# PRAM Algorithm for searching in a Semi-Ordered Matrix

We divided the given problem into subproblems and each subproblem is assigned to one thread. The number of threads to spawn for the key searching depends on the number of cores given as input. Number of threads would be equal to $3^{fLevel}$ , where fLevel is defined below. Each thread is given a region of the original input matrix to handle, on which it runs recursively to search for the key.

$$(numCores / 3^{fLevel}) < 1 < (numCores / 3^{fLevel-1})$$

To divide into subproblems, we calculated the top left and bottom right indices of matrix region that a thread will handle. For this, we created a function which is initially called by master thread, and inside that function the master thread spawns 2 more threads, and then in the next recursive call, each of the 3 threads spawn 2 more threads (net 9 threads) and this process goes on till each threads reached the fLevel. And at fLevel, each thread calculates the boundaries of region for a thread to work upon. At any level, if the middle element of the matrix is equal to the key being searched, then program prints the index and exits.

After we get bounds of matrix for each thread to work on, each thread calls the findElement() function which recursively searches for the key in the region assigned to it, and if a thread finds the key, then this thread prints the index and program exits. The findElement function has the same design as the simple sequential version of finding an element in a semi ordered matrix.

## Cost Analysis

Sequential time complexity = $O(n^{1.58})$

Sequential cost = $O(n^{1.58})$

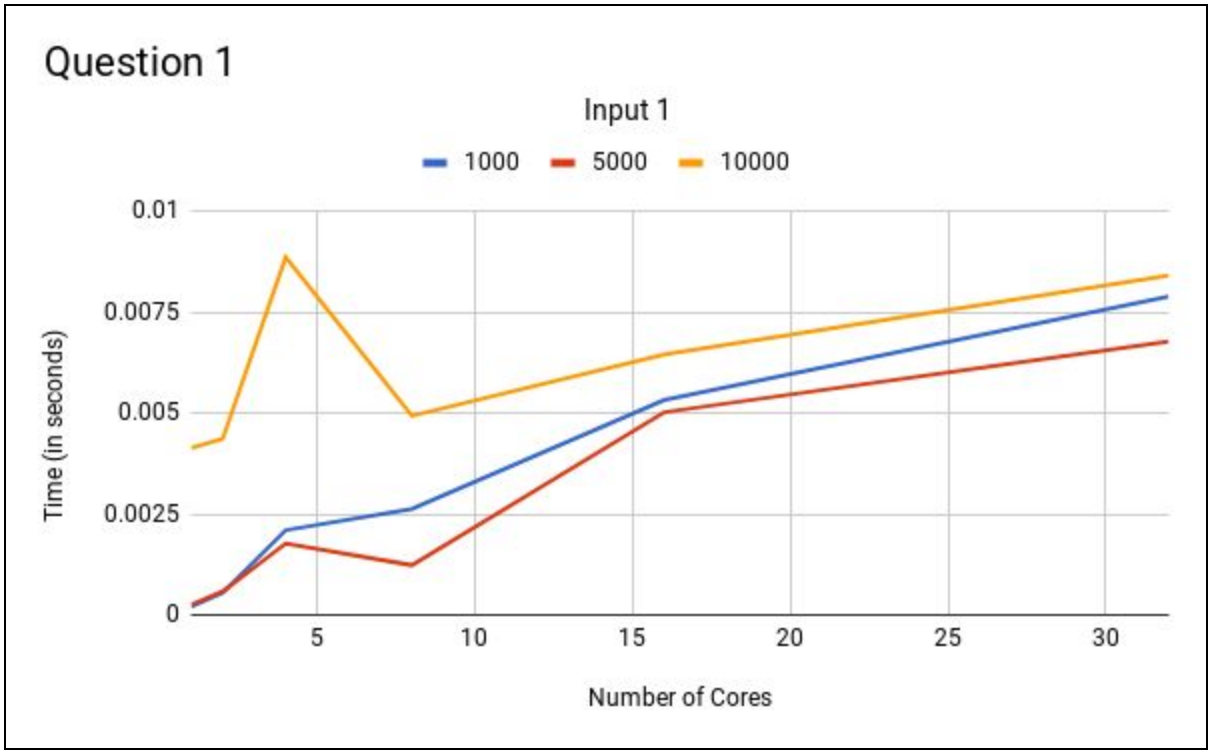
Parallel Time complexity = $O((n / 2^{fLevel})^{1.58})$

Parallel cost = $O((numCores)*(n / 2^{fLevel})^{1.58})$

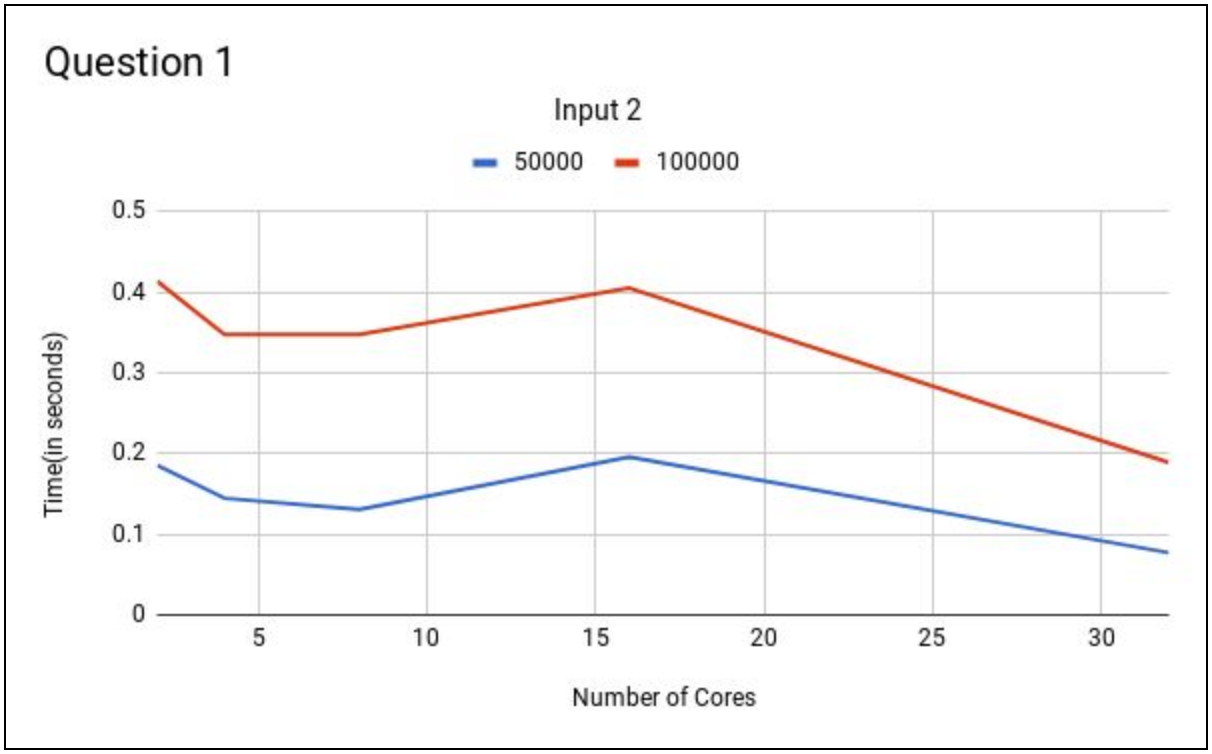
where fLevel is as defined above and numCores = number of cores given as input.

**Results of Question 1 are shown below:**

| | No. of Cores | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|---|
| **Matrix Size (nxn)** | **1000** | 0.000202 | 0.000543 | 0.002099 | 0.002625 | 0.005328 | 0.007898 |
| | **5000** | 0.000254 | 0.000578 | 0.001768 | 0.001232 | 0.005024 | 0.006775 |
| | **10000** | 0.004141 | 0.004362 | 0.008871 | 0.004936 | 0.006458 | 0.00841 |
| | **50000** | 0.185292 | 0.144615 | 0.130466 | 0.195909 | 0.077035 | 0.047801 |
| | **100000** | 0.413775 | 0.347528 | 0.347797 | 0.405385 | 0.188904 | 0.076415 |




As the input size is very small, above graphs shows that increase in number of threads (due to increase in number of cores), increases the Time Taken.




As the input size is significant, above graphs shows that increase in number of threads (due to increase in number of cores), decreases the Time Taken.