**Section 1 Comparative Sequential Performance**

Sequential

|  |  |  |
| --- | --- | --- |
| Data Set | Go Runtimes (s) | C Runtimes (s) |
| DS1 | 15.91639141 | 15.222714 |
| DS2 | 68.21682146 | 65.688663 |

**Section 2 Comparative Parallel Performance Measurements**

**Section 2.1 Runtimes**

Dataset 1

|  |  |  |
| --- | --- | --- |
| Threads | Go | OpenMP |
| 1 | 16.2292 | 15.18 |
| 2 | 12.44243 | 11.668 |
| 4 | 7.478488 | 7.022 |
| 8 | 4.147318 | 3.9 |
| 12 | 2.887657 | 2.7 |
| 16 | 2.201418 | 2.279 |
| 24 | 1.628665 | 1.432 |
| 32 | 1.270825 | 1.11 |
| 48 | 0.982692 | 0.898 |
| 64 | 0.878035 | 0.822 |

Dataset 2

|  |  |  |
| --- | --- | --- |
| Threads | Go | OpenMP |
| 1 | 69.29402 | 71.705 |
| 2 | 53.03127 | 55.978 |
| 4 | 31.78972 | 31.284 |
| 8 | 17.68323 | 16.716 |
| 12 | 12.31131 | 11.547 |
| 16 | 9.407323 | 8.822 |
| 24 | 6.699683 | 6.185 |
| 32 | 5.323557 | 4.855 |
| 48 | 4.30356 | 3.837 |
| 64 | 3.81732 | 3.429 |

Dataset 3

|  |  |  |
| --- | --- | --- |
| Threads | Go | OpenMP |
| 8 | 74.89538 | 70.478 |
| 16 | 39.81903 | 37.504 |
| 32 | 22.33225 | 20.811 |
| 64 | 15.64302 | 14.783 |

**Section 2.2 Speedups**

**Section 2.3**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Language | Sequential Runtime (s) | Best Parallel Runtime (s) | Best Speedup | No. Threads |
| Go | 15.91639 | 0.878035 | 18.12729 | 64 |
| C+OpenMP | 15.22271 | 0.822 | 18.51912 | 64 |

**Section 2.4**

The runtimes and speedups for both Go and C+OpenMP are very similar, with C+OpenMP being slightly faster on average in both aspects. The speedup graphs for DS1 and DS2 show C+OpenMP causing a greater speedup compared to Go, starting at around 16 threads, and DS2 seems to suggest this trend will continue. However, with few datapoints, both speedup lines could merge at some upper limit, as the trend for DS1 suggests.

Looking at the numbers for speedups, the results for DS1, DS2 and DS3 are very similar, with 64 threads providing 18-19 times speedup on average for all three. The graph is also mostly linear, which suggests the parallelism is ideal, especially for higher number of threads.

**Section 3 Programming Model Comparison**

The biggest challenge I had was deciding how to split the dataset into correctly sized chunks for the Go program. This was especially difficult as the count of goroutines would often not be a factor of the dataset (e.g. 60000 / 64) and due to Go’s integer division always rounding down this could cause some numbers to be skipped/processed twice. I first experimented with converting the integers to floats and rounding explicitly, but this was messy and didn’t guarantee the final upper number would be correct (e.g. could be rounded up to 60001). I finally decided to stick with the current functionality, which uses integer division to divide the dataset into a variable called chunk (threads\*chunk\_size < dataset because integer division rounds down). Each goroutine would process work of chunk\_size, and the last goroutine would handle whatever was remaining. This guarantees that every number is processed and splits the data into almost even chunks.

**Appendix A C+OpenMP TotientRange Program**

**Code:**

// totientRangePar.go - Parallel Euler Totient Function (Go Version)

// compile: go build

// run:     totientRangePar lower\_num upper\_num num\_threads

// Author: Max Kirker Burton 2260452b   07/11/2019

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

// upper limit, and can be run with several Goroutines

//

// Each function has an executable Haskell specification

//

// It is based on earlier work by: Greg Michaelson, Patrick Maier, Phil Trinder,

// Nathan Charles, Hans-Wolfgang Loidl and Colin Runciman

package main

import (

    "fmt"

    "os"

    "strconv"

    "sync"

    "time"

)

// Compute the Highest Common Factor, hcf of two numbers x and y

//

// hcf x 0 = x

// hcf x y = hcf y (rem x y)

func hcf(x, y int64) int64 {

    var t int64

    for y != 0 {

        t = x % y

        x = y

        y = t

    }

    return x

}

// relprime determines whether two numbers x and y are relatively prime

//

// 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

//

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

func euler(lower int64, upper int64, ch chan int64, wg \*sync.WaitGroup) {

    var length, i, j int64

    for i = lower; i < upper; i++ {

        length = 0

        for j = 1; j < i; j++ {

            if relprime(i, j) {

                length++

            }

        }

        ch <- length

    }

    wg.Done()

}

// sumTotient lower upper sums the Euler totient values for all numbers

// between "lower" and "upper".

//

// sumTotient lower upper = sum (map euler [lower, lower+1 .. upper])

func sumTotient(lower, upper, cores int64) int64 {

    var sum, i int64

    ch := make(chan int64, 100000)

    sum = 0

    totalRange := upper - lower

    var goroutines int64 = cores

    chunkSize := totalRange / goroutines // floor division

    var wg sync.WaitGroup                // using a waitgroup to close the channel after all threads are completed

    for i = 0; i < goroutines; i++ {

        wg.Add(1)

        elower := lower + (chunkSize \* i)

        var eupper int64

        if i+1 == goroutines {

            eupper = upper + 1 // plus 1 to include final number

        } else {

            eupper = lower + (chunkSize \* (i + 1))

        }

        go euler(elower, eupper, ch, &wg) // round numbers, doesnt work perfectly as is on imperfect division

    }

    wg.Wait()

    close(ch)

    for value := range ch { // potentially change this so that summing happens as soon as one thread finishes

        sum += value

    }

    return sum

}

func main() {

    var lower, upper, cores 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"))

    }

    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"))

    }

    if cores, err = strconv.ParseInt(os.Args[3], 10, 64); err != nil {

        panic(fmt.Sprintf("Can't parse third argument"))

    }

    // Record start time

    start := time.Now()

    // Compute and output sum of totients

    fmt.Println("Sum of Totients between", lower, "and", upper, "is", sumTotient(lower, upper, cores))

    // Record the elapsed time

    t := time.Now()

    elapsed := t.Sub(start)

    fmt.Println("Elapsed time", elapsed)

}

Uses a Pragma-based Shared Memory paradigm (e.g. using shared variables). I used only one pragma in the final code, but I experimented with using nested pragmas, with another pragma inside of the euler function. I thought that I could parallelise the calculations of relprimes, and tried assigning threads in different ways (e.g. 2 threads for the euler function, and all other threads for the sumTotient function) but all of these methods resulted in slower runtimes than simply using one pragma.

**Appendix B Go TotientRange Program**

**Code:**

// TotientRangePar.c - Parallel Euler Totient Function (C Version)

// compile: gcc -Wall -O -o TotientRangePar TotientRangePar.c

// run:     ./TotientRangePar lower\_num upper\_num num\_threads(optional)

// Author: Max Kirker Burton 2260452b     13/11/19

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

// upper limit using C longs, and can be run with several Goroutines either set as an argument or

// as an environment variable

// It is based on earlier work by:

// Phil Trinder, Nathan Charles, Hans-Wolfgang Loidl and Colin Runciman

// The comments provide (executable) Haskell specifications of the functions

#include <stdio.h>

#include <omp.h>

#include <sys/time.h>

// hcf x 0 = x

// hcf x y = hcf y (rem x y)

long hcf(long x, long y)

{

  long t;

  while (y != 0) {

    t = x % y;

    x = y;

    y = t;

  }

  return x;

}

// relprime x y = hcf x y == 1

int relprime(long x, long y)

{

  return hcf(x, y) == 1;

}

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

long euler(long n)

{

  long length, i;

  length = 0;

  for (i = 1; i < n; i++)

    if (relprime(n, i))

      length += 1;

  return length;

}

// sumTotient lower upper = sum (map euler [lower, lower+1 .. upper])

long sumTotient(long lower, long upper, int n\_threads)

{

  long sum, i;

  sum = 0;

  #pragma omp parallel for reduction(+: sum) num\_threads(n\_threads)

    for (i = lower; i <= upper; i++)

      sum += euler(i);

  return sum;

}

int main(int argc, char \*\* argv)

{

  long lower, upper;

  int num\_threads = omp\_get\_num\_threads();

  float msec;

  struct timeval start, stop;

  if (argc < 3) {

    printf("fewer than 2 arguments\n");

    return 1;

  }

  sscanf(argv[1], "%ld", &lower);

  sscanf(argv[2], "%ld", &upper);

  if (argc == 4){

    sscanf(argv[3], "%d", &num\_threads);

  }

  gettimeofday(&start, NULL);

  printf("C: Sum of Totients  between [%ld..%ld] is %ld\n",

         lower, upper, sumTotient(lower, upper, num\_threads));

  gettimeofday(&stop, NULL);

  if (stop.tv\_usec < start.tv\_usec) {

    stop.tv\_usec += 1000000;

    stop.tv\_sec--;

  }

  msec = 1000 \* (stop.tv\_sec - start.tv\_sec) +

                (stop.tv\_usec - start.tv\_usec) / 1000;

  printf("%f\n", msec);  // Rename to elapsed time:

  return 0;

}

Uses a Message Passing paradigm (e.g. creating a channel with send and receives). The reason why low number of goroutines (e.g. 2) don’t provide a huge reduction in runtime is likely because of how the program separates work into chunks. My program splits work into mostly even chunks and calculating the sum of totients in a range 1-1000 would take much less time than 1000-2000, as more relative primes need to be calculated. Since I used a waitgroup to ensure all goroutines are completed before the sum of the channel is calculated, the program runtime will be the runtime of the slowest goroutine. This becomes less of a problem at higher counts of goroutines as the chunks are smaller. I thought of ways to have different sized chunks, with larger sized chunks for the earlier numbers (e.g. for a range of 2000 and 2 threads, split into chunks of 1-1500 and 1500-2000), but this proved difficult to implement, and I wasn’t sure the performance would be noticeably affected.