F20DP Distributed and Parallel Technology

Assessed Coursework 2:

Evaluating High-Level Parallel Programming models (Haskell, SaC)

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 of implementing euler totient sum equation, one using Haskell, the other using SaC. 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 would be implementing a SaC solution whereas Duncan and Nicholas would be implementing a solution in Haskell.

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 SaC and Haskell. Both of which are high level languages that aim to remove the workload from the programmer and onto the compiler when implementing parallel solutions. Writing in SaC and Haskel require less setup and knowledge of the underlying architecture in order to develop a parallel solution.

SaC stands for Single Assignment C which is an array based programming language that adds functional and implicit parallelism into a high level language. The functional qualities allows for a lack of side effects. The implementation written in this assignment used SaC version 1.2 - beta.

Haskell is a polymorphically-typed, lazy, purely-functional language. For the implementations in this assignment Haskell version 7.6.3 was used. Although Haskell allows organically allows for the creation of sequential solutions, the addition of data parallel haskell libraries provides the language with nested data parallel functionality.

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.

All of our implementations only utilise the CPUs on the robotarium. All testing was done using AMD Opteron 6376 processors.

Haskell Implementations:

As we implemented two versions of Haskell, we used different parallel strategies in order to compare them. Duncan's implementations used a mapping approach with chunks to further tune it while Nicholas used a divide and conquer strategy.

Sequential Performance Measurements

Execution Times:

Range	**1 - 15000**	**1 - 30000**	**1 - 100000**
Median 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 to 'hcf' 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 SaC calls to 'hcf' will be calculated in parallel utilising a with-loop. This with-loop applies parallelism by performing a map function across a set of n data items, where n is the upper limit of a given range. The number of threads calling the 'hcf' function will be equal to n.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. In Table 2.3 we can see a polynomial growth shown in function calls, this growing number of calls shows a clear inefficiency in the sequential implementation as it implies primes are being calculated multiple times. For instance there is a crossover in the values calculated for the euler(9) and euler(10) this problem is then amplified by the number of function calls to 'hcf' as the overlap will be calculated multiple times.

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.

3a Comparative Parallel Performance Measurements

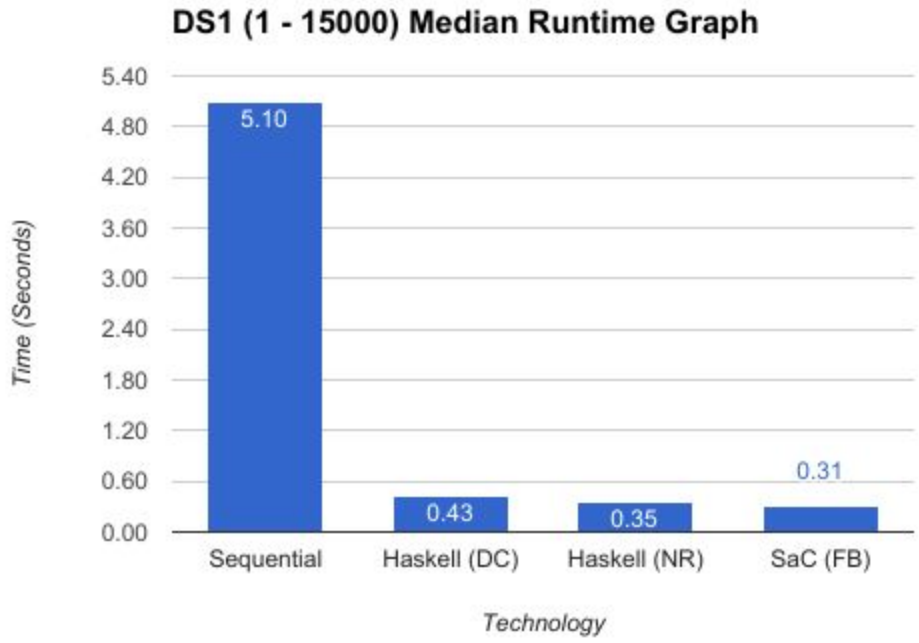


Figure 3.1 - Best Median Results for Each Technology in DS1 Range

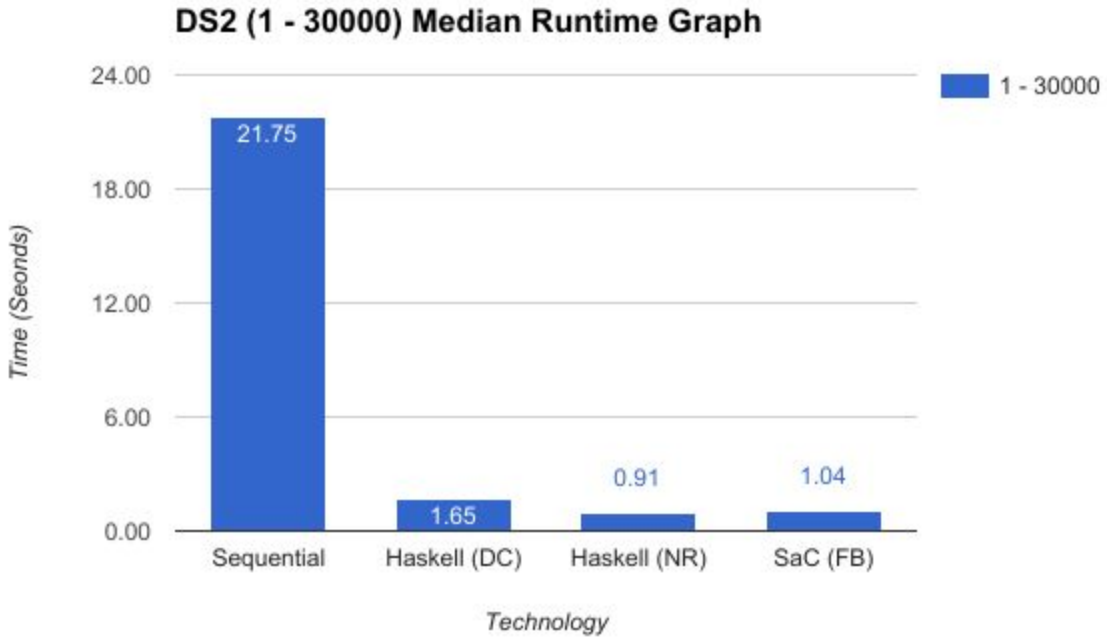


Figure 3.2 - Best Median Results for Each Technology in DS2 Range

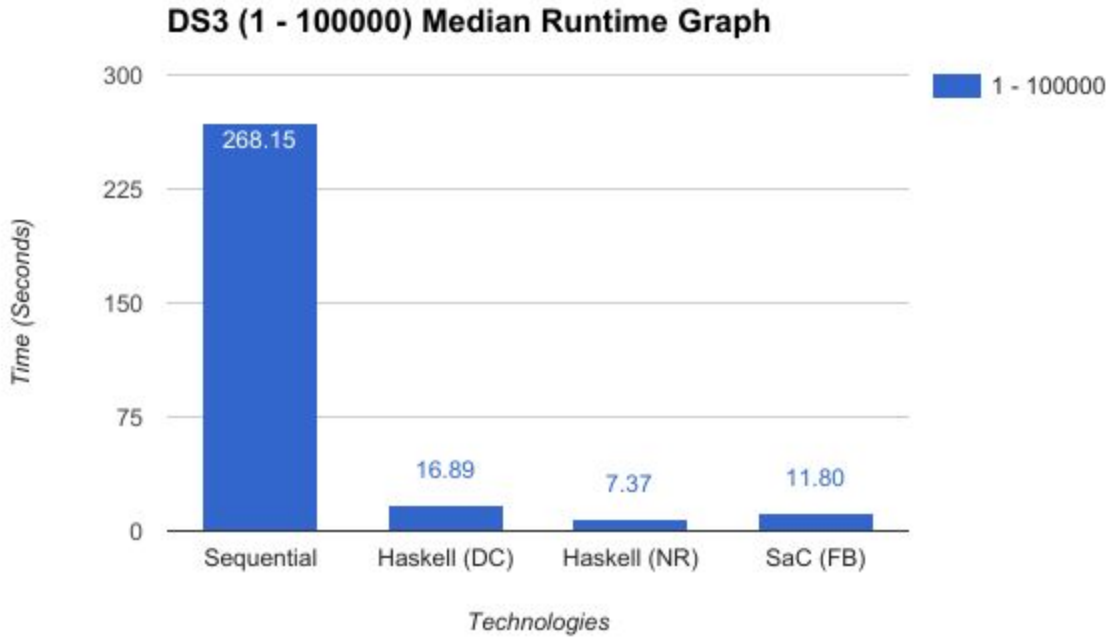


Figure 3.3 - Best Median Results for Each Technology in DS3 Range

3b Speedup

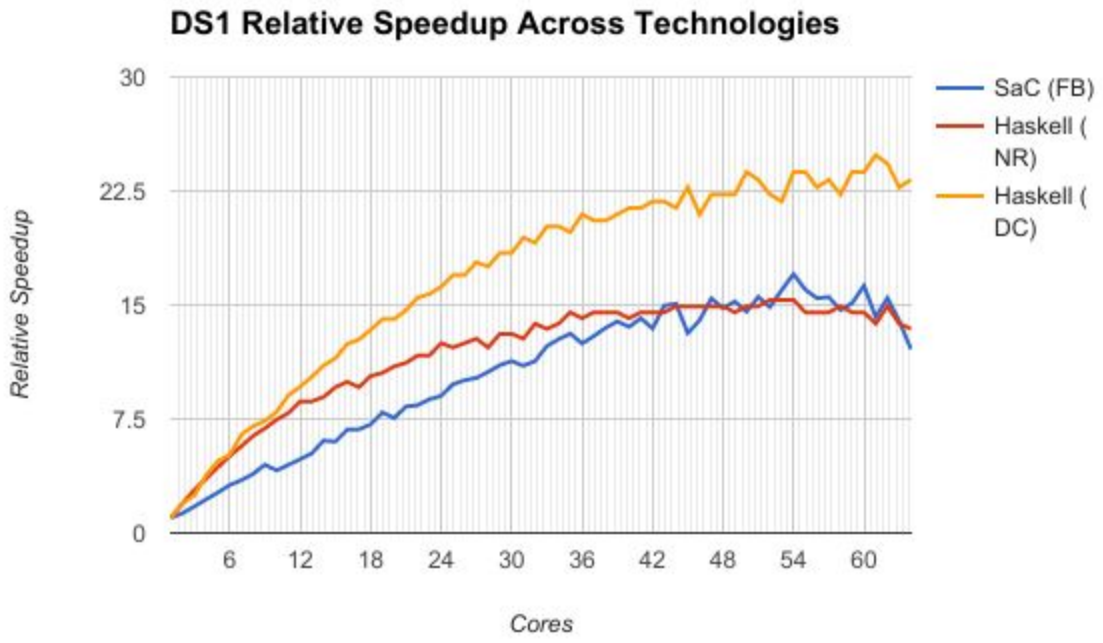


Figure 3.4 - DS1 Relative Speedup Graph

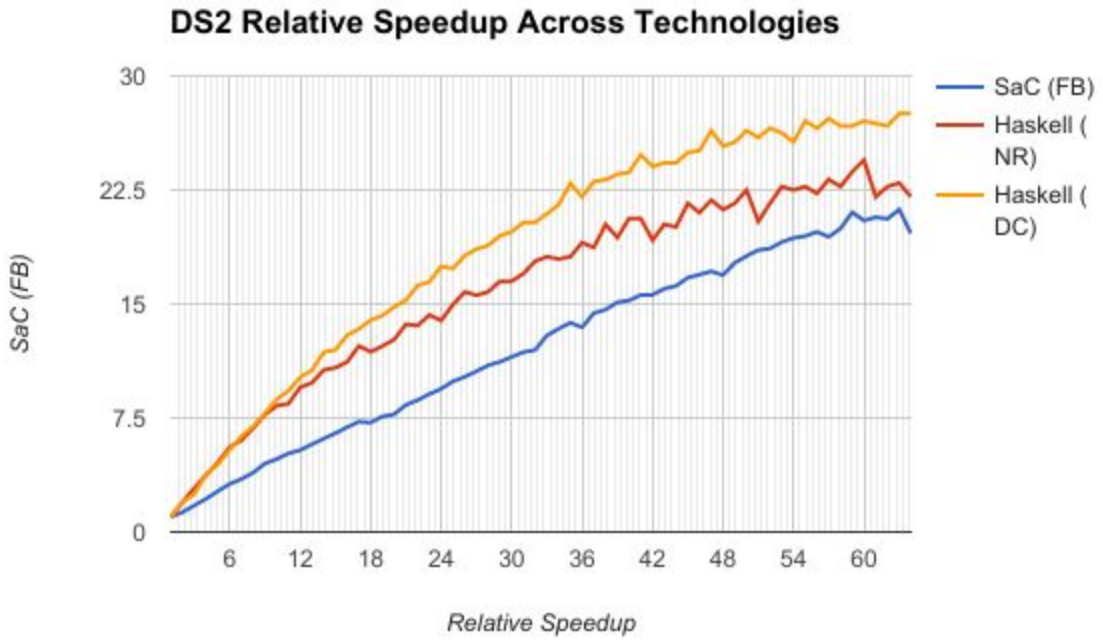


Figure 3.5 - DS2 Relative Speedup Graph

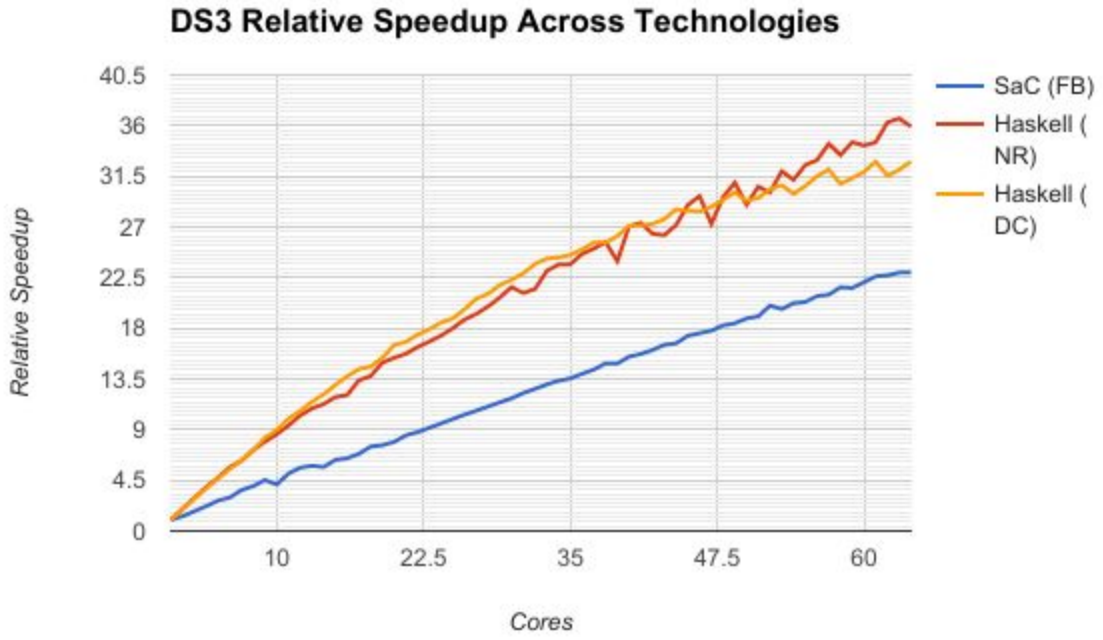


Figure 3.6 - DS3 Relative Speedup Graph

In all cases, all three implementations relative speedup effectively as they are run on more cores. The haskell implementations always have better speedup than SaC (with the exception of the divide and conquer on DS1 which is almost the same)

When comparing the runtimes, the map-based haskell implementation is much slower on 1 core and thus gets a larger speedup even though its overall performance is worse than the other two:

Runtime on 1 core	**Haskell (DC)**	**Haskell (NR)**	**SaC (FB)**	**Runtime on 64 cores**	**Haskell (DC)**	**Haskell (NR)**	**SaC (FB)**
DS1	10.69	5.37	5.23	**DS1**	0.46	0.4	0.43
DS2	45.42	22.27	22.17	**DS2**	1.65	1.01	1.13
DS3	554.28	270.05	270.82	**DS3**	16.89	7.52	11.81

3c Runtime Summary (Sequential vs Parallel)

Range	Sequential (seconds)	SaC (FB) (seconds)	Haskell (NR) (seconds)	Haskell (DC) (seconds)
1-15000	5.1	0.306837 (54 Cores)	0.35 (52 Cores)	0.43 (61 Cores)
1-30000	21.75	1.044839 (63 Cores)	0.91 (60 Cores)	1.65 (63 Cores)
1-100000	268.15	11.802248 (63 Cores)	7.37 (63 Cores)	16.89 (61 Cores)

3d Comparative Performance Discussion

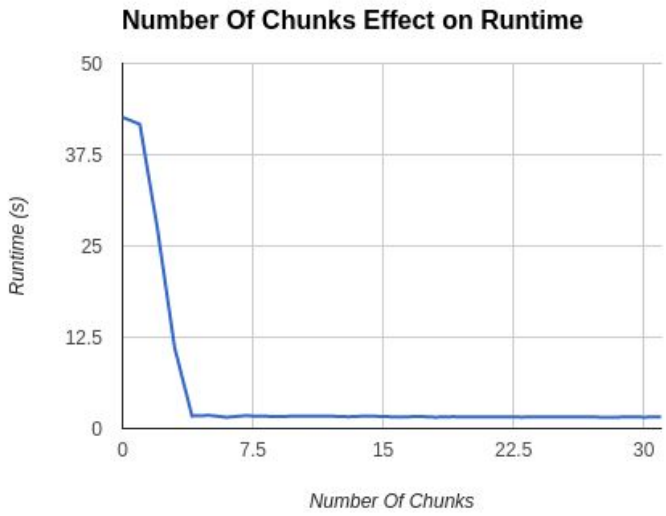
The majority of the SaC performance improvements come from rewriting the euler totient sum in a data parallel manner. In this way it can partially act as a control when comparing the two other implementations. In combination with the speed up graphs it can be argued that performance increase are cause by adding an explicit optimisation would have improved SaCs performance. However it is clear from the varying haskell results that the type of optimisation chosen has a grand effect on runtime. Therefore were normally in a low level framework such as OpenCL increase in performance is acquired through workgroup manipulations and kernel optimisations, whereas in high level languages such as Haskell and SaC this level of control is gained by the compiler and the generic optimisation chosen.

SaC vs Haskell

We see that generally the Haskell implementations have a higher relative speedup than the Sac one, but the actual runtimes on certain test sizes are quite close. However, on the largest dataset SaC's fastest time is a whole 4 seconds slower than the divide and conquer haskell implementation.

Haskell Mapping vs Divide and Conquer
For our two Haskell implementations we saw different results that we can draw from.

The map-based implementation has two ways it can be run - with chunks using `parListChunk` strategy or without chunks using `parMap`. We testing running using different numbers of chunks on 64 cores with DS2 adding chunks up to a point can be seen as having a positive effect to a certain point (at around 5) and then has no further improvements but does not get worse either.



When creating the divide and conquer, the size to use as a threshold to stop dividing was tested. Ultimately, the size settled with was 5. As lower values caused the system to unnecessarily spark new threads when computing list of a small size.

When testing the divide and conquer, the number of cores and the garbage collector size variables were tested. Using the full 64 cores did not help with lower values such as 10000, as in DS1. But from DS2 onwards using a greater amount of cores correlated with faster results. It seems that when all the cores are not utilised, then the system is better off with less cores but at high values where all cores are used, then greater cores provided a greater relative speedup. Testing the size, during all runs using less proved to be better with the best values at 20Mb. As the size increased, the system required less use of the garbage collector but caused longer times for each collection. This also caused worse cache behaviour.

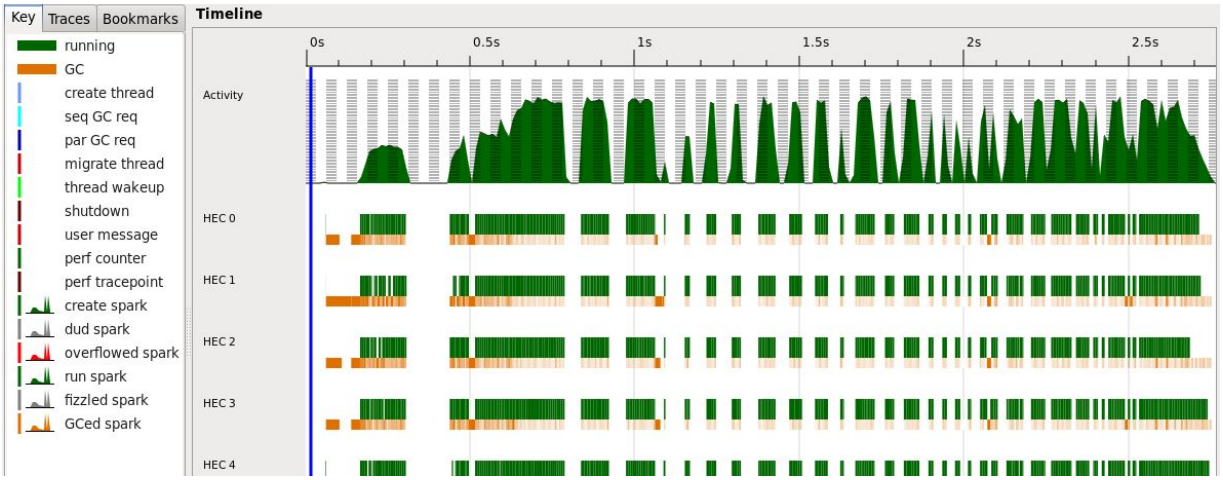
Our results show that generally the divide and conquer approach is faster than the map-based one. While the map-based approach gets higher relative speedups for DS1 and DS2, the divide and conquer catches up on DS3 due the larger data-set size. This relation suggests that on

even larger sizes, divide and conquer may overtake the map solution with regards to speedups. Furthermore the divide and conquer approach has lower runtimes in all cases tested.

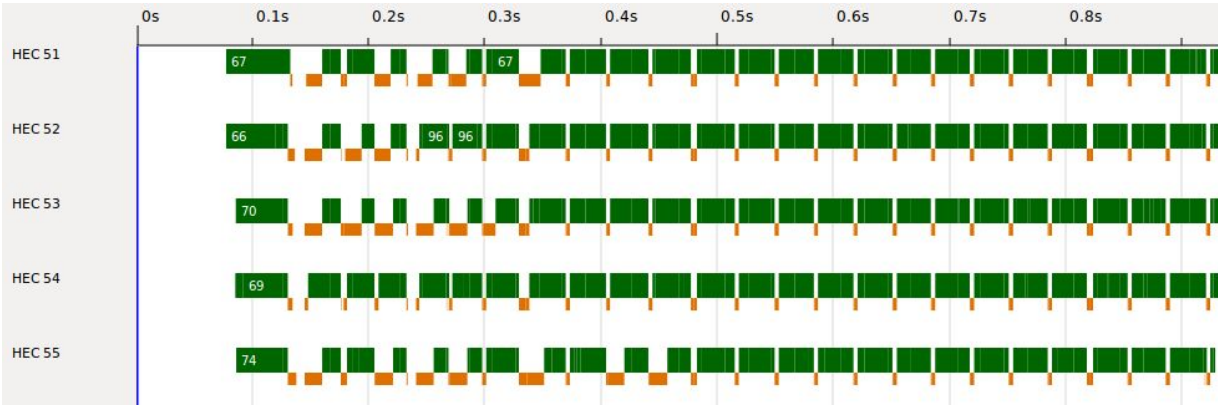
ThreadScope Analysis and Comparison

Using the threadscope tool we were able to get a more in-depth analysis of the two haskell implementations and why they had different results.

Mapping Version

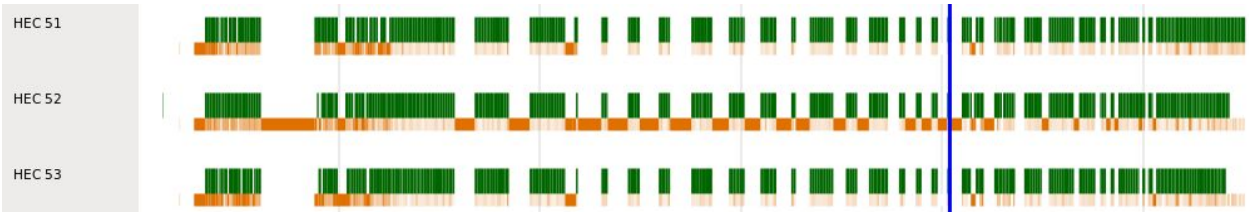


Divide and Conquer Version



While the programs are mainly broken down into small blocks for running with garbage collection in-between, it can be seen that the map version these are synced across all of the cores while in divide and conquer they are not. This means that the threads in divide and conquer can work more independently and can thus complete all of their tasks faster as they do not need to wait on more threads.

Another notable observation is that Haskell will allocate a thread to do large garbage collection at some points which temporarily halts execution of the program on other threads.



4 Programming Model Comparison

Haskell and SaC are platform independent making their results more comparable as there are no discrepancies with different hardware platforms seen with C+MPI and OpenCL. However it should be noted that each language tackles creating a parallel solution differently. Haskell allows the conversion from sequential program to a parallel solution with use of additional libraries. Whereas SaC requires that parallel solutions be built from scratch, even though there are functional language ideals that carry across, syntax does not, which may result in SaC having a slower uptake.

When using SaC it was clear that less compiler configuration and tweaking was required to get optimal performance from the code. Whereas Haskell had noticeable issues with garbage collection and memory management, requiring tweaking to be performed by the programer when compiling the script. Moving the programmers challenges to the writing the program itself and having a streamlined and simple to use compiler was a point for SaC in this case.

The support for both models should be noted. This may not be a fair comparison due to the difference in both development team size and years in development. However in programming models that are intended to alleviate strain from the programmer when developing parallel solutions, it would make sense to have more documentation and a potential community of users that can be asked. For this reason Haskell has more benefits than its SaC counterpart.

Challenges:

As with our first assessment the challenges were varied. In terms of SaC there were challenges such as lack of long support for % and summation functions, thus causing overflows in larger data ranges. Others such as lack of API and difficulty reaching pragmas to deal with chunking existed as well. However once compiled the parallel program would run as expected.

One of the challenges faced in the Haskell implementations was that adding extra cores began to slow down the program at a certain point and eventually it would slow down even below the time on 1 core. After some research we found the reason for this was due to the low amount of memory that haskell garbage collector can use. By solved this by adding the -A command line option when running the program to set a higher memory limit for the garbage collector. With the right value the program would then scale as expected and would speedup as more cores were added.

Software Implementations

With hardware no longer being a deciding factor as both solutions are platform independent, we looked at ease of use in addition to performance results in the euler totient sum solution. Haskell was simpler to write in as pervious functional language knowledge was easily applicable. In addition its ability to convert sequential programs to parallel ones ranks it higher in terms of ease of use. Haskell's production of better runtime and speedup results solidifies its position as the prefered implementation for our group.

Program Listings

SaC (Fraser Brown): Appendix A

The general approach in Fraser's SaC implementation below was to chunk the data ranges in order to gain a data parallel improvement within the SaC with-loops. For instance and example of possible grouping would be 0, 10, 20, ...150000 in one thread and 1, 11, 21 ...14991 in another thread etc. The intention here was to optimize the data sharing within the threads created in a with-loop. However there were challenges with the base euler totient sum application put serious time constraints on the ability to implement this optimisation. In addition the pragma to make this chunking process simpler on the programmer was not publicly available. Therefore the chunking of data ranges was not implemented. However a parallel solution for the euler totient sum function was implemented and proved effective in our above testing.

The SaC Implementation improves on the sequential in various ways. Each euler calculation has moved to a `with-loop` in SaC, this allows operations to be performed over a given range (in this case upper and lower limits of a given range) in a similar manner to a map function. This is where the shape-invariant programming ideology come into play within SaC, through the use of vectors (1D arrays). Vectors and with loops allowed us to perform data parallel operations without the use of indices. In addition the with loop allowed for more computation to be performed at once resulting in efficiently, admittedly this could have been extended with the use of the inline annotation, to further reduce jump operations.

Challenges:

The learning curve for SaC was rather steep to begin with, even with familiar C syntax in place. The lack of API, and having to search through the git repo for potential functions was less than ideal. There were some issues due to lack of support for various math functions for long types. Due to the large numbers we were calculating being restricted to ints was problematic, both moduleo and sum function were not supported which were two main functions required. As a result the summation for loop was created, however due to SaC compiler aggressively optimising the summation utilises tail recursions rather than a sequential loop. This allows the summation to be sub-totaled as it recurses down. Ideally I should have utilised another with-loop for the euler summation however time constraints restricted this.

Reflection:

In hindsight I should have utilised another optimisation technique such as divide and conquer when chunking was not feasible, however I was more focused on attaining a functioning base solution. The ability to gather around 20 times relative speedup in a high level language with little interaction with the hardware is incredibly impressive. It is more than worth the learning curve and struggles with libraries, which would no doubt be fixed with more experience in the language. SaC allows for a different method of thinking and frees the programmer from

restrictions in most parallel support for imperative languages, while keeping the familiar syntax. The potential power that could be achieved for solutions written with SaC is really exciting.

Haskell (Nicholas Robinson): Appendix B

Overview:
The implementation is using divide and conquer to parallelise the totient values. The system does so by halving the list whenever the list is of a greater size than 5. These halves are sparked to create threads for each half. When a list is of a size smaller than 5, it's values are computed with the euler function and then summed.

The system was optimised by using the thresholding size of 5, compared to ceasing dividing when the list is a single value.

Challenges:
Finding best garbage collection memory size was difficult, as I had no reference for good values. The values were found by running the program on multiple different sizes of garbage collection memory, from 20Mb to 80Mb.

Threshold limit was found by running the program with threshold values between 1 and 8, inclusive. The values higher than 5 could sometimes compute faster but were held back by garbage collection times. This led to 5 being the chosen value.

Choosing which of the Haskell `parallel` library functions to use was a challenge, ultimately I utilised `par` and `pseq` to convert the divide and conquer to parallel. These were chosen to allow the lower and upper halves of the list to be computed before summing to allow 3 threads to be used for each divide.

Reflection:
Using divide and conquer to parallelise the euler totient function proved very successful. Being the most successful Haskell implementation and proving to give a greater speedup than the SaC implementation.

Useful as not needing costly resources to see significant gains in speed. Even using 2 cores gives almost double the performance as a single core. This would make this a recommended method for implementing parallelisation. Haskell could also make use of large amount of cores very well, especially at high numbers, where it is likely to divide until it hits its limit.

Haskell also appeared as the stronger option of the two due to its widespread usage and the large amount of support available to it, both with documentation and libraries.

Haskell - Map (Duncan Cameron): Appendix C

Overview:
Duncan's approach was to solving the program in haskell was to utilize the use of parallel mapping functions. Two methods were developed in the source code, one using chunks with parListChunk and one using parmap without chunking.

Challenges:

While developing the tool, most of the challenge was with getting used to Haskell (especially the types). Once I got the hang of that and started experimenting with different parallel strategies things went easier. However, a problem arose when it came to testing the code on large numbers of cores. Once the initial parallel implementation was complete it would scale well when adding a few cores(up to around 4) and then adding any more the performance would start to tank. On large number of cores it would take much longer than it would on a single one. After implementing performance tuning such as the parallel mapping and chunks with the problem persisting, it turned out that the cause of this was due to Haskell garbage collector running out of memory as by default it is given very little and the parallelisation over a large number of cores requires much more. After setting the amount of memory it can access with the -A option when running it it scaled more as expected. I found 20M to be a good amount of memory.

Reflection:
I found program was very short and simple to read once implemented.

While it was not as effective overall as the divide and conquer implementation it still performed well and had very good speedups.

Appendix A

SaC (Fraser Brown): “etsfb-sac.c”

```
use ArrayTransform: {sum};
use StdIO: all;
use RuntimeError: all;
use ScalarArith: {-,!=, /, %};
use ArrayArith: all except {-, !=, /, %};
use RTClock: all;
use RTimer: all;
use CommandLine: all;
use String: {strtoi};


/* FRASER BROWN - H00155918
 * DISTRIBUTED AND PARALLEL TECHNOLOGIES
 * ASSESSED COURSEWORK 1
 * EULER TOTIENT FUNCTION IMPLEMENTATION
 * LANGUAGE: SAC (SINGLE ASSIGNMENT C)
 */

int hcf(int x, int y){
    while (y != 0){
        t = x % y; //div, mod = divmod(x, y);
        x = y;
        y = t;
    }
    return x;
}

/* Calculate the number of relprimes for each number in range n to 0 */
int relprime(int n){
    np = n - 1;
  len = with {
        (. <= [i] <= [np]) {
            hf  = (hcf(n, i)==1 ? 1: 0);
        }: hf;
    }: genarray([n], 0);
    len_sum = sum(len);
  return len_sum;
}

/* Custom sun function as stdlib sum does not work on longs */
long mysum(int[.] arr){
  long s;
  s = 0l;
  shp = shape(arr);
  for(i = 0; i < (shp[0]); i++){
```

```
46.     s = tol(s) +  tol(arr[i]);
47.   }
48.   return s;
49. }
50.
51. /*  Calculate the euler sum between n and lower (lwr) range */
52. ulong euler_tot_sum(int n, int lwr){
53.     if (lwr == 1){
54.         lwr = 2;        //due to euler 1 being ???? research and cite
55.     }
56.
57.   //touch(RTClock);  // bind function to timer
58.
59.     ets = with{
60.         ([lwr] <= [n] <= .){
61.               len = relprime(n);
62.       }: len;
63.     }: genarray([(n+1)], 0);
64.
65.   s = mysum(ets);
66.
67.   return toul(s);
68. }
69.
70.
71. int main()
72. {
73.   bool threading;
74.   int secs;
75.   int nsecs;
76.   double time;
77.   ulong ans;
78.
79.   if (argc() == 5){
80.     threading = true;
81.   } else {
82.     threading = false;
83.   }
84.
85.   /* gather comand line args */
86.   lwr, lwrstr = strtoi(argv(1), 10);
87.   upr, uprstr = strtoi(argv(2), 10);
88.   thrds, thrdstr = strtoi(argv(4), 10);
89.
90.   /* Set up and Start Timer */
91.   timer = createRTimer();
92.   startRTimer(timer);
93.
94.   /* Perform Euler Totient Sum Calculation */
95.     ets = euler_tot_sum(upr, lwr);
```

```
96.   ans = ets;
97.
98.   /* Stop and Retrive Timer Results, Destroy Timer */
99.   stopRTimer(timer);
100.      secs, nsecs = getRTimerInts(timer);
101.      time = getRTimerDbl(timer);
102.      destroyRTimer(timer);
103.
104.      /* Print Results */
105.      printf("Sum of Totients  between [%d..%d] is %lu \n", lwr, upr, ans);
106.      printf(" Thread Count: %d\n Time Taken in Seconds: %d\n Time Taken in
    NanoSeconds: %d \n Total Time: %f\n", thrds, secs, nsecs, time);
107.
108.      return 0;
109.    }
110.
```

Appendix B

Haskell (Nicholas Robinson): “ParTotientRange.hs”

```
1.  ------------------------------------------------------------------------------
2.  -- Parallel Euler Totient Function
3.  ------------------------------------------------------------------------------
4.  -- Created for Distributed and Parallel Technologies
5.  -- Coursework 2
6.  -- Nicholas Robinson
7.  ------------------------------------------------------------------------------
8.
9.  module Main(main) where
10.
11. import System.Environment
12. import System.IO
13. import Control.Parallel
14. import Control.Parallel.Strategies
15. import Control.DeepSeq
16.
17. ------------------------------------------------------------------------------
18. -- Main Function, sumTotient
19. ------------------------------------------------------------------------------
20. -- The main function, sumTotient
21. -- 1. Generates a list of integers between lower and upper
22. -- 2. Applies Euler's phi function to every element of the list
23. -- 3. Returns the sum of the results
24.
25. sumTotient :: Int -> Int -> Int
26. sumTotient lower upper = divideSumTotient lower upper
27.
28. divideSumTotient :: Int -> Int -> Int
29. divideSumTotient lower upper
30.         | (upper-lower) < 5 = sum (parMap rdeepseq euler [lower, lower+1 .. upper])
31.         | otherwise         = divideSumTotient' (length [lower, lower+1 .. upper])
    [lower, lower+1 .. upper]
32.
33. divideSumTotient' :: Int -> [Int] -> Int
34. divideSumTotient' _ []       = 0
35. divideSumTotient' size (x:xs)
36.         | (size) < 5         = sum (map euler (x:xs))
37.         | otherwise          = left `par` right `pseq` (left + right)
38.                 where mid   = size `div` 2
39.                       left  = divideSumTotient' mid (take mid (x:xs))
40.                       right = divideSumTotient' (mid) (drop (mid) (x:xs))
41.
42. ------------------------------------------------------------------------------
43. -- euler
44. ------------------------------------------------------------------------------
45. -- The euler n function
```

```
-- 1. Generates a list [1,2,3, ... n-1,n]
-- 2. Select only those elements of the list that are relative prime to n
-- 3. Returns a count of the number of relatively prime elements

euler :: Int -> Int
-- euler n = length (filter (relprime n) [1 .. n-1])
euler n = length [x | x <- [1 .. n-1], relprime n x]

----------------------------------------------------------------------------
-- relprime
----------------------------------------------------------------------------
-- The relprime function returns true if it's arguments are relatively
-- prime, i.e. the highest common factor is 1.

relprime :: Int -> Int -> Bool
relprime x y = hcf x y == 1

----------------------------------------------------------------------------
-- hcf
----------------------------------------------------------------------------
-- The hcf function returns the highest common factor of 2 integers

hcf :: Int -> Int -> Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

----------------------------------------------------------------------------
-- Interface Section
----------------------------------------------------------------------------

main = do args <- getArgs
        let
                lower = read (args!!0) :: Int -- lower limit of the interval
                upper = read (args!!1) :: Int -- upper limit of the interval
        hPutStrLn stderr ("Sum of Totients between [" ++
                (show lower) ++ ".." ++ (show upper) ++ "] is " ++
                        show (sumTotient lower upper))
```

Appendix C

Haskell (Duncan Cameron) - “coursework2.hs”

```
1.  module Main(main) where
2.
3.  import System.Environment(getArgs)
4.  import Control.Parallel
5.  import Control.Parallel.Strategies
6.  import Control.DeepSeq
7.  import GHC.Conc (numCapabilities)
8.  import Data.Time.Clock (NominalDiffTime, diffUTCTime, getCurrentTime)
9.
10.
11. main = do args <- getArgs    -- read command-line arguments
12.        let
13.               minVal = readIntegerK (args!!0)
14.               maxVal = readIntegerK (args!!1)
15.               chunkSize = readIntegerK (args!!2)
16.               answer = if chunkSize == 0 then eulerToitent minVal maxVal else
    eulerToitentChunks minVal maxVal chunkSize
17.        putStrLn("Chunk Size: "++(show chunkSize))
18.        putStrLn ("Workers: "++(show numCapabilities))
19.        putStrLn ("Running ...")
20.        t0 <- getCurrentTime
21.        putStrLn (answer `deepseq` "done") -- force it only, ignore result
22.        t1 <- getCurrentTime
23.        putStrLn (" Toitent Sum of " ++ (show maxVal) ++ " using " ++ (show
    numCapabilities) ++ " cores is "++(show (answer-1))) --for some reason answer is always
    +1...
24.
25. -- read large integer values using shorthand such as 90k
26. readIntegerK :: String -> Int
27. readIntegerK str = case (last str) of
28.                       'K' -> 1000 * (read (init str) :: Int)
29.                       'k' -> 1000 * (read (init str) :: Int)
30.                       'M' -> 1000000 * (read (init str) :: Int)
31.                       'm' -> 1000000 * (read (init str) :: Int)
32.                       'G' -> 1000000000 * (read (init str) :: Int)
33.                       'g' -> 1000000000 * (read (init str) :: Int)
34.                       _  -> (read str :: Int)
35.
36. -- With no chunking
37. eulerToitent :: Int -> Int -> Int
38. eulerToitent n m = eulerToitentParMap [n..m]
39.
40.
41. eulerToitentParMap :: [Int] -> Int
42. eulerToitentParMap l = sum totals
43.                       where
```

```
                        totals = parMap rdeepseq euler l

--with chunking
eulerToitentChunks :: Int -> Int -> Int -> Int
eulerToitentChunks n m chunkSize = eulerToitentChunkMap [n..m] chunkSize

eulerToitentChunkMap :: [Int] -> Int -> Int
eulerToitentChunkMap l chunkSize = sum totals
                        where
                        totals = map euler l `using` parListChunk chunkSize rdeepseq

euler :: Int -> Int
euler n = euler' n n

euler' :: Int -> Int -> Int
euler' _ 0 = 0
euler' _ 1 = 1
euler' n m = coprime + otherToitents
                where coprime = (isCoprime n m)
                        otherToitents = (euler' n (m-1))

isCoprime :: Int -> Int -> Int
isCoprime n m = if hcf n m == 1
                then 1
                else 0

-- Highest Common Factor Function
hcf :: Int -> Int -> Int
hcf m n = let small = min m n
                big = max m n
                remainder = big `mod` small
        in
        if remainder == 0 then small
        else hcf small remainder
```