

Operating Systems Lab

CPE 435-01

Lab 06: Serial and Parallel Performance

By: David Thornton

Lab Date: 15 February 2021

Lab Due: 22 February 2021

Demonstration Due: 22 February 2021

Introduction

The purpose of this lab is to give students an introduction to message queues in Linux.

Theory

Topic 1: Difference between Threads and Processes

- A process is the parent of a thread. Every thread is attached to one process.

Topic 2: Rectangular Decomposition Method

- To approximate a definite integral by using rectangles, first divide the definite function's interval $[a, b]$ into n pieces of equal length dx . Then, find the function value at the left corner, right corner, or middle of the rectangle. These are called LRAM, RRAM, and MRAM respectively. There also exists a trapezoidal rule, but that was implemented. In this lab, we will be using the LRAM method. Once you have dx and $f(x)$, you find the area of one rectangle, and move to the next one. Once you have found all of the areas, sum them for the integral approximation.

Topic 3: Thread Local Storage

- Thread local storage is a method by which variables are allocated such that there is one instance of the variable per thread.

Topic 4: Mutex and Semaphore

- A mutex locks a thread so that the producer or consumer can not write to the buffer at the same time. Only one thread can work with the entire buffer at a time. A semaphore is a generalized mutex, allowing for multiple buffers.

Lab Assignment

Write a program in a general pthreads manner, so that they can utilize an arbitrary number of threads, so the computation is divided among these computational entities as evenly as possible. Design program so that the number of intervals and the number of operating threads can be entered as a run time parameter (i.e. command line argument) by the user under the constraint that it must be greater than or equal to the number of threads that are used. Plot the execution time taken by 100,000 intervals with 1, 2, 4, 8 and 16 threads. You are expected to do the following tasks:

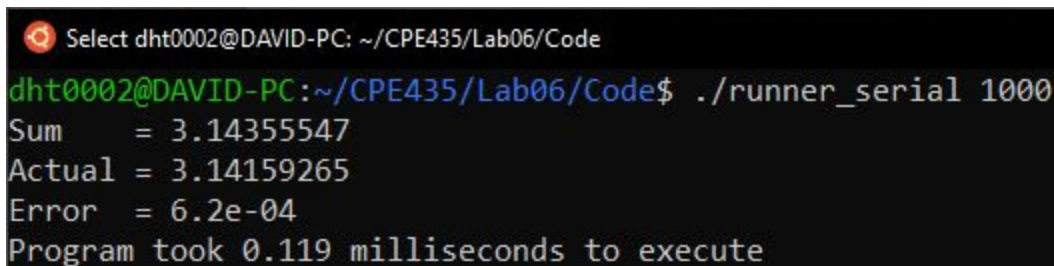
- Write a serial code version to compute integral, using a timing function to evaluate its execution time.
- Write a parallel version of code using pthreads model, compare the result and execution time.
- Compare the performance of two models (serial and pthreads). Try different numbers of intervals by increasing the number. Show graph to summarize the execution time of the two models versus different number of intervals (i.e. 1,000, 10,000 and 100,000 intervals with serial vs parallel with 2 threads only.)
- Compare the performance of the serial model and parallel based on the execution time. (i.e. 100,000 intervals with serial vs parallel with 1, 2, 4, 8 and 16 threads)

Observations

All code performs as expected.

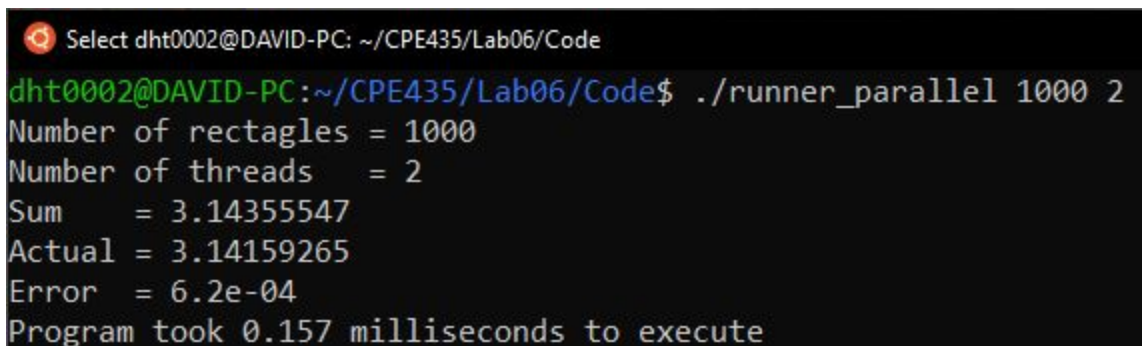
Test A: Comparison of performance between the two models

Example 1: Serial vs 2 pthreads for 1,000 rectangles



```
Select dht0002@DAVID-PC: ~/CPE435/Lab06/Code
dht0002@DAVID-PC:~/CPE435/Lab06/Code$ ./runner_serial 1000
Sum      = 3.14355547
Actual   = 3.14159265
Error    = 6.2e-04
Program took 0.119 milliseconds to execute
```

Figure 1: Serial model for 1,000 rectangles



```
Select dht0002@DAVID-PC: ~/CPE435/Lab06/Code
dht0002@DAVID-PC:~/CPE435/Lab06/Code$ ./runner_parallel 1000 2
Number of rectangles = 1000
Number of threads    = 2
Sum      = 3.14355547
Actual   = 3.14159265
Error    = 6.2e-04
Program took 0.157 milliseconds to execute
```

Figure 2: Pthread model with 2 threads for 1,000 rectangles

Example 2: Serial vs 2 pthreads for 10,000 rectangles

```
dht0002@DAVID-PC: ~/CPE435/Lab06/Code
dht0002@DAVID-PC:~/CPE435/Lab06/Code$ ./runner_serial 10000
Sum      = 3.14179148
Actual   = 3.14159265
Error    = 6.3e-05
Program took 0.264 milliseconds to execute
```

Figure 3: Serial model for 10,000 rectangles

```
dht0002@DAVID-PC: ~/CPE435/Lab06/Code
dht0002@DAVID-PC:~/CPE435/Lab06/Code$ ./runner_parallel 10000 2
Number of rectangles = 10000
Number of threads    = 2
Sum      = 3.14179148
Actual   = 3.14159265
Error    = 6.3e-05
Program took 0.162 milliseconds to execute
```

Figure 4: Pthread model with 2 threads for 10,000 rectangles

Example 3: Serial vs 2 pthreads for 100,000 rectangles

```
dht0002@DAVID-PC: ~/CPE435/Lab06/Code
dht0002@DAVID-PC:~/CPE435/Lab06/Code$ ./runner_serial 100000
Sum      = 3.14161262
Actual   = 3.14159265
Error    = 6.4e-06
Program took 2.077 milliseconds to execute
```

Figure 5: Serial model for 100,000 rectangles

```
dht0002@DAVID-PC: ~/CPE435/Lab06/Code
dht0002@DAVID-PC:~/CPE435/Lab06/Code$ ./runner_parallel 100000 2
Number of rectangles = 100000
Number of threads    = 2
Sum      = 3.14161262
Actual   = 3.14159265
Error    = 6.4e-06
Program took 0.154 milliseconds to execute
```

Figure 6: Pthread model with 2 threads for 100,000 rectangles

From this test, we can see that the 2 pthread model thrives when compared to the serial model.

	Serial Model Execution Time (ms)	Parallel Model Execution Time (ms)
Example 1	0.119	0.157
Example 2	0.264	0.162
Example 3	2.077	0.154

Table 1: Test A results

Test A: Serial vs Parallel Execution Times

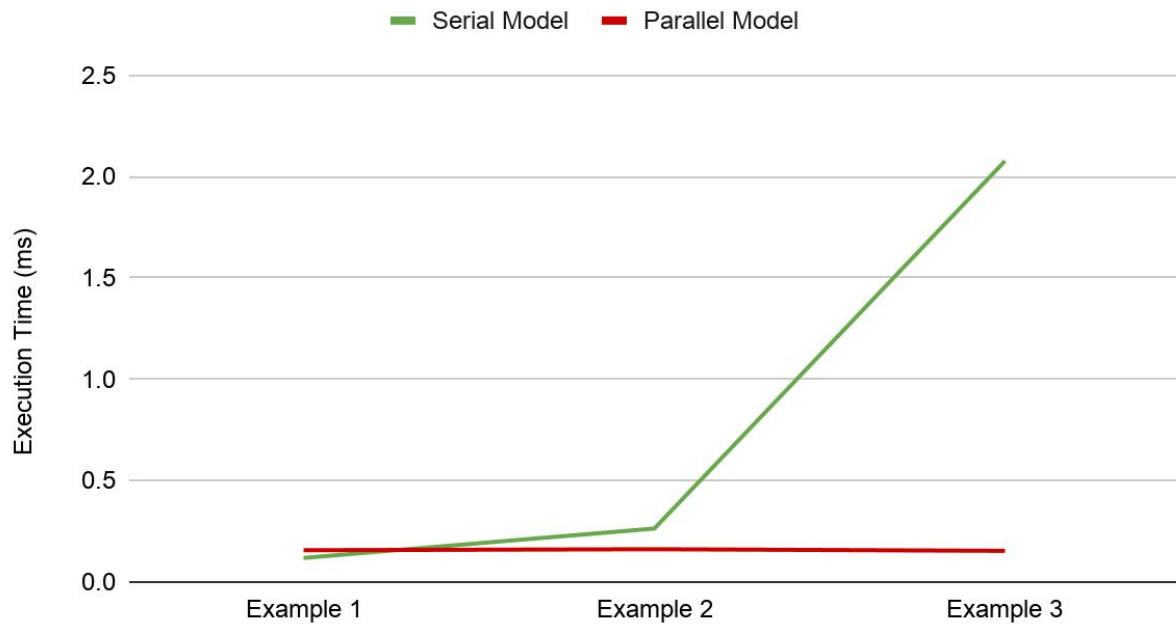
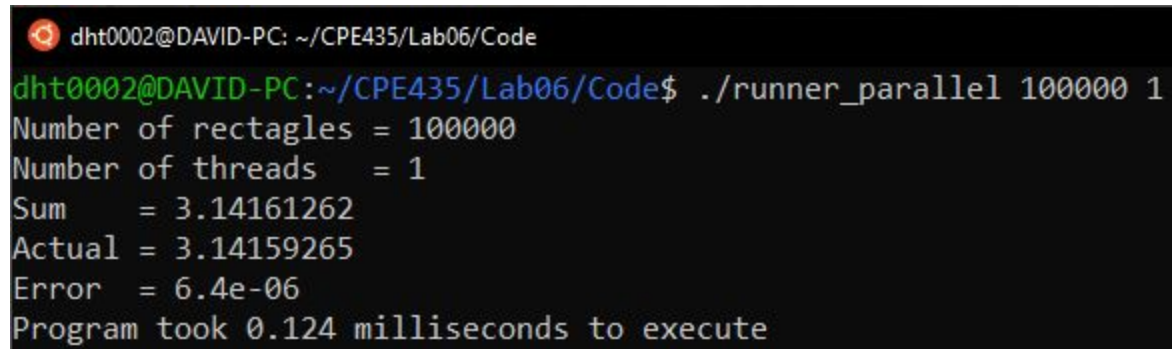


Figure 7: Plotting Serial vs Parallel Execution Timings based on data from Table 1

Test B: Comparison of performance based on execution time

Example 1: Serial vs 1 pthread for 100,000 rectangles



```
dht0002@DAVID-PC: ~/CPE435/Lab06/Code
dht0002@DAVID-PC:~/CPE435/Lab06/Code$ ./runner_parallel 100000 1
Number of rectagles = 100000
Number of threads   = 1
Sum      = 3.14161262
Actual   = 3.14159265
Error    = 6.4e-06
Program took 0.124 milliseconds to execute
```

Figure 8: Pthread model with 1 threads for 100,000 rectangles

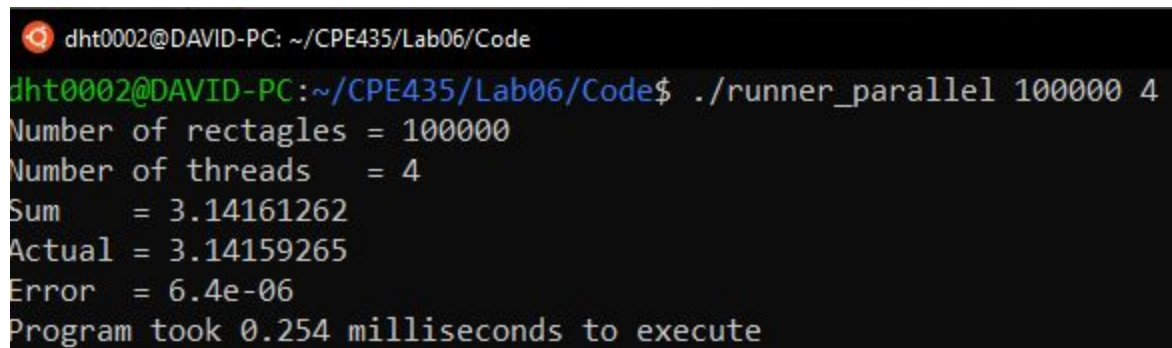
We can gather the serial model results from Figure 5.

Example 2: Serial vs 2 pthreads for 100,000 rectangles

We can gather the serial model results for 100,000 rectangles from Figure 5.

We can gather the 2 pthread model results for 100,000 rectangles from Figure 6.

Example 3: Serial vs 4 pthreads for 100,000 rectangles



```
dht0002@DAVID-PC: ~/CPE435/Lab06/Code
dht0002@DAVID-PC:~/CPE435/Lab06/Code$ ./runner_parallel 100000 4
Number of rectagles = 100000
Number of threads   = 4
Sum      = 3.14161262
Actual   = 3.14159265
Error    = 6.4e-06
Program took 0.254 milliseconds to execute
```

Figure 9: Pthread model with 4 threads for 100,000 rectangles

We can gather the serial model results from Figure 5.

Example 4: Serial vs 8 pthreads for 100,000 rectangles

```
dht0002@DAVID-PC: ~/CPE435/Lab06/Code
dht0002@DAVID-PC:~/CPE435/Lab06/Code$ ./runner_parallel 100000 8
Number of rectagles = 100000
Number of threads   = 8
Sum      = 3.14161262
Actual   = 3.14159265
Error    = 6.4e-06
Program took 0.641 milliseconds to execute
```

Figure 10: Pthread model with 8 threads for 100,000 rectangles
We can gather the serial model results from Figure 5.

Example 5: Serial vs 16 pthreads for 100,000 rectangles

```
dht0002@DAVID-PC: ~/CPE435/Lab06/Code
dht0002@DAVID-PC:~/CPE435/Lab06/Code$ ./runner_parallel 100000 16
Number of rectagles = 100000
Number of threads   = 16
Sum      = 3.14161262
Actual   = 3.14159265
Error    = 6.4e-06
Program took 2.875 milliseconds to execute
```

Figure 11: Pthread model with 16 threads for 100,000 rectangles
We can gather the serial model results from Figure 5.

From this test, we can see that the parallel model does very well when the number of threads is low. If we increase the number of threads, there comes a point where the parallel model performs worse than the serial model.

	Serial Model Execution Time (ms)	Parallel Model Execution Time (ms)
Example 1	2.077	0.124
Example 2	2.077	0.154
Example 3	2.077	0.254
Example 4	2.077	0.641
Example 5	2.077	2.875

Table 2: Test B results

Test B: Serial vs Parallel Execution Times

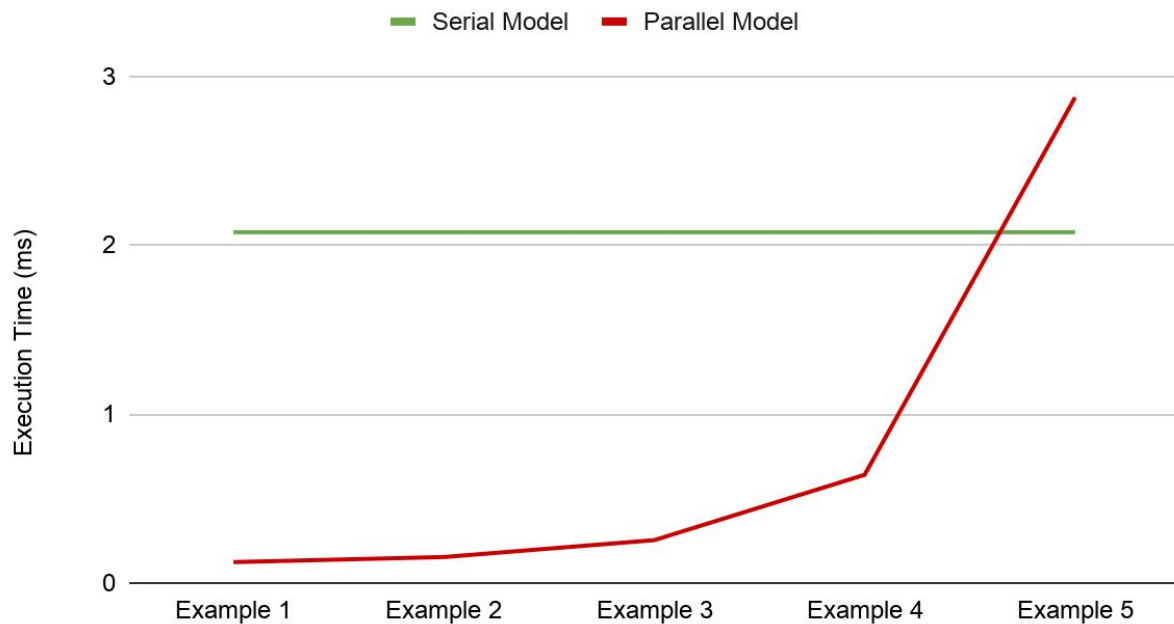


Figure 12: Plotting Serial vs Parallel Execution Timings based on data from Table 2

Conclusion

This lab was successful in expanding my knowledge of pthread implementation and the cost of pthreads.

[Demo link](#)

Appendix

Appendix 1: serial.c

```
// *****
// Program Title: Lab 06
// Project File: serial.c
// Name: David Thornton
// Course Section: CPE-435, SP 2021
// Due Date: 02/22/2021
// *****
#include <stdlib.h>
```



```

#include <stdio.h>
#include <time.h>
#include <math.h>
#define UPPER 1
#define LOWER 0

double getWidth(double upper, double lower, int num);
double getHeight(double width, int i);

int main(int argc, char* argv[])
{
    clock_t t = clock();
    if(argc != 2)
    {
        printf("ERROR!!! Wrong amount of input arguments. Try
again.\nTerminating program...\n");
        return 1;
    }

    unsigned long num_rectangles = atoi(argv[1]);

    if(num_rectangles < 0)
    {
        printf("ERROR!!! Input can not be negative. Try
again.\nTerminating program...\n");
        return -1;
    }

    double sum = 0;
    double width = getWidth(UPPER, LOWER, num_rectangles);

    for(int i = 0; i < num_rectangles; i++)
    {
        sum += width * getHeight(width, i);
    }

    sum *= 4;
    printf("Sum      = %1.9g\n", sum);
    printf("Actual  = %1.9g\n", M_PI);
    double error = (1 - M_PI / sum);
    printf("Error   = %.1e\n", error);
    t = clock() - t;

```

```

    double total_time = ((double)t)/CLOCKS_PER_SEC;
    double ms = total_time * 1000;
    if(ms < 1000)
    {
        printf("Program took %g milliseconds to execute\n", ms);
    }
    else if(ms >= 1000)
    {
        printf("Program took %g seconds to execute\n", total_time);
    }
    return 0;
}

double getWidth(double upper, double lower, int num)
{
    double width = ((upper - lower) / num);
    return width;
}

double getHeight(double width, int i)
{
    double x = width * i;
    double height = sqrt(1 - pow(x, 2));
    return height;
}

// gcc serial.c -o runner_serial -Wall -lm

```

Appendix 2: parallel.c

```

// *****
// Program Title: Lab 06
// Project File: parallel.c
// Name: David Thornton
// Course Section: CPE-435, SP 2021
// Due Date: 02/22/2021
// *****

#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>

```

```

#include <stdio.h>
#include <time.h>
#include <math.h>
#define LENGTH 1

double sum, dx;
unsigned long num_threads;
unsigned long num_rectangles;
double f(double inval);
void* updateSum(void* argument);
pthread_mutex_t myMutex;

int main(int argc, char* argv[])
{
    clock_t t = clock();
    if(argc != 3)
    {
        printf("ERROR!!! Wrong amount of input arguments. Try
again.\nTerminating program...\n");
        return 1;
    }
    sum = 0;
    num_rectangles = atoi(argv[1]);
    num_threads = atoi(argv[2]);
    printf("Number of rectangles = %ld\n", num_rectangles);
    printf("Number of threads   = %ld\n", num_threads);
    dx = LENGTH / (double)num_rectangles; // width of 1 rectangle
    pthread_t myThreads[num_threads];
    int start[num_threads];

    if(num_rectangles < 0)
    {
        printf("ERROR!!! Input can not be negative. Try
again.\nTerminating program...\n");
        return 2;
    }

    if(pthread_mutex_init(&myMutex, NULL))
    {
        printf("ERROR!!! Cannot init mutex lock. Try
again.\nTerminating program...\n");
        return 3;
    }
}

```

```

    }

    for(int i = 0; i < num_threads; i++)
    {
        start[i] = i; // tell each of the threads where to start
        if(pthread_create(&myThreads[i], NULL, updateSum, &start[i]))
        {
            printf("ERROR!!! Could not create the
threads.\nTerminating program...\n");
            return 4;
        }
    }

    for(int i = 0; i < num_threads; i++)
    {
        int retStatus = pthread_join(myThreads[i], NULL);
        if(retStatus)
        {
            printf("ERROR!!! Problem at thread id %d.\nError
number:%d\n", i, retStatus);
        }
    }

    pthread_mutex_destroy(&myMutex);
    sum *= 4;
    printf("Sum      = %1.9g\n", sum);
    printf("Actual = %1.9g\n", M_PI);
    double error = (1 - M_PI / sum);
    printf("Error   = %.1e\n", error);
    t = clock() - t;
    double total_time = ((double)t)/CLOCKS_PER_SEC;
    double ms = total_time * 1000;
    if(ms < 1000)
    {
        printf("Program took %g milliseconds to execute\n", ms);
    }
    else if(ms >= 1000)
    {
        printf("Program took %g seconds to execute\n", total_time);
    }
    return 0;
}

```

```
double f(double inval)
{
    return(sqrt(1 - pow(inval, 2)));
}

void* updateSum(void* arg)
{
    double x, temp;
    int myStart = *(int*) arg;
    temp = 0; // make sure the sum is 0 for each thread call
    for (unsigned long i = myStart; i < num_rectangles; i += num_threads)
    {
        x = dx * i;
        temp += dx * f(x);
    }
    pthread_mutex_lock(&myMutex);
    sum += temp;
    pthread_mutex_unlock(&myMutex);
    return(NULL);
}

// Compile with: gcc parallel.c -o runner_parallel -Wall -lm -pthread
```