Distributed and Parallel Technologies: Coursework Stage 1

Comparing Programming Models: Shared-Memory Models

This coursework is in two stages (worth 10% each). In this first stage you will develop parallel programming, experimentation, and technology evaluation skills. The parallel architecture is shared-memory multicore. In the second stage we consider distributed parallel implementations.

Given sequential versions of a program, you will develop and measure parallel versions using C+OpenMP (from now, we use OpenMP to mean C+OpenMP) and Go. You will then compare the performance of the implementations and compare the technologies.

Overview

This is individual coursework. You must develop and tune the parallel performance of the OpenMP and Go programs, make systematic measurements, and prepare a comparative report. While it is possible to produce a simple parallelisation of both programs, additional marks are available for well justified and thoughtful parallel performance tuning. Inversely, extremely **over-engineered solutions might lose marks**. *You should parallelise in an idiomatic style based on the programming model being used, and the code should be stand-alone (other than standard libraries).*

The Problem

We will parallelise a program that sums the Euler Totient values over sequences of integers.

The sequential versions of the program are available on Moodle and in the Appendix of this document. We assume 64 bit integers everywhere so you may need to use a lab machine to develop if you do not have a 64 bit processor.

The Hardware

Your (final) datasets should be gathered on a GPG (Glasgow Parallelism Group) cluster node. Nodes `gpgnode-01`, `gpgnode-02`, and `gpgnode-03` are freely available without booking. You should use `ssh` to access these nodes (and `scp` to copy files if needed) as you would `stlinux`. Remember you might need to do two hops to get to the cluster, i.e. going via `ssh1`.

If you do not have a unix account, or have forgotten your password, then please raise a "CoSE IT Support" helpdesk request: https://glasgow.saasiteu.com/Modules/SelfService/#home; check you have access as soon as possible as speed of getting user accounts at the deadline cannot be guaranteed.

Note: the `gcc` version on gpg is 11.4, so if you are using super modern `C` features they might not be supported. Likewise `go` is version `1.20.5`. If you really need the features to show off your approach then you can use a portable binary.

Data Sets

Performance measurements should use three data sets:

- **DS1:** The sum of totients between 1 and 15000.
- **DS2:** The sum of totients between 1 and 30000.
- **DS3:** The sum of totients between 1 and 60000.

Deliverable

The deliverable is a report (including code sources at the end) with the structure below. *We reserve the right to penalise reports not following this structure.*

The report should be uploaded to **Moodle** by the deadline.

Section 0 Coursework Title and Author

Include the title of the coursework (DPT Stage 1), your name and GUID.

Section 1 Comparing Sequential Performance (2 marks)

Complete the table below showing the runtimes of the sequential C and Go programs on the DS1, DS2 and DS3 data sets on a GPG cluster node. You should choose an appropriate number of significant figures. Briefly reflect on the differences.

Note: you do not need to run DS3 (as it is quite large for sequential execution)

Data Set	Go Runtimes (s)	C Runtimes (s)
DS1		
DS2		

Section 2 Comparative Parallel Performance Measurements (12 marks)

You should measure and record the following results in numbered sections of your report. Runtime measurements should be the middle (median) value of three executions on a GPG cluster node. You may wish to add a runtime parameter to your Go program to control the number of gorountines created.

For comparison purposes the performance of the Go and OpenMP should be reported on the same graph. You may also plot other graphs to show interesting features, or use larger numbers of threads.

You may create your graphs in any tool you wish, but make sure it has enough resolution (excel sometimes struggles with this; gnuplot works well).

Section 2.1 Runtime Graphs

- DS1: execution times for the sequential C and Go programs; and the parallel OpenMP and Go programs on 1, 2, 4, 8, 12, 16, 24, 32, 48, 64 threads/goroutines on a GPG Cluster node.
- DS2: execution times for the sequential C and Go programs; and the parallel OpenMP and Go programs on 1, 2, 4, 8, 12, 16, 24, 32, 48, 64 threads/goroutines on a GPG Cluster node.
- DS3: execution times for the parallel OpenMP and Go programs on 8, 16, 32, 64, threads/ goroutines on a GPG Cluster node.

Note in the first two cases, the sequential, and 1 thread/gorouontine version may have different runtimes. Please show both.

Section 2.2 Speedup Graphs

- Plot absolute speedup graphs corresponding to the runtime results for DS1, DS2.
- Plot a *relative* speedup graph (assuming runtime on 1 thread is 8x runtime on 8 threads) for DS3. All plots should show the ideal speedup and the speedups for the OpenMP and Go programs.

Recall that absolute speedup is calculated using the runtime of the **sequential** program.

Section 2.3 Summary Table

Complete the table below summarising the sequential performance and the best parallel runtimes of your OpenMP and Go programs.

Language	Sequential Runtime (s)	Best Parallel Runtime (s)	Best Speedup	Threads/Goroutines
DS1				
Go				
OpenMP				
DS2				
Go				
OpenMP				

Section 2.4 Discussion

A discussion of the comparative performance of the OpenMP and Go programs. Be sure to highlight any interesting results, and hypothesise/back up why this might be the case **Max 1 A4 page**.

Section 3 Reflection on Programming Models For Totient (6 marks).

An evaluation of the Go and OpenMP parallel programming models *for the totient application*. You should indicate any challenges you encountered in constructing your programs and the situations where each technology may usefully be applied. You may highlight particular interesting features of your approach. The comparison should be based on the TotientRange application and be supported by technical arguments. **Max 1 A4 page**.

Level M Only **Section 4: General Reflection on Programming Models (5 marks)**

An evaluation of the OpenMP and Go programming models **in general**. This section should discuss the advantages and disadvantages of the programming models. It should discuss issues related to the parallel performance, programmability and usability of the models (including when you might chose one over the other). It may also address issues of portability, in principle, discussing which kind of application are best targeted with each programming model. **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**.

Section 5 Source Code Listings (10 marks)

This will be Section 4 for level H students.

Please ensure these are of a readable fontsize. We do not accept sources as separate files.

Please use comments **in the code** to help explain your thinking

Section 5.1 OpenMP

Section 5.2 Go

Hints and Tips

1. You should complete the Go and OpenMP Lab exercises before starting the coursework.
2. You should develop and test on a lab/home machine, and only run performance measurements on a GPG cluster node. You have access to gpgnode-01, gpgnode-02, and gpgnode-03. These machines have 16 (real) cores each.
3. The measurements you report should be the median (middle) of three executions
4. You may find it useful to write scripts to run the measurements
5. To ensure a fair comparison all measurements should be made on a lightly loaded GPG cluster node. Check the load on nodes using something like the `top` command. The nodes are likely to be busy near the submissions deadline so please take this into account.
6. Graphs and tables should have appropriate captions, and the axes should have appropriate labels.
7. You may tweak the sequential code structure if required but please justify this in the report.

Starting Code (Also on Moodle)

```
// totientRange.go -- Sequential Euler Totient Function (Go Version)
// compile -- go build
// run --    totientRange lower_num upper_num
// Based on code from Phil Trinder from earlier work by: Greg Michaelson,
// Patrick Maier, Phil Trinder, Nathan Charles, Hans-Wolfgang Loidl and Colin
// Runciman

// This program calculates the sum of the totients between a lower and an upper
// limit

package main

import (
    "fmt"
    "os"
    "strconv"
    "time"
)


// Compute the Highest Common Factor, hcf of two numbers x and y
// hcf x 0 = x
// hcf x y = hcf y (mod x y)
func hcf(x, y int64) int64 {
    var t int64
    for (y != 0) {
        t = x % y
        x = y
        y = t
    }
    return x
}

// relprime x y = hcf x y == 1
func relprime(x, y int64) bool {
    return hcf(x, y) == 1
}

// euler(n) computes the Euler totient function, i.e. counts the number of
// positive integers up to n that are relatively prime to n
func euler(n int64) int64 {
    var length int64
    var i int64
    for i = 1; i < n; i++ {
        if relprime(n, i) {
            length++
        }
    }
    return length
}

// sumTotient lower upper sums the Euler totient values for all numbers
// between "lower" and "upper".
func sumTotient(lower,upper int64) int64 {
    var sum int64
    for i := lower; i <= upper; i++ {
        sum += euler(i)
    }
    return sum
}

func main() {
    var lower, upper int64
    var err error
    // Read and validate lower and upper arguments
    if len(os.Args) < 3 {
        panic(fmt.Sprintf("Usage: must provide lower and upper range limits as arguments"))
    }

    // Go if supports "If with a short statement"
    if lower, err = strconv.ParseInt(os.Args[1],10,64); err != nil {
        panic(fmt.Sprintf("Can't parse first argument"))
    }
    if upper, err = strconv.ParseInt(os.Args[2],10,64); err != nil {
        panic(fmt.Sprintf("Can't parse second argument"))
    }

    start := time.Now()
    totients := sumTotient(lower,upper)
    elapsed := time.Since(start)
    fmt.Println("Sum of Totients between", lower, "and", upper, "is", totients)
    fmt.Println("Elapsed time", elapsed)
}
```

```
// TotientRange.c - Sequential Euler Totient Function (C Version)
// compile: gcc -Wall -O3 -o TotientRange TotientRange.c
// run:     ./TotientRange lower_num upper_num

// Based on code from Phil Trinder from earlier work by: Greg Michaelson,
// Patrick Maier, Phil Trinder, Nathan Charles, Hans-Wolfgang Loidl and Colin
// Runciman

// This program calculates the sum of the totients between a lower and an upper
```

```c
// limit

#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
#include <time.h>

// Compute the Highest Common Factor, hcf of two numbers x and y
// hcf x 0 = x
// hcf x y = hcf y (rem x y)
int64_t hcf(int64_t x, int64_t y) {
  int64_t t;
  while (y != 0) {
    t = x % y;
    x = y;
    y = t;
  }
  return x;
}

// relprime x y = (hcf x y == 1)
int relprime(int64_t x, int64_t y) {
  return hcf(x, y) == 1;
}

// euler n = length (filter (relprime n) [1 .. n-1])
int64_t euler(int64_t n) {
  int64_t length = 0;
  for (int64_t i = 1; i < n; i++) {
    if (relprime(n, i)) {
      length++;
    }
  }
  return length;
}


// sumTotient lower upper = sum (map euler [lower, lower+1 .. upper])
int64_t sumTotient(int64_t lower, int64_t upper) {
  int64_t sum = 0;
  for (int64_t i = lower; i <= upper; i++) {
    sum = sum + euler(i);
  }
  return sum;
}


int main(int argc, char ** argv) {
  int64_t lower, upper;

  if (argc != 3) {
    printf("not 2 arguments\n");
    return 1;
  }

  sscanf(argv[1], "%" SCNd64, &lower);
  sscanf(argv[2], "%" SCNd64, &upper);

  struct timespec start, end;
  clock_gettime(CLOCK_MONOTONIC, &start);
  int64_t sum = sumTotient(lower, upper);
  clock_gettime(CLOCK_MONOTONIC, &end);

  double time_taken_ns = 1000000000L * (end.tv_sec - start.tv_sec) + end.tv_nsec - start.tv_nsec;

  printf("C: Sum of Totients  between [%ld..%ld] is %ld\n",
         lower, upper, sum);
  printf("Time Taken: %f s\n", time_taken_ns / 1000 / 1000 / 1000);

  return 0;
}
```