Comparing Parallel and Distributed Programming Models:

Coursework Stage 2

Max Kirker Burton, 2260452b

Section 1 Comparative Sequential Performance

Data Set	Go Runtimes (s)	C Runtimes (s)	Erlang Runtimes (s)
DS1	15.91639141	15.21514	27.560312
DS2	67.95405428	65.28168	118.87842

Figure 1.1: Comparing C, Go and Erlang 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). Erlang on the other hand is much slower than Go or C, which suggests this problem cannot be done efficiently by a functional language. CPU bound problems are also likely slower in Erlang, and the typical solution is to push the problem to another language. Erlang is also designed for reliability and communications, and not tuned for performance (it prioritises low latency over high throughput, which is the opposite for Go and C).

Section 2 Two Worker Totient Range Erlang Problem: totientrange2Workers

Ranges	Runtime (s)
(1, 4000)	1.386345
(1, 15000)	21.413966

Figure 1.2: Erlang runtime

Input: {1, 4000}

```
1> c(totientrange2Workers).
{ok,totientrange2Workers}
2> totientrange2Workers:startServer().
true
3> server ! {range, 1, 4000}.
Worker: computing range 1 to 2000
Worker: computing range 2001 to 4000
{range,1,4000}
Server: Received Sum: 1216588
Server: Received Sum: 3647014
Server: Sum of totients: 4863602
Time taken in Secs, MicroSecs 1 386345
Worker: finished
Worker: finished
4>
```

Input: {1, 15000}

```
1> c(totientrange2Workers).
{ok,totientrange2Workers}
2> totientrange2Workers:startServer().
true
3> server ! {range, 1, 15000}.
Worker: computing range 1 to 7500
Worker: computing range 7501 to 15000
{range,1,15000}
Server: Received Sum: 17099412
Server: Received Sum: 51294904
Server: Sum of totients: 68394316
Time taken in Secs, MicroSecs 21 413966
Worker: finished
Worker: finished
4>
```

Section 3 Multi Worker Erlang Totient Range Problem: totientrangeNWorkers

Ranges + Number of Workers	Runtime (s)
(1, 4000, 4)	0. 864819
(1, 15000, 6)	9.259248

Input: {1, 4000, 4}

```
1> c(totientrangeNWorkers).
{ok,totientrangeNWorkers}
2> totientrangeNWorkers:startServer().
true
3> server ! {range, 1, 4000, 4}.
Worker: started
Worker: started
Worker: started
Worker: started
{range,1,4000,4}
Worker: computing range 1 to 1000
Worker: computing range 1001 to 2000
Worker: computing range 2001 to 3000
Worker: computing range 3001 to 4000
Server: Received Sum: 304192
Worker: finished
Server: Received Sum: 912396
Worker: finished
Server: Received Sum: 1519600
Worker: finished
Server: Received Sum: 2127414
Server: Sum of totients: 4863602
Time taken in Secs, MicroSecs 0 864819
Worker: finished
4>
```

Input: {1, 15000, 6}

```
1> c(totientrangeNWorkers).
{ok,totientrangeNWorkers}
2> totientrangeNWorkers:startServer().
true
3> server ! {range, 1, 15000, 6}.
Worker: started
Worker: started
Worker: started
Worker: started
Worker: started
Worker: started
{range,1,15000,6}
Worker: computing range 1 to 2500
Worker: computing range 2501 to 5000
Worker: computing range 5001 to 7500
Worker: computing range 7501 to 10000
Worker: computing range 10001 to 12500
Worker: computing range 12501 to 15000
Server: Received Sum: 1899878
Worker: finished
Server: Received Sum: 5700580
Worker: finished
Server: Received Sum: 9498954
Worker: finished
Server: Received Sum: 13298074
Worker: finished
Server: Received Sum: 17100872
Worker: finished
Server: Received Sum: 20895958
Server: Sum of totients: 68394316
Worker: finished
Time taken in Secs, MicroSecs 9 259248
4>
```

Section 4 Reliable Multi Worker Erlang Totient Range Erlang Problem: totientrangeNWorkersReliable

Number of Workers + Number of Victims	Runtime (s)
(4, 3)	14.367165
(8, 6)	9.82087
(12, 10)	6.610769

Input: {4, 3}

```
1> c(totientrangeNWorkersReliable).
{ok,totientrangeNWorkersReliable}
2> totientrangeNWorkersReliable:testRobust(4,3).
Watcher: Watching Worker worker1
Watcher: Watching Worker worker2
Watcher: Watching Worker worker3
Watcher: Watching Worker worker4
Worker: computing range 1 to 3750
Worker: computing range 3751 to 7500
Worker: computing range 7501 to 11250
Worker: computing range 11251 to 15000
workerChaos killing worker3
Watcher: Watching Worker worker3
Worker: computing range 7501 to 11250
workerChaos killing worker3
Watcher: Watching Worker worker3
Worker: computing range 7501 to 11250
workerChaos killing worker4
Watcher: Watching Worker worker4
Worker: computing range 11251 to 15000
[true,true,true]
Server: Received Sum: 4275174
Server: Received Sum: 12824238
Server: Received Sum: 21371600
Server: Received Sum: 29923304
Server: Sum of totients: 68394316
Time taken in Secs, MicroSecs 14 367165
3>
```

Input: {8, 6}

```
1> c(totientrangeNWorkersReliable).
{ok,totientrangeNWorkersReliable}
2> totientrangeNWorkersReliable:testRobust(8,6).
Watcher: Watching Worker worker1
Watcher: Watching Worker worker2
Watcher: Watching Worker worker3
Watcher: Watching Worker worker4
Watcher: Watching Worker worker5
Watcher: Watching Worker worker6
Watcher: Watching Worker worker7
Watcher: Watching Worker worker8
Worker: computing range 1 to 1875
Worker: computing range 1876 to 3750
Worker: computing range 3751 to 5625
Worker: computing range 5626 to 7500
Worker: computing range 7501 to 9375
Worker: computing range 9376 to 11250
Worker: computing range 11251 to 13125
Worker: computing range 13126 to 15000
Server: Received Sum: 1069230
workerChaos killing worker4
Watcher: Watching Worker worker4
Worker: computing range 5626 to 7500
workerChaos killing worker5
Watcher: Watching Worker worker5
Worker: computing range 7501 to 9375
Server: Received Sum: 3205944
workerChaos killing worker4
Watcher: Watching Worker worker4
Worker: computing range 5626 to 7500
workerChaos killing worker5
Watcher: Watching Worker worker5
Worker: computing range 7501 to 9375
Server: Received Sum: 5343712
workerChaos killing worker3
workerChaos already dead: worker3
workerChaos killing worker7
Watcher: Watching Worker worker7
Worker: computing range 11251 to 13125
[true,true,true,true,ok,true]
Server: Received Sum: 7480526
Server: Received Sum: 11754000
Server: Received Sum: 9617600
Server: Received Sum: 16029232
Server: Received Sum: 13894072
Server: Sum of totients: 68394316
```

```
Time taken in Secs, MicroSecs 9 82087
3>
```

Input: {12, 10}

```
1> c(totientrangeNWorkersReliable).
{ok,totientrangeNWorkersReliable}
2> totientrangeNWorkersReliable:testRobust(12,10).
Watcher: Watching Worker worker1
Watcher: Watching Worker worker2
Watcher: Watching Worker worker3
Watcher: Watching Worker worker4
Watcher: Watching Worker worker5
Watcher: Watching Worker worker6
Watcher: Watching Worker worker7
Watcher: Watching Worker worker8
Watcher: Watching Worker worker9
Watcher: Watching Worker worker10
Watcher: Watching Worker worker11
Watcher: Watching Worker worker12
Worker: computing range 1 to 1250
Worker: computing range 1251 to 2500
Worker: computing range 2501 to 3750
Worker: computing range 3751 to 5000
Worker: computing range 5001 to 6250
Worker: computing range 6251 to 7500
Worker: computing range 7501 to 8750
Worker: computing range 8751 to 10000
Worker: computing range 10001 to 11250
Worker: computing range 11251 to 12500
Worker: computing range 12501 to 13750
Worker: computing range 13751 to 15000
Server: Received Sum: 475306
workerChaos killing worker12
Watcher: Watching Worker worker12
Worker: computing range 13751 to 15000
Server: Received Sum: 1424572
workerChaos killing worker5
Watcher: Watching Worker worker5
Worker: computing range 5001 to 6250
Server: Received Sum: 2375296
workerChaos killing worker2
workerChaos already dead: worker2
Server: Received Sum: 3325284
workerChaos killing worker7
Watcher: Watching Worker worker7
Worker: computing range 7501 to 8750
Server: Received Sum: 5224570
```

```
workerChaos killing worker7
Watcher: Watching Worker worker7
Worker: computing range 7501 to 8750
Server: Received Sum: 4274384
workerChaos killing worker6
workerChaos already dead: worker6
Server: Received Sum: 7122972
workerChaos killing worker8
workerChaos already dead: worker8
Server: Received Sum: 8073526
workerChaos killing worker7
Watcher: Watching Worker worker7
Worker: computing range 7501 to 8750
Server: Received Sum: 9027346
workerChaos killing worker9
workerChaos already dead: worker9
Server: Received Sum: 9970596
workerChaos killing worker9
workerChaos already dead: worker9
[true,true,ok,true,true,ok,ok,true,ok,ok]
Server: Received Sum: 10925362
Server: Received Sum: 6175102
Server: Sum of totients: 68394316
Time taken in Secs, MicroSecs 6 610769
2>
```

Section 5 Comparative Parallel Performance Measurements

Section 5.1 Runtimes

Threads	Go	OpenMP	Erlang
Sequential	15.91639141	15.21514	27.560312
1	16.35538	15.171	27.64324
2	9.066147	7.592	21.84232
4	4.740579	3.973	12.90331
8	2.461242	2.06	7.674511
12	1.647562	1.376	5.155668
16	1.264213	1.041	3.84141
24	0.972845	0.795	2.711483
32	0.80053	0.649	2.310508
48	0.801696	0.651	2.15471
64	0.8027	0.653	1.877161

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

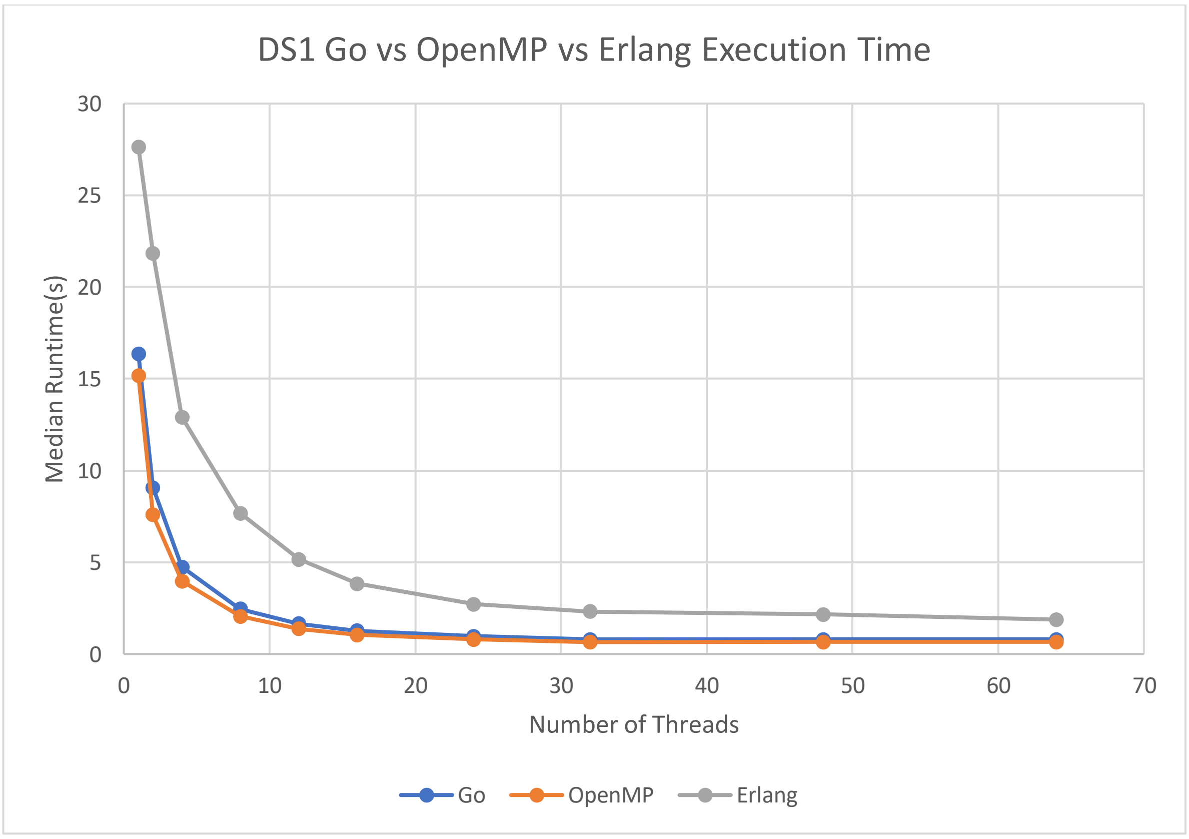

Figure 2.2: Comparing C+OpenMP, Go and Erlang Parallel Runtimes for DS1, plotted

Threads	Go	OpenMP	Erlang
Sequential	67.95405428	65.28168	118.87842
1	69.91227	65.085	117.7675
2	38.36811	32.562	89.97294
4	20.0648	17.042	56.27548
8	10.40659	8.839	31.34113
12	6.941679	5.894	21.14163
16	5.227036	4.43	16.74621
24	4.097904	3.399	12.12922
32	3.376901	2.76	9.570727
48	3.374534	2.761	7.924492
64	3.377058	2.76	7.197233

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

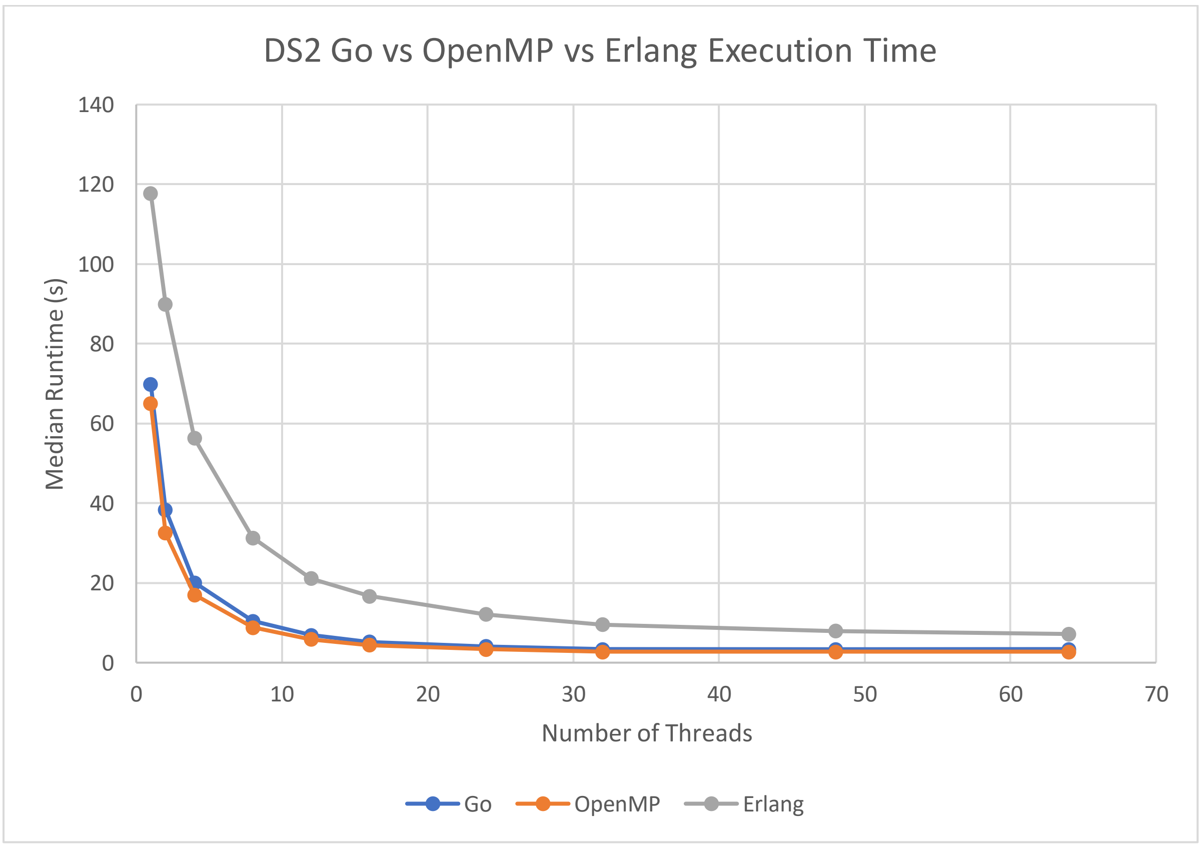

Figure 2.4: Comparing C+OpenMP, Go and Erlang Parallel Runtimes for DS2, plotted

Threads	Go	OpenMP	Erlang
8	43.90842	37.725	129.5024
16	22.00349	18.89	69.37089
32	14.206	11.741	42.37109
64	14.20571	11.741	32.1379

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

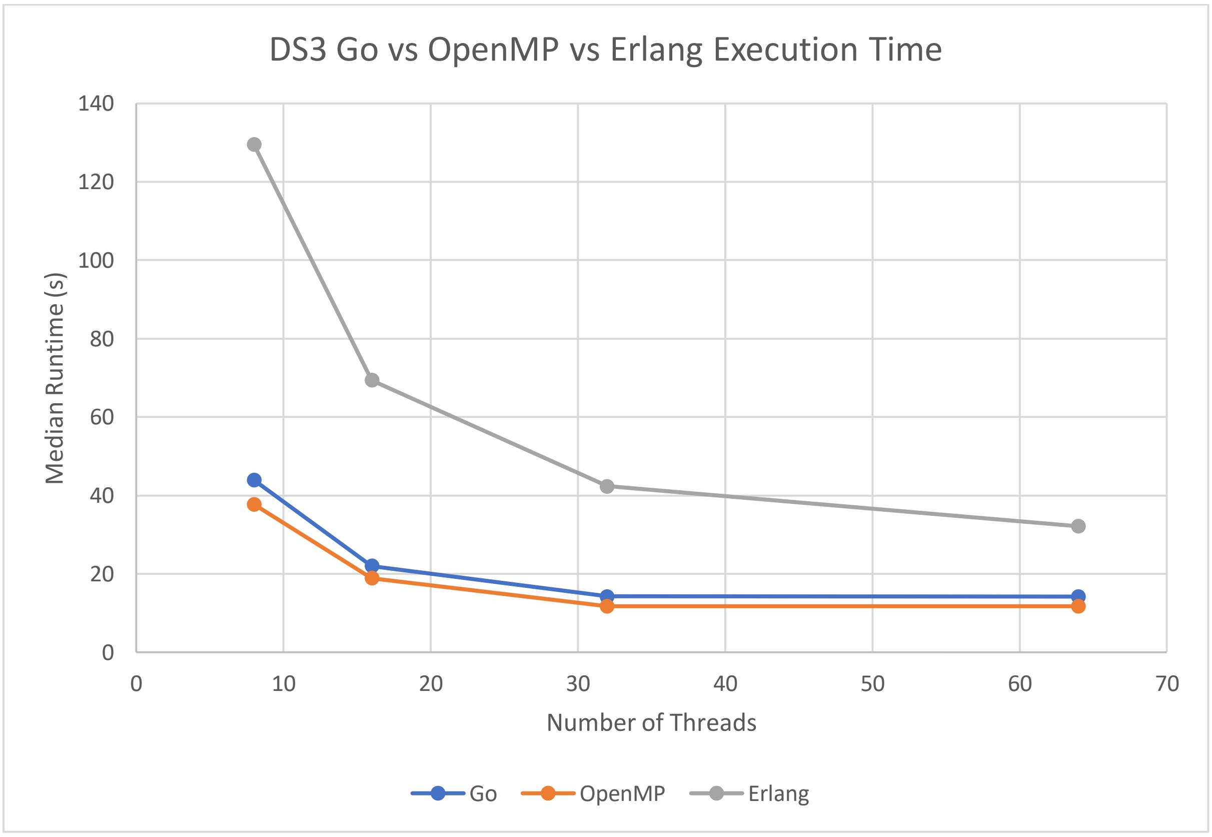

Figure 2.6: Comparing C+OpenMP, Go and Erlang Parallel Runtimes for DS3, plotted

Section 5.2 Speedups

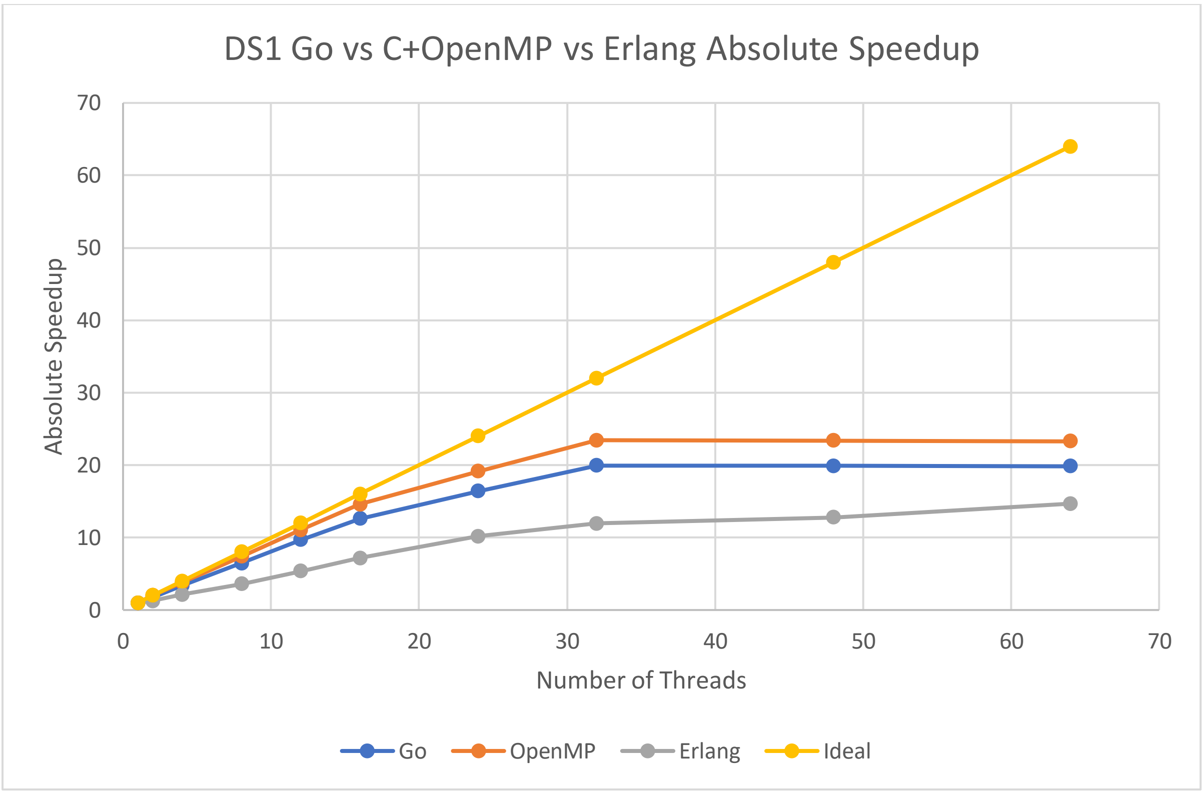

Figure 2.7: Comparing C+OpenMP, Go and Erlang Absolute Speedups for DS1

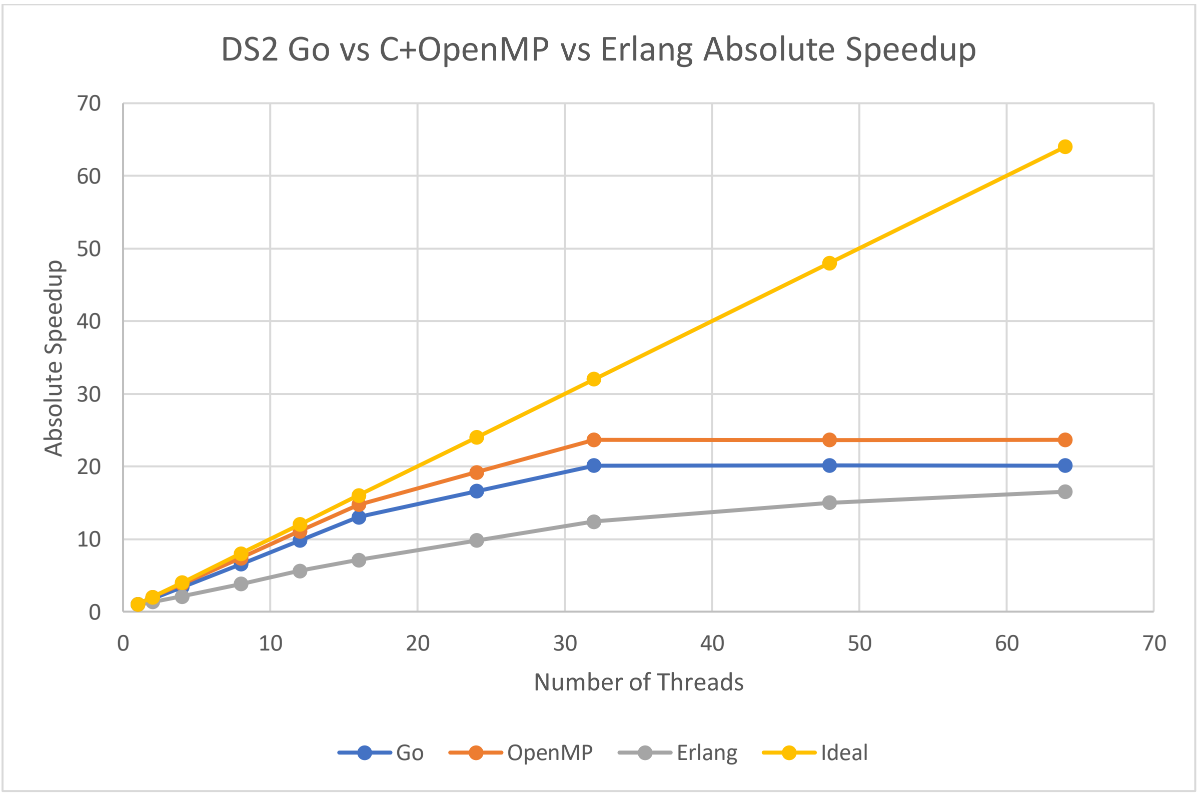

Figure 2.8: Comparing C+OpenMP, Go and Erlang Absolute Speedups for DS2

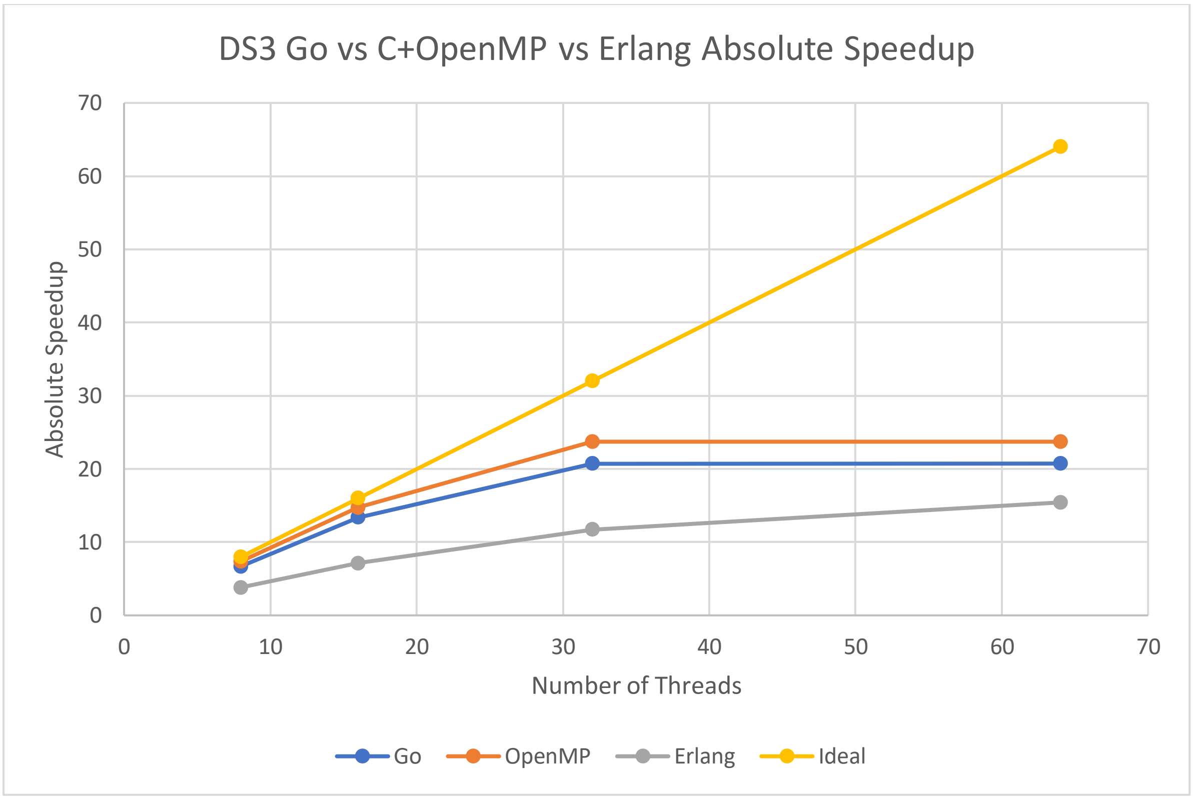

Figure 2.9: Comparing C+OpenMP, Go and Erlang Absolute Speedups for DS3

Section 5.3

Language	Sequential Runtime (s)	Best Parallel Runtime (s)	Best Speedup	No. Threads
DS1				
Go	15.91639	0.878035	19.93095	32
C+OpenMP	15.22271	0.822	23.44398	32
Erlang	27.56031	1.877161	14.68191	64
DS2				
Go	67.95405428	3.376901	20.1232	32
C+OpenMP	65.28168	2.76	23.65278	32
Erlang	118.8784	7.197233	16.51724	64
DS3				
Go	294.5406	14.206	20.73353	32
C+OpenMP	278.581	11.741	23.7272	32
Erlang	495.9446	32.1379	15.43177	64

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

Section 5.4

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.

Erlang is noticeably slower than both C+OpenMP and Go. This is likely because those languages are designed to increase throughput, and hence perform better on CPU bound tasks (such as summing totients). Erlang on the other hand priorities low latency over high throughput, and hence is understandably slower.

The Erlang code could be tweaked to improve its speedup (just like how C+OpenMP and Go have been tuned). This would mean it's runtime would converge at 32 threads just like the other two languages. But since the code is currently not tuned, using more workers will decrease the runtime, as more workers equals smaller chunks, and the program execution time is capped by the final worker (since it has the largest numbers to sum).

Section 6 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.

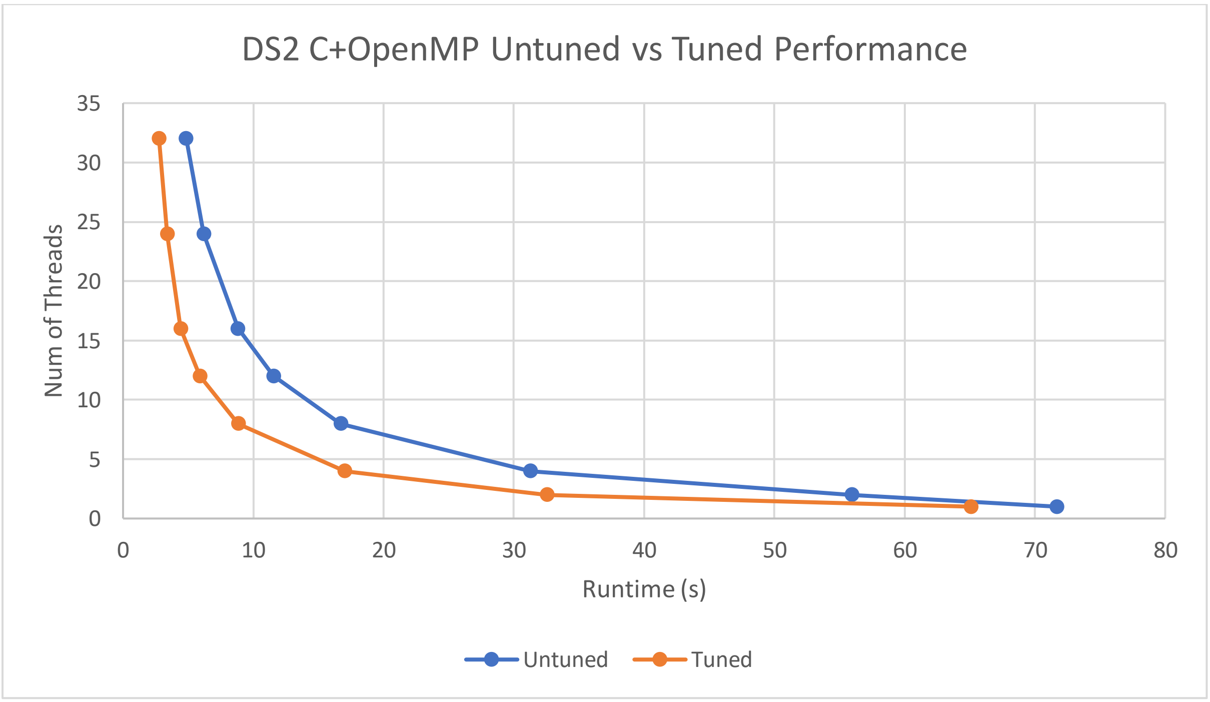

Figure 3.1: Comparing C+OpenMP runtime performance before and after tuning with DS2

Appendix B Go TotientRange Program

Code:

```
// 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 i
ts 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 an
d 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)

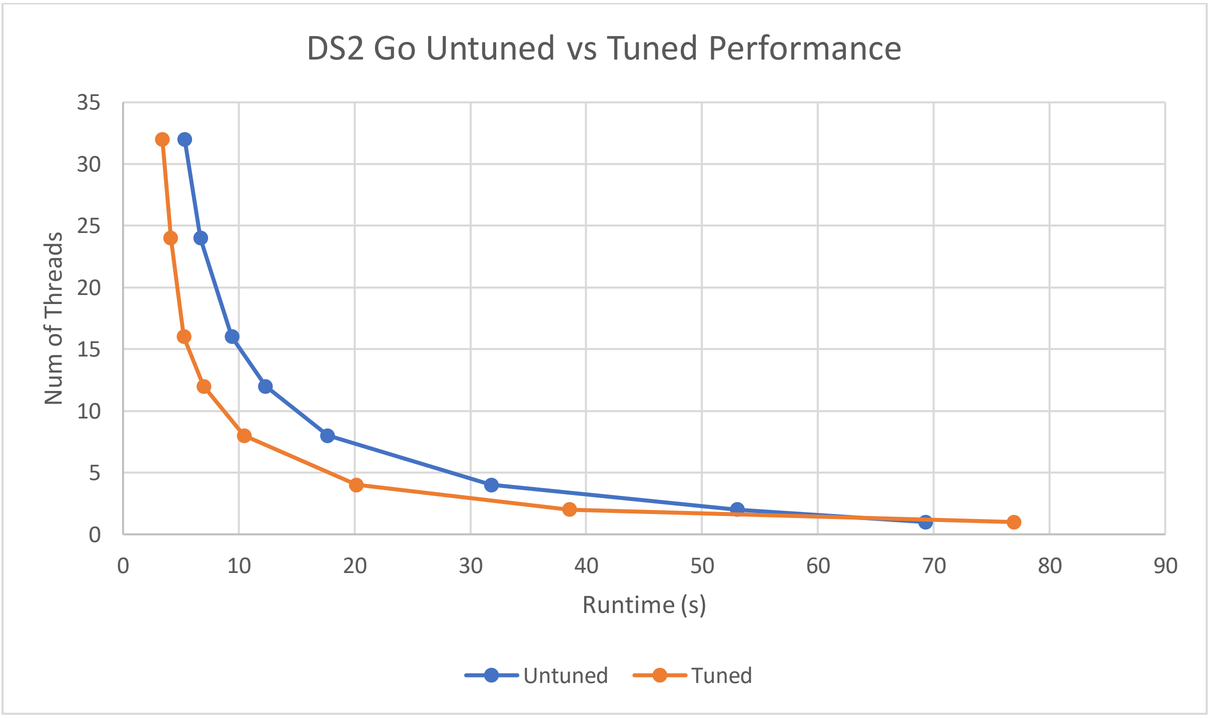

Figure 3.2: Comparing Go runtime performance before and after tuning with DS2

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.

Appendix C Erlang totientrange2Workers Program

```
-module(totientrange2Workers).
-export([hcf/2,
   relprime/2,
   euler/1,
     startServer/0,
     server/4,
     totientWorker/0
  ]).

%% totientrange2Workers.erl - Parellel Euler Totient Function (Erlang Version)
%% compile from the shell: >c(totientrange2Workers).
%% run from the shell:     >totientrange2Workers:startServer().
%%                         >server ! {range, x, y}.

%% Max Kirker Burton 2260452b

%% This program calculates the sum of the totients between a lower and an
%% upper limit. It is based on earlier work by: Nathan Charles,
%% Hans-Wolfgang Loidl and Colin Runciman

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

%% hcf x 0 = x
%% hcf x y = hcf y (rem x y)

hcf(X,0) -> X;
hcf(X,Y) -> hcf(Y,X rem Y).

%% relprime x y = hcf x y == 1

relprime(X,Y) ->
  V = hcf(X,Y),
  if
    V == 1
      -> true;
    true
      -> false
  end.

%%euler n = length (filter (relprime n) (mkList n))

euler(N) ->
  RelprimeN = fun(Y) -> relprime(N,Y) end,
  length (lists:filter(RelprimeN,(lists:seq(1,N)))).

%% Take completion timestamp, and print elapsed time
```

```erlang
printElapsed(S,US) ->
  {_, S2, US2} = os:timestamp(),
                       %% Adjust Seconds if completion Microsecs > start Micro
secs
  if
    US2-US < 0 ->
      S3 = S2-1,
      US3 = US2+1000000;
    true ->
      S3 = S2,
      US3 = US2
  end,
  io:format("Time taken in Secs, MicroSecs ~p ~p~n",[S3-S,US3-US]).

%% We pass in these variables to keep their value over successive receive call
s
server(Total, Count, S, US) ->
    receive
        %% Check if all workers have finished, otherwise continue
        {finished, Sum} when Count < 1 ->
            io:format("Server: Received Sum: ~p~n", [Sum]),
            server(Total+Sum, Count+1, S, US);
        {finished, Sum} ->
            io:format("Server: Received Sum: ~p~n", [Sum]),
            io:format("Server: Sum of totients: ~p~n", [Total+Sum]),
            worker1 ! finished,
            worker2 ! finished,
            printElapsed(S,US);
        {range, Lower, Upper} ->
            {_, Sec, USec} = os:timestamp(),
            register(worker1, spawn(totientrange2Workers, totientWorker, [])),
            worker1 ! {range, Lower, trunc(Upper/2)},
            register(worker2, spawn(totientrange2Workers, totientWorker, [])),
            worker2 ! {range, trunc(Upper/2 + 1), Upper},
            server(Total, Count, Sec, USec)
    end.

%% calculate sumTotient on a subset of the dataset
totientWorker() ->
    receive
        finished ->
            io:format("Worker: finished~n", []);
        {range, Lower, Upper} ->
            io:format("Worker: computing range ~p to ~p~n", [Lower, Upper]),
            Sum = lists:sum(lists:map(fun euler/1,lists:seq(Lower, Upper))),
            server ! {finished, Sum},
            totientWorker()
    end.
```

```
%%start the server process
startServer() ->
    register(server, spawn(totientrange2Workers, server, [0, 0, 0, 0])).
```

Appendix D Erlang totientrangeNWorkers Program

```
-module(totientrangeNWorkers).
-export([hcf/2,
     relprime/2,
     euler/1,
     startServer/0,
     workerName/1,
     registerWorker/3,
     server/7,
     totientWorker/0
    ]).

%% totientrangeNWorkers.erl - Parallel Euler Totient Function (Erlang Version)
%% compile from the shell: >c(totientrangeNWorkers).
%% run from the shell:     >totientrangeNWorkers:startServer().
%%                         >server ! {range, x, y, N}.

%% Max Kirker Burton 2260452b

%% This program calculates the sum of the totients between a lower and an
%% upper limit. It is based on earlier work by: Nathan Charles,
%% Hans-Wolfgang Loidl and Colin Runciman

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

%% hcf x 0 = x
%% hcf x y = hcf y (rem x y)

hcf(X,0) -> X;
hcf(X,Y) -> hcf(Y,X rem Y).

%% relprime x y = hcf x y == 1

relprime(X,Y) ->
  V = hcf(X,Y),
  if
    V == 1
      -> true;
    true
      -> false
  end.

%%euler n = length (filter (relprime n) (mkList n))

euler(N) ->
  RelprimeN = fun(Y) -> relprime(N,Y) end,
  length (lists:filter(RelprimeN,(lists:seq(1,N)))).
```

```
%% Take completion timestamp, and print elapsed time

printElapsed(S,US) ->
  {_, S2, US2} = os:timestamp(),
                       %% Adjust Seconds if completion Microsecs > start Micro
secs
  if
    US2-US < 0 ->
      S3 = S2-1,
      US3 = US2+1000000;
    true ->
      S3 = S2,
      US3 = US2
  end,
  io:format("Time taken in Secs, MicroSecs ~p ~p~n",[S3-S,US3-US]).

workerName(Num) ->
    list_to_atom( "worker" ++ integer_to_list( Num )).

workerNames([]) ->
    0;
workerNames(Range) ->
    [Head | Rest] = Range,
    WorkerName = workerName(Head),
    server ! {names, WorkerName, Head},
    workerNames(Rest).

registerWorker(WorkerNum, Lower, Upper) ->
    register(WorkerNum, spawn(totientrangeNWorkers, totientWorker, [])),
    WorkerNum ! {range, Lower, Upper}.

%% We pass in these variables to keep their value over successive receive call
s
server(Total, Count, Lower, Upper, N_workers, S, US) ->
    receive
        %% Check if all workers have finished, otherwise continue
        {finished, Sum, Pid} when Count < (N_workers-1) ->
            io:format("Server: Received Sum: ~p~n", [Sum]),
            Pid ! finished,
            server(Total+Sum, Count+1, Lower, Upper, N_workers, S, US);
        {finished, Sum, Pid} ->
            io:format("Server: Received Sum: ~p~n", [Sum]),
            io:format("Server: Sum of totients: ~p~n", [Total+Sum]),
            Pid ! finished,
            printElapsed(S,US);
        %% if this is the first worker, start from Lower
        {names, WorkerNum, Num} when Num == 1 ->
            LocalLower = Lower,
            LocalUpper = Lower + trunc(((Upper - Lower)/N_workers) * (Num)),
            registerWorker(WorkerNum, LocalLower, LocalUpper),
            server(Total, Count, Lower, Upper, N_workers, S, US);
        %% if this is the last worker, set the Upper limit
        {names, WorkerNum, Num} when Num == N_workers ->
            LocalLower = 1 + Lower + trunc(((Upper - Lower)/N_workers) * (Num
- 1)),
            LocalUpper = Upper,
            registerWorker(WorkerNum, LocalLower, LocalUpper),
            server(Total, Count, Lower, Upper, N_workers, S, US);
        %% otherwise calculate an equal range
        {names, WorkerNum, Num} ->
            LocalLower = 1 + Lower + trunc(((Upper - Lower)/N_workers) * (Num
- 1)),
            LocalUpper = Lower + trunc(((Upper - Lower)/N_workers) * (Num)),
            registerWorker(WorkerNum, LocalLower, LocalUpper),
            server(Total, Count, Lower, Upper, N_workers, S, US);
        {range, L, U, N} ->
            {_, Sec, USec} = os:timestamp(),
            Range = lists:seq(1, N),
            workerNames(Range),
            server(Total, Count, L, U, N, Sec, USec)
    end.

%% calculate sumTotient on a subset of the dataset
totientWorker() ->
    receive
        finished ->
            io:format("Worker: finished~n", []);
        {range, Lower, Upper} ->
            io:format("Worker: started~n", []),
            io:format("Worker: computing range ~p to ~p~n", [Lower, Upper]),
            Sum = lists:sum(lists:map(fun euler/1,lists:seq(Lower, Upper))),
            server ! {finished, Sum, self()},
            totientWorker()
    end.



%%start the server process
startServer() ->
    register(server, spawn(totientrangeNWorkers, server, [0, 0, 0, 0, 0, 0, 0]
)).
```

Appendix E Erlang totientrangeNWorkersReliable Program

```
-module(totientrangeNWorkersReliable).
-export([hcf/2,
     relprime/2,
     euler/1,
     startServer/0,
     workerNames/1,
     workerName/1,
     registerWorker/3,
     server/7,
     totientWorker/0,
     watcher/3,
     workerChaos/2,
     testRobust/2
    ]).

%% totientrangeNWorkersReliable.erl - Parallel & Reliable Euler Totient Functi
on (Erlang Version)
%% compile from the shell: >c(totientrangeNWorkersReliable).
%% run from the shell:     >totientrangeNWorkersReliable:startServer().
%%                         >server ! {range, x, y, N}.

%% Max Kirker Burton 2260452b

%% This program calculates the sum of the totients between a lower and an
%% upper limit. It is based on earlier work by: Nathan Charles,
%% Hans-Wolfgang Loidl and Colin Runciman

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

%% hcf x 0 = x
%% hcf x y = hcf y (rem x y)

hcf(X,0) -> X;
hcf(X,Y) -> hcf(Y,X rem Y).

%% relprime x y = hcf x y == 1

relprime(X,Y) ->
  V = hcf(X,Y),
  if
    V == 1
      -> true;
    true
      -> false
  end.

%%euler n = length (filter (relprime n) (mkList n))
```

```
euler(N) ->
  RelprimeN = fun(Y) -> relprime(N,Y) end,
  length (lists:filter(RelprimeN,(lists:seq(1,N)))).

%% Take completion timestamp, and print elapsed time

printElapsed(S,US) ->
  {_, S2, US2} = os:timestamp(),
                       %% Adjust Seconds if completion Microsecs > start Micro
secs
  if
    US2-US < 0 ->
      S3 = S2-1,
      US3 = US2+1000000;
    true ->
      S3 = S2,
      US3 = US2
  end,
  io:format("Time taken in Secs, MicroSecs ~p ~p~n",[S3-S,US3-US]).

%% Generate a worker name with a given number
workerName(Num) ->
    list_to_atom( "worker" ++ integer_to_list( Num )).

%% Recursively generate worker names from a sequential list
workerNames([]) ->
    0;
workerNames(Range) ->
    [Head | Rest] = Range,
    WorkerName = workerName(Head),
    server ! {names, WorkerName, Head},
    workerNames(Rest).

%% register a worker, then pass a message to it
registerWorker(WorkerNum, Lower, Upper) ->
    register(WorkerNum, spawn_link(totientrangeNWorkersReliable, totientWorker
, [])),
    WorkerNum ! {range, Lower, Upper}.

%% We pass in these variables to keep their value over successive receive call
s
server(Total, Count, Lower, Upper, N_workers, S, US) ->
    receive
        %% Check if all workers have finished, otherwise continue
        {finished, Sum, Pid} when Count < (N_workers-1) ->
            io:format("Server: Received Sum: ~p~n", [Sum]),
            Pid ! finished,
```

```erlang
            server(Total+Sum, Count+1, Lower, Upper, N_workers, S, US);
        {finished, Sum, Pid} ->
            io:format("Server: Received Sum: ~p~n", [Sum]),
            io:format("Server: Sum of totients: ~p~n", [Total+Sum]),
            Pid ! finished,
            printElapsed(S,US);
        %% if this is the first worker, start from Lower
        {names, WorkerNum, Num} when Num == 1 ->
            LocalLower = Lower,
            LocalUpper = Lower + trunc(((Upper - Lower)/N_workers) * (Num)),
            spawn(totientrangeNWorkersReliable, watcher, [WorkerNum, LocalLowe
r, LocalUpper]),
            server(Total, Count, Lower, Upper, N_workers, S, US);
        %% if this is the last worker, set the Upper limit
        {names, WorkerNum, Num} when Num == N_workers ->
            LocalLower = 1 + Lower + trunc(((Upper - Lower)/N_workers) * (Num
- 1)),
            LocalUpper = Upper,
            spawn(totientrangeNWorkersReliable, watcher, [WorkerNum, LocalLowe
r, LocalUpper]),
            server(Total, Count, Lower, Upper, N_workers, S, US);
        %% otherwise calculate an equal range
        {names, WorkerNum, Num} ->
            LocalLower = 1 + Lower + trunc(((Upper - Lower)/N_workers) * (Num
- 1)),
            LocalUpper = Lower + trunc(((Upper - Lower)/N_workers) * (Num)),
            spawn(totientrangeNWorkersReliable, watcher, [WorkerNum, LocalLowe
r, LocalUpper]),
            server(Total, Count, Lower, Upper, N_workers, S, US);
        {range, L, U, N} ->
            {_, Sec, USec} = os:timestamp(),
            Range = lists:seq(1, N),
            workerNames(Range),
            server(Total, Count, L, U, N, Sec, USec)
    end.

%% If an exit message is received and the worker had not completed, trap it an
d respawn the linked worker
watcher(WorkerNum, LocalLower, LocalUpper) ->
    process_flag(trap_exit, true),
    registerWorker(WorkerNum, LocalLower, LocalUpper),
    io:format("Watcher: Watching Worker ~p~n", [WorkerNum]),
    receive
        Msg ->
            {_Message, _Pid, Reason} = Msg,
            if
                Reason == chaos ->
                    watcher(WorkerNum, LocalLower, LocalUpper);
                true ->
                    0
            end
    end.

%% calculate sumTotient on a subset of the dataset
totientWorker() ->
    receive
        finished ->
            0;
        {range, Lower, Upper} ->
            io:format("Worker: computing range ~p to ~p~n", [Lower, Upper]),
            Sum = lists:sum(lists:map(fun euler/1,lists:seq(Lower, Upper))),
            server ! {finished, Sum, self()},
            totientWorker()
    end.

workerChaos(NVictims, NWorkers) ->
    lists:map(
        fun( _ ) ->
            timer:sleep(500),   %% Sleep for .5s
                                %% Choose a random victim
            WorkerNum = rand:uniform(NWorkers),
            io:format("workerChaos killing ~p~n",
                      [workerName(WorkerNum)]),
            WorkerPid = whereis(workerName(WorkerNum)),
            if %% Check if victim is alive
                WorkerPid == undefined ->
                    io:format("workerChaos already dead: ~p~n",
                              [workerName(WorkerNum)]);
                true -> %% Kill Kill Kill
                    exit(whereis(workerName(WorkerNum)),chaos)
                end
            end,
            lists:seq( 1, NVictims ) ).

testRobust(NWorkers, NVictims) ->
    ServerPid = whereis(server),
    if %% Is server already running?
        ServerPid == undefined ->
            startServer();
        true ->
            ok
    end,
    server ! {range, 1, 15000, NWorkers},
    workerChaos(NVictims,NWorkers).

%%start the server process
```

```erlang
startServer() ->
    register(server, spawn(totientrangeNWorkersReliable, server, [0, 0, 0, 0,
0, 0, 0])).
```