Section 1 Comparative Sequential Performance (2 marks)

Data Set	Go runtimes(s)	C runtimes (s)
DS1	16.51070541	19.509976
DS2	70.57904187	84.544474

Table 1: Comparing C and Go Sequential Runtimes

Section 2 Comparative Parallel Performance Measurements (12 marks)

I use the second as the unit. I only keep the first 3 numbers after point in C code.

Section 2.1 Runtimes.

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

Threads /goroutines	go runtime(s)	C runtime(s)
sequential	16.51070541	19.509
1	16.56481488	19.723
2	12.80199701	15.062
4	7.784804143	9.038
8	4.193619101	5.013
12	2.901805619	3.461
16	2.28248696	2.644
24	1.546794145	1.819
32	1.298523879	1.407
48	1.037974181	1.049
64	0.926042525	1.1084

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

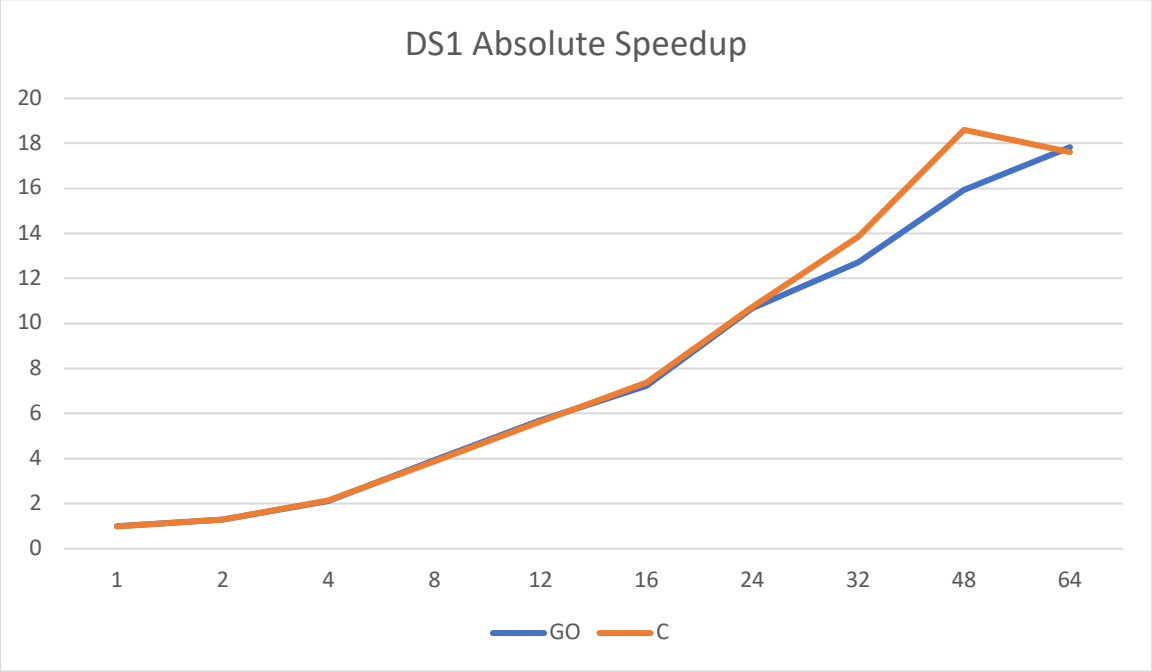
Threads /goroutines	go runtime(s)	C runtime(s)
sequential	70.57904187	84.544
1	70.5348553	84.285
2	54.01022328	63.725
4	32.24430842	38.512
8	17.88690512	21.33
12	12.36136477	14.762
16	9.522891973	11.2
24	6.633120542	7.901
32	5.341379413	6.015
48	4.02412511	4.509
64	3.820356588	4.245

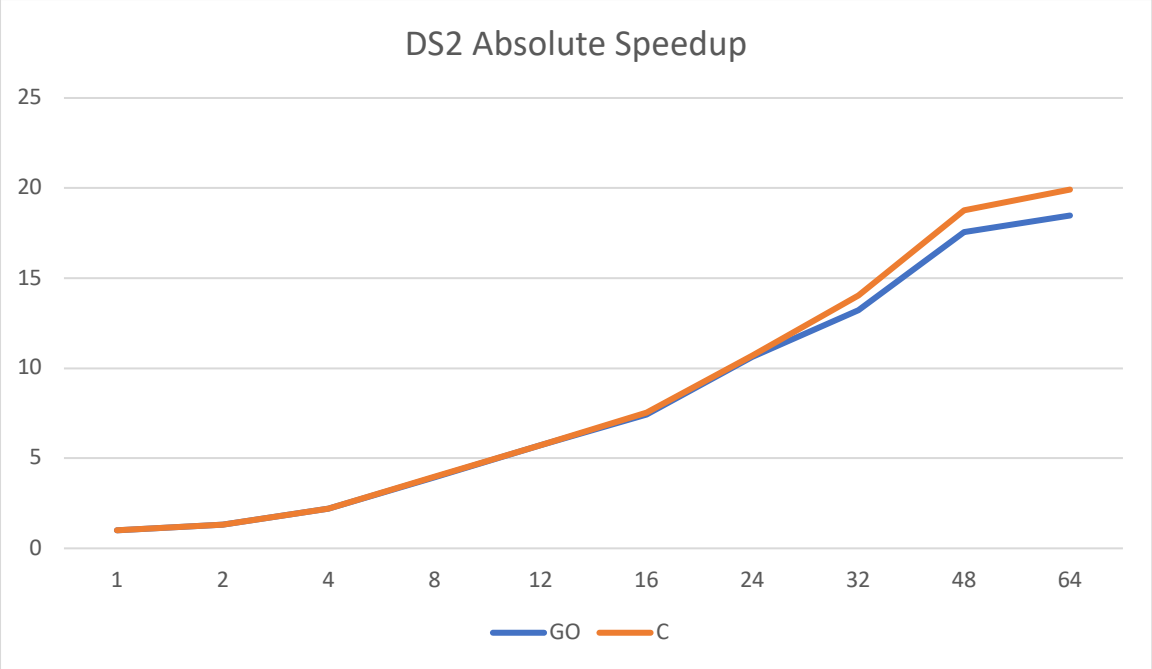
DS3: execution times for the parallel C+OpenMP and Go programs on 8, 16, 32, 64, threads/ goroutines on a GPG Cluster node

Threads /goroutines	go runtime(s)	C runtime(s)
8	77.073047662	90.206
16	43.11157582	47.782
32	22.97898839	26.073
64	16.69879699	17.615

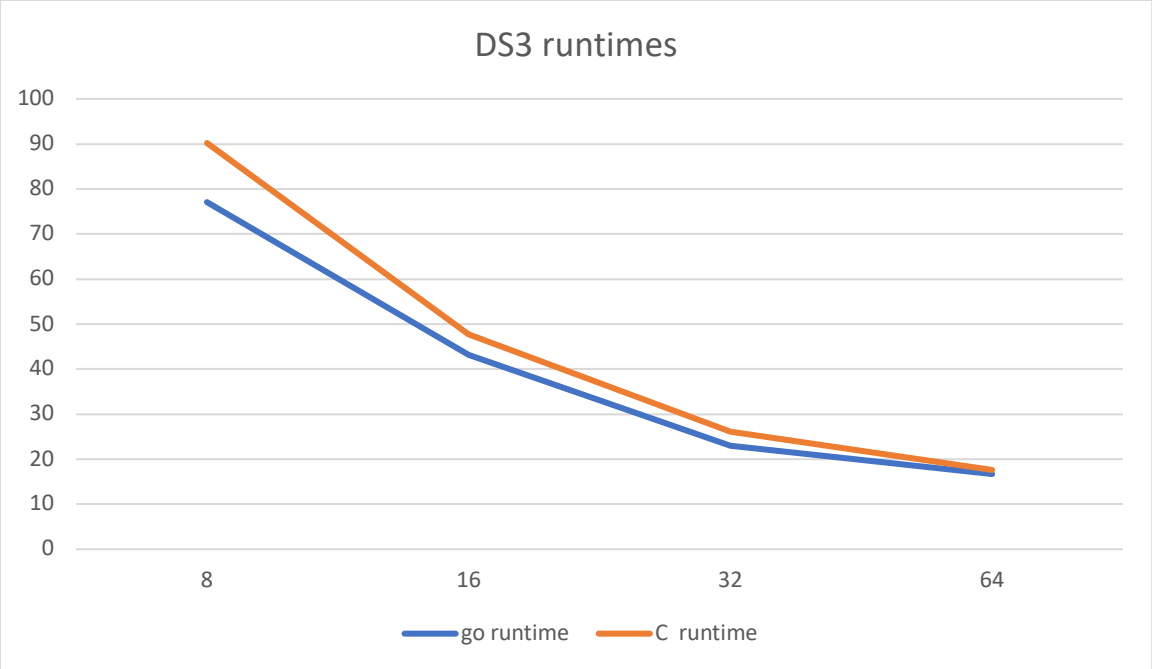
Section 2.2 Speedups.

X axis is threads/goroutines number, Y axis is absolute speedup





I did not test sequential programs in DS3, I think it is too slow to wait. I plot the vary runtime in different threads/goroutines of DS3



Section 2.3

Language	Sequential runtimes(s)	Best parallel runtimes(s)	Best Speedup	No.cores
go	16.510	0.926	17.829	64
C+OpenMP	19.509	1.049	18.597	48

Table 2: Comparing C+OpenMP and Go Parallel Runtimes and Speedups

Section 2.4
In computing 0-15000 totients sum, Go sequential program runtimes is shorter than C program. Go Parallel program runtimes is a bit less than C+OpenMP program in the same quantity of cores.

When adding more threads/goroutines, there will be a significant reducing time in computing the sum. However, when the threads increase, like after 32, the vary of runtime is not significant.

Section 3 Programming Model Comparison (6 marks).

When I programming the parallel program, I should assign specific tasks for Go program. For example, I should assign 1-1000, 1001-2001….to goroutines. But for C+OpenMP, it will assign the tasks automatically. But Go perform better in computing sum of totient compared C+OpenMP in the same settings. In comparing DS3, go performs much better than C+OpenMP in 8 and 16 cores. However, when the number of cores increase to 48 and 64, there is even no difference in go and openMP.

Section 4: Reflection on Programming Models MSc Students Only (5 marks)

Go performs better both in sequential program and parallel program than C+OpenMP. Maybe the advantage of go is that it performs well and you can manipulate threads easier. For openMP I do think you could do less things about threads. The disadvantage of go is that it is more difficult to learn but for c programmer you can master openMP in a very short time. OpenMP is more widely used in the world nowadays, but I think go language will be more and more popular in the future. I prefer to use Go to do parallel program if I have future parallel tasks. The go performance is better even though it is a bit harder to code for me.

Appendix A C+OpenMP TotientRange Program (5 marks)

I have some notation in my own language and that is easier for me to understand.

```c
#include <stdio.h>
#include <sys/time.h>
#include <omp.h>  /* Here's the header file for OpenMP. */

// 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;
    for (i = lower; i <= upper; i++)
        sum = sum + euler(i);
    return sum;
}


int main()
{
    unsigned long msec;
    double msperprime;
    struct timeval start, stop;
```

```
    long lower = 1;
    long upper = 30000;
    int number_t = 12;


    long sum = 0;
    long i = 0;

    gettimeofday(&start, NULL);       /* note start time */
//excute parallel
#pragma omp parallel for reduction(+:sum) num_threads(number_t)
    for (i = lower; i <= upper; i++)
        sum = sum + euler(i);

    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("current number of threads is %d \n",number_t);
    printf("Sum of totient is  ----- %ld  \n",sum);
    printf("running time is ----- %lu  ms\n",msec);
    return 0
}
```

Appendix B Go TotientRange Program (5 marks)

```
package main

import (
   "fmt"
   "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 hcf2(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 relprime2(x, y int64) bool {
   return hcf2(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 euler2(n int64) int64 {
   var length, i int64

   length = 0
   for i = 1; i < n; i++ {
      if relprime2(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 sumTotient2(lower, upper int64, c chan int64) {
   var sum, i int64

   sum = 0
   for i = lower; i <= upper; i++ {
      sum = sum + euler2(i)

   }
   c <- sum
   fmt.Println("sumof", lower, "and", upper)

}

func main() {
   var lower, upper, totalNum,subTotal int64
   var dones int64 = 0
   c := make(chan int64)


   var numberOfThread int64 = 4
   upper = 15000
   lower = 1

   intervalNum := upper / numberOfThread
   // Record start time
   start := time.Now()
   //
   var i int64
   for i = 0; i < numberOfThread; i++ {
      base := int64(i) * intervalNum + 1
      if (i == numberOfThread - 1) {
         go sumTotient2(base, upper, c)
         break
      }
      go sumTotient2(base, base + intervalNum-1, c)

   }
   //取数
   for dones < numberOfThread {

      subTotal  = <- c

      fmt.Println(subTotal) //当打印出一个数的时候 dones 不变，然后再循环，必须等取到 10 个 0
时候结束，这个时候也就是表示 10 个线程执行完毕
      dones++
      totalNum = subTotal + totalNum
```

```
    }
    fmt.Println("Sum of Totients between", lower, "and", upper, "is",totalNum )
    // Record the elapsed time
    t := time.Now()
    elapsed := t.Sub(start)
    fmt.Println("Number of go routines is ", numberOfThread,"----Elapsed time is ",
elapsed)
}
```