DTS202TC Foundation of Parallel Computing

DTS202TC Foundation of Parallel Computing
- I. Lab 2 (30 marks)
  - 1.1 Task1
    - 1.1.1 Code
    - 1.1.2 Screenshot
  - 1.2 Task2
    - 1.2.1 Code
    - 1.2.2 Screenshot
  - 1.3 Task3
    - 1.3.1 Code
    - 1.3.2 Screenshot
    - 1.3.3 Analysis the performance
- II. Lab 3 (35 Marks)
  - 2.1 Profiling
    - 2.1.1 Code
    - 2.1.2 Screenshot
    - 2.1.3 Analysis.txt
  - 2.2 Pthreads Hello World
    - 2.2.1 Code
    - 2.2.2 Screenshot
  - 2.3 Estimate π with Pthreads
    - 2.3.1 Code
    - 2.3.2 Screenshot
- III. Lab 4 (35 Marks)
  - 3.1 OpenMP Hello World
    - 3.1.1 Code
    - 3.1.2 Screenshot
  - 3.2 Estimate π with OpenMP
    - 3.2.1 Code
    - 3.2.2 Screenshot
  - 3.3 Comparison
    - 3.3.1 Comparison under different threads
      - 3.3.1.1 Elapsed Time
      - 3.3.1.2 Speedup
      - 3.3.1.3 Efficiency
    - 3.3.2 Comparison under different problem size

I. Lab 2 (30 marks)

1.1 Task1

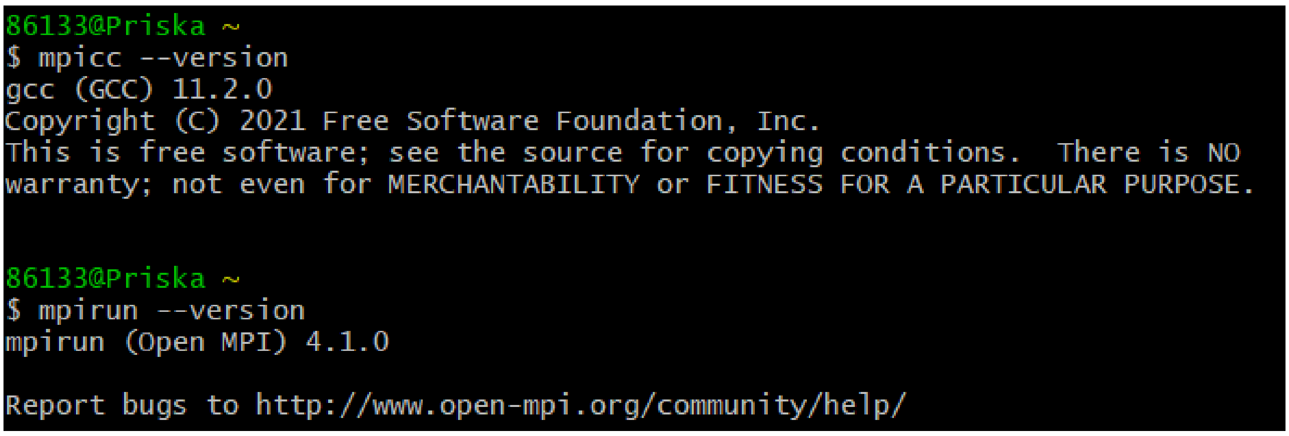
The author's computer system is Windows10, so using Cygwin to compile `.c` files. OpenMPI and related Package have been successfully installed on PC, and `.c` files with mpi header files can be successfully compiled on the computer.

Prove that the installation is successful by using a line command and compiling a test `.c` file

1.1.1 Code

```
mpicc --version
mpirun --version
```

1.1.2 Screenshot



```
86133@Priska ~
$ mpicc --version
gcc (GCC) 11.2.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.


86133@Priska ~
$ mpirun --version
mpirun (Open MPI) 4.1.0

Report bugs to http://www.open-mpi.org/community/help/
```

1.2 Task2

1.2.1 Code

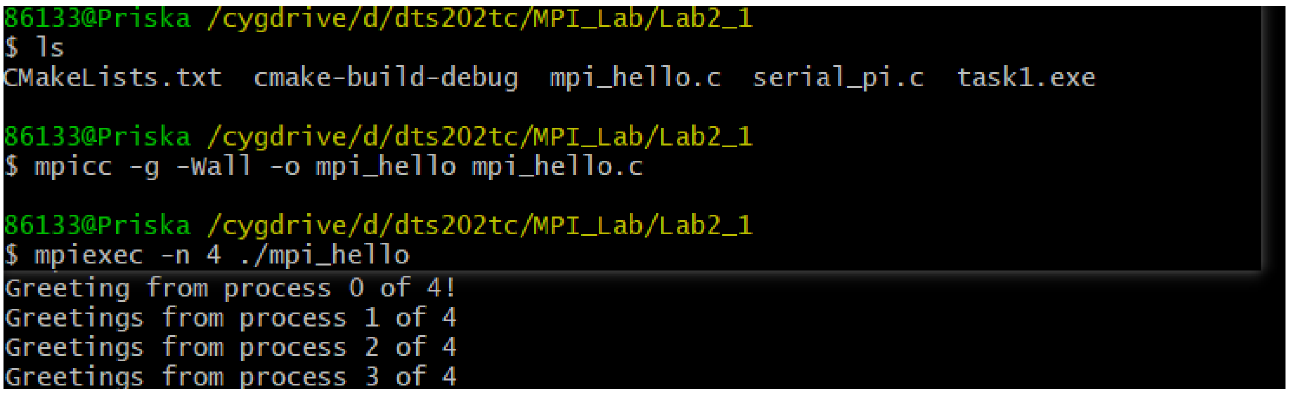
`Shell`

```shell
mpicc -g -Wall -o mpi_hello mpi_hello.c
mpiexec -n 4 ./mpi_hello
mpiexec -n 1 ./mpi_hello
mpiexec -n 2 ./mpi_hello
```

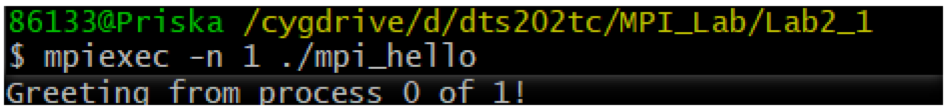
`.c`

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>

const int MAX_STRING = 100;

int main(void){
    char greeting[MAX_STRING];
    int comm_sz;
    int my_rank;

    MPI_Init(NULL,NULL);
    MPI_Comm_size(MPI_COMM_WORLD,&comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

    if(my_rank != 0){
        sprintf(greeting,"Greetings from process %d of %d",
                my_rank,comm_sz);
 MPI_Send(greeting,strlen(greeting)+1,MPI_CHAR,0,0,MPI_COMM_WORLD);
    } else{
        printf("Greeting from process %d of %d! \n",my_rank,
              comm_sz);
        for (int q = 1; q < comm_sz; q++) {
                MPI_Recv(greeting,MAX_STRING,MPI_CHAR,q,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                printf("%s\n",greeting);
            }
        }

    MPI_Finalize();
    return 0;
}
```

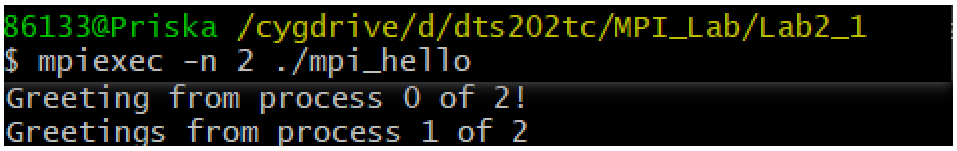
1.2.2 Screenshot



```
86133@Priska /cygdrive/d/dts202tc/MPI_Lab/Lab2_1
$ ls
CMakeLists.txt  cmake-build-debug  mpi_hello.c  serial_pi.c  task1.exe

86133@Priska /cygdrive/d/dts202tc/MPI_Lab/Lab2_1
$ mpicc -g -Wall -o mpi_hello mpi_hello.c

86133@Priska /cygdrive/d/dts202tc/MPI_Lab/Lab2_1
$ mpiexec -n 4 ./mpi_hello
Greeting from process 0 of 4!
Greetings from process 1 of 4
Greetings from process 2 of 4
Greetings from process 3 of 4
```



```
86133@Priska /cygdrive/d/dts202tc/MPI_Lab/Lab2_1
$ mpiexec -n 1 ./mpi_hello
Greeting from process 0 of 1!
```



```
86133@Priska /cygdrive/d/dts202tc/MPI_Lab/Lab2_1
$ mpiexec -n 2 ./mpi_hello
Greeting from process 0 of 2!
Greetings from process 1 of 2
```

1.3 Task3

1.3.1 Code

```
gcc -g -Wall -o serial_pi serial_pi.c
./serial_pi

mpicc -g -Wall -o parallel_pi parallel_pi.c
mpiexec -n 1 ./parallel_pi.c
mpiexec -n 2 ./parallel_pi.c
mpiexec -n 3 ./parallel_pi.c
mpiexec -n 4 ./parallel_pi.c
```

- serial_pi.c

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sys/time.h>

#define GET_TIME(now){ \
    struct timeval t;     \
    gettimeofday(&t,NULL);\
    now = t.tv_sec+t.tv_sec/1000000.0;\
}

double serial_pi(long long n);
int main(int argc,char** argv) {
    double start,finish;
    GET_TIME(start);
    double estimate_of_pi = serial_pi(1000000000);
    printf("\n Estimated of pi: %1.10f.\n",estimate_of_pi);
    GET_TIME(finish);

    printf("\nAcutal value of pi: %1.10f.\n\n", atan(1)*4);
    printf("The elapsed time is %e seconds\n",finish-start);
}

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;
}
```

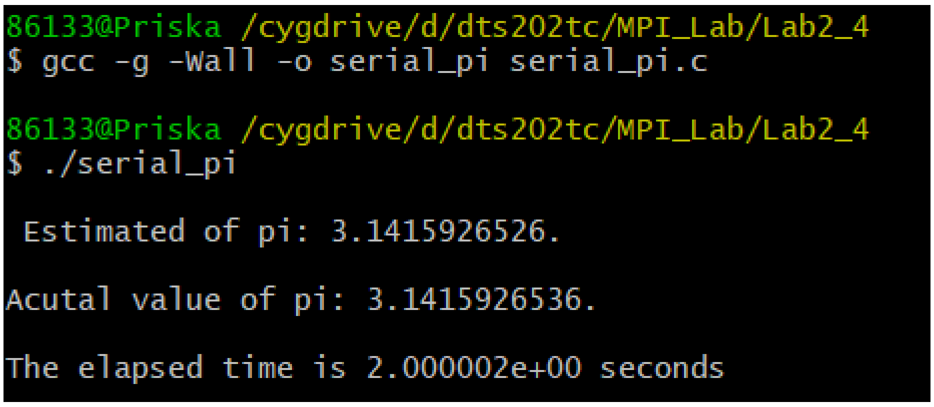
- parallel_pi.c

```
  /*File:
   *  parallel_pi.c
   *
   *Purpose:
   * Using MPI_Send and MPI_Recv to estimate_pi
   *
   *Compile:
   * mpicc -g -Wall -o parallel_pi parallel_pi.c
   *Usage:
   * mpiexec -n<number of processes> ./parallel_pi
   *
   *Input:
   *  None
   *Output:
   * n processes estimates the value of the PI
   *
   * Algorithm:
   *  1. Each process calculates its interval of integration pi
   *  2. Each process != 0 sends its integral to 0 process
   *  3. Process 0 sums the calculations received from the
individual
   *     processes and prints the result
   * */
  #include <stdio.h>
  #include <mpi.h>
  #include <math.h>
  #include <sys/time.h>

  #define GET_TIME(now){ \
      struct timeval t;     \
```

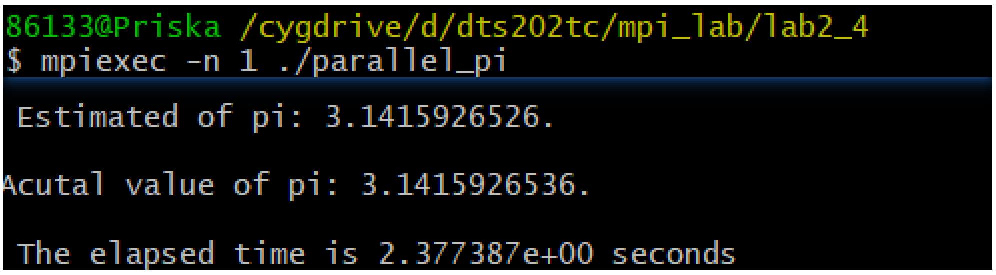
```c
      gettimeofday(&t,NULL);\
      now = t.tv_sec+t.tv_sec/1000000.0;\
  }
  double serial_pi(long long a,long long b);/*Calculate local integral*/

  int main(void) {
      int my_rank,comm_sz,source;/*My process rank, Number of processes*/
      long long n=1000000000,local_n,local_a,local_b;
      double local_est, total_est;
      double local_start,local_finish,local_elapsed,elapsed;

      MPI_Init(NULL,NULL);
      MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
      MPI_Comm_size(MPI_COMM_WORLD,&comm_sz);

      local_n = n/comm_sz;
      local_a = 0+my_rank*local_n;
      local_b = local_a+local_n;


      MPI_Barrier(MPI_COMM_WORLD);
      local_start=MPI_Wtime();
      local_est = serial_pi(local_a,local_b);

      if (my_rank!=0){
          MPI_Send(&local_est,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);

      } else{
          total_est=local_est;
          for (source = 1; source < comm_sz; source++) {
 MPI_Recv(&local_est,1,MPI_DOUBLE,source,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
              total_est+=local_est;
          }
      }
      local_finish = MPI_Wtime();
      local_elapsed = local_finish-local_start;
 MPI_Reduce(&local_elapsed,&elapsed,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
```

```c

    if (my_rank==0){
        printf("\n Estimated of pi: %1.10f.\n",total_est);
        printf("\nAcutal value of pi: %1.10f.\n\n",
atan(1)*4);
    }
    if(my_rank==0) {
        printf(" The elapsed time is %e seconds\n", elapsed);
    }
    MPI_Finalize();

    return 0;
}

double serial_pi(long long a,long long b){
    double sum = 0.0;
    double factor;
    if (a/2 == 0){
        factor = 1.0;
    } else{
        factor = -1.0;
    }

    for (; a < b; a++,factor=-factor) {
        sum += factor/(2*a+1);
    }

    return 4.0*sum;
}
```

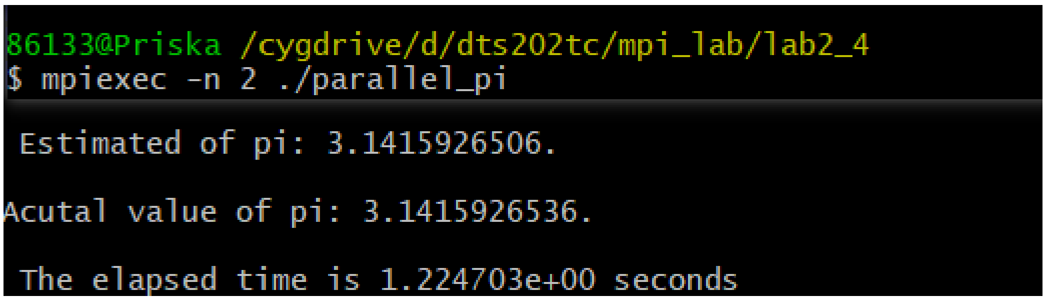
1.3.2 Screenshot



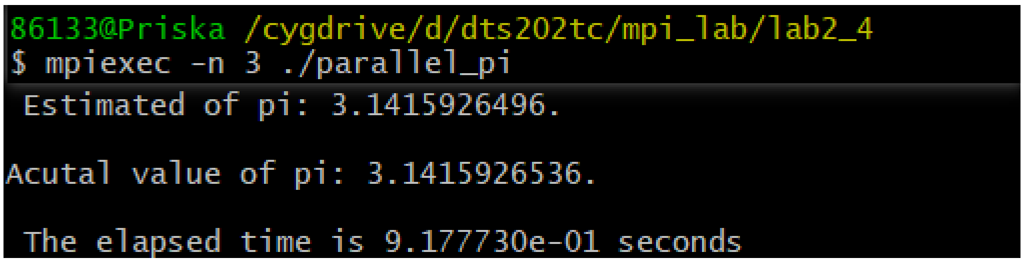
```
86133@Priska /cygdrive/d/dts202tc/MPI_Lab/Lab2_4
$ gcc -g -Wall -o serial_pi serial_pi.c

86133@Priska /cygdrive/d/dts202tc/MPI_Lab/Lab2_4
$ ./serial_pi

 Estimated of pi: 3.1415926526.

Acutal value of pi: 3.1415926536.

The elapsed time is 2.000002e+00 seconds
```



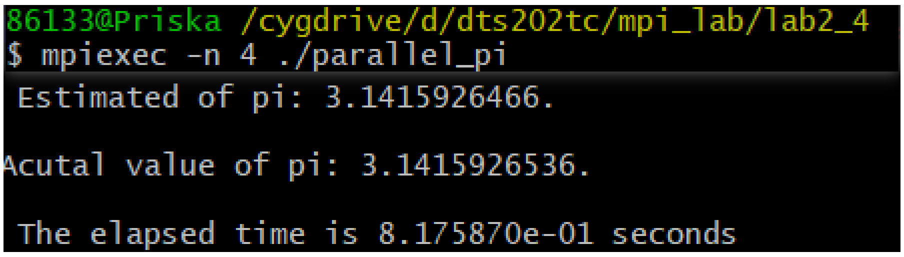
```
86133@Priska /cygdrive/d/dts202tc/mpi_lab/lab2_4
$ mpiexec -n 1 ./parallel_pi
 Estimated of pi: 3.1415926526.

Acutal value of pi: 3.1415926536.

 The elapsed time is 2.377387e+00 seconds
```



```
86133@Priska /cygdrive/d/dts202tc/mpi_lab/lab2_4
$ mpiexec -n 2 ./parallel_pi

 Estimated of pi: 3.1415926506.

Acutal value of pi: 3.1415926536.

 The elapsed time is 1.224703e+00 seconds
```



```
86133@Priska /cygdrive/d/dts202tc/mpi_lab/lab2_4
$ mpiexec -n 3 ./parallel_pi
 Estimated of pi: 3.1415926496.

Acutal value of pi: 3.1415926536.

 The elapsed time is 9.177730e-01 seconds
```



```
86133@Priska /cygdrive/d/dts202tc/mpi_lab/lab2_4
$ mpiexec -n 4 ./parallel_pi
 Estimated of pi: 3.1415926466.

Acutal value of pi: 3.1415926536.

 The elapsed time is 8.175870e-01 seconds
```

1.3.3 Analysis the performance

	1 PROCESSOR	2 PROCESSOR	3 PROCESSOR	4 PROCESSOR
Serial_pi	2.00 s			
Parallel_pi	2.37 s	1.22 s	0.92 s	0.82 s

Generally, the paralle program of 1 process should have the same time as the serial program, but there is a difference of time. This is due to the different time calculation functions used by the two programs. The parallel program uses the following time function to time the mpi program

```c
double local_start,local_finish,local_elapsed,elapsed;
...
MPI_Barrier(comm);
local_start = MPI_Wtime();
/*Code to be timed*/
...
local_finish = MPI_Wtime;
local_elapsed = local_finish - local_start;
MPI_Reduce(&localz_elapsed,&elapsed,1,MPI_DOUBLE,MPI_MAX,0,comm)
;

if(my_rank==0){
    printf("Elapsed time = %e seconds\n",elapsed);
}
```

Such a timing function causes time overhead, so it is slightly different from serial time.

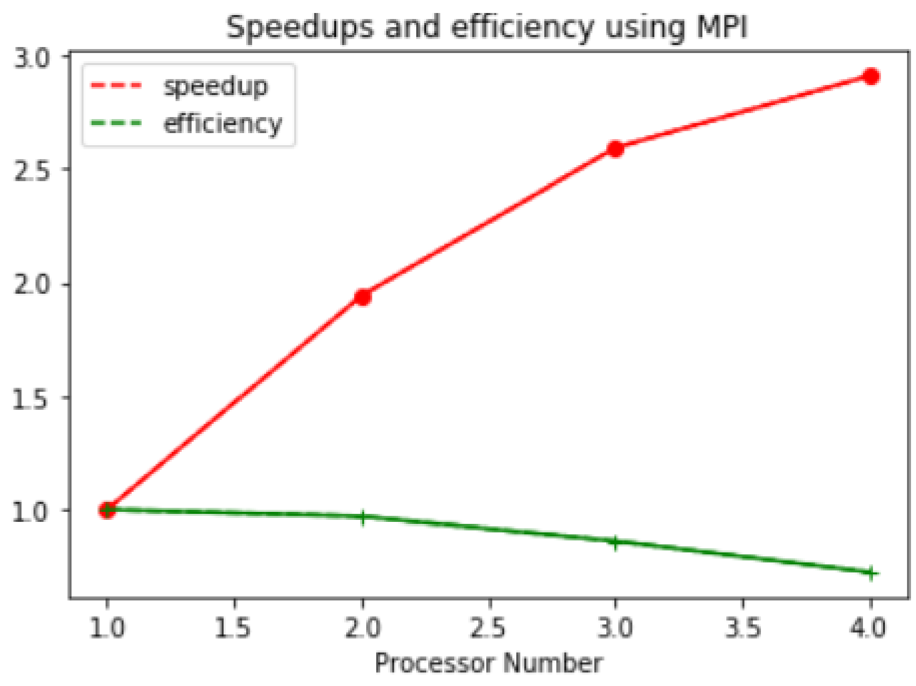
MPI Parallel Algorithm:

- Each process calculates its interval of integration pi
- Each process != 0 sends its integral to 0 process
- Process 0 sums the calculations received from the individual processes and prints the result

Such a parallel algorithm significantly shortens the time of serial programs.

Then compare the performance of different processor by analyzing speedup and efficiency

	1 PROCESSOR	2 PROCESSORS	3 PROCESSORS	4 PROCESSORS
speedup	1	1.94	2.59	2.91
efficiency	1	0.97	0.86	0.73



When the size of the problem remains the same, the speedup increases as the number of processes increases, because the more processes, the less time the program takes to execute. Ideally, speedup should be linear speedup, that is, the number of processes. And there will be an upper limit according to Amdahl's Law,speedup, but due to the limit on the number of machine processor, it is not possible to try more processes to see when the limit is reached.

However, it is found that the efficiency is decreasing, which is that when the Processor increases, although the time decreases, the relative communication overhead also increases, that is, the more processor, the greater the communication overhead, the lower the efficiency.

II. Lab 3 (35 Marks)

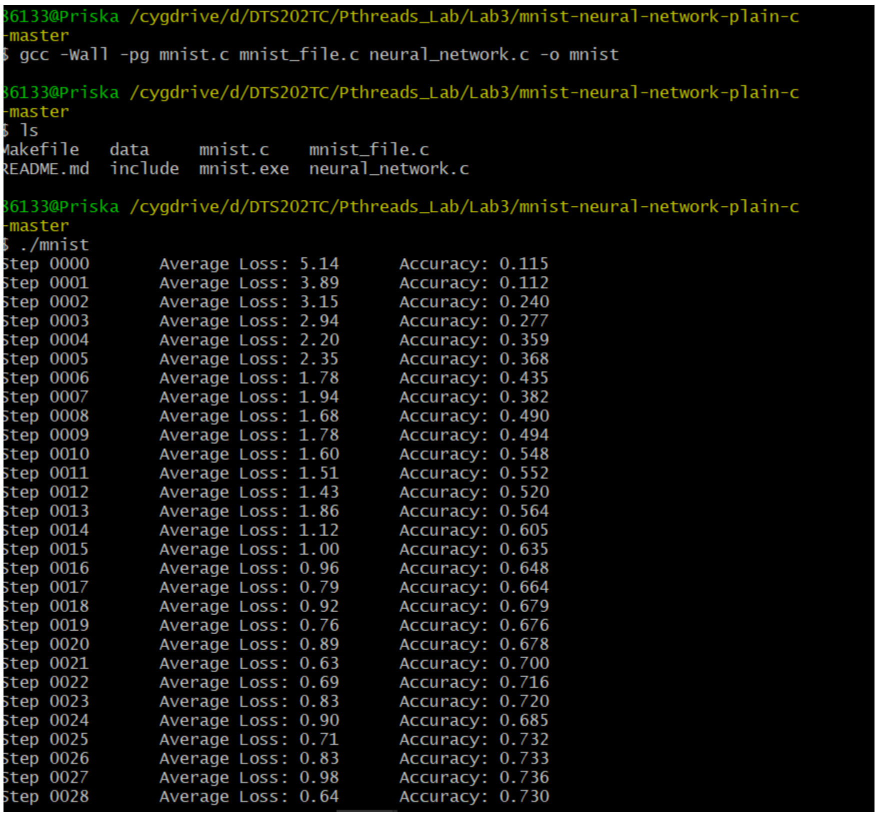
2.1 Profiling

2.1.1 Code

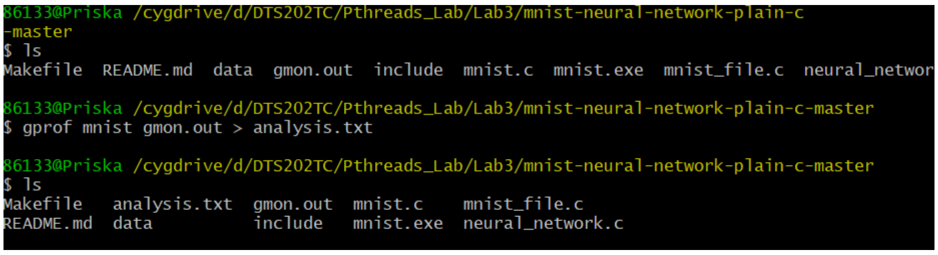
- On Cygwin

```
gcc -Wall -pg mnist.c mnist_file.c neural_network.c -o mnist
./mnist
gprof mnist gmon.out > analysis.txt
```

2.1.2 Screenshot

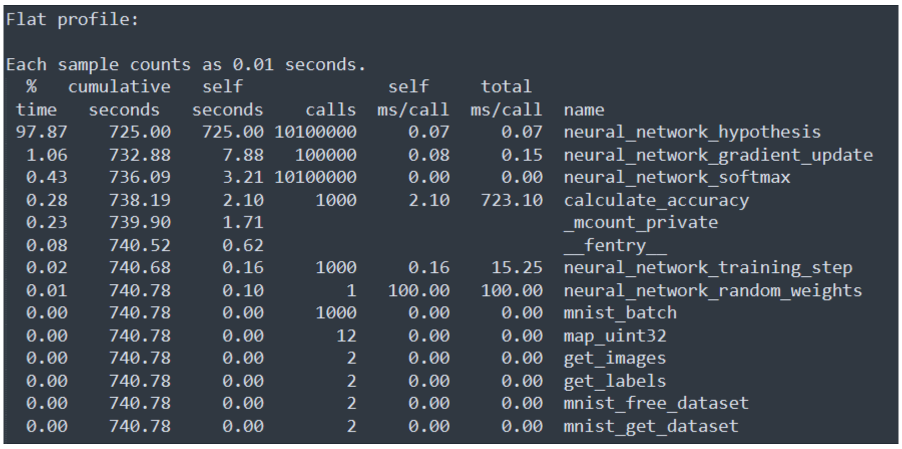


```
86133@Priska /cygdrive/d/DTS202TC/Pthreads_Lab/Lab3/mnist-neural-network-plain-c
-master
$ gcc -Wall -pg mnist.c mnist_file.c neural_network.c -o mnist

86133@Priska /cygdrive/d/DTS202TC/Pthreads_Lab/Lab3/mnist-neural-network-plain-c
-master
$ ls
Makefile   data     mnist.c    mnist_file.c
README.md  include  mnist.exe  neural_network.c

86133@Priska /cygdrive/d/DTS202TC/Pthreads_Lab/Lab3/mnist-neural-network-plain-c
-master
$ ./mnist
Step 0000       Average Loss: 5.14      Accuracy: 0.115
Step 0001       Average Loss: 3.89      Accuracy: 0.112
Step 0002       Average Loss: 3.15      Accuracy: 0.240
Step 0003       Average Loss: 2.94      Accuracy: 0.277
Step 0004       Average Loss: 2.20      Accuracy: 0.359
Step 0005       Average Loss: 2.35      Accuracy: 0.368
Step 0006       Average Loss: 1.78      Accuracy: 0.435
Step 0007       Average Loss: 1.94      Accuracy: 0.382
Step 0008       Average Loss: 1.68      Accuracy: 0.490
Step 0009       Average Loss: 1.78      Accuracy: 0.494
Step 0010       Average Loss: 1.60      Accuracy: 0.548
Step 0011       Average Loss: 1.51      Accuracy: 0.552
Step 0012       Average Loss: 1.43      Accuracy: 0.520
Step 0013       Average Loss: 1.86      Accuracy: 0.564
Step 0014       Average Loss: 1.12      Accuracy: 0.605
Step 0015       Average Loss: 1.00      Accuracy: 0.635
Step 0016       Average Loss: 0.96      Accuracy: 0.648
Step 0017       Average Loss: 0.79      Accuracy: 0.664
Step 0018       Average Loss: 0.92      Accuracy: 0.679
Step 0019       Average Loss: 0.76      Accuracy: 0.676
Step 0020       Average Loss: 0.89      Accuracy: 0.678
Step 0021       Average Loss: 0.63      Accuracy: 0.700
Step 0022       Average Loss: 0.69      Accuracy: 0.716
Step 0023       Average Loss: 0.83      Accuracy: 0.720
Step 0024       Average Loss: 0.90      Accuracy: 0.685
Step 0025       Average Loss: 0.71      Accuracy: 0.732
Step 0026       Average Loss: 0.83      Accuracy: 0.733
Step 0027       Average Loss: 0.98      Accuracy: 0.736
Step 0028       Average Loss: 0.64      Accuracy: 0.730
```

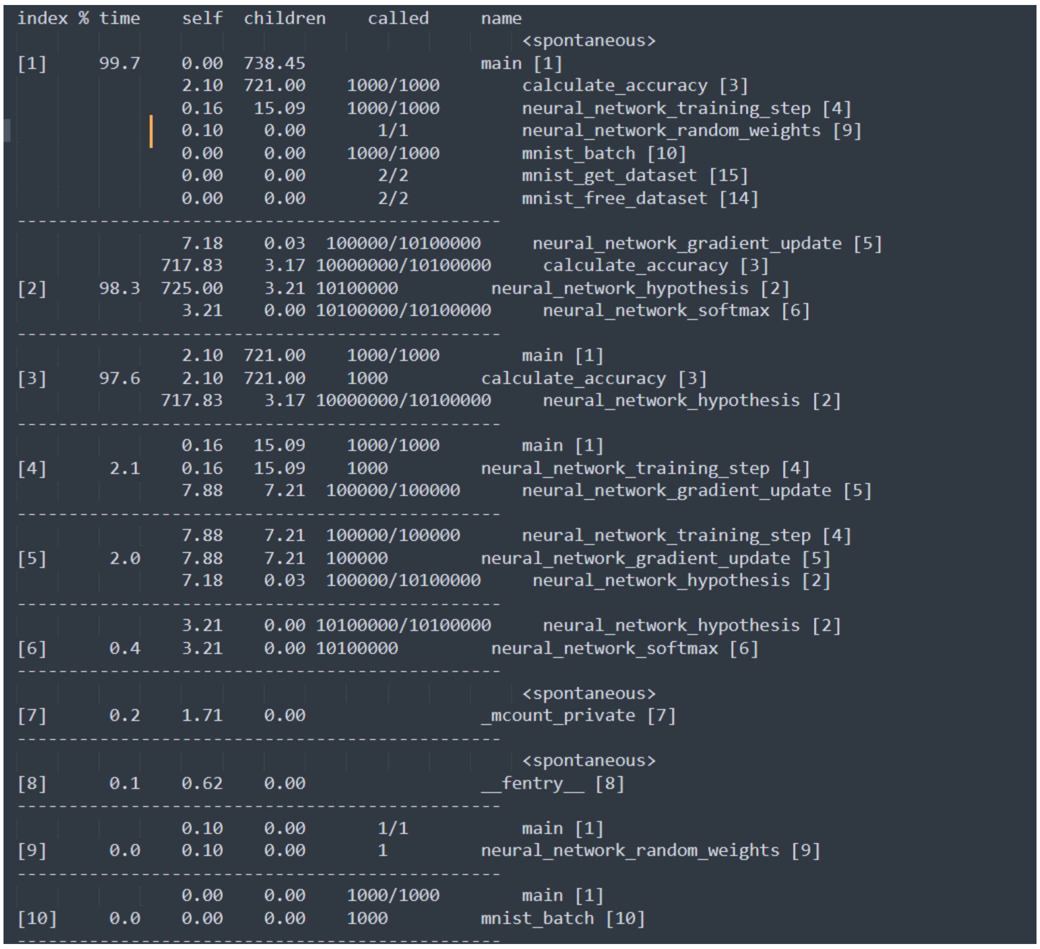


```
86133@Priska /cygdrive/d/DTS202TC/Pthreads_Lab/Lab3/mnist-neural-network-plain-c
-master
$ ls
Makefile  README.md  data  gmon.out  include  mnist.c  mnist.exe  mnist_file.c  neural_networ

86133@Priska /cygdrive/d/DTS202TC/Pthreads_Lab/Lab3/mnist-neural-network-plain-c-master
$ gprof mnist gmon.out > analysis.txt

86133@Priska /cygdrive/d/DTS202TC/Pthreads_Lab/Lab3/mnist-neural-network-plain-c-master
$ ls
Makefile   analysis.txt  gmon.out  mnist.c    mnist_file.c
README.md  data          include   mnist.exe  neural_network.c
```

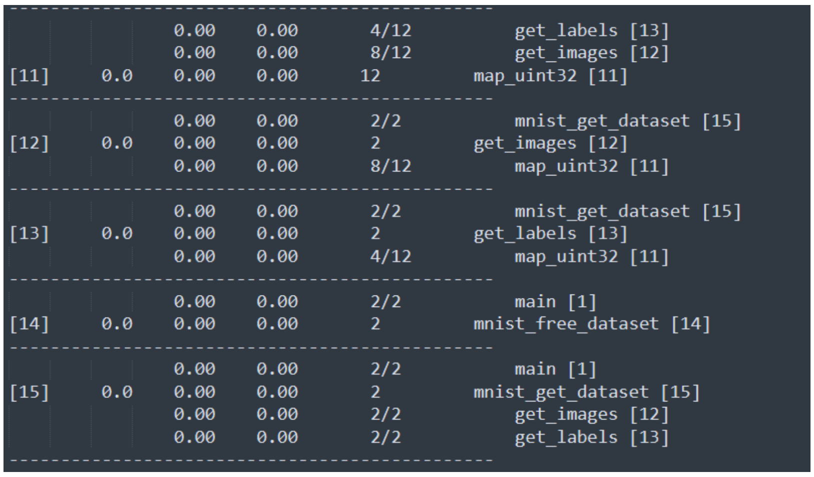
2.1.3 Analysis.txt



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 97.87    725.00   725.00 10100000     0.07     0.07  neural_network_hypothesis
  1.06    732.88     7.88   100000     0.08     0.15  neural_network_gradient_update
  0.43    736.09     3.21 10100000     0.00     0.00  neural_network_softmax
  0.28    738.19     2.10     1000     2.10   723.10  calculate_accuracy
  0.23    739.90     1.71                             _mcount_private
  0.08    740.52     0.62                             __fentry__
  0.02    740.68     0.16     1000     0.16    15.25  neural_network_training_step
  0.01    740.78     0.10        1   100.00   100.00  neural_network_random_weights
  0.00    740.78     0.00     1000     0.00     0.00  mnist_batch
  0.00    740.78     0.00       12     0.00     0.00  map_uint32
  0.00    740.78     0.00        2     0.00     0.00  get_images
  0.00    740.78     0.00        2     0.00     0.00  get_labels
  0.00    740.78     0.00        2     0.00     0.00  mnist_free_dataset
  0.00    740.78     0.00        2     0.00     0.00  mnist_get_dataset
```



```
index % time    self  children    called     name
                                                 <spontaneous>
[1]     99.7    0.00  738.45                 main [1]
                2.10  721.00    1000/1000        calculate_accuracy [3]
                0.16   15.09    1000/1000        neural_network_training_step [4]
                0.10    0.00       1/1           neural_network_random_weights [9]
                0.00    0.00    1000/1000        mnist_batch [10]
                0.00    0.00       2/2           mnist_get_dataset [15]
                0.00    0.00       2/2           mnist_free_dataset [14]
-----------------------------------------------
                7.18    0.03  100000/10100000     neural_network_gradient_update [5]
              717.83    3.17 10000000/10100000     calculate_accuracy [3]
[2]     98.3  725.00    3.21 10100000         neural_network_hypothesis [2]
                3.21    0.00 10100000/10100000     neural_network_softmax [6]
-----------------------------------------------
                2.10  721.00    1000/1000        main [1]
[3]     97.6    2.10  721.00    1000         calculate_accuracy [3]
              717.83    3.17 10000000/10100000     neural_network_hypothesis [2]
-----------------------------------------------
                0.16   15.09    1000/1000        main [1]
[4]      2.1    0.16   15.09    1000         neural_network_training_step [4]
                7.88    7.21  100000/100000      neural_network_gradient_update [5]
-----------------------------------------------
                7.88    7.21  100000/100000      neural_network_training_step [4]
[5]      2.0    7.88    7.21  100000         neural_network_gradient_update [5]
                7.18    0.03  100000/10100000     neural_network_hypothesis [2]
-----------------------------------------------
                3.21    0.00 10100000/10100000     neural_network_hypothesis [2]
[6]      0.4    3.21    0.00 10100000         neural_network_softmax [6]
-----------------------------------------------
                                                 <spontaneous>
[7]      0.2    1.71    0.00                 _mcount_private [7]
-----------------------------------------------
                                                 <spontaneous>
[8]      0.1    0.62    0.00                 __fentry__ [8]
-----------------------------------------------
                0.10    0.00       1/1           main [1]
[9]      0.0    0.10    0.00       1         neural_network_random_weights [9]
-----------------------------------------------
                0.00    0.00    1000/1000        main [1]
[10]     0.0    0.00    0.00    1000         mnist_batch [10]
-----------------------------------------------
```



```
-----------------------------------------------
                0.00    0.00       4/12          get_labels [13]
                0.00    0.00       8/12          get_images [12]
[11]     0.0    0.00    0.00      12         map_uint32 [11]
-----------------------------------------------
                0.00    0.00       2/2           mnist_get_dataset [15]
[12]     0.0    0.00    0.00       2         get_images [12]
                0.00    0.00       8/12          map_uint32 [11]
-----------------------------------------------
                0.00    0.00       2/2           mnist_get_dataset [15]
[13]     0.0    0.00    0.00       2         get_labels [13]
                0.00    0.00       4/12          map_uint32 [11]
-----------------------------------------------
                0.00    0.00       2/2           main [1]
[14]     0.0    0.00    0.00       2         mnist_free_dataset [14]
-----------------------------------------------
                0.00    0.00       2/2           main [1]
[15]     0.0    0.00    0.00       2         mnist_get_dataset [15]
                0.00    0.00       2/2           get_images [12]
                0.00    0.00       2/2           get_labels [13]
-----------------------------------------------
```

The profiling analysis of mnist-neural-network-plain-c-master is carried out by using GPROF tool. As can be seen from the above figure, `neural_network_hypothesis` takes it as the bottleneck of this program as the function with the longest proportion. Later improvements and upgrades take this function as the key object of study.

2.2 Pthreads Hello World

2.2.1 Code

- On Cygwin

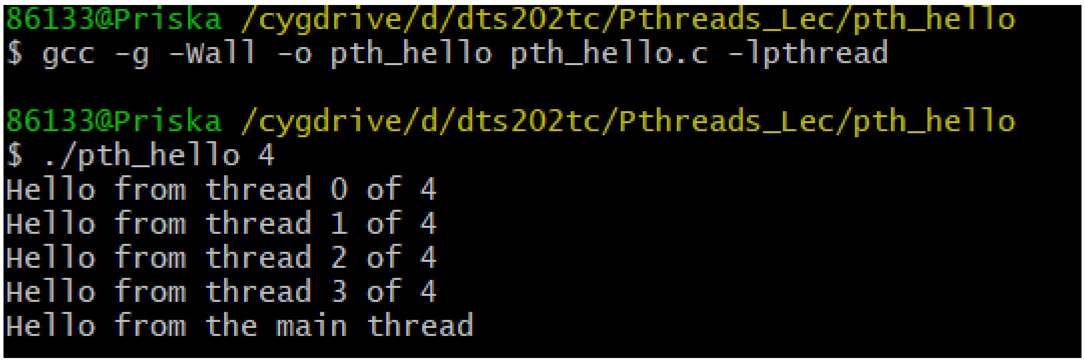
```
gcc -g -Wall -o pth_hello pth_hello.c -lpthread
./pth_hello 4
```

- pth_hello.c

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

/*Global variable: accessible to all threads*/
int thread_count;
void* Hello(void* rank);

int main(int argc,char* argv[]) {
    long thread;    /*Using long in case of a 64-bit system*/
    pthread_t* thread_handles;

    thread_count = strtol(argv[1],NULL,10);

    thread_handles = malloc(thread_count*sizeof(pthread_t));

    for (thread = 0; thread < thread_count; thread++) {
        pthread_create(&thread_handles[thread],NULL,
                       Hello,(void*)thread);
    }

    printf("Hello from the main thread\n");

    for (thread = 0; thread < thread_count; thread++) {
        pthread_join(thread_handles[thread],NULL);
    }

    free(thread_handles);
    return 0;
```

```c
}

void* Hello(void* rank){
    long my_rank = (long) rank;

    printf("Hello from thread %ld of %d\n",
           my_rank,thread_count);

    return NULL;
}
```

2.2.2 Screenshot



```
86133@Priska /cygdrive/d/dts202tc/Pthreads_Lec/pth_hello
$ gcc -g -Wall -o pth_hello pth_hello.c -lpthread

86133@Priska /cygdrive/d/dts202tc/Pthreads_Lec/pth_hello
$ ./pth_hello 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from the main thread
```

2.3 Estimate π with Pthreads

The program originally used `random()` for random numbers in the `Threadsum` function, but this function does not improve performance through threads, and should use the global state shared between threads, there is no way to guarantee thread safe. Finally, the program chooses `rand_r()` function to improve performance.

2.3.1 Code

- shell

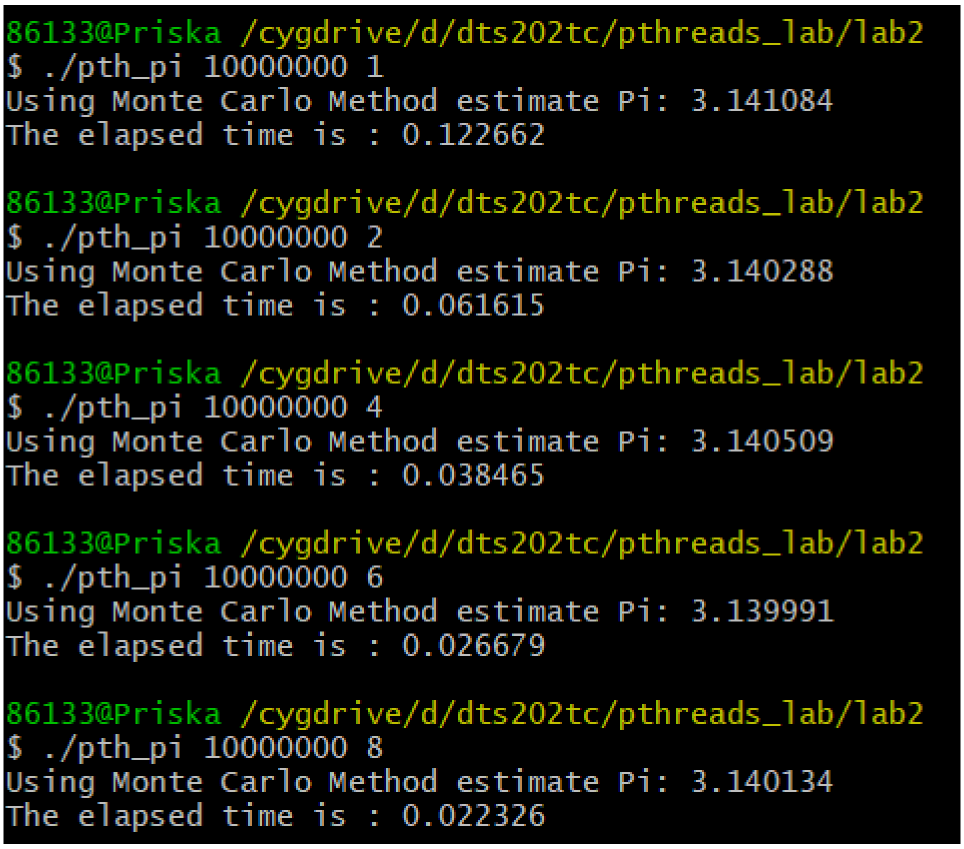
```
gcc -g -Wall -o pth_pi pth_pi.c -lm -lpthread
./pth_pi 10000000 1
./pth_pi 10000000 2
./pth_pi 10000000 4
./pth_pi 10000000 6
./pth_pi 10000000 8
./pth_pi 10000000 10
./pth_pi 10000000 12
./pth_pi 10000000 14
./pth_pi 10000000 16
```

- `pth_pi.c`

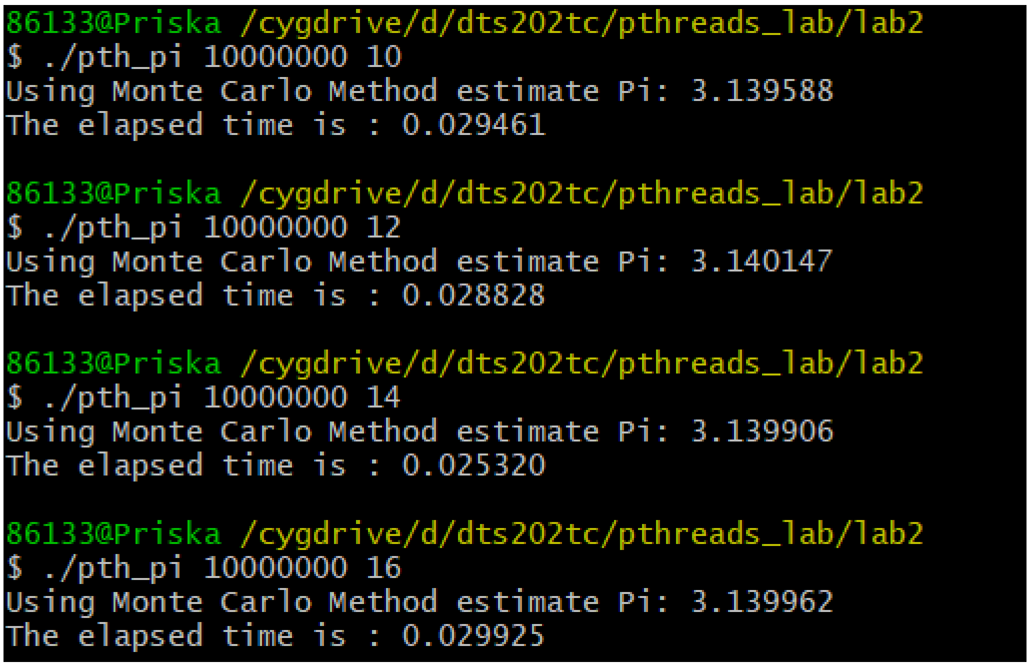
```
/*File: pth_pi.c
 * Purpose: Estimate pi using Monte Carlo Method
 *      (number in circle)/(total number of tosses) = pi/4
 *      This version used a mutex to protect the critical section
 *
 * Compile:      gcc -g -Wall -o pth_pi pth_pi.c -lm -lpthread
 * Run:          ./pth_pi <total_points><number of threads>
 *               total_points is the randomly toss total
 *               should be evenly divisible by the number of threads
 *
 *Input:    None
 *Output: The estimate of pi using Monte Carlo on multiple threads,
 *         Also elapsed times for the multithreaded and
singlethreaded
 *         computations
*/

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#ifndef _TIMER_H_
#define _TIMER_H_
#include <sys/time.h>
#include <stdlib.h>
/* The argument now should be a double (not a pointer to a
double) */
#define GET_TIME(now) { \
   struct timeval t; \
   gettimeofday(&t, NULL); \
   now = t.tv_sec + t.tv_usec/1000000.0; \
}
#endif

/*Global variables: accessible to all threads*/
const int MAX_THREADS = 1024; /*Threads maximum*/
long long in_circle = 0;      /*Each thread has in_circle
points*/
long long points_per_thread;  /*Each thread count points
number*/
long long total_points;       /*total points from user input*/
long long thread_count;       /*total threads from user input*/
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;/*Initialize
mutex*/

/*Only executed by main thread*/
void* Thread_sum();
void Get_args(int argc, char*argv[]);
void Usage(char* prog_name);

int main(int argc,char* argv[])
{
    double start,finish;
    /*Get number of threads from command line*/
```

```c
    Get_args(argc,argv);        /*Inputs*/
    points_per_thread = total_points / thread_count;

    /*Set random seed*/
    srand((unsigned)time(NULL));
    pthread_t *threads = malloc(thread_count * sizeof(pthread_t));

    /*Initialize the properties of thread*/
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    GET_TIME(start);
    int i;
    for (i = 0; i < thread_count; i++) {
        pthread_create(&threads[i], &attr, Thread_sum, (void *) NULL);
    }

    for (i = 0; i < thread_count; i++) {
        pthread_join(threads[i], NULL);
    }
    GET_TIME(finish);
    /*free mutex and threads*/
    pthread_mutex_destroy(&mutex);
    free(threads);

    printf("Using Monte Carlo Method estimate Pi: %f\n", (4. * (double)in_circle) / ((double)points_per_thread * thread_count));
    printf("The elapsed time is : %f\n",finish-start);

    return 0;
}

/*------------------------------------------------------------------
 * Function:    Get_args
 * Purpose:     Get the command line args
 * In args:     argc, argv
 * Globals out: total_points, thread_count
 */
void Get_args(int argc, char* argv[]) {
    if (argc != 3) Usage(argv[0]);
    total_points = strtol(argv[1], NULL, 10);
    if (total_points <= 0) Usage(argv[0]);
    thread_count = strtoll(argv[2], NULL, 10);
    if (thread_count <= 0 || thread_count > MAX_THREADS) 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 <total points> <threads>\n",prog_name);
    fprintf(stderr,"total points should be evenly divisible by the number of threads\n");
    fprintf(stderr,"threads is the number of terms and should be >= 1\n");
    exit(1);
}  /* Usage */

/*------------------------------------------------------------------
 * Function:  Thread_sum
 * Purpose:   Accumulate qualified random numbers
 * In arg:    None
 * Globals out: in_circle
 */
void *Thread_sum() {
    long in_circle_thread = 0;
    unsigned int rand_state = rand();
    long i;
    for (i = 0; i < points_per_thread; i++) {
        /*random() function does not improve performance
         * therefore chooses rand_f function
         * */
        double x = rand_r(&rand_state) / ((double)RAND_MAX + 1) * 2.0 - 1.0;
        double y = rand_r(&rand_state) / ((double)RAND_MAX + 1) * 2.0 - 1.0;
        if (x * x + y * y < 1) {
            in_circle_thread++;
        }
    }

    pthread_mutex_lock(&mutex);
    in_circle += in_circle_thread;
    pthread_mutex_unlock(&mutex);

    return NULL;
}/*Thread_sum*/
```

2.3.2 Screenshot



```
86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 10000000 1
Using Monte Carlo Method estimate Pi: 3.141084
The elapsed time is : 0.122662

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 10000000 2
Using Monte Carlo Method estimate Pi: 3.140288
The elapsed time is : 0.061615

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 10000000 4
Using Monte Carlo Method estimate Pi: 3.140509
The elapsed time is : 0.038465

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 10000000 6
Using Monte Carlo Method estimate Pi: 3.139991
The elapsed time is : 0.026679

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 10000000 8
Using Monte Carlo Method estimate Pi: 3.140134
The elapsed time is : 0.022326
```



```
86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 10000000 10
Using Monte Carlo Method estimate Pi: 3.139588
The elapsed time is : 0.029461

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 10000000 12
Using Monte Carlo Method estimate Pi: 3.140147
The elapsed time is : 0.028828

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 10000000 14
Using Monte Carlo Method estimate Pi: 3.139906
The elapsed time is : 0.025320

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 10000000 16
Using Monte Carlo Method estimate Pi: 3.139962
The elapsed time is : 0.029925
```

III. Lab 4 (35 Marks)

3.1 OpenMP Hello World

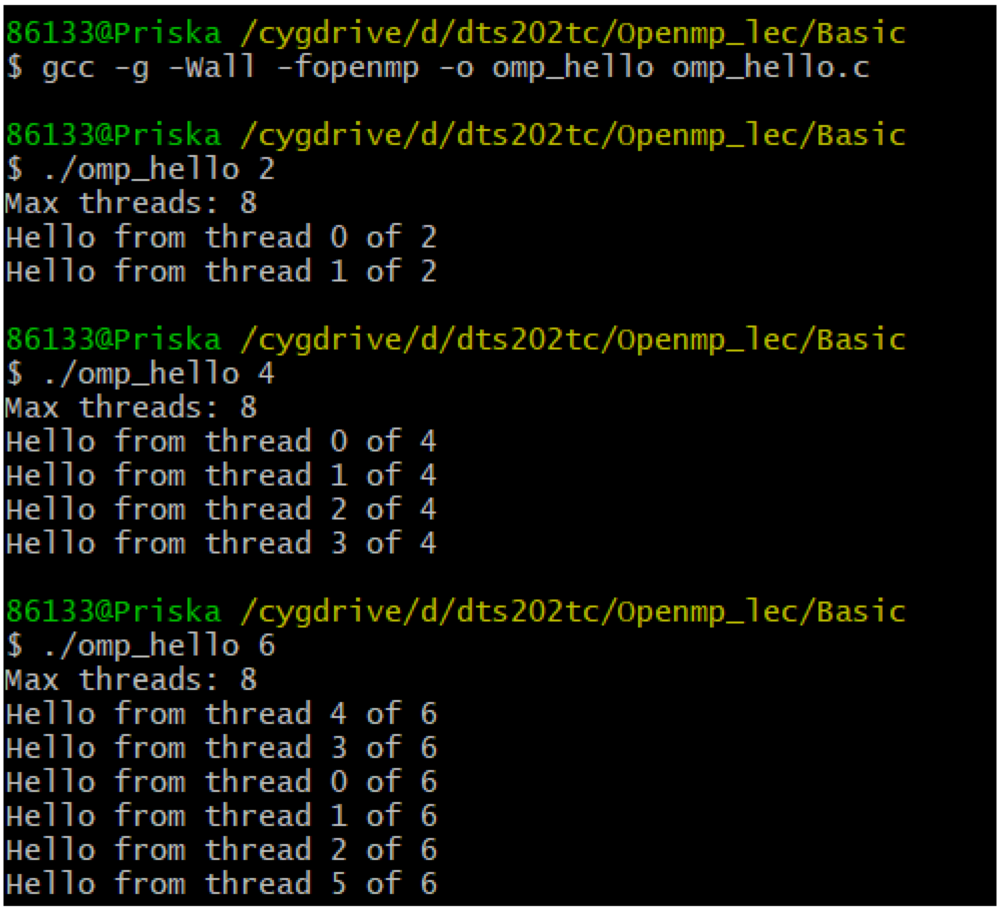
3.1.1 Code

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
./omp_hello 2
./omp_hello 4
./omp_hello 6
```

`omp_hello.c`

```
#include<stdlib.h>
#include<stdio.h>
#include<omp.h>

void Hello(void);//Threads function

int main(int argc,char* argv[]){
    /*Get number of threads from command line*/
    int thread_count = strtol(argv[1],NULL,10);
    printf("Max threads: %d \n",omp_get_num_procs());

#   pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
}

void Hello(void){
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n",my_rank,thread_count);
}
```

3.1.2 Screenshot



```
86133@Priska /cygdrive/d/dts202tc/Openmp_lec/Basic
$ gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

86133@Priska /cygdrive/d/dts202tc/Openmp_lec/Basic
$ ./omp_hello 2
Max threads: 8
Hello from thread 0 of 2
Hello from thread 1 of 2

86133@Priska /cygdrive/d/dts202tc/Openmp_lec/Basic
$ ./omp_hello 4
Max threads: 8
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

86133@Priska /cygdrive/d/dts202tc/Openmp_lec/Basic
$ ./omp_hello 6
Max threads: 8
Hello from thread 4 of 6
Hello from thread 3 of 6
Hello from thread 0 of 6
Hello from thread 1 of 6
Hello from thread 2 of 6
Hello from thread 5 of 6
```

It is observed that there is competition between threads to access standard output, so the output does not appear in thread numbering order.

3.2 Estimate π with OpenMP

3.2.1 Code

```
gcc -g -Wall -fopenmp -o omp_pi omp_pi.c
./omp_pi 10000000 1
./omp_pi 10000000 2
./omp_pi 10000000 4
./omp_pi 10000000 6
./omp_pi 10000000 8
./omp_pi 10000000 10
./omp_pi 10000000 12
./omp_pi 10000000 14
./omp_pi 10000000 16
```

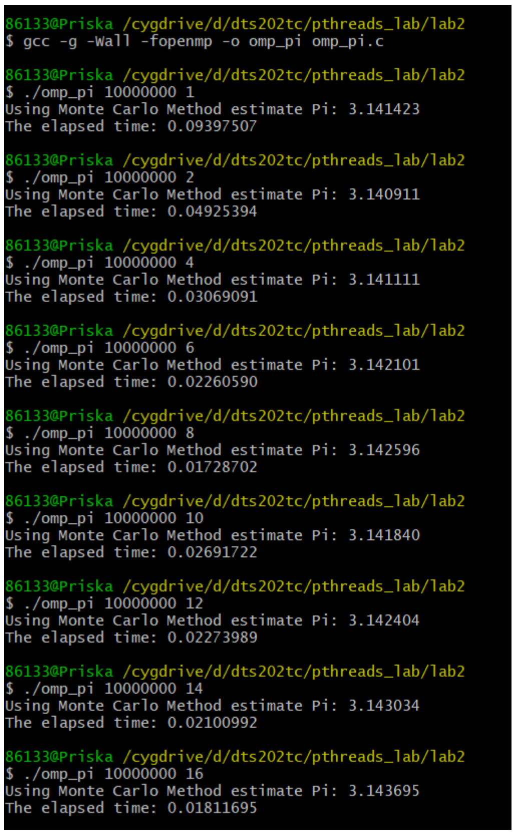
`omp_pi.c`

```
/*File: omp_pi.c
 * Purpose: Estimate pi using Monte Carlo Method
 *      (number in circle)/(total number of tosses) = pi/4
 *
 * Compile:       gcc -g -Wall -fopenmp -o omp_pi omp_pi.c

 * Run:          ./omp_pi <total_points><number of threads>
 *               total_points is the randomly toss total
 *               should be evenly divisible by the number of threads
 *
 *Input:    None
 *Output: The estimate of pi using Monte Carlo on multiple threads,
 *          Also elapsed times for the multithreaded and singlethreaded
 *          computations
*/

#include<stdlib.h>
#include<stdio.h>
#include<time.h>
#include<omp.h>
```

```c

#ifndef _TIMER_H_
#define _TIMER_H_
#include <sys/time.h>
#include <stdlib.h>
/* The argument now should be a double (not a pointer to a
double) */
#define GET_TIME(now) { \
   struct timeval t; \
   gettimeofday(&t, NULL); \
   now = t.tv_sec + t.tv_usec/1000000.0; \
}
#endif

const int MAX_THREADS = 1024;
void Usage(char prog_name);

int main(int argc,char** argv){
    double start,finish;
    long long in_circle=0;
    long long total_points;
    int thread_count;

    /*Get number of threads from command line*/
    if(argc != 3) Usage(argv[0]);
    total_points = strtol(argv[1],NULL,10);
    if(total_points <=0 ) Usage(argv[0]);
    thread_count=strtol(argv[2],NULL,10);
    if(thread_count<=0 || thread_count > MAX_THREADS)
Usage(argv[0]);

    /*Set random seed*/
    srand(time(NULL));

    /*thread compute local circle points*/
    double x,y,distance_point;
    long long int i;

    GET_TIME(start);
#pragma omp parallel for num_threads(thread_count) default(none)
\
        reduction(+:in_circle) shared(total_points)
private(i,x,y,distance_point)
```

```c
    for( i=0;i<total_points;i++){
        x=(double)rand()/(double)RAND_MAX;
        y=(double)rand()/(double)RAND_MAX;
        distance_point=x*x+y*y;
        if(distance_point<=1){
            in_circle++;
        }
    }
    double estimate_pi=(double)in_circle/total_points*4;
    GET_TIME(finish);

    printf("Using Monte Carlo Method estimate Pi: %lf\n",estimate_pi);
    printf("The elapsed time: %.8f\n",finish-start);
    return 0;
}


/*-----------------------------------------------------------------
 * 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 <total points> <threads>\n",prog_name);
    fprintf(stderr,"total points should be evenly divisible by the number of threads\n");
    fprintf(stderr,"threads is the number of terms and should be >= 1\n");
    exit(1);
}  /* Usage */
```

3.2.2 Screenshot



```
86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ gcc -g -Wall -fopenmp -o omp_pi omp_pi.c

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 10000000 1
Using Monte Carlo Method estimate Pi: 3.141423
The elapsed time: 0.09397507

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 10000000 2
Using Monte Carlo Method estimate Pi: 3.140911
The elapsed time: 0.04925394

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 10000000 4
Using Monte Carlo Method estimate Pi: 3.141111
The elapsed time: 0.03069091

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 10000000 6
Using Monte Carlo Method estimate Pi: 3.142101
The elapsed time: 0.02260590

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 10000000 8
Using Monte Carlo Method estimate Pi: 3.142596
The elapsed time: 0.01728702

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 10000000 10
Using Monte Carlo Method estimate Pi: 3.141840
The elapsed time: 0.02691722

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 10000000 12
Using Monte Carlo Method estimate Pi: 3.142404
The elapsed time: 0.02273989

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 10000000 14
Using Monte Carlo Method estimate Pi: 3.143034
The elapsed time: 0.02100992

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 10000000 16
Using Monte Carlo Method estimate Pi: 3.143695
The elapsed time: 0.01811695
```

3.3 Comparison

3.3.1 Comparison under different threads

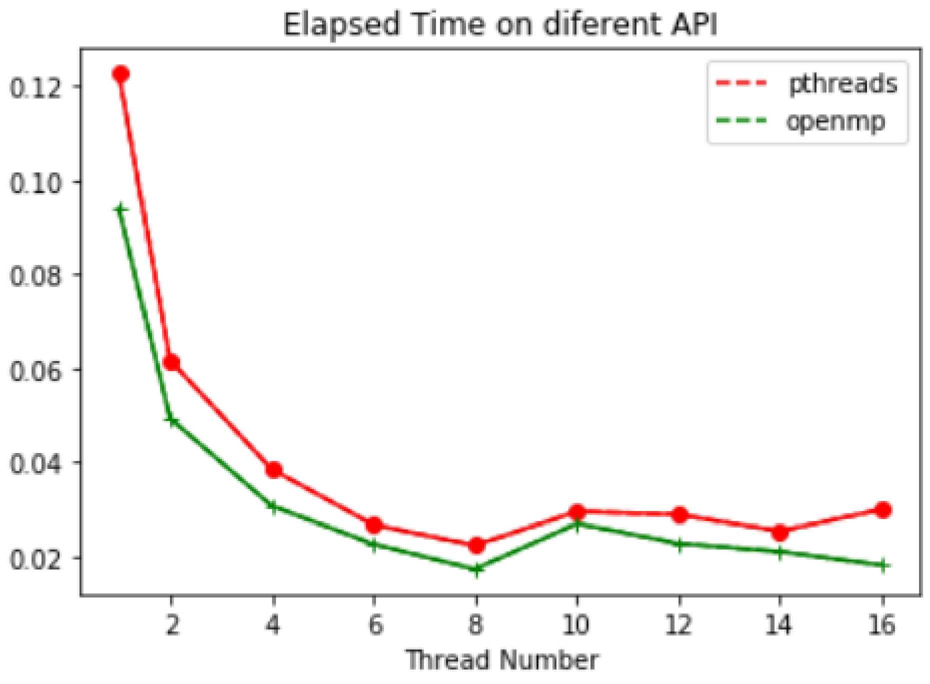
3.3.1.1 Elapsed Time

The following is a table of time under different threads using two different API, Pthreads and OpenMP

	1 THREAD	2 THREADS	4 THREADS	6 THREADS	8 THREADS
Pthreads	0.1226 s	0.06161 s	0.03846 s	0.02667 s	0.02232 s
OpenMP	0.09397 s	0.04925 s	0.03069 s	0.02260 s	0.01728 s

	10 THREADS	12 THREADS	14 THREADS	16 THREADS
Pthreads	0.02946 s	0.02882 s	0.02532 s	0.02992 s
OpenMP	0.02691 s	0.02273 s	0.02100 s	0.01811 s

The following figure compares the time consumption of `Pthreads` and `OpenMP` through a line chart.

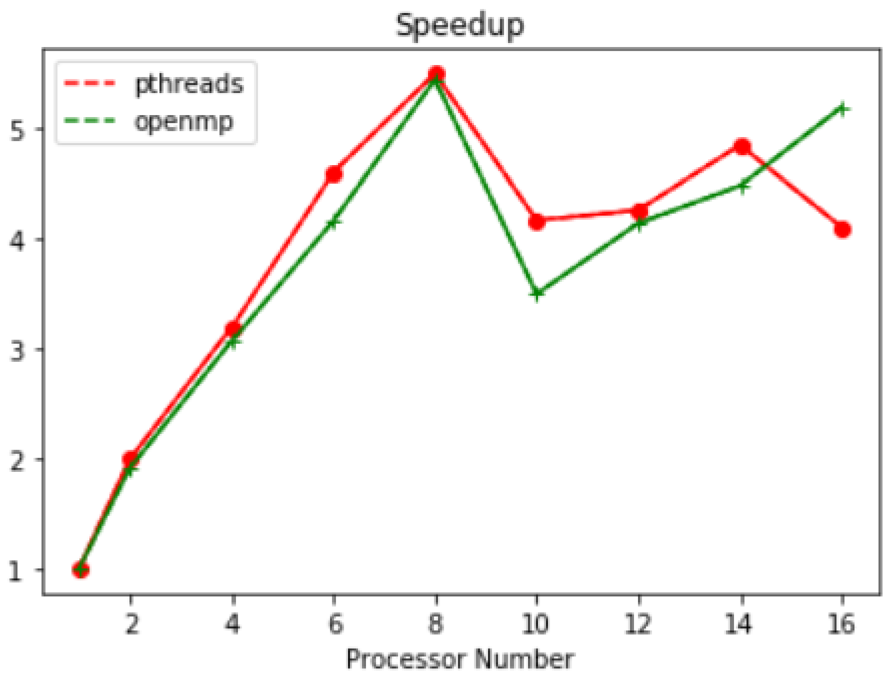


The author finds that when using `OpenMP`, there is less time for each process than when using `Pthreads`. Next, look at speedup and efficiency.

3.3.1.2 Speedup

	1 THREAD	2 THREADS	4 THREADS	6 THREADS	8 THREADS
Pthreads	1	1.9899	3.1877	4.5969	5.4928
OpenMP	1	1.9080	3.0619	4.1579	5.4380

	10 THREADS	12 THREADS	14 THREADS	16 THREADS
Pthreads	4.1615	4.2539	4.8420	4.0975
OpenMP	3.4920	4.1341	4.4747	5.1888



When running parallel programs with 1-8 threads, the speedup of the two is almost the same, and it is found that before 8 threads, speedup is often close to linear speedup.

In addition, there are some tasks that cannot be parallelized in the program. According to `Amdahl's Law`, there are parts with a ratio of r in serial programs that cannot be parallelized, and the speedup that can be achieved is close to $\frac{1}{r}$ [1]

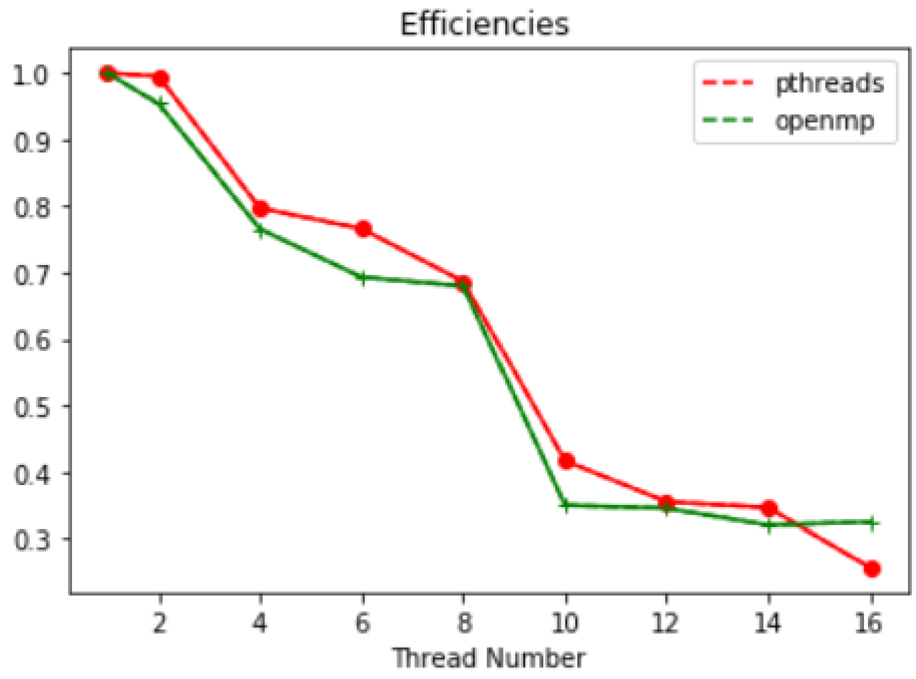
Both Pthreads and openmp reach the maximum speedup of 5.5 at 8 threads, so the non-parallelization task of both is 18%.

When more than 8 threads run a parallelized program, author will find that the speedup of the program decreases or fluctuates in an area. This phenomenon is that when the number of threads increases, the busy wating time between threads will increase, resulting in a significant increase in overhead time, and the speedup will fluctuate.

3.3.1.3 Efficiency

	1 THREAD	2 THREADS	4 THREADS	6 THREADS	8 THREADS
Pthreads	1	0.9949	0.7969	0.7661	0.6866
OpenMP	1	0.9540	0.7654	0.6797	0.6797

	10 THREADS	12 THREADS	14 THREADS	16 THREADS
Pthreads	0.4161	0.3544	0.3458	0.2560
OpenMP	0.3492	0.3445	0.3196	0.3243

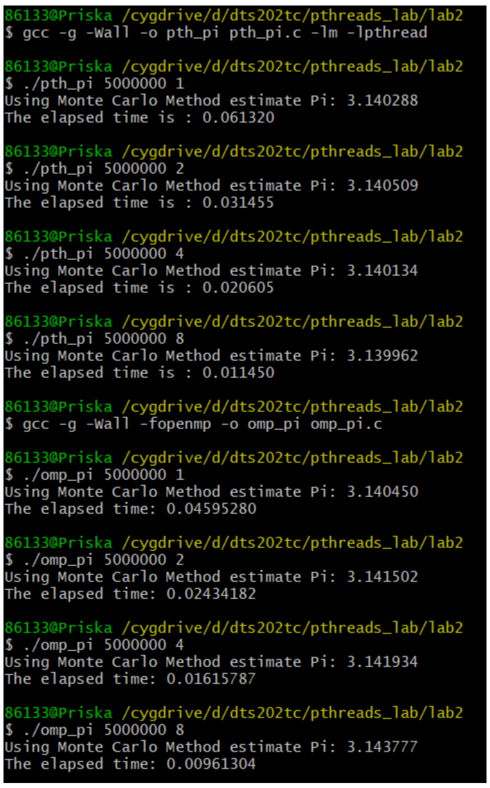


$E = \frac{speedup}{p}$, p is pthreads number;

As the number of threads increases, the efficiency decreases gradually. This is because the more threads, the higher the percentage of overhead time.

3.3.2 Comparison under different problem size

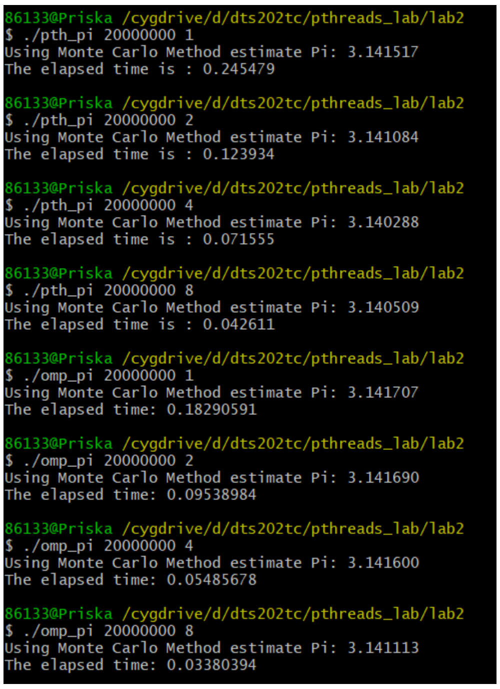
When problem size = `5000000`



```
86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ gcc -g -Wall -o pth_pi pth_pi.c -lm -lpthread

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 5000000 1
Using Monte Carlo Method estimate Pi: 3.140288
The elapsed time is : 0.061320

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 5000000 2
Using Monte Carlo Method estimate Pi: 3.140509
The elapsed time is : 0.031455

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 5000000 4
Using Monte Carlo Method estimate Pi: 3.140134
The elapsed time is : 0.020605

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 5000000 8
Using Monte Carlo Method estimate Pi: 3.139962
The elapsed time is : 0.011450

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ gcc -g -Wall -fopenmp -o omp_pi omp_pi.c

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 5000000 1
Using Monte Carlo Method estimate Pi: 3.140450
The elapsed time: 0.04595280

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 5000000 2
Using Monte Carlo Method estimate Pi: 3.141502
The elapsed time: 0.02434182

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 5000000 4
Using Monte Carlo Method estimate Pi: 3.141934
The elapsed time: 0.01615787

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 5000000 8
Using Monte Carlo Method estimate Pi: 3.143777
The elapsed time: 0.00961304
```

	1 THREAD	2 THREADS	4 THREADS	8 THREADS
Pthread	0.06132	0.031455	0.02060	0.01145
OpenMP	0.04595	0.02434	0.01615	0.00961

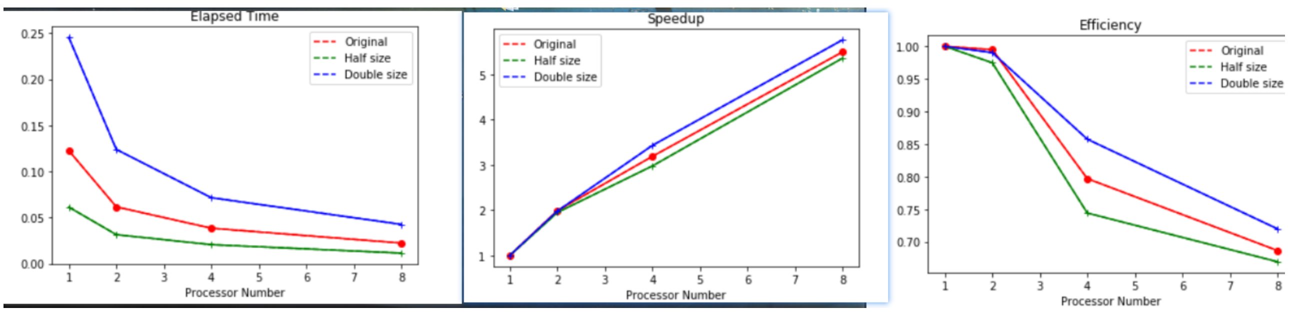
When problem size = `20000000`



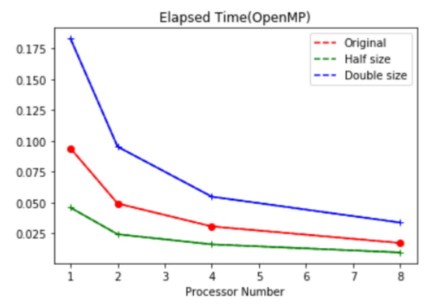
```
86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 20000000 1
Using Monte Carlo Method estimate Pi: 3.141517
The elapsed time is : 0.245479

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 20000000 2
Using Monte Carlo Method estimate Pi: 3.141084
The elapsed time is : 0.123934

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 20000000 4
Using Monte Carlo Method estimate Pi: 3.140288
The elapsed time is : 0.071555

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./pth_pi 20000000 8
Using Monte Carlo Method estimate Pi: 3.140509
The elapsed time is : 0.042611

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 20000000 1
Using Monte Carlo Method estimate Pi: 3.141707
The elapsed time: 0.18290591

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 20000000 2
Using Monte Carlo Method estimate Pi: 3.141690
The elapsed time: 0.09538984

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 20000000 4
Using Monte Carlo Method estimate Pi: 3.141600
The elapsed time: 0.05485678

86133@Priska /cygdrive/d/dts202tc/pthreads_lab/lab2
$ ./omp_pi 20000000 8
Using Monte Carlo Method estimate Pi: 3.141113
The elapsed time: 0.03380394
```

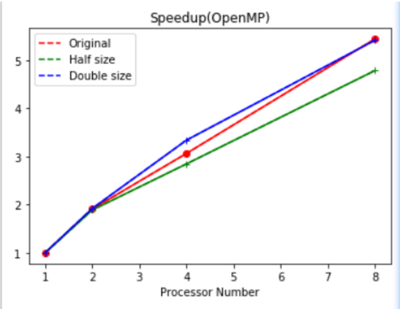
	1 THREAD	2 THREADS	4 THREADS	8 THREADS
Pthread	0.2454	0.1239	0.07155	0.04261
OpenMP	0.1829	0.0953	0.0548	0.03380

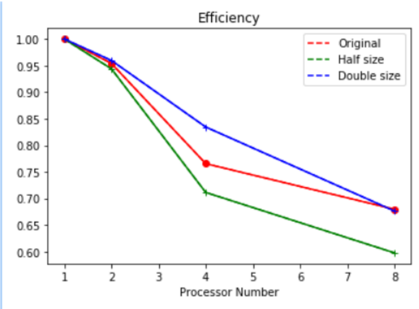
Pthread:



OpenMP:







The values of Elapsed time, Speedup and Efficiency depend not only on the number of threads, but also on the scale of the problem. When the scale of the problem grows, speedup and efficiency increase at the same time.

According to:

$$T_{Parallel} = \frac{T_{Serial}}{p} + T_{overhead}$$

As the scale of the problem increases, threads have more tasks to perform, and the relative time for coordinating work between threads is reduced. overhead time grows more slowly than serial time , consequently both Speedup and efficiency will increase.

And according to Gustafson's Law, as the scale of the problem increases, the proportion of the nonparallelizable part of the program also decreases [2].

From half the size to the original problem size and then to double the size, the maximum speedup is increasing, so according to the formula, the non-parallelization part is reduced.

- Reference List

[1] Krishnaprasad, S., 2001. Uses and abuses of Amdahl's law. Journal of Computing Sciences in colleges, 17(2), pp.288-293.

[2] McCool, Michael D.; Robison, Arch D.; Reinders, James (2012). "2.5 Performance Theory". Structured Parallel Programming: Patterns for Efficient Computation. Elsevier. pp. 61–62. ISBN 978-0-12-415993-8