                                payload[ restIndex ] = '0';
                            }
                        }
                    }
                }
            }
        }
    }
}
```c
MPI_Send(&payload, payload_size, MPI_LONG, ((i)/payload_size)%(processes-1)+1, MPI_COMM_WORLD);
MPI_Recv(&result, 1, MPI_LONG, ((i)/payload_size)%(processes-1)+1, MPI_COMM_WORLD, &status);
sum = sum + result;
from = from + payload_size;
}
return sum;
}
}

int main(int argc, char **argv)
{
    long runs_sum = 0;
    long lower, upper;
    int p, id;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);

    if (argc != 3) {
        printf(" not 2 arguments\n");
        return 1;
    }
    scanf(argv[1], "%ld", &lower);
    scanf(argv[2], "%ld", &upper);

    memory = malloc(upper * sizeof(long));
    memset(memory, NULL, upper * sizeof(memory[0]));
    ```
```c
int rest = 0;
rest = ((upper-lower)%((upper-lower)/(p-1)));

int payload_size;

if(rest == 0)
    payload_size = (upper-lower)/(p-1);
else
    payload_size = (upper-lower)/(p-1) + 1;

if(id == 0){
    int i;
    for(i=0;i<1;i++){
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
        long answer = sumTotient(lower, upper, payload_size, p);
        if((answer != 68394316) && (answer != 273571774) && (answer != 3039650754)) {
            printf("%ld wrong answer !!!\n", answer);
            clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &stop);
            double elapsed = (stop.tv_sec - start.tv_sec)*1000.0
                            + (double)(stop.tv_nsec - start.tv_nsec)/1000000.0;
            runs_sum += elapsed;
        }
        printf("%f\n", runs_sum/(double)3);
        MPI_Abort(MPI_COMM_WORLD, 1);
    } else{
        euler(id, payload_size);
    }
    MPI_Finalize();
    return 0;
}
```

Implementation 9: Boris Mocialov - Implementation3
// TotientRange.c - Sequential Euler Totient Function (C Version)
// compile: gcc -Wall -O -o TotientRange TotientRange.c
// run: ./TotientRange lower_num upper_num

// Greg Michaelson 14/10/2003
// Patrick Maier 29/01/2010 [enforced ANSI C compliance]

// This program calculates the sum of the totients between a lower and an
// upper limit using C longs. It is based on earlier work by:
// Phil Trinder, Nathan Charles, Hans-Wolfgang Loidl and Colin
// Runciman

/*
compile: mpicc -Wall -O -o
    ParallelTotientRangeExpandArraysForTheRestWithMemoization
ParallelTotientRangeExpandArraysForTheRestWithMemoization.c
run: mpirun -np 5
    ParallelTotientRangeExpandArraysForTheRestWithMemoization
mpiexec -n 2 -f hostfile
    ParallelTotientRangeExpandArraysForTheRestWithMemoization 1 15000
*/

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

long *memory = NULL;

struct timespec start, stop;

// hcf x 0 = x
// hcf x y = hcf y (rem x y)
long hcf(long x, long y)
{
    long t;

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

// relprime x y = hcf x y == 1
int relprime(long x, long y)
{
    return hcf(x, y) == 1;
}
// euler n = length (filter (relprime n) [1 .. n-1])

void euler(int pid, int payload_size, int rest)
{
    if (rest > 0) payload_size = payload_size + 1;
    while (1) {
        long sum = 0;
        long payload[payload_size];
        MPI_Status status;
        MPI_Recv(&payload, payload_size, MPI_LONG, 0, 0,
                  MPI_COMM_WORLD, &status);

        int j;
        for (j = 0; j < payload_size; j++) {
            long length, i;
            int memoizing = 0;
            int o, h;
            for (o = 1; o <= payload[j]; o++) {
                if (payload[j] % o == 0) {
                    for (h = 1; h <= payload[j]; h++) {
                        if (payload[j] % h == 0) {
                            if ((relprime(o, h)) && (o * h == payload[j])){
                                // payload[j] factors gcd == 1
                                if (memory[o] != NULL && memory[h] != NULL) {
                                    // in memory
                                    // printf("memoizing! %d and %d\n",
                                    // o, h);
                                    memoizing = 1;
                                    length = memory[o] * memory[h];
                                }
                            }
                        }
                    }
                }
            }

            if (memoizing == 0) {
                length = 0;
                for (i = 1; i <= *(payload+j); i++) {
                    if (relprime(*(payload+j), i))
                        length++;
                }
                memory[payload[j]] = length;  // remember
            }
            sum += length;
        }
        MPI_Send(&sum, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD);
    }
}

// sumTotient lower upper = sum (map euler [lower, lower+1 .. upper])
// separate every loop iteration to different processes
long sumTotient(long lower, long upper, int processes, int rest)
{
    long sum, result, i, j;
    MPI_Status status;
    int payload_size;
    if (rest == 0)
        payload_size = (upper - lower) / (processes - 1);
    else
        payload_size = (upper - lower) / (processes - 1) + 1;
    long payload[payload_size];
    sum = 0;
    for (i = lower; i <= upper; i += payload_size)
        for (j = i; j < (i + payload_size); j++)
            if (j <= upper)
                payload[(j - i)] = j;
            else
                payload[(j - i)] = '\0';
    MPI_Send(&payload /∗msg*/ , payload_size /∗items*/ , MPI_LONG,
              ((i) / payload_size)%(processes - 1) + 1 /∗dst*/ , 0 /∗tag*/ ,
              MPI_COMM_WORLD);
    MPI_Recv(&result /∗msg*/ , 1 /∗items*/ , MPI_LONG, ((i) / payload_size)%(processes - 1) + 1 /∗dst*/ , 0 /∗tag*/ ,
              MPI_COMM_WORLD , &status);
    sum = sum + result;
}

int main(int argc, char ** argv)
{
    double runs_sum = 0;
    long lower, upper;
    int p, id;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    if (argc != 3) {
        printf("not 2 arguments\n");
        return 1;
    }
    sscanf(argv[1], "%ld", &lower);
    sscanf(argv[2], "%ld", &upper);
    memory = malloc(upper * sizeof(long));
    memset(memory, NULL, upper * sizeof(memory[0]));
    int rest = 0;
```c
rest = ((upper−lower)%((upper−lower)/(p−1)));
if (id == 0) {
    int i;
    for (i = 0; i < 1; i++) {
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
        long answer = sumTotient(lower, upper, p, rest);
        if ((answer != 68394316) && (answer != 273571774) && (answer != 3039650754)) {
            printf("%ld wrong answer!!!\n", answer);
        }
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &stop);
    }
    double elapsed = (stop.tv_sec - start.tv_sec)*1000.0
                 + (double)(stop.tv_nsec - start.tv_nsec)/1000000.0;
    runs_sum += elapsed;
    printf("%f\n", runs_sum/(double)3);
    MPI_Abort(MPI_COMM_WORLD, 1);
} else {
    euler(id, (upper−lower)/(p−1), rest);
    MPI_Finalize();
    return 0;
}
```

Implementation 10: Boris Mocialov - Implementation4