Comparing Parallel and Distributed Programming Models:

Coursework Stage 1: Performance and Programmability of Shared-memory Parallel Models

Max Kirker Burton, 2260452b

Section 1 Comparative Sequential Performance

Data Set	Go Runtimes (s)	C Runtimes (s)
DS1	15.91639141	15.21514
DS2	67.95405428	65.28168

Figure 1: Comparing C and Go Sequential Runtimes

The difference in sequential runtimes are almost negligible, however C does consistently achieve a runtime of 500-2500ms faster than Go. This is likely because of Go's garbage collector (explained in further detail in section 2.4).

Section 2 Comparative Parallel Performance Measurements

Section 2.1 Runtimes

Threads	Go	OpenMP
1	16.35538	15.171
2	9.066147	7.592
4	4.740579	3.973
8	2.461242	2.06
12	1.647562	1.376
16	1.264213	1.041
24	0.972845	0.795
32	0.80053	0.649
48	0.801696	0.651
64	0.8027	0.653

Figure 2.1: Comparing C+OpenMP and Go Parallel Runtimes for DS1

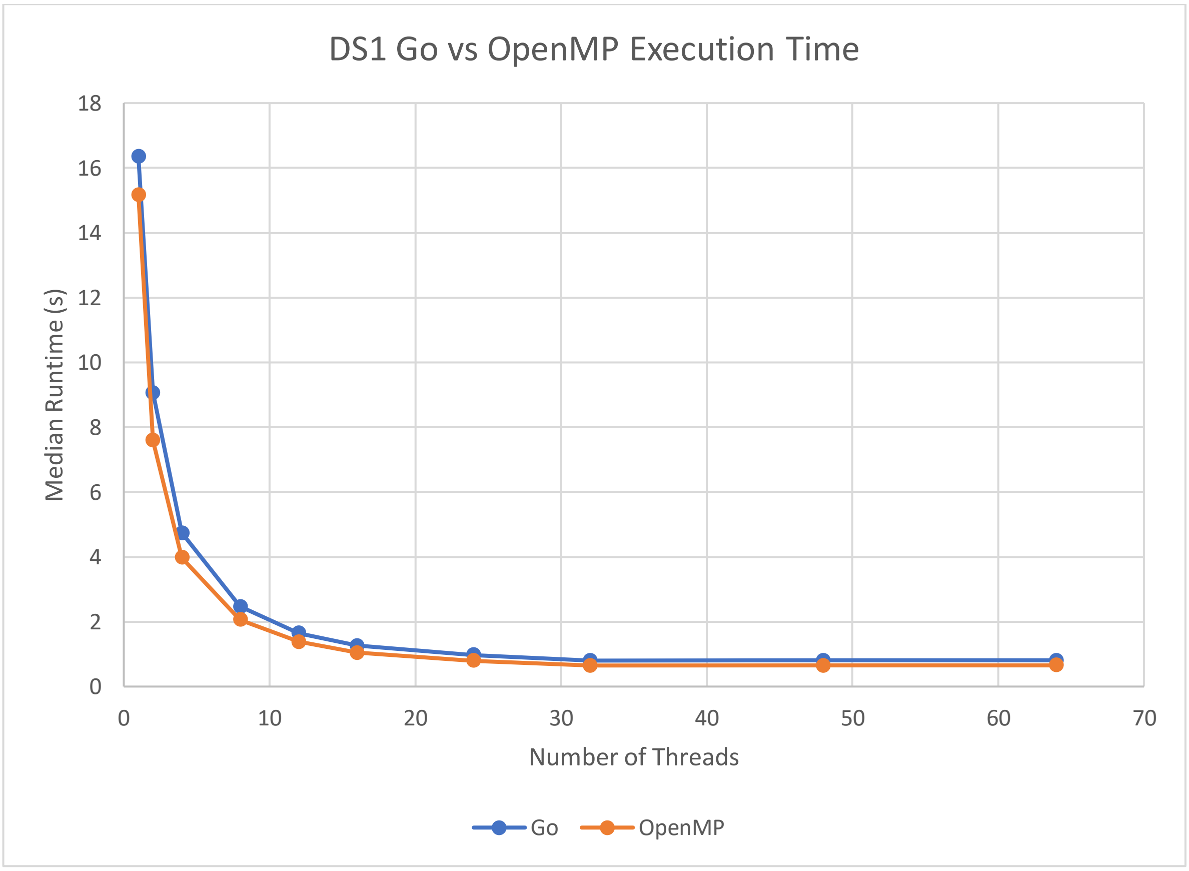

Figure 2.2: Comparing C+OpenMP and Go Parallel Runtimes for DS1, plotted

Threads	Go	OpenMP
1	69.91227	65.085
2	38.36811	32.562
4	20.0648	17.042
8	10.40659	8.839
12	6.941679	5.894
16	5.227036	4.43
24	4.097904	3.399
32	3.376901	2.76
48	3.374534	2.761
64	3.377058	2.76

Figure 2.3: Comparing C+OpenMP and Go Parallel Runtimes for DS2

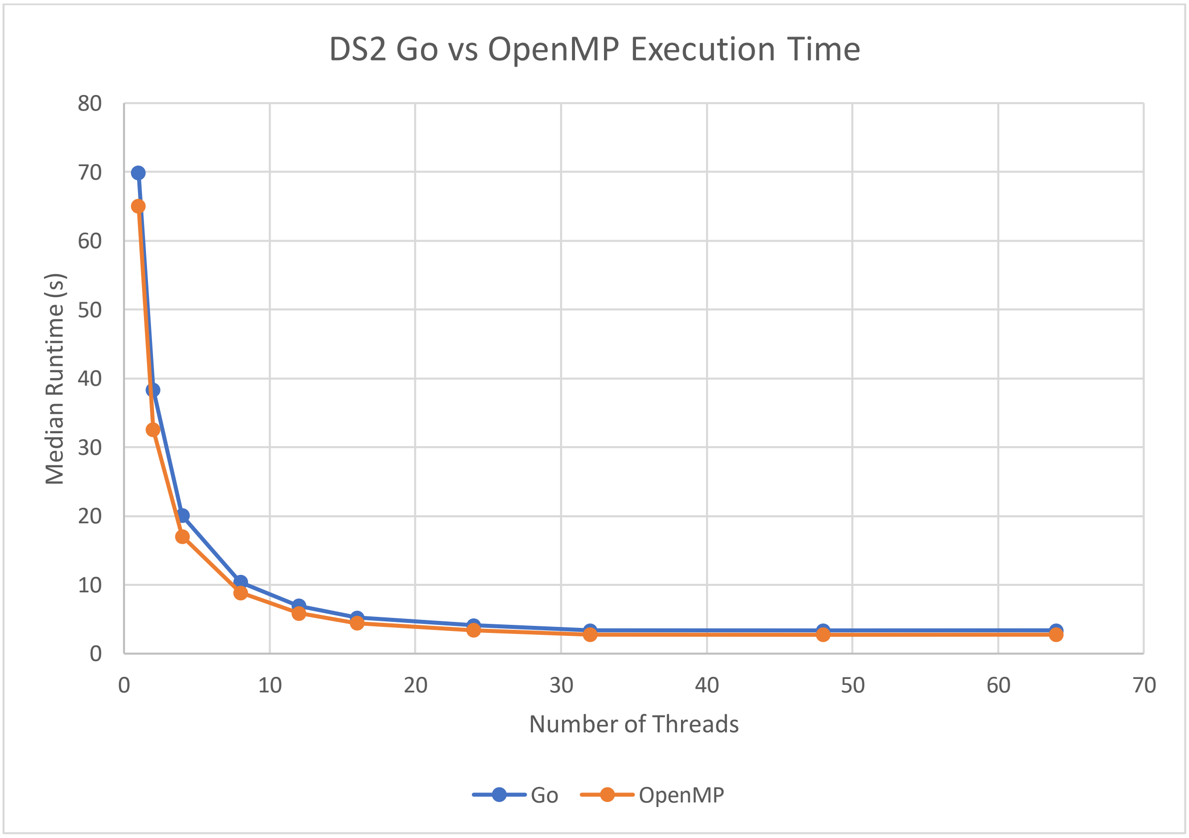

Figure 2.4: Comparing C+OpenMP and Go Parallel Runtimes for DS2, plotted

Threads	Go	OpenMP
8	43.90842	37.725
16	22.00349	18.89
32	14.206	11.741
64	14.20571	11.741

Figure 2.5: Comparing C+OpenMP and Go Parallel Runtimes for DS3

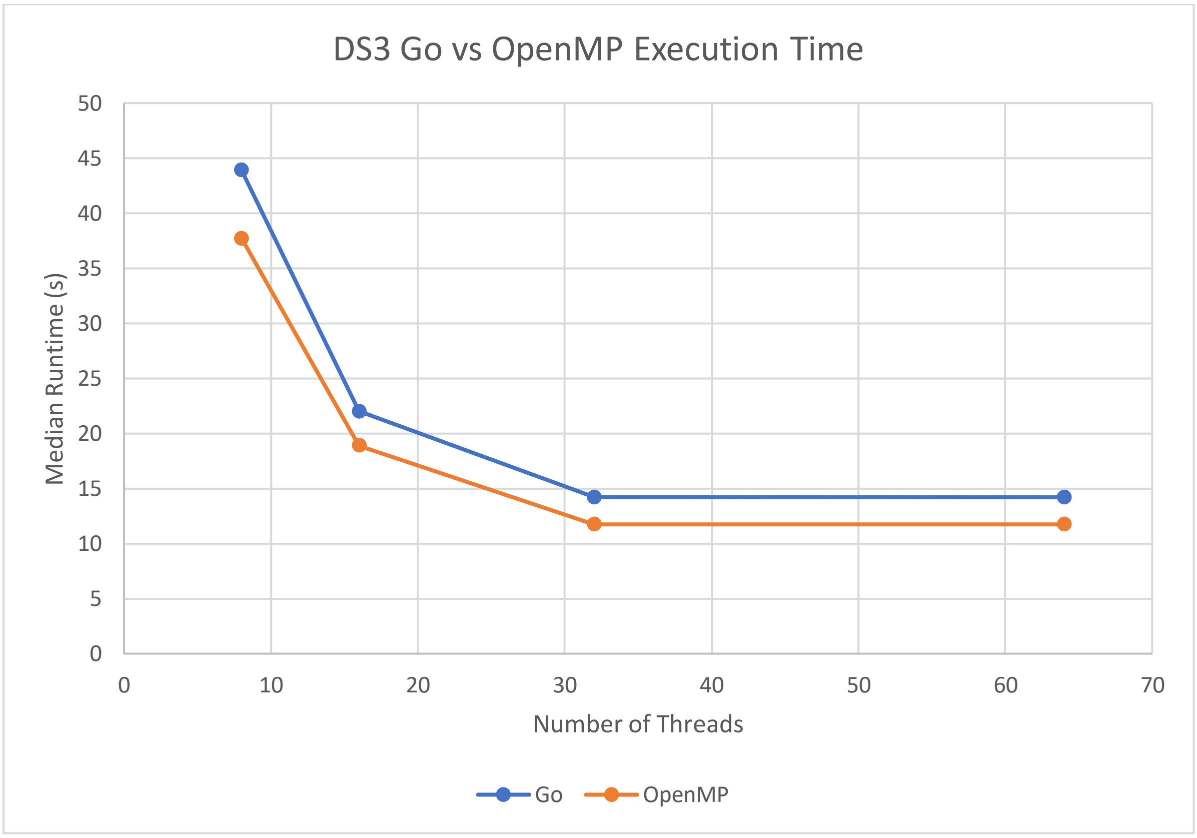

Figure 2.6: Comparing C+OpenMP and Go Parallel Runtimes for DS3, plotted

Section 2.2 Speedups

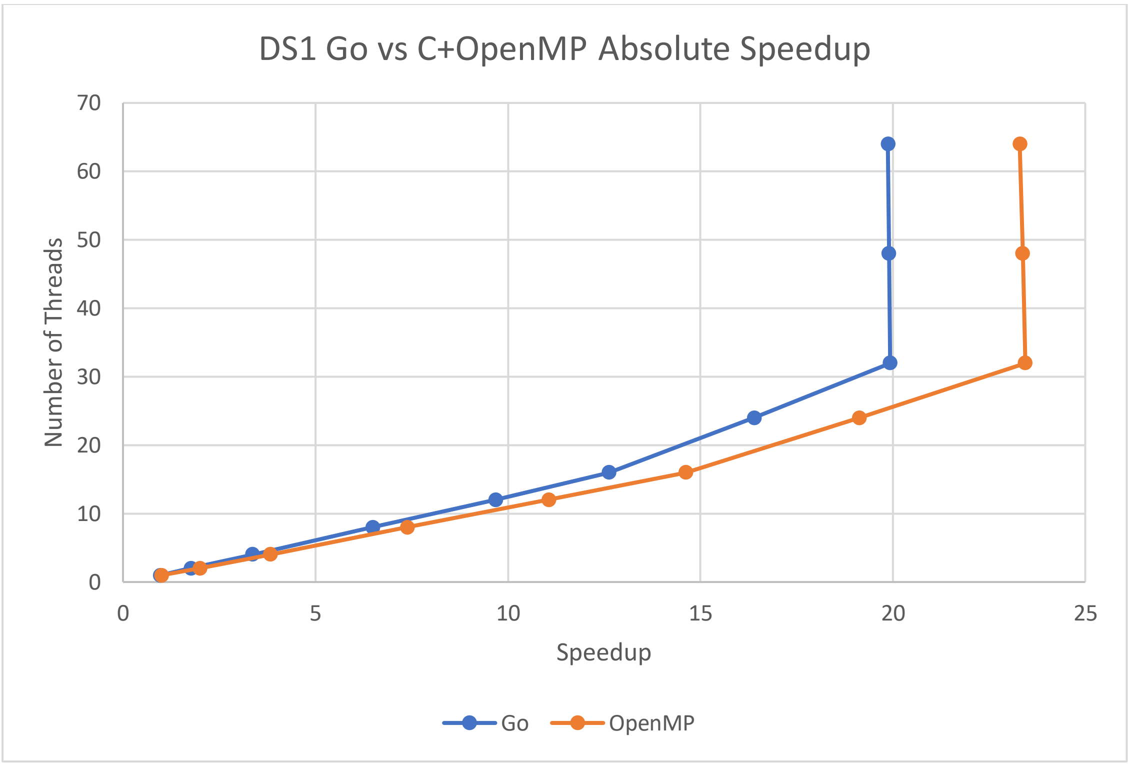

Figure 2.7: Comparing C+OpenMP and Go Absolute Speedups for DS1

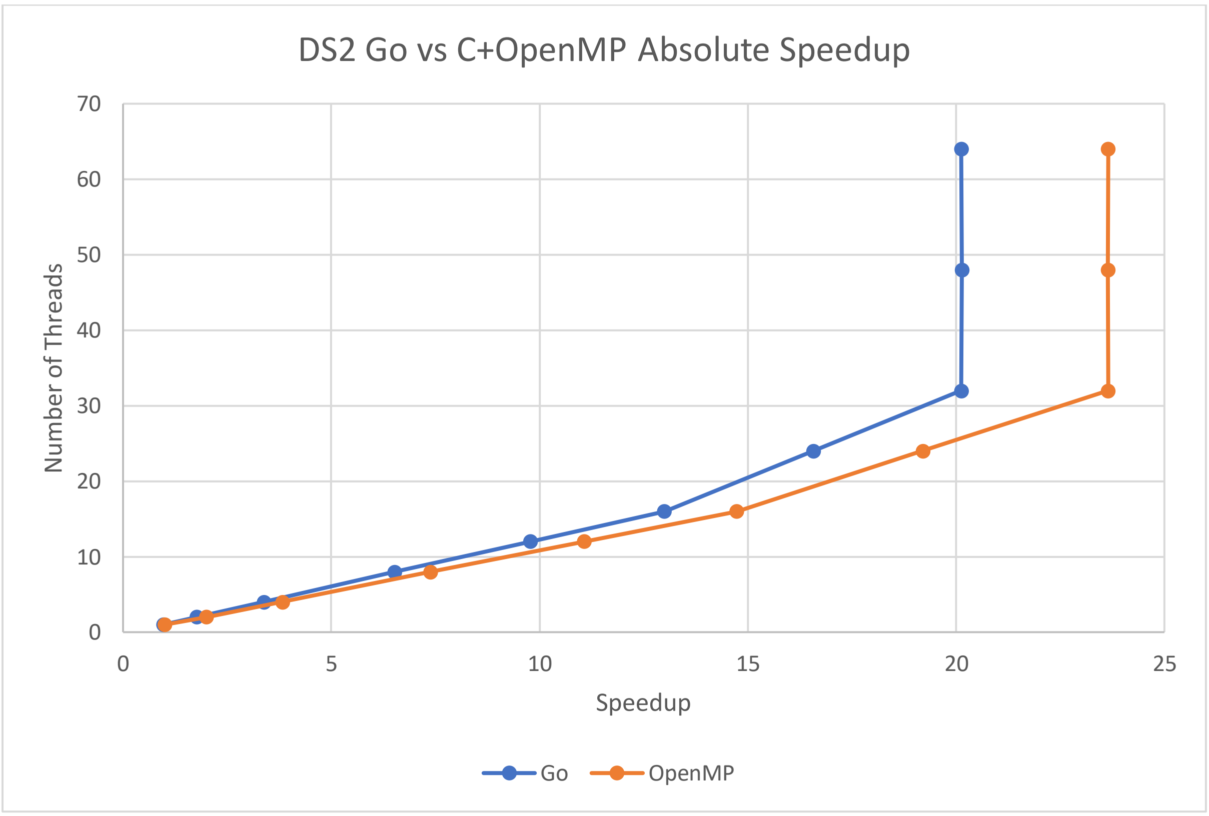

Figure 2.8: Comparing C+OpenMP and Go Absolute Speedups for DS2

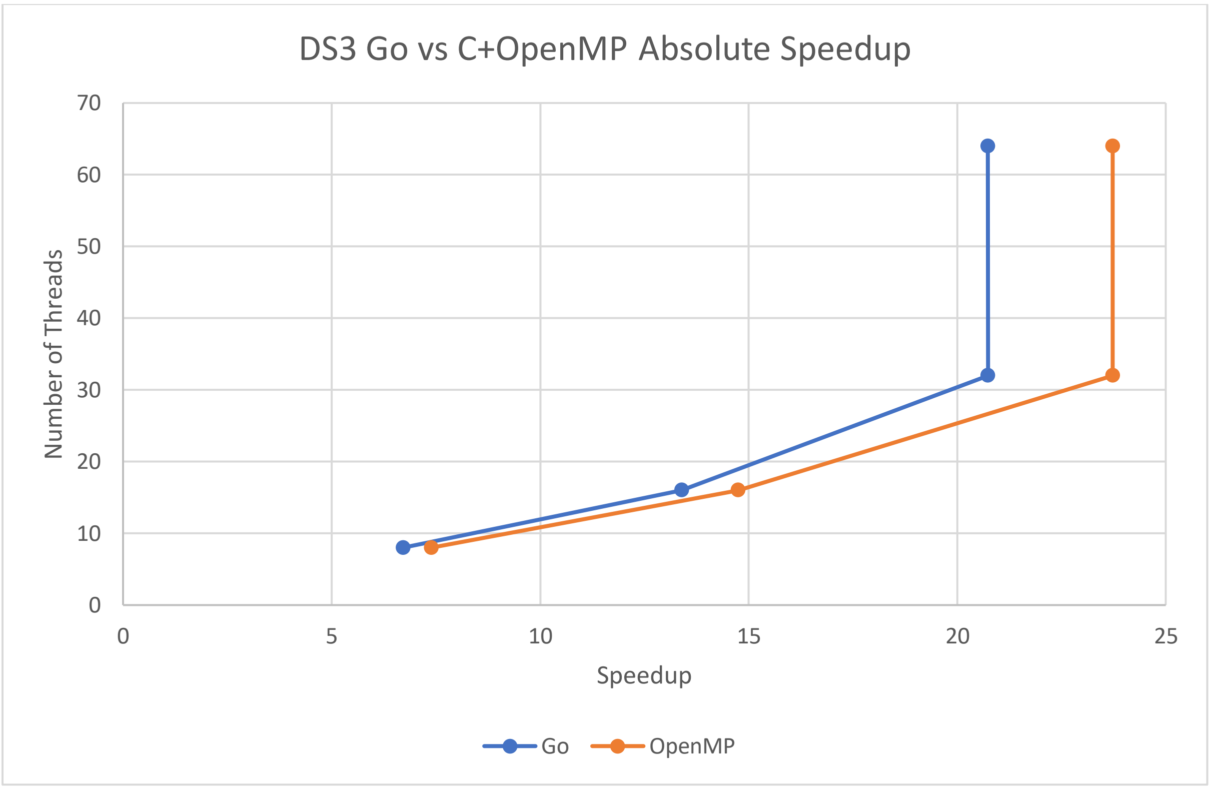

Figure 2.9: Comparing C+OpenMP and Go Absolute Speedups for DS3

Section 2.3

Language	Sequential Runtime (s)	Best Parallel Runtime (s)	Best Speedup	No. Threads
Go (DS1)	15.91639	0.878035	19.93095	32
C+OpenMP (DS1)	15.22271	0.822	23.44398	32
Go (DS2)	67.95405428	3.376901	20.1232	32
C+OpenMP (DS2)	65.28168	2.76	23.65278	32
Go (DS3)	294.5406	14.206	20.73353	32
C+OpenMP (DS3)	278.581	11.741	23.7272	32

Figure 2.10: Comparing C+OpenMP and Go Parallel Runtimes and Speedups for all data sets

Section 2.4

Looking at the graphs, the runtimes for both Go and C+OpenMP seem very similar, with C+OpenMP being slightly faster on average in all cases. Looking at figure 2.10 however, we can see that C+OpenMP clearly outperforms Go for best parallel runtimes, which is more apparent in DS2 and DS3. This is even true for sequential runtimes.

Looking at the numbers for speedups, the results for DS1, DS2 and DS3 are very similar, with 32 threads providing ~20 times speedup on average for Go and ~24 times speedup on average for C+OpenMP. C+OpenMP consistently shows a significantly higher speedup than Go, which is immediately obvious starting threads>1. The graphs are mostly linear and inversely proportional (runtime is almost exactly half when number of threads are doubled), which suggests the parallelism is ideal, until it plateaus at 32 threads. By running omp_get_max_threads I found that 32 was the maximum number of cores in the gpg-nodes cluster (There are 32 CPUs) so this makes sense, as increasing the number of threads at this point would result in CPUs needing to context switch to give time to multiple threads.

It makes sense that C+OpenMP would be faster than Go because Go includes a garbage collector that will increase runtime, especially when handling a lot of data in a loop.

Section 3 Programming Model Comparison

The Go and OpenMP models each have their advantages. Goroutines have very little overhead (minimum size 8kb) and fast startup times, and hence you can run many more of them (millions, if desired) than traditional threads. Goroutines are also simpler to manage, as mutexes are avoided and they will scale according to the complexity of the work. Goroutines are best used if development time and memory usage is a concern. I also found goroutines easier to grasp than OpenMP, as it is written in a traditional coding style. OpenMP with its one-line pragma comments was a little more unusual, so for programmers unfamiliar with parallelism I would recommend goroutines.

However, if peak performance is desired, then C+OpenMP should be used. It can be seen from my previous results in the totientRange application that C+OpenMP outperforms Go. The pragma paradigm in OpenMP is much simpler and easier to setup than pthreads and provides powerful tools for reductions. There are also scheduling styles (e.g. guided) that can adapt to the work being done and improve the efficiency of the runtimes.

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 then decided to use integer division to divide the dataset into a variable named chunk_size (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.

However, I ultimately changed this to a round-robin style where goroutines are spawned and only deal with one number from a channel containing all numbers to be processed. This proved to be significantly faster than the original method (as seen in Appendix B).

My biggest challenge with OpenMP was figuring out how to tune it to run faster (I gathered that my runtimes weren't ideal from the original speedup graphs, as the results were not inversely proportional). This is discussed further in Appendix A.

Appendix A C+OpenMP TotientRange Program

Code:

```c
// TotientRangePar.c - Parallel Euler Totient Function (C Version)
// compile: gcc -Wall -O2 -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 se
t 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)
```

```c
{
  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 schedule(guided) reduction(+: sum) num_threads(n_th
reads)
    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 Pragma-based Shared Memory paradigm (e.g. using shared variables and reduction for loops).

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 these methods resulted in similar or slower runtimes than simply using one pragma.

What did improve runtimes noticeably was changing the schedule parameter of the pragma. By default, it takes the "static" argument, which splits chunks into even sizes. Because my original Go program did something similar, I tried to find a way to improve the runtimes in OpenMP. I found that setting schedule to "guided" which starts chunk_size high then decreases it for later threads. This reduces runtime since higher numbers take longer to compute (explained in Appendix B).

The runtimes stop increasing after 32 threads, because the gpgnode cluster "only" has 32 CPUs. I tried adding multiple parallel sections to see if that could increase performance with threads > CPUs but all methods resulted in the same runtimes.

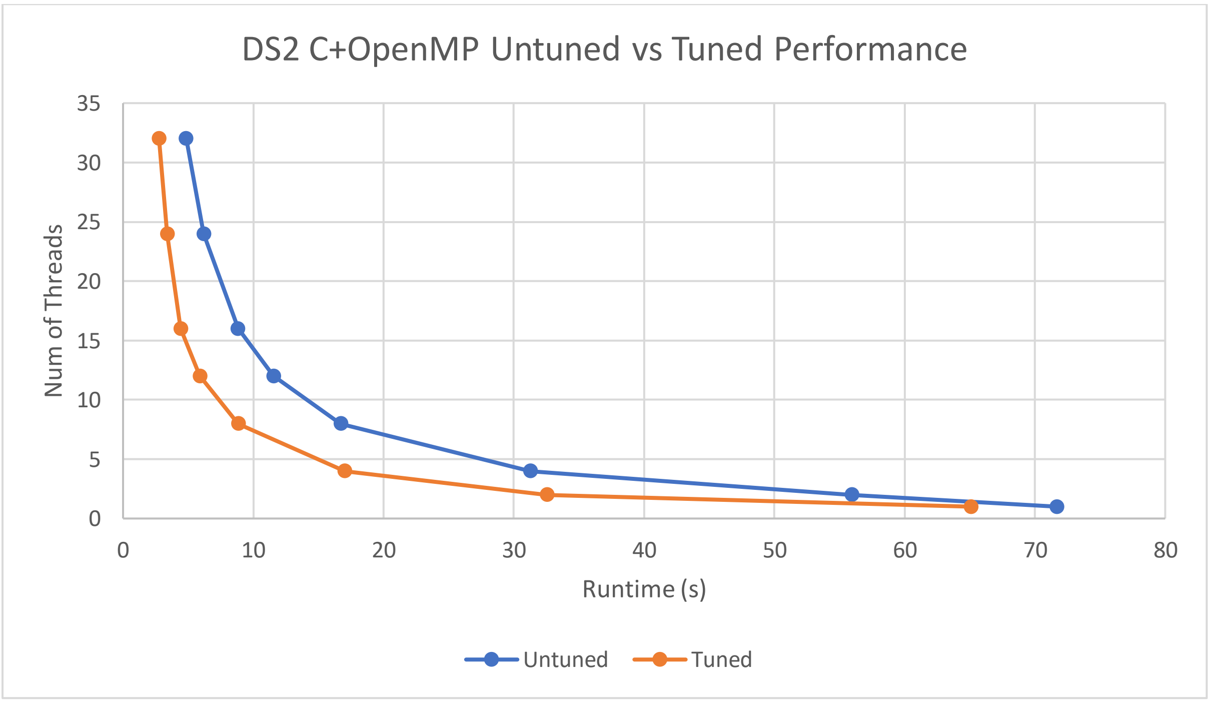

Figure 3.1: Comparing C+OpenMP runtime performance before and after tuning with DS2

Appendix B Go TotientRange Program

Code:

```go
// 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 Trinde
r,
// Nathan Charles, Hans-Wolfgang Loidl and Colin Runciman

package main

import (
    "fmt"
    "os"
    "runtime"
    "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
```

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

// sequential euler function

func euler_seq(n int64) int64 {
    var length, i int64

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

// 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(value int64, ch chan int64, wg *sync.WaitGroup) {
    var length, i int64

    length = 0
    for i = 1; i < value; i++ {
        if relprime(i, value) {
            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
    sum = 0

    // If more than 1 core is being used, utilise parallelism, otherwise run s
equentially (with 1 core in this program, sequential is faster)
```

```go
    if cores > 1 {
        chIn := make(chan int64, 100000)
        chSum := make(chan int64, 100000)
        var goroutines int64 = cores
        runtime.GOMAXPROCS(int(goroutines))
        var wg sync.WaitGroup // using a waitgroup to close the channel after all threads are completed

        // add all numbers in range of lower and upper to a channel
        for i = lower; i <= upper; i++ {
            chIn <- i
        }
        close(chIn)

        // repeatedly spawn goroutines to take 1 entry in chIn and calculate its totient, which is then sent onto channel chSum
        for value := range chIn {
            wg.Add(1)
            go euler(value, chSum, &wg)
        }
        wg.Wait()
        close(chSum)

        // sum all values in channel chSum
        for value := range chSum {
            sum += value
        }
    } else {
        for i = lower; i <= upper; i++ {
            sum = sum + euler_seq(i)
        }
    }
    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 and number of cores 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", sumTotie
nt(lower, upper, cores))

    // Record the elapsed time
    t := time.Now()
    elapsed := t.Sub(start)
    fmt.Println("Elapsed time", elapsed)
}
```

Uses a Message Passing paradigm (e.g. creating a channel with send and receives).

Originally my program didn't provide a large reduction in runtime because of how it separated work into chunks. My program split work into 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 would require heavy tuning to be optimal.

Instead, I changed my program to disregard chunk_size and instead spawn an arbitrary number of goroutines that only compute 1 number each, until all numbers are processed. This works as at the start of the program, all numbers are sent to a separate channel, and the goroutines receive one number from the channel to compute. This significantly improved runtimes compared to my original method, and reaches ideal parallelism based on absolute speedups (double the threads = half the runtime)

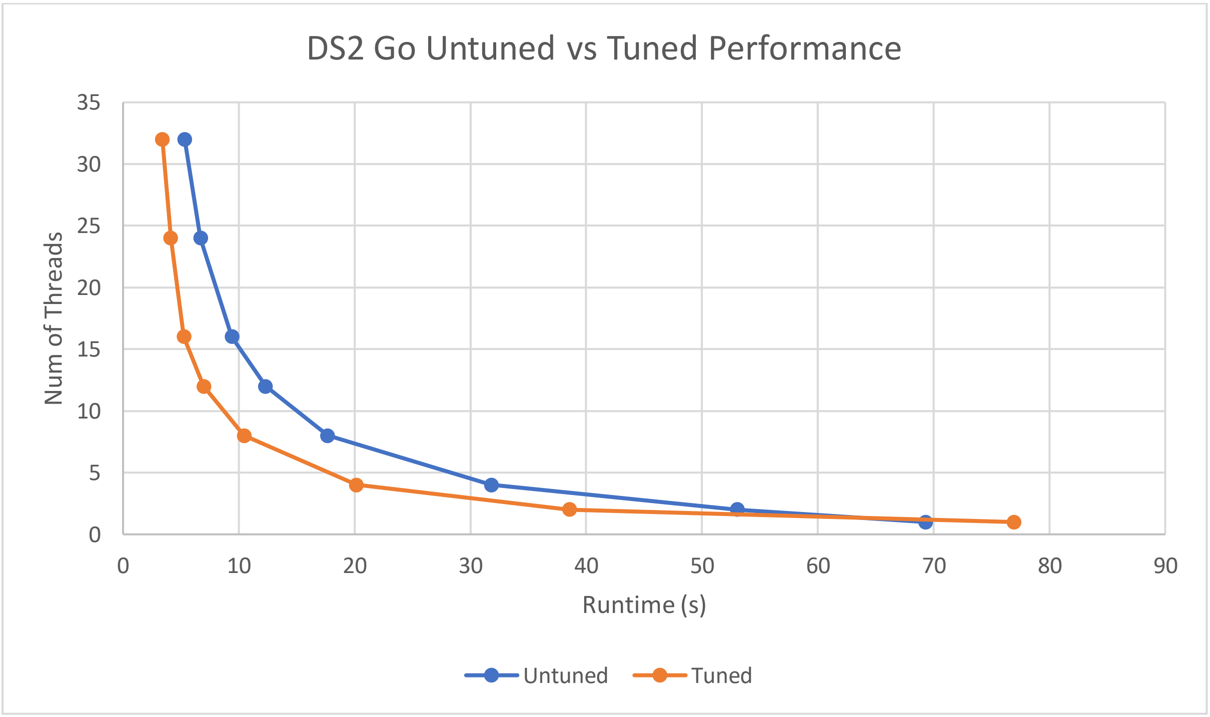

Figure 3.2: Comparing Go runtime performance before and after tuning with DS2

However, as you can see from the graph above, this method also increased the runtime for 1 thread. To improve that, I added an if statement that would run the program sequentially if only 1 core was set by runtime.GOMAXPROCS. The result is my final Go program.