OperatingSystems Assignment_2
Es21btech11003

Report : Estimating the π value using monte-carlo method

Input_handling :

```
// input_handling
   FILE *input_fp = NULL;
   input_fp = fopen(argv[1] , "r");
   if(input_fp == NULL){
       printf("Error : Unable to open file");
       exit(1);
   }
   else{
       fscanf(input_fp , "%d %d" , &N , &K);
   }
```

Program takes input from a file name of which it take as a command line argument **argv[1]** . Using fopen & fscanf function it read first two space separated integer as N and K values .

Creating and joining threads :

```
//creating K threads
   for(int i = 1 ; i <= K ; i++){

       // initializing and allocating memory to the elements of data_thread array
       thread_data_arr[i-1].thread_no = i;
       thread_data_arr[i-1].hits_in = (point *)calloc((N/K + 1) , sizeof(point));

       /* create the thread */
       pthread_create(&tid[i-1], &attr, runner, (void*)&thread_data_arr[i-1]);

   }
```

Using a loop:
1) we initialize the parameters required for each of the thread
2) Using **pthread_create** we create each thread and invoke **runner** function
3) pthread_join stops the main thread and waits for all other thread to finish before further executing.
4) **inside_circle** keeps **sum** of no. of points inside the **circle** found by each thread
5) **inside_square** keeps **sum** of no. of points inside the **square** found by each thread

```
for(int i = 1 ; i <= K ; i++){
    /* wait for the thread to exit */
    pthread_join(tid[i-1],NULL);
    inside_circle += thread_data_arr[i-1].N_circle;
    inside_square += thread_data_arr[i-1].N_square;
}
```

Runner function :

Implementation of runner function :

1) Each thread evaluates atleast N/K ordered pairs(**x,y**).
2) At first in every iteration , an random ordered pair between is generated using rand() function with each x and y between **(-1 , 1).**

```
// generate x,y
double no1 = (double)rand();
double no2 = (double)rand();

//converting no1 & no2 to doubles between [-1 , 1]
double _x = (no1/(RAND_MAX/2.0)) - 1;
double _y = (no2/(RAND_MAX/2.0)) - 1;
```

3) A thread store the points(x and y) which it has checked in a points array(**hits_in**) for the output later.
4) Each thread also keeps track of no. of points that are inside the circle of unit radius.
5) For every point , **in_circle** function is called such that it returns 1(true value) if point is inside the circle else returns 0(false value) .

```
//checking if the point is inside circle or not
      if(in_circle(_x , _y)){
          thread_data->hits_in[i].inside = 1;
          thread_data->N_circle++;
      }
      else{
          thread_data->hits_in[i].inside = 0;
      }
```

6) Finally each thread exits(pthread_exit(0);).

Structs:

1)**point:**

```
typedef struct{
   double x;
   double y;
   bool inside;
}point;
```

Struct point consist of two double x , y which are used to stores the value of each each randomly ordered pair and a bool inside : 1 if the point is inside units circle else 0.

2)**Data_thread:**

```
// data to be passed as the parameter with each thread
typedef struct{
   int thread_no;
   point *hits_in;
   int N_circle;
   int N_square;
}data_thread;
```

This is used to store all the info which can be associated with each thread:

1) Thread_no
2) hits_in - A point array storing info about each point associated with that thread
3) N_circle - No. of points found inside the unit circle for that thread.
4) N_square - No. of threads found inside the square.

π_value function:

Function return estimated π value using expression :

$$\pi = 4\times \textit{(number of points in circle)} / \textit{(total number of points)}$$

OutMain file :

Program creates OutMain.txt in following format :

1) Time taken by the program to evaluate given no. of points
2) Estimated π value
3) Log which contains all the points evaluated by the each thread.

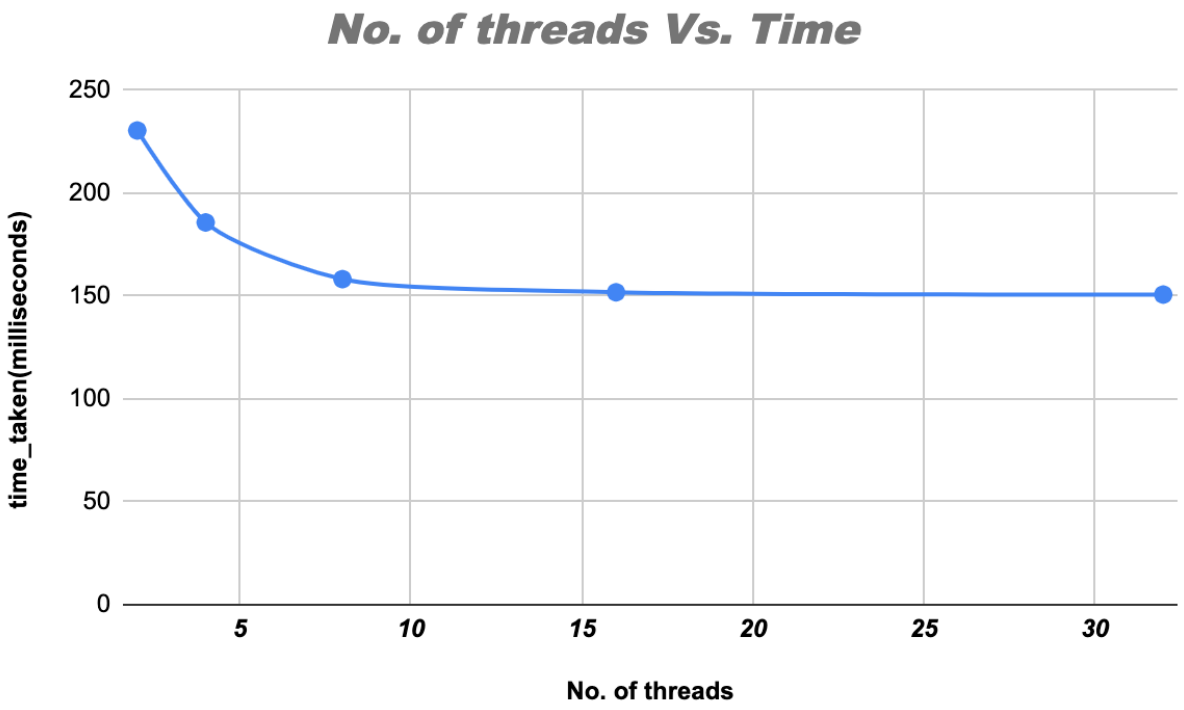
Output analysis :

Here we see the effects of no. of threads and the no. of points on execution time of the program :

1) No. of threads vs Time

Threads	Time_taken in each iterations					Time average
	1st	2nd	3rd	4th	5th	
2	229.091	231.744	228.746	230.625	232.062	230.4536
4	187.906	185.017	188.788	182.057	185.079	185.7694
8	162.305	154.888	158.29	159.294	156.284	158.2122
16	151.284	153.492	152.63	150.506	151.188	151.82
32	154.26	148.764	149.079	152.389	149.068	150.712

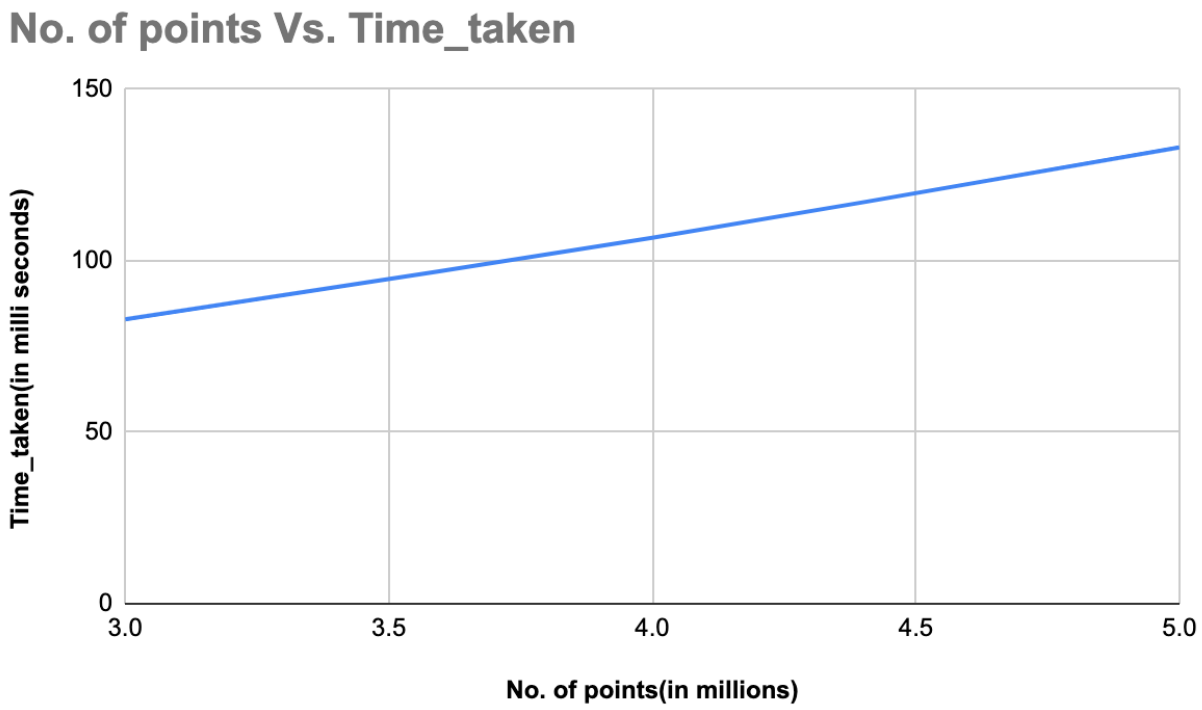
Here we can see as we increase the no of threads(for same no. of points to be evaluated) , the execution time of the program decreases . The variation can be seen in the given plot :



2) No. of points vs time

No. of points (10^6)	Time_taken in each iterations					Time average
	1st	2nd	3rd	4th	5th	
1	26.989	29.77	27.122	28.894	29.568	28.4686
2	56.247	54.763	55.532	57.359	56.068	55.9938
3	81.719	83.756	81.255	83.025	84.81	82.913
4	102.699	106.732	110.015	105.737	108.359	106.7084
5	127.924	131.432	132.639	137.872	135.316	133.0366

Here keeping the no. of thread to be 32 , we can see as we increase the no. of points the execution time of the program increases and can also be seen in the given plot :



Note : the time for execution of the program is calculated using `chrono library` available for C++.
The time taken for creating out file is excluded from execution time.