F21DP2
Distributed and Parallel Technology
Assessed Coursework 1

Evaluating Parallel Programming Models

Andrew Beveridge

ab441@hw.ac.uk / H00013703

Problem

Develop and measure parallel versions of a program which calculates Euler's totient using one of the following high-level parallel programming technologies:

Glasgow parallel Haskell (GpH) or Single Assignment C (SaC).

Compare the performance and programming models of these technologies.

Implementation

Following on from my previous coursework submission with a similar topic, I found the provided sample code in C to be very inefficient and used my own, simpler algorithm as a sequential starting point for all other code. I was also dissatisfied with only calculating three sets of results, as in my initial testing that was not enough to draw confident conclusions.

As such, I decided to build a fairly substantial web-based build and testing platform for parallel code, written in PHP using the Symfony MVC framework and a MySQL database to store the results. My platform generates, compiles, executes and measures test results for any specified number of test ranges with dynamically generated input parameters.

Tests are run on the single machine my platform runs on, which is a dedicated server with a 8 core, 16 thread Intel Xeon E5-2650v2 CPU, 128GB RAM and an Intel SSD. Before starting any new test, the system checks the 1-minute server load average and only starts a test if it is less than 1.

As this server is not loaded by very much at the moment, while testing I found that this load average was only ever above 1 when a test was running. As such, I am confident that the test results I recorded are not affected by any other load or server bottlenecks.

Since my testing platform stores the output from each program in the database it is easy to check the resulting Sum of Totients is actually correct (e.g. compare with the result from the sequential code). This helped me find logical flaws in my algorithm early on, and helps ensure every test result is accurate.

As for the actual totient calculation code, I initially found it fairly easy to adapt my C to SAC but I found it to be really slow - slower than my normal C code! After playing around with it for ages and profiling smaller parts of it individually with a friend, I eventually realised it was the fmod method I was using from the Math library which was slowing it down. I wrote my own version of this function rather than using the library version and suddenly it was perfect!

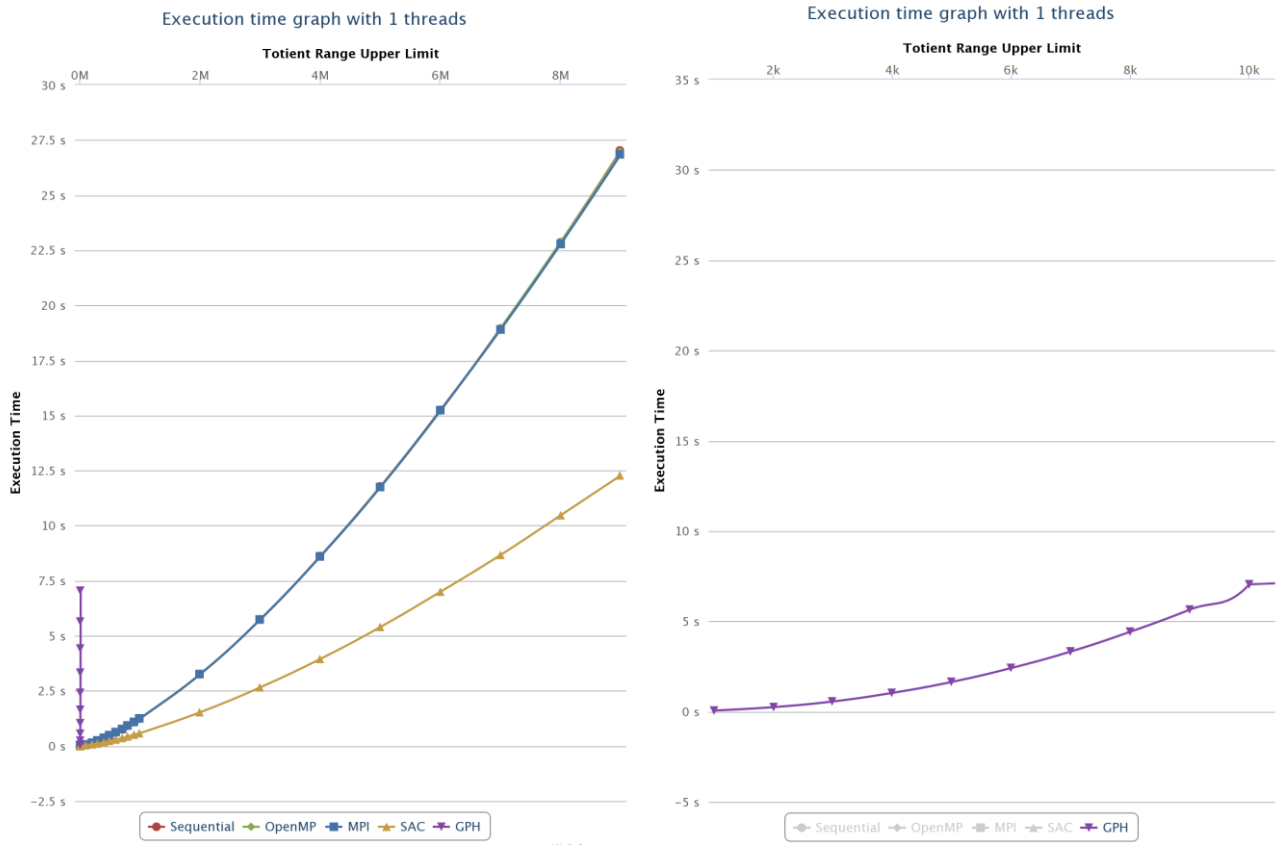
Haskell was… less easy. I found it very difficult to get my head around it, despite reading and trying to adapt the code from this paper (http://www.dcs.gla.ac.uk/~trinder/papers/TPDS.pdf). Eventually I did manage to get a version working, using parListChunk and rdeepseq, but despite being successfully parallelised it is massively slower than the C code I wrote, and I don’t really understand why.

Sequential Performance Measurements

As you can see from the diagram below (left), the line representing my original C sequential code is hidden beneath the results for the OpenMP and MPI code, both of which match the run time exactly when run with only one thread. However, even when run with only 1 thread, my SAC port is clearly faster than my plain C, despite using the same algorithm.

As for Haskell, with an upper limit of 100000 it takes 15 minutes to execute, so I have never tested it with a greater range than that. Given the large performance gap, there isn't much purpose in showing it on the same graph as the more efficient code results.

However, when displayed on its own zoomed in to the relevant results, it displays the expected characteristics of a sequentially increasing sum, and it is worth noting the time it takes to run when calculating the sum of totients up to 10K: 7 seconds.



To view these execution time results/graphs in an interactive way, visit http://parallel.techfixuk.com/ in any modern browser. The first graph on that page shows the test results for one thread (sequential), and you can click/drag to zoom to specific areas, or hide/show specific data series by clicking the label.

Comparative Parallel Performance Runtime Measurements

Again like my last submission, the exact measurement requests of DS1/2/3 are no longer relevant to me due to my drastically improved sequential algorithm and much larger scale of testing.

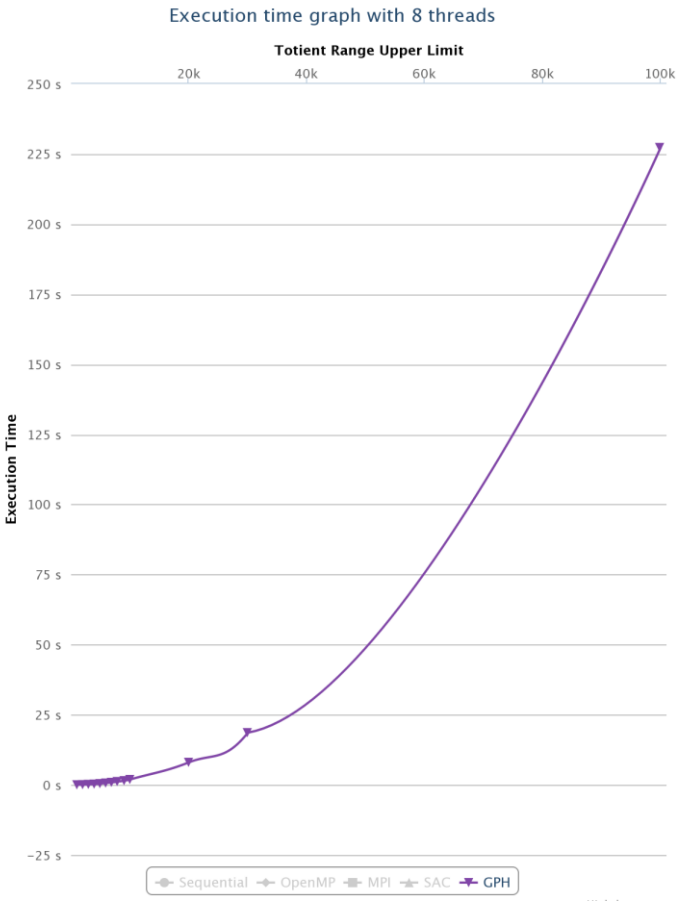
I have built and executed over 7000 unique tests. For all specified values of the upperLimit parameter between 1000 and 9000000, I executed a test for every combination of the 5 test types (C sequential, OpenMP, MPI, SAC, Haskell) and all numbers of threads from 1 to 16. Then that tests were executed at least 3 times to allow for a useful median value to help eliminate outliers caused by external factors.

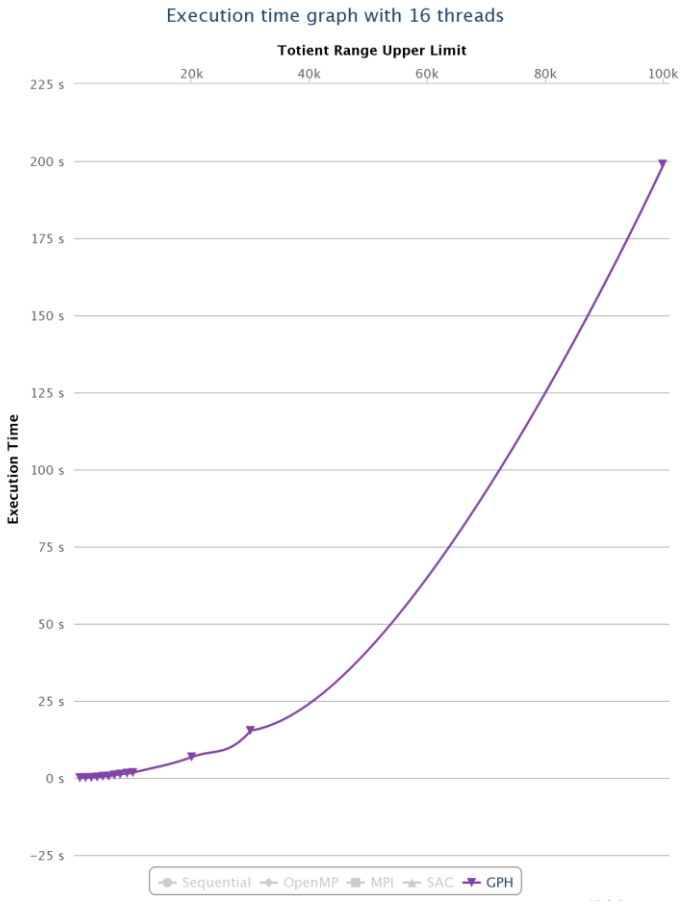
Hopefully all of the following graphs are easy to follow; every plotted point is a median value of at least three results in the database. Out of curiosity I plotted the same graphs using mean values instead and there was very little difference, however if the tests were being performed on a less closely controlled system with greater fluctuation in system load a median value would definitely be preferred as it would be less likely to be skewed by a single unusually large or small value.

Firstly I'm going to show a couple of parallel results for Haskell on its own. As mentioned before it is not useful to plot it on the same graph as the efficient programs as there is such a difference in scale.

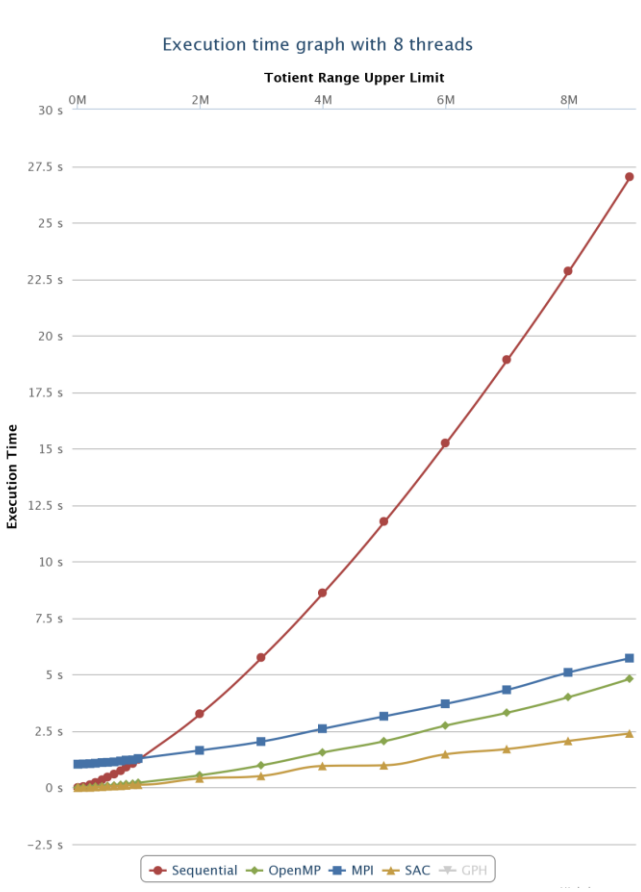
Yes, that is 200 seconds. Yes, it was actually using all 16 cores at 100% for that time.

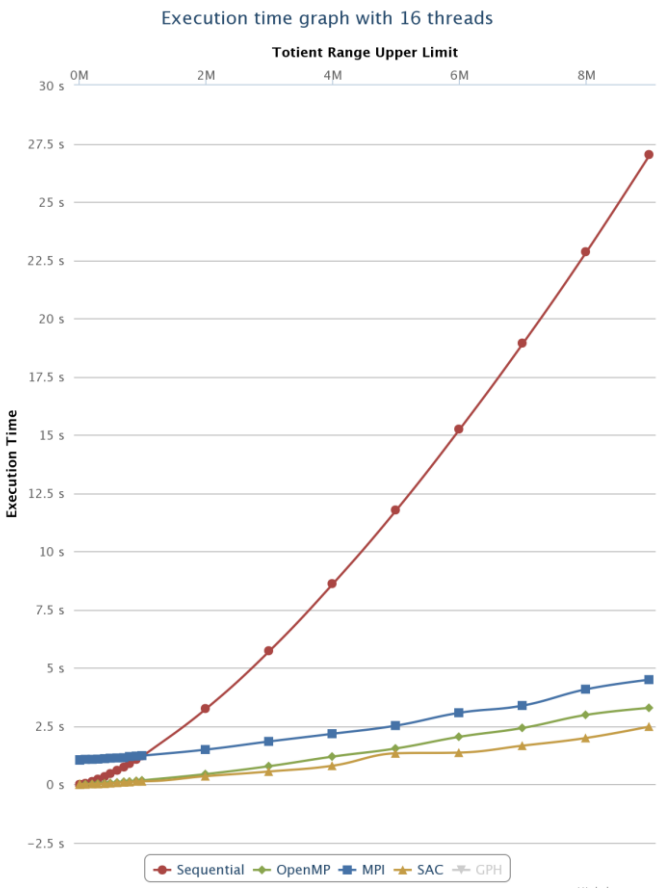
No, I don't know why it is so bad.



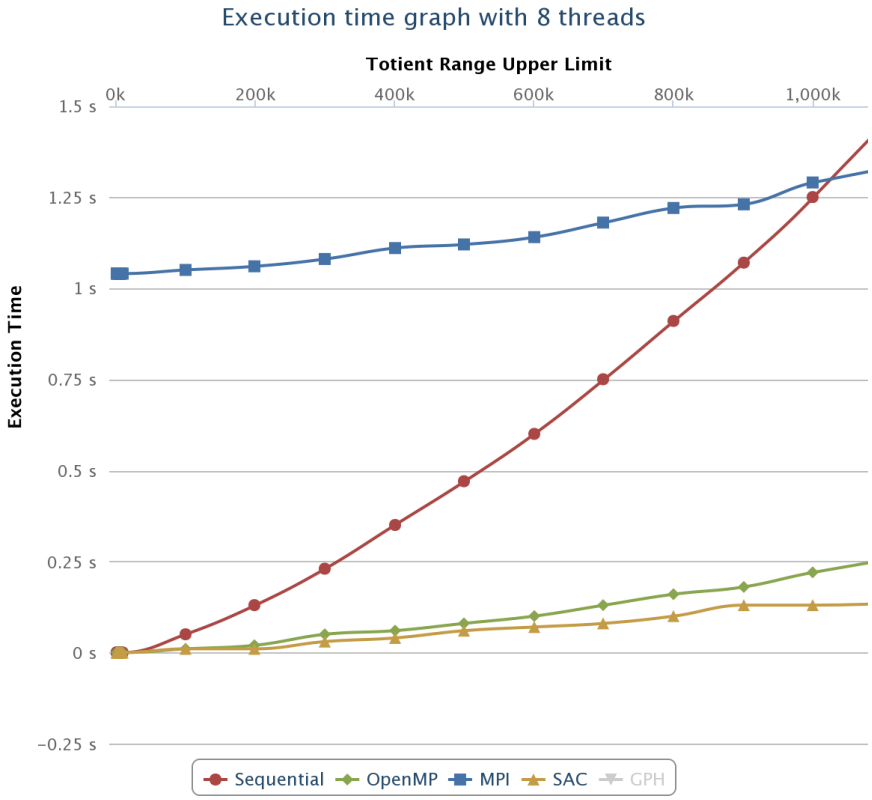


Now for some much nicer execution time diagrams, here are the results for the other programs, including SAC.

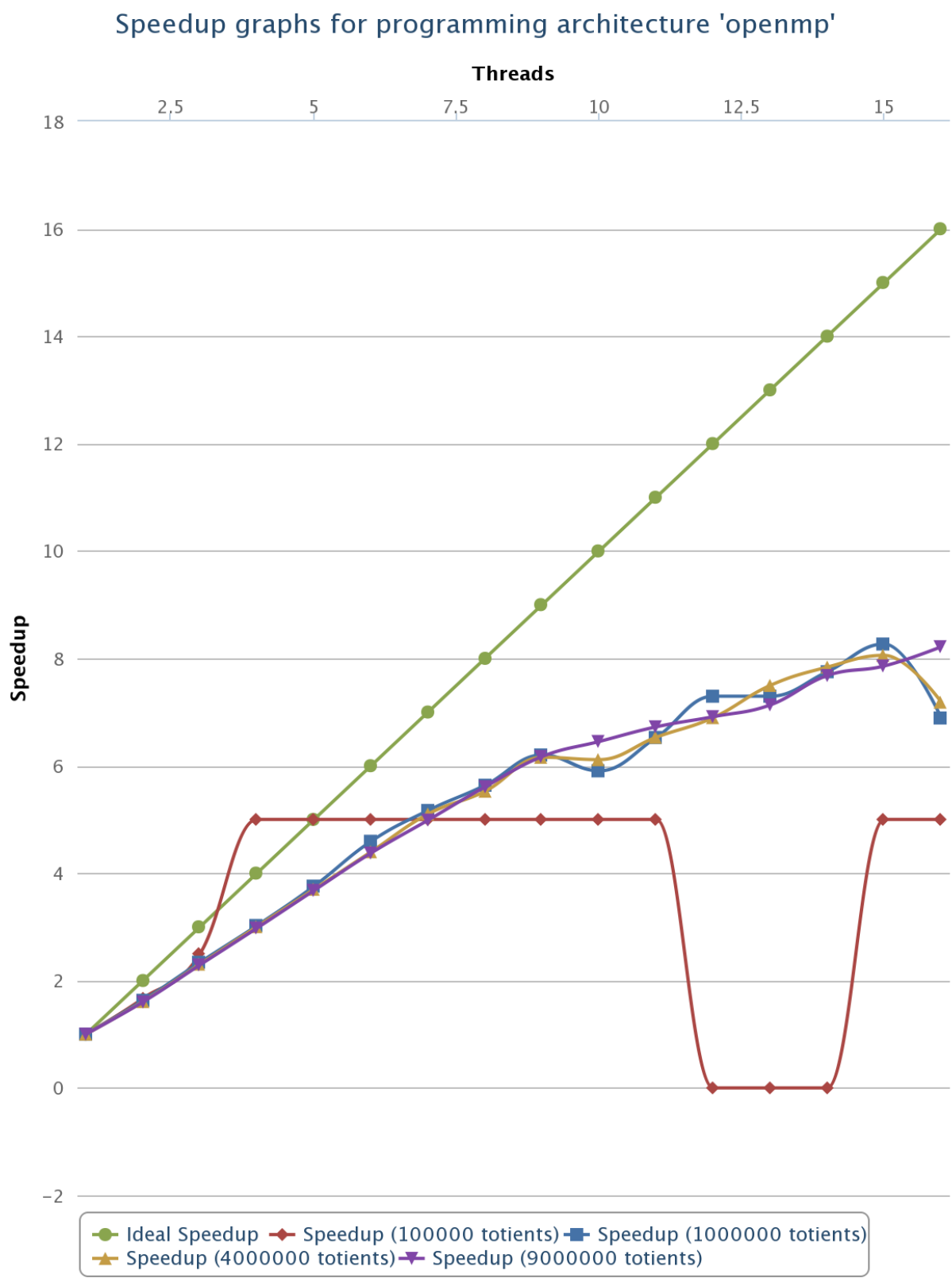




As you can clearly see, when dealing with inputs of less than 1 million, MPI has too much overhead to be useful as it is actually slower than sequential code. SAC, however does an even better job than OpenMP at every step of the way, executing quicker even when given small input values such as 300K. The graph below, which is the same as above but zoomed to the results less than 1 million, shows this better.

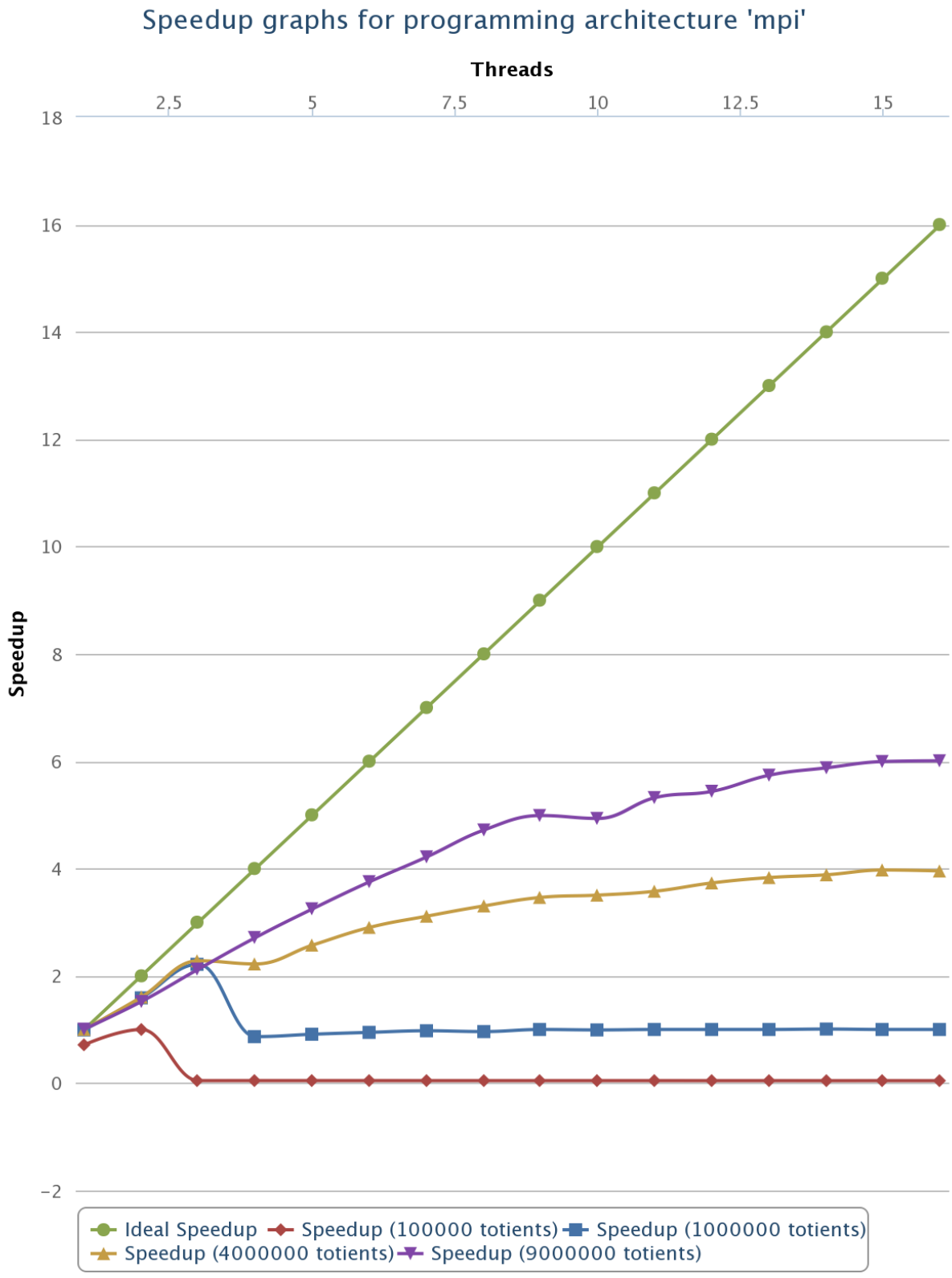


Speedups

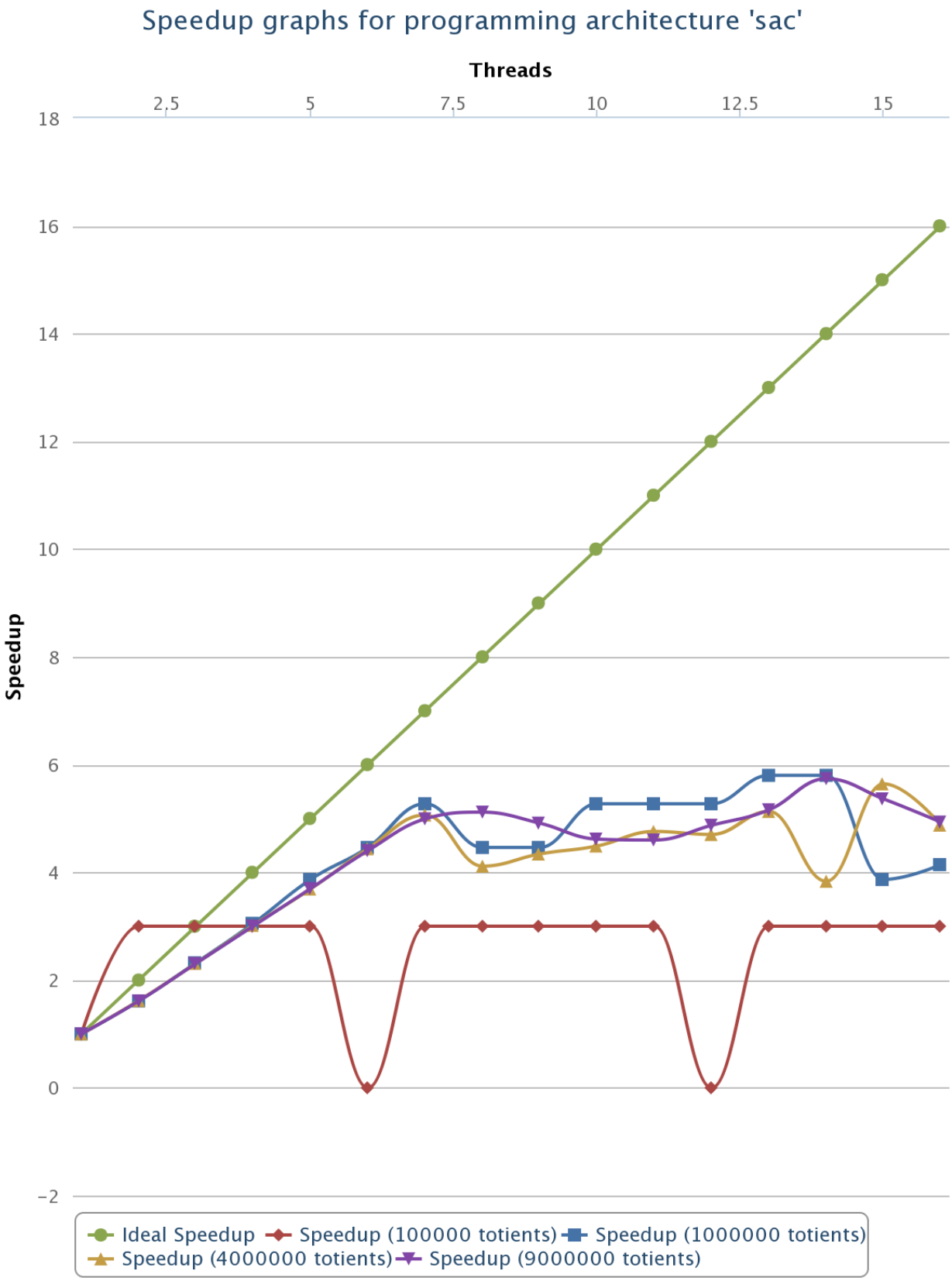


As you can see OpenMP only starts being stable with input values over 100000. I omitted the speedup line for tests with ranges less than 100000, because they almost all execute in milliseconds and don't provide a useful speedup line. With similar results from the next few graphs I don't read too much into this other than; larger/longer calculation ranges increase execution speed stability.

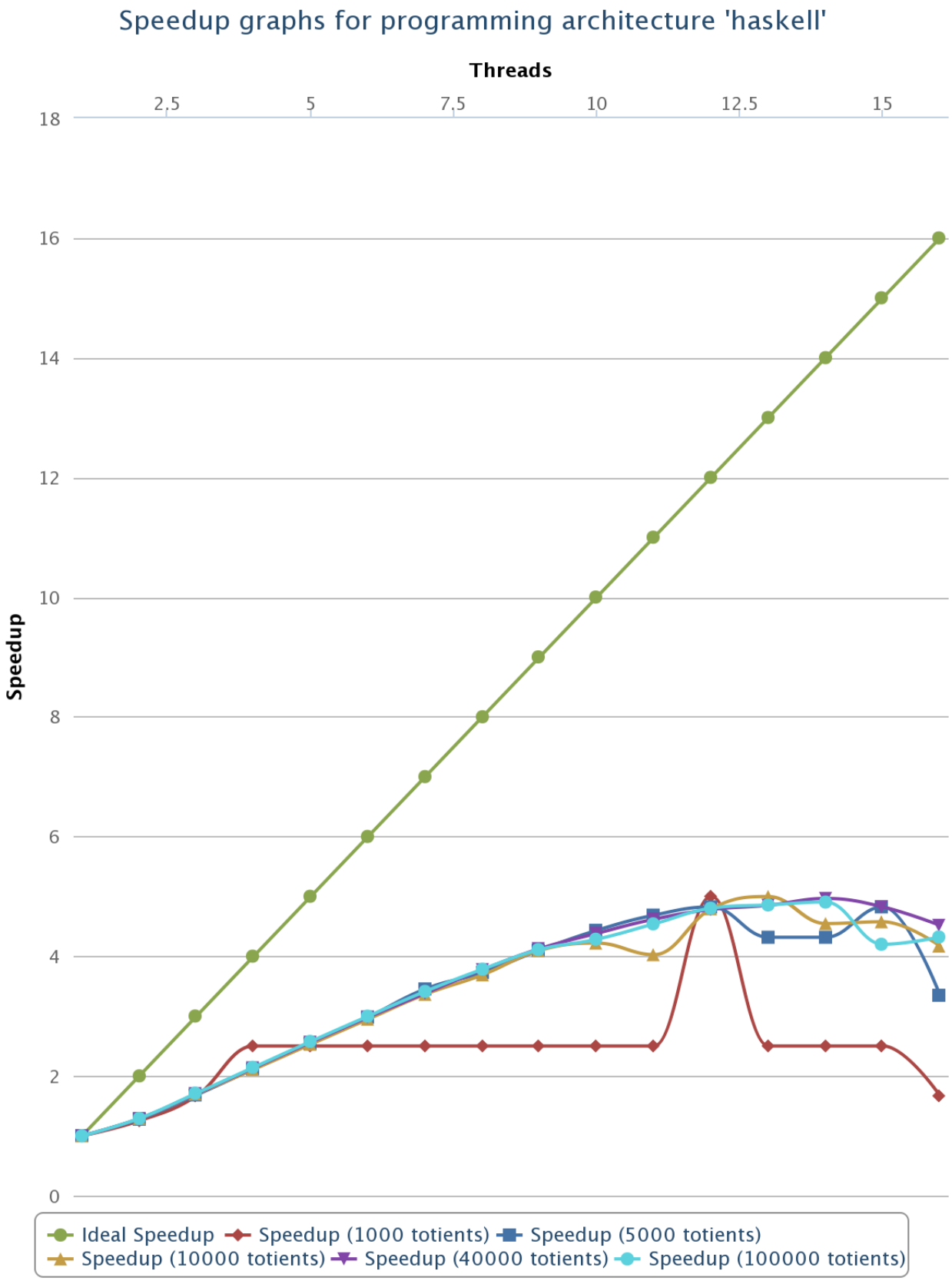
Overall this is a pretty decent speedup, with a peak speedup of 8x around 16 threads.



MPI has an even more distinctive drop-off when given smaller input ranges to process. I suspect what is happening here is that a larger portion of time is being used for overheads; to set up MPI rather than to actually calculate totients. However once you start dealing with larger data it proves to be fairly stable with a peak of 6x.



SAC exhibits the similar issues with smaller input ranges to OpenMP but has very different characteristics from the previous two technologies; it shows a very strong speedup with up to 7 threads (peak of 5.3x) then drops off and becomes quite unpredictable, with a second noticeable drop-off at 16 threads. Perhaps the current development of SAC has mostly tested with machines with 8 or less cores?

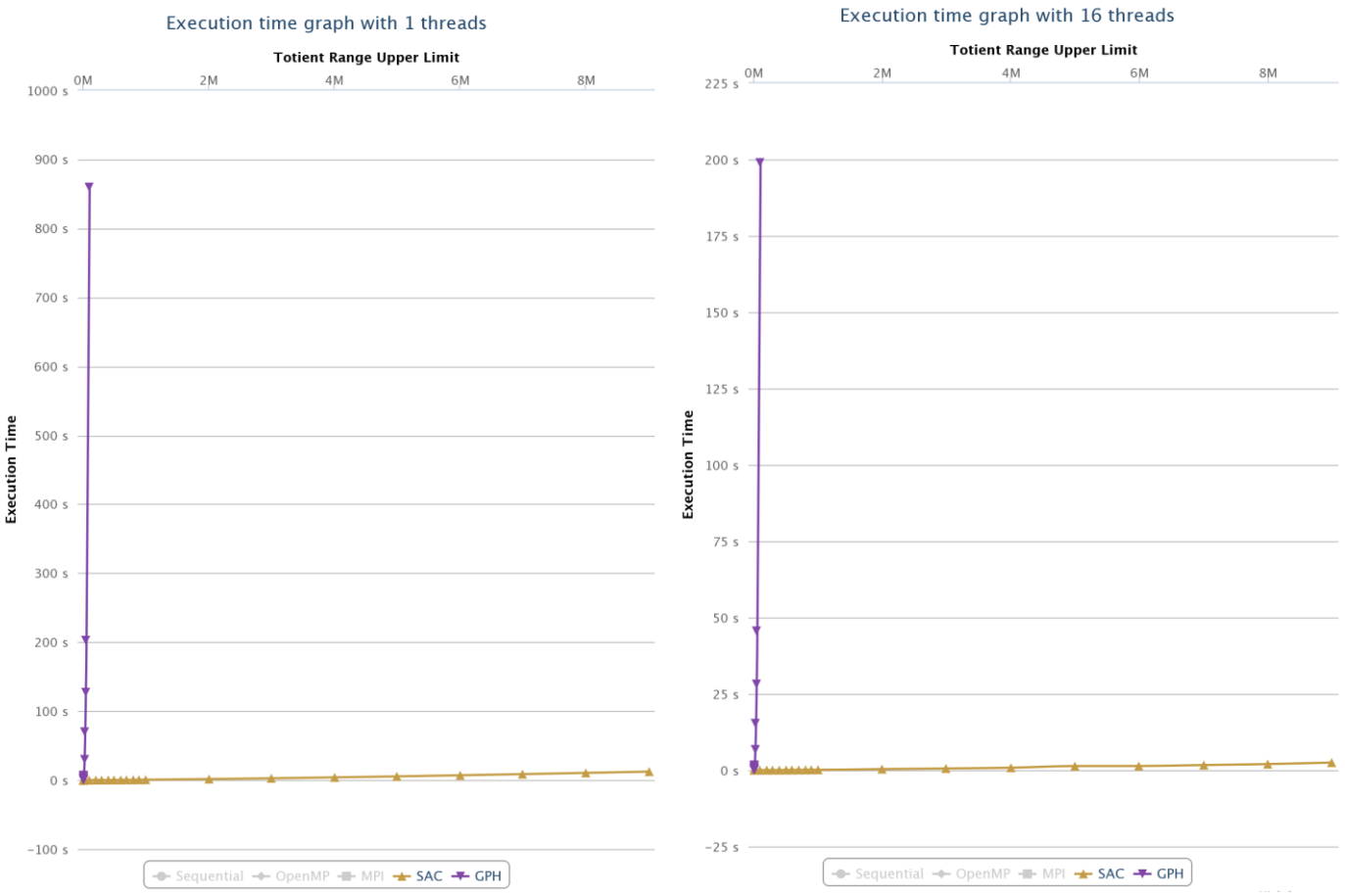


Despite being terribly slow compared to the various C codes above, the Haskell code I hacked together clearly is being executed in parallel. The speedup is fairly poor in comparison, with a peak of 4.9x with 12 cores. Drop-off after that is noticeable, and it is still affected by the same issues with small inputs.

Parallel Performance Summary

Comparing Single Assignment C to Glasgow Parallel Haskell is difficult. Without a thorough understanding of Haskell in the first place (does such a thing even exist?!), I find it very hard to find any insight into why my code runs so slowly.

However, I can provide two summary graphs, one sequential with both plotted, one with 16 threads:



Isn't that just beautifully informative…

If you would like to see the results from my tests in a more useful / interactive form, go to:

Execution Time Graphs: http://parallel.techfixuk.com/

Speedup Graphs: http://parallel.techfixuk.com/speedup

Programming Model Comparison

To work how to start parallelising the sumTotient method in Haskell, I stripped it down and replaced portions with dummy code and timed its execution for each test. From my results I realised most of the time was spent in the hcf function, but I couldn't figure out how to implement the algorithm I implemented in C to replace this function in Haskell. So, I googled a bit, found a couple of Trinder's papers and tried to implement Strategies in the sumTotient method with the parListChunk strategy. After experimenting with chunk sizes I eventually settled on one chunk per thread as this made the most sense and proved to be fastest.

SAC was fairly easy to implement, I sort of just took my existing C code and switched a few things for with-loops, changed a whole load of longs to doubles, used fmod instead of the modulus symbol. The most difficult part was tracking down why it was running so slowly, and that turned out to be a poorly written fmod function in the Math library, which I replaced with my own and found the code to run 10 times faster!

I would suggest that SAC is much more intuitive to people with backgrounds in imperative languages such as C, and while Haskell may be a very powerful language I feel the parallel technologies are not yet very mature. Perhaps I was just doing it very wrong, but given the results I measured I would suggest SAC to be a better choice.

Appendix A: TotientRange.sac

```
// Distributed and parallel technologies, Andrew Beveridge, 03/04/2014
// To Compile:        sac2c -doAWLF -mt -o ab-totient-sac ab-totient-sac.sac
// To Run / Time:    /usr/bin/time -v ./ab-totient-sac range˙start range˙end -mt cores

use StdIO: all;
use Array: all;
use CommandLine: all;

// Function exists already in Math, but seems to be really slow in comparison to this?
inline double[] fmod(double[] a, double[] b)
–
    k = toi(a / b);
    s = a - tod(k) * b;
    return(s);
”

//  When input is a prime number, the totient is simply the prime number - 1. Totient is always even (except for 1).
//  If n is a positive integer, then ϕ(n) is the number of integers k in the range 1 ≤ k ≤ n for which gcd(n, k) = 1
inline double getTotient (double number) –
     result = number;

    // Check every prime number below the square root for divisibility
    if( fmod(number, 2d) == 0d )–
        result -= result / 2d;
        do
            number /= 2d;
        while( fmod(number, 2d) == 0d );
    ”

    // Primitive replacement for a list of primes, looping through every odd number
     for(prime = 3d; prime * prime ¡= number; prime += 2d)–
        if( fmod(number, prime) == 0d )–
```

```
            result -= result / prime;
            do
                number /= prime;
            while( fmod(number, prime) == 0d );
        ″
    ″

    // Last common factor
    if(number ¿ 1d)
        result -= result / number;

    // Return the result.
    return( result );
″

// Main method.
int main() –
    // Load inputs
    lower = String::toi( argv(1) );
    upper = String::toi( argv(2) );

    // Sum all totients in the specified range
    result = with –
        ( [0] ¡= iv ¡= [upper - lower]) : getTotient( tod(iv[0] + lower) );
    ″ : fold(+, 0d);

    // Print the result
    printf("Sum of Totients between [%d..%d] is %.f "n", lower, upper, result);

    // A-OK!
    return(0);
″
```

Appendix B: TotientRange.hs

```
-- Distributed and parallel technologies, Andrew Beveridge, 03/04/2014
-- To Compile:        ghc -Wall --make -threaded -cpp -O2 -rtsopts -o ab-totient-gph ab-totient-gph.hs
-- To Run / Time:     /usr/bin/time -v ./ab-totient-gph [range˙start] [range˙end] [threads] +RTS -N[threads]

module Main(main) where

import System.Environment
import System.IO
import Control.Parallel.Strategies

-- The main function, sumTotient
-- 1. Generates a list of integers between min and max
-- 2. Applies totient function to every element of the list
-- 3. Returns the sum of the results

sumTotient :: Int -¿ Int -¿ Int -¿ Int
sumTotient min max threads = sum (
        (map euler ([min, min+1 .. max])
            `using` parListChunk (((max-min) `div` threads)+1) rdeepseq))

-- 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])

-- 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

-- 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
            min = read (args!!0) :: Int -- min limit of the interval
            max = read (args!!1) :: Int -- max limit of the interval
            threads = read (args!!2) :: Int -- Number of threads to be run on
          hPutStrLn stderr ("Sum of Totients between [" ++
                    (show min) ++ ".." ++ (show max) ++ "] is " ++
                     show (sumTotient min max threads) )
```