CSE435
HW2

Burak Kocausta
1901042605

Content

1- Installation, Compilation, Run

Installation of MPI:
```
$ sudo apt install mpich
```

Compilation:
```
$ make
```

Run Tests:
```
$ chmod +x ./tests.sh
$ ./tests.sh
```

Run Single Thread:
```
$./mc_single
```
or
```
$./mc_single <number_of_tosses>
```

Run MPI:

```
$ mpiexec -n <number_of_processes> ./mc_mpi
```

or

```
$ mpiexec -n <number_of_processes> ./mc_mpi <number_of_tosses>
```

Run Multiple Thread:

```
$./mc_thread
```

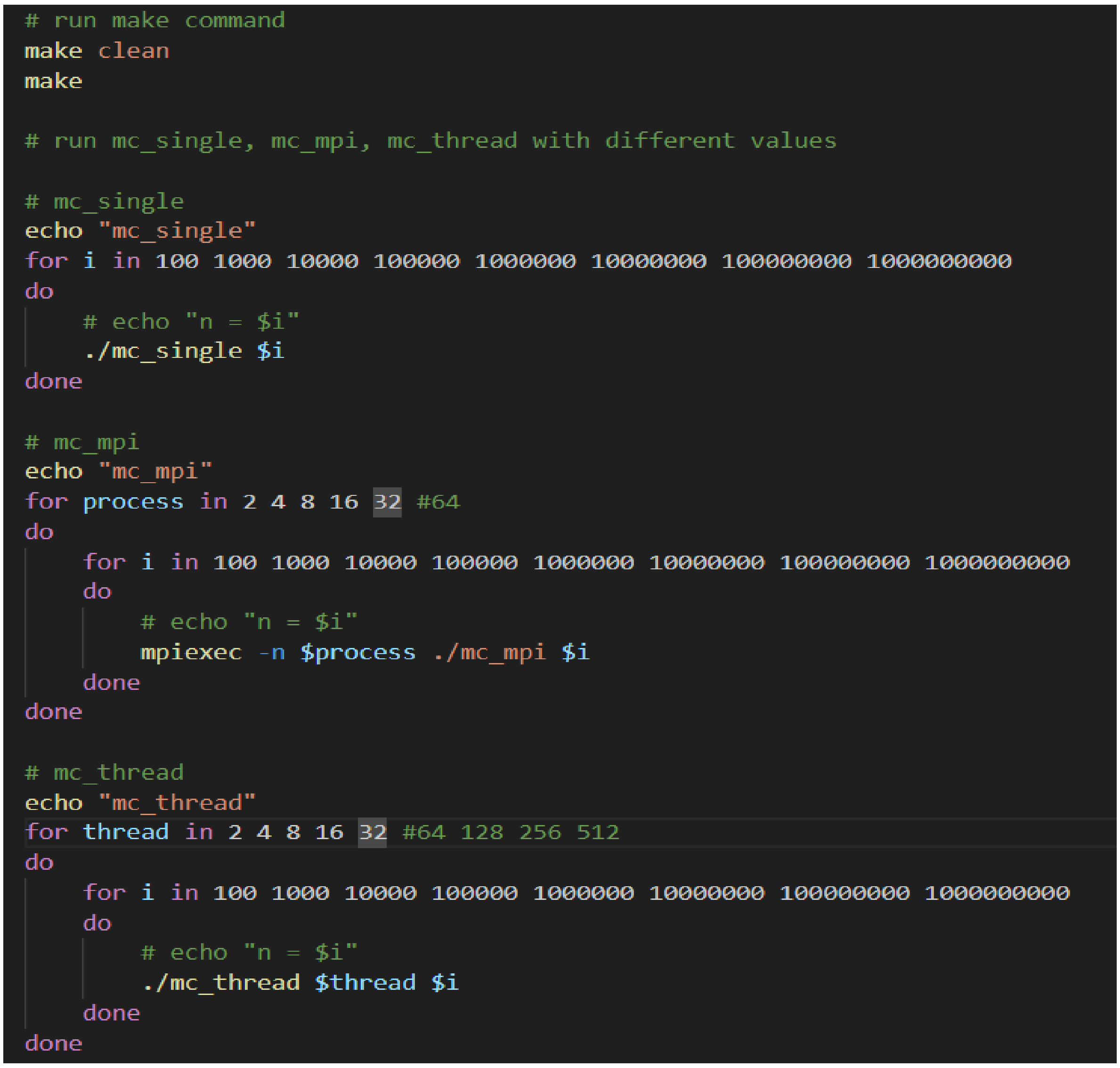
or

```
$./mc_thread <number_of_threads>
```

or

```
$./mc_thread <number_of_threads> <number_of_tosses>
```

Tests.sh:



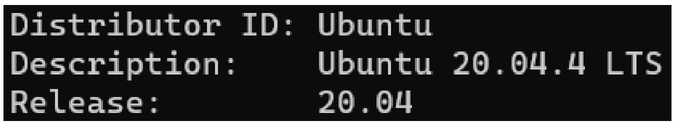
```
# run make command
make clean
make

# run mc_single, mc_mpi, mc_thread with different values

# mc_single
echo "mc_single"
for i in 100 1000 10000 100000 1000000 10000000 100000000 1000000000
do
    # echo "n = $i"
    ./mc_single $i
done

# mc_mpi
echo "mc_mpi"
for process in 2 4 8 16 32 #64
do
    for i in 100 1000 10000 100000 1000000 10000000 100000000 1000000000
    do
        # echo "n = $i"
        mpiexec -n $process ./mc_mpi $i
    done
done

# mc_thread
echo "mc_thread"
for thread in 2 4 8 16 32 #64 128 256 512
do
    for i in 100 1000 10000 100000 1000000 10000000 100000000 1000000000
    do
        # echo "n = $i"
        ./mc_thread $thread $i
    done
done
```

- This file makes compilation and runs the tests. Each run creates csv file (separated with semicolon).

2- Information about Computing Environment (Q1)

- Operating System:

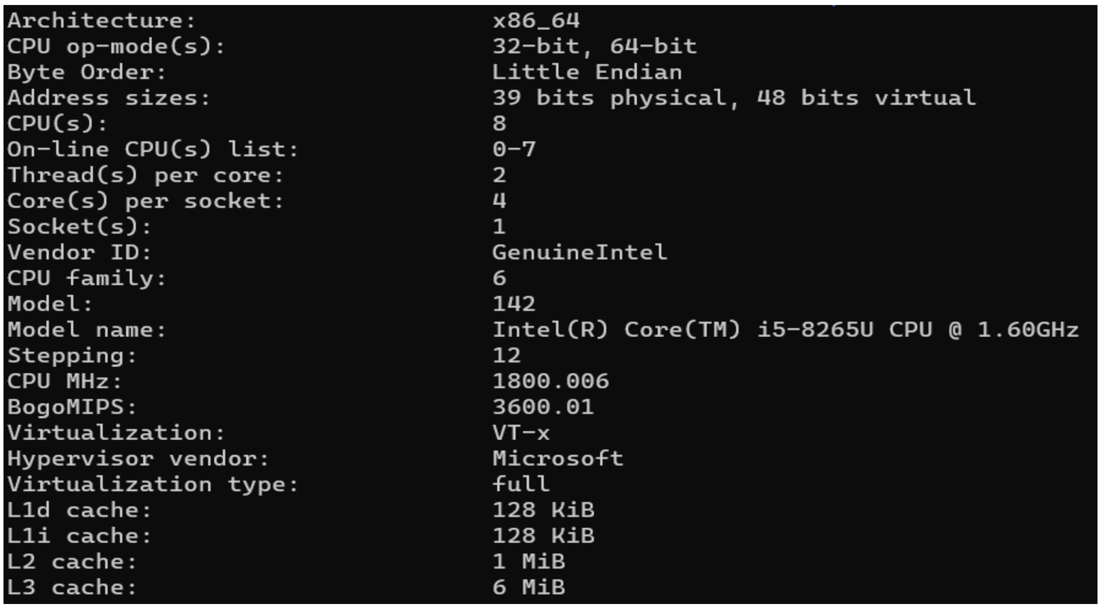
WSL2 (Windows Subsystem for Linux) Environment is used.



```
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.4 LTS
Release:        20.04
```

Ubuntu version is 20.04

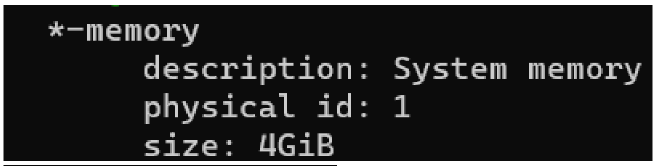
- CPU Information:



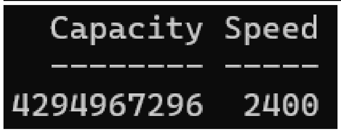
```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           142
Model name:                      Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
Stepping:                        12
CPU MHz:                         1800.006
BogoMIPS:                        3600.01
Virtualization:                  VT-x
Hypervisor vendor:               Microsoft
Virtualization type:             full
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        1 MiB
L3 cache:                        6 MiB
```

Core number is 8 and 2 threads per core.

- Memory Information

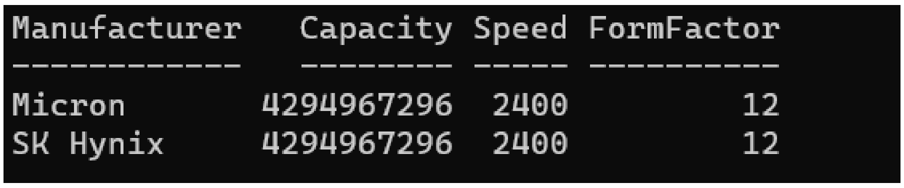


```
  *-memory
       description: System memory
       physical id: 1
       size: 4GiB
```



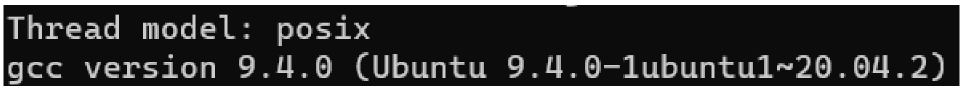
```
  Capacity Speed
  -------- -----
4294967296  2400
```

2400 Ghz, 4GB



```
Manufacturer   Capacity Speed FormFactor
------------   -------- ----- ----------
Micron       4294967296  2400         12
SK Hynix     4294967296  2400         12
```

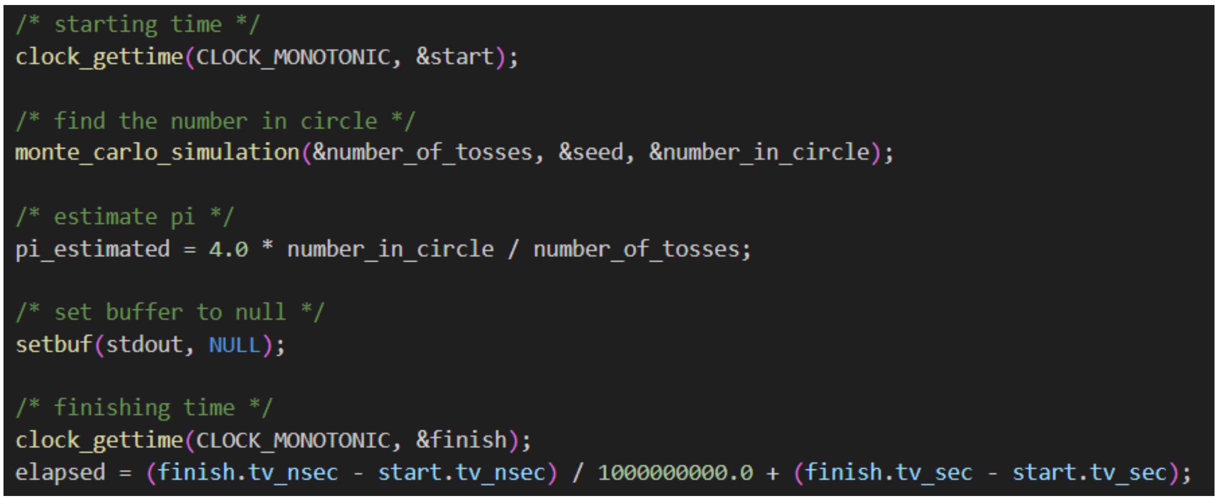
- Compiler Information:



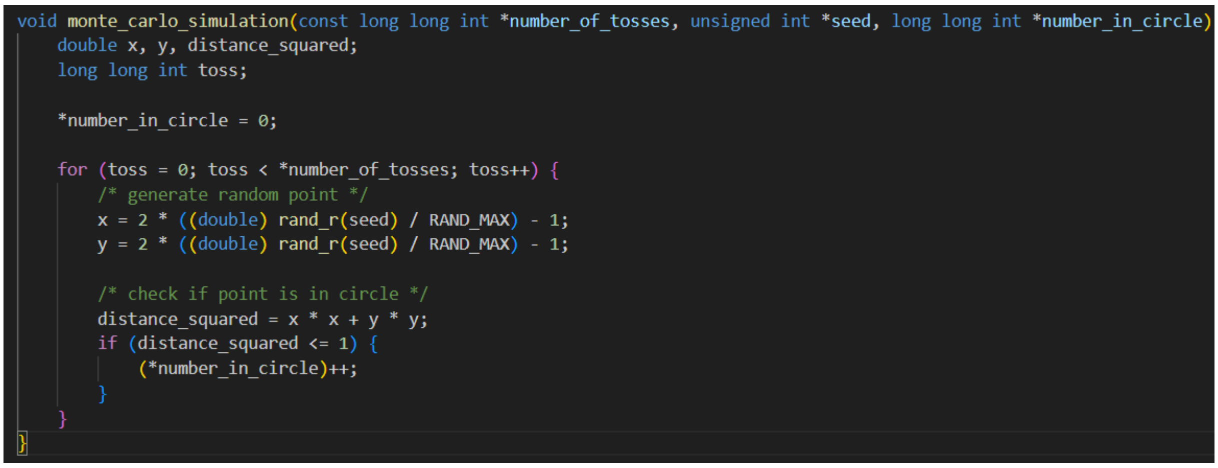
```
Thread model: posix
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)
```

3- Single Threaded Implementation Monte Carlo Method (Q2)

Implementation:



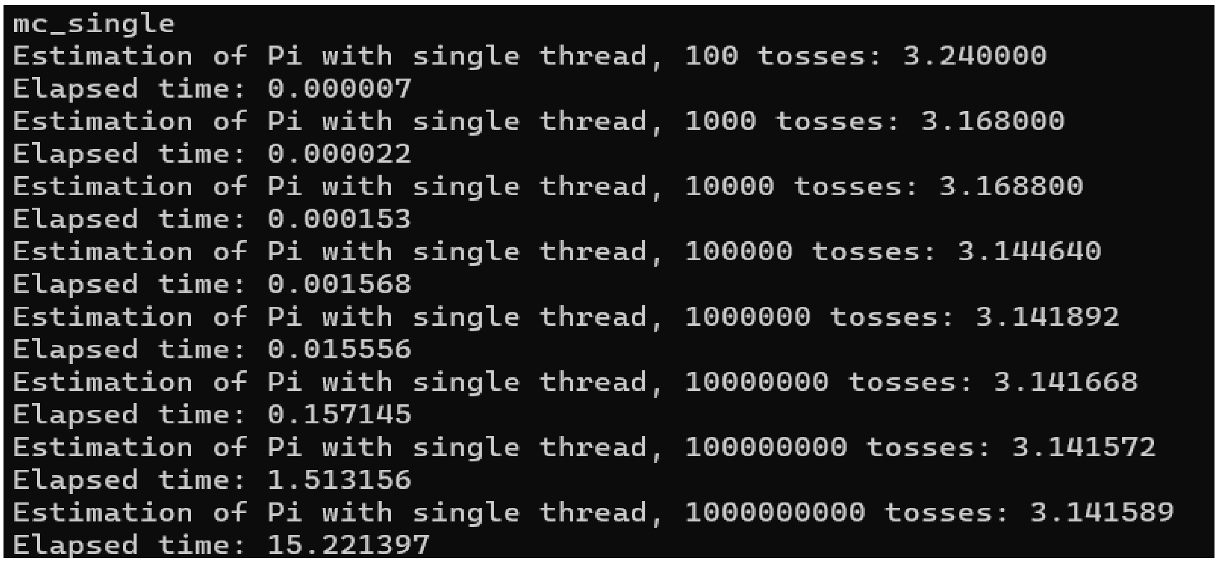
```c
/* starting time */
clock_gettime(CLOCK_MONOTONIC, &start);

/* find the number in circle */
monte_carlo_simulation(&number_of_tosses, &seed, &number_in_circle);

/* estimate pi */
pi_estimated = 4.0 * number_in_circle / number_of_tosses;

/* set buffer to null */
setbuf(stdout, NULL);

/* finishing time */
clock_gettime(CLOCK_MONOTONIC, &finish);
elapsed = (finish.tv_nsec - start.tv_nsec) / 1000000000.0 + (finish.tv_sec - start.tv_sec);
```



```c
void monte_carlo_simulation(const long long int *number_of_tosses, unsigned int *seed, long long int *number_in_circle)
    double x, y, distance_squared;
    long long int toss;

    *number_in_circle = 0;

    for (toss = 0; toss < *number_of_tosses; toss++) {
        /* generate random point */
        x = 2 * ((double) rand_r(seed) / RAND_MAX) - 1;
        y = 2 * ((double) rand_r(seed) / RAND_MAX) - 1;

        /* check if point is in circle */
        distance_squared = x * x + y * y;
        if (distance_squared <= 1) {
            (*number_in_circle)++;
        }
    }
}
```

- Direct implementation of the algorithm.

Output (tests.sh):

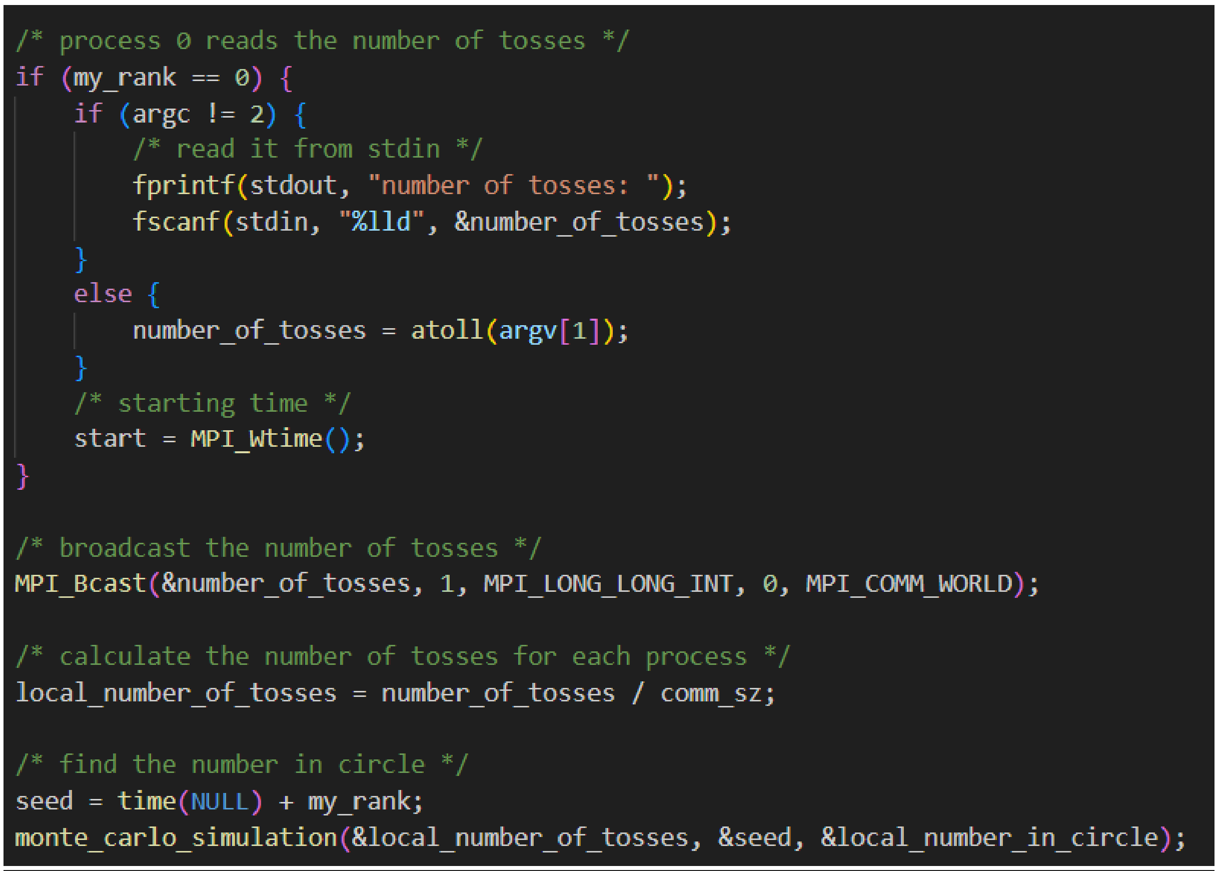


```
mc_single
Estimation of Pi with single thread, 100 tosses: 3.240000
Elapsed time: 0.000007
Estimation of Pi with single thread, 1000 tosses: 3.168000
Elapsed time: 0.000022
Estimation of Pi with single thread, 10000 tosses: 3.168800
Elapsed time: 0.000153
Estimation of Pi with single thread, 100000 tosses: 3.144640
Elapsed time: 0.001568
Estimation of Pi with single thread, 1000000 tosses: 3.141892
Elapsed time: 0.015556
Estimation of Pi with single thread, 10000000 tosses: 3.141668
Elapsed time: 0.157145
Estimation of Pi with single thread, 100000000 tosses: 3.141572
Elapsed time: 1.513156
Estimation of Pi with single thread, 1000000000 tosses: 3.141589
Elapsed time: 15.221397
```

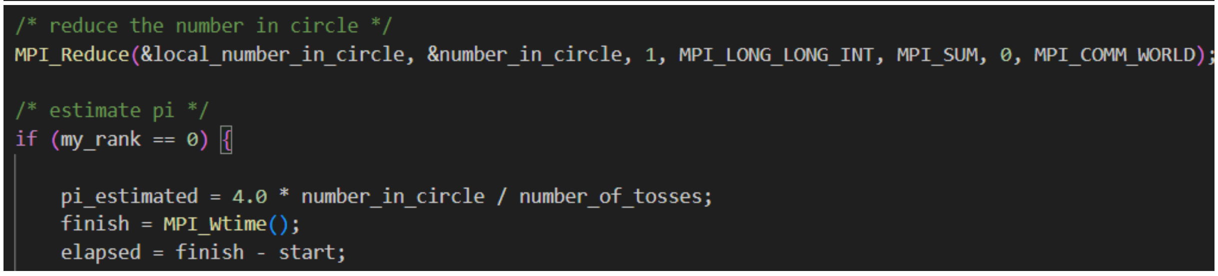
- Accurate estimation with the increase of the tosses.

4- Monte Carlo Method Using MPI (Q3)

Implementation:



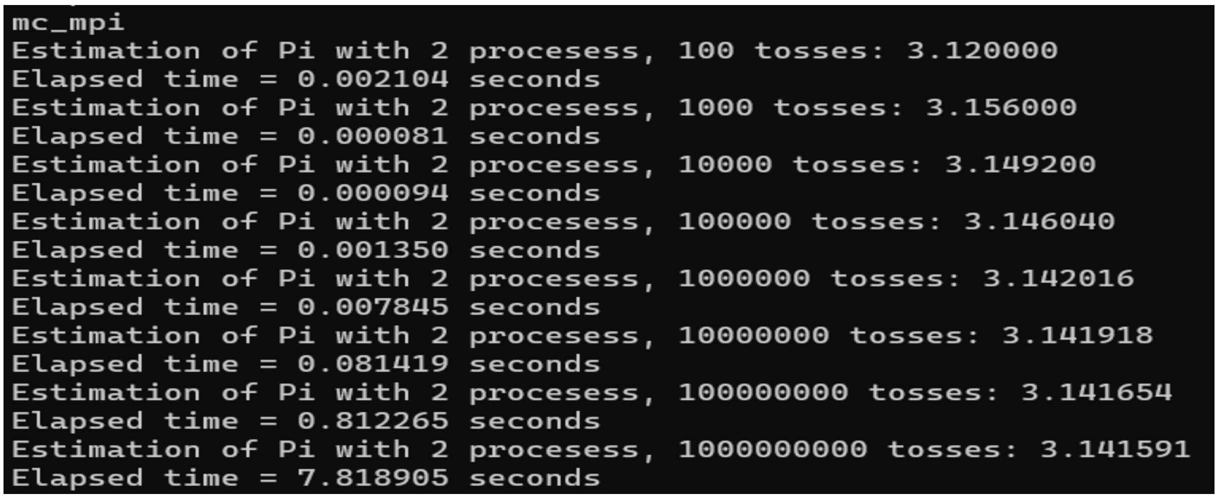
```c
/* process 0 reads the number of tosses */
if (my_rank == 0) {
    if (argc != 2) {
        /* read it from stdin */
        fprintf(stdout, "number of tosses: ");
        fscanf(stdin, "%lld", &number_of_tosses);
    }
    else {
        number_of_tosses = atoll(argv[1]);
    }
    /* starting time */
    start = MPI_Wtime();
}

/* broadcast the number of tosses */
MPI_Bcast(&number_of_tosses, 1, MPI_LONG_LONG_INT, 0, MPI_COMM_WORLD);

/* calculate the number of tosses for each process */
local_number_of_tosses = number_of_tosses / comm_sz;

/* find the number in circle */
seed = time(NULL) + my_rank;
monte_carlo_simulation(&local_number_of_tosses, &seed, &local_number_in_circle);
```



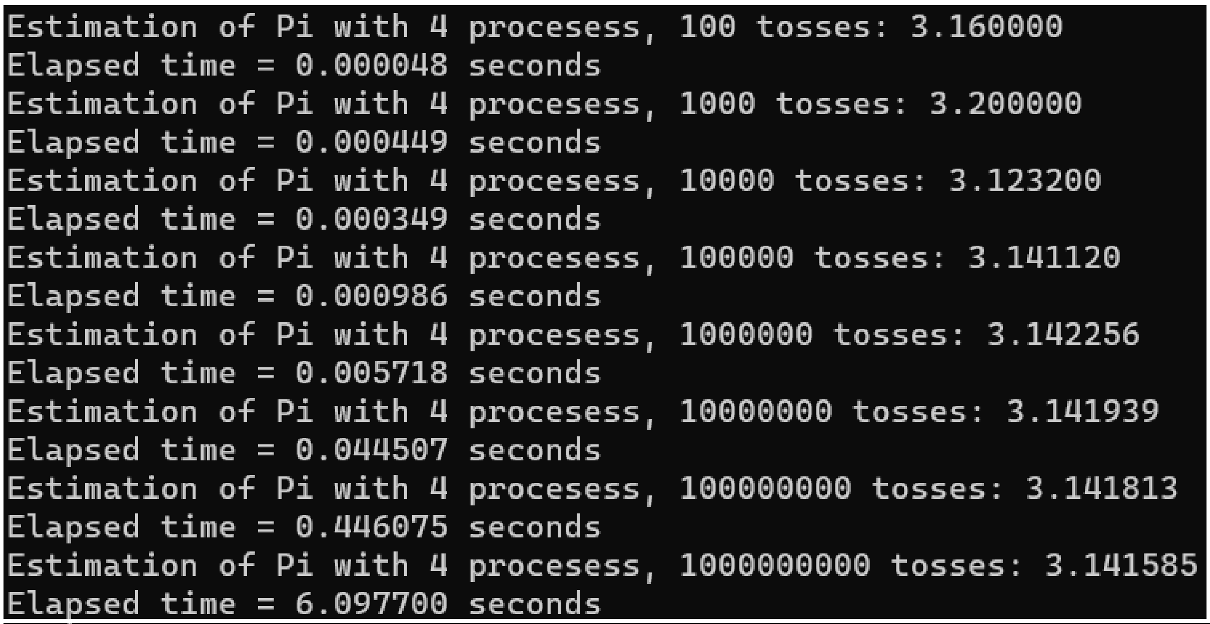
```c
/* reduce the number in circle */
MPI_Reduce(&local_number_in_circle, &number_in_circle, 1, MPI_LONG_LONG_INT, MPI_SUM, 0, MPI_COMM_WORLD);

/* estimate pi */
if (my_rank == 0) {

    pi_estimated = 4.0 * number_in_circle / number_of_tosses;
    finish = MPI_Wtime();
    elapsed = finish - start;
```

Total number is calculated with reduce function. Time is measured with MPI_Wtime.

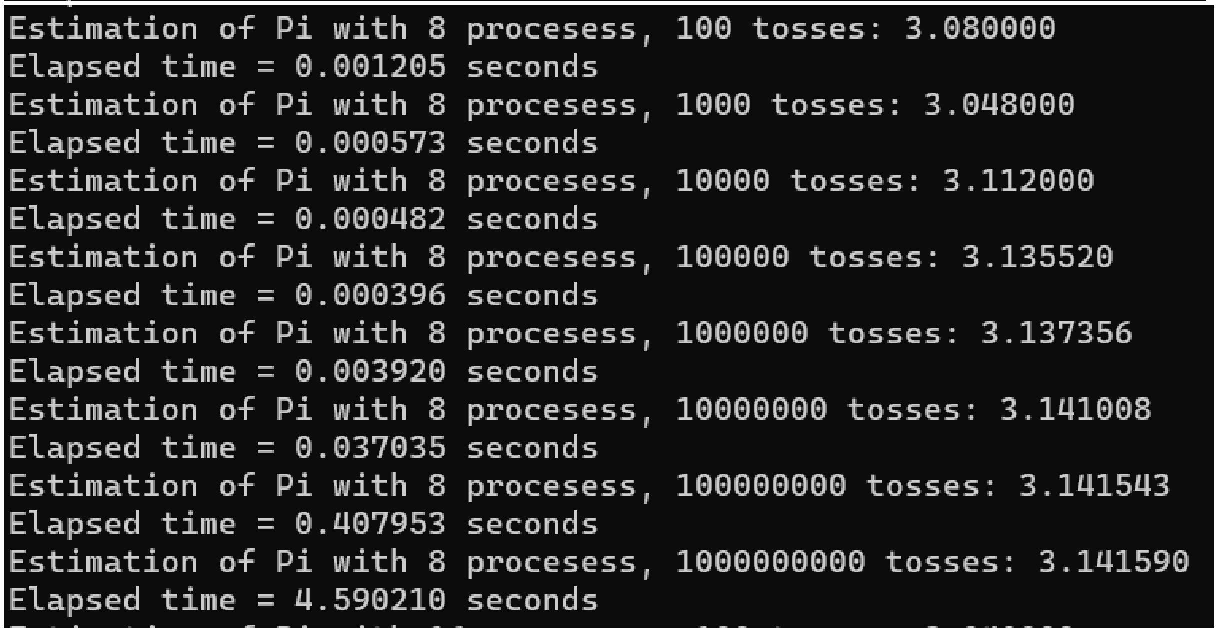
Output (tests.sh):



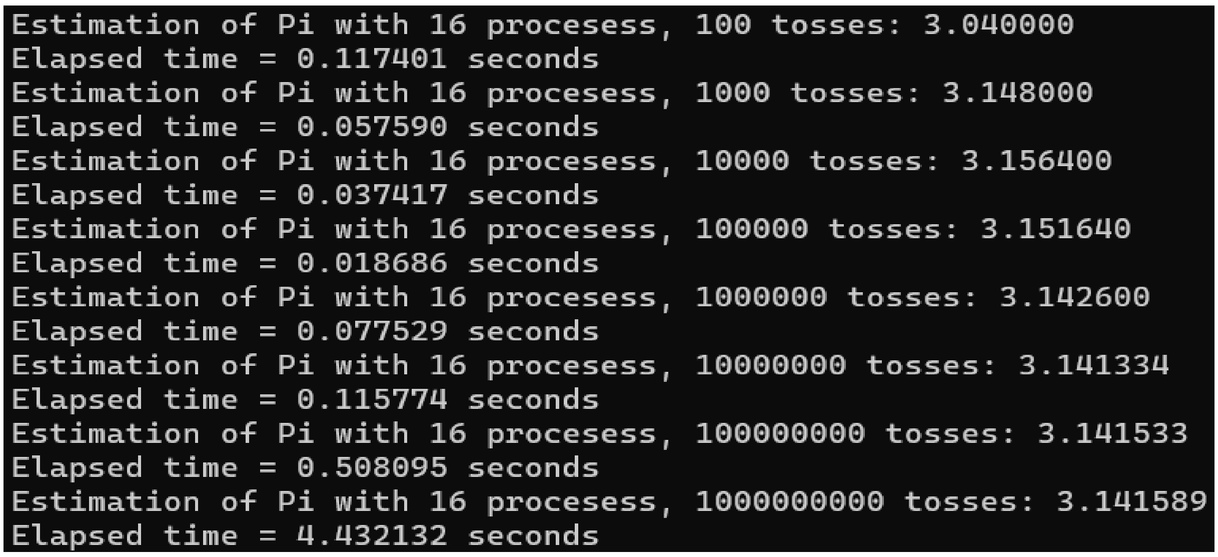
```
mc_mpi
Estimation of Pi with 2 processess, 100 tosses: 3.120000
Elapsed time = 0.002104 seconds
Estimation of Pi with 2 processess, 1000 tosses: 3.156000
Elapsed time = 0.000081 seconds
Estimation of Pi with 2 processess, 10000 tosses: 3.149200
Elapsed time = 0.000094 seconds
Estimation of Pi with 2 processess, 100000 tosses: 3.146040
Elapsed time = 0.001350 seconds
Estimation of Pi with 2 processess, 1000000 tosses: 3.142016
Elapsed time = 0.007845 seconds
Estimation of Pi with 2 processess, 10000000 tosses: 3.141918
Elapsed time = 0.081419 seconds
Estimation of Pi with 2 processess, 100000000 tosses: 3.141654
Elapsed time = 0.812265 seconds
Estimation of Pi with 2 processess, 1000000000 tosses: 3.141591
Elapsed time = 7.818905 seconds
```



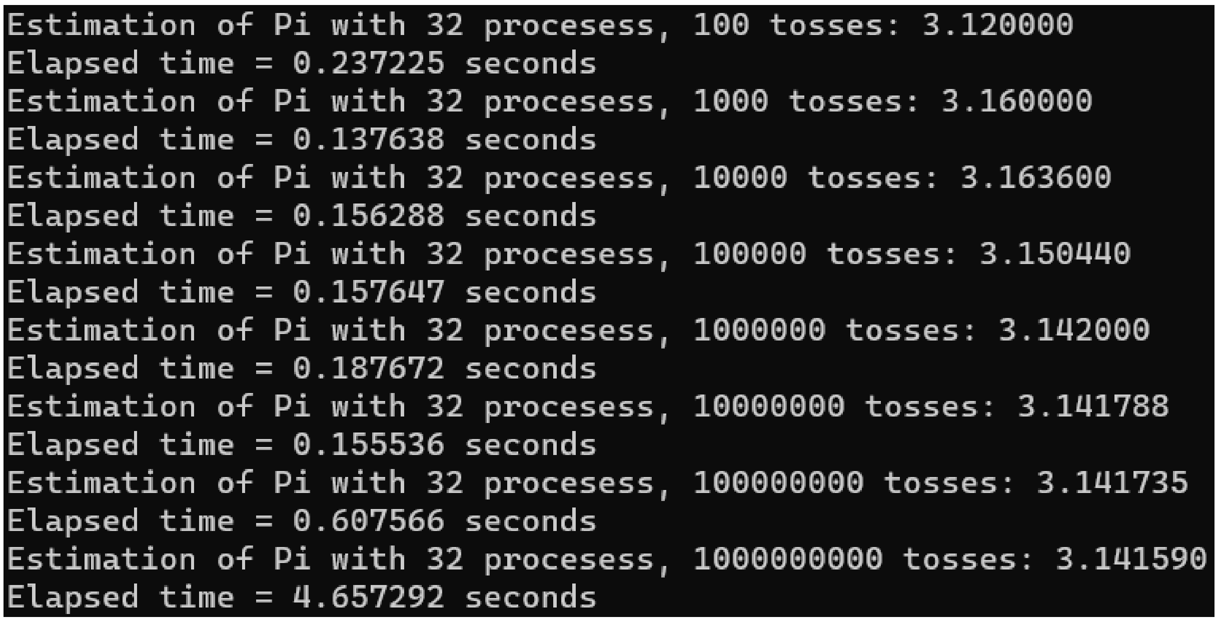
```
Estimation of Pi with 4 procesess, 100 tosses: 3.160000
Elapsed time = 0.000048 seconds
Estimation of Pi with 4 procesess, 1000 tosses: 3.200000
Elapsed time = 0.000449 seconds
Estimation of Pi with 4 procesess, 10000 tosses: 3.123200
Elapsed time = 0.000349 seconds
Estimation of Pi with 4 procesess, 100000 tosses: 3.141120
Elapsed time = 0.000986 seconds
Estimation of Pi with 4 procesess, 1000000 tosses: 3.142256
Elapsed time = 0.005718 seconds
Estimation of Pi with 4 procesess, 10000000 tosses: 3.141939
Elapsed time = 0.044507 seconds
Estimation of Pi with 4 procesess, 100000000 tosses: 3.141813
Elapsed time = 0.446075 seconds
Estimation of Pi with 4 procesess, 1000000000 tosses: 3.141585
Elapsed time = 6.097700 seconds
```



```
Estimation of Pi with 8 procesess, 100 tosses: 3.080000
Elapsed time = 0.001205 seconds
Estimation of Pi with 8 procesess, 1000 tosses: 3.048000
Elapsed time = 0.000573 seconds
Estimation of Pi with 8 procesess, 10000 tosses: 3.112000
Elapsed time = 0.000482 seconds
Estimation of Pi with 8 procesess, 100000 tosses: 3.135520
Elapsed time = 0.000396 seconds
Estimation of Pi with 8 procesess, 1000000 tosses: 3.137356
Elapsed time = 0.003920 seconds
Estimation of Pi with 8 procesess, 10000000 tosses: 3.141008
Elapsed time = 0.037035 seconds
Estimation of Pi with 8 procesess, 100000000 tosses: 3.141543
Elapsed time = 0.407953 seconds
Estimation of Pi with 8 procesess, 1000000000 tosses: 3.141590
Elapsed time = 4.590210 seconds
```



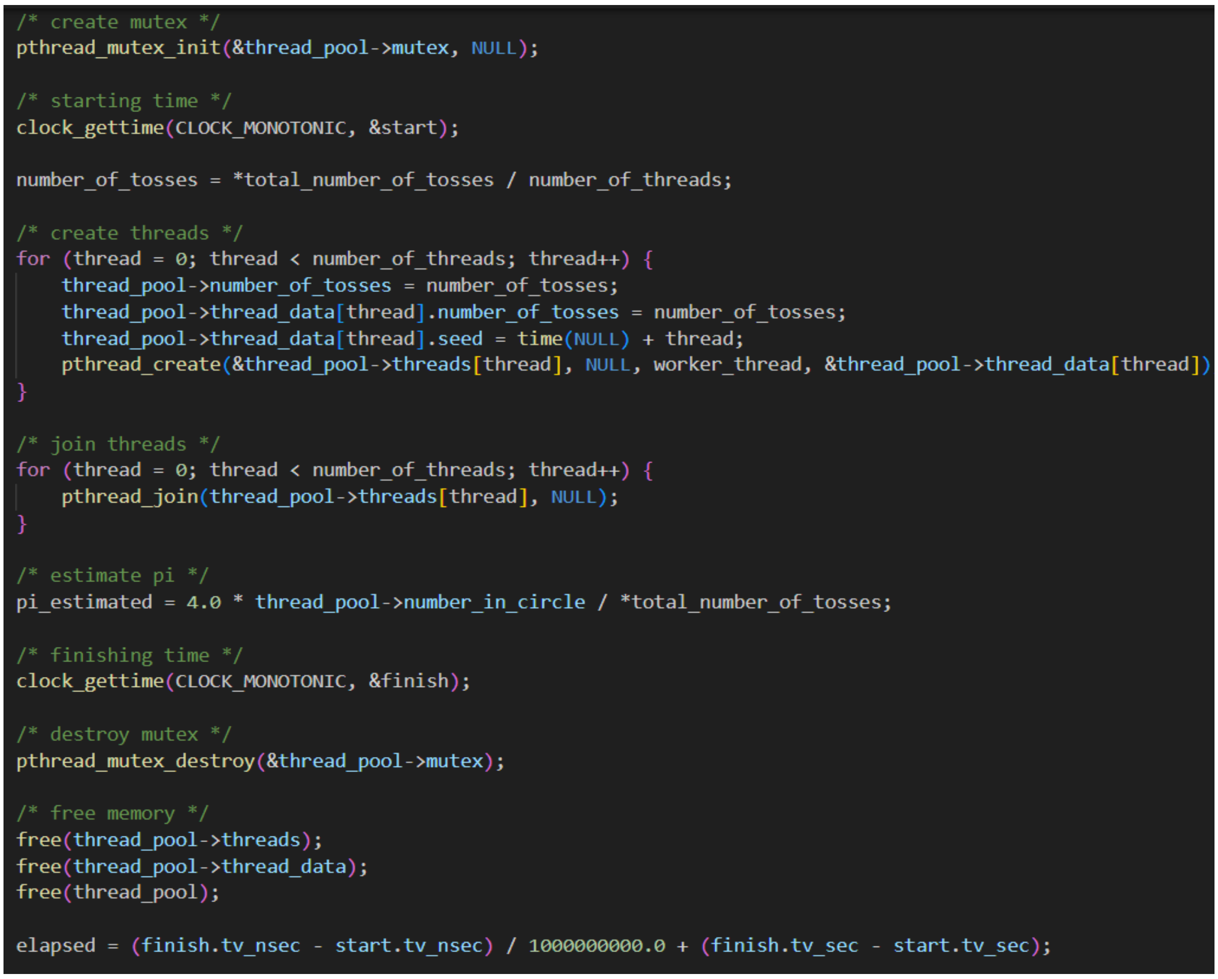
```
Estimation of Pi with 16 procesess, 100 tosses: 3.040000
Elapsed time = 0.117401 seconds
Estimation of Pi with 16 procesess, 1000 tosses: 3.148000
Elapsed time = 0.057590 seconds
Estimation of Pi with 16 procesess, 10000 tosses: 3.156400
Elapsed time = 0.037417 seconds
Estimation of Pi with 16 procesess, 100000 tosses: 3.151640
Elapsed time = 0.018686 seconds
Estimation of Pi with 16 procesess, 1000000 tosses: 3.142600
Elapsed time = 0.077529 seconds
Estimation of Pi with 16 procesess, 10000000 tosses: 3.141334
Elapsed time = 0.115774 seconds
Estimation of Pi with 16 procesess, 100000000 tosses: 3.141533
Elapsed time = 0.508095 seconds
Estimation of Pi with 16 procesess, 1000000000 tosses: 3.141589
Elapsed time = 4.432132 seconds
```



```
Estimation of Pi with 32 processess, 100 tosses: 3.120000
Elapsed time = 0.237225 seconds
Estimation of Pi with 32 processess, 1000 tosses: 3.160000
Elapsed time = 0.137638 seconds
Estimation of Pi with 32 processess, 10000 tosses: 3.163600
Elapsed time = 0.156288 seconds
Estimation of Pi with 32 processess, 100000 tosses: 3.150440
Elapsed time = 0.157647 seconds
Estimation of Pi with 32 processess, 1000000 tosses: 3.142000
Elapsed time = 0.187672 seconds
Estimation of Pi with 32 processess, 10000000 tosses: 3.141788
Elapsed time = 0.155536 seconds
Estimation of Pi with 32 processess, 100000000 tosses: 3.141735
Elapsed time = 0.607566 seconds
Estimation of Pi with 32 processess, 1000000000 tosses: 3.141590
Elapsed time = 4.657292 seconds
```

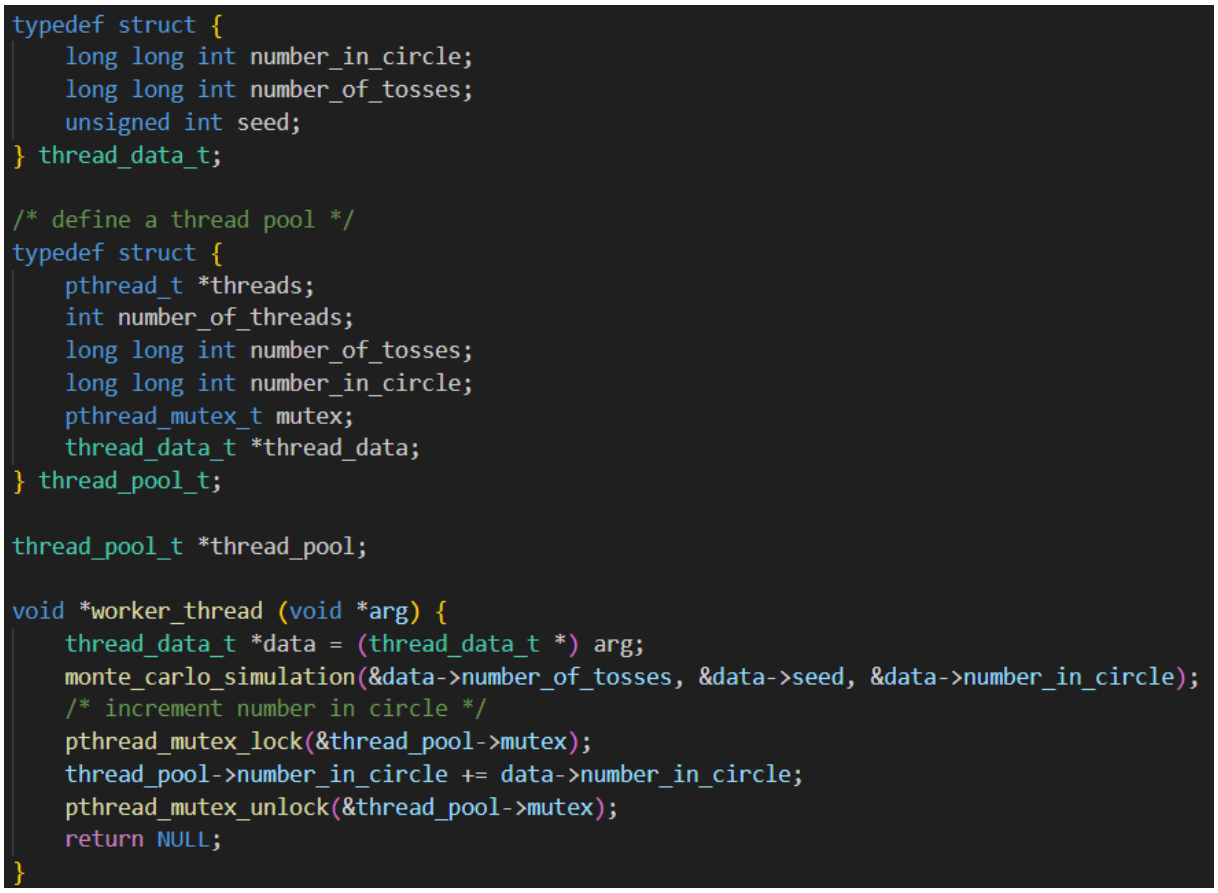
5- Monte Carlo Method Using Pthread (Q4)

Implementation:

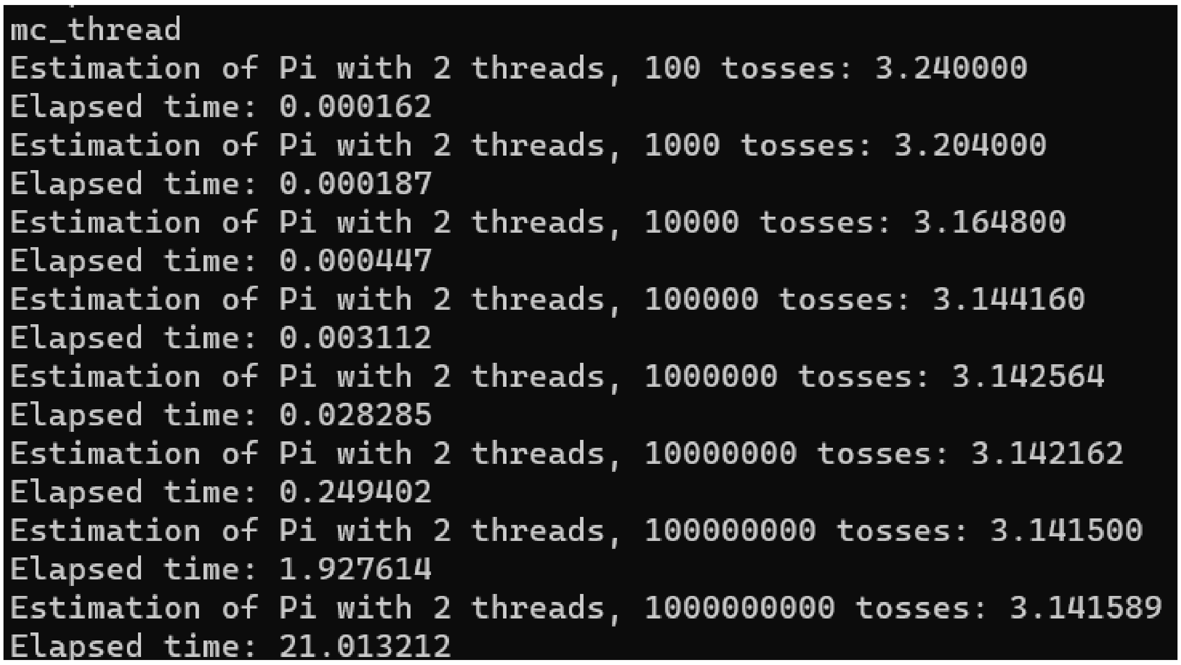


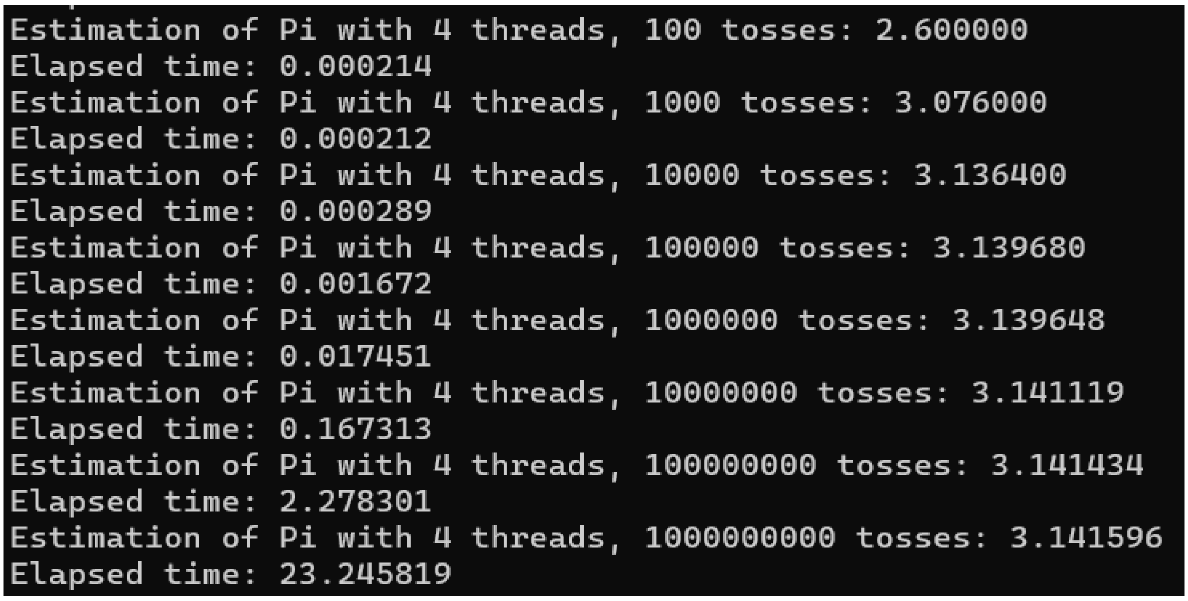
```c
/* create mutex */
pthread_mutex_init(&thread_pool->mutex, NULL);

/* starting time */
clock_gettime(CLOCK_MONOTONIC, &start);

number_of_tosses = *total_number_of_tosses / number_of_threads;

/* create threads */
for (thread = 0; thread < number_of_threads; thread++) {
    thread_pool->number_of_tosses = number_of_tosses;
    thread_pool->thread_data[thread].number_of_tosses = number_of_tosses;
    thread_pool->thread_data[thread].seed = time(NULL) + thread;
    pthread_create(&thread_pool->threads[thread], NULL, worker_thread, &thread_pool->thread_data[thread])
}

/* join threads */
for (thread = 0; thread < number_of_threads; thread++) {
    pthread_join(thread_pool->threads[thread], NULL);
}

/* estimate pi */
pi_estimated = 4.0 * thread_pool->number_in_circle / *total_number_of_tosses;

/* finishing time */
clock_gettime(CLOCK_MONOTONIC, &finish);

/* destroy mutex */
pthread_mutex_destroy(&thread_pool->mutex);

/* free memory */
free(thread_pool->threads);
free(thread_pool->thread_data);
free(thread_pool);

elapsed = (finish.tv_nsec - start.tv_nsec) / 1000000000.0 + (finish.tv_sec - start.tv_sec);
```

- Mutex and threads are created. Number in circle is global variable inside thread pool, it is changed by threads.

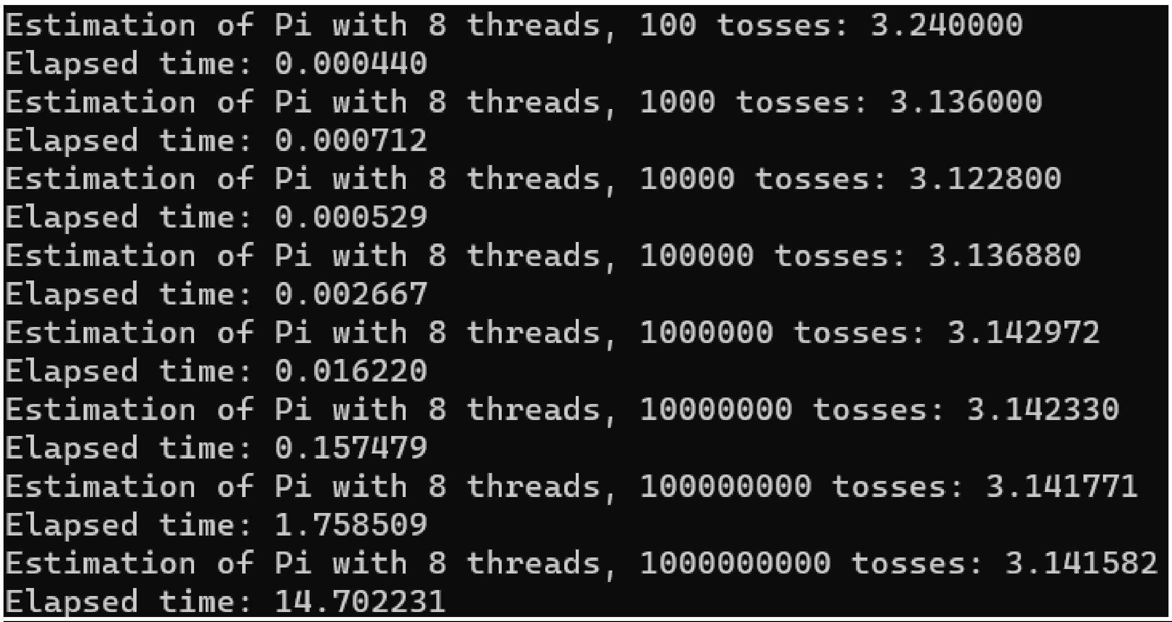

```c
typedef struct {
    long long int number_in_circle;
    long long int number_of_tosses;
    unsigned int seed;
} thread_data_t;

/* define a thread pool */
typedef struct {
    pthread_t *threads;
    int number_of_threads;
    long long int number_of_tosses;
    long long int number_in_circle;
    pthread_mutex_t mutex;
    thread_data_t *thread_data;
} thread_pool_t;

thread_pool_t *thread_pool;

void *worker_thread (void *arg) {
    thread_data_t *data = (thread_data_t *) arg;
    monte_carlo_simulation(&data->number_of_tosses, &data->seed, &data->number_in_circle);
    /* increment number in circle */
    pthread_mutex_lock(&thread_pool->mutex);
    thread_pool->number_in_circle += data->number_in_circle;
    pthread_mutex_unlock(&thread_pool->mutex);
    return NULL;
}
```
Thread pool, and worker thread are defined as this. I used mutex for calculating number in circle part in order to prevent data race.

Output (tests.sh):

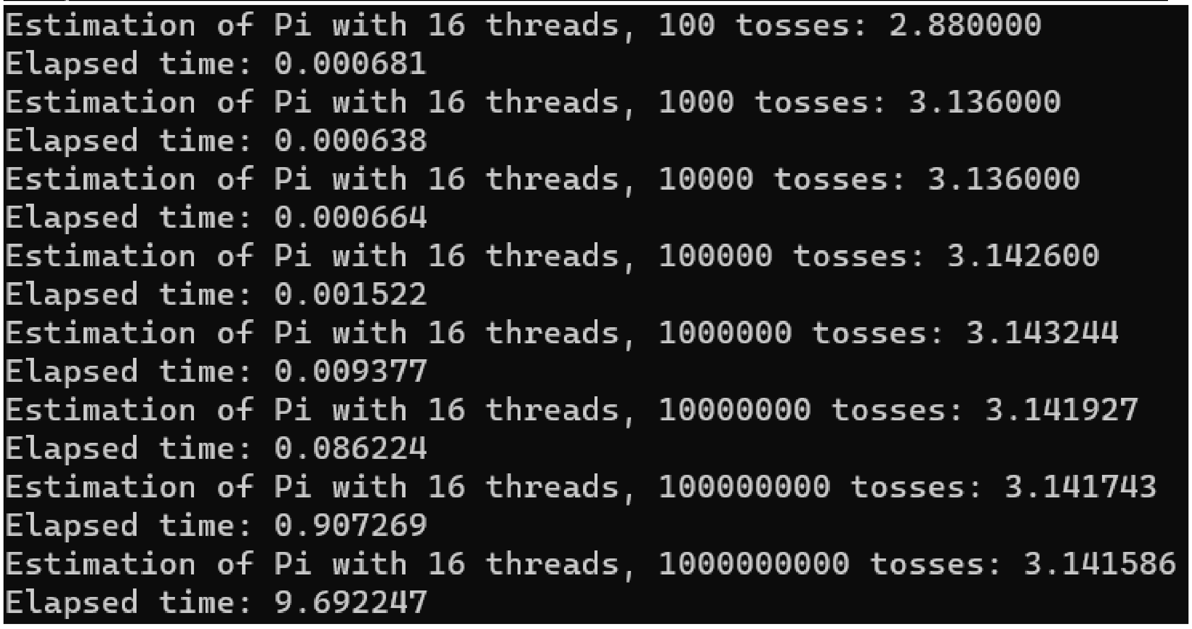

```
mc_thread
Estimation of Pi with 2 threads, 100 tosses: 3.240000
Elapsed time: 0.000162
Estimation of Pi with 2 threads, 1000 tosses: 3.204000
Elapsed time: 0.000187
Estimation of Pi with 2 threads, 10000 tosses: 3.164800
Elapsed time: 0.000447
Estimation of Pi with 2 threads, 100000 tosses: 3.144160
Elapsed time: 0.003112
Estimation of Pi with 2 threads, 1000000 tosses: 3.142564
Elapsed time: 0.028285
Estimation of Pi with 2 threads, 10000000 tosses: 3.142162
Elapsed time: 0.249402
Estimation of Pi with 2 threads, 100000000 tosses: 3.141500
Elapsed time: 1.927614
Estimation of Pi with 2 threads, 1000000000 tosses: 3.141589
Elapsed time: 21.013212
```



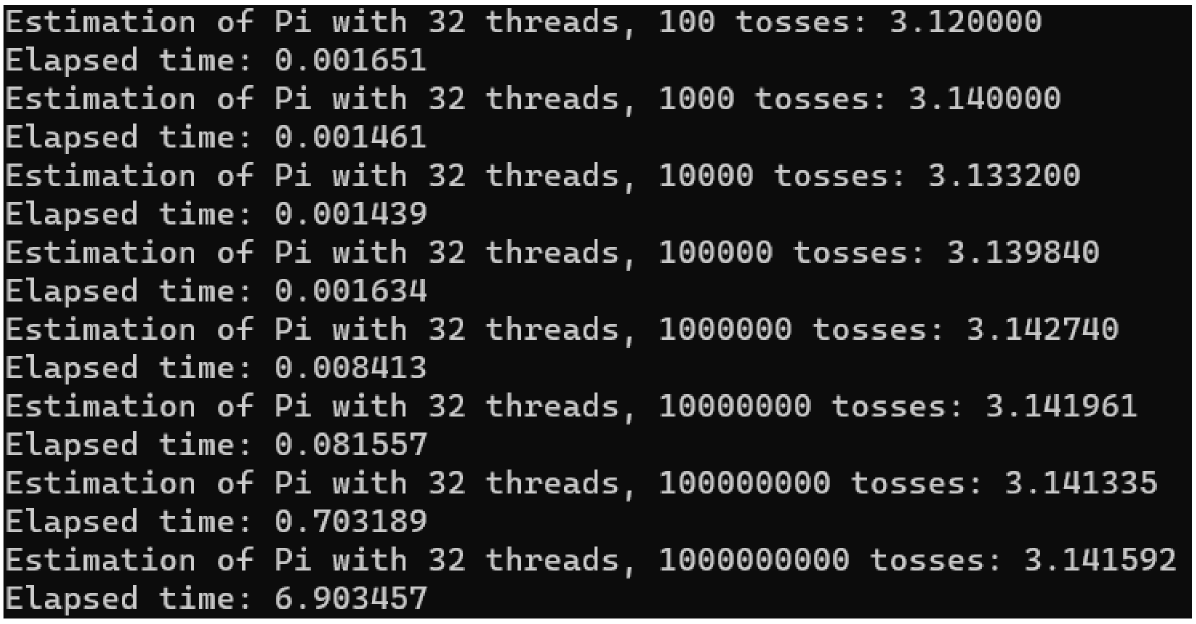
```
Estimation of Pi with 4 threads, 100 tosses: 2.600000
Elapsed time: 0.000214
Estimation of Pi with 4 threads, 1000 tosses: 3.076000
Elapsed time: 0.000212
Estimation of Pi with 4 threads, 10000 tosses: 3.136400
Elapsed time: 0.000289
Estimation of Pi with 4 threads, 100000 tosses: 3.139680
Elapsed time: 0.001672
Estimation of Pi with 4 threads, 1000000 tosses: 3.139648
Elapsed time: 0.017451
Estimation of Pi with 4 threads, 10000000 tosses: 3.141119
Elapsed time: 0.167313
Estimation of Pi with 4 threads, 100000000 tosses: 3.141434
Elapsed time: 2.278301
Estimation of Pi with 4 threads, 1000000000 tosses: 3.141596
Elapsed time: 23.245819
```



```
Estimation of Pi with 8 threads, 100 tosses: 3.240000
Elapsed time: 0.000440
Estimation of Pi with 8 threads, 1000 tosses: 3.136000
Elapsed time: 0.000712
Estimation of Pi with 8 threads, 10000 tosses: 3.122800
Elapsed time: 0.000529
Estimation of Pi with 8 threads, 100000 tosses: 3.136880
Elapsed time: 0.002667
Estimation of Pi with 8 threads, 1000000 tosses: 3.142972
Elapsed time: 0.016220
Estimation of Pi with 8 threads, 10000000 tosses: 3.142330
Elapsed time: 0.157479
Estimation of Pi with 8 threads, 100000000 tosses: 3.141771
Elapsed time: 1.758509
Estimation of Pi with 8 threads, 1000000000 tosses: 3.141582
Elapsed time: 14.702231
```



```
Estimation of Pi with 16 threads, 100 tosses: 2.880000
Elapsed time: 0.000681
Estimation of Pi with 16 threads, 1000 tosses: 3.136000
Elapsed time: 0.000638
Estimation of Pi with 16 threads, 10000 tosses: 3.136000
Elapsed time: 0.000664
Estimation of Pi with 16 threads, 100000 tosses: 3.142600
Elapsed time: 0.001522
Estimation of Pi with 16 threads, 1000000 tosses: 3.143244
Elapsed time: 0.009377
Estimation of Pi with 16 threads, 10000000 tosses: 3.141927
Elapsed time: 0.086224
Estimation of Pi with 16 threads, 100000000 tosses: 3.141743
Elapsed time: 0.907269
Estimation of Pi with 16 threads, 1000000000 tosses: 3.141586
Elapsed time: 9.692247
```



```
Estimation of Pi with 32 threads, 100 tosses: 3.120000
Elapsed time: 0.001651
Estimation of Pi with 32 threads, 1000 tosses: 3.140000
Elapsed time: 0.001461
Estimation of Pi with 32 threads, 10000 tosses: 3.133200
Elapsed time: 0.001439
Estimation of Pi with 32 threads, 100000 tosses: 3.139840
Elapsed time: 0.001634
Estimation of Pi with 32 threads, 1000000 tosses: 3.142740
Elapsed time: 0.008413
Estimation of Pi with 32 threads, 10000000 tosses: 3.141961
Elapsed time: 0.081557
Estimation of Pi with 32 threads, 100000000 tosses: 3.141335
Elapsed time: 0.703189
Estimation of Pi with 32 threads, 1000000000 tosses: 3.141592
Elapsed time: 6.903457
```

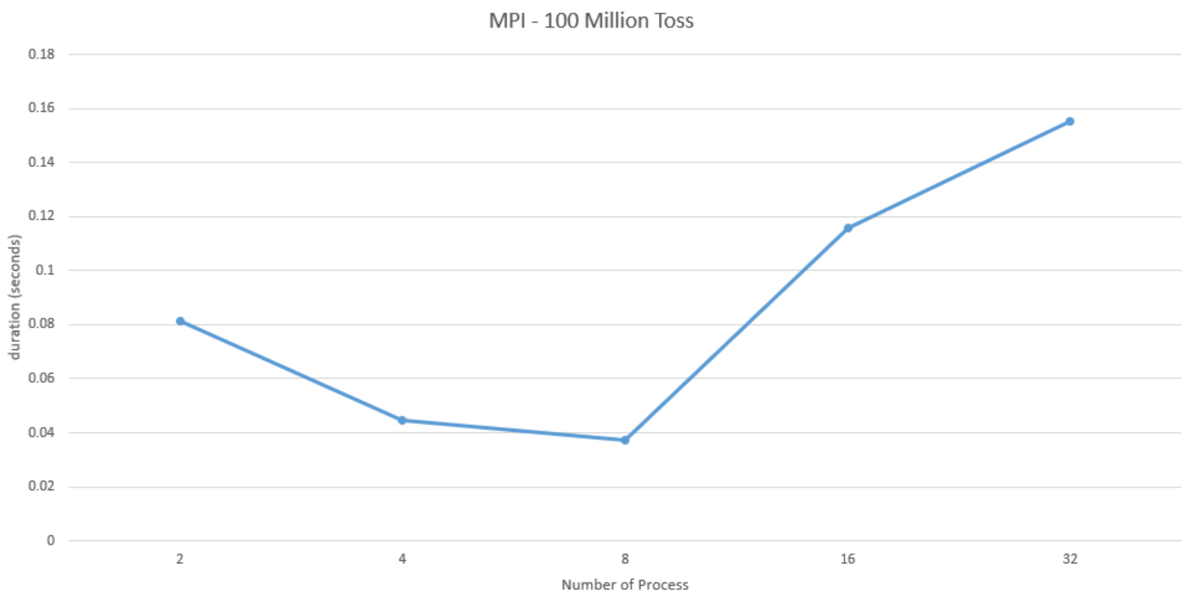
6- Result Analysis

A. Execution Time Comparison

N: Number of processes
K: Number of threads

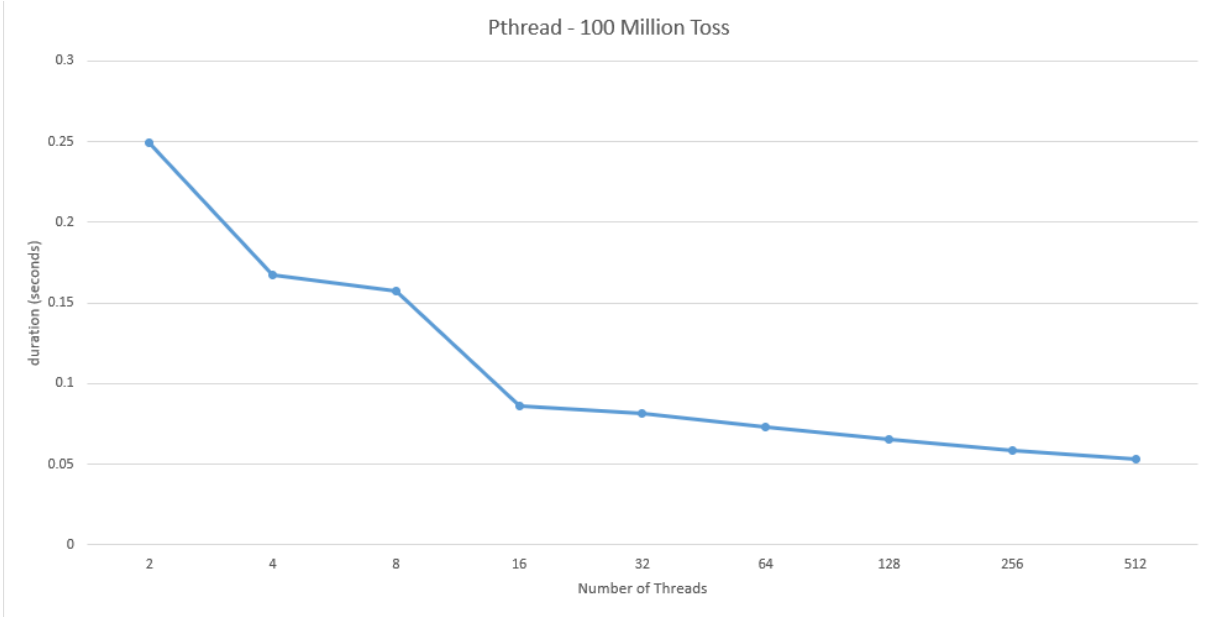
- Execution time and accuracy of pi compared between N process MPI execution, K thread pthread execution, and single thread execution. N and K values are 2, 4, 8, 16, 32. So there are 5 execution time graphs.
- I did not test single thread with pthread library, it is a direct program. So, overhead of library scheduling does not apply for single thread case.

MPI performance for 100 million Toss:



- MPI execution time performs better till 8 processes, but it becomes worse after 8. Because there are 8 core in the processor. It utilizes perfectly till 8. After 8, All processes cannot run concurrently, so “critical path length” increases. This will cause performance degradation.

Pthread Performance for 100 million Toss:

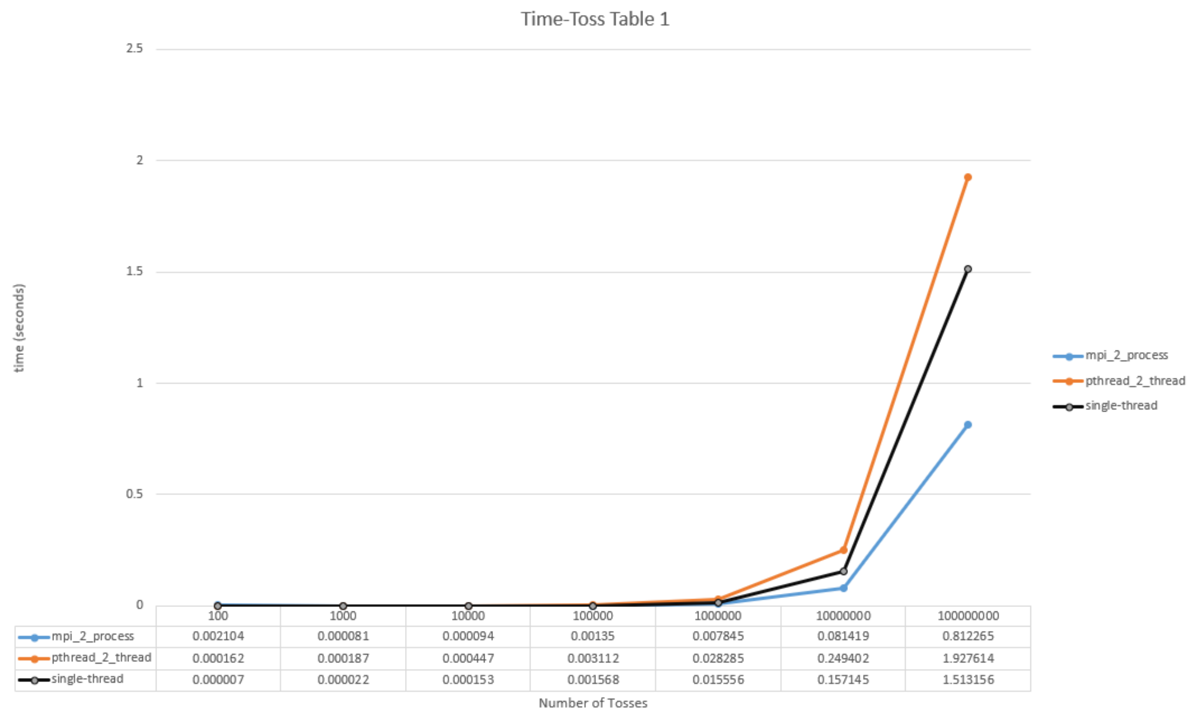


- If we increase number of threads performance will get better. Even if there are 8 core in the system, still performance get better. Because, scheduling is not precise as MPI scheduling. When we execute 8 thread, it is not guaranteed that 8 thread will use different core. If number of threads increase to at some point, performance will become better because parallelization chance will increase.

Single Thread Performance for 100 million Toss:

- Single thread without using Pthread library took 0.157 seconds. It is worse than MPI for number of processes less than 64 and it is worse than multithread which has more threads than 8.
- If concurrency has not fully provided and at some point overhead causes concurrent programs give worse performance than serial programs.

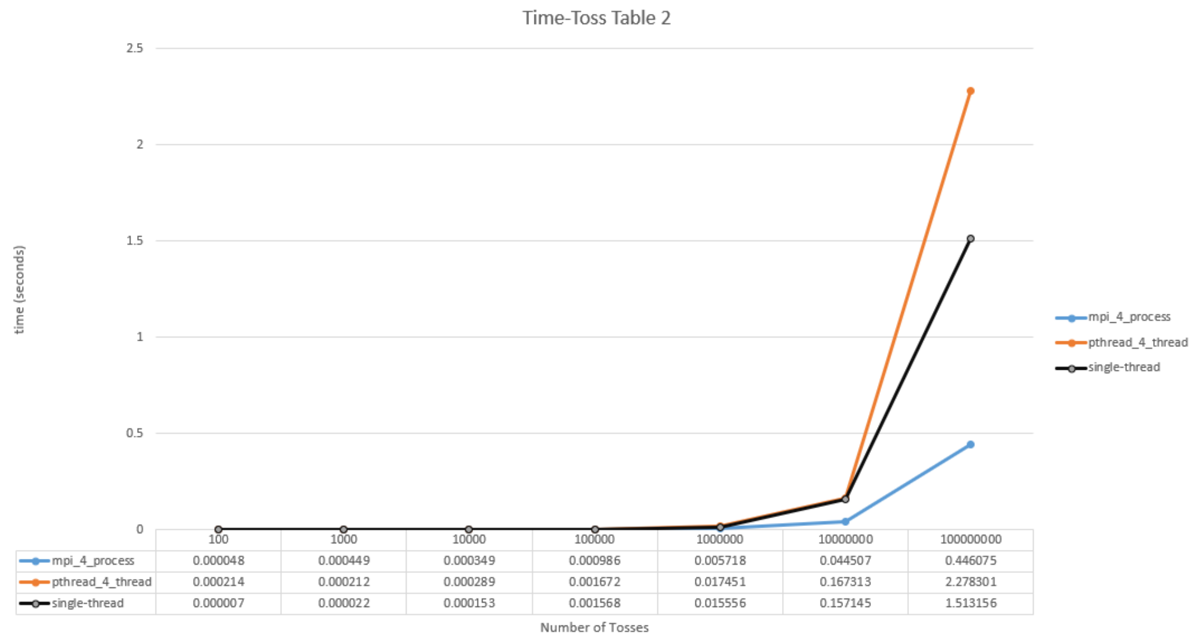
Duration (seconds) of MPI (N = 2), Pthread (K = 2), Single thread:



	100	1000	10000	100000	1000000	10000000	100000000
mpi_2_process	0.002104	0.000081	0.000094	0.00135	0.007845	0.081419	0.812265
pthread_2_thread	0.000162	0.000187	0.000447	0.003112	0.028285	0.249402	1.927614
single-thread	0.000007	0.000022	0.000153	0.001568	0.015556	0.157145	1.513156

- MPI took the least time, thread is the longest. 2 threads do not enough for it to perform computation in less time. Overhead of creating and joining threads causes this difference. Also, it is dependent on the library and computing environment.
- MPI worked better because total core is 8, and there are 2 processes. MPI library used different cores. I expected pthread to perform better than single thread but it didn’t utilize as good as MPI.

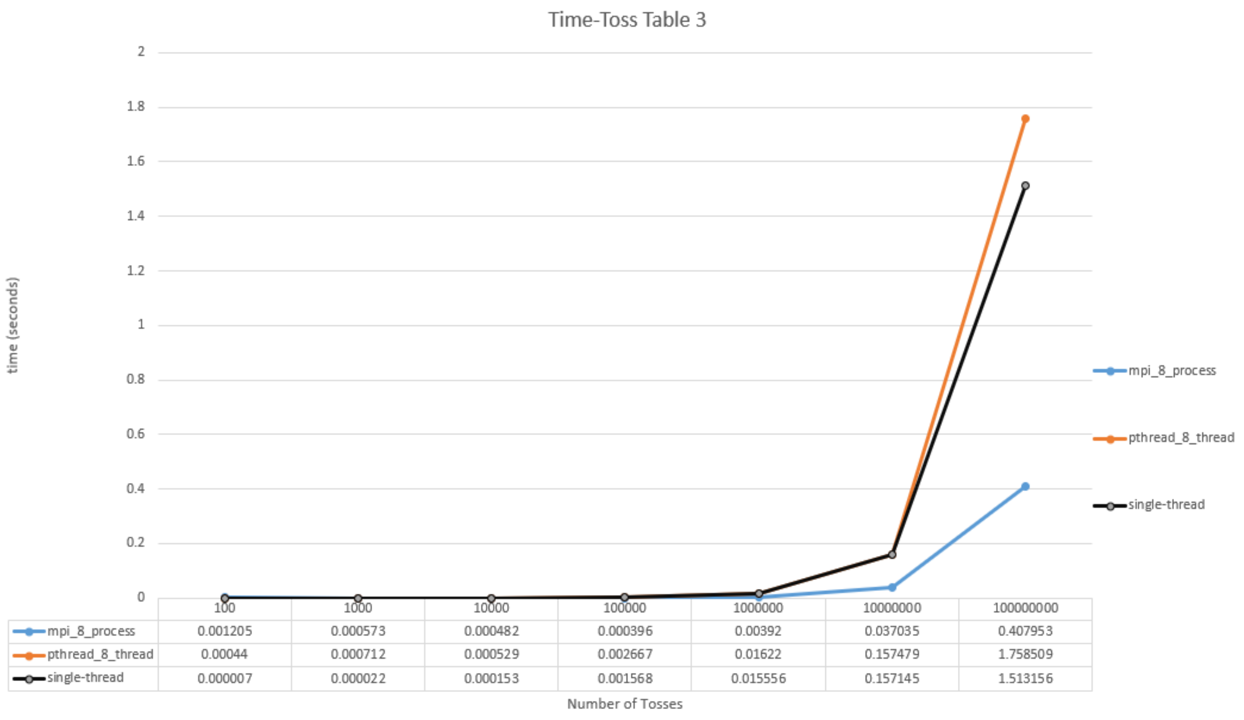
Duration (seconds) of MPI (N = 4), Pthread (K = 4), Single thread:



	100	1000	10000	100000	1000000	10000000	100000000
mpi_4_process	0.000048	0.000449	0.000349	0.000986	0.005718	0.044507	0.446075
pthread_4_thread	0.000214	0.000212	0.000289	0.001672	0.017451	0.167313	2.278301
single-thread	0.000007	0.000022	0.000153	0.001568	0.015556	0.157145	1.513156

- MPI is still best. Graph is similar to the first one. Multithread duration becoming closer for the tosses less than 100 million. Still, pthread performance is not good for concurrency.

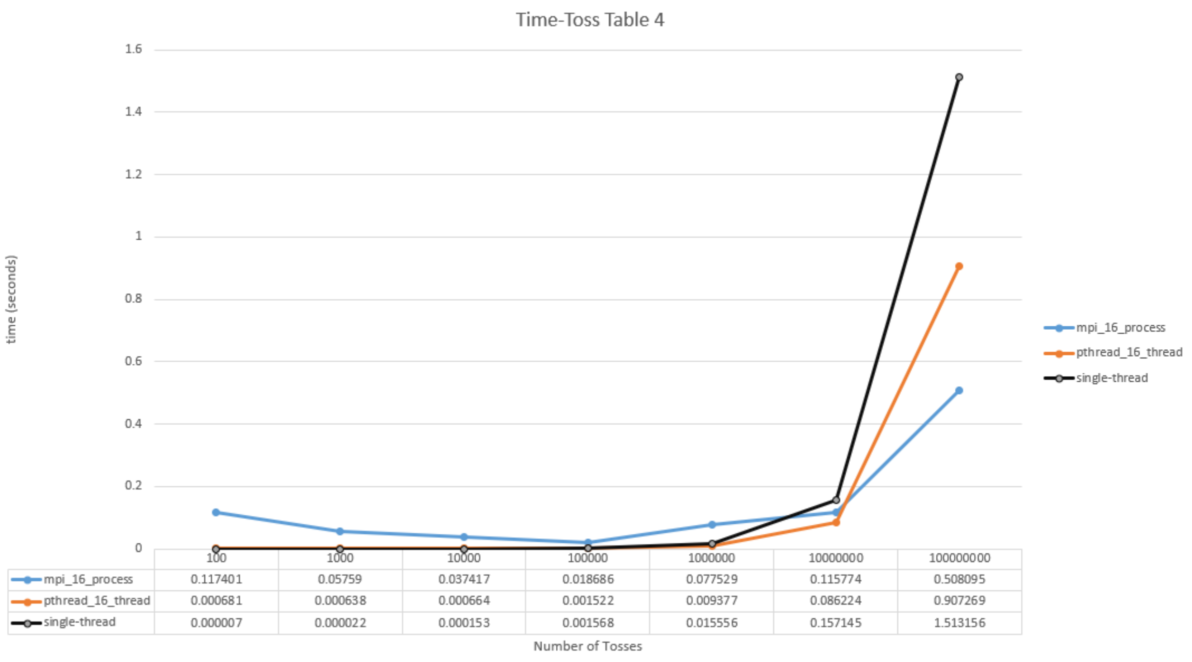
Duration (seconds) of MPI (N = 8), Pthread (K = 8), Single thread:



	100	1000	10000	100000	1000000	10000000	100000000
mpi_8_process	0.001205	0.000573	0.000482	0.000396	0.00392	0.037035	0.407953
pthread_8_thread	0.00044	0.000712	0.000529	0.002667	0.01622	0.157479	1.758509
single-thread	0.000007	0.000022	0.000153	0.001568	0.015556	0.157145	1.513156

- MPI uses all 8 cores. It is still best. Multithread is very close to single thread.

Duration (seconds) of MPI (N = 16), Pthread (K = 16), Single thread:

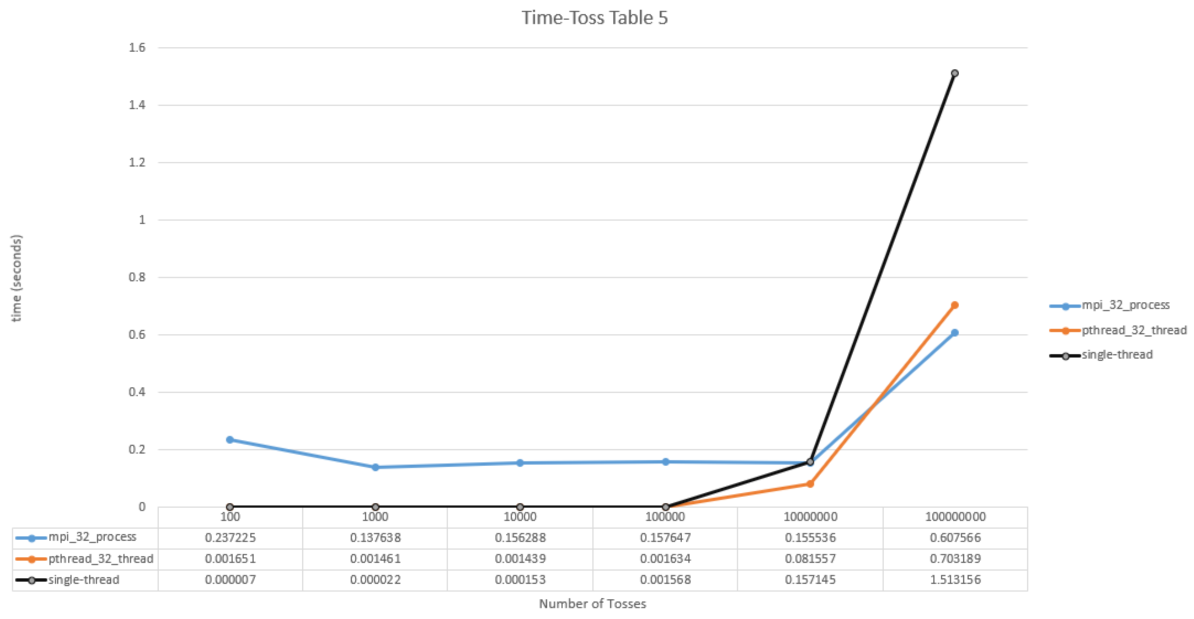


	100	1000	10000	100000	1000000	10000000	100000000
mpi_16_process	0.117401	0.05759	0.037417	0.018686	0.077529	0.115774	0.508095
pthread_16_thread	0.000681	0.000638	0.000664	0.001522	0.009377	0.086224	0.907269
single-thread	0.000007	0.000022	0.000153	0.001568	0.015556	0.157145	1.513156

- This time pthread is clearly better than single thread. After 100000 toss, it performed much better than single thread.

- MPI this time isn’t best. Especially for small tosses. Because CPU utilization is problematic this time. Total core number is 8, but there are 16 processes. So, MPI performance decreased this time. Still best for the largest value. But its performance worser than 8 processes.

Duration (seconds) of MPI (N = 32), Pthread (K = 32), Single thread:



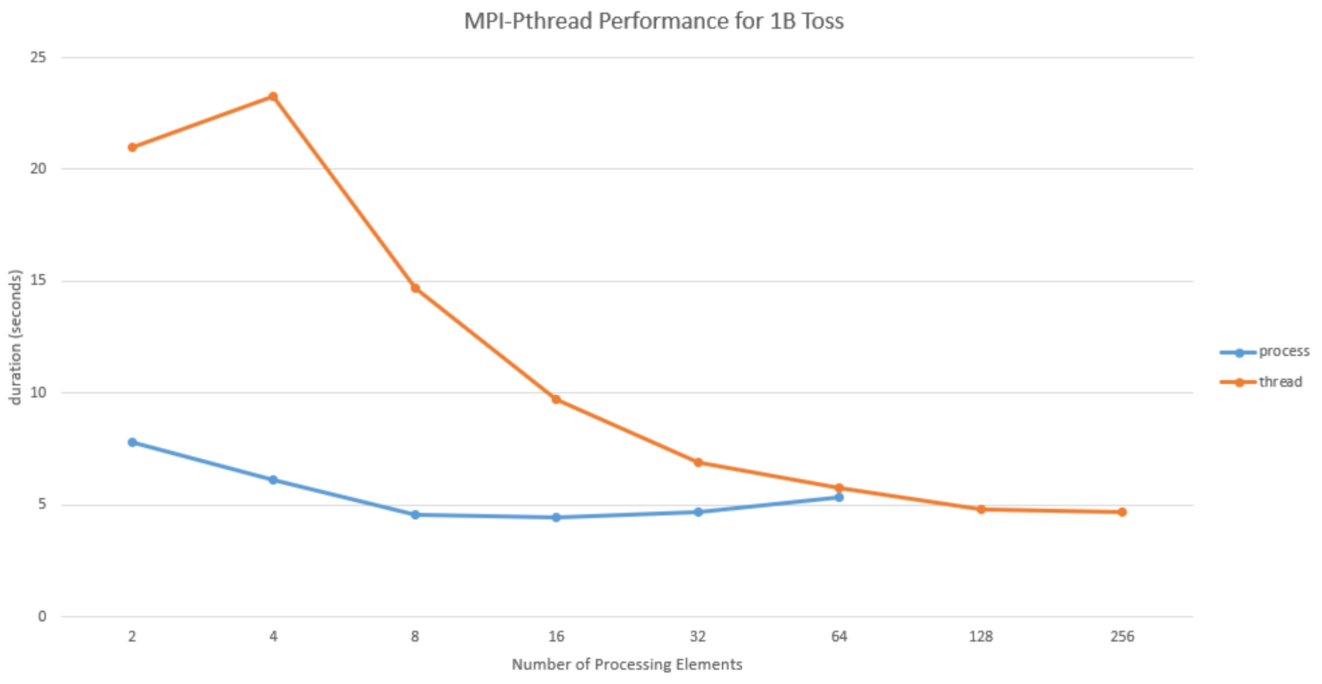
	100	1000	10000	100000	10000000	100000000
mpi_32_process	0.237225	0.137638	0.156288	0.157647	0.155536	0.607566
pthread_32_thread	0.001651	0.001461	0.001439	0.001634	0.081557	0.703189
single-thread	0.000007	0.000022	0.000153	0.001568	0.157145	1.513156

- Pthread is very close to MPI for the largest value, because it done utilization better when there are 32 processing units. Also, it is better than single thread.

B. MPI vs Pthread

I compared MPI and Pthread with different number of processing elements to test which does parallelism better for Monte Carlo Problem.

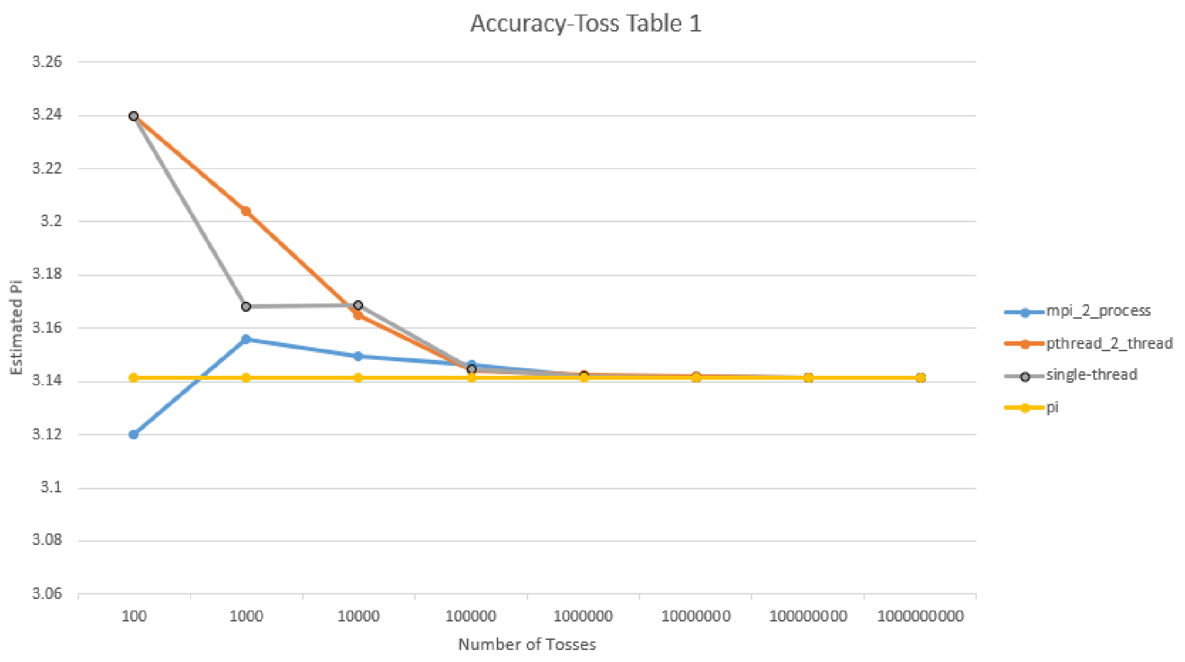
The results for 1 billion toss are compared. For MPI maximum process is 64 because there is a limit. Pthreads thread limit is much more than MPI.



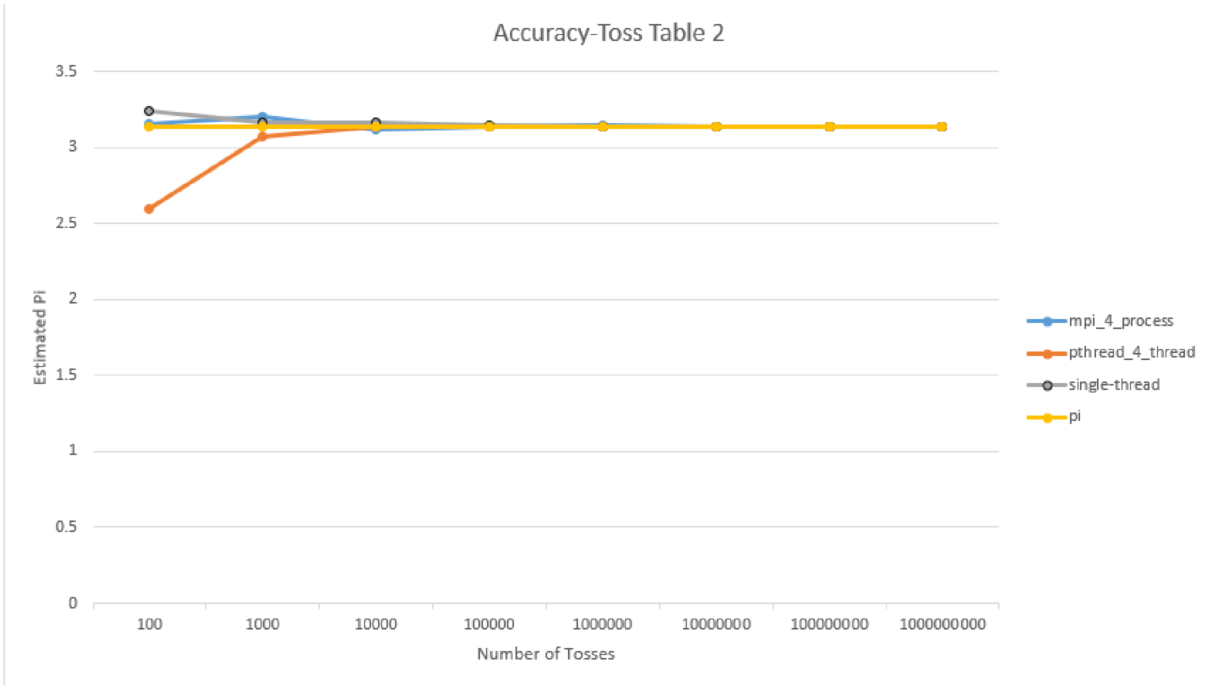
- For this problem MPI gave better results if the number of processing elements is small. I cannot test more than 64 for MPI because it is the maximum for my system. But it can be seen that Pthread's execution time significantly decreases if number of threads increases. MPI's execution time also decreases, but its decreasing is slower than Pthread.
- MPI performance becomes worse after 8, because there are 8 cores in my system. So, its performance decreases. Pthread becomes better because its chance of concurrency increases if we use more threads. Pthread does not guarantee fully concurrency, therefore if we use more threads concurrency possibility increases. MPI is more precise than pthread.
- Normally, I expected Pthread to perform better than MPI because it runs on a single process, shares data and address space. It is dependent on the library, and problem type. If problem were required more communication, pthread would perform better because communication in same address space is faster than message passing.

C. Accuracy of Prediction

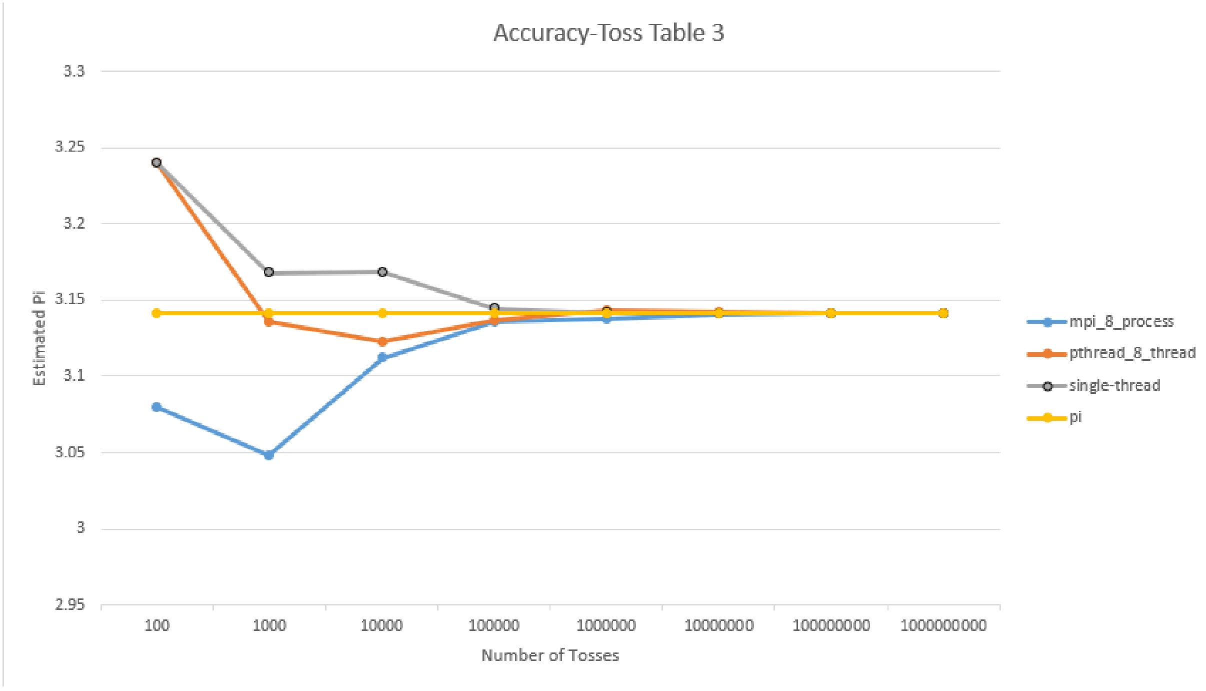
Estimated Pi of MPI (N = 2), Pthread (K = 2), Single thread:



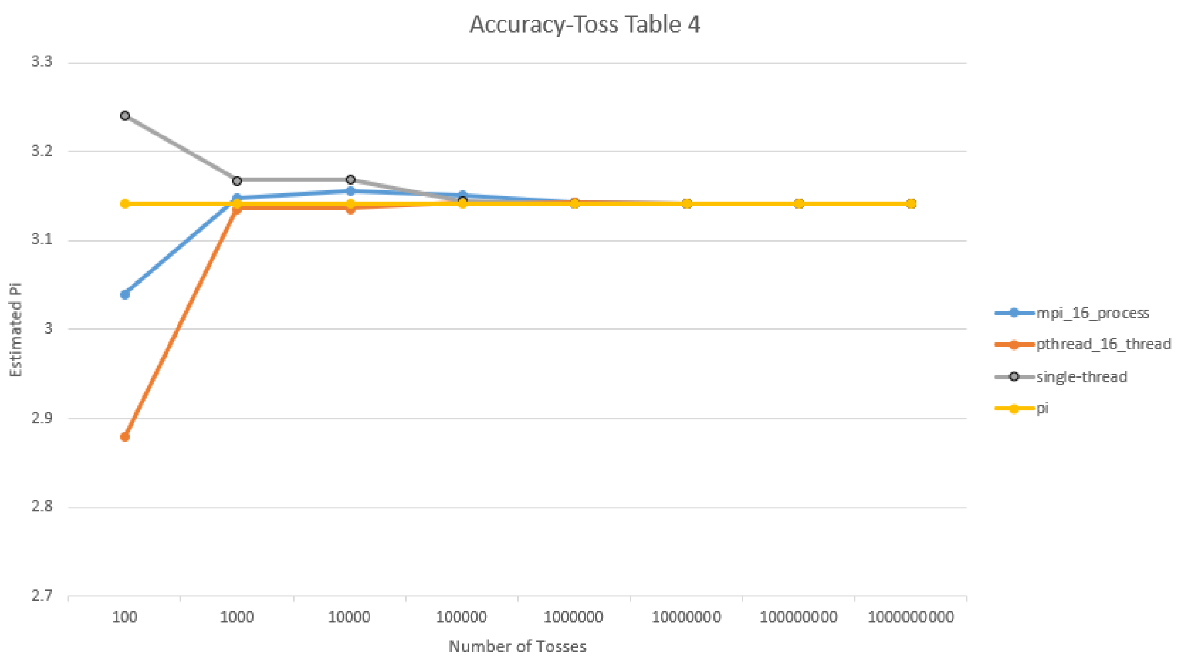
Estimated Pi of MPI (N = 4), Pthread (K = 4), Single thread:



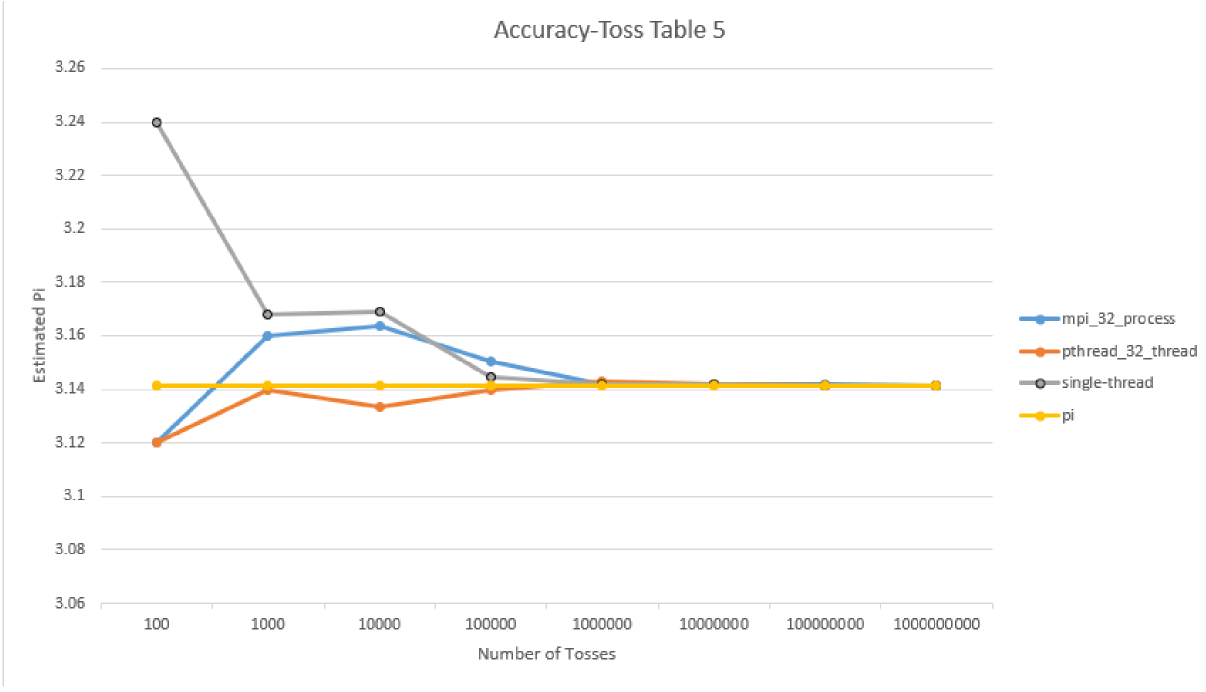
Estimated Pi of MPI (N = 8), Pthread (K = 8), Single thread:



Estimated Pi of MPI (N = 16), Pthread (K = 16), Single thread:

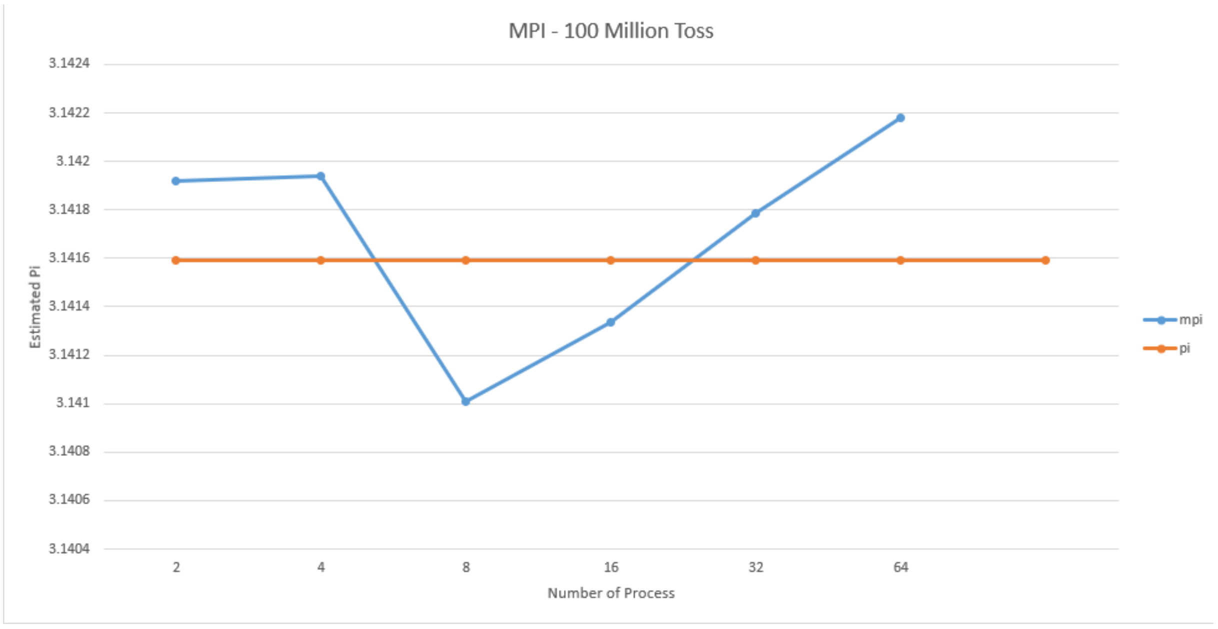


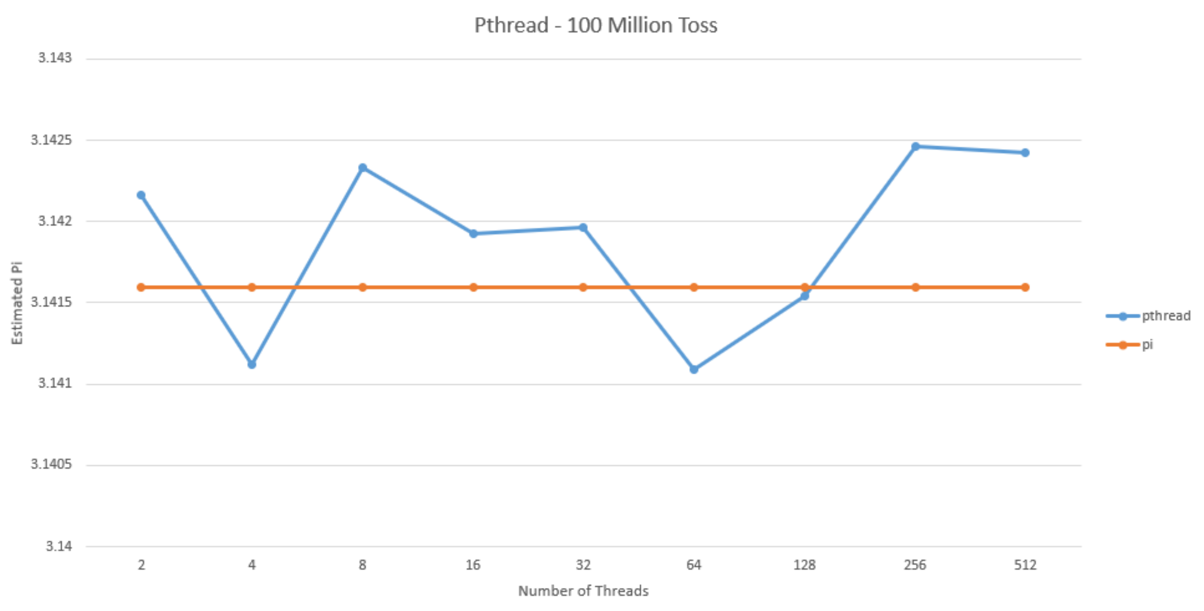
Estimated Pi of MPI (N = 32), Pthread (K = 32), Single thread:



- Independent of the processing elements, their accuracy increases with number of tosses. For single thread, multithread, and multiple processes pi accuracy is better. It is independent from thread, process or serial implementation is used.

Accuracy – Number of Processing Units Correlation Test:





- As it can be seen, there is no correlation between number of processing units and estimation. Accuracy of the estimation is only dependent on number of tosses.