Heriot Watt University

Student ID: H00148811 & H00174168

Student Name: Wanchana Ekakkharanon & Yuting Zhu

Programmes: MSc in Data Science & MSc in IT (Software Systems)

Modules: Distributed and Parallel Technology (F21DP)

Term/Year: Term 2 / 2015

Date of Submission: 31th Mar 2015

Contents

Introduction

Parallel programing is a crucial technics to answer the rapidly growing data nowadays. Parallel application take the advantage of multiple core machine to process multiple task in multiple cores of CPU or GPU at the same time resulted in an increasing performance of the application. But, writing a parallel code is not an easy thing to do. The High-level Parallel programing offer the tools for a developer to flexibly coding without worried about the actual hardware inside the machine or about macro managing the massage passing like in the low-level parallel programming. The aims of this coursework are to study High level Parallel programing, mainly focus on Haskell and Single Assignment C (SaC) with C also looking into the performances, advantages, and disadvantages of both programming languages and compare them with each other and with traditional sequential programme. The task will be perform by a pair. One person will develop a code for Haskell and another will develop the code for SaC. Then the pair will tune all the codes (Both Parallel and Sequential) together.

All programmes will run on a single Beowulf 29. The Haskell programmes is compiled by Glasgow Haskell Compilation System version 7.6.3 and SaC programme is compiled by SaC version 0.9.1. All the result will be collect and discuss in this report.

Sequential Performances Measurements

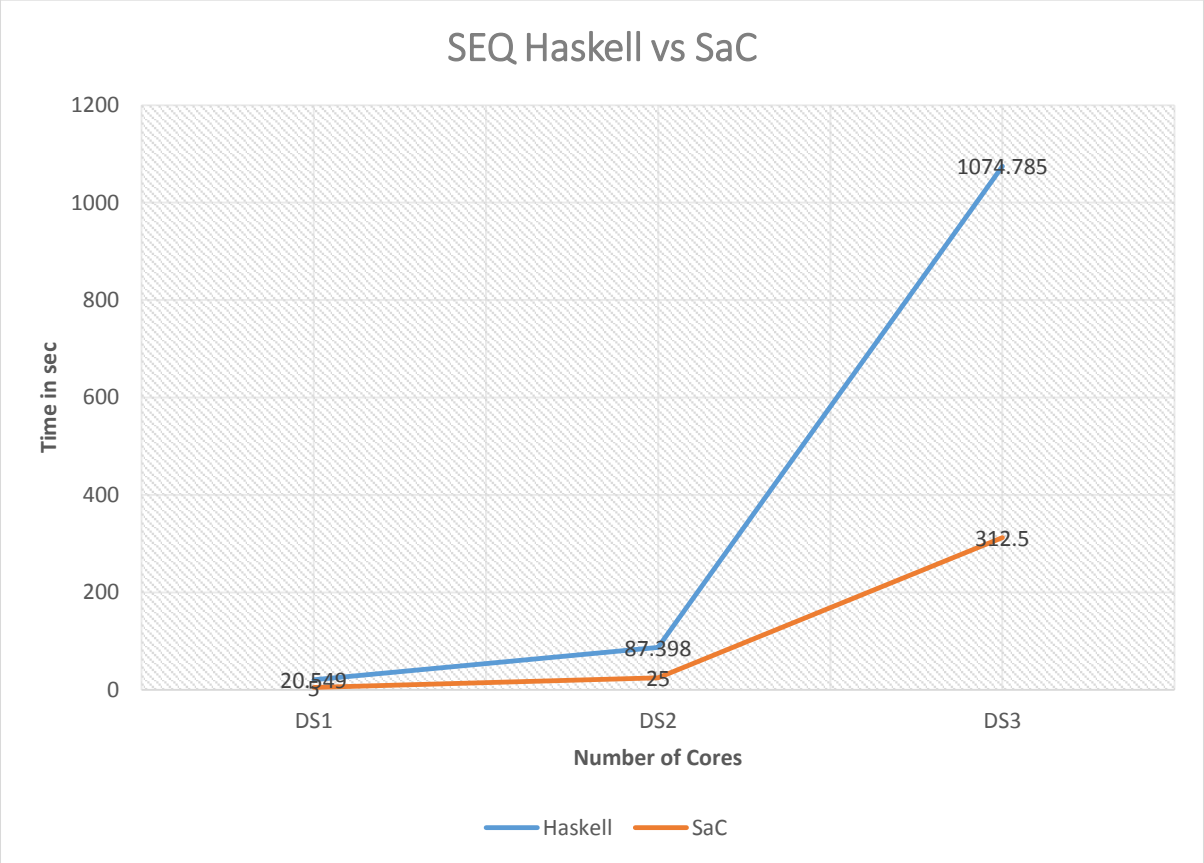

Figure1: Runtime Chart of sequential programmes Haskell vs SaC.

The sequential programs run on the same machine. We calculated the sum of Totient between 1 and 15000, 1 and 30000, 1 and 100000 original version of both programming languages. The run-times are shown in Figure 1 above.

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

Haskell sequential program obviously slower because Haskell use CPU to process the task and technically single core CPU cannot keep up with even a cheap GPU with SaC programming language as you can see in Figure1. When comparing the performance to the first coursework both Haskell and SaC have lower performance than a simple C programming. The reason may come from their standard library because they are designed to do high performance computing on multiple cores. So in order to run one core they still have to do more minor check in the libraries than a simple C programme

Comparative Parallel Performance Measurements

Runtimes Graphs

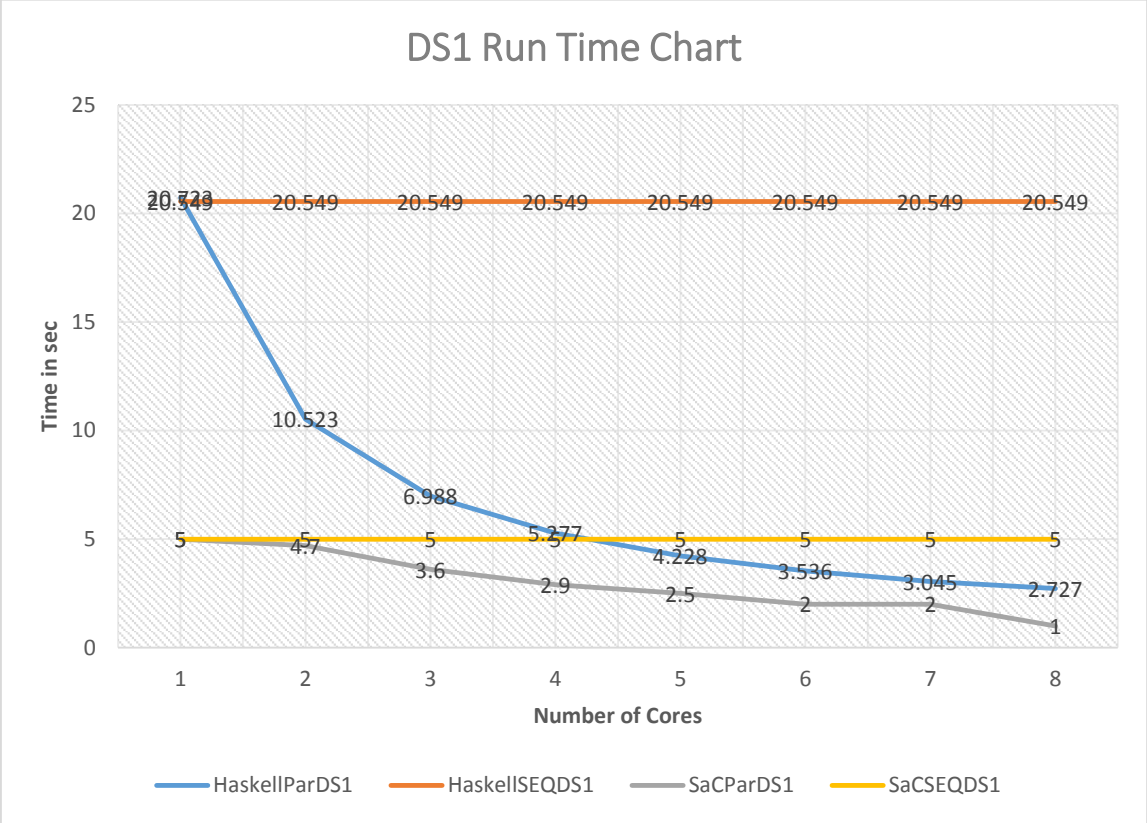

Figure2: Runtime Graphs of DS1 parallel-sequential programmes.

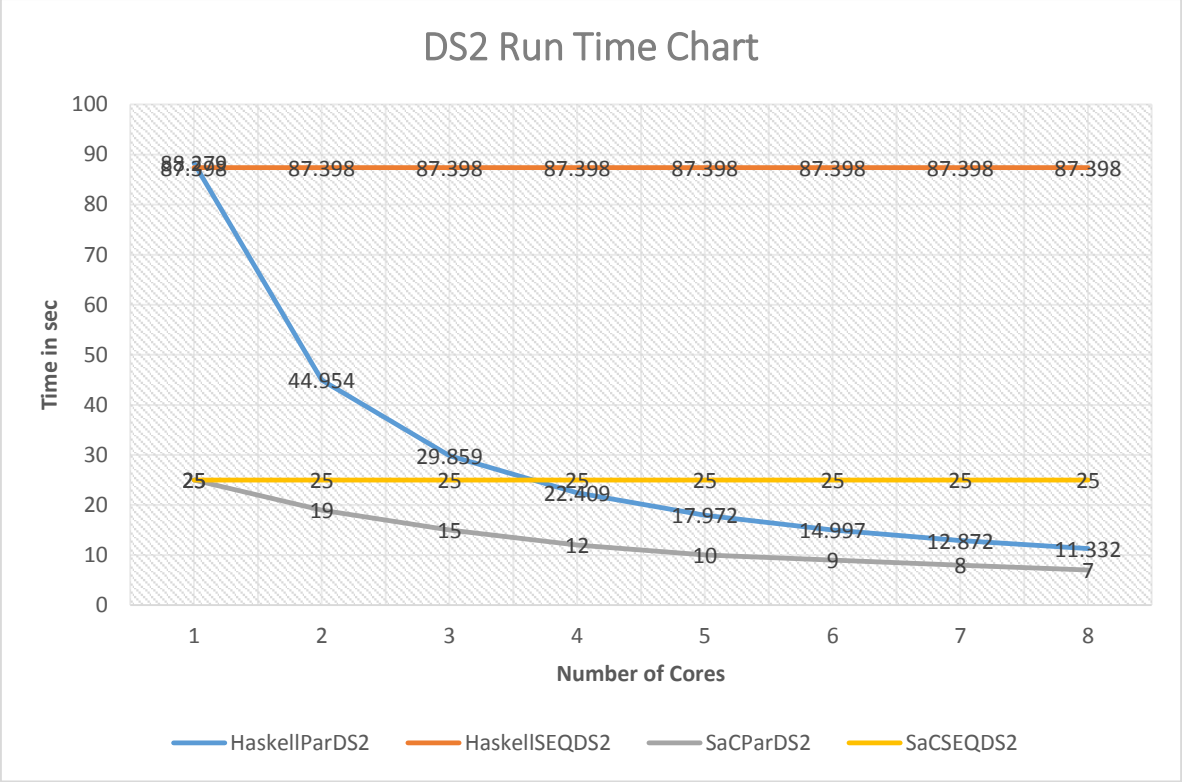

Figure3: Runtime Graphs of DS2 parallel-sequential programmes.

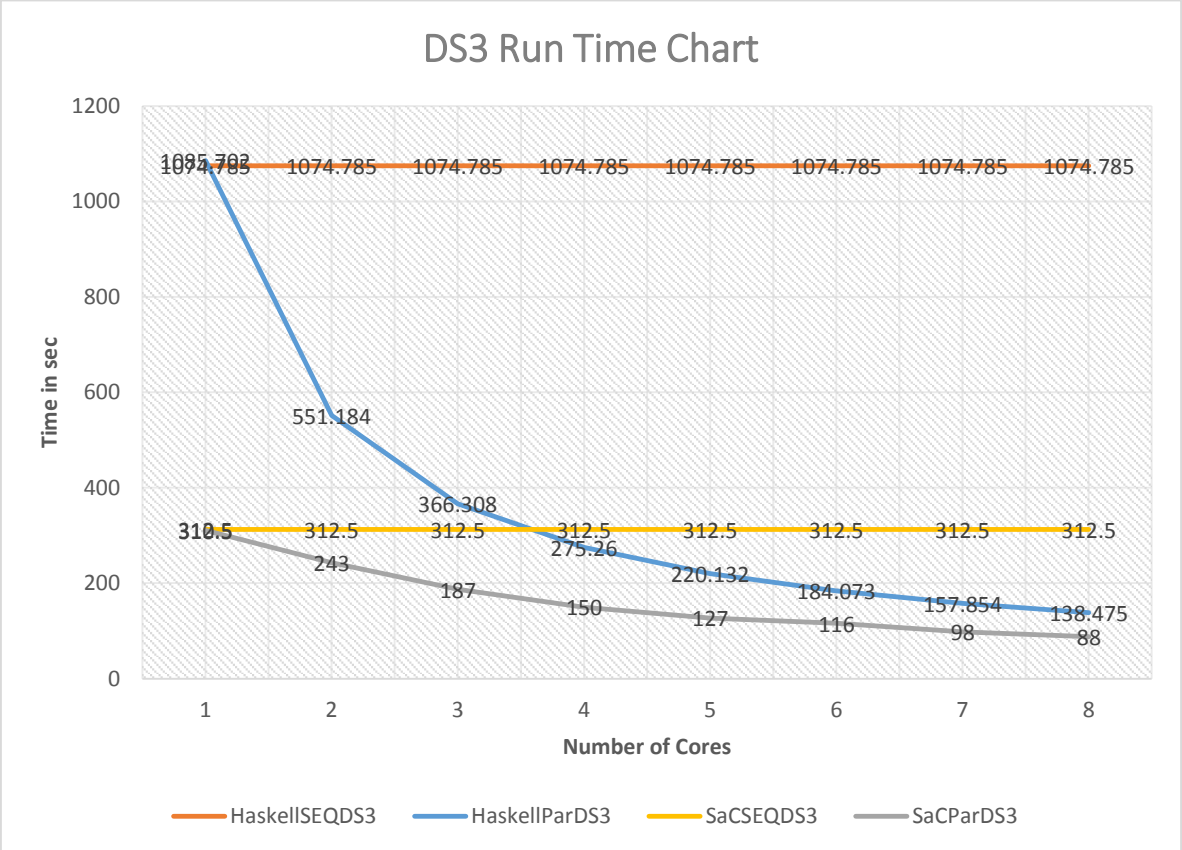

Figure4: Runtime Graphs of DS3 parallel-sequential programmes.

Speedups Graphs

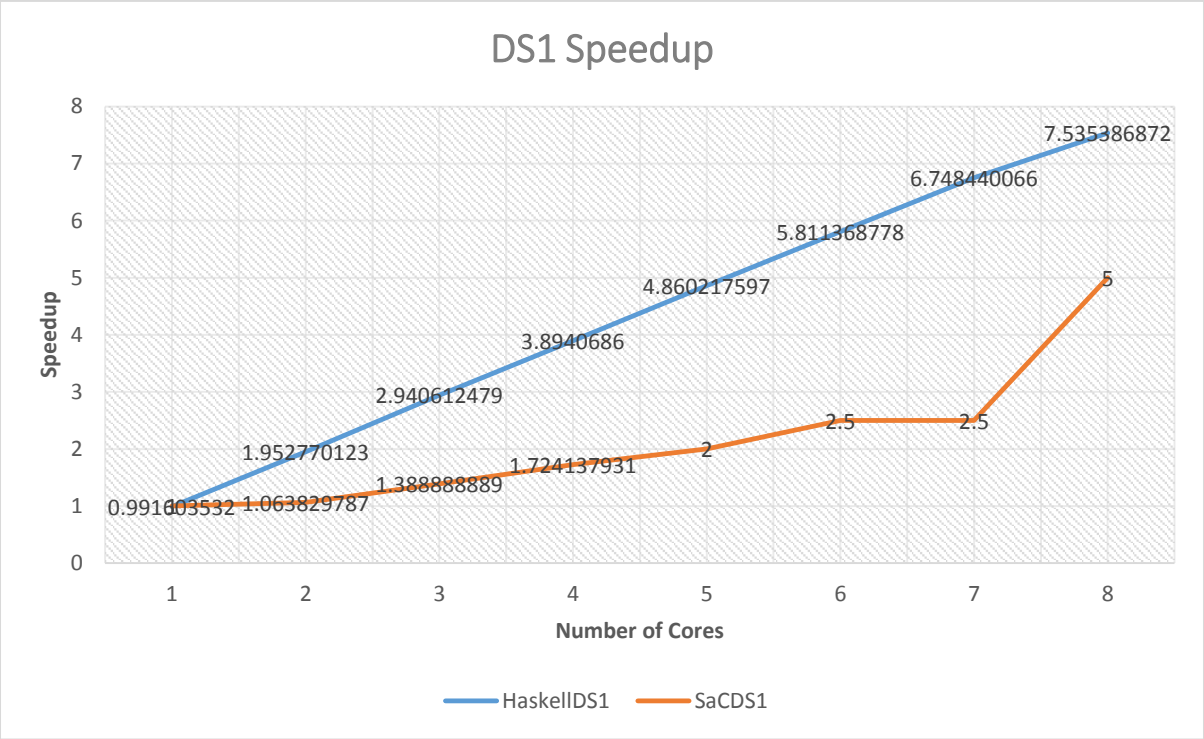

Figure5: Speedup Parallel programmes against the Sequential programme on DS1.

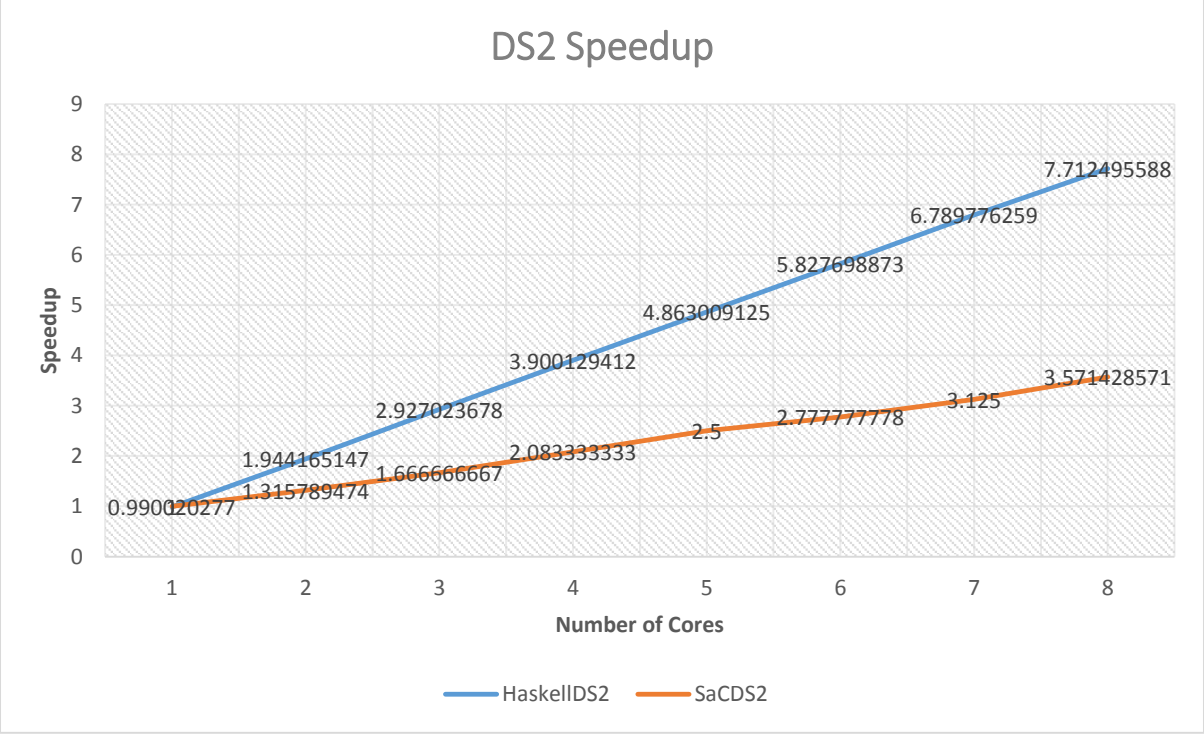

Figure6: Speedup Parallel programmes against the Sequential programme on DS2.

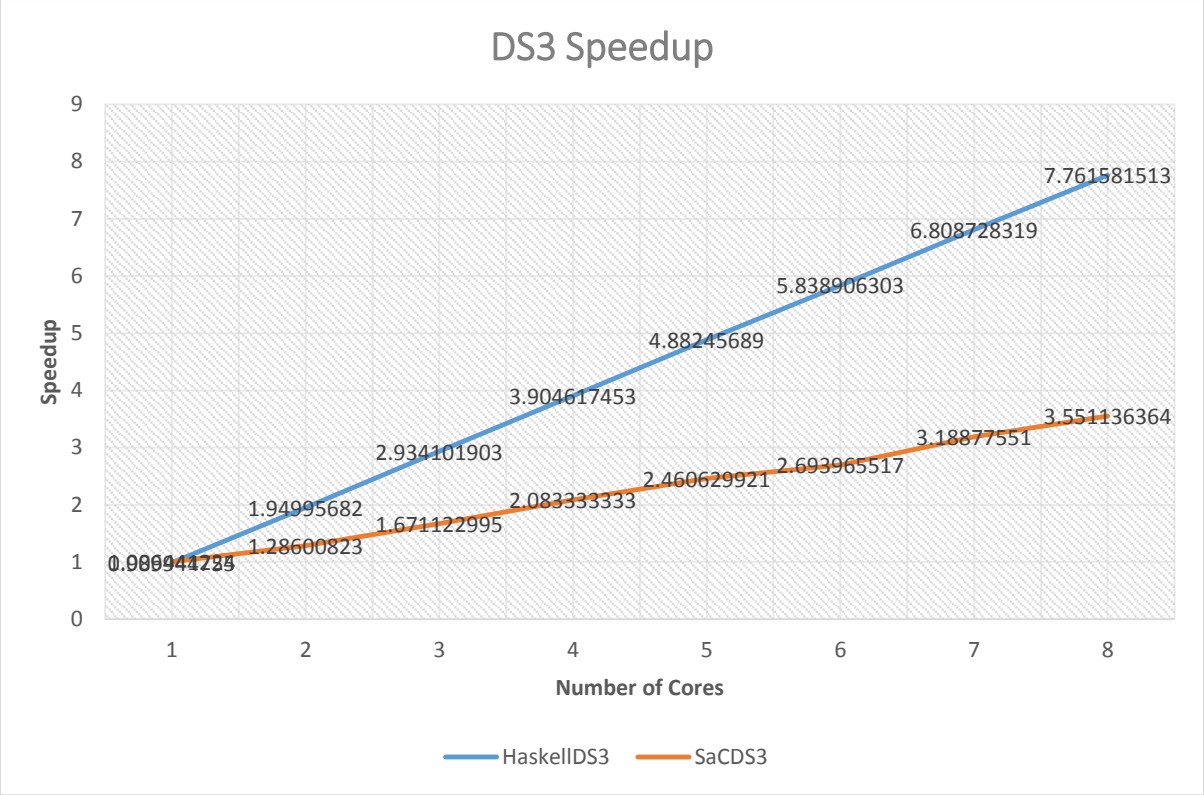

Figure7: Speedup Parallel programmes against the Sequential programme on DS3.

Performances Table

Median Value Table								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
HaskellParDS1	20.723	10.523	6.988	5.277	4.228	3.536	3.045	2.727
HaskellParDS2	88.279	44.954	29.859	22.409	17.972	14.997	12.872	11.332
HaskellParDS3	1085.702	551.184	366.308	275.26	220.132	184.073	157.854	138.475
HaskellSEQDS1	20.549	20.549	20.549	20.549	20.549	20.549	20.549	20.549
HaskellSEQDS2	87.398	87.398	87.398	87.398	87.398	87.398	87.398	87.398
HaskellSEQDS3	1074.785	1074.785	1074.785	1074.785	1074.785	1074.785	1074.785	1074.785
SaCParDS1	5	4.7	3.6	2.9	2.5	2	2	1
SaCParDS2	25	19	15	12	10	9	8	7
SaCParDS3	310.5	243	187	150	127	116	98	88
SaCSEQDS1	5	5	5	5	5	5	5	5
SaCSEQDS2	25	25	25	25	25	25	25	25
SaCSEQDS3	312.5	312.5	312.5	312.5	312.5	312.5	312.5	312.5

Table1: Table contain the all median from every cores of all programmes.

Comparison

The sequential programme of both Haskell and SaC took a lot of runtime to process the task as you can see in Figure1. But the performance of the Haskell is outperformed by SaC which is quite obvious that GPU should have better performance than CPU on a single thread.

When in come to parallel programme, SaC outperform the sequential programme and also Haskell in all Dataset in all number of thread (1-8 threads in this report) the number of thread greater increased the performance in each dataset as you can see in Figure2, Figure3 and Figure4 but when we consider the speed up chart in Figure 5 to Figure 7, increasing the number of thread did not gain much speed up compare to Haskell. On another hand Haskell performance gain a lot more speed up against its sequential programme and SaC. Overall runtimes of Haskell also greatly improve when the number of thread go up, at 8 cores CPU it almost catch up with SaC programme performance.

Both programming may also improving the performance by modify a filter function that find a relative prime for N number but since we have a problem implement this with Haskell so we keep both parallel the same to reduce the bias that may happen in this report.

Programming Model Comparison

The Haskell development in this coursework is not simple at all in our case. Mainly because we have only a few experience with function programming. Haskell use CPUs to process the task. The main feature of Haskell is Lazy Evaluation and List comprehension. The good point of the High-level parallel programming language like Haskell is you don't have to worry much about managing point to point messaging like in OpenMPI. The list of problem are as follow.

- First time when we start to write the code, the structure of the code have so much different from the C and Java that we have lot experience with them. It took us quite some times before I can run the code.
- After we can run the code next we apply Divide and Conquer design pattern to the programme with a threshold to a programme but when we run the programme with threshold = 1000 for example, the performance is lower than threshold = 0 which is weird and we still don't know the reason behind it. When we use the profiler to check the sparks creation and running, it's appeared that there are a lot of activity from garbage collector than the running sparks.
- Next we try to implement parFilter function to do a filtering in parallel using Control.Monad.Par package but we cannot be able to compile the code. The Error message show that the compiler cannot find this package. We did try to install the package but the console just show that it has already been installed. In the end we have to use only Divide and Conquer design pattern in our programme but you can still see this part of the code inside our source file
- We also tried to use other Design pattern but it's hard to find an example code.

Overall the Haskell programme still run quite well as you can see from the result in previous section. The main problem is Haskell have only a few tutorial and document on the parallel programing. So it require more experience in Haskell language before we can actually produce a well optimise programme.

SaC is also high-level programming the same as Haskell but SaC process its task by using GPU inside machine. Developing Totient application in SaC compare to the Haskell, the code structure is much easier than in Haskell because it very similar to C programming language. We found that SaC is more suitable for us to developing totient application since it have structure like: "with-loop" in SaC which similar to a simple for loop in C if you can put thing into array by using "with-loop" SaC will do a parallel for you. But we still found some problem while coding a programme.

Following are the challenges we encountered in constructing SaC.

- At the beginning we always make mistakes on the type of variables as there no need to have type declaration.
- We didn't know how to receive parameters from user at the beginning. We searched and finally we use *String::argv* to do this.

We think the SaC is powerful language the same as OpenCL in previous coursework. The code is quite easy to read and easy to find an example. The performance of programmes is also better than the Haskell with the same thread size. The only problem is you have to put your task in the array format and each element in array should be independence to each other which is hard in some case.

Reflection on Programming Model

Haskell programming model should be easy for a people who do a math all the time it is also straightforward when you understand it. You can't control much thing like in the low-level parallel programing but it easier to learn because you have to care only which part of the code is sequence and which you can do it in parallel. Haskell also provided a lot of parallel strategies to use within your code. But there are few example and document it can be troublesome to really optimise your code (Maybe not if you are a Mathematician).

The Haskell performance are mainly depended on how well you separated the parallel task and sequential task inside your programme (Load balancing) and how many thread you want to run the programme. Separating the sequential task and parallel task required some experience if you put the very expensive task in you sequential block it not going to help improving the performance at all. The number of thread in Haskell can cause the performance to drop if you try to run on thread size larger than number of your CPU cores. Haskell process the task with CPU, as you can see in the result section it's slower than SaC that use a GPU to process the tasks but Haskell still can run on a cluster of machine but it not cover in this report. The performance can become close to SaC.

SaC is much easier to write the code, you can easily port your old C code to make it run as a parallel programme. SaC has also have good scalability as Haskell and it also can run on cluster machine. SaC is efficient. Compared to the run time of sequential program, the run time of using 8 threads to run the parallel program has been reduced nearly 80%. The only problem that we found is SaC have less available variable type than C. For example we thought it is suitable to using "long" type in Totient application. However "long" is undefined in SaC, this can be problem in some case. Furthermore the cost of the GPU base cluster machine may be cheaper than CPU base (It might not be true because all machine required a CPU to operate).

Basically if you have no programming background but storing in Mathematic it much easier to develop your programme by using Haskell. Bu if you already have your old code and you are C guy, SaC is right for you.

Appendix A

Parallel Haskell Totient programme.

```
============================================================================
Name        : ParHaskell.hs
Author      : Wanchana Ekakkharanon
Version     :
Copyright   : Your copyright notice
Description: The code below parallel Totient programme using Haskell programming language.
By using Divide and Conquer design pattern with Threshold.
============================================================================
-----------------------------------------------------------------------
-------
-- Sequential Euler Totient Function
-----------------------------------------------------------------------
-------
-- This program calculates the sum of the totients between a lower
and an
-- upper limit, using arbitrary precision integers.
-- Phil Trinder, 26/6/03
-- Based on earlier work by Nathan Charles, Hans-Wolfgang Loidl and
-- Colin Runciman
-----------------------------------------------------------------------
-------

module Main(main) where

import System.Environment(getArgs)
import Control.Parallel
import Control.Parallel.Strategies
--import Control.Monad.Par.Scheds.Trace
--import Control.Monad.Par (NFData)
import Control.DeepSeq
import GHC.Conc (numCapabilities)
import Data.Time.Clock (NominalDiffTime, diffUTCTime,
getCurrentTime)

-----------------------------------------------------------------------
-------
-- Main Section
-----------------------------------------------------------------------
-------
main = do args <- getArgs       -- read command-line arguments
          let
          lower = read (args!!0) :: Int -- lower limit of the
interval
            upper = read (args!!1) :: Int -- upper limit of the
interval
          t = read (args!!2) :: Int -- threshold
          res_dnc = parsumTotient lower upper t --result of
parsumTotient
```

```
        putStrLn ("Workers: "++(show numCapabilities)) -- print
number of concurrency
        putStrLn ("Running ...")
        t0 <- getCurrentTime
        putStrLn (res_dnc `deepseq` "done") -- force it only, ignore
result
        t1 <- getCurrentTime
        putStrLn ("Sum of Totients between [" ++ (show lower) ++
".." ++ (show upper) ++ "] is " ++ (show res_dnc)) -- divide-and-
conquer version

-------------------------------------------------------------------
-------
-- Parallel worker function, divide-and-conquer with thresholding
-------------------------------------------------------------------
-------
parsumTotient :: Int -> Int -> Int -> Int
parsumTotient m n t | m>n        = error ("Invalid range: "++(show
(m,n)))
             | (n-m) <= t = sumTotient m n           -- seq version
if interval size <= t
             | otherwise  = left `par` right `pseq`  -- par d&c
version
                            (left + right)
                            where mid = (m + n) `div` 2
                                  left = parsumTotient m mid t
                                  right = parsumTotient (mid+1) n t


-------------------------------------------------------------------
-------
-- Sequence Function, sumTotient
-------------------------------------------------------------------
-------
sumTotient :: Int -> Int -> Int
sumTotient lower upper = sum (map euler [lower, lower+1 .. upper])

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

-------------------------------------------------------------------
-------
-- parFillter
```

```
-------------------------------------------------------------------
-------
--parFilter :: (NFData a) => (a -> Bool) -> [a] -> Par [a]
--parFilter f [] = return []
--parFilter f (x:[]) = if f x then [x] else return []
--parFilter f xs = do let (as,bs) = halve xs
--             v1 <- spawn $ parFilter f as
--             v2 <- spwan $ parFilter f bs
--             left <- get v1
--             right <- get v2
--             return $ left ++ right
--      where
--    halve :: [a] -> ([a], [a])
--    halve xs = splitAt (length xs `div` 2) xs

-------------------------------------------------------------------
-------
-- 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)
```

Appendix B

Parallel SaC Totient programme

```
/*

============================================================================
========
 Name        : ParSaC.SaC
 Author      : Yuting zhu
 Version     :
 Copyright   : Your copyright notice
 Description : This is the prallel version of Totient application.
The euler part of the    sequence program is beng pralleled. This
appliaction use "with" to concurrently calculate euler then put each
result into the array i.

============================================================================
========
 */
//Parallel by using origional sequence program
use StdIO: all;
use Array: all;
use CommandLine: all;

inline double euler(double n) {

  length = 0d;
  b = toi(n);
  for (i = 1; i <= b; i++) {
    x = b;
    y = i;
    while(y != 0) {
     t = x % y;
     x = y;
     y = t;
    }
    if(x == 1) {
     length=length+1d;
    }
  }

  return (length);
}



int main() {

   lower = String::toi( argv(1) );
   upper = String::toi( argv(2) );

  total = with {
        ( [lower] <= [i] <= [upper]) : euler( tod(i) );
                } : fold(+, 0d);
   printf("%10.0d\n",total);

    return (0);
```

}