Monte Carlo method by Multithreading

Aaryan, CO21BTECH11001

Contents of the zip file:

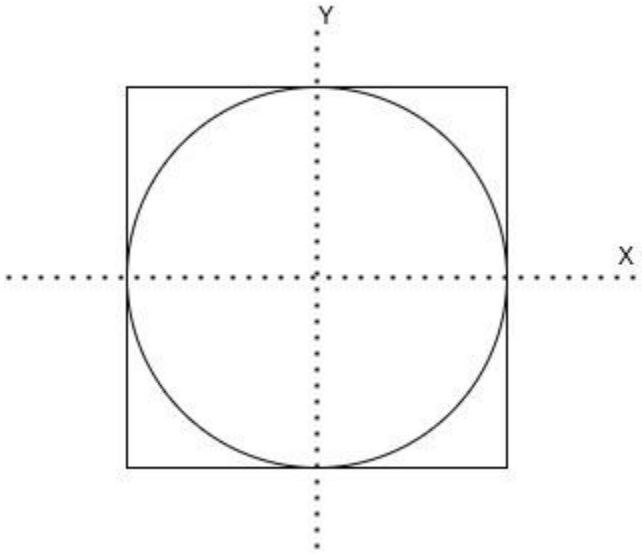
Input file: in.txt contains the inputs to be given to the program.
Report: Assign1_Report_CO21BTECH11001.pdf
Source code: Assign1_Src_CO21BTECH11001.cpp
Readme file: Assign1_README_CO21BTECH11001.txt

Note: Multithreading is done using pthread.

Working of code:

It can be described as follows:

1. A unit circle is centered at the origin, circumscribed by a unit square. If we generate random coordinates (x, y) such that it lies within the square, then value of π can be approximated by following formula:
   $\pi = 4\times (number\ of\ points\ in\ circle) / (total\ number\ of\ points)$



2. In main function, input is taken from input file using fscanf.
   Let number of points to be generated = n
   Let number of threads to be created = k

3. A variable *total_points_inside* is initialized to 0. This variable will keep track of the number of random points found inside the circle.
   A pointer to this variable will be passed as an argument to each thread. Every thread will update its value while execution.

4. An array *arguments* is created by using a struct *thread_args,* to pass arguments to the thread.
   i^{th} element of the array *arguments* is passed as an argument to the i^{th} thread.
   The struct *thread_args* has the following contents:
   a. *num_points*: It is the number of random points to be generated by the thread.
   b. *num_inside*: Out of all numbers generated by the thread, it is the number of points inside the circle. It is initialized to 0 while passing as an argument.
   c. *total_points_inside*: It is a pointer to an integer that will keep track of the number of random points found inside the circle.
   d. *arr*: An array, which will store some data of the points generated by the thread. A struct *data* is created to store this data.
      This includes the following contents:
      i. *x*: The x coordinate of the point.
      ii. *y*: The y coordinate of the point.
      iii. *inside*: A boolean variable. If the point lies inside the circle, it will be set to *true* by the thread. Otherwise, it will be set to *false* by the thread.

5. Total number of points to be generated are distributed to threads. Distribution is as follows:
   1st thread: (n / k) points

   2nd thread: (n / k) points
   .
   .
   .

   (k-1)th thread: (n / k) points

   (k)th thread: Remaining number of points i.e., (n - (k-1)*(n / k)) points

   Size of array *arr* is set to the number of points to be generated by the thread.

6. The thread will generate two floating point random numbers x and y in the range $[-1, 1]$ and store them in the array *arr*.
   It is done using $rand()$ function.
   Then, it is checked if $x^2 + y^2 < 1$.

   If yes, then *inside* is set to *true* and *num_inside* is incremented.
   Else, *inside* is set to *false*.

   After *num_points* number of points are generated and processed, *total_points_inside* will be dereferenced and incremented by *num_inside.*

7. After all threads are executed, the main thread will calculate the approximated value of π by
   $\pi = 4.0 * (total\_points\_inside)/(n)$

   Time taken is calculated using $clock()$ function in the library *time.h*
   Now, the main thread will print the time taken and value calculated in *output.txt* file.

8. Now the main thread will print the log in the output file.
   It will first print the thread number along with the number of points inside the square and the number of points inside the circle in the following way:

   Thread1: Number of points inside the square, Number of points inside the circle
   Points inside the square: point1, point2 .....
   Points inside the circle: point1, point2 .....

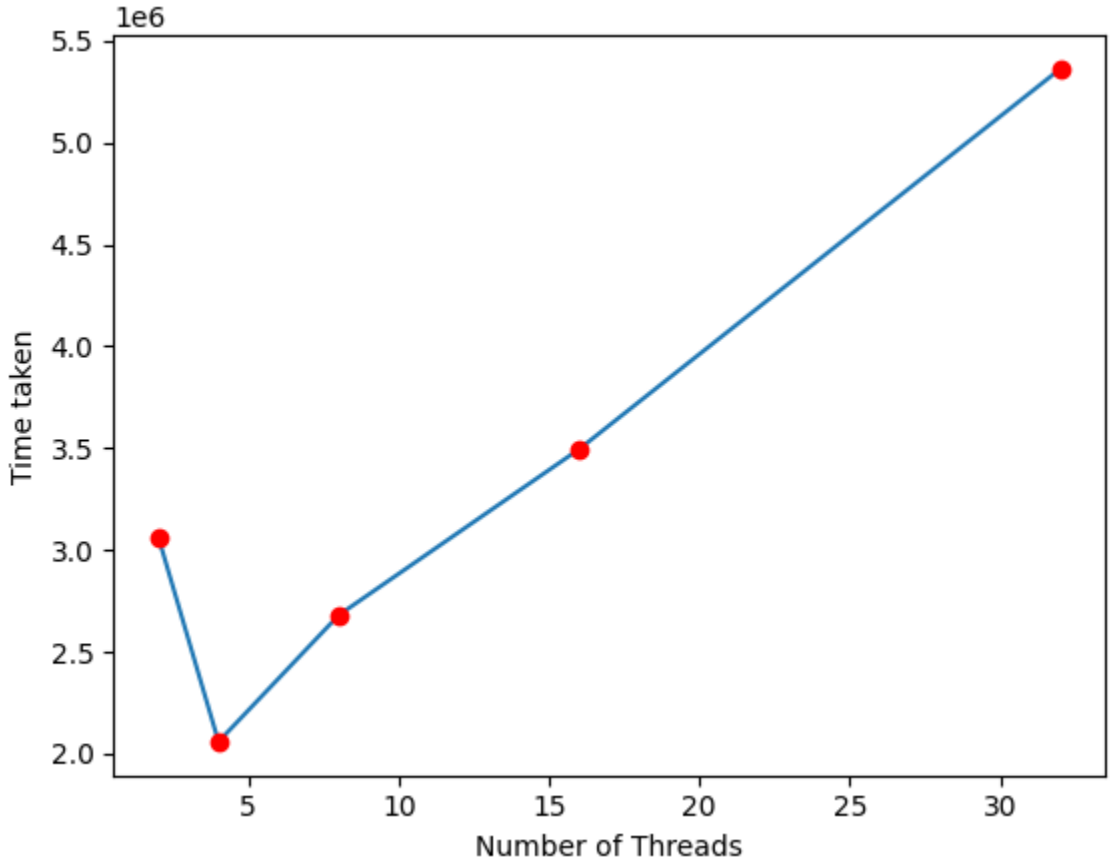
   Thread2: Number of points inside the square, Number of points inside the circle
   Points inside the square: point1, point2 .....
   Points inside the circle: point1, point2 .....
   .
   .
   .

   $Number\ of\ points\ inside\ the\ circle\ = num\_inside$
   $Number\ of\ points\ inside\ the\ square\ = num\_points\ - num\_inside$

   The main thread will traverse the array *arguments* two times.
   In the first traversal, it will print (in the output file) all points inside array *arr,* which are inside the square and outside the circle i.e, the field *inside* is *false.*
   In the second traversal, it will print (in the output file) all points inside array *arr,* which are inside the circle i.e., the field *inside* is *true.*

Results :

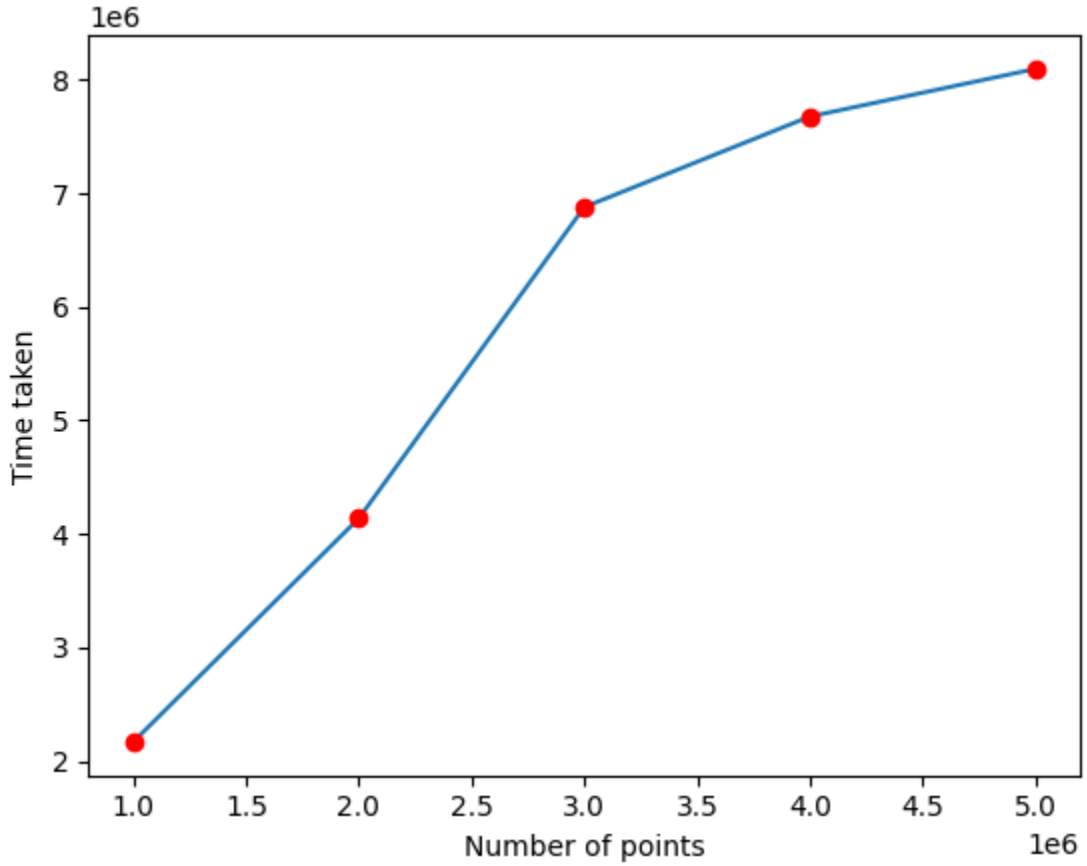
1. When the number of points is kept constant and the number of threads is increased gradually, the time taken to execute first decreases and then increases.



The above plot is for time taken vs number of threads when the number of points is kept constant at $6 * 10^6$.

The reason for this behaviour is that the operating system can efficiently manage only a limited number of parallel threads. So, when number of threads is increased from 2 to 4, the time of execution decreases.
But the management of a high number of threads can not be done efficiently and the time taken to communicate between threads overrides the efficiency due to parallel computations. So, when the number of threads is increased to 32, time of execution is increased by a large extent.

2. When the number of threads is kept constant and the number of points is increased, the time taken to execute increases.



The above plot is for time taken vs number of points when the number of threads is kept constant at 32.
The reason for this behaviour is that number of points increases the number of computations. Therefore the time of execution increases.