Austin Brown

CPE 434-01

1/28/2021

Lab 6

Theory:

A process is a program that is being executed. A thread is just a segment of a process, or a subprocess. They are much more lightweight than processes are and can be created quicker that processes can. While processes have their own memory space, threads share memory. Although they share a space, they do have their own stack for local storage.

The rectangular method is a way of approximating a definite integral. It involves breaking the area under the curve into rectangles and summing them up. The more rectangles that are used, the more accurate the result will be.

When you are using a mutex, only one thread can access the memory space at a time. They are either locked or unlocked. A semaphore has a wait and signal operation. Instead of locking a memory region, they tell threads to wait or signal them.

Observations:

Compilation

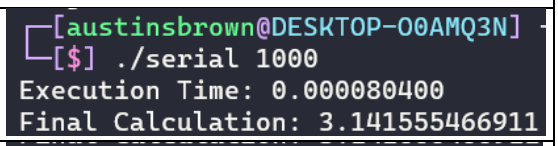
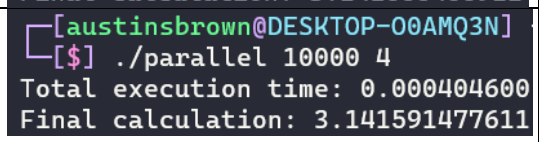
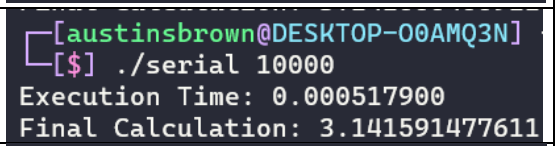
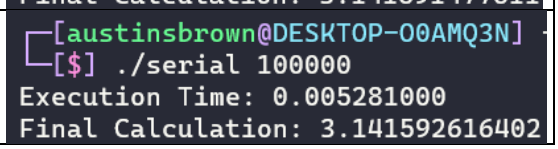
```
┌─[austinsbrown@DESKTOP-O0AMQ3N] - [/mnt/c/User
└─[$] make
gcc parallel.c -o parallel -lm -lpthread
gcc serial.c -o serial -lm
┌─[austinsbrown@DESKTOP-O0AMQ3N] - [/mnt/c/User
└─[$]
```

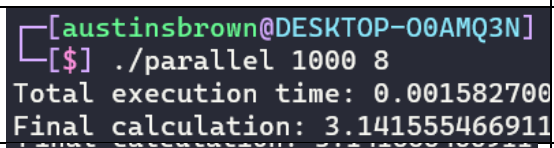
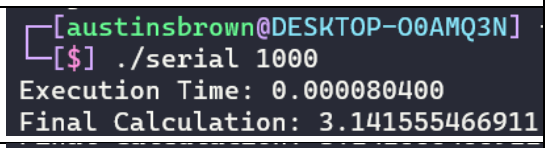
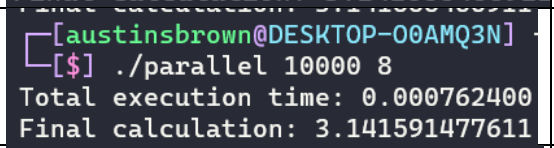
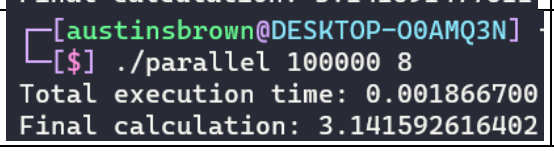
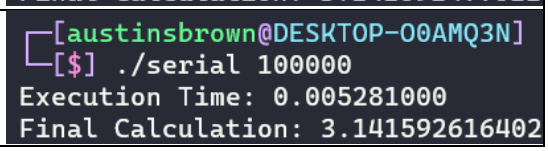
Intervals	Parallel (2 Threads)	Serial

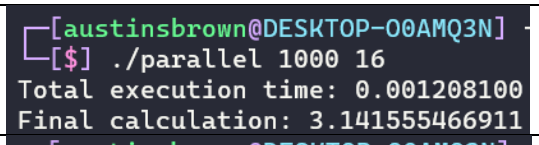
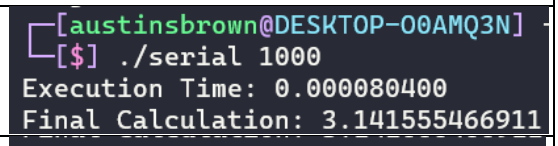
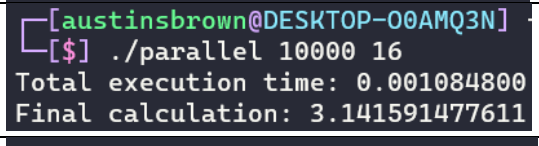
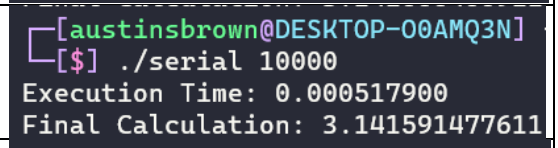
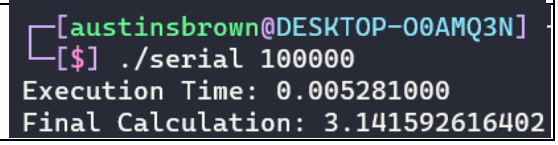
1000	`[austinsbrown@DESKTOP-O0AMQ3N]`
	`[$] ./parallel 1000 2`
	`Total execution time: 0.000233500`
	`Final calculation: 3.141555466911`
		`[austinsbrown@DESKTOP-O0AMQ3N]`
		`[$] ./serial 1000`
		`Execution Time: 0.000080400`
		`Final Calculation: 3.141555466911`

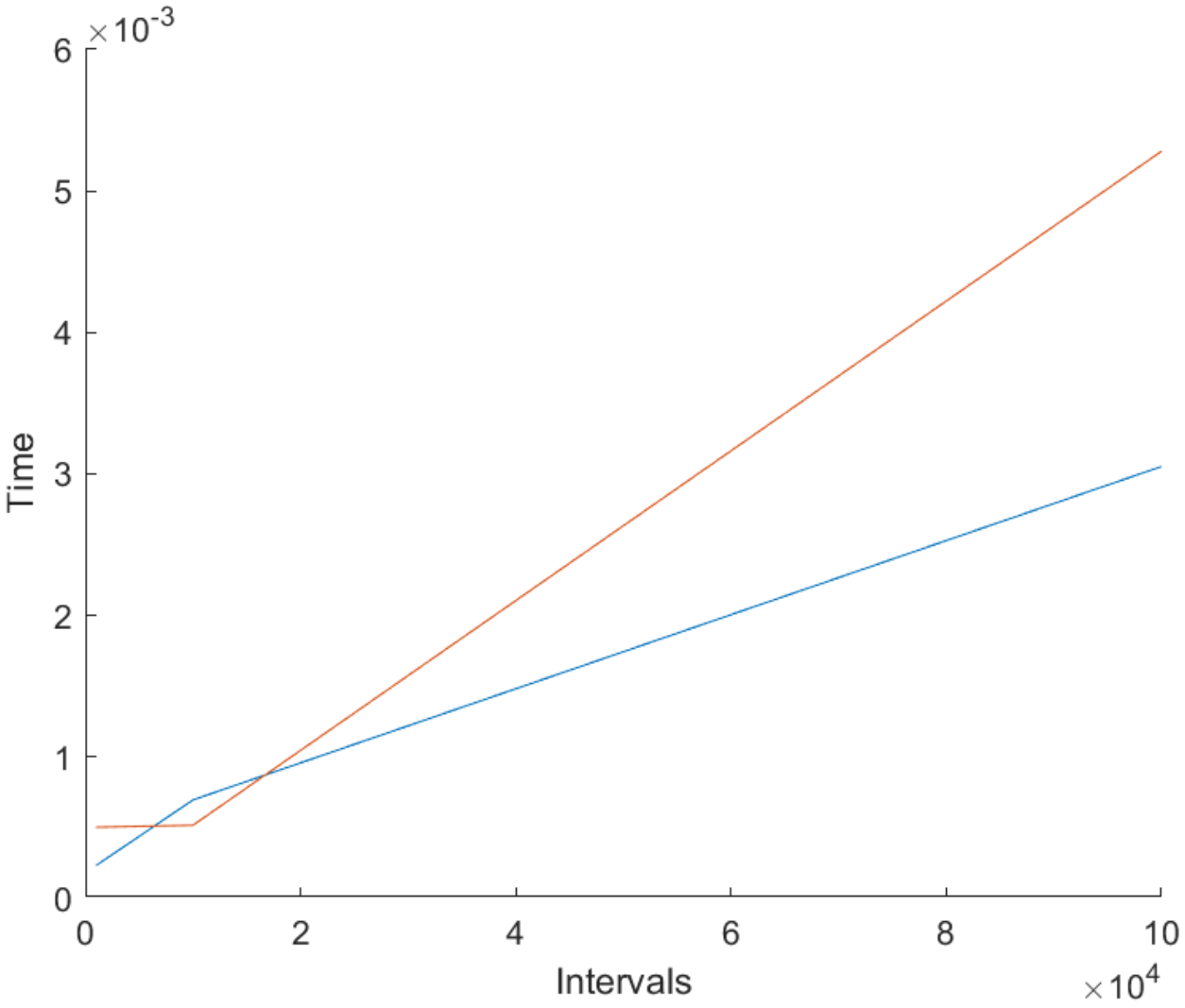
10000	`[austinsbrown@DESKTOP-O0AMQ3N]`
	`[$] ./parallel 10000 2`
	`Total execution time: 0.000697000`
	`Final calculation: 3.141591477611`
		`[austinsbrown@DESKTOP-O0AMQ3N]`
		`[$] ./serial 10000`
		`Execution Time: 0.000517900`
		`Final Calculation: 3.141591477611`

100000	`[austinsbrown@DESKTOP-O0AMQ3N]`
	`[$] ./parallel 100000 2`
	`Total execution time: 0.003053200`
	`Final calculation: 3.1415926416402`
		`[austinsbrown@DESKTOP-O0AMQ3N]`
		`[$] ./serial 100000`
		`Execution Time: 0.005281000`
		`Final Calculation: 3.1415926416402`

Intervals	Parallel (4 Threads)	Serial
1000		
10000		
100000		

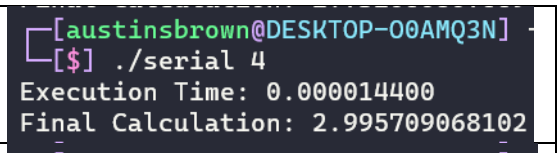
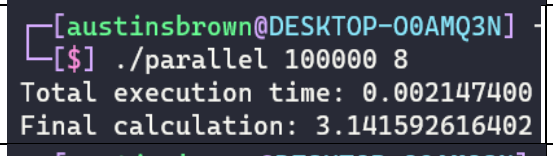
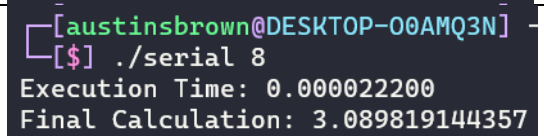
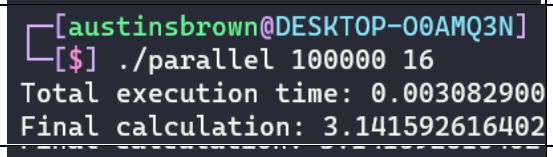
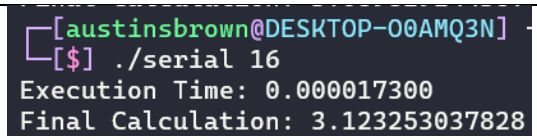
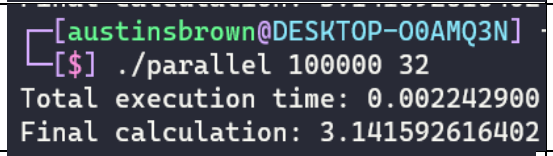
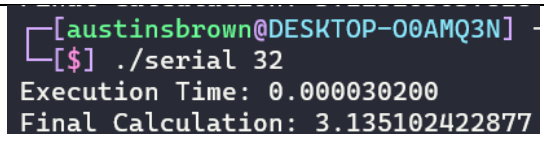
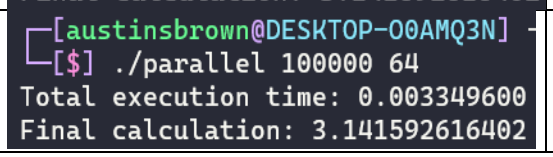
Intervals	Parallel (4 Threads)	Serial
1000		
10000		
100000		

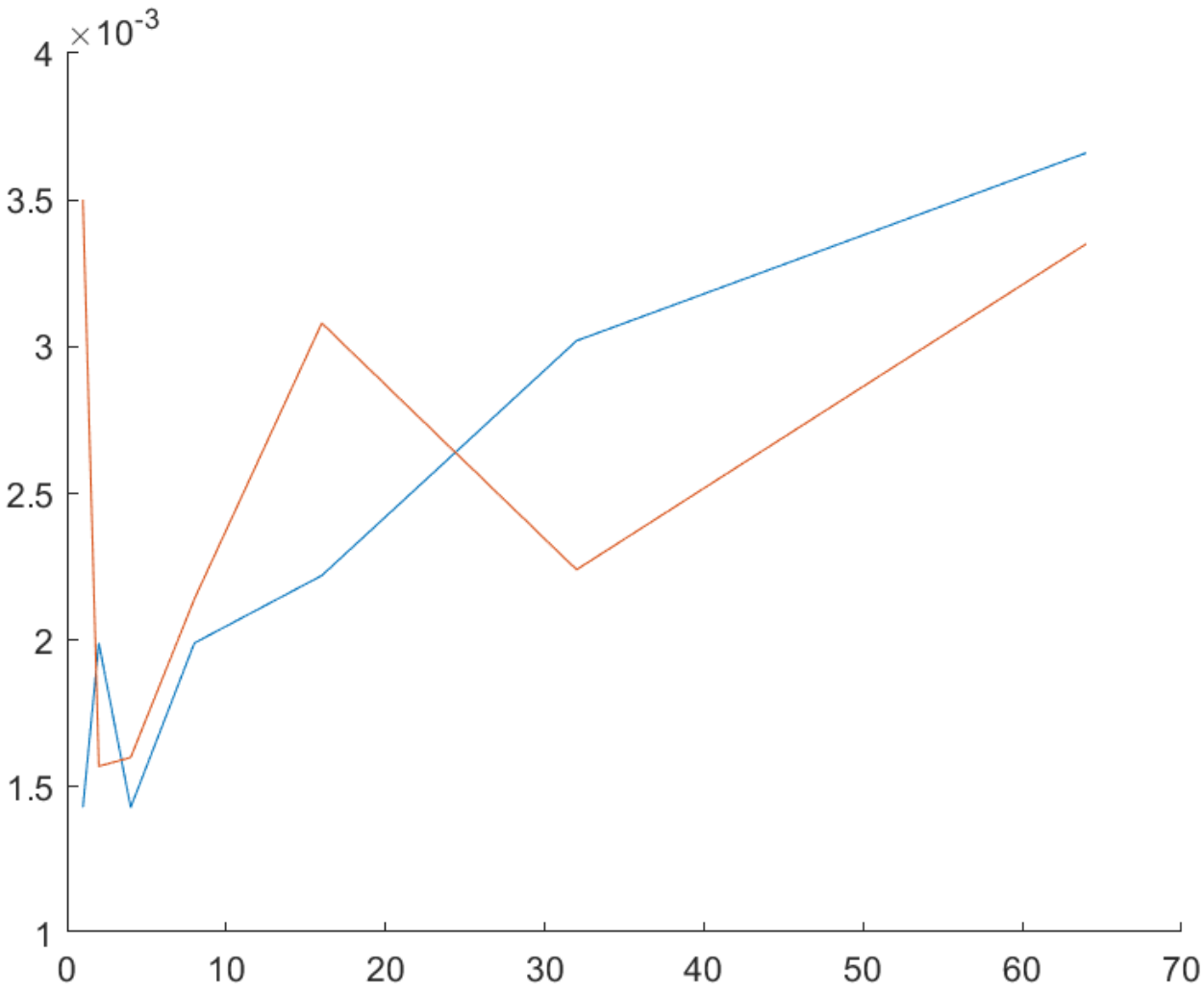
Intervals	Parallel (16 Threads)	Serial
1000		
10000		
100000		



100000 Intervals

Number of Threads	Parallel	Serial
1	[austinsbrown@DESKTOP-O0AMQ3N] - [$] ./parallel 100000 1 Total execution time: 0.003586700 Final calculation: 3.141592616402	[austinsbrown@DESKTOP-O0AMQ3N] - [$] ./serial 1 Execution Time: 0.000014300 Final Calculation: 2.000000000000
2	[austinsbrown@DESKTOP-O0AMQ3N] - [$] ./parallel 100000 2 Total execution time: 0.001575400 Final calculation: 3.141592616402	[austinsbrown@DESKTOP-O0AMQ3N] - [$] ./serial 2 Execution Time: 0.000019900 Final Calculation: 2.732050807569

4	 [austinsbrown@DESKTOP-O0AMQ3N] [$] ./parallel 100000 4 Total execution time: 0.001605000 Final calculation: 3.141592616402	 [austinsbrown@DESKTOP-O0AMQ3N] [$] ./serial 4 Execution Time: 0.000014400 Final Calculation: 2.995709068102
8	 [austinsbrown@DESKTOP-O0AMQ3N] [$] ./parallel 100000 8 Total execution time: 0.002147400 Final calculation: 3.141592616402	 [austinsbrown@DESKTOP-O0AMQ3N] [$] ./serial 8 Execution Time: 0.000022200 Final Calculation: 3.089819144357
16	 [austinsbrown@DESKTOP-O0AMQ3N] [$] ./parallel 100000 16 Total execution time: 0.003082900 Final calculation: 3.141592616402	 [austinsbrown@DESKTOP-O0AMQ3N] [$] ./serial 16 Execution Time: 0.000017300 Final Calculation: 3.123253037828
32	 [austinsbrown@DESKTOP-O0AMQ3N] [$] ./parallel 100000 32 Total execution time: 0.002242900 Final calculation: 3.141592616402	 [austinsbrown@DESKTOP-O0AMQ3N] [$] ./serial 32 Execution Time: 0.000030200 Final Calculation: 3.135102422877
64	 [austinsbrown@DESKTOP-O0AMQ3N] [$] ./parallel 100000 64 Total execution time: 0.003349600 Final calculation: 3.141592616402	 [austinsbrown@DESKTOP-O0AMQ3N] [$] ./serial 64 Execution Time: 0.000036600 Final Calculation: 3.139296912780



Conclusion:

Threads can offer better performance for certain programs. There are limitations, however. There are only so many threads that you can create without slowing the program down. This is partially due to overhead, but you can also generate race conditions is synchronization is not implemented properly. You can use mutexes or semaphores to help this.

Appendix:

serial.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

int main(int argc, char **argv)
{
    if(argc != 2)
    {
        printf("Usage: ./b.out intervals\n");
        exit(-1);
    }

    struct timespec start, stop;
    clock_gettime(CLOCK_REALTIME, &start); // start timer

    int intervals = atoi(argv[1]);
    if(intervals <= 0)
    {
        printf("Interval needs to be greater than 0.\n");
        exit(-1);
    }

    double width = (double)1/intervals;
    double sum = 0;
    double temp1, temp2;
    int i = 0;

    temp1 = sqrt(1 - pow(i * width, 2));
    for (i = 0; i < intervals; i++)
    {
        temp2 = sqrt(1 - pow((i+1) * width, 2));
        sum += ((temp1 + temp2) / 2) * width;
```

```
        temp1 = temp2;
    }


    sum *= 4;

    clock_gettime(CLOCK_REALTIME, &stop);
    unsigned long long totalSeconds = (long long)(stop.tv_sec - start
.tv_sec);
    unsigned long totalNanoseconds = stop.tv_nsec - start.tv_nsec;
    printf("Execution Time: %llu.%.9lu\n", totalSeconds, totalNanosec
onds);
    printf("Final Calculation: %.12f\n", sum);

    return 0;
}
```

integralArgs.h

```
typedef struct integralArgs
{
    int threadID;
    int threadCount; // total number of threads
    int intervalCount; // total number of intervals
    int intervalStart; // what interval each thread starts at
    int intervalsPerThread;
    double width;
}integralArgs;
```

parallel.c

```
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>
```

```c
#include "integralArgs.h"

double globalSum; // sum for every thread
pthread_mutex_t mutex; // used to lock access to globalSum

void *decomp(void *); // calculate the integral

int main(int argc, char **argv)
{
    if (argc != 3)
    {
        printf("Usage: ./parallel intervals threads\n");
        exit(-1);
    }

    struct timespec start, stop;
    clock_gettime(CLOCK_REALTIME, &start); // track the execution tim
e

    integralArgs *args;
    int intervals = atoi(argv[1]);
    int threads = atoi(argv[2]);

    if (intervals <= 0) // perform error checking
    {
        printf("Number of intervals must be greater than 0.\n");
        exit(-1);
    }
    if (threads <= 0)
    {
        printf("Number of threads must be greater than 0.\n");
        exit(-1);
    }
    if (threads > intervals)
    {
        printf("You cannot have more intervals than threads.\n");
        exit(-1);
    }
```

```
    double width = (double)1/intervals;
    int remainder = intervals%threads;  // certain threads will get e
xtra intervals
    int status; // error checking for pthread_create
    int currentInterval = 0;
    globalSum = 0;

    pthread_t threadArray[threads];

    for (int i=0; i<threads; i++)
    {
        args = malloc(sizeof(integralArgs));
        args->threadID = i+1;
        args->intervalCount = intervals;
        args->intervalStart = currentInterval;
        args-
>intervalsPerThread = intervals/threads;   // intervals per thread
        currentInterval += args->intervalsPerThread;

        if (remainder > 0) // handle the remainder
        {
            remainder--; // use one less interval to calculate
            args-
>intervalsPerThread++; // threads will calculate an extra interval
            currentInterval++;
        }

        args->threadCount = threads;
        args->width = width;

        status = pthread_create(&threadArray[i], NULL, decomp, (void*
)args); // create pthread
        if (status)
        {
            printf("Error in creating the threads: %d\n", i+1);
            return -1;
        }
    }
```

```
    for (int i = 0; i < args->threadCount; i++)
    {
        pthread_join(threadArray[i], NULL);
    }

    double finalCalc = 4*globalSum; // get final answer

    clock_gettime(CLOCK_REALTIME, &stop);
    unsigned long long totalSeconds = (long long)(stop.tv_sec - start
.tv_sec);
    unsigned long totalNanoseconds = stop.tv_nsec - start.tv_nsec;

    printf("Total execution time: %llu.%.9lu\n", totalSeconds, totalN
anoseconds);
    printf("Final calculation: %.12f\n", finalCalc);

    free(args);
    return 0;
}


void *decomp(void *argument)
{
    integralArgs *threadArgs = (integralArgs*)argument;
    double sum = 0;
    int myInterval = threadArgs-
>intervalStart; // keep up with what interval you are on

    double temp1, temp2;
    temp1 = sqrt(1-pow(myInterval*threadArgs-
>width, 2)); // use trapezoid theorem to solve the integral
    for (int i=0; i < threadArgs->intervalsPerThread; i++)
    {
        temp2 = sqrt(1-pow((myInterval+1)*threadArgs->width, 2));
        sum += ((temp1+temp2)/2)*threadArgs->width;
        temp1 = temp2;
        myInterval++;
    }
```

```
    pthread_mutex_lock(&mutex);      // get access to globalSum
    globalSum += sum;
    pthread_mutex_unlock(&mutex);    // release exclusive access to gl
obalSum
}
```