- In this assignment we will use the parallel loop construct of OpenMp and will also look into how loop scheduling works in **OpenMp for** and also study the effect of varying number of threads on the execution time of a program.

- Using **#pragma Openmp parallel for** all threads parallelly work on different parts of the problem.

- Number of threads working together to solve a problem can be controlled using

  omp_set_num_threads(threads); (threads is accepted as a command line parameter)

- There are different scheduling policies available in OpenMp loop scheduling
  1. **Static scheduling**
     Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size
  2. **Dynamic Scheduling**
     Uses the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue
  3. **Guided Scheduling**
     Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations.
  4. **Auto Scheduling**
     When schedule (auto) is specified, the decision regarding scheduling is delegated to the compiler
  5. **Runtime Scheduling**
     Uses the OMP_schedule environment variable to specify which one of the three loop-scheduling types should be used.

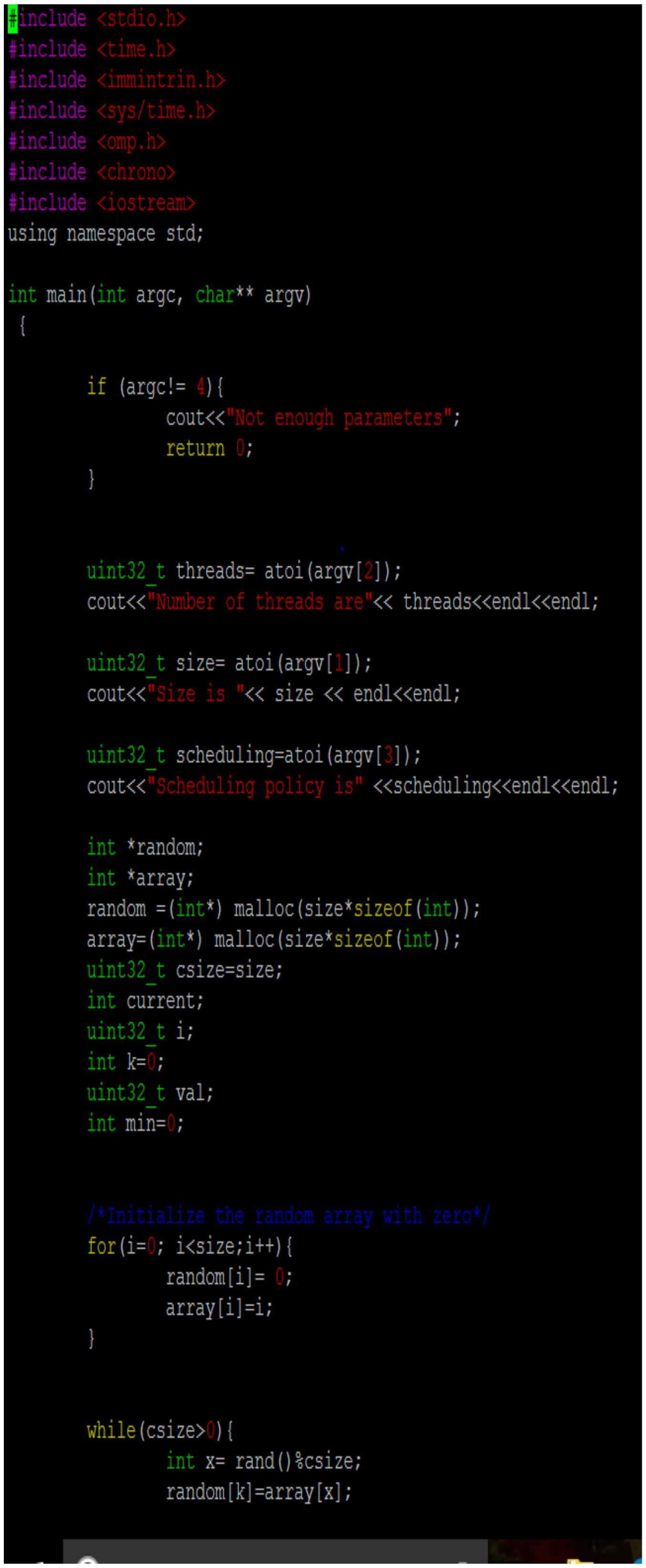
Minimum.c Position.c Square.c

https://github.com/abhi1212/High-performance-Computing

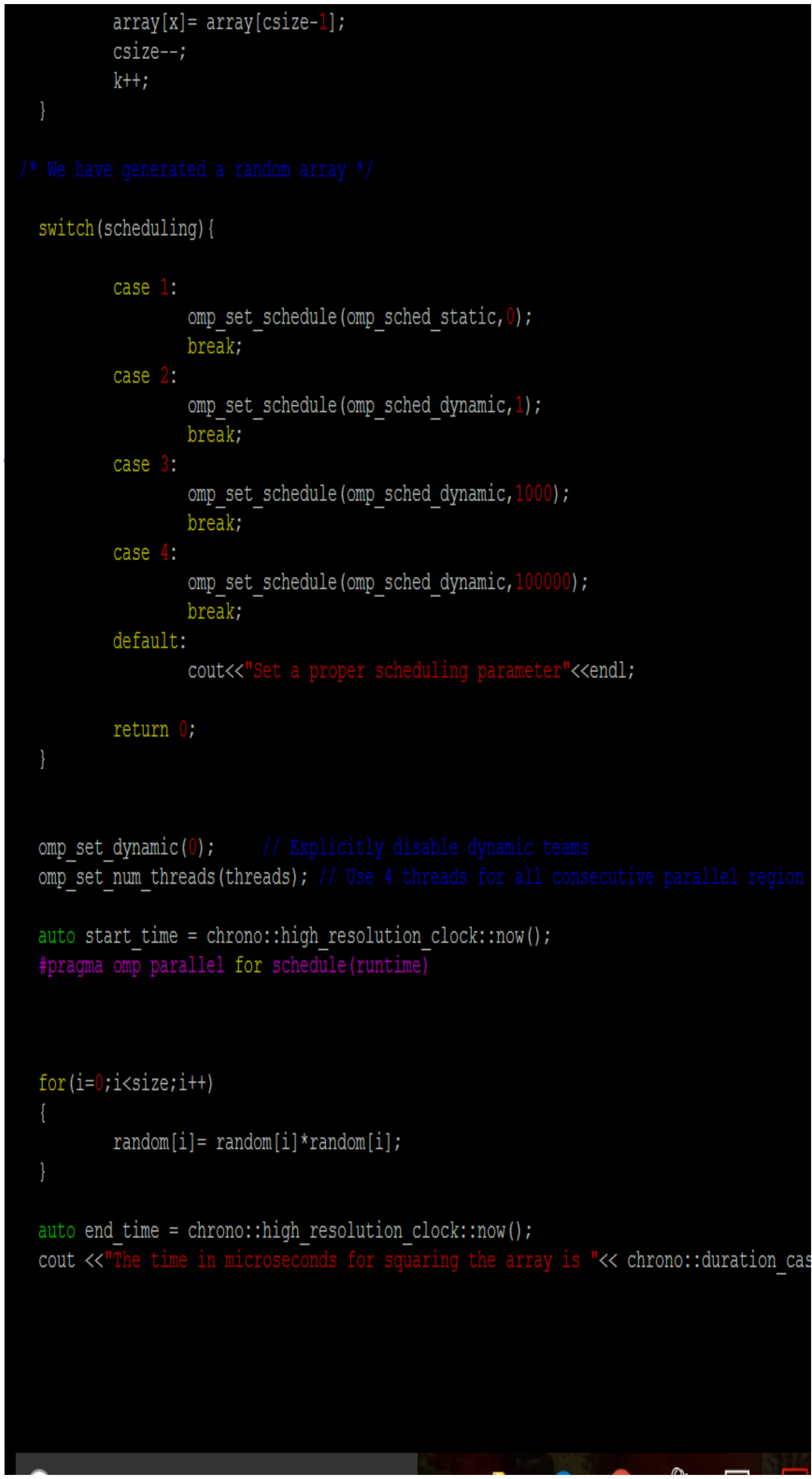
Assignment 3 AbhishekNikam

Code for Squaring an array using OpenMp for

- The size of the array, number of threads and scheduling policy are passed as command line parameters.



```
#include <stdio.h>
#include <time.h>
#include <immintrin.h>
#include <sys/time.h>
#include <omp.h>
#include <chrono>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
 {

        if (argc!= 4){
                cout<<"Not enough parameters";
                return 0;
        }


        uint32_t threads= atoi(argv[2]);
        cout<<"Number of threads are"<< threads<<endl<<endl;

        uint32_t size= atoi(argv[1]);
        cout<<"Size is "<< size << endl<<endl;

        uint32_t scheduling=atoi(argv[3]);
        cout<<"Scheduling policy is" <<scheduling<<endl<<endl;

        int *random;
        int *array;
        random =(int*) malloc(size*sizeof(int));
        array=(int*) malloc(size*sizeof(int));
        uint32_t csize=size;
        int current;
        uint32_t i;
        int k=0;
        uint32_t val;
        int min=0;


        /*Initialize the random array with zero*/
        for(i=0; i<size;i++){
                random[i]= 0;
                array[i]=i;
        }


        while(csize>0){
                int x= rand()%csize;
                random[k]=array[x];
```

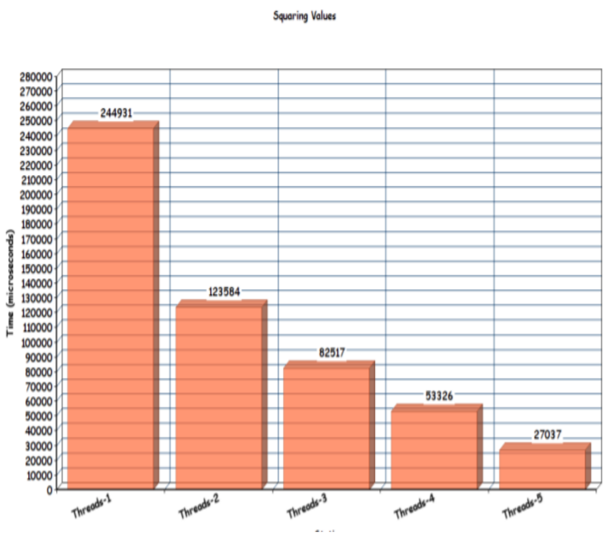


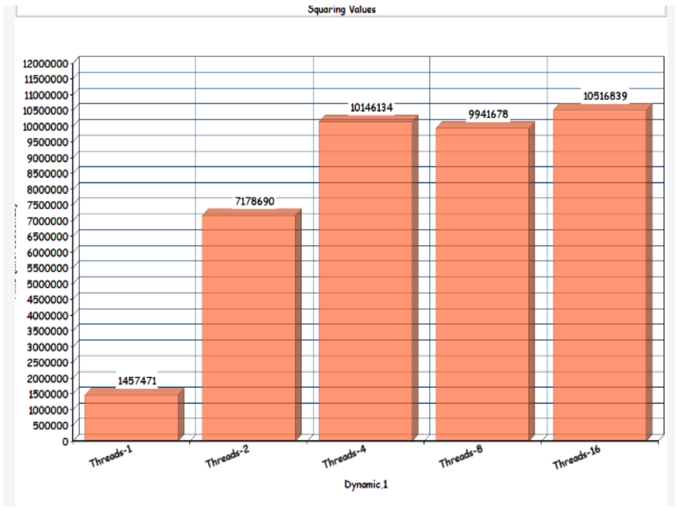
```
                array[x]= array[csize-1];
                csize--;
                k++;
        }

/* We have generated a random array */

        switch(scheduling){

                case 1:
                        omp_set_schedule(omp_sched_static,0);
                        break;
                case 2:
                        omp_set_schedule(omp_sched_dynamic,1);
                        break;
                case 3:
                        omp_set_schedule(omp_sched_dynamic,1000);
                        break;
                case 4:
                        omp_set_schedule(omp_sched_dynamic,100000);
                        break;
                default:
                        cout<<"Set a proper scheduling parameter"<<endl;

                return 0;
        }


        omp_set_dynamic(0);     // Explicitly disable dynamic teams
        omp_set_num_threads(threads); // Use 4 threads for all consecutive parallel region

        auto start_time = chrono::high_resolution_clock::now();
        #pragma omp parallel for schedule(runtime)



        for(i=0;i<size;i++)
        {
                random[i]= random[i]*random[i];
        }

        auto end_time = chrono::high_resolution_clock::now();
        cout <<"The time in microseconds for squaring the array is "<< chrono::duration_cas
```

Minimum.c Position.c Square.c

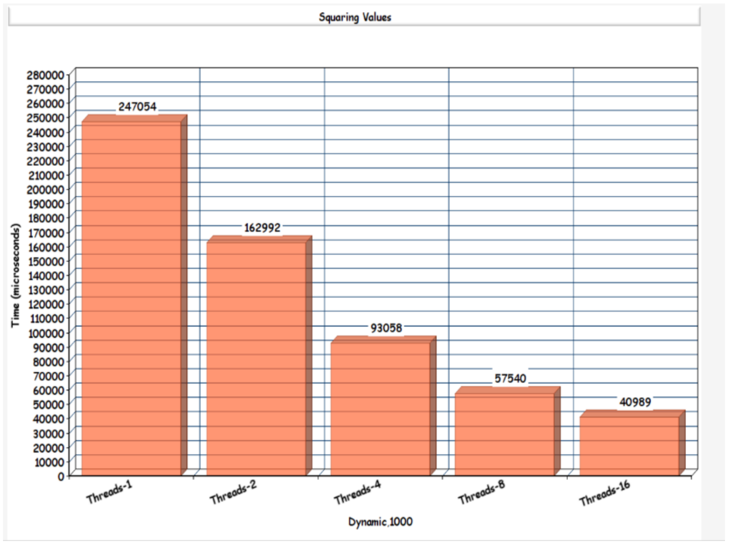
- After accepting parameters from the command line, we check whether the scheduling selected is a valid one else we terminate the process with a proper error message.
- If the arguments are more or less than the expected then also we terminate the process with appropriate message.
- With the help of **omp_set_num_threads(threads)** we set number of threads which will work on the problem.
- Using Runtime scheduling we take the help of OMP_schedule environment variable to specify the schedule.

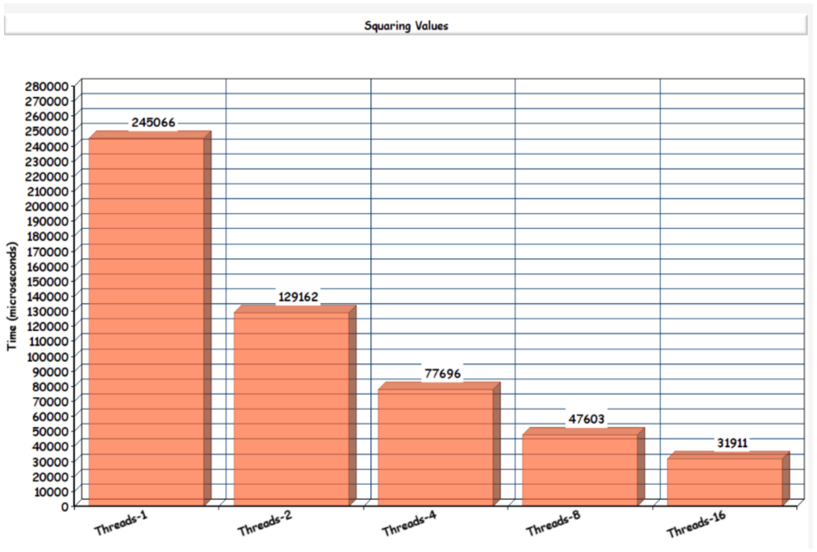
Results





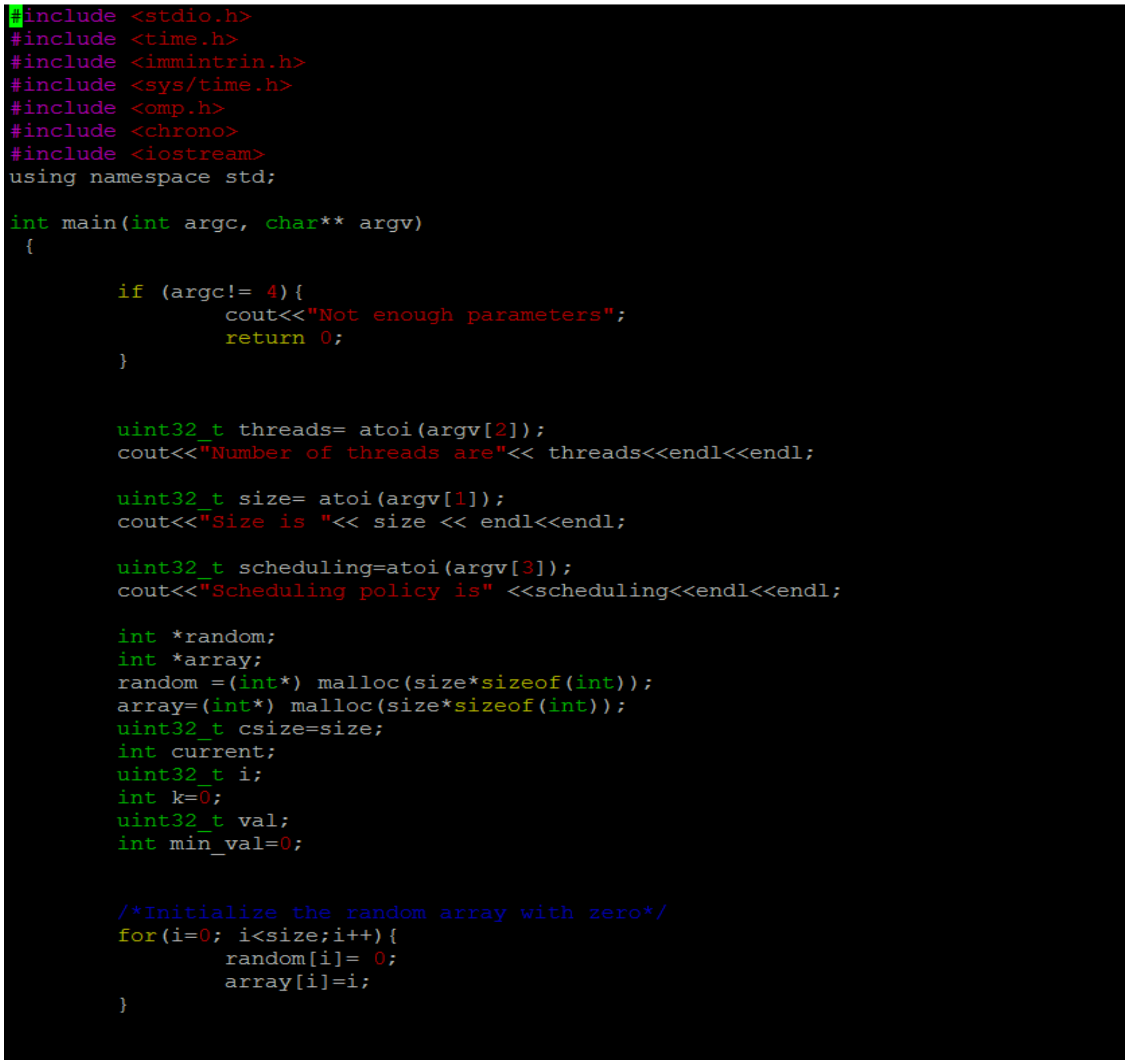
Minimum.c Position.c Square.c



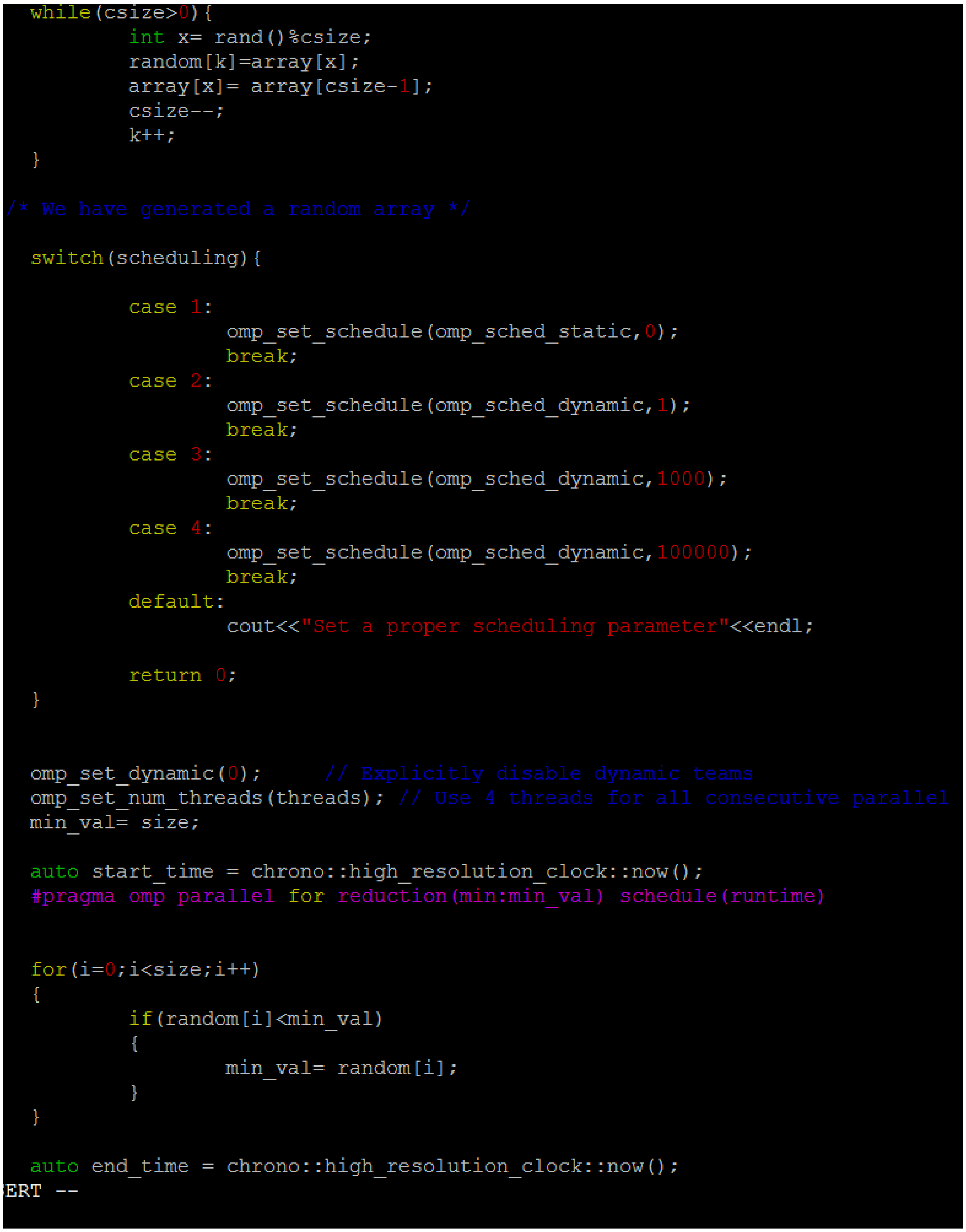


- For number of threads varying from 1 to 16 (1,2,4,8,16) we use all possible scheduling and plot the above graph.
- The time required for execution drastically as number of threads are increased.
- Also, time required for execution in Dynamic scheduling policy with chunk size as 1 is too much because the overhead for context switching dominates the time required for computation.
- Also, using Dynamic scheduling with chunk size of 100000 gives us better result than Dynamic 1000 as context switching time overhead is decreased.

Code for finding minimum value using OpenMp for

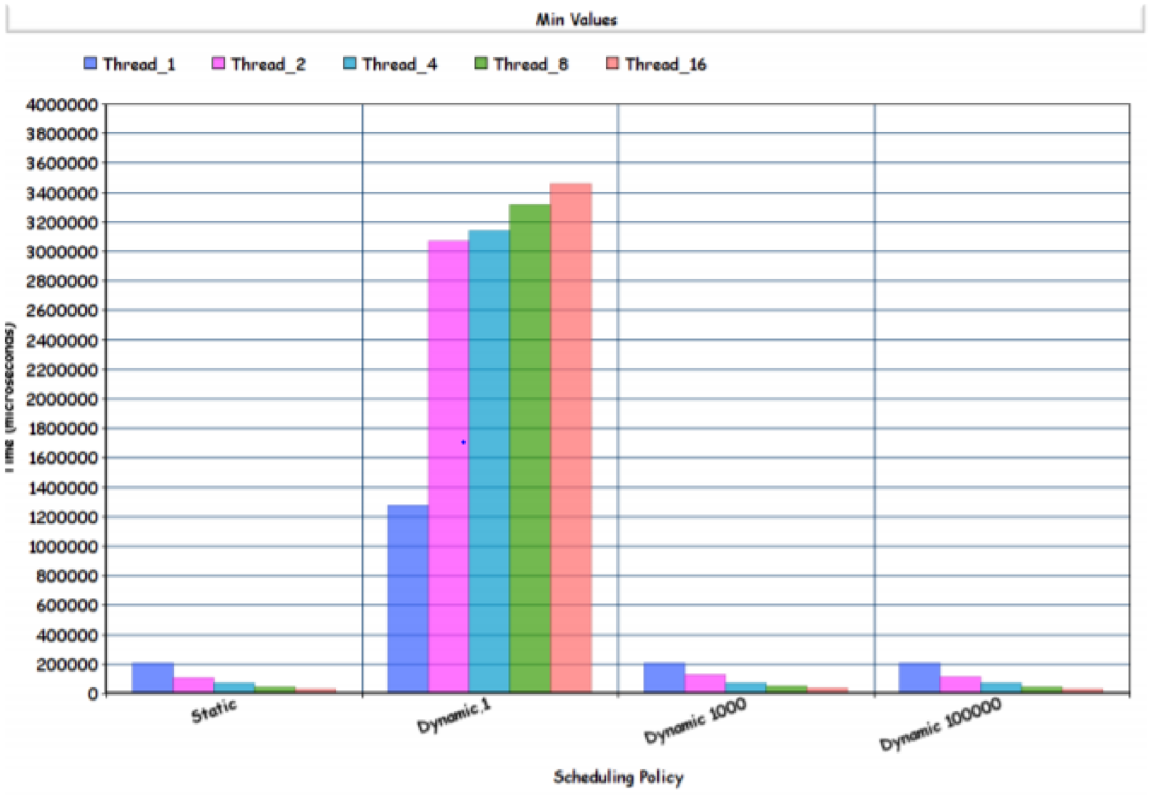


```
#include <stdio.h>
#include <time.h>
#include <immintrin.h>
#include <sys/time.h>
#include <omp.h>
#include <chrono>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
 {

        if (argc!= 4){
                cout<<"Not enough parameters";
                return 0;
        }


        uint32_t threads= atoi(argv[2]);
        cout<<"Number of threads are"<< threads<<endl<<endl;

        uint32_t size= atoi(argv[1]);
        cout<<"Size is "<< size << endl<<endl;

        uint32_t scheduling=atoi(argv[3]);
        cout<<"Scheduling policy is" <<scheduling<<endl<<endl;

        int *random;
        int *array;
        random =(int*) malloc(size*sizeof(int));
        array=(int*) malloc(size*sizeof(int));
        uint32_t csize=size;
        int current;
        uint32_t i;
        int k=0;
        uint32_t val;
        int min_val=0;


        /*Initialize the random array with zero*/
        for(i=0; i<size;i++){
                random[i]= 0;
                array[i]=i;
        }
```



```
  while(csize>0){
          int x= rand()%csize;
          random[k]=array[x];
          array[x]= array[csize-1];
          csize--;
          k++;
  }

/* We have generated a random array */

  switch(scheduling){

          case 1:
                  omp_set_schedule(omp_sched_static,0);
                  break;
          case 2:
                  omp_set_schedule(omp_sched_dynamic,1);
                  break;
          case 3:
                  omp_set_schedule(omp_sched_dynamic,1000);
                  break;
          case 4:
                  omp_set_schedule(omp_sched_dynamic,100000);
                  break;
          default:
                  cout<<"Set a proper scheduling parameter"<<endl;

          return 0;
  }


  omp_set_dynamic(0);      // Explicitly disable dynamic teams
  omp_set_num_threads(threads); // Use 4 threads for all consecutive parallel
  min_val= size;

  auto start_time = chrono::high_resolution_clock::now();
  #pragma omp parallel for reduction(min:min_val) schedule(runtime)


  for(i=0;i<size;i++)
  {
          if(random[i]<min_val)
          {
                  min_val= random[i];
          }
  }

  auto end_time = chrono::high_resolution_clock::now();
ERT --
```

Results Observed

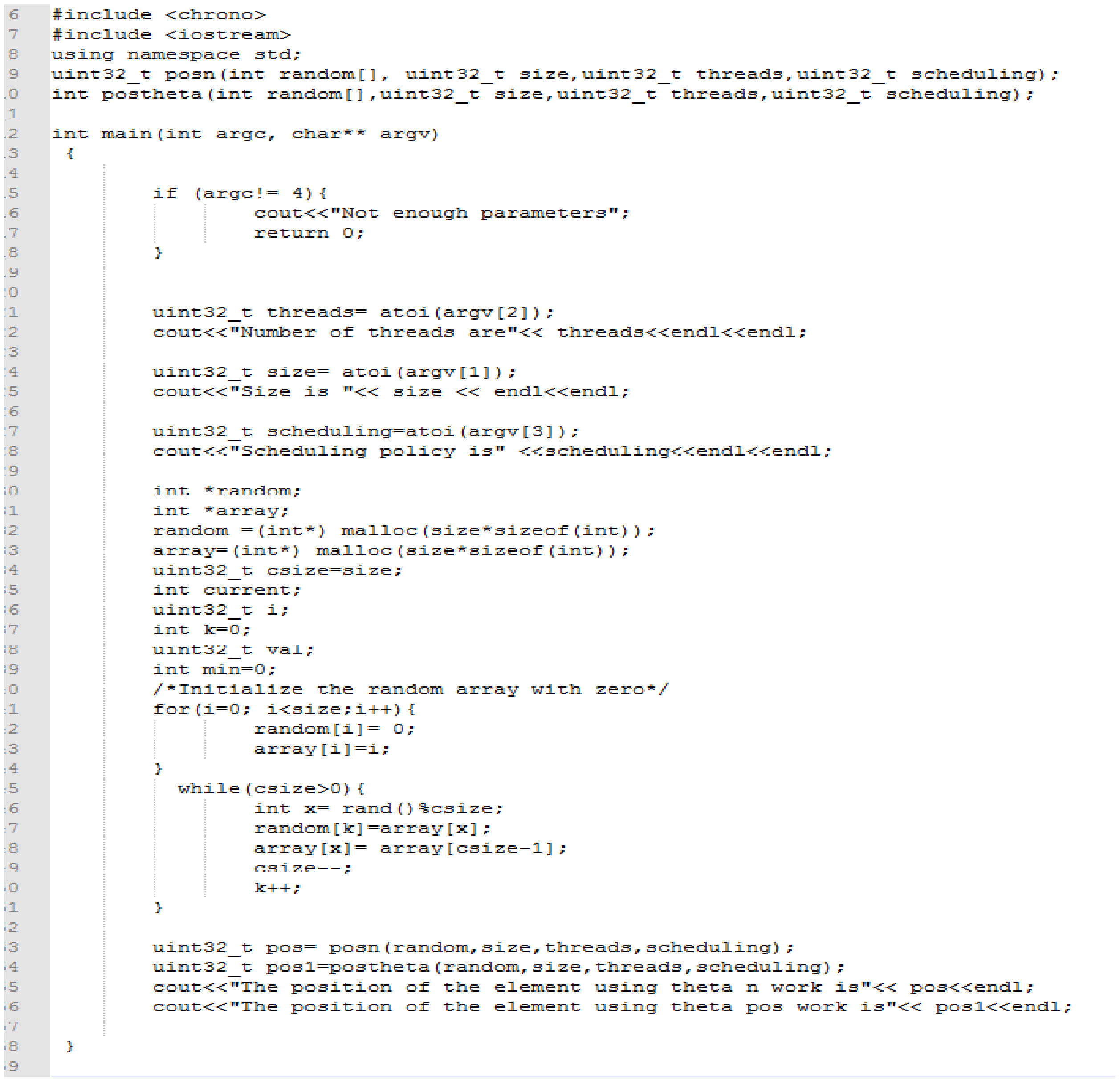


- In this code we are required to find minimum value within an array.
- As each thread works on different part of our array processing, each thread can get a different minimum values.
- To ensure that we have a global minimum value we use **for reduction(min:min_val)** which chooses the most smallest value of min_val which is a shared variable amongst the threads.
- As we observed while squaring array values, here also dynamic,1 scheduling is inefficient and as you increase your threads your execution time decreases considerably.

Code for finding the first occurrence of an integer using theta(n) and theta (pos) work

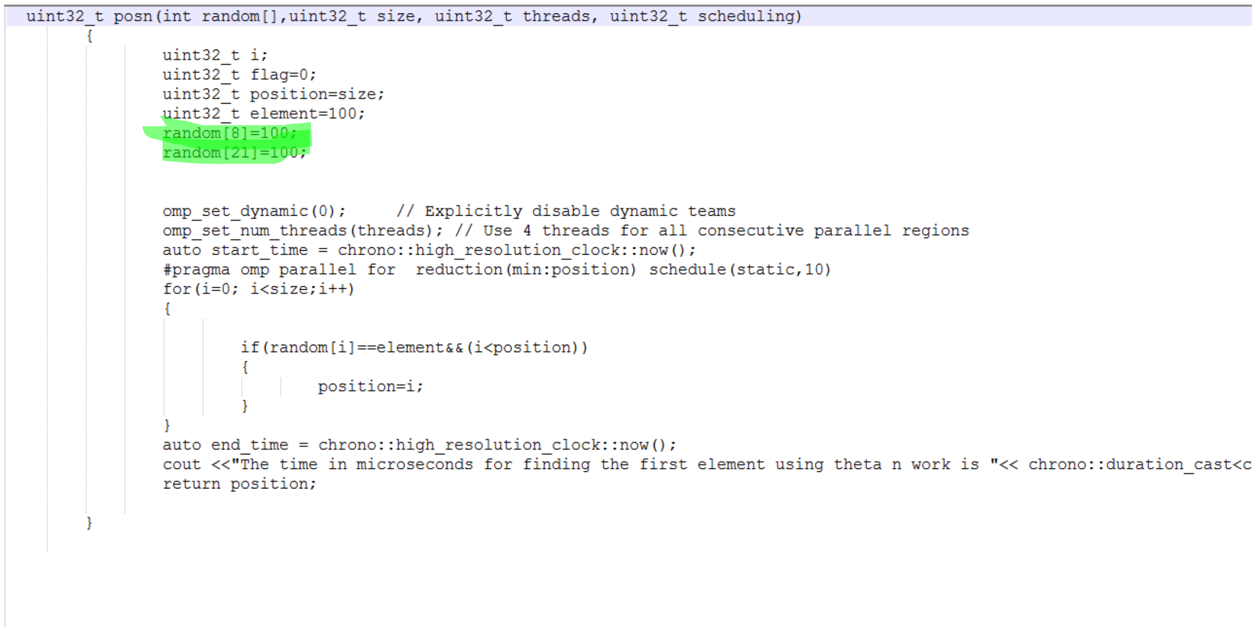
- I selected static scheduling because dynamic scheduling type is appropriate when the iterations require different computational costs. This means that the iterations are poorly balanced between each other. The dynamic scheduling type has higher overhead then the static scheduling type because it dynamically distributes the iterations during the runtime.

Main Loop



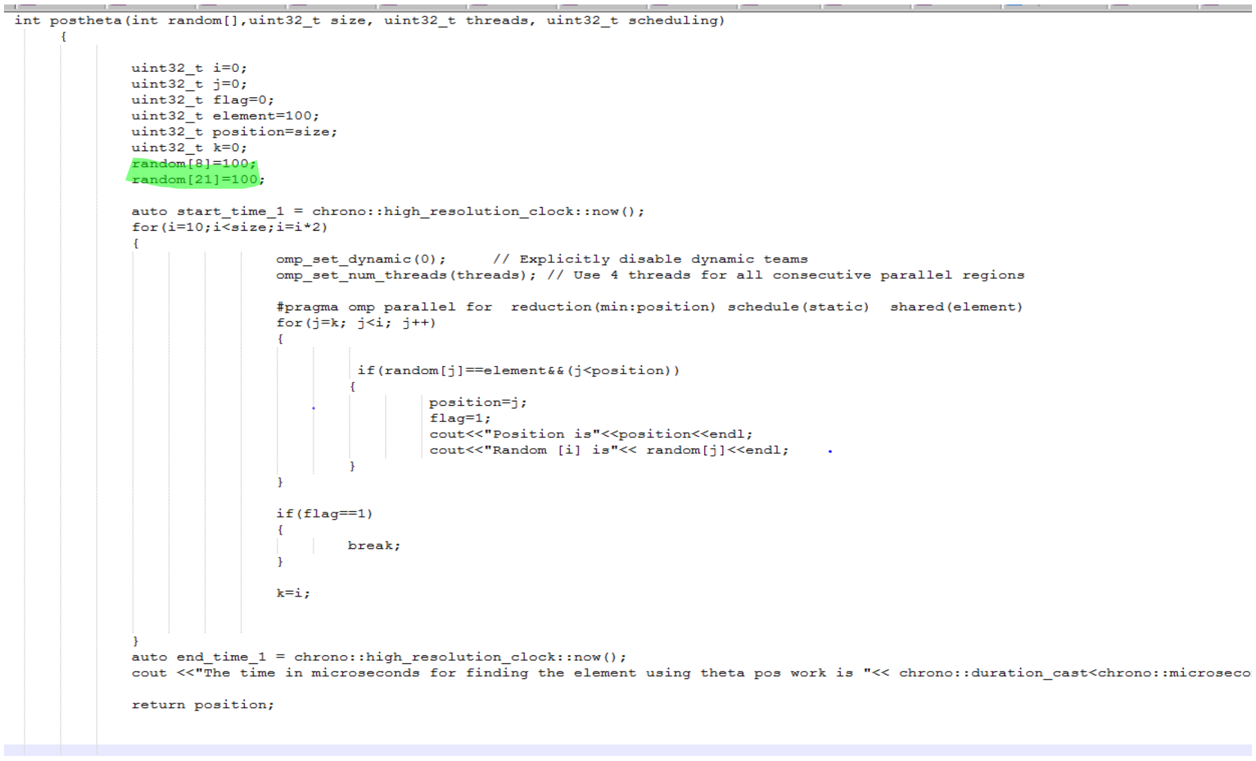
```
#include <chrono>
#include <iostream>
using namespace std;
uint32_t posn(int random[], uint32_t size,uint32_t threads,uint32_t scheduling);
int postheta(int random[],uint32_t size,uint32_t threads,uint32_t scheduling);

int main(int argc, char** argv)
 {

        if (argc!= 4){
                cout<<"Not enough parameters";
                return 0;
        }


        uint32_t threads= atoi(argv[2]);
        cout<<"Number of threads are"<< threads<<endl<<endl;

        uint32_t size= atoi(argv[1]);
        cout<<"Size is "<< size << endl<<endl;

        uint32_t scheduling=atoi(argv[3]);
        cout<<"Scheduling policy is" <<scheduling<<endl<<endl;

        int *random;
        int *array;
        random =(int*) malloc(size*sizeof(int));
        array=(int*) malloc(size*sizeof(int));
        uint32_t csize=size;
        int current;
        uint32_t i;
        int k=0;
        uint32_t val;
        int min=0;
        /*Initialize the random array with zero*/
        for(i=0; i<size;i++){
                random[i]= 0;
                array[i]=i;
        }
          while(csize>0){
                int x= rand()%csize;
                random[k]=array[x];
                array[x]= array[csize-1];
                csize--;
                k++;
        }

        uint32_t pos= posn(random,size,threads,scheduling);
        uint32_t pos1=postheta(random,size,threads,scheduling);
        cout<<"The position of the element using theta n work is"<< pos<<endl;
        cout<<"The position of the element using theta pos work is"<< pos1<<endl;

 }
```

Function to find pos in theta (n) time



```
uint32_t posn(int random[],uint32_t size, uint32_t threads, uint32_t scheduling)
        {
                uint32_t i;
                uint32_t flag=0;
                uint32_t position=size;
                uint32_t element=100;
                random[8]=100;
                random[21]=100;


                omp_set_dynamic(0);     // Explicitly disable dynamic teams
                omp_set_num_threads(threads); // Use 4 threads for all consecutive parallel regions
                auto start_time = chrono::high_resolution_clock::now();
                #pragma omp parallel for  reduction(min:position) schedule(static,10)
                for(i=0; i<size;i++)
                {

                        if(random[i]==element&&(i<position))
                        {
                                position=i;
                        }
                }
                auto end_time = chrono::high_resolution_clock::now();
                cout <<"The time in microseconds for finding the first element using theta n work is "<< chrono::duration_cast<c
                return position;

        }
```

Function to find pos in theta (pos) time

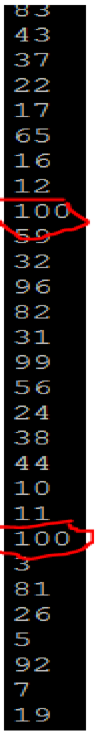


```
int postheta(int random[],uint32_t size, uint32_t threads, uint32_t scheduling)
        {

                uint32_t i=0;
                uint32_t j=0;
                uint32_t flag=0;
                uint32_t element=100;
                uint32_t position=size;
                uint32_t k=0;
                random[8]=100;
                random[21]=100;

                auto start_time_1 = chrono::high_resolution_clock::now();
                for(i=10;i<size;i=i*2)
                {
                                omp_set_dynamic(0);     // Explicitly disable dynamic teams
                                omp_set_num_threads(threads); // Use 4 threads for all consecutive parallel regions

                                #pragma omp parallel for  reduction(min:position) schedule(static)  shared(element)
                                for(j=k; j<i; j++)
                                {

                                        if(random[j]==element&&(j<position))
                                        {
                                                position=j;
                                                flag=1;
                                                cout<<"Position is"<<position<<endl;
                                                cout<<"Random [i] is"<< random[j]<<endl;
                                        }
                                }

                                if(flag==1)
                                {
                                        break;
                                }

                                k=i;


                }
                auto end_time_1 = chrono::high_resolution_clock::now();
                cout <<"The time in microseconds for finding the element using theta pos work is "<< chrono::duration_cast<chrono::microseco

                return position;
```

Note->All the excess print statements were used for debugging purposes and were removed during time measurement.

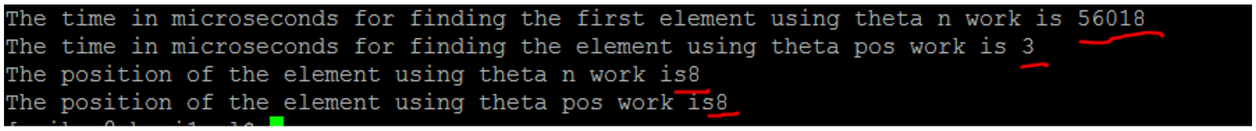
Minimum.c Position.c Square.c

- So basically while implementing the function posn(), we had to take care that we have to return the first occurrence position of the desired element.
- So, we use the **if(random[i]==element&&(i<position))** condition to get the smallest position value within a thread.
- Then, we apply **reduction(min:position)** to get the minimum pos value from all threads.
- The variable position is private to each thread as we have used a reduction with it.
- For finding the first occurrence position of the desired element using theta pos work we need to use a concept called as Geometric doubling.
- In geometric doubling we keep on doubling the size of dataset under consideration, until we get the desired element.
- We might get that element in multiple threads , so to ensure that we have the first occurrence position we, use min reduction
- Also **if(random[i]==element&&(i<position))** condition allows us to choose the lowest position value if there are more than two occurrences of the desired element in the same thread.
- So in function thetapos we break out of the outer for loop as soon as we find the first occurrence of the element.
- We cannot break out of the Omp for.
- I have highlighted some part in Green color, where I am initializing the 8th and 21st element of array to be 100.



```
83
43
37
22
17
65
16
12
100
59
32
96
82
31
99
56
24
38
44
10
11
100
3
81
26
5
92
7
19
```

- When we run the code the output we get is the following



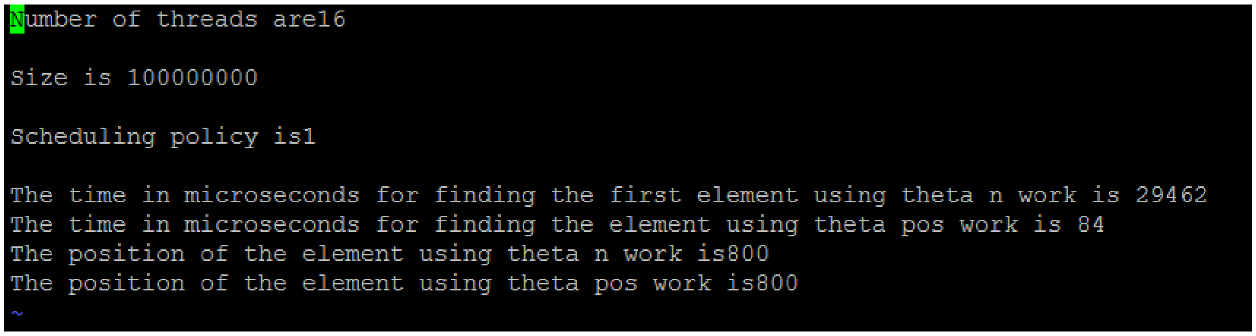
```
The time in microseconds for finding the first element using theta n work is 56018
The time in microseconds for finding the element using theta pos work is 3
The position of the element using theta n work is8
The position of the element using theta pos work is8
```

We can observe that in spite of multiple copies of element 100 we get the position of 1st occurrence of it.

Also the time required to find while doing n work is 56018 microseconds.
While we can do it by theta pos work in 3 microseconds.

Results

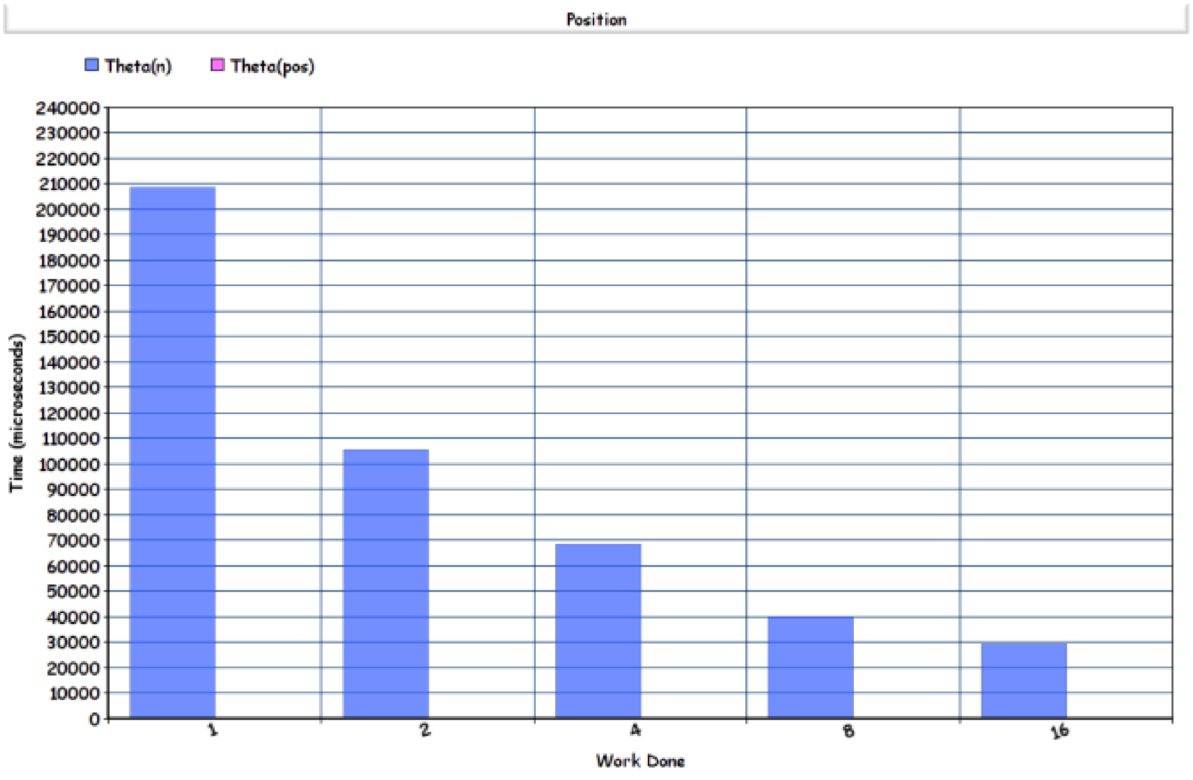
- ➔ We are searching element 100 which first occurrence is in 800th position
- ➔ A snapshot of a sample output would be as follows.



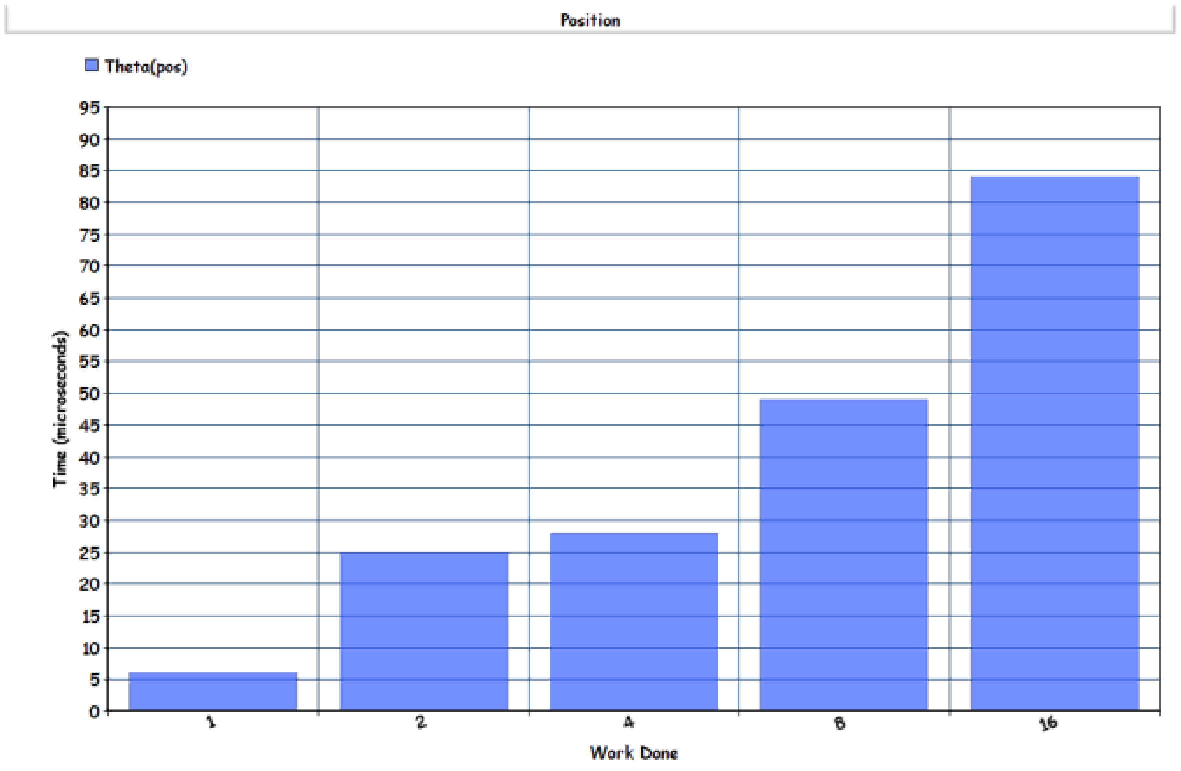
```
Number of threads are16

Size is 100000000

Scheduling policy is1

The time in microseconds for finding the first element using theta n work is 29462
The time in microseconds for finding the element using theta pos work is 84
The position of the element using theta n work is800
The position of the element using theta pos work is800
~
```

➔

Time graph for theta(n) work done



Time graph for theta pos work done



- ➔ For theta(n) work done the time required for computation decreases as we the number of threads
- ➔ For theta pos work done the time taken for computation increases as number of threads increase because the element is found at 800th position which is not too far away.
- ➔ So, the reason why my measured time goes up is that it takes more time to generate and handle more threads (which don't really do anything)