Module:	Introduction to Parallel Programming Techniques
Module ID:	EE4107
Student Name:	Umid Narziev
Student ID:	200234832
Assignment Number:	4
Academic Year:	2020 - 2021

Contents

System Specification

Below, shown the system which was used to run and get the result of the simulation:

CPU:	Intel i5-7200U
Architecture:	Kaby Lake
Segment:	Mobile Processors
The number of cores:	2
Number of threads	4
Clock Frequency	2.50-3.10GHz (Turbo Boost)
Cache levels:	3
Cache level 1 size:	128KBytes
Cache level 2 size:	512Kbytes
Cache level 3 size:	3MBytes
RAM	12 GB
SSD:	250 GB
Operating System:	Ubuntu 20.04.2 LTS
Compiler:	Gcc and its libraries
IDE:	Clion (2020.03)

TASK 1

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include "timer.h"

long thread_count;
long long int number_of_tosses;
long long int number_in_circle;
pthread_mutex_t mutex;

void *estimate_PI(void *rank);

void Usage(char *prog_name);

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;
   if (argc != 1) Usage(argv[0]);
   pthread_mutex_init(&mutex, NULL);
   for (long long int number_of_toss = 1000; number_of_toss <= 100000000; number_of_toss *= 100) {
      for (int thread_number = 1; thread_number < 16; thread_number <<= 1) {
         thread_count = thread_number;
         number_of_tosses = number_of_toss;
         number_in_circle = 0;
         thread_handles = (pthread_t *) malloc(thread_count * sizeof(pthread_t));
         GET_TIME(start);
         for (thread = 0; thread < thread_count; thread++)
            pthread_create(thread_handles + thread, NULL, (void *) estimate_PI,
                        (void *) thread);       /* or &thread_handles[thread]*/

         for (thread = 0; thread < thread_count; thread++)
            pthread_join(*(thread_handles + thread), NULL);                 /* or thread_handles[thread]  */


         double pi_estimate = 4 * ((double) number_in_circle / (double) (number_of_tosses));
         GET_TIME(finish);
         elapsed = finish - start;

         printf("pi_estimate -> %f, elapsed_time -> %0.3e, thread_count -> %ld, number_of_tosses -> %lld\n",
               pi_estimate, elapsed, thread_count, number_of_tosses);
      }
   }
   pthread_mutex_destroy(&mutex);
   free(thread_handles);
   return 0;

}


void *estimate_PI(void *rank) {

   long my_rank = (long) rank;
   long long int local_number_tosses = number_of_tosses / thread_count;
   double local_x = 0;
   double local_y = 0;
   double distance_squared = 0;
```

```c
    long long int local_number_in_circle = 0;
    srand(my_rank);
    for (long long int i = 0; i < local_number_tosses; ++i) {
        local_x = rand() / (RAND_MAX + 1.0);
        local_y = rand() / (RAND_MAX + 1.0);

        distance_squared = local_x * local_x + local_y * local_y;
        if (distance_squared <= 1) {
            local_number_in_circle++;
        }
    }
    pthread_mutex_lock(&mutex);
    number_in_circle += local_number_in_circle;
    pthread_mutex_unlock(&mutex);
    return NULL;

}

void Usage(char *prog_name) {
    fprintf(stderr, "usage: %s No argument required to run this program\n", prog_name);
    exit(0);
}
```

Output:

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 4/4.1$ gcc -g -Wall -o 4.1 4.1.c -pthread
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 4/4.1$ ./4.1
pi_estimate -> 3.141520, elapsed_time -> 0.01706, thread_count -> 1, number_of_tosses -> 100000
pi_estimate -> 3.141664, elapsed_time -> 0.04680, thread_count -> 1, number_of_tosses -> 1000000
pi_estimate -> 3.141130, elapsed_time -> 0.37192, thread_count -> 1, number_of_tosses -> 10000000
pi_estimate -> 3.146360, elapsed_time -> 0.02027, thread_count -> 2, number_of_tosses -> 100000
pi_estimate -> 3.143948, elapsed_time -> 0.68554, thread_count -> 2, number_of_tosses -> 1000000
pi_estimate -> 3.141491, elapsed_time -> 6.06619, thread_count -> 2, number_of_tosses -> 10000000
pi_estimate -> 3.137200, elapsed_time -> 0.02058, thread_count -> 4, number_of_tosses -> 100000
pi_estimate -> 3.140196, elapsed_time -> 0.16024, thread_count -> 4, number_of_tosses -> 1000000
pi_estimate -> 3.141389, elapsed_time -> 1.59365, thread_count -> 4, number_of_tosses -> 10000000
pi_estimate -> 3.144440, elapsed_time -> 0.02567, thread_count -> 8, number_of_tosses -> 100000
pi_estimate -> 3.139596, elapsed_time -> 0.18654, thread_count -> 8, number_of_tosses -> 1000000
pi_estimate -> 3.142652, elapsed_time -> 2.86301, thread_count -> 8, number_of_tosses -> 10000000
```

Figure 1: Simulation Result

Table 1: Table of calculation

Result	Elapsed time	Number of tosses	Correctness	Speed-UP	Efficiency	Number of threads
3,141520	1,71E-01	100000	99,9977%	1,0	100,0%	1
3,141664	4,68E-02	1000000	99,9977%	1,0	100,0%	1
3,141130	3,72E-01	10000000	99,9853%	1,0	100,0%	1
3,146360	2,03E-02	100000	99,8482%	8,4	420,8%	2
3,143948	6,86E-01	1000000	99,9250%	14,6	732,4%	2
3,141491	6,07E+00	10000000	99,9968%	16,3	815,5%	2
3,137200	2,06E-02	100000	99,8602%	0,1	3,0%	4
3,140196	1,60E-01	1000000	99,9556%	3,4	85,6%	4
3,141389	1,59E+00	10000000	99,9935%	4,3	107,1%	4
3,144440	2,57E-02	100000	99,9093%	0,2	1,9%	8
3,139596	1,87E-01	1000000	99,9365%	4,0	49,8%	8
3,142652	2,86E+00	10000000	99,9663%	16,8	209,8%	8

Conclusion:

- The correct prediction of pi improves with an increase in the number of tosses
- A strange tendency is observed, with two threads speedups are 8,14,16 and efficiency is in between 420-815%.
- That result decreases with the increase of threads, for 4 and 8 number of threads.
- This proves shows that a number of threads need to match the number of cores, to have good performance and increasing thread numbers beyond that don’t give much efficiency.

TASK 2

Code:

```
/*
 * Compile:  gcc -g -Wall -o 4.2 4.2.c -lm -lpthread
 * No -> Arguments required
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include "timer.h"

const int MAX_THREADS = 1024;

long thread_count;
long long n;
double sum;
pthread_mutex_t mutex;

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 */
   if (argc != 1) {
      printf("No argument required!!!");
   }


   pthread_mutex_init(&mutex, NULL);
   n = 1000000000;
   for (long thread_number = 4; thread_number < 1024; thread_number *= 2) {
      sum = 0.0;
      thread_count = thread_number;
      thread_handles = (pthread_t *) malloc(thread_count * sizeof(pthread_t));
      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("[MultiThread]The elapsed time -> %e\t Estimated_Pi-> %.15f\t n -> %lld\t thread_count ->
%ld\n", elapsed,
             sum, n, thread_count);
      GET_TIME(start);
```

```
      sum = Serial_pi(n);
      GET_TIME(finish);
      elapsed = finish - start;
      printf("[SinglThread]The elapsed time -> %e\t Estimated_Pi-> %.15f\t n -> %lld\t thread_count ->
%ld\n", elapsed,
            sum, n, thread_count);
      printf("\n");
   }
   printf("[Original]-> %.15lf\n", 4.0 * atan(1.0));

   pthread_mutex_destroy(&mutex);
   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);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   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);
}
```

Output:

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 4/4.2$ ./4.b
[MultiThread]The elapsed time -> 1.166674e+00    Estimated_Pi-> 3.141592652589210        n -> 1000000000         thread_count -> 4
[SinglThread]The elapsed time -> 3.034009e+00    Estimated_Pi-> 3.141592652588050        n -> 1000000000         thread_count -> 4

[MultiThread]The elapsed time -> 1.431960e+00    Estimated_Pi-> 3.141592652589324        n -> 1000000000         thread_count -> 8
[SinglThread]The elapsed time -> 3.179619e+00    Estimated_Pi-> 3.141592652588050        n -> 1000000000         thread_count -> 8

[MultiThread]The elapsed time -> 1.151873e+00    Estimated_Pi-> 3.141592652590205        n -> 1000000000         thread_count -> 16
[SinglThread]The elapsed time -> 3.357267e+00    Estimated_Pi-> 3.141592652588050        n -> 1000000000         thread_count -> 16

[MultiThread]The elapsed time -> 1.141358e+00    Estimated_Pi-> 3.141592652590107        n -> 1000000000         thread_count -> 32
[SinglThread]The elapsed time -> 3.226326e+00    Estimated_Pi-> 3.141592652588050        n -> 1000000000         thread_count -> 32

[MultiThread]The elapsed time -> 1.417788e+00    Estimated_Pi-> 3.141592652589864        n -> 1000000000         thread_count -> 64
[SinglThread]The elapsed time -> 2.994104e+00    Estimated_Pi-> 3.141592652588050        n -> 1000000000         thread_count -> 64

[MultiThread]The elapsed time -> 1.223544e+00    Estimated_Pi-> 3.141592652589726        n -> 1000000000         thread_count -> 128
[SinglThread]The elapsed time -> 3.495514e+00    Estimated_Pi-> 3.141592652588050        n -> 1000000000         thread_count -> 128

[MultiThread]The elapsed time -> 1.146172e+00    Estimated_Pi-> 3.141592652589577        n -> 1000000000         thread_count -> 256
[SinglThread]The elapsed time -> 3.191379e+00    Estimated_Pi-> 3.141592652588050        n -> 1000000000         thread_count -> 256

[MultiThread]The elapsed time -> 1.424902e+00    Estimated_Pi-> 3.141592652589717        n -> 1000000000         thread_count -> 512
[SinglThread]The elapsed time -> 3.027768e+00    Estimated_Pi-> 3.141592652588050        n -> 1000000000         thread_count -> 512

[Original]-> 3.141592653589793
```

Figure 2: The simulation result

Table 2: Result Calculation

The original PI	Predicted PI values	Thread Number	Error	N
3,14159265358979	3,141592652589210	4	0,000000001000580	1000000000
	3,141592652589320	8	0,000000001000470	1000000000
	3,141592652590200	16	0,000000000999590	1000000000
	3,141592652590100	32	0,000000000999690	1000000000
	3,141592652589860	64	0,000000000999930	1000000000
	3,141592652589720	128	0,000000001000070	1000000000
	3,141592652589570	256	0,000000001000220	1000000000
	3,141592652589710	512	0,000000001000080	1000000000

Conclusion:

- Mutex scheduling is dependent on OS, and it is not guaranteed that mutex in the queue will get access to the shared variable and the program may be terminated.
- From, the result it is seen up to 64 threads prediction of PI is improved by increasing the number of threads, however, this tendency is lost when the number of threads is 128 and more that's proofs that not all thread will get access to the shared variable.

TASK 3

Code:

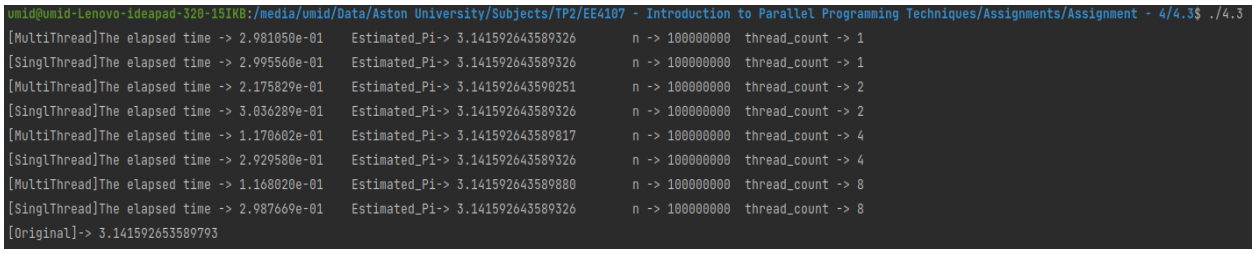
```
/*
 *
 * Compile:  gcc -g -Wall -o 4.3 4.3.c -pthread
 * No -> Arguments are required!!!!
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include "timer.h"
#include "semaphore.h"

#define Semaphore //define one of this to change the program
/*                                  1. Original_P_Mutex
 *                                  2. Original_for_loop_Mutex
 *                                  3. Original_Busy_Waiting
 *                                  4. Semaphore
 */
const int MAX_THREADS = 1024;

long thread_count;
long long n;
double sum;
int flag;
pthread_mutex_t mutex;
sem_t sempahore;

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);
   if(argc != 1){
      printf("No arguments are needed!!!!!");
      return 0;
   }
   pthread_mutex_init(&mutex, NULL);
   sem_init(&sempahore, 1, 1);

   for (long number_threads = 1; number_threads < 16; number_threads *= 2) {

         n = 100000000;
         thread_count = number_threads;
         thread_handles = (pthread_t *) malloc(thread_count * sizeof(pthread_t));
         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("[MultiThread]The elapsed time -> %e\t Estimated_Pi-> %.15f\t n -> %lld\t thread_count ->
%ld\n", elapsed,
               sum, n, thread_count);
         GET_TIME(start);
         sum = Serial_pi(n);
         GET_TIME(finish);
         elapsed = finish - start;
         printf("[SinglThread]The elapsed time -> %e\t Estimated_Pi-> %.15f\t n -> %lld\t thread_count ->
%ld\n", elapsed,
               sum, n, thread_count);

   }
   printf("[Original]-> %.15lf\n", 4.0 * atan(1.0));
   pthread_mutex_destroy(&mutex);
   sem_destroy(&sempahore);
   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;
#  ifdef Original_P_Mutex
   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
     my_sum += factor/(2*i+1);
  }
  pthread_mutex_lock(&mutex);
  sum += my_sum;
  pthread_mutex_unlock(&mutex);
#  endif

#  ifdef Original_for_loop_Mutex
   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      pthread_mutex_lock(&mutex);
      sum += factor/(2*i+1);
      pthread_mutex_unlock(&mutex);
  }
#  endif
#  ifdef Original_Busy_Waiting // ask if it is after the loop or inside the loop
   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
     while (flag != my_rank);
     sum += factor/(2*i+1);
     flag = (flag+1) % thread_count;
```

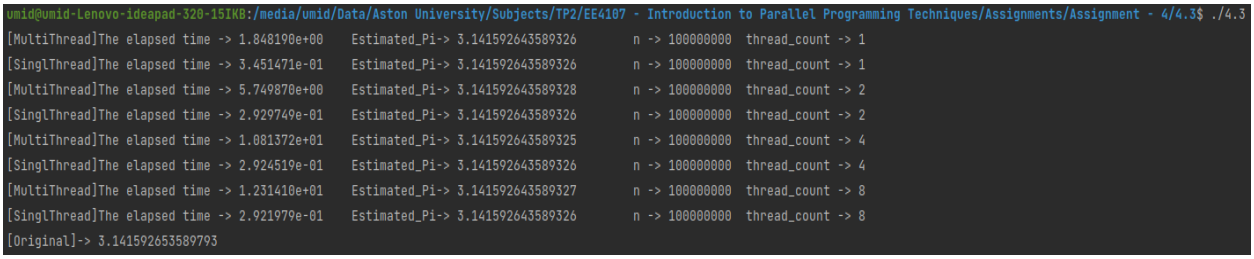
```
  }
#  endif

#  ifdef Semaphore // ask if it is after the loop or inside the loop
   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      sem_wait(&sempahore);
      sum += factor / (2 * i + 1);
      sem_post(&sempahore);
   }
#  endif

   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 */
```

Result:

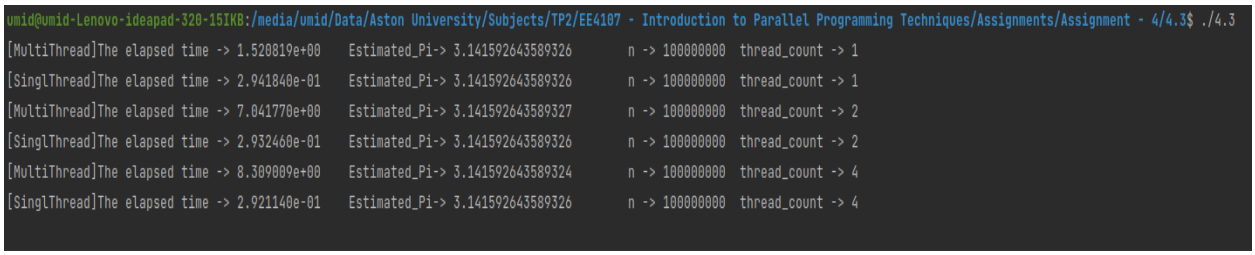
1. Original mutex version



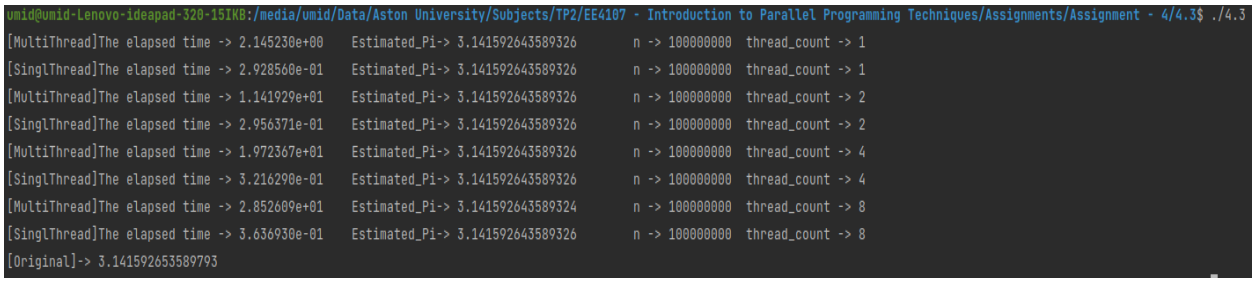
2. Modified version critical section in for loop



3. Busy waiting loop version.

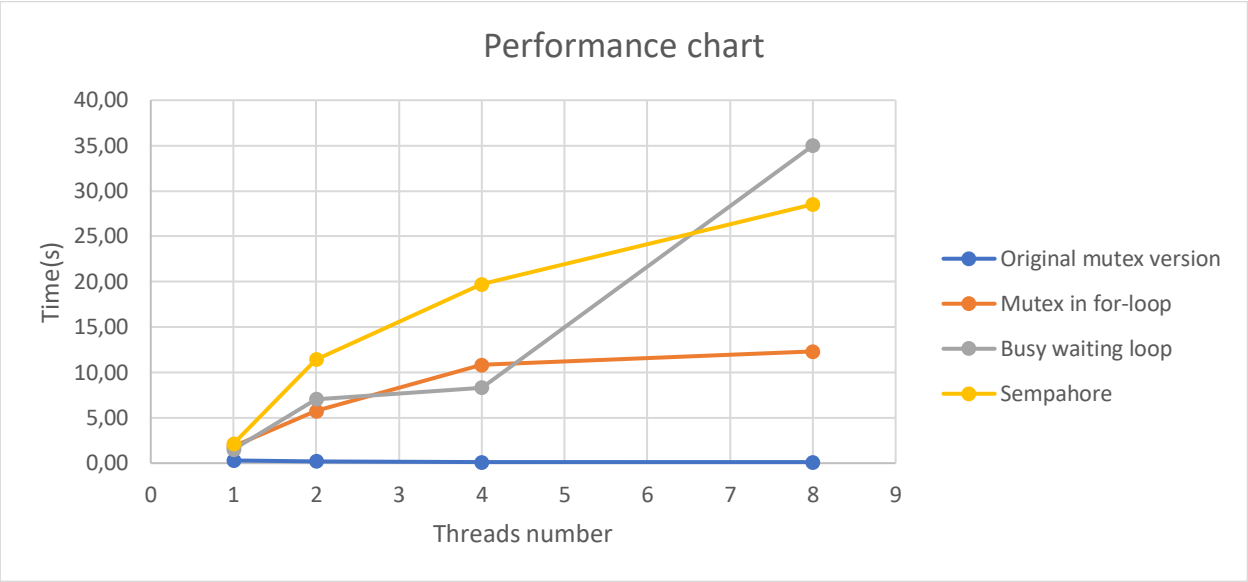


4. Semaphore



Thread number	Elpased time of original mutex version	Elpased time of mutex in for loop	Busy waiting loop	Sempahore
1	0,30	1,85	1,52	2,15
2	0,22	5,75	7,04	11,42
4	0,12	10,81	8,31	19,72
8	0,12	12,31	35,00	28,53

Table 3: Result table



Conclusion:

a) Mutex in the for-loop section has better performance than the busy waiting-loop version, busy waiting loop had infinity waiting time in my simulation with a large number of n and threads.

b) The Semaphore version is still worth than mutex version, mutex version has shown the best performance among all type of modification, the graph shows it all.