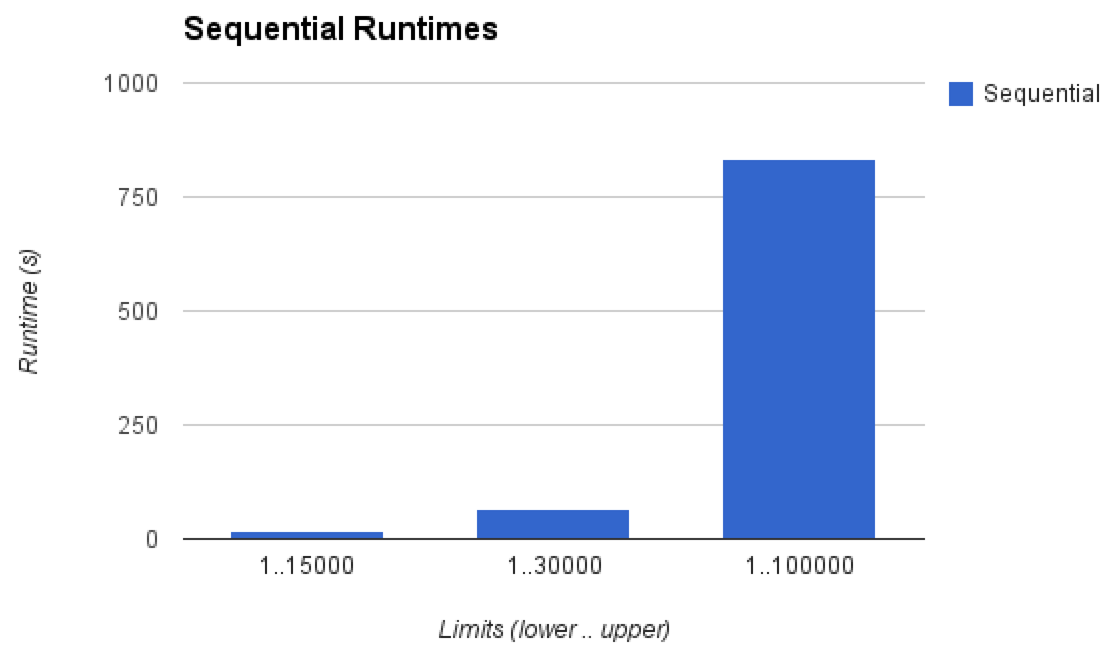
Distributed and Parallel Programming

Coursework 1 - Lucas Labanca H00176404 - Tomas Robertson H00009282

0. Sequential Performance Measurements

The sequential performance is largely predictable and understandable. The runtimes increase exponentially with larger upper limits because for each n between the lower and upper limits the program performs n*(n-1) operations.



A print out from gprof shows as expected that over 90% of execution time is spent computing the hcf() function.

```
 time    seconds    seconds    calls    s/call    s/call  name
 93.71      6.35      6.35 49995000     0.00      0.00  hcf
  4.09      6.63      0.28    10000     0.00      0.00  euler
  2.53      6.80      0.17 49995000     0.00      0.00  relprime
  0.45      6.83      0.03                              frame_dummy
  0.00      6.83      0.00        1     0.00      6.80  sumTotient
```

Looking at the hcf function is clearly the best way to optimise this problem. Either by parallelizing it or producing a solution which relies on it less or not at all. For this particular problem there is actually an entirely different solution given by $\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right),$

Cache usage is not critical as there are no large structures in memory being used. Although cachegrind shows a very large number of reads these are all of local variables and so the cache is not so important and the cache misses are negligible.

1. Comparative Parallel Performance Measurements

1.1. Runtimes

Hardware Used

OpenMP and sequential

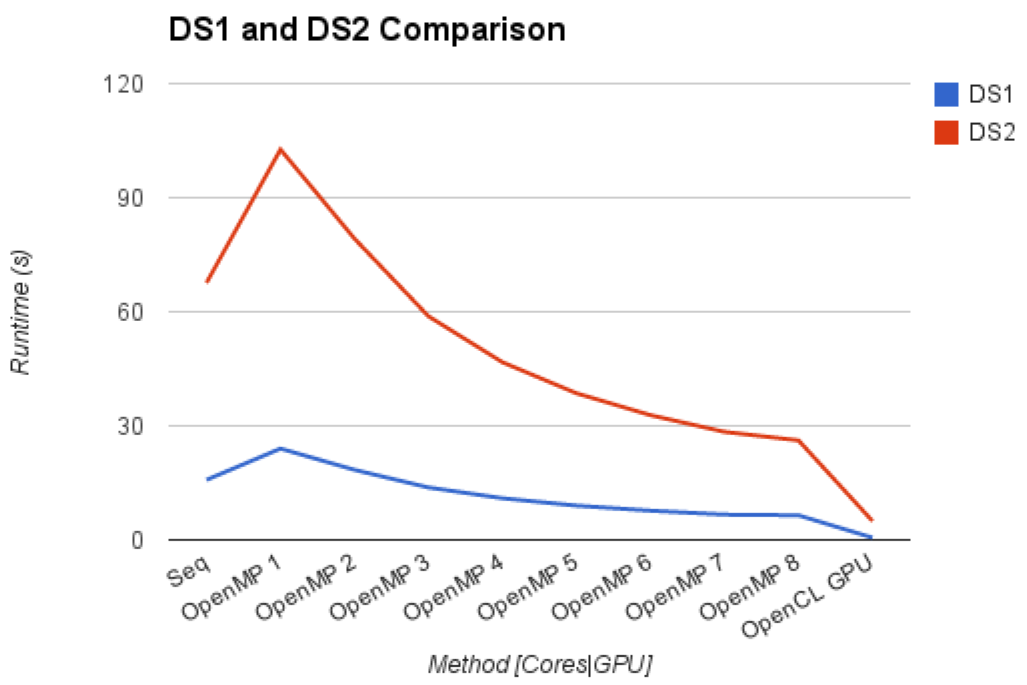
Beowulf - Intel(R) Xeon(R) E5504

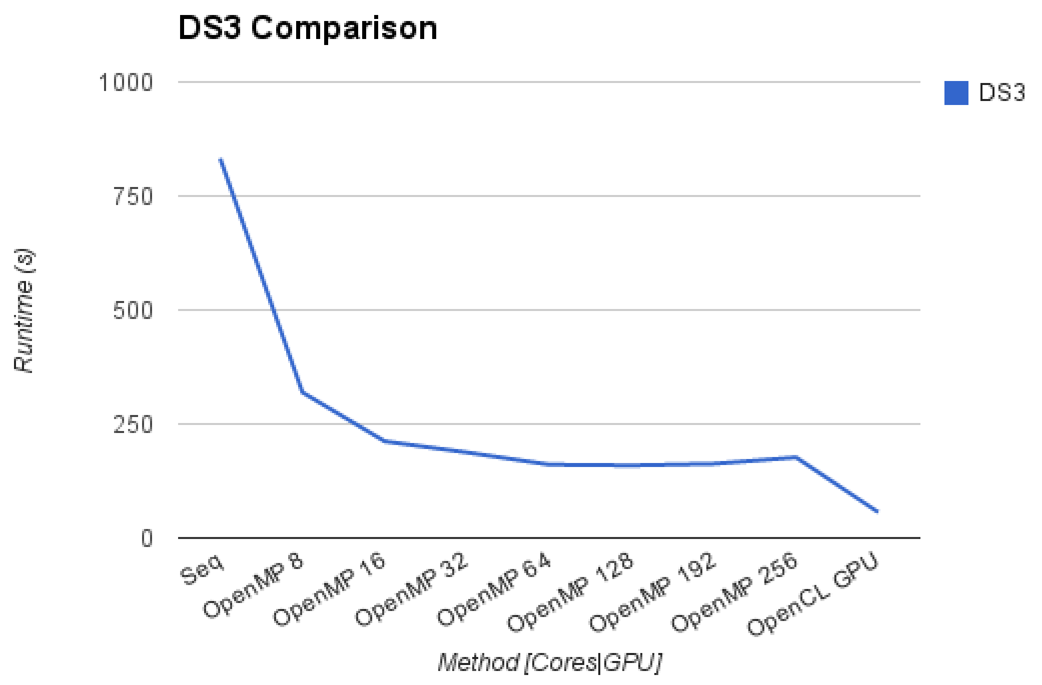
Processor Number	E5504
# of Cores	4
# of Threads	4
Clock Speed	2 GHz
Intel® Smart Cache	4 MB

OpenCL

Linux 08 - NVIDIA GeForce GT 520

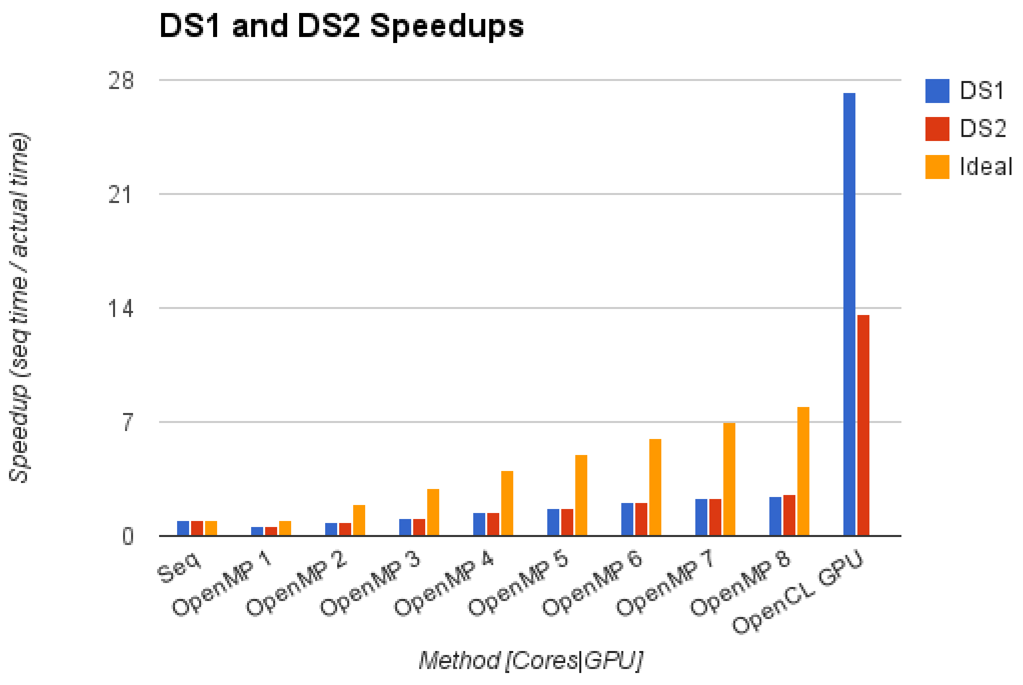
CUDA Cores	48
Graphics Clock	810
Processor Clock	1620
Memory	1024mb

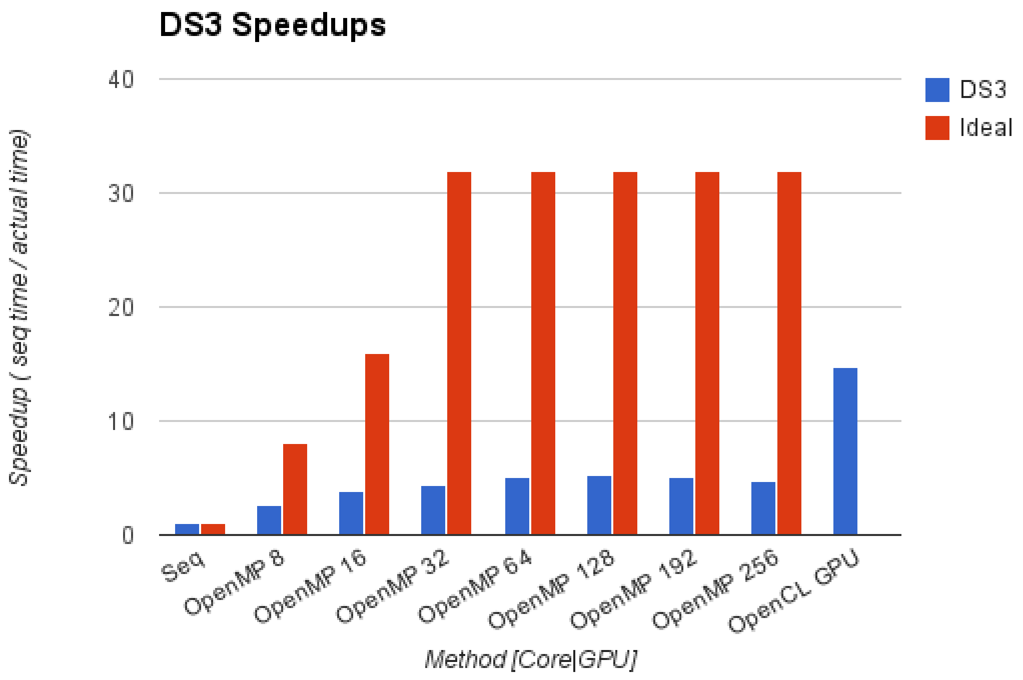




Raw Numbers available in appendix.
* The OpenCL implementation is not correct for DS3 as described in section 2.

1.2. Speedups





The ideal speedups are capped at 32 because with 32 cores the ideal speedup is 32 even when using more threads.

Raw Numbers available in appendix.

1.3. Best Results

The table below summarizes the best results obtained from the median of three running of each implementation, Sequential, using OpenMP and OpenCL. It is clear that we have some optimization at the end, in comparison with the Sequential version, but for OpenMP the best results was using a large number of threads, which demands a massive computational power from the system in general.

Table - Best running times for sequential and parallel programs

Input	Sequential	OpenMP with 128*	OpenCL
1..15000	0m15.778s	3.036s	0.579s
1..30000	1m7.589s	12.884s	4.956s
1..100000	13m52.431s	2m39.143s	56.429s

*number of threads used

1.4. Comparison between OpenMP and OpenCL

From the speedup charts we can see that OpenCL achieves a significant speedup, primarily because it utilizes the large number of cores available on the GPU and the solution is tailored to run well in parallel.

The OpenCL solution only accesses memory once per hcf call (which corresponds to one thread) which obviously allows it to run faster. This avoids the problem of transferring memory to and from the GPU which could have a performance hit on other OpenCL solutions.

The distribution of work in this particular problem is also handled in the OpenCL solution. Because as n increases there is proportionately more work to be done to solve it if the euler function is parallelized then the threads will have very uneven workloads. This would slow down a more naive solution which is possible hitting the OpenMP version depending on the scheduling used.

OpenMP manages its best speedup of ~5 times speedup when using 128 threads. This is running on a 32 core machine. We maybe expect to get a speedup similar to the number of threads used or cores, but there is many other facts that can interfere in the speedup. The bandwidth is not 128 times faster, for instance. OpenMP also reuses threads, which incurs in synchronization overhead. We believe that the overhead is one of the reason that for many threads we end up getting worst issues for running time, even with the use of dynamic scheduling. Dynamic scheduling synchronizes the threads among them to use each one at its best, avoiding waste time waiting for other threads to terminate. For small number of threads this synchronization optimizes the speedup comparing to other types of scheduling or whitout using of it.

2. Programming Model Comparison

In OpenCL organising the threads is not trivial. To produce a parallel version it is necessary to think about the problem in a different way and essentially produce a codebase.

This makes writing and understanding OpenCL solutions more complex than writing sequential C or OpenMP programs. Because the code is executed on the GPU it is necessary to communicate with the GPU and to understand the differences. A number of problems were encountered when creating the OpenCL solution as detailed in the blog.

The final solution relies on the sqrt function which as far as I was able to ascertain does not give the accuracy required for very large numbers such as when running DS3. I included the results for DS3 because although the final result is incorrect the computation done is the same and so the timings are still relevant for this report.

In OpenMP however it is only necessary to add a few lines of markup and be aware of the risks and how to avoid of race conditions. With this additional markup it is possible to retain the readability, logic and code of the original solution.

OpenCL has the big advantage of using the large number of cores on a common and relatively cheap piece of hardware. For problems where the computation can be broken down into many small discrete units this makes it a powerful option. For OpenMP to achieve its modest speedups it was necessary to run on a relatively powerful multi core machine. Depending on the hardware available this could be a strong reason to choose one model over another.

This task involved relatively little memory access, especially with the OpenCL solution using the index values as inputs rather than reading from memory. Therefore it is possible that the overhead in transferring memory to and from the GPU is not felt as badly as it might be for other problems.

Appendix

OpenCL Solution - Tomas Robertson

totient.c (Uses a simple.c which is modified to use a LongArr instead of FloatArr)

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

#include <CL/cl.h>
#include "simple.h"

#define MAX_SOURCE_SIZE (0x100000)

int main(int argc, char ** argv) {
 long upper;
 int WGS;

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

```
//Calculate the number of hcf calls required
long results_size = ((upper-1)*upper)/2;

//Array for summing the totals
long* results = (long *) malloc(sizeof(long)*WGS);
int i;
for(i = 0; i < WGS; i ++) results[i] = 0;

//Read in the kernel
FILE *fp;
char *KernelSource;
cl_kernel kernel;
fp = fopen("totient_kernel.cl", "r");
if (!fp) {
  fprintf(stderr, "Failed to load kernel.\n");
  exit(1);
}
KernelSource = (char*)malloc(MAX_SOURCE_SIZE);
fread( KernelSource, 1, MAX_SOURCE_SIZE, fp);
fclose( fp );

//Set up and run the kernel using the simple.c functions
size_t local[1];
size_t global[1];
local[0] = WGS;
global[0] = results_size;

initGPU();

kernel = setupKernel( KernelSource, "totient", 2,
                      LongArr, WGS, results,
                      IntConst, WGS);

runKernel( kernel, 1, global, local);

//Sum the Totals
long tot = 0;
int l;
for(l = 0; l < WGS; l ++)
  tot += results[l];

printf("C: Sum of Totients between [1..%ld] is %ld\n",
  upper, tot);
  return 0;
}
```

totient_cl.c

```
//Kernel source
#pragma OPENCL EXTENSION cl_khr_fp64: enable
#pragma OPENCL EXTENSION cl_khr_int64_base_atomics: enable
__kernel void totient(__global long* g_results, const int wgs) {

  size_t i = get_global_id(0);
  size_t x, y, t;

  //Reverse Triangle number formula to find x in the sequence
  // [0, 0, 1, 0, 1, 2, 0, 1, 2, 3 ... n] given index i
  double op = (8*(double)i)+1;
  x = floor( (sqrt(op)+1)/2 )+1;

  //As above to find y
  y = x-(i-((x-1)*((x-1)+1)/2));

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

  //increment an element in the results array
  if(x == 1) atom_inc(&g_results[i % wgs]);
}
```

OpenMP Solution - Lucas Labanca

```
#include <stdio.h>
#include <time.h>
#include <omp.h>

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++)
    if (relprime(n, i))
      length++;
  return length;
}
// sumTotient lower upper = sum (map euler [lower, lower+1 .. upper])

long sumTotient(long lower, long upper)
{
  long sum[1];
  sum[0] = 0;
   // Do this part in parallel

   omp_set_num_threads(8);
   omp_set_schedule(omp_sched_dynamic, 1);

 #pragma omp parallel
 {
           long i;
           #pragma omp for
           for (i = lower; i <= upper; i++)
           {
                     long e = euler(i);

                     #pragma         omp     critical
                     sum[0] += e;

           }
 }
  return sum[0];
}

int main(int argc, char ** argv)
{
  long lower, upper;

  if (argc != 3) {
    printf("not 2 arguments\n");
    return 1;
  }

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

printf("C: Sum of Totients  between [%ld..%ld] is %ld\n", lower, upper, sumTotient(lower, upper));

  return 0;
}
```

Runtime Numbers

	Seq	OpenMP 1	OpenMP 2	OpenMP 3	OpenMP 4	OpenMP 5	OpenMP 6	OpenMP 7	OpenMP 8	OpenCL GPU
DS1	15.778	24.005	18.418	13.777	10.948	9.006	7.684	6.655	6.458	0.579
DS2	67.589	102.655	79.213	58.795	46.71	38.55	32.795	28.407	26.168	4.956
	Seq	OpenMP 8	OpenMP 16	OpenMP 32	OpenMP 64	OpenMP 128	OpenMP 192	OpenMP 256	OpenCL GPU	
DS3	832.431	319.386	212.013	187.653	160.974	159.143	162.801	176.769	56.429	

Speedup Numbers

Speedup	Seq	OpenMP 1	OpenMP 2	OpenMP 3	OpenMP 4	OpenMP 5	OpenMP 6	OpenMP 7	OpenMP 8	OpenCL GPU
DS1	1	0.6572797334	0.8566619611	1.14524207	1.441176471	1.751943149	2.053357626	2.370848986	2.44317126	27.25043178
DS2	1	0.6584092348	0.8532564099	1.149570542	1.446992079	1.753281453	2.060954414	2.379307917	2.582887496	13.63781275
Ideal	1	1	2	3	4	5	6	7	8	
Speedup	Seq	OpenMP 8	OpenMP 16	OpenMP 32	OpenMP 64	OpenMP 128	OpenMP 192	OpenMP 256	OpenCL GPU	
DS3	1	2.606347805	3.926320556	4.436012214	5.171213985	5.230710744	5.113181123	4.709145834	14.75182973	
Ideal	1	8	16	32	64	128	192	256		