F20DP Distributed and Parallel Technology

Assessed Coursework 1:

Evaluating Low-level Parallel Programming models (C+MPI, OpenCL)

Duncan Cameron - H00153427

Nicholas Robinson - H00155243

Fraser Brown - H00155918

Introduction

We were given the task of parallelising a euler totient range summation equation, which is as follows:

$$Give: \ \Phi(n) \quad \equiv |\ \{m \mid m \in \{1, \ldots, n\} \wedge m \perp n\}$$
$$Give: \ m \perp n \equiv \quad gcd\, mn = 1$$
$$Compute: \ \sum_{i=0}^{n} \Phi(n)$$

Work Breakdown:
We had to design two methods, one using C+MPI, the other using OpenCL. Since we were a group of three, three implementations were required in a 2:1 split between technologies. The technology with two people working on it had to use different optimisations in their implementation. We all agreed that Fraser and Nicholas would be implementing OpenCL solutions and Duncan would be implementing a solution in C+MPI.

Learning Outcome:
Throughout this coursework we will gain practical experience in two low level parallel technologies. In addition we will gain the ability to critically analyse the technologies and determine which is best for purpose by running various speed tests and tweaking each algorithm. The result will be a more well rounded understanding of both technologies, their limits, and methods of extracting the best performance from each.

Parallel Technologies:
The parallel Technologies at use throughout this coursework will be C+MPI and OpenCL.

C+MPI is a Message Passing Interface (MPI) used in combination with C. This technology allows for the sending, receiving, broadcasting of messages between host and worker threads. It also utilises barrier etc. as a form of synchronisation. While C+MPI is built with CPUs in mind OpenCL is platform independent.

OpenCL is an open source standard for parallel computing across multiple platforms. It can be used with either C or C++, with third party wrappers to utilise in other languages, and can make parallel solutions for either GPGPUs or CPUs. In this coursework we will be utilising its ability to run parallel programs on GPGPUs.

Environment:
The environment we would be running our solutions in is known as Robotarium. This computing cluster consists of a head node and 10 computes nodes. The compute nodes consist of 8 AMD

machines and 2 Intel machines. The following table describes in more detail the hardware available in each series of nodes:

Node Type	**Processor**	**RAM**	**Other Components**
Head Node	AMD Opteron 6320 (8 cores)	64 GB	-
AMD Compute Node	4x AMD Opteron 6376 (64 cores)	512 GB	NVIDIA K20 GPU
AMD Compute Node (Large RAM)	4x AMD Opteron 6376 (64 cores)	1024 GB	NVIDIA K20 GPU, NVIDIA Quadro K6000
Intel Compute Node	2x Intel Xeon E5-2650v2 (16 cores)	128 GB	-

Table 1.1 - Environment Hardware Breakdown

The robotarium utilises a batch system known as SLURM. SLURM uses a series of queues to allocate tasks to nodes for a period of time. On the robotarium there are two queues we could use, a short queue which ran tasks for a maximum of 15 min or a long queue, which ran for 7 days. The benefits of such a system are that our solutions would get the maximum computing power available for the period they were ran, therefore giving us the best possible results.

Sequential Performance Measurements

Execution Times:

Range	**1 - 15000**	**1 - 30000**	**1 - 100000**
Median Average Time (seconds)	*5.1*	*21.75*	*268.15*

Table 2.1 - Sequential Execution Times

Gprof:

Range	**1 - 15000**	**1 - 30000**	**1 - 100000**
% Time in 'hcf' Function	90.6	92.27	92.89

Table 2.2 - Percentage Time in 'hcf' Function

Range	**1 - 15000**	**1 - 30000**	**1 - 100000**
Number of Calls	112,492,500.00	449,985,000.00	704,982,704.00

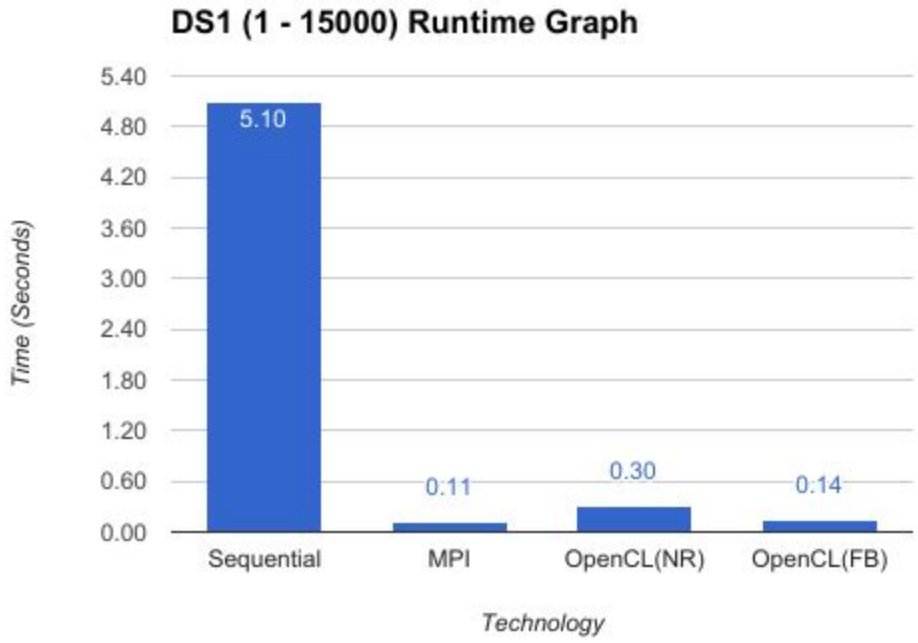
Table 2.3 - Number of Function Calls vs. Range

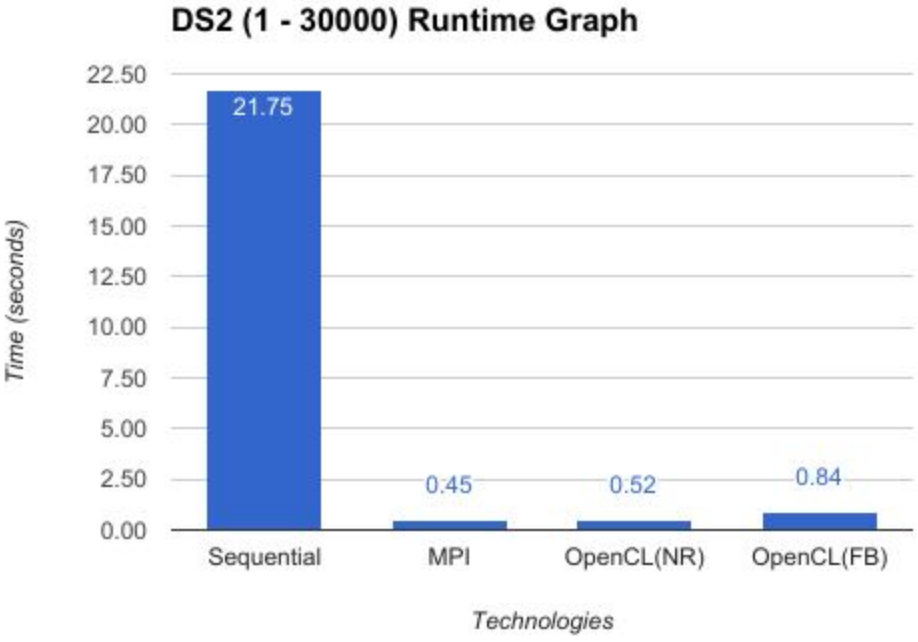
Using the extract from our gprof analysis shown in Table 2.2 we can see a bottleneck in the execution time. Over 90% of the time is spent calculating the Highest Common Factor (HCF). Combine this with the increasing execution times as the range increases and there is a clear need for parallel implementation.

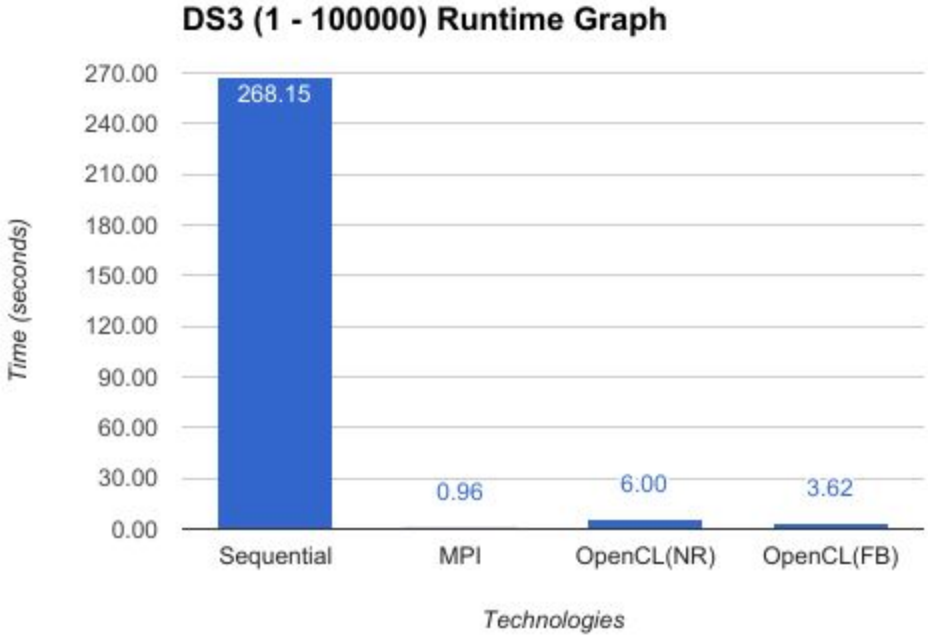
We are hoping to see that by parallelising the equation we will cut down on this bottleneck and overall runtime. In the case of OpenCL each HCF will be calculated in parallel by a single thread. The number of threads will be equal to the max value in the data range.Therefore the time spent waiting for computation time will be reduced and we should see a decrease in time spent in the HCF function.

Another interesting point in the gprof results was the total calls to each function, specifically the calls to the HCF function. Due to the exponential growth of function calls, it can be argued that there is a flaw with the method in which Highest Common Factors are calculated. For instance there is a crossover in the values calculated for the euler(9) and euler(10) therefore a possible improvement on the sequential implementation would be to have a prime caching of previously calculated values. However doing such a method in parallel would cause syncing issues among threads.

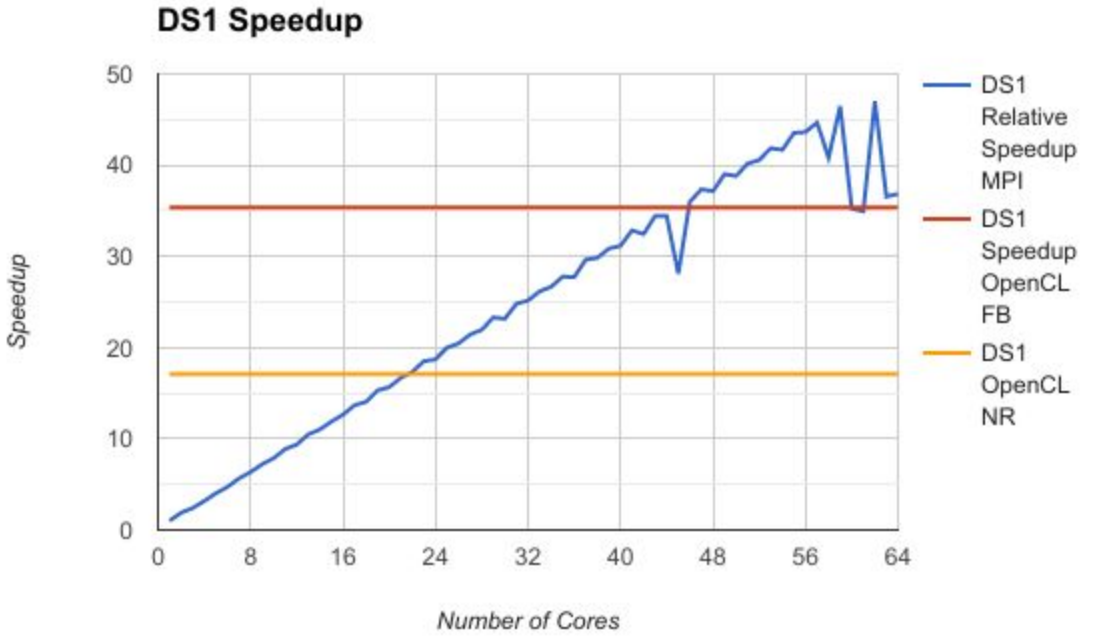
Comparative Parallel Performance Measurements

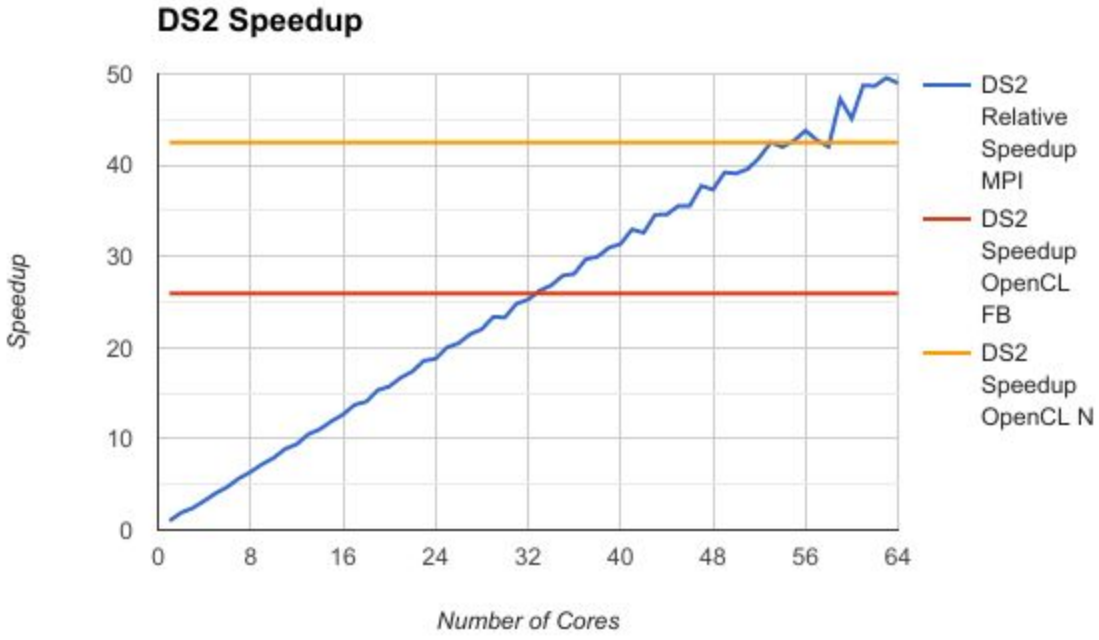


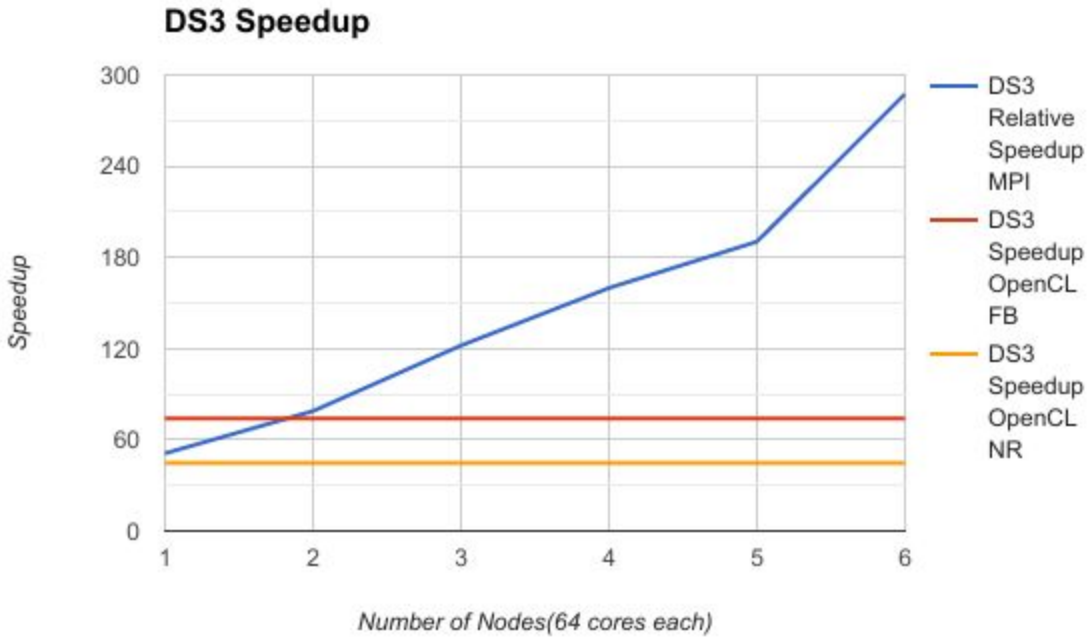



DS3 (1 - 100000) Runtime Graph
270.00
240.00
210.00
180.00
150.00
120.00
90.00
60.00
30.00
0.00
Time (seconds)
268.15
0.96
6.00
3.62
Sequential
MPI
OpenCL(NR)
OpenCL(FB)
Technologies

Speedups


DS1 Speedup
Speedup
50
40
30
20
10
0
0
8
16
24
32
40
48
56
64
Number of Cores
DS1 Relative Speedup MPI
DS1 Speedup OpenCL FB
DS1 OpenCL NR


DS2 Speedup
Speedup
50
40
30
20
10
0
0
8
16
24
32
40
48
56
64
Number of Cores
DS2 Relative Speedup MPI
DS2 Speedup OpenCL FB
DS2 Speedup OpenCL N



NOTE: We were not sure as to how we would gain relative speed up on multiple processors for the OpenCL implementations. Therefore we gathered the speed up value for each run of a given range and plotted it as a constant. We were not sure how to gain access to multiple processors on the GPU to gain a speed up trend that is similar to the MPI line.

The speedups of the MPI program are relative and determined based on running the parallel program on one core. This removes some of the overheads from the final speedup so the parallel performance increase is what we see. While the “optimal” measurement is not met (which would be a linear 1:1 increase with the number of cores) the program still reliably continues to scale upwards as more cores are added.

Runtime Summary (Sequential vs Parallel)

Range	Sequential (seconds)	OpenMPI (seconds)	OpenCL (NR) (seconds)	OpenCL (FB) (seconds)
1-15000	5.1	0.12 (57 Cores)	0.30	0.14
1-30000	21.75	0.45 (64 Cores)	0.52	0.84
1-100000	268.15	0.97 (6 Nodes, 64 Cores per node)	6.00	3.62

Comparative Performance Discussion

It is clear from the above runtime summary table above that MPI is much faster than both the sequential and OpenCL equivalent solutions. However discussing runtime is not a fair representation of each performance due to how closely coupled it is to the hardware. MPI provides much more direct and easier access to hardware, scaling a program on more cores much easier in MPI than in OpenCL.

The difference in runtime performance between the OpenCL implementations could be attributed to the different summing methods used by Nicholas and Fraser. Nicholas used a local workgroup subtotaling method. Which should ideally make the number of values in the final summation smaller. Whereas Fraser performed his summation in a separate thread on all the euler values in the array after they have been computed. Fraser's implementation proved faster, this may be due to the increase in synchronization required in Nicholas' solution and the extra time taken to do local summation. However Duncan used a similar local summing method within his MPI workers and has proven to be the fastest implementation. This could be due to the increased access to hardware discussed above.

As a result we will look at speed up produces compared to the sequential program. This will allow us to see the performance increase based on the improvement to the algorithm. As can be seen from the speedup graphs above the improvements in performance become much closer. However the performance for OpenCL is still dynamic in nature from DS1-3. This could be due to the difference in work group sizes between each run, both OpenCL implementations used widely different workgroup sizes when calculating the factor.

Programming Model Comparison

The two main models, C+MPI and OpenCL, take very different approaches to parallel design. MPI is run using a defined number of cpu cores while OpenCL will dynamically allocates work to as many cores as it can. This allows MPI to be more configurable and be used on more cores across several different networked processors.

The workgroup size used by each OpenCL implementation proved to cause a difference in the results obtained. Both OpenCL implementations did not, at any point, use the same workgroup size. This could factor into why there was a difference in runtime between the same technology, as well as the MPI implementation.

Also the current method of calculating the HCF for each number causes some overlap as there is an equivalent range of prime values generated for `euler(i)` and `euler(i+1)`. Therefore if a method of prime caching was implemented this would improve performance. In OpenCL for instance this could be done in a separate kernel before the euler sum kernel is ran.

Challenges

Overall the challenges faced throughout the coursework were varied, however there appeared to be more during the OpenCL implementation. We believe that this was due to the complicated setup process to gain access to the parallel capabilities of the GPGPU. For instance we had to modify simple.c to support our required data type. Locating the editable places in the setup procedure was difficult. Some other challenges in OpenCL were: thread mapping correctly to our data range, and distributing the spread of work across the the threads in each local work group.

While not many problems occurred with the MPI implementation, the issues that did arise with it was related to getting it to run on the Robotarium cluster. This was due to the fact that there was no guide and MPI was not mentioned in the robotarium howto page.

Software Implementations

The infrastructure and hardware required to gain the performance increase that MPI produced was substantial. Whereas with OpenCL a good increase in performance can be achieved with moderate hardware. Such considerations should be made when analysing the parallel technologies for use in industry and consumer focused software. In this paradigm the extensive hardware of the Robotarium cluster is not available to most users. Therefore taking the slight performance hit obtained by using OpenCL on a GPGPU rather than MPI may be worth it. However from our results above there is no doubt MPI is best for high performance computing.

Program Listings

OpenCL (Fraser Brown): Appendix A

Overview:
The general approach in Fraser's OpenCL implementation below was to have each data item within the given range to execute on a single thread. Therefore for range 1-15000 there would be 15000 threads performing the euler calculation and storing the results in an array. There was some initial experimentation to decide if it was best to sum up the euler array on the GPU or the CPU. It was believed that performing the final summation outside of the GPU would not be completing the task given. There were tests ran with both summing methods, on the GPU and on the CPU. The CPU summation proved to be 0.1 and 0.3 seconds faster at 100000 and 15000 limits respectively.

Challenges:
In order to implement GPU summation some element of syncing threads had to be performed. Therefore barriers were used to sync local threads. As a result the max thread count was increased from `upper` to `upper + 1` this allowed for a check for the final thread after all threads triggered the barrier.

An issue that was uncovered during development was the calculation performed vs the threadID. Early on given an upper limit of 9 for example there would be 0-8 threads generated. This meant that `euler(9)` would not be calculated. To fix this problem the number of threads was increased by 1 finally giving us `upper + 2` overall. This did have a negative effect on the size of the local workgroup. As there are now two more threads the local workgroups have to evenly divide `upper+2`.

Another challenged that occurred was safely calculating the results without obtaining an overflow. As a result editing of Bodo's simple.c occurred and support for Long Arrays (`LongArr`) was added.

It should be noted that to run the appliation the following structure should be taken:
<robotarium cluster commands> fbets-gpusum <lower> <upper> <workgroup size that divides upper+2 evenly>
`E.g. srun --gres=gpu:1 fbets-gpusum 0 15000 26`

Reflection:
In hindsight the summation would be performed by a thread that already has been generated and not an extra one. This would aid in alleviating the limit on local workgroup size.

OpenCL (Nicholas Robinson): Appendix B

Overview:
The implementation is split into 2 kernel calls. The first call is to calculate the euler totient values and to sum the values by local group. The second kernel call sums the remaining values and outputs the global array with the first value as the total result.

Kernel 1 uses one thread for each euler totient function. The threads are divided into local groups. Each local group calculates the value for each thread and then the first thread in the group sums up the values, being stored in a global array. This allows for less data to be passed to the GPU compared to storing every value in a global array to be summed.

Kernel 2 splits the global array in half and sums it using 2 threads. The first thread takes the total by combining the two summed values and a third if the global array is not an even size.

Challenges:
There is a limit on the number of threads in a local group using this program of 31 when using the robotarium. While this is easy to work-around. The capability of the local summing is severely limited.

Using local groups to sum the values causes issues when doing a summation of the full global array. As the local groups cannot see each other, there is no reliable way to do global summation using local groups. A second kernel was created and ran to sum up the array.

Bodo's simple.c file was modified to allow the use of other data types. Unsigned longs are available using a LongArr declaration.
The use of local arrays required more modifications of the file. In OpenCL arrays cannot be declared in the kernel if the length is a variable, as this could change between threads. The variable type LocalLongArr was created to solve this problem.

Reflection:
Doing the summation of the values sequentially, instead of reducing the array proved to give better results. The resources sacrificed to use local summation proved to be not worth it.

When doing summation of the array of local group summations, summing using the CPU proved to be consistently faster than using the GPU. Despite this, I felt to meet the specification of the coursework, the summing of the values would take place in the GPU.

C+MPI (Duncan Cameron): Appendix C

Overview:
Duncan's approach was to attempt to spread the work across all processes of the program as evenly as possible and to reduce message passing as it is often the bottleneck of mpi programs. In order to achieve this, each process would identify its workload based on its id and the total number of processes. This removed the requirements for any message passing to send to the processes at the start and throughout the process.

Each instance would calculate the the euler totient related to its rank value and then step up by the total number of instances. The processing was done in steps as opposed to doing it in chunks as calculating the euler totient for smaller values is quicker than for higher values; if it was done this way, the lower instances would have much less work to do.

Each instance also calculates the total over all of the numbers it finds the euler totient for, reducing the number of messages passed back as well as allowing some of the summations to be done in parallel.

Challenges:

The main challenge with this was getting the program to run on the robotarium cluster. In the end, there were several things that had to be considered: using the -lmpi option when compiling the program, adding the --mpi=pmi2 option when executing the program and ensuring that the correct module was loaded.

Reflection:
The program ran very fast and scaled nicely with the addition of more cores and nodes.

Appendix A

OpenCL (Fraser Brown): “fbets-gpusum.c”

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <time.h>
4.
5.  #include <CL/cl.h>
6.  #include "simple.h"
7.
8.  const char *KernelSource = "\n"
9.      "    long hcf(long x, long y)                    \n"
10.     "    {                                           \n"
11.     "        long t;                                 \n"
12.     "                                                \n"
13.     "        while (y != 0) {                        \n"
14.     "             t = x % y;                         \n"
15.     "             x = y;                             \n"
16.     "            y = t;                              \n"
17.     "        }                                       \n"
18.     "        return x;                               \n"
19.     "    }                                           \n"
20.     "                                                \n"
21.     "    long relprime(long x, long y)               \n"
22.     "    {                                           \n"
23.     "          return hcf(x, y) == 1;                \n"
24.     "    }                                           \n"
25.     "                                                \n"
26.     "    long euler(long n)                          \n"
27.     "    {                                           \n"
28.     "          long length, i;                       \n"
29.     "                                                \n"
30.     "          length = 0;                           \n"
31.     "          for (i = 1; i < n; i++)               \n"
32.     "        if (relprime(n, i))                     \n"
33.     "             length++;                          \n"
34.     "          return length;                        \n"
35.     "    }                                           \n"
36.     "                                                \n"
37.     "  __kernel void sumEuTot(                       \n"
38.     "        __global long *test,                    \n"
39.     "           int lower,                           \n"
40.     "          int upper,                            \n"
41.     "          int gthreadCount)                     \n"
42.     "  {                                             \n"
43.     "          long sum, i;                          \n"
44.     "          sum = 0;                              \n"
45.     "          int id = get_global_id(0);            \n"
```

```
46.     "        if (id != gthreadCount-1){                    \n"
47.     "            test[id] = euler(id);                     \n"
48.     "        }                                             \n"
49.     //syncronyses each local thread, all threads stop at this point
50.     // and flush local memory
51.     "         barrier(CLK_LOCAL_MEM_FENCE);                \n"
52.     "        if (id == gthreadCount-1){                    \n"
53.     "             //test[id] = 0;                          \n"
54.     "            for (int i = 0; i <= gthreadCount-1; i++){ \n"
55.     "                 sum = sum + test[i];                 \n"
56.     "            }                                         \n"
57.     "               test[id] = sum;                        \n"
58.     "        }                                             \n"
59.     "  }                                                   \n"
60.     " \n";
61.
62. struct timespec start, stop;
63.
64. void printTimeElapsed( char *text)
65. {
66.   double elapsed = (stop.tv_sec -start.tv_sec)*1000.0
67.                  + (double)(stop.tv_nsec -start.tv_nsec)/1000000.0;
68.   printf( "%s: %f msec\n", text, elapsed);
69. }
70.
71. int main (int argc, char * argv[])
72. {
73.   cl_int err;
74.   cl_kernel kernel;
75.   size_t global[1];   /* Number of threads */
76.   size_t local[1];    /* Size of Workgroups */
77.
78.   int upper = 0;    /* upper bounds */
79.   int lower = 0;    /* lower bounds */
80.   if( argc == 4) {
81.     lower = atoi(argv[1]);
82.     upper = atoi(argv[2]);
83.     local[0] = atoi(argv[3]);
84.   } else {
85.     lower = 0;
86.     upper = 9;
87.     local[0] = atoi(argv[1]);
88.   }
89.
90.   /* Create data for the run.  */
91.   long *test = NULL;
92.   int gthreadCount = upper + 2;     //thread for highest value is not run unless n+1 threads are
    generated
93.   global[0] = gthreadCount;
94.   test = (long *) malloc (gthreadCount * sizeof (long));
```

```
95.   err = initGPU();
96.
97.   if( err == CL_SUCCESS) {
98.     kernel = setupKernel( KernelSource, "sumEuTot", 4,
99. LongArr, gthreadCount, test,
100.                                   IntConst, lower,
101.                                   IntConst, upper,
102.                                   IntConst, gthreadCount);
103.     clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &start);
104.
105.     runKernel(kernel, 1, global, local);
106.
107.     //print euler toitent sum
108.     printf("\n openCL_GPU_TOTAL: %ld \n\n", test[gthreadCount-1]);
109.
110.     clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &stop);
111.
112.     printKernelTime();
113.     printTimeElapsed( "CPU time spent");
114.
115.     /* Validate our results.  */
116.     err = clReleaseKernel (kernel);
117.     err = freeDevice();
118.     }
119.
120.     return 0;
121.   }
```

OpenCL (Fraser Brown): “simple.c” (Edited Simple.c File to incorporate Long data type):

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <stdarg.h>
4.  #include <time.h>
5.
6.  #include <CL/cl.h>
7.  #include "simple.h"
8.
9.  typedef struct {
10.   clarg_type arg_t;
11.   cl_mem dev_buf;
12.   float *host_buf;
13.   long *host_bufl;
14.   int    num_elems;
15.   int    val;
16. } kernel_arg;
17.
18. #define MAX_ARG 10
19.
20.
21. #define die(msg, ...) do {                    \
22.   (void) fprintf (stderr, msg, ## __VA_ARGS__); \
23.   (void) fprintf (stderr, "\n");               \
24. } while (0)
25.
26. /* global setup */
27.
28. static cl_platform_id cpPlatform;     /* openCL platform.  */
29. static cl_device_id device_id;        /* Compute device id.  */
30. static cl_context context;            /* Compute context.  */
31. static cl_command_queue commands;     /* Compute command queue.  */
32. static cl_program program;            /* Compute program.  */
33. static int num_kernel_args;
34. static kernel_arg kernel_args[MAX_ARG];
35.
36. static struct timespec start, stop;
37.
38.
39. cl_int initDevice ( int devType)
40. {
41.   cl_int err = CL_SUCCESS;
42.
43.   /* Connect to a compute device.  */
44.   err = clGetPlatformIDs (1, &cpPlatform, NULL);
45.   if (CL_SUCCESS != err) {
46.     die ("Error: Failed to find a platform!");
47.   } else {
48.     /* Get a device of the appropriate type.  */
```

```
49.     err = clGetDeviceIDs (cpPlatform, devType, 1, &device_id, NULL);
50.     if (CL_SUCCESS != err) {
51.       die ("Error: Failed to create a device group!");
52.     } else {
53.       /* Create a compute context.  */
54.       context = clCreateContext (0, 1, &device_id, NULL, NULL, &err);
55.       if (!context || err != CL_SUCCESS) {
56.         die ("Error: Failed to create a compute context!");
57.       } else {
58.         /* Create a command commands.  */
59.         commands = clCreateCommandQueue (context, device_id, 0, &err);
60.         if (!commands || err != CL_SUCCESS) {
61.           die ("Error: Failed to create a command commands!");
62.         }
63.       }
64.     }
65.   }
66.
67.  return err;
68. }
69.
70. cl_int initCPU ()
71. {
72.   return initDevice( CL_DEVICE_TYPE_CPU);
73. }
74.
75. cl_int initGPU ()
76. {
77.   return initDevice( CL_DEVICE_TYPE_GPU);
78. }
79.
80. cl_kernel setupKernel( const char *kernel_source, char *kernel_name, int num_args, ...)
81. {
82.   cl_kernel kernel = NULL;
83.   cl_int err = CL_SUCCESS;
84.   va_list ap;
85.   int i;
86.
87.   /* Create the compute program from the source buffer.  */
88.   program = clCreateProgramWithSource (context, 1,
89.                                        (const char **) &kernel_source,
90.                                        NULL, &err);
91.   if (!program || err != CL_SUCCESS) {
92.     die ("Error: Failed to create compute program!");
93.   }
94.
95.   /* Build the program executable.  */
96.   err = clBuildProgram (program, 0, NULL, NULL, NULL, NULL);
97.   if (err != CL_SUCCESS)
98.     {
```

```
99.       size_t len;
100.          char buffer[2048];
101.
102.          clGetProgramBuildInfo (program, device_id, CL_PROGRAM_BUILD_LOG,
103.                                 sizeof (buffer), buffer, &len);
104.          die ("Error: Failed to build program executable!\n%s", buffer);
105.        }
106.
107.      /* Create the compute kernel in the program.  */
108.      kernel = clCreateKernel (program, kernel_name, &err);
109.      if (!kernel || err != CL_SUCCESS) {
110.        die ("Error: Failed to create compute kernel!");
111.        kernel = NULL;
112.      } else {
113.
114.        num_kernel_args = num_args;
115.        va_start(ap, num_args);
116.        for(i=0; (i<num_args) && (kernel != NULL); i++) {
117.          kernel_args[i].arg_t =va_arg(ap, clarg_type);
118.          switch( kernel_args[i].arg_t) {
119.            case LongArr:
120.              kernel_args[i].num_elems = va_arg(ap, int);
121.              kernel_args[i].host_bufl = va_arg(ap, long *);
122.              /* Create the device memory vector  */
123.              kernel_args[i].dev_buf = clCreateBuffer (context, CL_MEM_READ_WRITE,
124.                                                       sizeof (long) * kernel_args[i].num_elems,
    NULL, NULL);
125.
126.              //printf("LongArr dev_buf: %ld \n", kernel_args[i].dev_buf);
127.
128.              if (!kernel_args[i].dev_buf ) {
129.                die ("Error: Failed to allocate device memory for arg %d!", i+1);
130.                kernel = NULL;
131.              } else {
132.                err = clEnqueueWriteBuffer( commands, kernel_args[i].dev_buf, CL_TRUE, 0,
133.                                                      sizeof (long) * kernel_args[i].num_elems,
134.                                                      kernel_args[i].host_bufl, 0, NULL, NULL);
135.                if( CL_SUCCESS != err) {
136.                  die ("Error: Failed to write to source array for arg %d!", i+1);
137.                  kernel = NULL;
138.                }
139.                err = clSetKernelArg (kernel, i, sizeof (cl_mem), &kernel_args[i].dev_buf);
140.                if( CL_SUCCESS != err) {
141.                  die ("Error: Failed to set kernel arg %d!", i);
142.                  kernel = NULL;
143.                }
144.              }
145.              break;
146.            case FloatArr:
147.              kernel_args[i].num_elems = va_arg(ap, int);
```

```
148.              kernel_args[i].host_buf = va_arg(ap, float *);
149.              /* Create the device memory vector  */
150.              kernel_args[i].dev_buf = clCreateBuffer (context, CL_MEM_READ_WRITE,
151.                                                       sizeof (float) * kernel_args[i].num_elems,
    NULL, NULL);
152.              if (!kernel_args[i].dev_buf ) {
153.                die ("Error: Failed to allocate device memory for arg %d!", i+1);
154.                kernel = NULL;
155.              } else {
156.                err = clEnqueueWriteBuffer( commands, kernel_args[i].dev_buf, CL_TRUE, 0,
157.                                                      sizeof (float) * kernel_args[i].num_elems,
158.                                                      kernel_args[i].host_buf, 0, NULL, NULL);
159.                if( CL_SUCCESS != err) {
160.                  die ("Error: Failed to write to source array for arg %d!", i+1);
161.                  kernel = NULL;
162.                }
163.                err = clSetKernelArg (kernel, i, sizeof (cl_mem), &kernel_args[i].dev_buf);
164.                if( CL_SUCCESS != err) {
165.                  die ("Error: Failed to set kernel arg %d!", i);
166.                  kernel = NULL;
167.                }
168.              }
169.              break;
170.            case IntConst:
171.              kernel_args[i].val = va_arg(ap, unsigned int);
172.              err = clSetKernelArg (kernel, i, sizeof (unsigned int), &kernel_args[i].val);
173.              if( CL_SUCCESS != err) {
174.                die ("Error: Failed to set kernel arg %d!", i);
175.                kernel = NULL;
176.              }
177.              break;
178.            default:
179.              die ("Error: illegal argument tag for executeKernel!");
180.              kernel = NULL;
181.          }
182.        }
183.      }
184.      va_end(ap);
185.
186.      return kernel;
187.    }
188.
189.    cl_int runKernel( cl_kernel kernel, int dim, size_t *global, size_t *local)
190.    {
191.      cl_int err;
192.
193.      clock_gettime( CLOCK_REALTIME, &start);
194.      int result = clEnqueueNDRangeKernel (commands, kernel, dim, NULL, global, local, 0, NULL,
    NULL);
195.      printf("RUNKERNEL-RESULT: %d\n", result);
```

```
196.     if (CL_SUCCESS != result) {
197.       printf("clEnqueue error: %d", result);
198.       die ("Error: Failed to execute kernel!");
199.     }
200.
201.     /* Wait for all commands to complete.  */
202.     err = clFinish (commands);
203.     clock_gettime( CLOCK_REALTIME, &stop);
204.
205.     for( int i=0; i< num_kernel_args; i++) {
206.       if( kernel_args[i].arg_t == FloatArr) {
207.         err = clEnqueueReadBuffer (commands, kernel_args[i].dev_buf,
208.                                    CL_TRUE, 0, sizeof (float) * kernel_args[i].num_elems,
209.                                    kernel_args[i].host_buf, 0, NULL, NULL);
210.         if( err != CL_SUCCESS)
211.           die( "FloatArr Error: Failed to transfer back arg %d!", i);
212.       }
213.       if( kernel_args[i].arg_t == LongArr) {
214.         err = clEnqueueReadBuffer (commands, kernel_args[i].dev_buf,
215.                                    CL_TRUE, 0, sizeof (long) * kernel_args[i].num_elems,
216.                                    kernel_args[i].host_bufl, 0, NULL, NULL);
217.         //printf("LongArr kernel_args[i].num_elems %ld \n", kernel_args[i].num_elems);
218.
219.         if( err != CL_SUCCESS){
220.           printf("LongArr kernel_args[i].num_elems %ld \n", (sizeof (long) *
    kernel_args[i].num_elems)/8);
221.           printf("LongArr Transfer GPU -> Host error: %d \n", err);
222.           die( "LongArr Error: Failed to transfer back arg %d!", i);
223.         }
224.       }
225.     }
226.
227.     return err;
228.   }
229.
230.   void printKernelTime()
231.   {
232.     double elapsed = (stop.tv_sec -start.tv_sec)*1000.0
233.                      + (stop.tv_nsec -start.tv_nsec)/1000000.0;
234.     printf( "time spent on kernel: %f msec\n", elapsed);
235.   }
236.
237.   cl_int freeDevice()
238.   {
239.     cl_int err;
240.
241.     for( int i=0; i< num_kernel_args; i++) {
242.       if( kernel_args[i].arg_t == FloatArr)
243.         err = clReleaseMemObject (kernel_args[i].dev_buf);
244.       if( kernel_args[i].arg_t == LongArr)
```

```
245.         err = clReleaseMemObject (kernel_args[i].dev_buf);
246.     }
247.     err = clReleaseProgram (program);
248.     err = clReleaseCommandQueue (commands);
249.     err = clReleaseContext (context);
250.
251.     return err;
252. }
253.
```

Appendix B

OpenCL (Nicholas Robinson): “OpenCLTotientGPU.c”

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <time.h>
4.
5.  #include <CL/cl.h>
6.  #include "simple.h"
7.
8.  const char *KernelSource =                                    "\n"
9.  "long sumArray(unsigned long a[], int numElements)           \n"
10. "{                                                           \n"
11. "    int i;                                                  \n"
12. "    unsigned long sum=0;                                    \n"
13. "    for (i = 0; i < numElements; i++)                       \n"
14. "        sum = sum + a[i];                                   \n"
15. "    return(sum);                                            \n"
16. "}                                                           \n"
17. "long hcf(unsigned long x, unsigned long y) {                \n"
18. "    long t;                                                 \n"
19. "    while (y != 0) {                                        \n"
20. "        t = x % y;                                          \n"
21. "        x = y;                                              \n"
22. "        y = t;                                              \n"
23. "    }                                                       \n"
24. "    return x;                                               \n"
25. "}                                                           \n"
26. "int relprime(unsigned long x, unsigned long y) {            \n"
27. "    return hcf(x, y) == 1;                                  \n"
28. "}                                                           \n"
29. "long euler(unsigned long n) {                               \n"
30. "    unsigned long length, i;                                \n"
31. "    length = 0;                                             \n"
32. "    for (i = 1; i < n; i++)                                 \n"
33. "        length += (relprime(n, i));                         \n"
34. "    return length;                                          \n"
35. "}                                                           \n"
36. "__kernel void totient(                                      \n"
37. "    const unsigned int lower,                               \n"
38. "    __local unsigned long* locRes,                          \n"
39. "    __global unsigned long* results)                        \n"
40. "{                                                           \n"
41. "    int j = (int)get_local_id(0);                           \n"
42. "    int lsize = (int)get_local_size(0);                     \n"
43. "    int groupNum = (int)get_group_id(0);                    \n"
44. "    locRes[j] = euler(lower + (groupNum * lsize) + j);      \n"
45. "    barrier(CLK_LOCAL_MEM_FENCE);                           \n"
```

```
"    if (j==0) {                                          \n"
"        results[groupNum] = sumArray(locRes, lsize);     \n"
"    }                                                    \n"
"}                                                        \n"
"__kernel void sumResults(                                \n"
"    const unsigned int resNum,                           \n"
"    __global unsigned long* results)                     \n"
"{                                                        \n"
"    if (resNum < 4) {                                    \n"
"        results[0] = sumArray(results, resNum);          \n"
"    } else {                                             \n"
"        int j = (int)get_local_id(0);                    \n"
"        int halfSize = resNum / 2;                       \n"
"        int index = j * halfSize;                        \n"
"        for (int i = 1; i < halfSize; i++)               \n"
"            results[index] += results[index + i];        \n"
"        barrier(CLK_LOCAL_MEM_FENCE);                    \n"
"        if (j==0) {                                      \n"
"            results[0] += results[halfSize];             \n"
"            if (halfSize * 2 != resNum)                  \n"
"                results[0] += results[resNum-1];         \n"
"        }                                                \n"
"    }                                                    \n"
"}                                                        \n"
"\n";

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])
long euler(long n) {
    long length, i;
    length = 0;
    for (i = 1; i < n; i++)
```

```
96.         if (relprime(n, i))
97.             length++;
98.     return length;
99. }
100.
101.    // sumTotient lower upper = sum (map euler [lower, lower+1 .. upper])
102.    long sumTotient(long lower, long upper) {
103.        long sum, i;
104.        sum = 0;
105.        for (i = lower; i <= upper; i++) {
106.            long res = euler(i);
107.            sum = sum + res;
108.        }
109.        return sum;
110.    }
111.
112.    // sumArray a numElements = sum(a)
113.    unsigned long sumArray(unsigned long a[], int numElements)
114.    {
115.        int i;
116.        unsigned long sum=0;
117.        for (i=0; i<numElements; i++)
118.        {
119.            sum = sum + a[i];
120.        }
121.        return(sum);
122.    }
123.
124.    void printTimeElapsed( char *text)
125.    {
126.        double elapsed = (stop.tv_sec -start.tv_sec)*1000.0
127.                        + (double)(stop.tv_nsec -start.tv_nsec)/1000000.0;
128.        printf( "%s: %f msec\n", text, elapsed);
129.    }
130.
131.    void timeDirectImplementation( int lower, int upper)
132.    {
133.        clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &start);
134.        long sum = sumTotient(lower, upper);
135.        clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &stop);
136.        printf("Result: %ld\n",sum);
137.        printTimeElapsed( "kernel equivalent on host");
138.    }
139.
140.
141.    int main (int argc, char * argv[])
142.    {
143.        int lower;
144.        int upper;
145.        cl_int err;
```

```
146.        cl_kernel eulerKernel;
147.        cl_kernel sumKernel;
148.        size_t global[1];
149.        size_t local[1];
150.
151.        if (argc < 2) {
152.            lower = 1;
153.            upper = 30;
154.            local[0] = 10;
155.        } else if (argc < 3) {
156.            lower = 1;
157.            upper = atoi(argv[1]);
158.            local[0] = 10;
159.        } else if (argc < 4) {
160.            lower = 1;
161.            upper = atoi(argv[1]);
162.            local[0] = atoi(argv[2]);
163.        } else {
164.            lower = atoi(argv[1]);
165.            upper = atoi(argv[2]);
166.            local[0] = atoi(argv[3]);
167.        }
168.
169.        printf( "work group size: %d\n", (int)local[0]);
170.
171.
172.        /* Create data for the run.    */
173.        int count = (upper - lower) + 1;
174.        global[0] = count;
175.        int resSize = upper/local[0];
176.        unsigned long result = 0;
177.
178.        unsigned long *results = NULL;  /* Results returned from device.         */
179.        unsigned long *locres = NULL;  /* Array for summing local Results       */
180.
181.        results = (unsigned long *) malloc (resSize * sizeof (unsigned long));
182.        locres = (unsigned long *) malloc (local[0] * sizeof (unsigned long));
183.
184.        err = initGPU();
185.
186.        if( err == CL_SUCCESS) {
187.            //Compute Euler Totient Sum and sum local groups
188.            eulerKernel = setupKernel( KernelSource, "totient", 3, IntConst, lower,
189.                                                                   LocalLongArr, local[0], locres,
190.                                                                   LongArr, resSize, results);
191.
192.            clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &start);
193.            runKernel( eulerKernel, 1, global, local);
194.            printKernelTime();
195.
```

```
196.            //Sum local group values and store result in first value
197.            sumKernel = setupKernel( KernelSource, "sumResults", 2, IntConst, resSize,
198.                                                                    LongArr, resSize, results);
199.            global[0] = 2;
200.            runKernel( sumKernel, 1, global, global);
201.            //result = sumArray(results, resSize);
202.            result = results[0];
203.            clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &stop);
204.            printf("Result: %ld\n", result);
205.
206.            printKernelTime();
207.            printTimeElapsed( "CPU time spent");
208.
209.            err = clReleaseKernel (eulerKernel);
210.            err = clReleaseKernel (sumKernel);
211.            err = freeDevice();
212.
213.            //timeDirectImplementation( lower, upper);
214.
215.        }
216.        return 0;
217.    }
```

OpenCL (Nicholas Robinson): “simple.c”

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <stdarg.h>
4.  #include <time.h>
5.
6.  #include <CL/cl.h>
7.  #include "simple.h"
8.
9.  typedef struct {
10.   clarg_type arg_t;
11.   cl_mem dev_buf;
12.   float *host_buf;
13.   unsigned long *host_bufl;
14.   int    num_elems;
15.   int    val;
16. } kernel_arg;
17.
18. #define MAX_ARG 10
19.
20.
21. #define die(msg, ...) do {                    \
22.   (void) fprintf (stderr, msg, ## __VA_ARGS__); \
23.   (void) fprintf (stderr, "\n");               \
24. } while (0)
25.
26. /* global setup */
27.
28. static cl_platform_id cpPlatform;     /* openCL platform.  */
29. static cl_device_id device_id;        /* Compute device id.  */
30. static cl_context context;            /* Compute context.  */
31. static cl_command_queue commands;     /* Compute command queue.  */
32. static cl_program program;            /* Compute program.  */
33. static int num_kernel_args;
34. static kernel_arg kernel_args[MAX_ARG];
35.
36. static struct timespec start, stop;
37.
38.
39. cl_int initDevice ( int devType)
40. {
41.   cl_int err = CL_SUCCESS;
42.
43.   /* Connect to a compute device.  */
44.   err = clGetPlatformIDs (1, &cpPlatform, NULL);
45.   if (CL_SUCCESS != err) {
46.     die ("Error: Failed to find a platform!");
47.   } else {
48.     /* Get a device of the appropriate type.  */
```

```
49.                 err = clGetDeviceIDs (cpPlatform, devType, 1, &device_id, NULL);
50.                 if (CL_SUCCESS != err) {
51.                   die ("Error: Failed to create a device group!");
52.                 } else {
53.                   /* Create a compute context.  */
54.                   context = clCreateContext (0, 1, &device_id, NULL, NULL, &err);
55.                   if (!context || err != CL_SUCCESS) {
56.                     die ("Error: Failed to create a compute context!");
57.                   } else {
58.                     /* Create a command commands.  */
59.                     commands = clCreateCommandQueue (context, device_id, 0, &err);
60.                     if (!commands || err != CL_SUCCESS) {
61.                       die ("Error: Failed to create a command commands!");
62.                     }
63.                   }
64.                 }
65.               }
66.
67.             return err;
68.           }
69.
70.           cl_int initCPU ()
71.           {
72.             return initDevice( CL_DEVICE_TYPE_CPU);
73.           }
74.
75.           cl_int initGPU ()
76.           {
77.             return initDevice( CL_DEVICE_TYPE_GPU);
78.           }
79.
80.           cl_kernel setupKernel( const char *kernel_source, char *kernel_name, int num_args, ...)
81.           {
82.             cl_kernel kernel = NULL;
83.             cl_int err = CL_SUCCESS;
84.             va_list ap;
85.             int i;
86.
87.             /* Create the compute program from the source buffer.  */
88.             program = clCreateProgramWithSource (context, 1,
89.                                                  (const char **) &kernel_source,
90.                                                  NULL, &err);
91.             if (!program || err != CL_SUCCESS) {
92.               die ("Error: Failed to create compute program!");
93.             }
94.
95.             /* Build the program executable.  */
96.             err = clBuildProgram (program, 0, NULL, NULL, NULL, NULL);
97.             if (err != CL_SUCCESS)
98.               {
```

```
99.                   size_t len;
100.                  char buffer[2048];
101.
102.                  clGetProgramBuildInfo (program, device_id, CL_PROGRAM_BUILD_LOG,
103.                                         sizeof (buffer), buffer, &len);
104.                  die ("Error: Failed to build program executable!\n%s", buffer);
105.                }
106.
107.              /* Create the compute kernel in the program.  */
108.              kernel = clCreateKernel (program, kernel_name, &err);
109.              if (!kernel || err != CL_SUCCESS) {
110.                die ("Error: Failed to create compute kernel!");
111.                kernel = NULL;
112.              } else {
113.
114.                num_kernel_args = num_args;
115.                va_start(ap, num_args);
116.                for(i=0; (i<num_args) && (kernel != NULL); i++) {
117.                  kernel_args[i].arg_t =va_arg(ap, clarg_type);
118.                  switch( kernel_args[i].arg_t) {
119.                    case LocalLongArr:
120.                      kernel_args[i].num_elems = va_arg(ap, int);
121.                      kernel_args[i].host_bufl = va_arg(ap, unsigned long *);
122.                      /* Create the device memory vector  */
123.                      kernel_args[i].dev_buf = clCreateBuffer (context, CL_MEM_ALLOC_HOST_PTR,
124.                                                               sizeof (unsigned long) *
              kernel_args[i].num_elems, NULL, NULL);
125.
126.                      //printf("LongArr dev_buf: %ld \n", kernel_args[i].dev_buf);
127.
128.                      if (!kernel_args[i].dev_buf ) {
129.                        die ("LocalLongArr Error: Failed to allocate device memory for arg %d!",
              i+1);
130.                        kernel = NULL;
131.                      } else {
132.                        err = clEnqueueWriteBuffer( commands, kernel_args[i].dev_buf, CL_TRUE, 0,
133.                                                    sizeof (unsigned long) *
              kernel_args[i].num_elems,
134.                                                    kernel_args[i].host_bufl, 0, NULL,
              NULL);
135.                        if( CL_SUCCESS != err) {
136.                          die ("LocalLongArr Error: Failed to write to source array for arg %d!",
              i+1);
137.                          kernel = NULL;
138.                        }
139.                        err = clSetKernelArg (kernel, i, sizeof (cl_mem), NULL);
140.                        if( CL_SUCCESS != err) {
141.                          printf("LocalLongArr Error: clSetKernelArg %d\n", err);
142.                          die ("LocalLongArr Error: Failed to set kernel arg %d!", i);
143.                          kernel = NULL;
```

```
144.                       }
145.                     }
146.                     break;
147.                   case LongArr:
148.                     kernel_args[i].num_elems = va_arg(ap, int);
149.                     kernel_args[i].host_bufl = va_arg(ap, unsigned long *);
150.                     /* Create the device memory vector  */
151.                     kernel_args[i].dev_buf = clCreateBuffer (context, CL_MEM_READ_WRITE,
152.                                                              sizeof (unsigned long) *
            kernel_args[i].num_elems, NULL, NULL);
153.
154.                     //printf("LongArr dev_buf: %ld \n", kernel_args[i].dev_buf);
155.
156.                     if (!kernel_args[i].dev_buf ) {
157.                       die ("LongArr Error: Failed to allocate device memory for arg %d!", i+1);
158.                       kernel = NULL;
159.                     } else {
160.                       err = clEnqueueWriteBuffer( commands, kernel_args[i].dev_buf, CL_TRUE, 0,
161.                                                   sizeof (unsigned long) *
            kernel_args[i].num_elems,
162.                                                   kernel_args[i].host_bufl, 0, NULL,
            NULL);
163.                       if( CL_SUCCESS != err) {
164.                         die ("LongArr Error: Failed to write to source array for arg %d!", i+1);
165.                         kernel = NULL;
166.                       }
167.                       err = clSetKernelArg (kernel, i, sizeof (cl_mem), &kernel_args[i].dev_buf);
168.                       if( CL_SUCCESS != err) {
169.                         die ("LongArr Error: Failed to set kernel arg %d!", i);
170.                         kernel = NULL;
171.                       }
172.                     }
173.                     break;
174.                   case FloatArr:
175.                     kernel_args[i].num_elems = va_arg(ap, int);
176.                     kernel_args[i].host_buf = va_arg(ap, float *);
177.                     /* Create the device memory vector  */
178.                     kernel_args[i].dev_buf = clCreateBuffer (context, CL_MEM_READ_WRITE,
179.                                                              sizeof (float) *
            kernel_args[i].num_elems, NULL, NULL);
180.                     if (!kernel_args[i].dev_buf ) {
181.                       die ("FloatArr Error: Failed to allocate device memory for arg %d!", i+1);
182.                       kernel = NULL;
183.                     } else {
184.                       err = clEnqueueWriteBuffer( commands, kernel_args[i].dev_buf, CL_TRUE, 0,
185.                                                   sizeof (float) *
            kernel_args[i].num_elems,
186.                                                   kernel_args[i].host_buf, 0, NULL,
            NULL);
187.                       if( CL_SUCCESS != err) {
```

```
188.                         die ("FloatArr Error: Failed to write to source array for arg %d!", i+1);
189.                         kernel = NULL;
190.                       }
191.                       err = clSetKernelArg (kernel, i, sizeof (cl_mem), &kernel_args[i].dev_buf);
192.                       if( CL_SUCCESS != err) {
193.                         die ("FloatArr Error: Failed to set kernel arg %d!", i);
194.                         kernel = NULL;
195.                       }
196.                     }
197.                     break;
198.                   case IntConst:
199.                     kernel_args[i].val = va_arg(ap, unsigned int);
200.                     err = clSetKernelArg (kernel, i, sizeof (unsigned int), &kernel_args[i].val);
201.                     if( CL_SUCCESS != err) {
202.                       die ("IntConst Error: Failed to set kernel arg %d!", i);
203.                       kernel = NULL;
204.                     }
205.                     break;
206.                   default:
207.                     die ("IntConst Error: illegal argument tag for executeKernel!");
208.                     kernel = NULL;
209.                 }
210.               }
211.             }
212.             va_end(ap);
213.
214.             return kernel;
215.           }
216.
217.           cl_int runKernel( cl_kernel kernel, int dim, size_t *global, size_t *local)
218.           {
219.             cl_int err;
220.
221.             clock_gettime( CLOCK_REALTIME, &start);
222.             int result = clEnqueueNDRangeKernel (commands, kernel, dim, NULL, global, local, 0,
          NULL, NULL);
223.             printf("RUNKERNEL-RESULT: %d\n", result);
224.             if (CL_SUCCESS != result) {
225.               die ("Error: Failed to execute kernel!");
226.             }
227.
228.             /* Wait for all commands to complete.  */
229.             err = clFinish (commands);
230.             clock_gettime( CLOCK_REALTIME, &stop);
231.
232.             for( int i=0; i< num_kernel_args; i++) {
233.               if( kernel_args[i].arg_t == FloatArr) {
234.                 err = clEnqueueReadBuffer (commands, kernel_args[i].dev_buf,
235.                                            CL_TRUE, 0, sizeof (float) * kernel_args[i].num_elems,
236.                                            kernel_args[i].host_buf, 0, NULL, NULL);
```

```
237.                 if( err != CL_SUCCESS)
238.                   die( "FloatArr Error: Failed to transfer back arg %d!", i);
239.               }
240.               if( kernel_args[i].arg_t == LongArr) {
241.                 err = clEnqueueReadBuffer (commands, kernel_args[i].dev_buf,
242.                                            CL_TRUE, 0, sizeof (unsigned long) *
            kernel_args[i].num_elems,
243.                                            kernel_args[i].host_bufl, 0, NULL, NULL);
244.               //printf("LongArr kernel_args[i].num_elems %ld \n", kernel_args[i].num_elems);
245.
246.               if( err != CL_SUCCESS){
247.                 printf("LongArr kernel_args[i].num_elems %ld \n", (sizeof (unsigned long) *
            kernel_args[i].num_elems)/8);
248.                 printf("LongArr Transfer GPU -> Host error: %d \n", err);
249.                 die( "LongArr Error: Failed to transfer back arg %d!", i);
250.               }
251.               }
252.               if( kernel_args[i].arg_t == LocalLongArr) {
253.                 err = clEnqueueReadBuffer (commands, kernel_args[i].dev_buf,
254.                                            CL_TRUE, 0, sizeof (unsigned long) *
            kernel_args[i].num_elems,
255.                                            kernel_args[i].host_bufl, 0, NULL, NULL);
256.               //printf("LongArr kernel_args[i].num_elems %ld \n", kernel_args[i].num_elems);
257.
258.               if( err != CL_SUCCESS){
259.                 printf("LocalLongArr kernel_args[i].num_elems %ld \n", (sizeof (unsigned long)
            * kernel_args[i].num_elems)/8);
260.                 printf("LocalLongArr Transfer GPU -> Host error: %d \n", err);
261.                 die( "LocalLongArr Error: Failed to transfer back arg %d!", i);
262.               }
263.               }
264.             }
265.
266.             return err;
267.           }
268.
269.           void printKernelTime()
270.           {
271.             double elapsed = (stop.tv_sec -start.tv_sec)*1000.0
272.                              + (stop.tv_nsec -start.tv_nsec)/1000000.0;
273.             printf( "time spent on kernel: %f msec\n", elapsed);
274.           }
275.
276.           cl_int freeDevice()
277.           {
278.             cl_int err;
279.
280.             for( int i=0; i< num_kernel_args; i++) {
281.               if( kernel_args[i].arg_t == FloatArr)
282.                 err = clReleaseMemObject (kernel_args[i].dev_buf);
```

```
283.                 if( kernel_args[i].arg_t == LongArr)
284.                   err = clReleaseMemObject (kernel_args[i].dev_buf);
285.               }
286.               err = clReleaseProgram (program);
287.               err = clReleaseCommandQueue (commands);
288.               err = clReleaseContext (context);
289.               return err;
290.             }
```

Appendix C

C+MPI (Duncan Cameron) - “mpi-totient.c”

```
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>
#include <stdlib.h>
#include <time.h>

void printHostInfo(int numberOfInstances, int id);
void toitentMaster(int numberOfInstances, long n, long lower);
void toitentWorker(int id, int numberOfInstances, long n, long lower);
    long hcf(long a, long h);

    unsigned long findGroupTotal(int id, int numberOfInstances, long n, long lower) ;

    double elapsedTime;
    int printMode; //0: print all normally, 1: ignore hostInfo, 2: print only times

    int main(int argc, char ** argv)
    {
        int numberOfInstances;
        int id;
        long n;
        long lower;
        printMode = 0;

        if(argc < 2) {
            printf("No arguments");
            return 0;
        }
        else if (argc < 3) {
            n = atoi(argv[1]);
            lower = 1; //1 as 0 is always 0... so is 1.  should it start at 2
        }
        else {
            n = atoi(argv[2]);
            lower = atoi(argv[1]);
            if(argc > 3){
                printMode = atoi(argv[3]);
                if(printMode < 0 || printMode > 2){
                    printf("Invalid printMode (must be 0, 1 or 2)");
                    exit(0);
                }
            }
        }
```

```
45.         if(lower < 0 || n < lower)
46.             printf("invalid arguments: 0 <= lower <= n");
47.
48.         MPI_Init(&argc, &argv);                                    // start up "virtual
    machine"
49.         MPI_Comm_size(MPI_COMM_WORLD, &numberOfInstances);         // get size of VM
50.         MPI_Comm_rank(MPI_COMM_WORLD, &id);                        // get own rank in VM
51.
52.         if(printMode==0)
53.             printHostInfo(numberOfInstances, id);                  // will be useful for
    cluster
54.
55.         MPI_Barrier(MPI_COMM_WORLD);                               //sync processes and
    start timer
56.         elapsedTime = - MPI_Wtime();
57.
58.
59.         if(id==0)
60.             toitentMaster(numberOfInstances, n, lower);
61.         else
62.             toitentWorker(id, numberOfInstances, n, lower);
63.
64.         MPI_Finalize();
65.         return 0;
66.     }
67.
68.     void printHostInfo(int numberOfInstances, int id) {
69.         char name[80];
70.         int z = gethostname(name, 80);
71.         if (z==0)
72.             printf("Hello, I am %d of %d (hostname is %s)\n", id, numberOfInstances,
    name);
73.         else
74.             printf("Hello, I am %d of %d (hostname UNKNOWN)\n", id, numberOfInstances);
75.     }
76.
77.     void toitentMaster(int numberOfInstances, long n, long lower) {
78.         unsigned long long total = 0;
79.         MPI_Status status;
80.         total=findGroupTotal(0, numberOfInstances, n, lower);
81.         for(int i=1; i < numberOfInstances; i++) {
82.             unsigned long processTot;
83.             MPI_Recv(&processTot, 1, MPI_UNSIGNED_LONG,i, 0, MPI_COMM_WORLD, &status);
84.             total += processTot;
85.         };
86.
87.         elapsedTime += MPI_Wtime();
88.         if(printMode < 2) {
89.             printf("Elapsed time: %f secs\n", elapsedTime);
90.             printf("Total calculated: %llu\n", total);
```

```
91.         } else {
92.             printf("%f",elapsedTime);
93.         }
94.     };
95.
96.     void toitentWorker(int id, int numberOfInstances, long n, long lower) {
97.         long total = findGroupTotal(id, numberOfInstances, n, lower);
98.         MPI_Send(&total, 1, MPI_UNSIGNED_LONG, 0, 0, MPI_COMM_WORLD);
99.     }
100.
101.    unsigned long findGroupTotal(int id, int numberOfInstances, long n, long lower) {
102.        unsigned long total = 0;
103.        for(long i = id+lower; i <= n; i += (numberOfInstances)) //first number for
    coprime
104.            for(long j = 1; j < i; j++)                         //second number for coprime
105.                if(hcf(i,j)==1)
106.                    total++;
107.        return total;
108.    }
109.
110.    long hcf(long x, long y) {
111.        long t;
112.        while (y != 0) {
113.            t = x % y;
114.            x = y;
115.            y = t;
116.        }
117.        return x;
118.    }
```