Design and Performance Report

On

Parallel Semi Ordered Matrix Search

by

SHIVANKIT GAIND	2015A7PS0076P
ROHIT GHIVDONDE	2015A7PS0077P

For the partial fulfilment of the course on

Parallel Computing

Submitted to
Prof. Shan Sundar Balasubramaniam

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
April, 2018

Design

1. The task of searching a number in a semi ordered matrix is divided among threads.

2. The design involves using each thread to apply the divide and conquer algorithm for semi ordered matrix search **independently** on the block of matrix assigned to it.

3. The matrix is divided into **blocks of equal sizes**(mostly square blocks) among threads.

4. Each thread then runs the given divide and conquer algorithm on its block:
    a. Consider matrix A

    $$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix}$$

    b. First check whether the input element **x** is equal to the middle element of the matrix.
    c. If x == middle element, return the middle element indices.
    d. Else if x > middle element, search for x in submatrices A_2, A_3, A_4.
    e. Else, search for x in submatrices A_1, A_2, A_3.

5. There are two variables shared by all threads, **i** and **j** where **i** represents the row number and **j** represents the column number of the searched element. Initially both are initialized to **-1**.

6. Once the searched element is found, the corresponding thread updates values of **i** and **j**.

7. Each thread checks for the values of the shared indices(whether they are filled or not) before entering the recursive calls. This way as soon as one of the threads finds the element(thus updating **i**, **j** and returning), all the other threads return as well thus saving a lot of unnecessary computations.

Alternate Design - Not chosen in our Solution

1. Instead of dividing the matrix into blocks the entire matrix is divided into subproblems in the form of **OpenMP tasks.**

2. Consider matrix A

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix}$$

    a. First check whether the input element **x** is equal to the middle element of the matrix.
    b. If x == middle element, return the middle element indices.
    c. Else if x > middle element, search for x in submatrices A_2, A_3, A_4 using **OpenMP tasks**. This way the subproblems are formed into tasks and added to the tasks queue. Whenever a thread gets free it picks up a task from the queue and starts implementing it.
    d. Else, search for x in submatrices A_1, A_2, A_3. Again the subproblems formed are added to the tasks queue.

3. Whenever a thread finds the element, it updates the shared indices **i** and **j** thus forcing other threads to return.

The reason this design doesn't work well here is because the work done per thread is very less as compared to the overhead of creating tasks dynamically as we encounter the subproblems. Thread switching between openMP tasks also has an overhead making this solution take more time than the previous one.

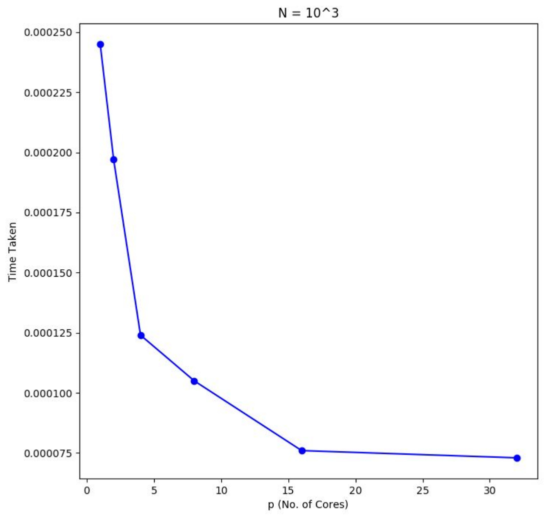
Performance Evaluation

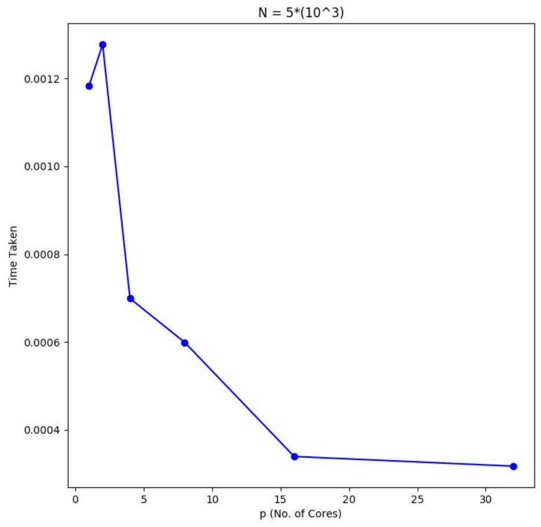
The performance measurements for different values of p (where p is the number of cores used) on an NxN matrix (i.e N^2 elements) at every value of N from 10^3 to 10^5 is given below:

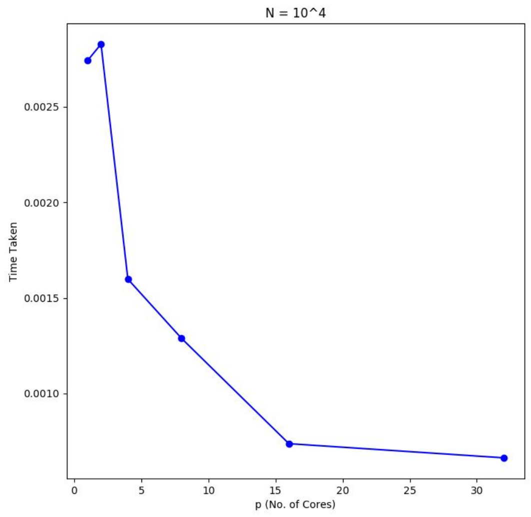
N/p	**1**	**2**	**4**	**8**	**16**	**32**
10^3	0.000245	0.000197	0.000124	0.000105	0.000076	0.000073
5*(10^3)	0.001183	0.001278	0.000699	0.000599	0.000339	0.000317
10^4	0.002742	0.002828	0.001598	0.001288	0.000737	0.000663
3*(10^4)	0.014863	0.012919	0.006644	0.004775	0.002166	0.001206
7*(10^4)	0.049942	0.02819	0.021201	0.01185	0.009945	0.005901
10^5	0.107643	0.048618	0.037454	0.024861	0.020848	0.013078

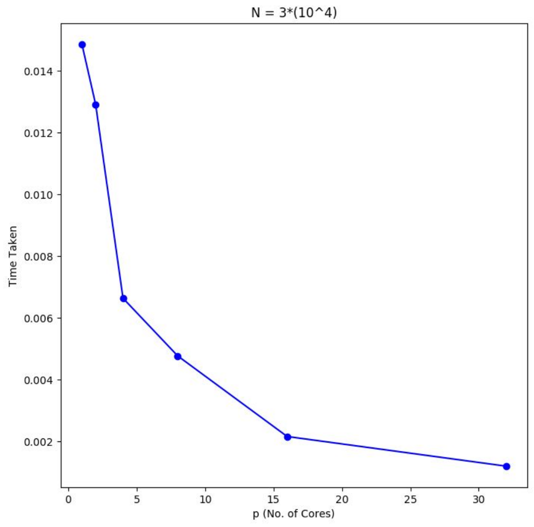
The plots showing how performance improves with increase in the number of cores at each value of N are given below:

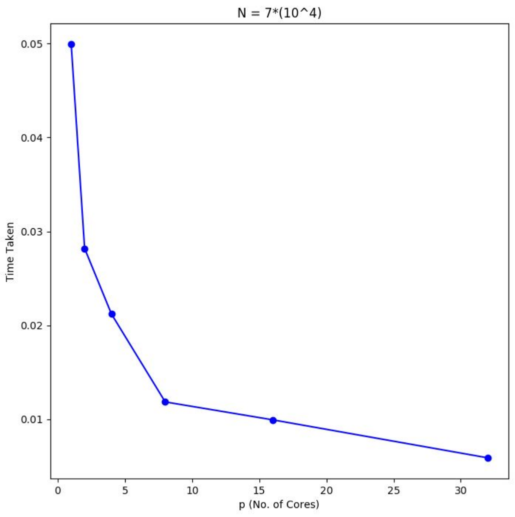
It is clear that the performance improves with utilization of more cores. Small anomalies are seen for small values of N (when we switch from 1 to 2 cores) because of **overhead of thread creation and dividing tasks** among threads whereas actually the work to be done is not distributed much (among 2 threads only).

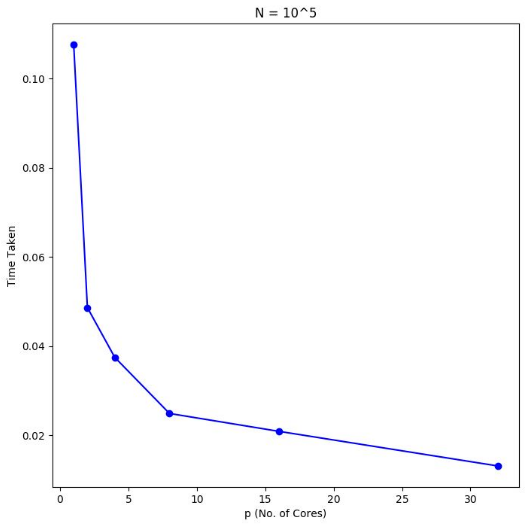

N = 10^3
Time Taken
0.000250
0.000225
0.000200
0.000175
0.000150
0.000125
0.000100
0.000075
0
5
10
15
20
25
30
p (No. of Cores)


N = 5*(10^3)
Time Taken
0.0012
0.0010
0.0008
0.0006
0.0004
0
5
10
15
20
25
30
p (No. of Cores)


N = 10^4
Time Taken
0.0025
0.0020
0.0015
0.0010
0
5
10
15
20
25
30
p (No. of Cores)

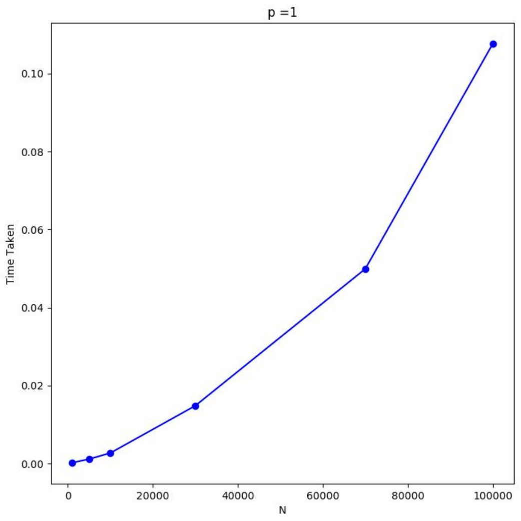

N = 3*(10^4)
Time Taken
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0
5
10
15
20
25
30
p (No. of Cores)

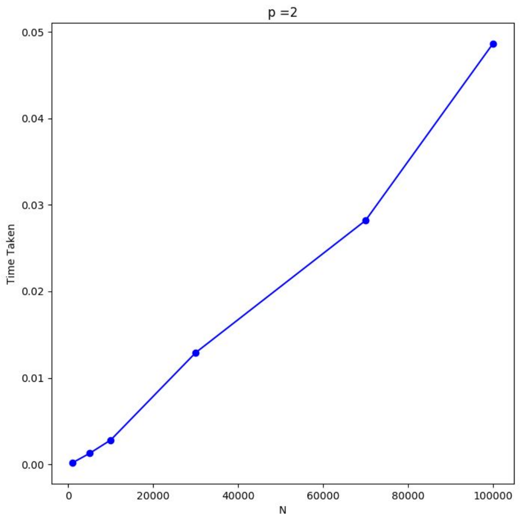


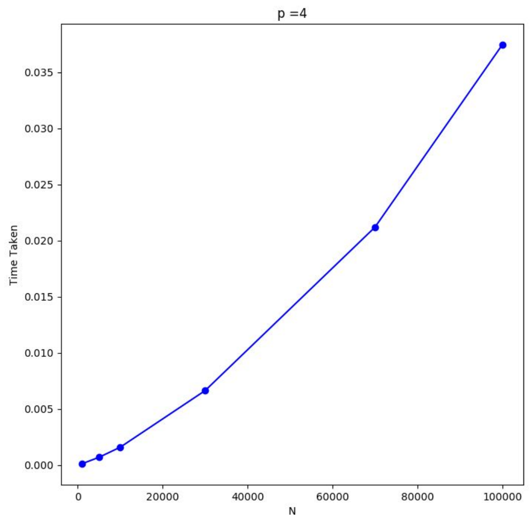


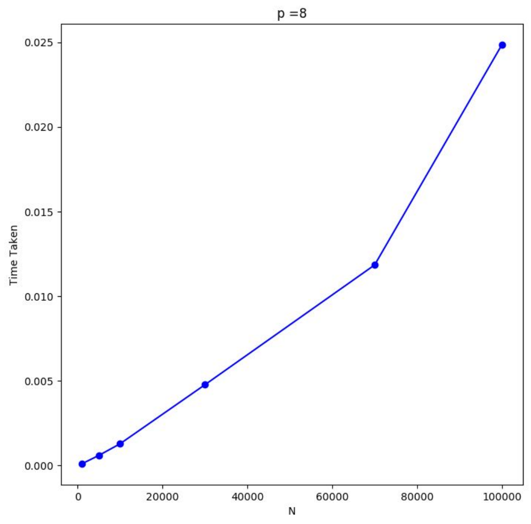
The plots showing how time increases with increase in the data size (i.e value of N) at a fixed value of p (number of cores used) are given below:

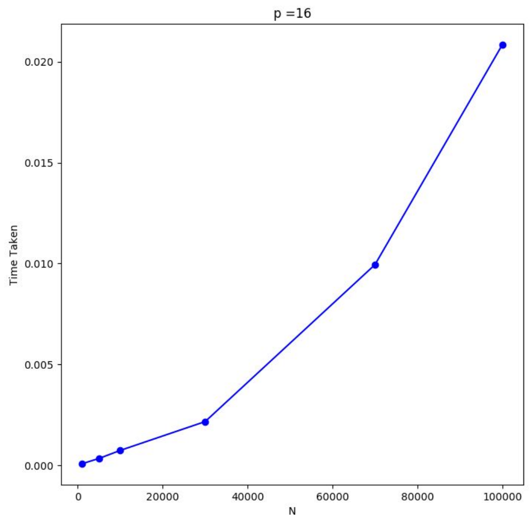
It is clear that keeping the number of cores constant, time increases with increase in the amount of work to do - i.e the value of N.

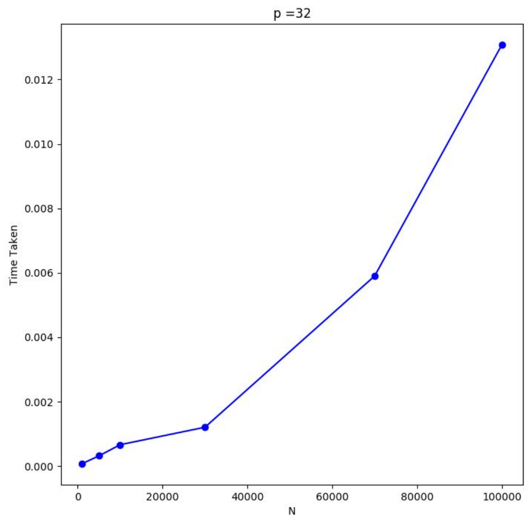





p =4
Time Taken
N
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035
0
20000
40000
60000
80000
100000


p =8
Time Taken
N
0.000
0.005
0.010
0.015
0.020
0.025
0
20000
40000
60000
80000
100000


p =16
Time Taken
N
0.000
0.005
0.010
0.015
0.020
0
20000
40000
60000
80000
100000


p =32
Time Taken
N
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0
20000
40000
60000
80000
100000