Assignment 3

Operating Systems | Anthony Nguyen

p1.c = Program 1 source code | **p2.c** = Program 2 source code

I confirm that I will keep the content of this assignment confidential. I confirm that I have not received any unauthorized assistance in preparing for or writing this assignment. I acknowledge that a mark of 0 maybe assigned for copied work."
Anthony Nguyen | 104792283

Implementation Details and Explanation:

p1.c (program 1):

```
double random_double()
{
    return random() / ((double) RAND_MAX + 1);
}
```

Figure 1: random_double (program 1)

Figure 1: random_double is a function that returns a double between 0 and 1 (exclusive). The random function generates a number between 0 and RAND_MAX, then if you divide that number by one integer more than RAND_MAX you will always get a value less than 1 and more than 0. The double cast on RAND_MAX is used to guarantee a double after division and not an integer.

```
18  void *thread_calc(void *arg)
19  {
20      // Variable declaration
21      double x,y;          // for cartesian coordina
22      int hit_count;       // tracks # of points gen
23
24      // loop used to generate points and calculate
25      for(int i = 0 ; i < NUMBER_OF_POINTS ; ++i)
26      {
27          // generate random number between -1 and
28          x = random_double() * 2.0 - 1.0;
29          y = random_double() * 2.0 - 1.0;
30
31          // checks if (x,y) is in the circle
32          if( sqrt(x*x + y*y) < 1.0 )
33              ++hit_count;         // increment coun
34      }
35      circle_count = hit_count;    // set global var
36  }
```

Figure 2: thread_calc (program 1)

Figure 2: The thread_calc is a thread function that loops based on the NUMBER_OF_POINTS value. Line 28 and 29 calls random_double() (figure 1) which outputs a value we will call v such that 0 < v < 1. This v is then multiplied by 2 and subtracted by 1 resulting in an output from -1 and 1 (exclusive). Therefore, the x and y variables contain a value from -1 and 1 (exclusive). The if statement on line 32 checks if these x/y coordinates are within the circle to decide if it will increase the hit_count variable that keeps track of number of points in the circle.

After the loop ends, we would have successfully generated the correct number of points (based on macro definition) and a count of the number of generated points in the circle. The last step assigns the global variable "circle_count" to "hit_count" to be used later for the pi estimation in main.

```
38  int main()
39  {
40      // Seeds the random number generator so you get different random numbers
41      srandom((unsigned)time(NULL));
42
43      // Variables
44      pthread_t t;                    // For the single thread (slave)
45      struct timespec start, finish;  // time spec used to measure timnes
46      double time_taken;              // used to calculate time
47
48      // The use of clock() gave incorrect times with running threads (due to cp
49      // and found online that clock_gettime() was much more accurate
50      clock_gettime(CLOCK_MONOTONIC, &start);                 // grab time (the
51
52      pthread_create(&t, NULL, thread_calc, NULL);        // create the slave th
53      pthread_join(t, NULL);                              // waits for the threa
54
55      clock_gettime(CLOCK_MONOTONIC, &finish);                // grab time (the
56      time_taken = (finish.tv_sec - start.tv_sec);            // get the elapsed
57      time_taken += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;  // add the
58
59      printf("Points: %d\n", NUMBER_OF_POINTS);
60      printf("pi estimation: %f\n", 4*(circle_count/(double)NUMBER_OF_POINTS));
61      printf("Time Taken: %f\n", time_taken);
62
63      return 0;
64  }
```

Figure 3 – main (program 1)

Figure 3: The main of program 1 first seeds the random number generator so that it does not keep providing the same random numbers after every run. Then on line 52, I create a single thread "t" and assign it to the "thread_calc" routine (figure 2). The thread_calc does calculations (explained above) and sets the global variable circle_count to the correct value. Then after in our main I have a pthread_join for "t" to wait till thread finishes. I then output the pi estimation with the given calculation to use with the Monte Carlo technique.

I should note that I also used clock_gettime function throughout the code to measure how long the thread takes to executes. Using the clock method was very inaccurate since it measures cpu time and running threads will mess with actual cpu times. Therefore, using this clock_gettime that sets a timespec struct when called, you can easily calculate the elapsed time (ie. Line 56/57).

p2.c (program 2):

Note: Program 2 is very similar to program 1 but with some modifications. So, some parts will not be fully explained since it has been explained in program 1 (above).

```
#define NUMBER_OF_POINTS 1000000
#define NUMBER_OF_SLAVES 100

// global variable
int circle_count = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Figure 4: Mutex & # of slave's addition (program 2)

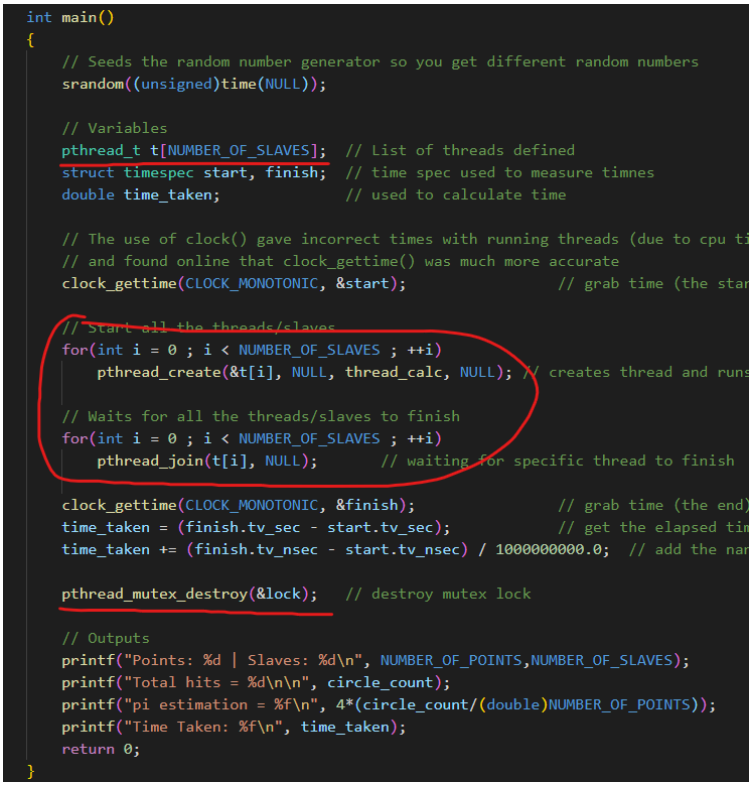
Figure 4: One addition to program 2 is the NUMBER_OF_SLAVES macro that will be used to determine how many slaves (threads) the program will run. There is also a pthread_mutex_t variable "lock" that is initialized statically and will be used for locking critical areas.

```
void *thread_calc(void *arg)
{
    // Variable declaration
    double x,y;         // for cartesian coordinates bounded in t
    int hit_count;      // tracks # of points generated inside th

    // loop used to generate points and calculate if its inside t
    for(int i = 0 ; i < NUMBER_OF_POINTS/NUMBER_OF_SLAVES ; ++i)
    {
        // generate random number between -1 and 1 (exlusive)
        x = random_double() * 2.0 - 1.0;
        y = random_double() * 2.0 - 1.0;

        // checks if (x,y) is in the circle
        if( sqrt(x*x + y*y) < 1.0 )
            ++hit_count;        // increment counter if (x,y) in
    }

    // Using mutex lock to prevent run condition
    // when trying to access the global variable
    pthread_mutex_lock(&lock);
    circle_count += hit_count;  // add hit_count onto circle_coun
    pthread_mutex_unlock(&lock);
}
```

Figure 5: thread_calc (program 2)

Figure 5: Very similar to program 1 thread_calc (figure 2) but there are 2 main changes. One is that the for-loop loops based on the number of points divided by the number of slaves (macro definitions).

The other change is that the circle_count global variable is being added to by the hit_count calculation instead of being assigned. This is because there will be multiple threads/slaves working on the point generation. Another thing to note is the use of mutex to lock and unlock the circle_count incrementation. This is to prevent the race condition and only allow a single thread to actively change this value one at a time.



```c
int main()
{
    // Seeds the random number generator so you get different random numbers
    srandom((unsigned)time(NULL));

    // Variables
    pthread_t t[NUMBER_OF_SLAVES];  // List of threads defined
    struct timespec start, finish;  // time spec used to measure timnes
    double time_taken;              // used to calculate time

    // The use of clock() gave incorrect times with running threads (due to cpu ti
    // and found online that clock_gettime() was much more accurate
    clock_gettime(CLOCK_MONOTONIC, &start);                // grab time (the star

    // Start all the threads/slaves
    for(int i = 0 ; i < NUMBER_OF_SLAVES ; ++i)
        pthread_create(&t[i], NULL, thread_calc, NULL); // creates thread and runs

    // Waits for all the threads/slaves to finish
    for(int i = 0 ; i < NUMBER_OF_SLAVES ; ++i)
        pthread_join(t[i], NULL);       // waiting for specific thread to finish

    clock_gettime(CLOCK_MONOTONIC, &finish);               // grab time (the end)
    time_taken = (finish.tv_sec - start.tv_sec);           // get the elapsed tim
    time_taken += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;  // add the nan

    pthread_mutex_destroy(&lock);   // destroy mutex lock

    // Outputs
    printf("Points: %d | Slaves: %d\n", NUMBER_OF_POINTS,NUMBER_OF_SLAVES);
    printf("Total hits = %d\n\n", circle_count);
    printf("pi estimation = %f\n", 4*(circle_count/(double)NUMBER_OF_POINTS));
    printf("Time Taken: %f\n", time_taken);
    return 0;
}
```

Figure 6: main (program 2)

Figure 6: The main of program 2 has an array of pthread variable that's size is based on the number of slaves macro definition. For the pthread creation I use a loop to create the correct number of threads to perform the thread_calc function. Also, a loop was used to wait for each thread to finish executing. The last thing that is done is to destroy the mutex lock "lock".

Sample Run of Program 1 and 2:

Program 1

```
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ gcc p1.c -lm -lpthread
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ ./a.out
Points: 100
pi estimation: 3.200000
Time Taken: 0.000266
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ gcc p1.c -lm -lpthread
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ ./a.out
Points: 1000
pi estimation: 3.140000
Time Taken: 0.000272
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ gcc p1.c -lm -lpthread
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ ./a.out
Points: 10000
pi estimation: 3.133200
Time Taken: 0.000564
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ gcc p1.c -lm -lpthread
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ ./a.out
Points: 100000
pi estimation: 3.147920
Time Taken: 0.003386
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ gcc p1.c -lm -lpthread
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ ./a.out
Points: 1000000
pi estimation: 3.144076
Time Taken: 0.031575
```

Program 2

```
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ gcc p2.c -lm -lpthread
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ ./a.out
Points: 1000000 | Slaves: 2
Total hits = 785472

pi estimation = 3.141888
Time Taken: 0.089346
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ gcc p2.c -lm -lpthread
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ ./a.out
Points: 1000000 | Slaves: 20
Total hits = 785966

pi estimation = 3.143864
Time Taken: 0.207834
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ gcc p2.c -lm -lpthread
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ ./a.out
Points: 1000000 | Slaves: 40
Total hits = 785299

pi estimation = 3.141196
Time Taken: 0.207833
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ gcc p2.c -lm -lpthread
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ ./a.out
Points: 1000000 | Slaves: 80
Total hits = 785012

pi estimation = 3.140048
Time Taken: 0.204941
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ gcc p2.c -lm -lpthread
anthony@LAPTOP-NERLKBT4:.../Assignment_03$ ./a.out
Points: 1000000 | Slaves: 100
Total hits = 785532

pi estimation = 3.142128
Time Taken: 0.209033
```

Program 1 (one slave thread)	Pi Estimation	Elapsed Time (seconds)
100 points	3.2	0.000266
1,000 points	3.14	0.000272
10,000 points	3.1332	0.000564
100,000 points	3.14792	0.003386
1,000,000 points	3.14076	0.031575

Program 2 (1 million points)	Pi Estimation	Elapsed Time (seconds)
2 slaves	3.141888	0.089346
20 slaves	3.143864	0.207834
40 slaves	3.141196	0.207833
80 slaves	3.140048	0.204941
100 slaves	3.142128	0.209033d

(1)

The results from running program 1 with various number of points show some very significant pi estimation and run time. First thing that should be noted is that it is very noticeable that the higher number of points leads to a more accurate pi estimation in this Monte Carlo technique. If you look at the table (blue) you will notice that there is a slight pattern where the higher number of points generated leads to a value closer to pi. It should be noted that the pattern is not exactly sound due to randomization and probability. This is the reason for my run on 1,000 points gains a better pi estimation than at 10,000 points. This can be considered outlier but considering probability if I were to run it again the results would most likely output the opposite. All in all, it is quite noticeable that higher number of points generated leads to a more accurate pi estimation (hence the 1 million points having the closest estimation). The run time of the thread program calculations shows that the increase in points generated leads to an increase in run time execution. This makes sense since the loop for generating points and doing calculations would increase with a higher number of points.

There is statement that states: "a multi threaded program (program 1) should run faster than a non-threaded program". I would normally agree with this statement, but I would disagree with this in this instance. In program 1 this "multi threaded program" acts more like a non threaded program since you only create a single slave to do the work and do not make use of splitting the work. So, comparing this to a program that uses no threads to do the work, the run time should be almost the same. If anything, the non-threaded program may even run faster since it does not have the overhead of creating a thread just to do the work in that single thread (slave).

(2)

(i) your multi-threaded Program-2 should run faster than Program-1

Based on my output times the multi threaded program (program 2) does not run faster than my program 1. I tested this on my computer with a 4 core cpu and in theory it should run faster for a certain number of threads/slaves. However, as we can see the times are quite the opposite. This can be due to overheads of splitting and creating the threads, but the 2 slaves SHOULD have run faster (I have hypothesis at the end of page on why it does not).

(ii) a larger value of NUMBER_OF_SLAVES should give a faster running time than a smaller value

The run times on program 2 shows that more threads take more time to execute. It should be noted that there is not a consistent run time pattern recorded between 20, 40, 80, and 100 slaves (the times were essentially the same). Most likely the overhead of splitting and creating the threads far exceed the benefits when running many threads and can be used to explain why running many threads does not scale infinitely. The

number of cores also plays a large factor on how much threads your program should run before performance is dropped. I believe running 2, 4, 6, or 8 threads on a 4 core CPU (what I have) would be much more efficient than the testing 20,40 ,80 and100.

(iii) what can you say about the estimated pi values as NUMBER_OF_SLAVES changes from 2 to 100?

It is very noticeable that the number of slaves (threads) ran for program 2 does not affect the pi estimation. You will notice that the average difference between each pi estimation tested from running 2 to 100 threads is about 0.001. Each pi estimation is essentially around 3.14 with very slight difference. This makes sense because the pi estimation using the Monte Carlo technique only depends on the number of generated points and not the number of threads generating the points.

Some Important Notes:

I am fairly certain that the program 2 timing was skewed and slower because of the random() function used. I've done a test without the use of the random() function and found that the multi threaded program ran faster on a certain number of slaves then just on a single slave. As in, testing 2, 4, and 8 slaves ran faster than just using 1 slav. Of course, increasing this number of slaves started to slow down the program which makes sense because of the limited hardware and overhead caused. This is why I believe most of the run times calculated in program 2 is invalid and skewed.

The explanation for the random function causing this phenomenon is quite simple, the random function is blocking the threads running because the random function is not thread compatible. Therefore running 2 slaves is somehow slower than running 1 slave. A solution would be to implement a random function with states, so each thread has its own local random instead of each thread sharing the random seed.

How to compile:

```
gcc p1.c -lm -lpthread
gcc l2.c -lm -lpthread
```