F21DP Distributed and Parallel Technologies.
Coursework: Sum Totient in Parallel Haskell and OpenMP on multicore CPUs.

Summary

In this coursework you will use and compare a *quite* high level parallel programming model (OpenMP) with a *very* high level parallel programming mode (parallel Haskell) for programming multicore CPUs.

You will develop and measure parallel versions of a program using one of the following parallel programming technologies: Parallel Haskell or OpenMP. You will compare the performance and programming models of these technologies. The parallel machines are nodes in the Robotarium high-performance cluster at Heriot-Watt.

Robotarium cluster website: `https://ecr-cluster.github.io`

You can expect feedback on your coursework within 3 weeks of the deadline date.

Plagiarism

In your group pair you can share ideas for your parallel code implementation and work together on the report text. However you must not plagiarise text from the internet or from external publication sources. Your report will be checked for plagiarism, and *if any is detected your submission will be treated as an academic misconduct case*. Please read carefully the university's plagiarism and discipline policies:

`https://www.hw.ac.uk/uk/students/studies/examinations/plagiarism.htm`
`https://www.hw.ac.uk/uk/students/doc/discguidelines.pdf`

Use of AI systems that generate text, e.g. ***ChatGPT***, is **not permitted** for this coursework. The university's guidance *"Introduction to AI content creation tools and university study: Student Guide"* states:

> *"Submitting work which is not your own is plagiarism i.e. cheating. If you were to use an AI content tool to draft/write your assessment, then the assessment would not be your own work."*

`https://lta.hw.ac.uk/wp-content/uploads/Student-Guide-Introduction-to-Artificial-Intelligence-content-creation-tools-university-study.pdf`

Submission instructions

There are two parts to the submission process: your code and your report.

1. **Your source code implementation.** Your code must be submitted to GitLab Student.
   - One student in your coursework group should **fork** this project. **Do not rename** the project when forking it.
     `https://gitlab-student.macs.hw.ac.uk/f20dp-2022-23/f20dp-cwk1`
     That student should then add their coursework partner to that GitLab project.
   - Include the URL of your project fork to the GitLab project in your report.
2. **A report**. A PDF should be submitted on Canvas.
3. **Report contributions**. Your report should state who wrote each section in your report.
4. **Authorship declaration**. Each student must submit a declaration of authorship on Canvas.

Marks will be lost if your report does not follow the structure outlined below or does not contain all of the specified results. The deliverable is due 3:30pm UK time on 7^{th} April 2023.

Learning Objectives of coursework 1

The Learning Outcomes (LO) of this F21DP course are listed at the end of this document.
The learning objectives of this coursework is tied to many of these Learning Outcomes:

- Learn how to use high level programming models for parallel programming (LO5).
- Compare and contrast declarative parallel programming (Parallel Haskell) and imperative parallel programming (OpenMP) skills (LO2, LO4, LO6, LoL8).
- Ability to benchmark parallel performance: *efficiency*, *scalability* and *speedup* (LO3, LO6, LO7).
- Compare the parallel programming models available with Parallel Haskell and with OpenMP (LO1, Lo7, LO10).
- Reflect on the abstractions that Parallel Haskell and OpenMP offer for programming multicore CPUs (LO10).
- Peer collaboration on the development of Parallel Haskell and OpenMP implementation and comparison (LO9).

The Algorithm to parallelise

The program that should be parallelised is the computation of the *sum of Euler totient computations* over a range of integer values.

The *Euler totient function* computes, for a given integer n, the number of positive integers smaller than n and relatively prime to n, i.e. $\Phi(n) \equiv\mid \{m \mid m \in \{1, \ldots, n-1\} \wedge m \perp n\} \mid$. Two numbers m and n are relatively prime, if the only integer number that divides both is 1. To test this, it is sufficient to establish that their greatest common divisor is 1, i.e. $m \perp n \equiv \gcd\ m\ n = 1$. Thus, the task for this program is: for a given integer n, compute $\Sigma_{i=1}^{n}\Phi(n)$.

A video has been created to describe the Sum Totient algorithm to be implemented in Parallel Haskell and OpenMP. The video also gives suggestions on how to measure, plot and analyse your performance results:
`https://web.microsoftstream.com/video/39738d62-57ad-4669-8d97-5de387b81471`

Sequential reference versions

C

The following C code is a direct implementation of the above specification, as a starting point for your OpenMP implementation:
`https://gitlab-student.macs.hw.ac.uk/f20dp-2022-23/totient-range`
Try it out:

```
$ make
$ make test
$ ./totient 1 1000
$ ./totient 10 2000
```

Haskell

The sequential Haskell reference implementation is:
`https://gitlab-student.macs.hw.ac.uk/f20dp-2022-23/f20dp-haskell-totient`
Follow the README file in that project for guidance on running this sequential reference version.

Organisation

The **assessed coursework work is to be carried out in pairs**, and you should choose your own partner. If you do not have a partner, contact me (R.Stewart@hw.ac.uk) and I will pair people together.

Each member of the team chooses either Parallel Haskell or OpenMP, and implements a parallel version of *totient range* in this technology and evaluates the performance of this parallel version:

- Parallel Haskell for shared memory multicore CPU parallelism;
- C with OpenMP for shared memory multicore CPU parallelism.

Together you can share ideas about how to parallelise the code and how to tune the parallel performance. However, it needs to be clearly stated who implemented the parallel version using which technology. *Together* you should prepare a comparative report using the structure below. It is relatively easy to produce a simple parallelisation of both programs, however *additional marks are available for thoughtful sequential and parallel performance tuning* (Question 6).

Tools that can help you, and have been discussed on these slides, are `gprof` and `cachegrind` for sequential C and OpenMP profiling functions like `omp_get_wtime`. For Haskell there is ThreadScope. You are welcome to use other tools, if you find them useful. In each case you should motivate your choice of tool and reflect how useful it was for tuning performance. For plotting parallel performance results `gnuplot`, `R`, `Excel` or one of many Python libraries are recommended. For Haskell benchmarks, the `criterion` library is recommended – see the README on the GitLab `f20dp-2022-23-haskell-totient` repository.

Structure of the report

Each question should be answered **in its own section** in your report. For example your answer to Q2 should be beneath heading *Section 2: Sequential Performance Measurements*, your answer to Q3b should be beneath heading *Section 3b: Runtime Graphs*, etc.

1. (4 points) **Introduction**

   This should give a short summary of the task to implement and parallelise, describe the software and hardware environments it is performed in, the parallel technologies used, and the learning objective of this coursework.

2. (4 points) **Sequential Performance Measurements**

   Run the sequential version of the programs and analyse the performance, for an extra mark using a profiling tool such as those mentioned above. Discuss the sequential performance of, and possible improvements to, these programs. Of interest are in particular code on the critical path (hotspots) in the program and good cache usage. *max 1 A4 page*

3. **Comparative Parallel Performance Measurements**

   You should measure and record the following results in numbered sections of your report. The measurements are based on these inputs:

   - DS1: calculating the sum of totients between 1 and 15000.
   - DS2: calculating the sum of totients between 1 and 30000.
   - DS3: calculating the sum of totients between 1 and 100000.

   For each of these inputs, measure:

   - SEQ: the sequential runtime on one compute node (computer) on the cluster.
   - Haskell: the runtime of the Haskell implementation one Robotarium compute node, measuring on 1-64 cores on the CPU.
   - OpenMP: the runtime of the OpenMP implementation one Robotarium compute node, measuring on 1-64 cores on the CPU.

   Runtime measurements should be repeated 3 times each.

(a) (6 points) **Runtime Table**

Present your measurements in a table containing the median of the three runtimes as well as difference between the mean and the extreme values. Use a suitable unit to ensure readability.

(b) (2 points) **Runtime Graphs**

Present your runtimes in **three** graphs, one for each problem size. The x-axis should be the parallel degree (how many CPU cores used), the y-axis runtime.

(c) (3 points) **Speedups**

Plot graphs for Parallel Haskell and OpenMP, showing speedup graphs corresponding to the runtime results for DS1, DS2 and DS3. Include the ideal speedups as a line as well. Also show a table with the sequential performance and the best parallel runtimes of OpenMP and Parallel Haskell.

(d) (4 points) **Efficiency**

Plot the corresponding efficiency graphs. Analyse the efficiency performance of OpenMP and Parallel Haskell, speculate on the reasons for the efficiency performance, and discuss for which speedup graphs an efficiency graph is a sensible addition.

(e) (2 points) **Hardware Utilisation (OpenMP only)**

For the OpenMP implementation, figure out how many operations on `long` values are being performed for the three different inputs. Then compute the number of operations per second that you achieved in your OpenMP implementation for the different inputs and for different number of CPU cores used. Plot these findings as graphs and explain how you calculated the number of operations.

(f) (5 points) **Discussion**

A discussion of the comparative performance of scalability, efficiency and speedups for both parallel implementations. Try to identify a suitable performance measure and derive those figures from your experiments. Discuss the impact of the shared-memory model of OpenMP and Parallel Haskell, and how this model might explain your results. Also discuss how Amdahl's law impacts parallel performance of the Totient Range algorithm. *max 1 A4 page*

4. (4 points) **Comparing Programming Models for Sum Totient**

An evaluation of the parallel programming models, specifically for *implementing the totient application. Your answer for this question should be written jointly by both group members.* You should indicate any challenges you encountered in constructing and parallelising your Sum Totient programs, and discuss situations/algorithms where each technology may usefully be applied.

5. (10 points) **Reflection on Programming Models**

This section should draw on your experiences in using the different programming models and discuss their suitability for parallelisation of applications in general. *Your answer for this question should be written jointly by both group members.* The discussion should include aspects such as performance expectations, programmability, advantages/disadvantages, debugging support, performance tuning challenges. This discussion needs to be general, but can draw on the experience you gained in using the models on the Totient application. *max 1 A4 page.*

6. (4 points) **Performance tuning**

Additional marks are available for thoughtful sequential and parallel performance profiling and tuning throughout the report.

7. (4 points) **Algorithmic knowledge**

Reflect on how to use *algorithmic* knowledge of the $\Sigma_{i=1}^{n}\Phi(n)$ computation, with the goal of creating parallel tasks of approximately equivalent computation size in your OpenMP and Parallel Haskell programs. Hint: try `runBenchmark()` in `TotientRange.c` and design a partitioning and agglomeration strategy to shorten runtimes further.

8. (10 points) **Parallel computing concepts**

   This section should draw on your knowledge about parallel computing concepts and challenges. In this section, specifically answering the following questions:

   1. What hardware architectures trends have motivated the growth of parallel programming languages and libraries?
   2. Why is the software industry increasingly embracing parallel computing?
   3. What steps should a programmer take to translate algorithms to parallel code, to efficiently run on multi-core hardware?
   4. Why is parallel programming hard?
   5. What are examples of parallel patterns that high level programming models abstract?

   For your answer, refer to the ***recommended reading list on Canvas*** for *Introduction to parallelism*. ***You must not copy the text***. Instead, you should summarise the key messages in those articles in your own words. Provide references in sentences where appropriate, include a bibliography list of these articles in a *References* section at the end of your report. *max 1 A4 page.*

9. (20 points) **Appendix A and B**

   For each parallel implementation, the appendix should include a GitLab URL for your parallel Totient programs. Each implementation should clearly labelled with the single author's name in the report and in the GitLab source code as a comment. Include a paragraph in the appendix, and possibly diagram(s), identifying the *parallel paradigm* used, and *performance tuning approaches* used.

The total coursework mark (bonus points are included in your total mark) is calculated as follows:

Question:	1	2	3	4	5	6	7	8	9	Total
Points:	4	4	22	4	10	4	4	10	20	82

Notes

- Complete the Parallel Haskell and OpenMP Lab exercises before starting the coursework.
- Check these slides for practical tips on performance measurements, which has a focus on C based parallel programs.
- Graphs and tables must have appropriate captions, and the axes must have appropriate labels.
- You should use your own individual Robotarium accounts to login to the head node and deploy jobs to the compute nodes.
- The Robotarium cluster uses a batch-job system to give you exclusive access to the cluster when you need it for measurements. Follow this link to familiarise yourself with the batch-job system.

Learning Outcomes of this Course

The course aims and learning outcomes of this course are available course descriptor online:
`https://www.hw.ac.uk/documents/pams/202122/F21DP_202122.pdf`

The Learning Outcomes of this F21DP course are:

LO1 Understanding of foundational concepts of distributed and parallel software.

LO2 Knowledge and application of contemporary techniques for constructing practical distributed and parallel systems using both declarative and imperative languages.

LO3 Parallel performance tuning using appropriate tools and methodologies.

LO4 Understand the role of control and data abstraction in software design and implementation.

LO5 Appreciation of relationship between imperative and declarative models of parallelism.

LO6 Critically analyse parallel and distributed problems.

LO7 Generate, interpret and evaluate parallel performance graphs.

LO8 Develop original and creative parallel problem solutions.

LO9 Showing initiative, creativity and team working skills in shared distributed and parallel application development.

L10 Demonstrate critical reflection, e.g. understanding of applicability of, and limitations to, parallel and distributed systems.

END OF COURSEWORK