**Assignment 3**

**Parallel Programming**

**Student ID: 2018380130**

**Student Name: Khan Md Shahedul Islam**

**Data：2021/10/15**

***Answer 1:***

The chapter four was about shared-memory, threads and pthreads. Reading this chapter taught me about how to start, run and stop threads. How to perform error checking and various other approaches that can be taken towards threads. I have learned about matrix-vector multiplication, Busy -waiting, mutexes, producer-consumer synchronization and semaphores. I have acquired valuable knowledges on Barriers and Condition variables, read-write locks, caches, cache coherence and false sharing. Alongside, I have learned about thread-safety as well. After going through and learning the contents of this chapter I have applied those in the tasks that were assigned to me.

***Answer 2:***

***i.***

For the given program the code will remain unchanged if the column is not evenly divisible by the number of threads(t). However, if the number of rows isn’t evenly divisible by the number

of threads I need to decide who will be responsible for the rows left over after integer division of m, the number of rows, by t, the number of threads.

For example, if m = 10 and t = 4, then there are two “extra” rows, since 10 % 4 = 2.

Clearly, there are a number of possibilities here. One is to give the first two threads each an extra row. So, I’d assign three rows to threads 0 and 1 and two rows to threads 2 and 3.

The algorithm will be as followed:

int quotient = m/t; /\* Every thread gets at least m/t rows \*/

int remainder = m % t;

if (my\_rank < remainder) { /\* I get m/t + 1 rows \*/

local\_m = quotient + 1;

my\_first\_row = my\_rank\*local\_m;

my\_last\_row = my\_first\_row + local\_m - 1;

} else { /\* I get m/t rows \*/

local\_m = quotient;

/\* Each of the threads 0, 1, . . . , remainder - 1 gets \*/

/\* an extra row. So add in these rows to get my\_first\_row \*/

my\_first\_row = my\_rank\*local\_m + remainder;

my\_last\_row = my\_first\_row + local\_m - 1;

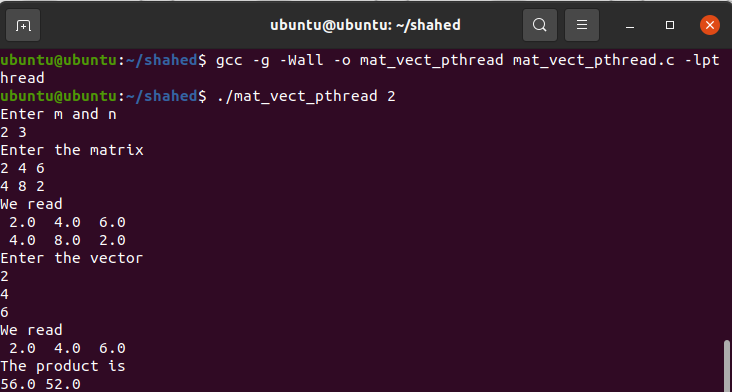
}

***ii.***

For this program, ‘n’ will be representing the columns. Changing columns will not leave any impact on the code, it will rather remain unchanged. As long the rows are evenly divisible by (t) the program will keep performing. This are the tests that I have conducted:

1. t = 2

n = 3



1. t = 3

n = 4



**Analyze:**

Here we can see that, changing `n` with different `t` doesn’t change the

Way the code works and I am getting the result I should be getting.

***Answer 3:***

**Mutex:**

Mutex or Mutual Exclusion Object is used to give access to a resource to only one process at a time. The mutex object allows all the processes to use the same resource but at a time, only one process is allowed to use the resource. Mutex uses the lock-based technique to handle the critical section problem.

Whenever a process requests for a resource from the system, then the system will create a mutex object with a unique name or ID. So, whenever the process wants to use that resource, then the process occupies a lock on the object. After locking, the process uses the resource and finally releases the mutex object. After that, other processes can create the mutex object in the same manner and use it.

By locking the object, that particular resource is allocated to that particular process and no other process can take that resource. So, in the critical section, no other processes are allowed to use the shared resource. In this way, the process synchronization can be achieved with the help of a mutex object.

**Semaphore:**

Semaphore is an integer variable S, that is initialized with the number of resources present in the system and is used for process synchronization. It uses two functions to change the value of S i.e. wait() and signal(). Both these functions are used to modify the value of semaphore but the functions allow only one process to change the value at a particular time i.e. no two processes can change the value of semaphore simultaneously. There are two categories of semaphores i.e. Counting semaphores and Binary semaphores.

In Counting semaphores, firstly, the semaphore variable is initialized with the number of resources available. After that, whenever a process needs some resource, then the wait() function is called and the value of the semaphore variable is decreased by one. The process then uses the resource and after using the resource, the signal() function is called and the value of the semaphore variable is increased by one. So, when the value of the semaphore variable goes to 0 i.e all the resources are taken by the process and there is no resource left to be used, then if some other process wants to use resources then that process has to wait for its turn. In this way, we achieve the process synchronization.

In Binary semaphores, the value of the semaphore variable will be 0 or 1. Initially, the value of semaphore variable is set to 1 and if some process wants to use some resource then the wait() function is called and the value of the semaphore is changed to 0 from 1. The process then uses the resource and when it releases the resource then the signal() function is called and the value of the semaphore variable is increased to 1. If at a particular instant of time, the value of the semaphore variable is 0 and some other process wants to use the same resource then it has to wait for the release of the resource by the previous process. In this way, process synchronization can be achieved. It is similar to mutex but here locking is not performed.

**Using semaphore instead of mutex version:**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <pthread.h>

#include <semaphore.h>

#include "timer.h"

const int MAX\_THREADS = 1024;

long thread\_count;

long long n;

double sum;

sem\_t sem;

void\* Thread\_sum(void\* rank);

/\* Only executed by main thread \*/

void Get\_args(int argc, char\* argv[]);

void Usage(char\* prog\_name);

double Serial\_pi(long long n);

int main(int argc, char\* argv[]) {

long thread; /\* Use long in case of a 64-bit system \*/

pthread\_t\* thread\_handles;

double start, finish, elapsed;

/\* Get number of threads from command line \*/

Get\_args(argc, argv);

thread\_handles = (pthread\_t\*) malloc (thread\_count\*sizeof(pthread\_t));

sem\_init(&sem, 0, 1);

sum = 0.0;

GET\_TIME(start);

for (thread = 0; thread < thread\_count; thread++)

pthread\_create(&thread\_handles[thread], NULL,

Thread\_sum, (void\*)thread);

for (thread = 0; thread < thread\_count; thread++)

pthread\_join(thread\_handles[thread], NULL);

GET\_TIME(finish);

elapsed = finish - start;

sum = 4.0\*sum;

printf("With n = %lld terms,\n", n);

printf(" Our estimate of pi = %.15f\n", sum);

printf("The elapsed time is %e seconds\n", elapsed);

GET\_TIME(start);

sum = Serial\_pi(n);

GET\_TIME(finish);

elapsed = finish - start;

printf(" Single thread est = %.15f\n", sum);

printf("The elapsed time is %e seconds\n", elapsed);

printf(" pi = %.15f\n", 4.0\*atan(1.0));

sem\_destroy(&sem);

free(thread\_handles);

return 0;

} /\* main \*/

/\*------------------------------------------------------------------\*/

void\* Thread\_sum(void\* rank) {

long my\_rank = (long) rank;

double factor;

long long i;

long long my\_n = n/thread\_count;

long long my\_first\_i = my\_n\*my\_rank;

long long my\_last\_i = my\_first\_i + my\_n;

double my\_sum = 0.0;

if (my\_first\_i % 2 == 0)

factor = 1.0;

else

factor = -1.0;

for (i = my\_first\_i; i < my\_last\_i; i++, factor = -factor) {

my\_sum += factor/(2\*i+1);

}

sem\_wait(&sem);

sum += my\_sum;

sem\_post(&sem);

return NULL;

} /\* Thread\_sum \*/

/\*------------------------------------------------------------------

\* Function: Serial\_pi

\* Purpose: Estimate pi using 1 thread

\* In arg: n

\* Return val: Estimate of pi using n terms of Maclaurin series

\*/

double Serial\_pi(long long n) {

double sum = 0.0;

long long i;

double factor = 1.0;

for (i = 0; i < n; i++, factor = -factor) {

sum += factor/(2\*i+1);

}

return 4.0\*sum;

} /\* Serial\_pi \*/

/\*------------------------------------------------------------------

\* Function: Get\_args

\* Purpose: Get the command line args

\* In args: argc, argv

\* Globals out: thread\_count, n

\*/

void Get\_args(int argc, char\* argv[]) {

if (argc != 3) Usage(argv[0]);

thread\_count = strtol(argv[1], NULL, 10);

if (thread\_count <= 0 || thread\_count > MAX\_THREADS) Usage(argv[0]);

n = strtoll(argv[2], NULL, 10);

if (n <= 0) Usage(argv[0]);

} /\* Get\_args \*/

/\*------------------------------------------------------------------

\* Function: Usage

\* Purpose: Print a message explaining how to run the program

\* In arg: prog\_name

\*/

void Usage(char\* prog\_name) {

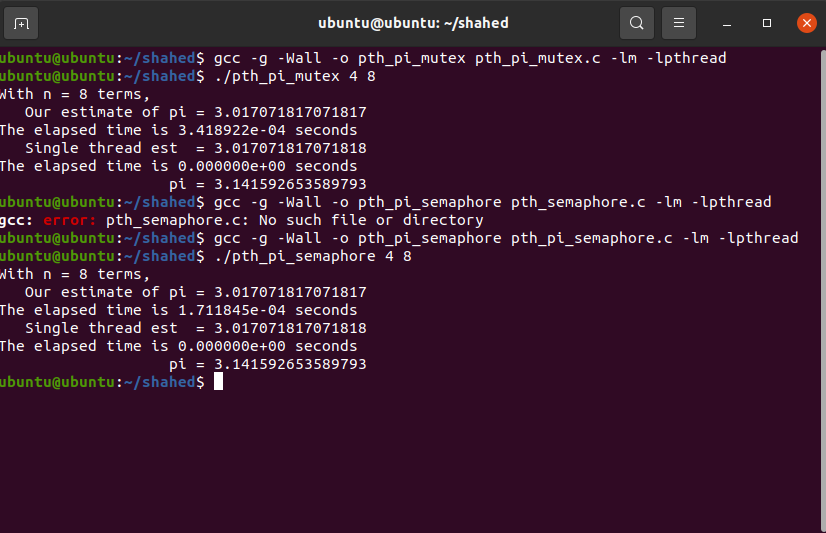
fprintf(stderr, "usage: %s <number of threads> <n>\n", prog\_name);

fprintf(stderr, " n is the number of terms and should be >= 1\n");

fprintf(stderr, " n should be evenly divisible by the number of threads\n");

exit(0);

} /\* Usage \*/



**Analyzation:**

I have used semaphore instead mutex and ran both programs. But the runtime in both cases appeared to be similar. Therefore, we can say in this particular case the performance is same for using both approaches.

**The End!**