COMP 3300

Assignment 3

February 13th, 2020

Bilal Malik

104435995

Hasan Malik

104610462

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 may be assigned for copied work. Bilal Malik 104435995.

Program 1

compile this with ‘gcc -pthread assign3\_program1.c -lm’

This program begins by prompting the user to enter the number of points. This value entered is stored in the num\_points variable holder, printed out as confirmation to the user, then passed to the num\_of\_points\_calc() function. In the function, it is divided by the total number of threads, a constant value defined at the top of the C program as two. The two threads have their own relationships, one as a master thread and one as a slave thread. An array of type thread is created the size of that same constant. This array will house all the threads created to run this computation. A random number is generated based on the time. This number can never repeat; thus, it will give points to dot on the plane that are always unique. The clock as well is started to help in timing the computation, and a for loop is ran through the array of threads, create each one, and have them tasked to run the runner function. Here is where the computation is completed by the individual threads. The number of points that exact thread is also passed in the creation by the points\_per\_thread variable. The runner function then takes the point\_per\_thread value as a parameter and stores it in the POINTS variable. This is the total number of points the thread needs to check for to see if it in or not in the circle. Each thread then generates a point from two random integers (this is done by calling a separate function which implements the random() function), a total number of times that it is supposed to generate points for, utilizing the POINTS variable holder. It checks if each point hits the circle or not, using the distance formula sqrt(x^2+y^2) and seeing if this value is less then 1, the radius of the circle. If this is the case it increments the hit\_count variable holder and then increments the global circle\_count variable by this value. This way all the threads can run there hit check, then increment by that much, the global variable holder. The global variable can be placed in a race condition when to threads may try to increment at the same time, losing the data. Final the pthread is killed through pthread\_exit and its data is collected form the pthread\_join() call. Lastly, Monte Carlo’s theorem is utilized, the global variable circle\_count accounts for the total number of hits on the circle once all child threads have terminated. This value is multiplied by 4, and then divided by the total number of points, the variable that was first prompted from the user at the very beginning of the code. This value is stored in the PI variable. The clock is stopped, finally the speed is calculated by seeing the entire time the code was running for subtract int the start instance of the clock() with the time in the begin to give us the total time of the code. Finally, the total number of points is printed out, the value of Pi, and the time it took to reach this answer.

I run this program 4 times with varying inputs for the number of points. I start at a smaller number, and work my way to larger numbers. First I run the program with 5 as the number of points.

A screenshot of a computer screen

Description automatically generated

As it can only utilize five points to give an estimate for pi, it responds with 1.600 for the value of pi, a very far cry from the actual value of (3.14159265). As the number of points increases, the estimated value of pi will converge with the real value.

We then run with the number of points as 5 and get an answer for pi as 3.2000

A screenshot of a computer screen

Description automatically generated

Now the threads can compute with 100 points to work with between them. Thus, a more accurate value is displayed that already has the correct single digit value. We can see as we continue too increase the number off points, we also increase the opportunities for a closer evaluation. The time as well begins to slow down, as the number of points increase, so does the workload of the threads. As we continue to increase the number of points, the time will continue to slow down.

Next the program takes in 1,000 as the number of points to be created. This results in pi being 3.044. This situation is understandable, because the points are randomly generated, there is a chance if standard error.

A screenshot of a computer screen

Description automatically generated

Though the more points you generate, the more likely you will get a closer answer, at times points will be generated that just are not good sample values and throw the computation of. In this case the difference away from pi has increased but only by a small fraction. Still the point is made, that after multiple runs the value for pi is now hovering around the actual value and will get even closer. The time it takes to run the computation, continues to slow down, as was expected when increasing the load of work for the same number of threads.

Finally, the program is run with 10,000 as the value for number of points. The value of pi is now estimated at 3.13200. This is the closest the estimation has gotten to pi, in comparison with previous runs. We can see that with larger number of points, the estimate will continue to converge with the actual value of pi. Also, this execution is slowest of all the executions that took place, again to do with the increased workload of each thread.

A screenshot of a computer screen

Description automatically generated

Program 2

The implementation begins the same, prompting the user for the number of points and storing it in num\_points. This time OpenMP allows the use of threads implicitely without having to create and join them. Srandom is utilized again to ensure that random gives new values at a different time.The number of points is then passed to the num\_of\_point\_calc() function. Here, the clock() is used to get the time prior to the threaded execution. Then the ‘#pragma omp parallel’ call is made such that the following block of code is to be executed in parallel by the multiple threads instead of one at a time. The logic follows in this block, such that a loop is made, that will not end till the total number of points is reached. Thus, instead of divying up the number of points between threads, the same fixed number of points is too be completed by each thread. Since the value is shared between the threads, they will all increment the value in the loop if one incrementrs it, all threads will run a fraction of the total number of points. Again, two integers are randomly created and checked if they are inside of the function through the distance formula. If this is the case, then to increment the local hit\_count variable and then add this value after the thread is finished creating points, to the global circle\_count variable. Since, this circle\_count value is global, and shared between the threads, a race condition can arise by having multiple threads attempting to increment the count at the same time. Monte Carlo’s formula is utilized again to compute pi. Since, each process ran the number of points seperaely instead of sharing, the number of points needs to be multiplied with the number of processes un the omp\_get. This will give the true number of points utilized in total to compute this value. This value then divides the circle\_count multiplied by 4; and is stored in pi. The end instance of the clock is used, and this number is subtracted by the original, to give the complete time it took for the processes to compute the problem.

Following are sample runs of the program. We utilize 5 at first two compare to the run in the first program.

A screenshot of a computer screen

Description automatically generated

Pi is computed to 3.3000 and does this quicker then the first part. This reason for speed up, can be pointed towards the fact that these threads run in parallel instead of sequentially through priority. Again, as we increase the number of points, the accuracy of pi should increase and the time it takes should increase as well.

The next run uses 100 points to be comparable to the 100 point run in the second execution of program 1.

A screenshot of a computer screen

Description automatically generated

This again is faster then its counter part and more accurate yet slower then the previous run of program 2.

We try for 1,000 as the number of points, and again should see convergence at pie as well as a slow down in speed, yet faster than its program 1 equivalent. This is the case.

A screenshot of a computer screen

Description automatically generated

And one last time with 10,000, which proves that this works faster and more accurately then its program 1 counterpart as well as being more accurate then the previous program 2 execution at the expense of speed.

A screenshot of a computer screen

Description automatically generated

Program 3

The code is implemented similarly to program 1’s code outside of the utilization of a lock. The number of threads is initialized as 1 and number of points a 1000000. The user is prompted with the number of slave threads to utilize in the program. Before storing it and printing to confirm with the user as well as passing it to the next function. Inside the diff\_num\_of\_slaves function, it then divides the total points between the number of threads. An array of threads is created at the size of the number of threads. Then, the mutex lock is initialized, and the clock begins with a for loop creating all the threads which then run the runner function. The logic in the runner is the same, with random points being created that are unique from the previous because of the srandom, again it increments a hit count, which after the for, increments a global variable. Here when the global variable is to be incremented, a lock occurs. This is to remove any case of the race condition by having only one thread increment at a time. It is then unlocked to allow other threads to increment the variable. Once the global counter has been incremented, and all slave threads have been terminated, the number of points and slaves is printed out. With the estimation of pi and the time it took to complete the task.

The first run is tested with 1 slave. As the number of slaves grow, the accuracy will not change as they are still computing the same number of points, only the time at which it takes to complete the task will speed up.

A screenshot of a computer screen

Description automatically generated

Now we compare this with running ten slaves. We receive almost identical values as the previous, but at a tenth of the speed. The is because the same static number of points is being utilized, only that more slaves can now work to compute the number of points, thus decreasing speed.

A screenshot of a computer screen

Description automatically generated

As we continue to increase the number of slaves, the accuracy towards pi will float around the same number, yet the speed will continually increase. Here we run 100 slaves and see that our predictions are true.

A screenshot of a computer screen

Description automatically generated

This pattern will continue, with the more slave threads we add.

Program 4

This program is logically built the same way as program two. It utilizes OpenMP to run threads implicitly. Again, the pragma omp parallel allows for threads to run in parallel. The different is in the incrementing of the global variable for the count. ‘#pragma omp critical’ creates a critical section that can only be entered one at a time. This only one thread can increment and store this value before then next. This helps to remove the race condition seen in program 2. We start by prompting the user for the number of slaves, before passing this value to a sperate function. This function then gives the time before the operation occurs, with clock and the srandom is utilized to create unique random values. Inside of the ‘#pragma omp parallel’ block of code, the parallel execution of threads begins. Again, a for loop is created that creates a point through to random integers, checks if this point is in the circle or not and increments the counter accordingly. When the thread exits the for loop and attempts to increment the global variable, it must be the only one entering its critical section at that time. If there is another thread in its critical section, then the current thread will be in a blocked state in a queue waiting to be utilized next. Again, this lock allows for more accurate estimations of pi immediately. The same reaction on executions will occur as in program 3, as the number of slaves increase, the quicker the code is completed and the accuracy will stay the same from run to run. In comparison to program 3, where locking is also utilized in single threading functions, this will move faster, as well it will move faster then the other two previous programs, making it the fastest in the assignment.

The following is an execution with 1 slave threads. As is the case, it moves much faster than the previous three programs and is able to hit an accurate estimation to pi immediately.

A screenshot of a computer screen

Description automatically generated

The following is an execution with 10 slave threads. This is even faster then its previous program 4 execution, due to the added number of threads working the same number of points. As well this is faster then the program 2 execution of 10 points due to its use of parallelization and locking together versus simply locking.

A screenshot of a computer screen

Description automatically generated

The following is an execution with 100 slave threads. The pattern continues. Predictions can be drawn, that as the number of slaves goes up, the task they need to do in parallel decreases, speeding up execution.

Program 2 and Program 4

Parallelization of the program occurs by involving the #pragma omp parallel statement before a block of code. This code then executes in parallel, to utilize this command you need to include the omp.h library.

A screenshot of a computer

Description automatically generated

Program 3 and Program 4

The protection of race condition occurred through utilizing locks. In program 3 a mutex lock was utilized, and when the line where the race condition occurred appeared, to have the lock become locked, increment then unlock. In program 4, utilizing the #pragma omp critical statement allowed the following block of code to be run as a critical section. Meaning only one thread has access to this section at a time.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated