Distributed and Parallel Technology
Assessed Coursework Two
Evaluating High-level Parallel Programming models
F21DP

Boris Mocialov (H00180016), Max Baird (H00153951)
MSc Software Engineering

Heriot-Watt University, Edinburgh

Tuesday 31, 2015
List of Figures

1. Haskell heap consumption for [1 - 15000] dataset ................................. 6
2. SaC vs Haskell Absolute Speedups ....................................................... 7
3. SaC vs Haskell [1 - 15000] ..................................................................... 8
4. SaC vs Haskell [1 - 30000] ................................................................. 8
5. SaC vs Haskell [1 - 100000] ............................................................... 9
6. Speedup SaC vs Haskell [1 - 15000] ..................................................... 10
7. Speedup SaC vs Haskell [1 - 30000] ..................................................... 10
8. Speedup SaC vs Haskell [1 - 100000] ................................................. 11
9. Haskell granularity effect on the performance ..................................... 14
10. Load Balancing example for SAC Implementation ............................. 17
11. Load Balancing for Haskell Implementation with chunk size of 200 elements .......................... 19
12. Execution of parallel implementation using 32 threads on [1 - 15000] dataset 64 core machine 20
List of Tables

1. Hardware Execution Details ..................................................... 3
2. Sequential C profiling with largest dataset .................................. 4
3. Best Sequential & Parallel Runtimes .......................................... 12
1 Introduction

This report describes the work performed to accomplish the task of developing and measuring performance of the parallel version of Euler’s totient function. The task shall be approached using two different technologies - Single Assignment C (SaC) and Glasgow parallel Haskell (GpH). Both technologies can provide parallel implementations of a program. For discussion, both technologies will be compared in terms of performance and underlying programming models. The objective of the described task is to gain appreciation for different available technologies and parallel implementation paradigms (both conceptual and practical) and to gain an understanding of the importance of the parallelism in software development.

The report will begin with giving a description to the employed technologies and the environment. Later, implemented parallel execution runtimes will be presented in graphs to give an overview over the development of the relative runtime. Speedup graphs are there to show how the execution proceeds with respect to the number of threads utilised. A table summarising the best performances of both technologies over different datasets from sequential and parallel executions will be presented and discussed, giving some speculative opinions of different aspects of the presented graphs and visible traits. At the end, both used programming models will be compared and discussed. Implementation details are available in the appendix of this report.

1.a Description of Technologies

1.a.1 Single Assignment C

Single Assignment C (SAC) is defined as a functional array language for efficient multi-threaded execution. SAC aims to combine high level array programming with fully automatic resource management so that programmers can work with a simple abstraction of underlying complex data structures and without concern of low-level hardware details. This enables high productivity (since less time needs to be spent on optimizing for concurrency) and maintenance.

A positive side effect of SAC’s aims is that it tends to lessen the divide between following good software engineering principles and developing efficient parallel code. What serves as key in achieving SAC’s aim and this side effect is that the language incorporates optimisation and parallelisation into the language design. This allows for a more “traditional” (sequential-like) programming experience since parallel features are not “tacked-on” as an afterthought.

SAC is adopted from the C programming language which is a fairly well known and standard language that facilitates a more natural transition for newcomers to the language. The high-level, implicitly parallel language features of SAC achieves encouraging runtime behavior because the compiler does code restructuring into a seemingly obfuscated representation (to humans) that is quite machine friendly. However, SAC is not the single answer for all high-level parallel programming tasks, its single array data structure means that the design favors array-intensive tasks which can be executed in a data-parallel manner as was the case for this programming task.

The version of Single Assignment C used for the parallel implementation is as follows:
1.a.2 Glasgow parallel Haskell

Glasgow parallel Haskell (GpH) is a parallel extension to a traditional Haskell programming language that allows parallel programs to be executed both on clusters and on multi-core machines.

GpH requires programmer only to annotate existing sequential code (i.e. `x 'par' y`) to spark the evaluation of `x`. Sparks are further queued in FIFO order and, during runtime, may be converted into threads if scheduler detect an idle CPU. This constitutes the parallelism paradigm in the GpH. The paradigm allows programmer to do only few changes to the sequential code and let the language implementation manage the rest.

Sparked evaluation should be relatively heavy so that no unnecessary overheads will be introduced during the execution. Parallel coordination of execution is done using two primitives: `par` and `seq` of which the former, as mentioned earlier, schedules a parallel evaluation. In case of the latter, this primitive is used to control the execution order.

The language provides abstracts over the `par` and `seq` primitives, which are called evaluation strategies. A strategy specifies the course of the execution of a particular function, distinguishing between computation and coordination of tasks.

The version of Haskell used for the parallel implementation is as follows:

```text
The Glorious Glasgow Haskell Compilation System, version 7.6.3
```
1.b Execution Environment Details

A summary of the details of the execution environment are as follows:

<table>
<thead>
<tr>
<th>Architecture</th>
<th>x86_64</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU op-mode(s)</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Byte Order</td>
<td>Little Endian</td>
</tr>
<tr>
<td>CPU(s)</td>
<td>64</td>
</tr>
<tr>
<td>On-line CPU(s) list</td>
<td>0-63</td>
</tr>
<tr>
<td>Thread(s) per core</td>
<td>2</td>
</tr>
<tr>
<td>Core(s) per socket</td>
<td>8</td>
</tr>
<tr>
<td>Socket(s)</td>
<td>4</td>
</tr>
<tr>
<td>NUMA node(s)</td>
<td>8</td>
</tr>
<tr>
<td>Vendor ID</td>
<td>AuthenticAMD</td>
</tr>
<tr>
<td>CPU family</td>
<td>21</td>
</tr>
<tr>
<td>Model</td>
<td>2</td>
</tr>
<tr>
<td>Stepping</td>
<td>0</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>1400.000</td>
</tr>
<tr>
<td>BogoMIPS</td>
<td>4999.30</td>
</tr>
<tr>
<td>Virtualization</td>
<td>AMD-V</td>
</tr>
<tr>
<td>L1d cache</td>
<td>16K</td>
</tr>
<tr>
<td>L1i cache</td>
<td>64K</td>
</tr>
<tr>
<td>L2 cache</td>
<td>2048K</td>
</tr>
<tr>
<td>L3 cache</td>
<td>6144K</td>
</tr>
<tr>
<td>NUMA node0 CPU(s)</td>
<td>0-7</td>
</tr>
<tr>
<td>NUMA node1 CPU(s)</td>
<td>8-15</td>
</tr>
<tr>
<td>NUMA node2 CPU(s)</td>
<td>16-23</td>
</tr>
<tr>
<td>NUMA node3 CPU(s)</td>
<td>24-31</td>
</tr>
<tr>
<td>NUMA node4 CPU(s)</td>
<td>31-39</td>
</tr>
<tr>
<td>NUMA node5 CPU(s)</td>
<td>40-47</td>
</tr>
<tr>
<td>NUMA node6 CPU(s)</td>
<td>48-55</td>
</tr>
<tr>
<td>NUMA node7 CPU(s)</td>
<td>56-63</td>
</tr>
</tbody>
</table>

Table 1: Hardware Execution Details

It is important to note that all tests and experiments were executed on the above platform in order to ensure consistent comparisons. Additional hardware details of the execution environment are provided in table 1. Finally, before each execution the load of the machine was verified to be light via the top command.
2 Sequential Performance Measurements

This section presents the sequential analysis carried out for the sum of Euler totient computations as ported to Haskell and SAC. There is also a discussion of the sequential performance and its relation to similar implementations with previous technologies; OpenCL and C+MPI. Each subsection concludes with suggestions of possible improvements to these programs i.e. the sequential Haskell and SAC implementations.

2.a Sequential Performance Measurements - SAC

The sequential implementation of the totient program ported to SAC resulted in an automatic speedup of approximately 50%. While there are currently no profiling tools for SAC, it was assumed that the automatic speedup is an attribute of the code generation performed by the SAC compiler which is understood to be highly efficient. However while this may have been so there was still room for improvement. The default execution being performed at this point did not take advantage of any of the language features offered by SAC and was compiled without multithreading.

Previous analysis on the provided sequential code using GNU Profiler identified the \texttt{hcf} and \texttt{relprime} functions to be the most called functions across executions (with over 90% of execution time spent in \texttt{hcf}). These functions were therefore a good candidates for parallel optimizations. The table below shows the profiling information for the provided sequential C program. While default speedups were observed in SAC with an identical port, it is believed that the profiling of its sequential C counterpart would provide good \textit{hints} for areas of improvement since it is essentially the same algorithm. Being an identical port, though automatically faster, theoretically the same bottlenecks should occur for the single threaded SAC implementation.

<table>
<thead>
<tr>
<th>% Time</th>
<th>Cumulative Seconds</th>
<th>Self Seconds</th>
<th>Calls</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>92.22</td>
<td>433.64</td>
<td>433.64</td>
<td>705082704</td>
<td>hcf</td>
</tr>
<tr>
<td>6.10</td>
<td>462.32</td>
<td>28.68</td>
<td>100000</td>
<td>euler</td>
</tr>
<tr>
<td>2.16</td>
<td>472.46</td>
<td>10.15</td>
<td>75082704</td>
<td>relprime</td>
</tr>
<tr>
<td>0.30</td>
<td>473.87</td>
<td>1.40</td>
<td></td>
<td>frame_dummy</td>
</tr>
<tr>
<td>0.00</td>
<td>473.87</td>
<td>0.00</td>
<td>1</td>
<td>sumTotient</td>
</tr>
</tbody>
</table>

Table 2: Sequential C profiling with largest dataset

As depicted in table 2, the \texttt{hcf} and \texttt{relprime} functions were called the same amount of times with the majority of the execution time being spent within the \texttt{hcf}.

CacheGrind was also used as an analysis tool to identify cache misses, the data and cache misses were found to be insignificant (less than 0.000%) on the sequential C. However, because the SAC compiler generates C code, the profiling information derived from this analysis was not not considered since it remains uncertain how efficient the generated code would be with respect to the hardware architecture.

Based on the analysis performed on providing sequential code, it was (hopefully correctly) assumed that the nearly identical SAC implementation would have the same performance characteristics, in the context
of function calls, which therefore meant parallelization of the hcf function would improve execution times.

Comparison with the previous sequential port to OpenCL cannot be done in a straight forward manner since GP-GPU programming uses multiple cores by default. It remains a subject for further experimentation, but it is expected implementing a purely sequential OpenCL implementation would be several orders slower than its SAC counterpart. The expected slow down would have to do with such a forced sequential execution not fitting the Single Instruction Multiple Threads (SIMT) model of OpenCL. Another contributing factor which is expected to introduce additional overhead would be the required shipping of the data back and forth across the bus to send data to the OpenCL device and then back to the host.

2.b Sequential Performance Measurements - Haskell

Investigation of a sequential Haskell implementation performance had been done by analysing both time spent in each function of the program and memory allocations made on the heap during the execution. To allow for such investigations, a sequential program had been compiled for profiling using 

```
-prof -auto-all -rtsopts
```

options and later ran using 

```
+RTS -pT -hC -stderr
```

options.

After the program finished execution, following files were generated:

- TotientRange.hp (heap profiling information)
- TotientRange.prof (runtime profiling information)

In addition to generated files, summary is presented that tells how much time had been spent running the program as well as the time spent on garbage collection.

Generally, the productivity (MUT time out of total time, without GC) of sequential execution is >90%.

As for the heap allocations, the sequential program allocates all the memory at the beginning of the execution and then gradually descends. The pattern, that is consistent over all datasets, can be seen on the Figure 1.

When it comes to execution of particular functions, .prof file shows partitioning of the overall time per function. For the sequential program executed to find the sum of Euler totient from 1 to 100000, partitioning is as follows (time partitioning remains consistent over datasets, but the allocations partitioning is different):

- sumTotient: time - 0.0%; alloc - 0.0%
- hcf: time - 83.5%; alloc - 75.3%
- euler: time - 10.5%; alloc - 19.3%
- relprime: time - 6.0%; alloc - 5.4%

Using this information it can be inferred that the most of the runtime was spent inside hcf function, while most of the allocations had been done inside euler function.
The graph presented above looks strange as it implies that the heap consumption is gradually becoming less as garbage collection collects used data after the computation had finished. That said, the graph was expected to be linear and constant as the time goes by, but turns out this is not the case.

**Potential parallelism**

The `hcf` function cannot be tackled in divide and conquer approach due to dependency of values. The `euler` function, on the other hand, generates a list of independent values that can be passed to `relprime` in parallel. The `sumTotient` can also pass every element of a list in parallel to the `euler` function.

**Comparison to C+MPI**

Sequential runtime of the program, written in Haskell is textbf5 times slower on average than the one written in C. One possible explanation to that is that although Haskell code is being compiled, C has lesser overhead when compiling program to the machine code.
3 Comparative Parallel Performance Measurements

A comparison of the best runtime measurements for the parallel approaches is depicted in Figure 2 where the numerical values represent the median of three executions. The values show the absolute speedups of each technology for each dataset, this means irrespective of the number of cores used. The time taken is measured in seconds. In this case, the parallel SAC implementation outperforms its parallel Haskell counterpart for all datasets.

![Figure 2: SaC vs Haskell Absolute Speedups](image)
3.a Runtime Graphs

Figure 3: SaC vs Haskell [1 - 15000]

Figure 4: SaC vs Haskell [1 - 30000]
Presented above graphs show relative runtimes of parallel implementations of both technologies using three different datasets. Measurements were done 3 times and the mean value taken to represent a particular runtime. Runtimes were measured running parallel implementation using 1-256 threads on a machine that has 64 cores.

The general runtime development tells that the more threads are used by a program, the faster execution becomes. What is worth noting is that in all cases, between 32 and 64 threads, execution slows down. The slow-down is different for both technologies. The slow-down is explained by the necessity of a scheduler that must come into the picture and distribute the workload between the CPUs.
3.b Speedups

Figure 6: Speedup SaC vs Haskell [1 - 15000]

Figure 7: Speedup SaC vs Haskell [1 - 30000]
Figure 8: Speedup SaC vs Haskell [1 - 100000]

Presented above graphs show relative speedups, which is a runtime of a run divided by the runtime on a single core for the same dataset. It can be seen that the relative speedup is similar for both technologies. The only difference emerges when using more than 32 threads, which can again be explained by the scheduler.
3.c Sequential Performance & Best Parallel Runtimes

Table 3 is a depiction of the sequential and associated parallel runtime for each technology for every dataset measured using best runtimes out of three executions.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Sequential SAC</th>
<th>Parallel SAC</th>
<th>Sequential Haskell</th>
<th>Parallel Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1 - 15000]</td>
<td>5.923</td>
<td>0.436</td>
<td>54.546</td>
<td>0.517</td>
</tr>
<tr>
<td>[1 - 30000]</td>
<td>20.319</td>
<td>0.921</td>
<td>230.597</td>
<td>1.808</td>
</tr>
</tbody>
</table>

Table 3: Best Sequential & Parallel Runtimes

3.d Discussion of Comparative Performance

In this section we discuss the comparative performance of both technologies in their application to the sum of Euler totients. The absolute speed ups achieved for each implementation is a measure of the median time taken to arrive at a solution for each dataset irrespective of number threads used. To repeat, for clarity, tests for each dataset for both implementations were executed on identical hardware.

In calculating the sum Totients for the first dataset (1 - 15000) both implementations achieved quite similar execution times (reference Figure 2 in section 3) with a difference of 0.144 seconds. In this case, the SAC implementation marginally outperformed its Haskell counterpart. For the second dataset (1 - 30000) the difference between the two implementations became more significant with a difference of 1.366 seconds. Again, the SAC implementation resulted in better performance. The trend of SAC’s better performance is shown to be most significant upon inspection of the results when computing sum totients for the range 1 - 100000, with a difference of 12.043 seconds. Theories on these performance differences will now follow with references to the runtime and speedup graphs.

In defence of Haskell, the relative runtime improvements for sequential implementation and the parallel one are considerable - parallel implementation improved more than 100 times with the smallest dataset....

The three runtime graphs presented in section 3.a are graphs of time taken by each implementation for each dataset using increasing numbers of threads. The trend depicted across all graphs shows each implementation performing with similar execution speeds for matching numbers of threads. Peak efficiency, for each dataset, occurs around the same number of treads, however, both implementations simultaneously depict diminishing returns (at different rates) beyond this point. These observations are more solidly confirmed through an examination of the relative speed up graphs in section 3.b.

Our theory in understanding why diminishing returns occurred when they did has a bearing on the underlying hardware architecture. The table 1 in section 1.b illustrates the hardware environment details. Of particular interest is the processor, which has a 64 cores. We theorize that a number of threads specified which is less than, or matches, the number of physical cores yields good performance since each thread can be mapped directly to a core. If the system is lightly loaded the operating system will only need to carry out minimal scheduling.
Speedups begin to diminish with thread amounts closer to the core limit since it is expected that the operating system is required to perform more scheduling to facilitate its background processes. What this means is additional threads need to be suspended and resumed which introduces the overhead. Thread numbers greater than 64 really show a performance breakdown as the operating system has no choice but to perform scheduling. The Haskell implementation shows a much greater loss of efficiency when compared to the SAC implementation beyond 64 threads. This may be attributed to characteristics inherent to the method used to approach the solution. For SAC, most code is generated to be machine friendly in a parallel fashion. SAC’s parallel code-generation approach requires less explicit parallelization from the programmer, this may be the reason its automatically generated code scales better.

An execution characteristic, which may not be immediately obvious from the graphs, that is worthy of mentioning is that for the smallest dataset (1 - 15000) inefficiency started taking place before the number of specified threads matched then number of cores for both algorithms. Our reasoning for this characteristic is that the time taken to set up the threads and partition the work negatively affects performance. In essence, the time taken to create each thread is more than the time needed to perform their jobs. It is expected that the remaining datasets would exhibit similar characteristics in the presence of more cores, however, this remains to be tested since other unanticipated characteristics may emerge which can prove this expectation untrue.

4 Programming Model Comparison

4.a SAC

As described in section 1.a.1, SAC is functional array programming language. In relation to the TotientRange application, SAC’s data-parallel model mapped well to the characteristics of the application. The characteristic of the TotientApplication which facilitated the data-parallel approach had to do with the limited to no data dependency between calculations. Thus, taking a data-parallel style came with advantages such as less error prone indexing and better maintenance, but most importantly was the huge exposure of concurrency.

The difficulties encountered were minimal and had little to do with the actual data-parallel approach and more to do with understanding SAC’s constructs which facilitate parallelism as indicated by blog post made on Wednesday, 25 March 2015. Parallelism was attempted at a high level by simply mapping each euler calculation to each value in the dataset’s range. This level of parallelism is considered high because prior profiling indicated that the real bottle-neck (over 90% of time spent) occurred through repeated calls to the \texttt{hcf} function.

More technically, use of the with-loop as provided by SAC, was the source of concurrency. The with-loop construct enables a mapping of a function to all elements in a vector or matrix in a data-parallel fashion. Since the Euler values followed a linear range (1,2,3,...,N), the values were generated “on-the-fly” via the generation expression block supported by the with-loop. It was anticipated that pre-generating the array elements would have introduced some overhead, this remains a subject for further experimentation however. On the subject of generating, load-balancing was also done “on-the-fly” during generation. More on
the approach to load balancing used for this technology can be found in the Appendix.

4.b Haskell

As it had been discussed in Section 2.b, two possible choices exist in the sequential implementation that would allow for parallel execution. The second choice had been taken for consideration, namely, to make \texttt{euler} execution parallel.

To make \texttt{euler} parallel, it had been decided to employ a common pattern - load balancing (blog post per Tuesday, 24 March 2015 11:50:41). In load balancing, the total range is split into corresponding chunks of ranges and ranges are distributed across the chunks in such a fashion as to make the load (either amount of items per chunk, the sum of elements, etc.) even for every chunk.

A common issue of using load balancing approach is to choose an optimal chunk size based on some quantification method. For this particular task, a chunk size of 200 elements had been chosen after a number of runs using various chunk sizes. A graph of how granularity affects performance is given in Figure 9. From the graph it can be seen that the fastest execution is achieved when using 200 elements/chunk. It should be noted, however, that the affect is nearly trivial even for computation of the dataset from 1 to 100000 elements.

![Figure 9: Haskell granularity effect on the performance](image)

A further parallelisation attempt had been made to evaluate every list element of \texttt{euler} function (in addition to the \texttt{sumTotient} function) in parallel. As per (Tuesday, 31 March 2015) blog post, such an attempt resulted in rather unexpected result - the performance dropped down considerably. This can be explained by the overhead introduced with too high partitioning, where every partition has too little work load to do.

No particular challenges were encountered during the course of parallelisation of the given sequential program. Since the same program had previously been attempted to be parallelised by the author using different technologies, the approach was straightforward biased highly by the previous attempts.

Since Haskell is purely-functional, lazy language, programming paradigm is different from conventional imperative programming languages. Programming in functional languages requires thinking in terms of...
recursions. The direct consequence for programming in functional languages is that the final program will be short, concise and easily readable.

It is obvious that heavy-weight commercial software would not be written using functional languages as it would be infeasible. As an alternative, heavy-weight commercial applications, written using imperative sequential programming languages, can make use of parallel solutions for some repetitive functions within those programs. As a simple example, a company that has a mathematical model as a backbone of all of its applications, can re-write that particular model using parallel solutions.

5 Reflection on Programming Models

This section describes the chosen parallel programming model for each technology and discusses their advantages and disadvantages. A discussion detailing issues surrounding performance, programmability and usability is also included and their appropriateness for specific hardware architectures.

5.a Programming Model - SAC

The data parallel paradigm is among the simplest of parallel programming paradigms and is well suited for operations which are *explicitly parallel*. Explicitly parallel, in this context, means that operations to be performed can be done independently where the desired outcome is unaffected. In general, any task which operates on very large arrays of data can submit to parallelization quite easily.

The benefit of programs written using the data parallel model is easy debugging since the logic is essentially sequential. Among other advantages such as not requiring the programmer to explicitly specify communication channels (as this is done by the compiler), it has restrictions in that all algorithms cannot be expressed in a data parallel manner.

Through experimenting with this data parallel solution and measuring its performance with larger input sizes and more threads of execution, it suggests a data parallel model is well suited for multi-core architectures. Where more cores means better performance as threads of execution can be isolated per core. But as seen from experiments, if there is not enough work per thread creating too many can result in a slowdown. The point was made to illustrate that simply having many physical cores does not guarantee an automatic speed up if there is not enough work per thread.

From the experience gained, portability is not much of an issue since the parallel data model does not require precise tuning specific to any hardware architecture. During lab experiments the code was tested on 8, 24, 48 and 64 core architectures without any modification. In these cases execution speed was dependent on a thread to core ratio.

5.b Programming Model - Haskell

Programming model used in the Haskell implementation can be classified as static farmer, where the farmer is `sumTotient` function that takes in a list of static chunks and spreads them over to the fixed amount of workers - `sumTotient'`. The workers find totients for every elements in a chunk and return it
back to the fatmer. In turn, farmer accumulates results and sums them together to produce the sum of
totients. Every worker in Haskell implementation is sparked and is a potential thread during the execution.

Described model should not be called load balancing, as both chunk sizes and the number of workers are
always the same for a given input. No intelligent balancing is done during the run-time, neither on the
farmers side nor on the workers (peer-to-peer balancing) side. Farmer simply splits and distributes further
the work and then waits for the work to be done.

The advantage of using workers over, for example, a simple pipeline programming model, is that workers
compute in parallel, while farmer plays a role of both distributor and the merger. Disadvantage of this
model is that more parameters have to be considered and either an educated guess made regarding, for
example, a chunk size and/or amount of workers. Bad choices of these parameters can result in an extra
overhead in case if chunks are either too small or too big or amount of workers is too big or too small.

Programming models are intuitive and support in general the notion of modularity and hierarchy of a pro-
gram, thus, making a program more readable and understandable. It is important to note that a choice
of programing model depends highly on the problem at hand. If a problem cannot and should not be ap-
proached in divide and conquer fashion, then division of a task into independent sub-tasks is probably not
the best idea.

A problem, in any scenario, should be viewed in an abstract way to allow a problem solver to think in
terms of patterns and models. By doing so, programmer enters into a new dimension, from where prob-
lems can be collected into clusters that exhibit similar patterns (i.e. repeatability over the same calcula-
tions of the same values or similar) and generalised version of a problem at hand can be solved using a
particular programming model. Therefore, portability of programming models should be common in be-
tween different problems.

6 Appendices

The appendices provides the code listings for each parallel TotientRange program, as well as descriptions
of the parallel paradigm used and performance tuning approaches used.

6.a SAC Implementation - Max Baird

For this implementation a data-parallel paradigm was used. Speedups were anticipated via this approach
because of the nature of the problem at hand; essentially, there was no data dependency between calcu-
lations which meant that each could be done irrespective of order. The summation operation of the final
values can also be done irrespective of order.

With respect to performance tuning, the generated matrix elements were interleaved so that the earlier
threads did not have much less work that later threads, the order of operations have no effect in obtain-
ing the correct result. Since the sum of totients occur over a linearly increasing data set it means there is
more computation to be done for larger values which occur in the tail end of the series. Reference figure
11 below for the an example of the load balancing arrangement with a 10x10 matrix.
In essence, numbers from the start of the series are swapped with numbers toward the end of the series. This arrangement showed immediate performance gains since both earlier and later threads share the “heavy lifting”. Threads operating on the values closer to the center of the array have an average amount of work.

6.a.1 SAC Code Listing

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

#define DS1 1
#define DS2 2
#define DS3 3

#define DS1_SIZE 15
#define DS2_SIZE 30
#define DS3_SIZE 100

int hcf(int x, int y)
{
    int t;

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

    return x;
}

bool relprime(int x, int y)
{
    bool result;

    if(hcf(x, y) == 1){
```

Figure 10: Load Balancing example for SAC Implementation
result = true;
}
else{
    result = false;
}
return result;

int euler(int n)
{
    int length, i;
    length = 0;
    for (i = 1; i <= n; i++){
        if (relprime(n, i) == true)
            length++;
    }
    return length;
}

/* Does the load balancing, determines what value should go into the matrix location. */
int getVal(int n, int maxN)
{
    int res;
    if (n%2 == 0)
    {
        res = maxN - n;
    }
    else
    {
        res = n;
    }
    return res;
}

int main()
{
    ds = 0;
    if (argc() < 2)
    {
        error(-1, "Invalid argument amount\n");
    }
    input1 = String::atoi(argv(1));
    if (input1 == 0)
{  
    error(-1, "Only numbers 1, 2, 3\n");  
}

if (input1 == DS1)
{
    ds = DS1_SIZE;
}
elser if (input1 == DS2)
{
    ds = DS2_SIZE;
}
elser if (input1 == DS3)
{
    ds = DS3_SIZE;
}

k = 0;
x = (ds * 1000) + 2;

/*
   The with loop generates the matrix and applies
   the euler calculation in a single pass
*/

a = with {
    ( [0,0] < [i,j] < [ds,1000]) {k=euler(getVal(((i*1000)+j)+1, x));:k;}
} : genarray([ds,1000],0);
	total = sum(a); //Get the sum
printf("Sum of Totients between [1..%ld] is %ld\n", ds * 1000, total);

return(0);
}

6.b Haskell Implementation - Boris Mocialov

\[
\begin{bmatrix}
1 & 202 & 403 & 604 & \ldots \\
2 & 203 & 404 & 605 & \ldots \\
3 & 204 & 405 & 606 & \ldots \\
4 & 205 & 406 & 607 & \ldots \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
\end{bmatrix}
\]

Figure 11: Load Balancing for Haskell Implementation with chunk size of 200 elements

Although the load balancing had been performed on the level of items in every chunk, the runtime did not seem to be improved as much compared to the implementation without such load balancing. An as-
Assumption can be made that this type of load balancing does not work for Haskell implementation the same way it did in C+MPI implementation. For the future work it is worth exploring the dynamic allocation of chunks to different workers as well as some sort of peer-to-peer load balancing, where workers are responsible to transfer data across other workers.

For the current parallel implementation, the only parallel feature available by Glasgow parallel Haskell had been exploited, namely, the strategy of processing every list element in parallel, which resembles the farmer programming model.

Figure 12: Execution of parallel implementation using 32 threads on [1 - 15000] dataset 64 core machine

Figure 12 obtained from threadscope shows how threads are being executed in parallel implementation of the program. From the diagram, it can be seen that initially, some time is taken for partitioning and management of the data to be processed (i.e. creating lists and splitting into chunks). Later all threads are executed simultaneously on different cores without any visible significant interruptions. Garbage collection is also in place for every thread working along with the execution, collecting used data structures that are not needed any more.

6.b.1 Haskell Code Listing

```
module Main (main) where

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

-- sumTotient -- processed every balanced chunk in parallel

-- 1. Receives a list of balanced chunks of data
-- 2. Sends every chunk in parallel further to the main function -- sumTotient'
-- 3. Sums received back list of totients
-- 4. Sums summed lists of totients

sumTotient :: [[Int]] -> Int
sumTotient aList = sum . map sum $ (map (\x -> sumTotient' x) aList) 'using' parList rdeepseq

-- Main Function, sumTotient'

-- 1. Receives a list from sumTotient
-- 2. Evaluates every element of the list in parallel by passing that element further to the euler function
-- 3. Returns a list of elements resulting when Euler's phi function is applied

sumTotient' :: [Int] -> [Int]
sumTotient' aList = (map euler aList) 'using' parList rdeepseq

-- 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 (filter (relprime n) [1 .. n])

-- 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)

--- mkList

--- make a balanced list (element position + chunk size) -> i.e. 1, 202, 403, ...

mkList :: Int -> Int -> [Int]
mkList 201 _ = []
mkList lower upper = [lower, (lower + 200) .. upper] ++ (mkList (lower+1) (upper+1))

--- chunk -- splits one list into list of chunks of size specified in first parameter

chunk :: Int -> [a] -> [[a]]
chunk n [] = []
chunk n xs = as : chunk n bs
  where (as, bs) = splitAt n xs

--- 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
      aList = mkList lower upper -- make a general list
      putStrLn("so . . " ++ show(Main.sumTotient (chunk 200 aList)))