Comparing Parallel and Distributed Programming Models: Coursework Stage 1: Performance and Programmability Of Shared-memory Parallel Models

Iain McLean
2308499m

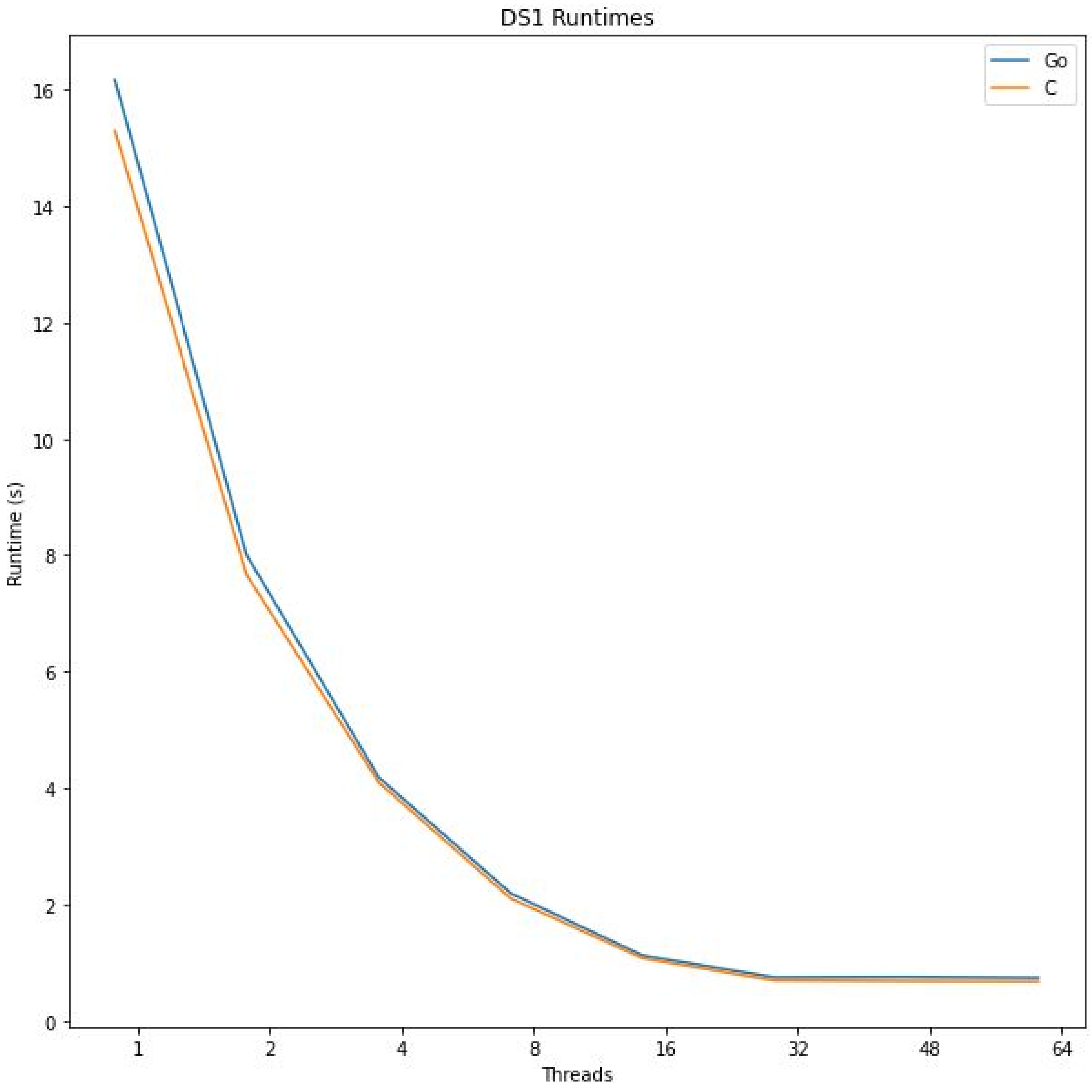
1 Comparative Sequential Performance

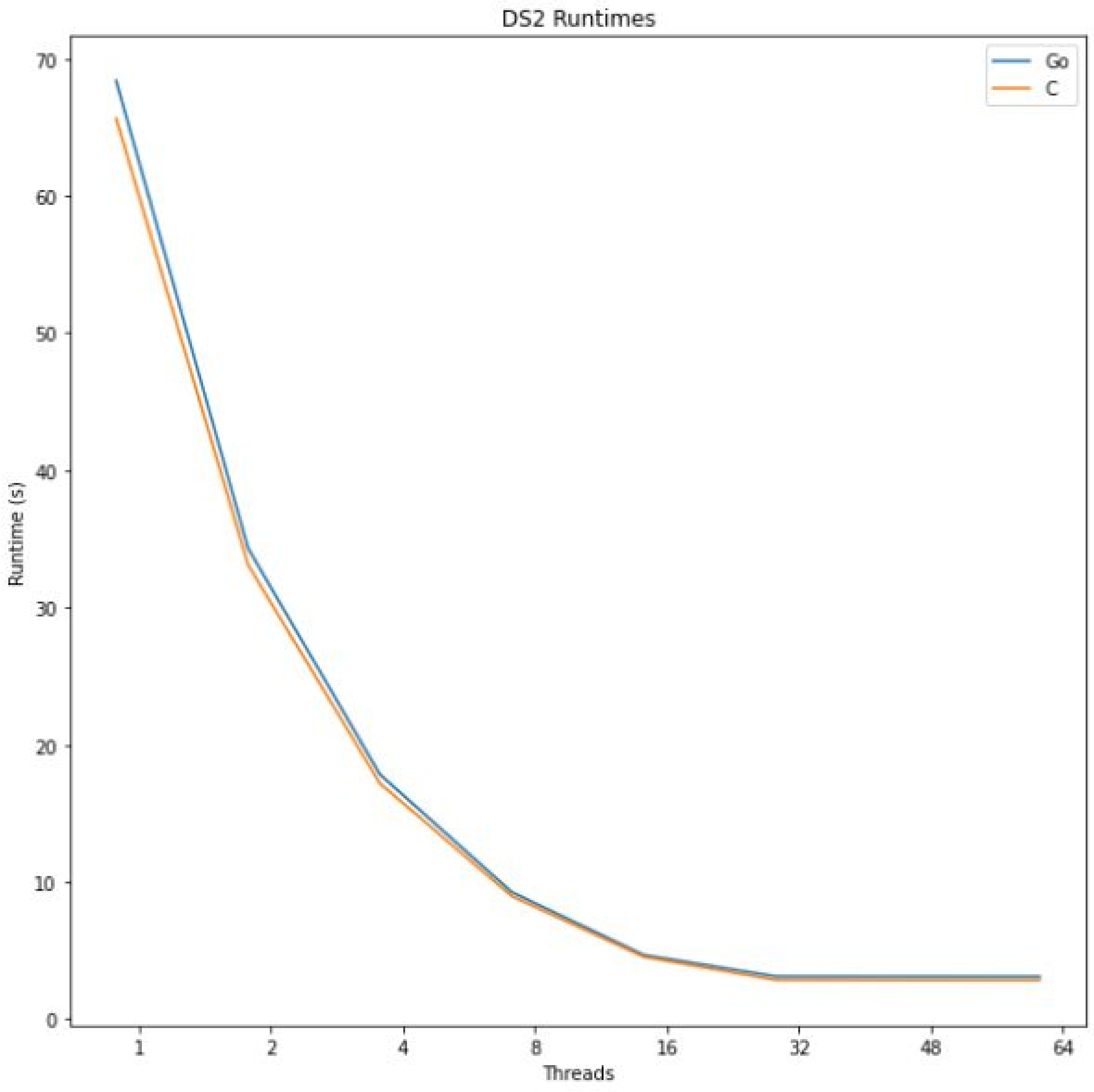
Data Set	Go Runtimes (s)	C Runtimes (s)
DS1	15.963	15.795
DS2	68.538	65.970

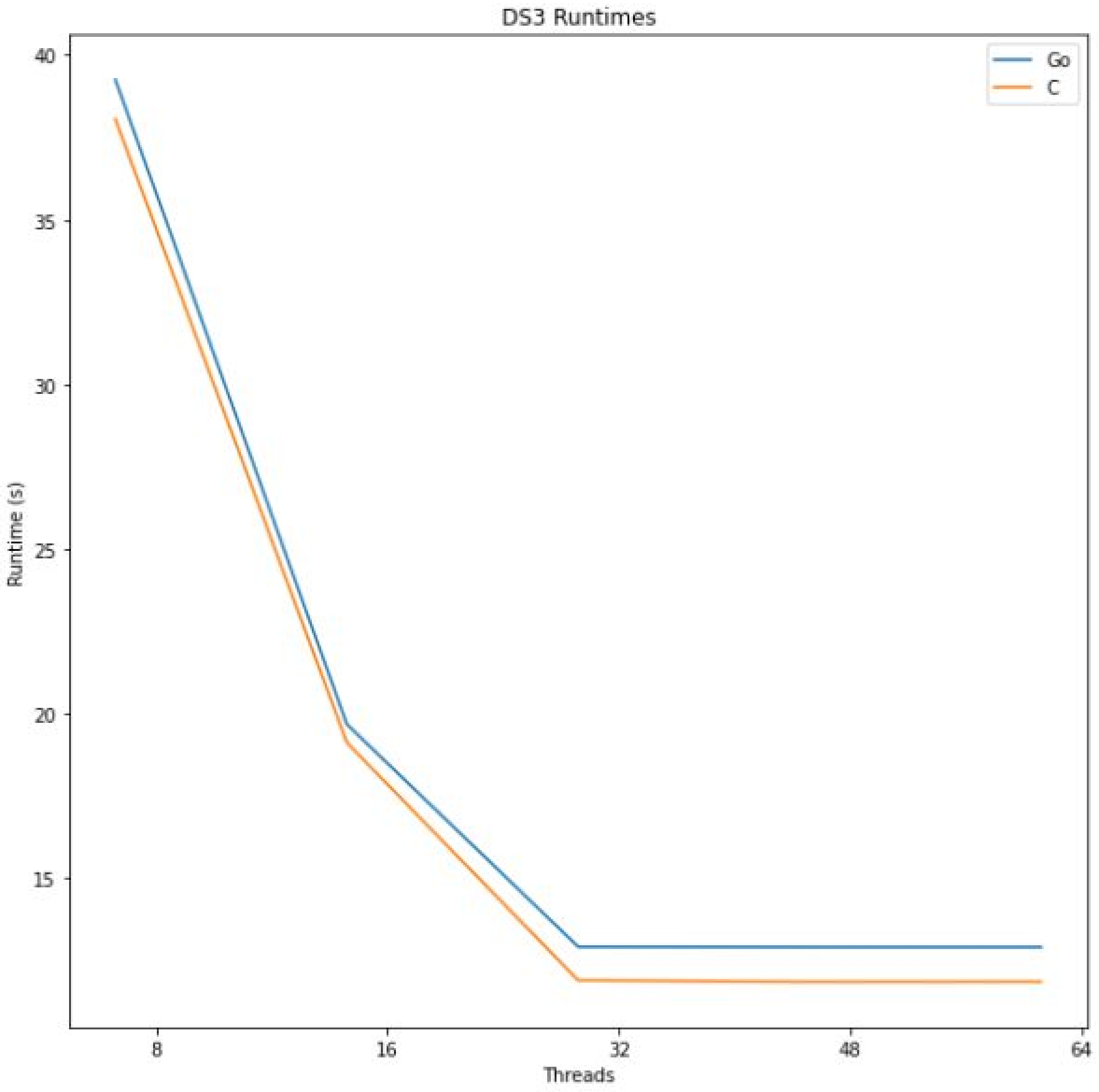
The runtimes for the sequential programs are very similar for the smaller data set (DS1) with C being negligibly faster, however the difference is accentuated using the much larger data set (DS2). There are many reasons why C is faster on average although primarily the main difference between Golang and C is the maturity of the languages. Since C has been in existence since 1972 there has been significantly more incremental optimisations within the language over the years. There is also a significantly larger user base actively using C meaning there is a large audience with a vested interest in improving the language. Yet you can also argue that since Go has been developed by Google that some of the brightest minds currently in the Computer Science field have been actively working on the project, making significant optimisations to it since 2012. Another significant difference between C and Go is garbage collection. C delegates all the garbage collection responsibility to the user which means that there is no dedicated garbage collector process running in parallel with the program. Since one of Go's main goals was simplicity they enforce the use of a garbage collector taking up useful CPU time, however since there has been significant focus on optimising the garbage collection system the runtimes compared with C are not absolutely disproportionate as with maybe Python which has a dynamic type system as well.

2 Comparative Parallel Performance Measurements

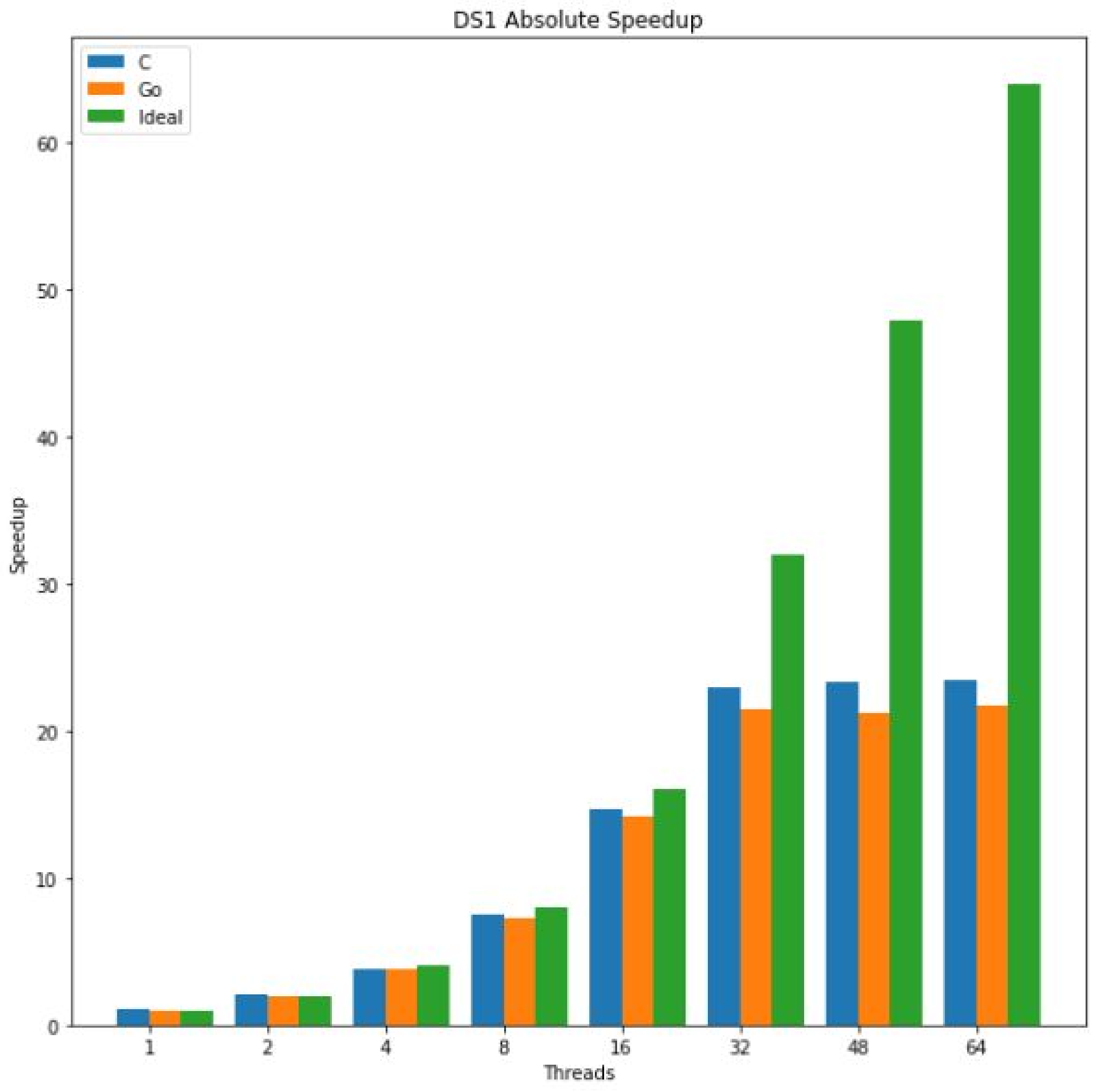
2.1 Runtimes

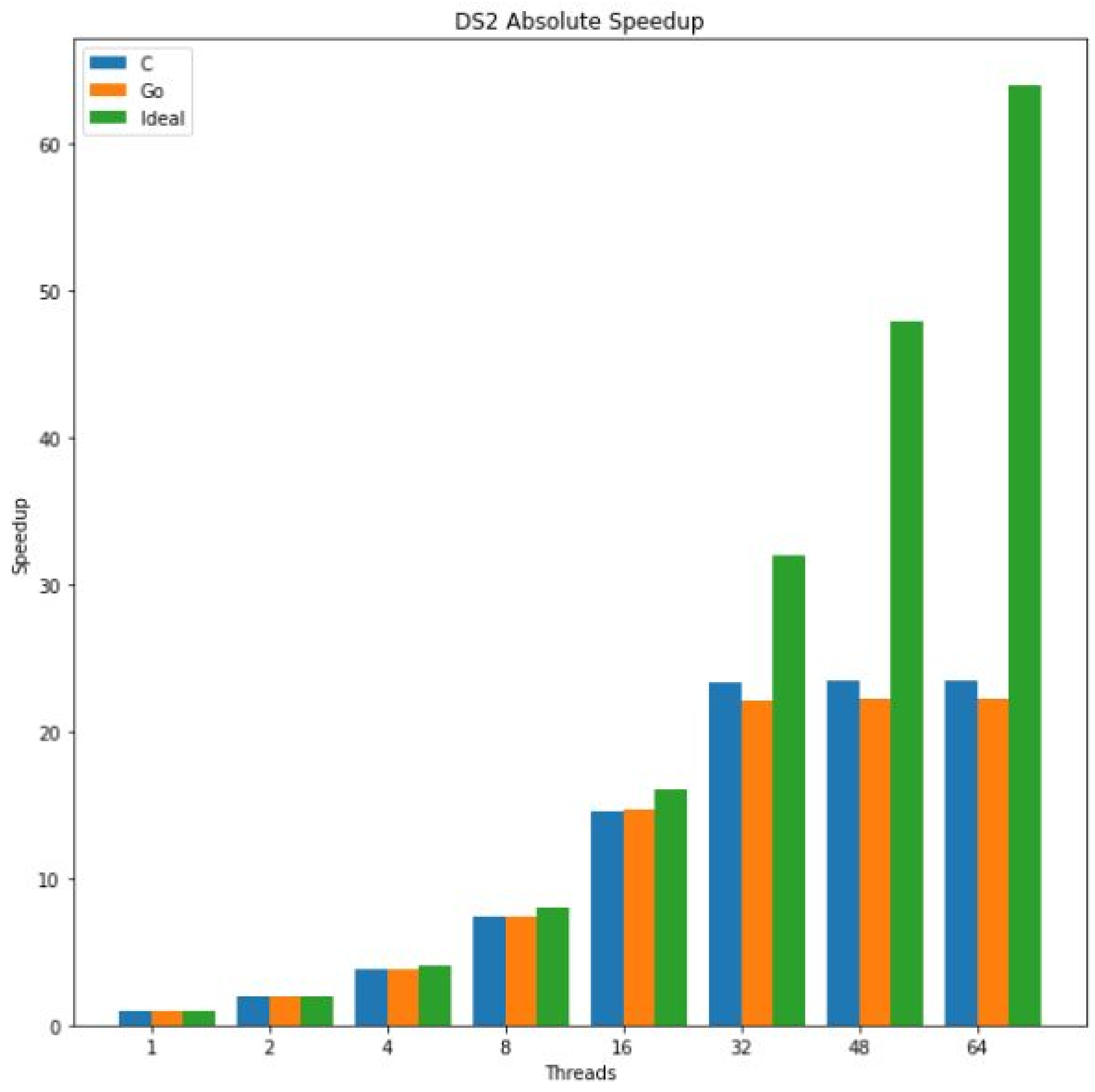


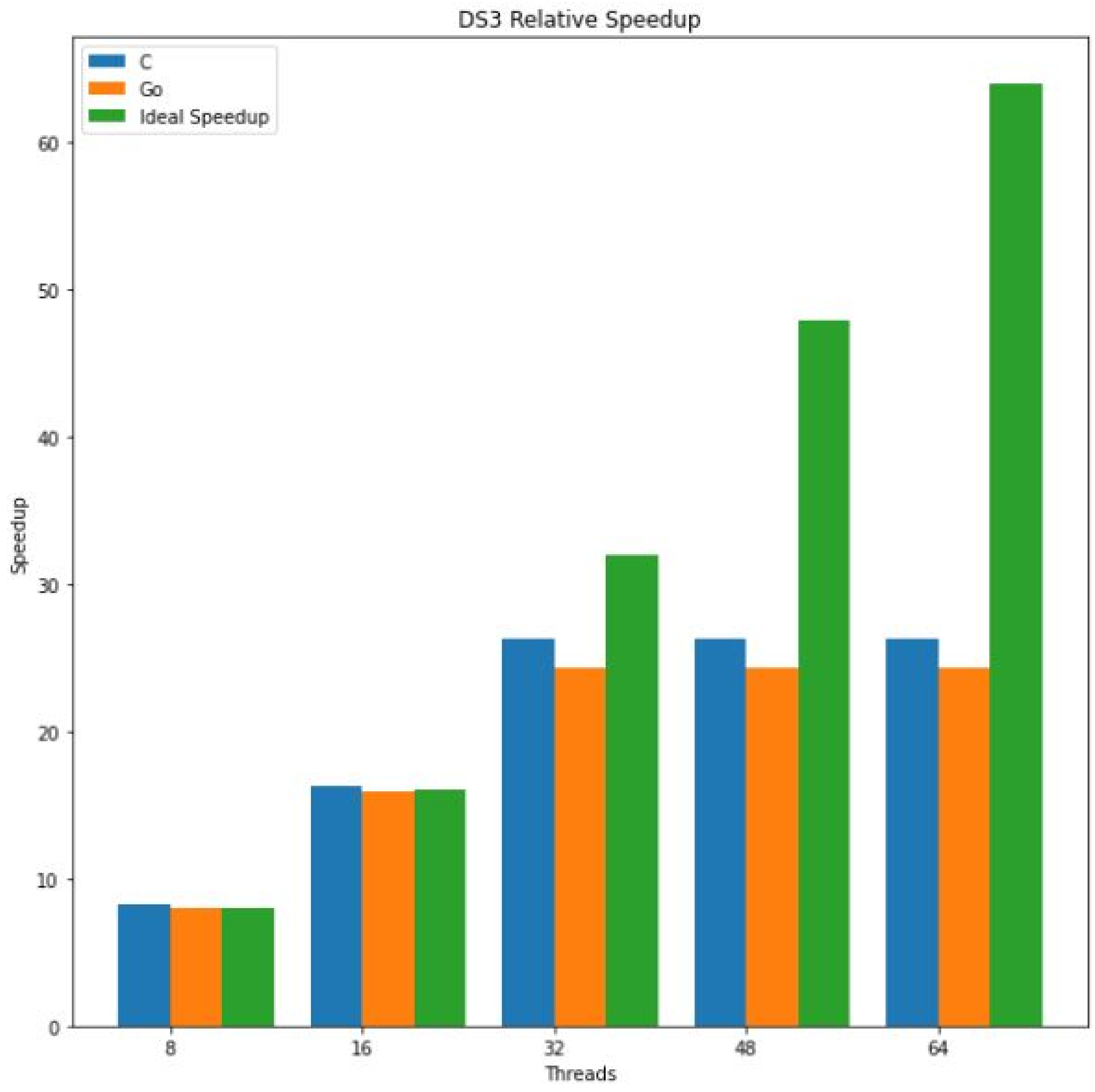

DS2 Runtimes
Runtime (s)
0
10
20
30
40
50
60
70
Threads
1
2
4
8
16
32
48
64
Go
C


DS3 Runtimes
Go
C
Runtime (s)
40
35
30
25
20
15
Threads
8
16
32
48
64

2.2 Speedups


DS1 Absolute Speedup
C
Go
Ideal
Speedup
0
10
20
30
40
50
60
Threads
1
2
4
8
16
32
48
64


DS2 Absolute Speedup
C
Go
Ideal
Speedup
0
10
20
30
40
50
60
Threads
1
2
4
8
16
32
48
64


DS3 Relative Speedup
C
Go
Ideal Speedup
Speedup
Threads
0
10
20
30
40
50
60
8
16
32
48
64

2.3

Language	Sequential Runtime (s)	Best Parallel Runtimes (s)	Best Speedup	No. Threads
DS1				
Go	15.963	0.736	22.026	64
C	15.795	0.674	22.564	64
DS2				
Go	68.538	3.078	22.257	64
C	65.970	2.808	23.391	64

As we can see from the graphs in the runtime section, C+Openmp consistently outperforms the Go implementations and achieves a marginally greater speedup in all cases. It must be noted as well though that multiple runs on the gpg cluster would provide slightly different medians, I'm assuming this was dependent on the load on the server at the time, even though I tried to use them when they were least busy.

For both C+Openmp and Go there is a very noticeable difference in runtimes going from 1 - 32 threads, almost matching the ideal speedups expected. However past 32 the differences are almost unnoticeable with the runtimes improving very marginally. It has to be noted that in every case the best runtimes were always achieved on 64 cores (with the exception of DS3 in Go on 48 threads but the difference was by 0.01), however when comparing the ideal speedups it is evident that the overhead introduced by context switches causes the further introduction of threads to not significantly improving the runtimes.

Section 3 Programming Model Comparison

Both programming models are very intuitive to use when in this domain. Parallelising the C code for example was trivial after a lot of trial and error. Adding in several pragmas throughout the code would produce very inconsistent results and adding extra parameters to the pragmas would produce confusing results. However after familiarising myself more with the solid documentation online it became evident that the solution could be achieved through adding one pragma in the sumTotient function. As was hinted in the example code provided in the slides and the lab, using a reduction on “sum” would be key. After an extensive testing period I had an implementation that matched the expected speedup.

The Go implementation was more difficult than implementing the OpenMP solution however it was very intuitive to introduce channels into the code. I chose to create a simple scheduling algorithm that launched the number of goroutines required and communicated that back through channels. Also since this programming model is based on CSP then my extensive research into Pi-Calculus came into use as the similar nature in which programs are structured allowed me to simply model this program. Since the runtime environment is responsible for scheduling and garbage collection this reduces the amount of work the programmer needs to do in order to make an efficient program, and since the language supports very high level paradigms such as channels and goroutines it massively simplifies the work needed to be done to create code that runs close to the speeds of C code.

It really all comes down to the level of abstraction that suits the developer best. If a start-up team was to develop a very efficient parallelised codebase, with a very quick iteration time frame, then Go is the obvious option. Using goroutines has so many benefits, and Golang in general is a lot more intuitive to debug. But you will be sacrificing a slight bit of efficiency, and there will be a greater load on the hardware. Therefore if for example you are working on a safety critical system, then the obvious choice would be C. C may be significantly harder to write, and not have an easy to use type system, but all of this is under the guise of efficiency. If the hardware use needed to be optimised so that lives are saved then C hands down would be the better programming model to choose.

In conclusion C+Openmp does have the edge in efficiency, but Golang has the edge in simplicity, and shows more promise for the future due to its growing popularity and backing from Google. As the classic phrase goes: “Keep it simple stupid”.

C + OpenMP

```
> times.txt
echo "-----------Starting Execution------------" >> times.txt;

echo "--------------DS1 C+OpenMP---------------" >> times.txt;
for j in 1 2 4 8 16 32 48 64;
   do for i in {0..2};
       do  echo "DS1 with $j threads" >> times.txt;
       { time ./totientrange 1 15000 $j; } 2>> times.txt;
       echo "" >> times.txt;
   done;
done;

echo "--------------DS2 C+OpenMP---------------" >> times.txt;
for j in 1 2 4 8 16 32 48 64;
   do for i in {0..2};
       do echo "DS2 with $j threads";
       { time ./totientrange 1 30000 $j; } 2>> times.txt;
   done;
done;

echo "--------------DS3 C+OpenMP---------------" >> times.txt;
for j in 8 16 32 48 64;
   do for i in {0..2};
       do echo "DS3 with $j threads";
       { time ./totientrange 1 60000 $j; } 2>> times.txt;
   done;
done;
echo "-----------Finished Execution------------" >> times.txt;
```

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

// Greg Michaelson 14/10/2003
// Patrick Maier   29/01/2010 [enforced ANSI C compliance]

// This program calculates the sum of the totients between a lower and an
// upper limit using C longs. It is based on earlier work by:
```

```c
// 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 <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++;
 return length;
}


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

long sumTotient(long lower, long upper)
{
 long sum, i;

 sum = 0;
 int chunk = 32;

 #pragma omp parallel for schedule(dynamic, chunk) reduction(+:sum)
 for (i = lower; i <= upper; i++)
       sum = sum + euler(i);

 return sum;
}


int main(int argc, char ** argv)
{


 long lower, upper;
 int threads;

 if (argc != 4) {
   printf("not 3 arguments\n");
   return 1;
 }
 sscanf(argv[1], "%ld", &lower);
 sscanf(argv[2], "%ld", &upper);
 sscanf(argv[3], "%d", &threads);
 omp_set_num_threads(threads);

 printf("C: Sum of Totients  between [%ld..%ld] is %ld\n",
        lower, upper, sumTotient(lower, upper));
  return 0;
}
```

The parallel paradigm used here is data parallelism, where each core can access every memory address, and all the memory access is hidden in the preprocessor stage, as OpenMP handles all this behind the scenes. This approach is much simpler for the programmer to implement parallelism.

GO

```
> times.txt

echo "-----------Starting Execution------------" >> times.txt;

echo "-----------------DS1 GO------------------" >> times.txt;

for j in 1 2 4 8 16 32 48 64;
   do for i in {0..2};
       do echo "DS1 with $j threads" >> times.txt;
       { time go run TotientRange.go 1 15000 $j; } 2>> times.txt;
       echo "" >> times.txt;
   done;
done;

echo "-----------------DS2 GO------------------" >> times.txt;

for j in 1 2 4 8 16 32 48 64;
   do for i in {0..2};
       do echo "DS2 with $j threads" >> times.txt;
       { time go run TotientRange.go 1 30000 $j; } 2>> times.txt;
       echo "" >> times.txt;
   done;
done;

echo "-----------------DS3 GO------------------" >> times.txt;

for j in 8 16 32 48 64;
   do for i in {0..2};
       do echo "DS3 with $j threads" >> times.txt;
       { time go run TotientRange.go 1 60000 $j; } 2>> times.txt;
       echo "" >> times.txt;
   done;
done;
echo "-----------Finished Execution------------" >> times.txt;
```

```
// totientRange.go - Sequential Euler Totient Function (Go Version)
// compile: go build
// run:     totientRange lower_num upper_num
```

```go
// Phil Trinder    30/8/2018

// This program calculates the sum of the totients between a lower and an
// upper limit
//
// 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"
   "runtime"
   "strconv"
   "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
}

// 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(n int64) int64 {
   var length, i int64

   length = 0
   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".
//
// sumTotient lower upper = sum (map euler [lower, lower+1 .. upper])

func sumTotient(lower, upper int64, c chan int64) {
   var sum, i int64

   sum = 0
   for i = lower; i <= upper; i++ {
       sum = sum + euler(i)
   }
   c <- sum
}

func scheduling(lower, upper int64) int64 {

   var chunkSize, step, sum, i int64
   chunkSize = 32
   step = lower
   sum = 0

   c := make(chan int64)

   for step <= upper {
       if (step + chunkSize) > upper {
           go sumTotient(step, upper, c)
           i++
           break
       }
       go sumTotient(step, step+chunkSize, c)
       step = step + chunkSize + 1
       i++
   }

   for i != 0 {
       sum += <-c
       i--
   }

   return sum

}

func main() {
   var lower, upper int64
   var threads int
   var err error
   // Read and validate lower and upper arguments
   if len(os.Args) < 4 {
       panic(fmt.Sprintf("Usage: must provide lower and upper range limits
and number of threads 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 threads, err = strconv.Atoi(os.Args[3]); err != nil {
        panic(fmt.Sprintf("Can't parse third argument"))
    }

    runtime.GOMAXPROCS(threads)

    // Record start time
    start := time.Now()
    // Compute and output sum of totients
    fmt.Println("Sum of Totients between", lower, "and", upper, "is",
scheduling(lower, upper))
    // Record the elapsed time
    t := time.Now()
    elapsed := t.Sub(start)
    fmt.Println("Elapsed time", elapsed)
}
```

The parallel paradigm used here is message passing, with each thread running in isolation working particularly on its designated load, and returning results using with a simple message that does not overshare unnecessary data, protecting the memory being used to calculate its specific chunk.