REPORT

Abhay Vivek Kulkarni (ak6277)

Parallel DP - OBST

Code workflow:

---

1. Get the size of keyset from program parameters,

   Eg: java parallelOBST 1000
   (for 1000 keys)

2. Multiple hard-coded global parameters are set for:

   - RANDOM_RANGE: range of randomly generated keys.
   - TEST: true -> Performs tests on some hard-coded inputs.
     false -> Skips all tests.
   - NUM_OF_THREADS: Specifies the number of threads to work on.
     (0 < threads <= available)
   - ERROR: Maximum error margin for double comparison.
   - BUILD_TREE: true -> Builds and prints the binary tree.

3. Generates an array of random values within a given range of the user input size. Normalizes the values such that:

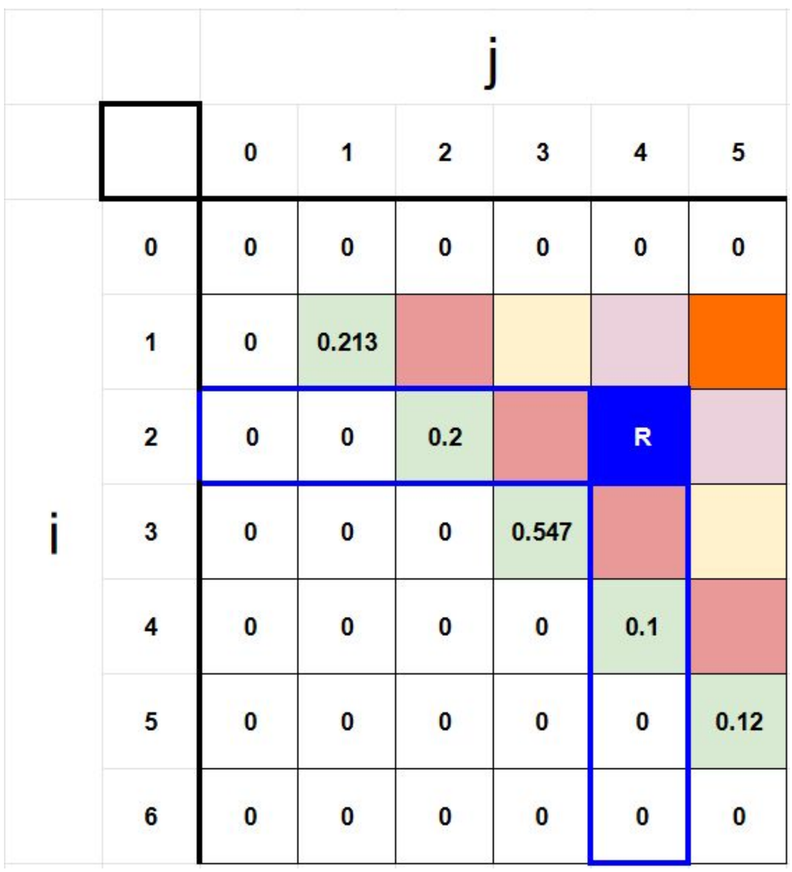
   [k1, k2, k3, …, kn] → [p1, p2, p3, …, pn]
   Such that ∑(p) = 1
   This converts the frequency of the keys to probabilities.

4. Main algorithm *findOptimalSolution*:
   - Creates a [n+2][n+1] size matrix.
   - Pads first row, last row, and first column with 0’s for easier index and edge case management.
   - Each unique diagonal has to be sequential.

Eg: For n=5
      Matrix = [7][6]
      Diagonals are: (1,1), (2,2), (3,3), (4,4), (5,5)   Difference=0
      (1,2), (2,3), (3,4), (4,5)   Difference=1
      (1,3), (2,4), (3,5)   Difference=2
      (1,4), (2,5)   Difference=3
      (1,5)   Difference=4

For each diagonal; different number of nodes are considered
From single -> all
Each entry in the diagonal can be calculated parallelly; but the diagonal
Have to be sequential.

- All individual values are updated parallelly by creating a forkJoinPool and parallel streams.



		j					
		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0.213	🟥	🟨	🟪	🟧
	2	0	0	0.2	🟥	R	🟪
i	3	0	0	0	0.547	🟥	🟨
	4	0	0	0	0	0.1	🟥
	5	0	0	0	0	0	0.12
	6	0	0	0	0	0	0

	A	B	C	D	E
p(key)	0.213	0.2	0.547	0.1	0.12

j-i is the diagonal id

For any element (R); the entire column below it and entire row besides it is required.

Each color represents the diagonal operation.

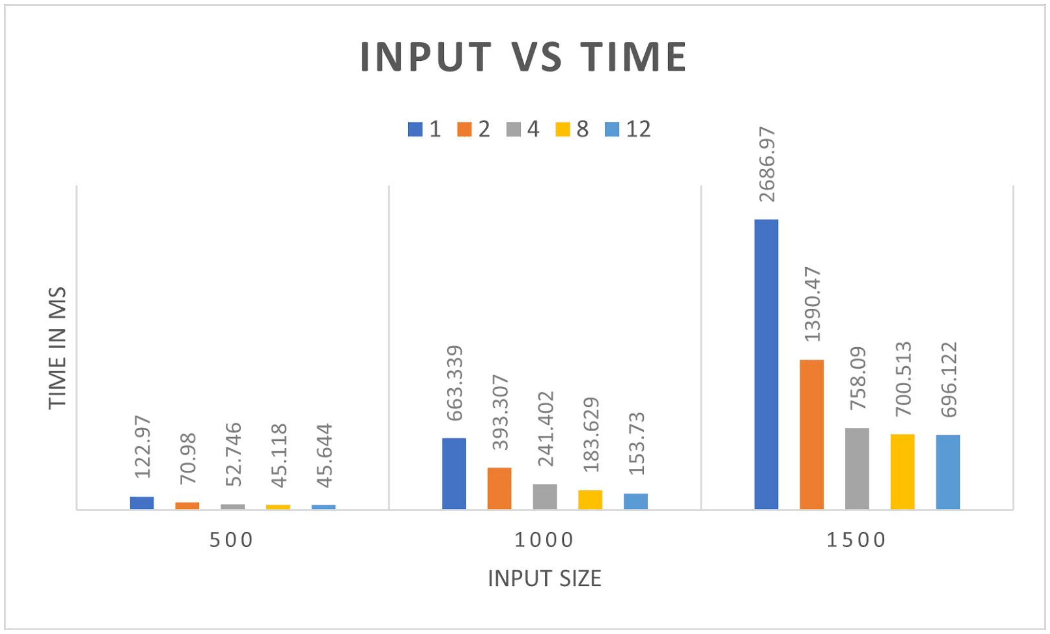
Final result will be (1,5) where all nodes are considered for constructing the tree.

The diagonal with id=0 is initiated with key probabilities.

Issues, Observations, and Results :

- The problem could have been solved using MPI, but everytime each thread would have to be given it’s required row and column.

  Hence, a shared memory model is more suited to this operation.



- Above is the chart for input sizes from 500 to 1500 where there’s no significant difference in the performance of 4,8,12 cores.

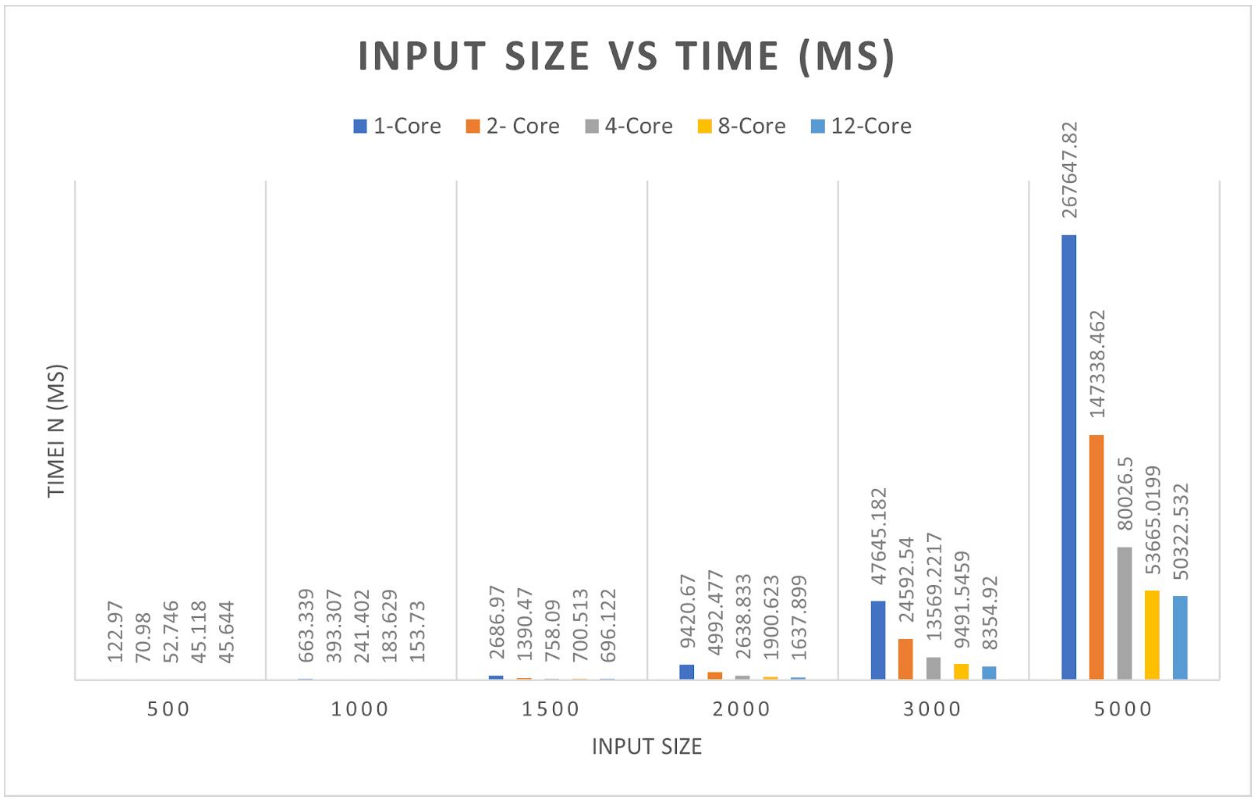
  To see a huge difference that can counter the effect of thread creation the input size had to be increased.

- Java’s system method:
  System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "X")

  Where X is the number of threads doesn’t seem to work for parallel streams; and streams consume maximum possible threads, hence all the parallel stream operations had to be used alongside a Consumer object inside a forkJoinPool object.

ForkJoinPool is an implementation of ExecutorService that has a work-stealing algorithm which helps in load balancing.
Hence, when I implemented the parallel part of the algorithm by creating threads, and joining them after every diagonal the performance was poor as compared to the load-balanced ForkJoin framework where irrespective of resource allocation every thread “works” till the barrier condition.

At an input size of 5000 we can observe it takes the sequential program ~267s whereas the same task is completed in ~50s by a 12-threaded parallel program.


INPUT SIZE VS TIME (MS)
1-Core
2- Core
4-Core
8-Core
12-Core
TIMEI N (MS)
INPUT SIZE