Felicity Escarzaga
CS 599: HPC
Assignment 4

Activity 1: This brute force algorithm runs for each query, checking each data point's `x` and `y` to see if it is in bounds of the queries `x_min, y_min, x_max, y_max`. If the point is within bounds increment the query's counter. All queries' counters are summed and then `MPI_Reduce` is used to sum the count across all data points.

Question 1: *Excluding the fact that the algorithm is brute force, what is one potential inefficiency described in the programming activity?*

Each query must read through all the data which means there will be a lot of cache-access and in turn possible cache misses which may make the algorithm more inefficient.

Table 1: Total response time, speedup, and parallel efficiency (one node).

# of Ranks (p)	Time (s)	Speedup	Parallel Efficiency	Global Sum	Job Script Name (*.sh)
1	794.217944	1	1	466380694	range_act1_fhe2
4	198.177054	4.007618077	1.001904519	469453172	range_act1_fhe2
8	100.673231	7.88906779	0.9861334738	468426414	range_act1_fhe2
12	67.515945	11.76341298	0.9802844153	469588802	range_act1_fhe2
16	51.003192	15.57192624	0.9732453902	467780256	range_act1_fhe2
20	40.79007	19.47086494	0.9735432472	467634204	range_act1_fhe2

Question 2: *Describe the performance of the brute force algorithm, does it scale well? Explain.*

The brute force algorithm scales very well with a high parallel efficiency even at 20 ranks and maintains a near perfect speedup as shown in *Fig. 1* below. This is likely because the number of queries is split between the ranks so as the number of ranks increases the number of queries per rank decreases.

Fig 1. Parallel Speedup of Brute Force approach

Felicity Escarzaga
CS 599: HPC
Assignment 4

Activity 2: In this activity the R-tree was constructed exactly as shown in the movie example. Then the search is done for each query using `tree.Search` method from the `RTree` class.

Table 2: Total response time, index construction time, and search time (one node).

# of Ranks (p)	Total Time (s)	R-Tree Construction (s)	Search Time (s)	Global Sum	Job Script Name (*.sh)
1	17.784628	2.420146	15.364482	466380694	range_act2_fhe2
4	6.318219	2.45933	3.87051	469453172	range_act2_fhe2
8	4.564927	2.55971	2.033319	468426414	range_act2_fhe2
12	3.961781	2.594259	1.394197	469588802	range_act2_fhe2
16	3.976909	2.705752	1.288269333	467780256	range_act2_fhe2
20	3.661679667	2.686295	0.9753846667	467634204	range_act2_fhe2

Table 3: Speedup and parallel efficiency of the search phase from Table 2.

# of Ranks (p)	Speedup	Parallel Efficiency
1	1	1
4	3.969627258	0.9924068146
8	7.556355889	0.9445444861
12	11.02030918	0.9183590985
16	11.92645172	0.7454032322
20	15.75222835	0.7876114176

Question 3: *Each rank constructs an identical R-tree but searches the index for different range queries. How does the time to construct the index change with increasing p? Explain this performance behavior.*

Increasing the number of ranks appears to increase the time to construct but only slightly. This is likely because while increasing the number of ranks means increasing the number of trees that need to be constructed; all ranks construct their tree at relatively the same time and rate.

Question 4: *Which implementation has better performance, the R-tree (search only time) or the brute force algorithm? Explain.*

Performance in terms of search time, the R-tree for 1 rank is about 50 times faster than the brute force search on 1 rank. 1 rank with the R-tree is also faster than running the brute force on 20 ranks. This is likely because by using the R-tree algorithm the program only needs to iterate through the queries; unlike the brute force algorithm which iterates through both the queries and all the data points for each query.

Felicity Escarzaga
CS 599: HPC
Assignment 4

Question 5: *Which has the highest parallel efficiency, the brute force algorithm or the R-tree? Why do you think the parallel efficiency varies between algorithms?*

The brute force algorithm has the highest parallel efficiency with near perfect efficiency at 20 ranks, whereas the R-tree algorithm immediately drops slightly to 0.99 at 4 ranks and goes as low as 0.74 at 16 ranks (oddly going up slightly to 0.78 at 20 ranks). The efficiency for the brute force algorithm is higher because increasing the number of ranks significantly decreases the amount of work each rank performs, whereas the search time for the R-tree is already relatively low with most of the work done in construction.

Table 4: Total response time, index construction time, and search time (two nodes).

# of Ranks (p)	Total Time (s)	R-Tree Construction (s)	Search Time (s)	Global Sum	Job Script Name (*.sh)
1	17.551803	2.418923	15.132879	466380694	range_act2_2N_fhe2
4	6.330169	2.441511	3.934171	469453172	range_act2_2N_fhe2
8	4.395925	2.465541	1.955255	468426414	range_act2_2N_fhe2
12	3.892382	2.547968	1.39078	469588802	range_act2_2N_fhe2
16	3.593397	2.573442	1.026025	467780256	range_act2_2N_fhe2
20	3.435258	2.614993	0.847799	467634204	range_act2_2N_fhe2

Table 5: Speedup and parallel efficiency of the search phase from Table 4.

# of Ranks (p)	Speedup	Parallel Efficiency
1	1	1
4	3.84652294	0.9616307349
8	7.739593557	0.9674491946
12	10.8808575	0.9067381254
16	14.74903535	0.9218147097
20	17.84960704	0.8924803521

Question 6: *Compare the speedup and parallel efficiency between Table 3 (one node) and Table 5 (two nodes). All p=20 ranks can be run on a single node (Table 3), or they can be evenly distributed between two nodes (Table 5). How does the speedup and parallel efficiency compare? Is there anything interesting about performance?*

The speedup and parallel efficiency in **Table 3** (one node) has a kink at about 16 ranks but for two nodes in **Table 5** the speedup continues in a relatively linear path at and after 16 ranks, and does not drop at 16. The difference is best shown between *fig. 2* and *fig. 3*.This

suggests that running on 1 node causes a memory bottle-neck whereas running on 2 does not.

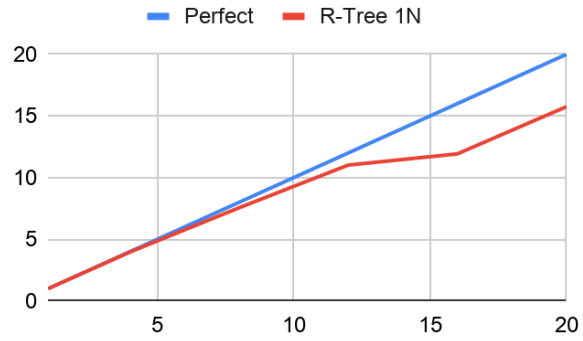

Fig 2. Perfect vs R-tree on 1 Node speedup

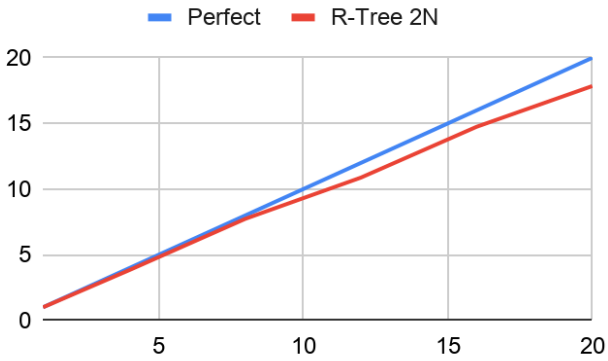

Fig 3. Perfect vs R-tree on 2 Nodes speedup

Question 7: *Explain what parameters you used in your job script. Report the results in the Tables below.*

In this experiment, instead of splitting the ranks evenly between two nodes, the ranks are split evenly between 2 sockets on one node. The parameter used in the job script was `--ntasks-per-socket=` half the number of tasks ran for that test. For example:

```
srun --ntasks-per-socket=2 -n4 range_act2_fhe2 2000000 100000
```

Table 6: Total response time, index construction time, and search time (Even socket split).

# of Ranks (p)	Total Time (s)	R-Tree Construction (s)	Search Time (s)	Global Sum	Job Script Name (*.sh)
1	17.828951	2.420246	15.408704	466380694	range_act2_E_fhe2
4	6.298545	2.422045	3.8765	469453172	range_act2_E_fhe2
8	4.59395	2.508954	2.096794	468426414	range_act2_E_fhe2

12	3.920807	2.578308	1.366905	469588802	range_act2_E_fhe2
16	3.6553	2.617349	1.050646	467780256	range_act2_E_fhe2
20	3.495293	2.64877	0.861222	467634204	range_act2_E_fhe2

Table 7: Speedup and parallel efficiency of the search phase from Table 6.

# of Ranks (p)	Speedup	Parallel Efficiency
1	1	1
4	3.974901071	0.9937252676
8	7.348697106	0.9185871383
12	11.27269562	0.9393913013
16	14.66593315	0.9166208219
20	17.89167485	0.8945837426

Question 8: *How does performance compare to your experiment in Table 7 to Tables 3 and 5? Did your new experiment outperform the 2 node experiment?*

In the experiment shown in **Table 7** and *fig 4.*, the speed up and parallel efficiency is about the same as shown in **Table 5** (2 nodes, fig 3). However, by using two sockets on 1 node the experiment outperformed the 2 node experiment by using less resources. Since the ranks were split evenly between the 2 sockets they did not have the same memory bottle-neck as running on 1 node (**Table 3**) without distributing the ranks between sockets. This indicates that memory is handled separately for each socket on a node.

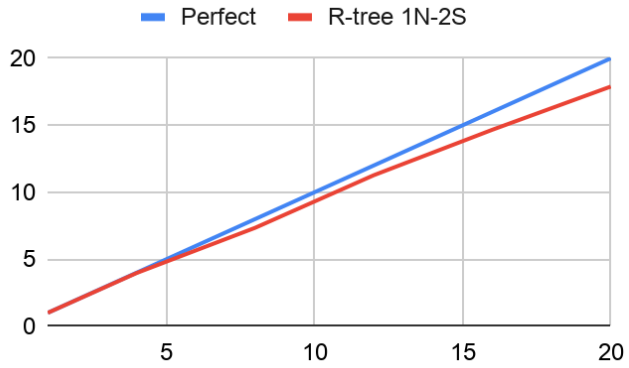

Fig 4. Perfect vs R-tree on 2 Sockets Speedup

Question 9: *Consider the case where you want to run the range query algorithm with the R-tree, and you need to run the algorithm on a cluster that is shared with one other user. Would you rather the other user be running a memory-bound algorithm or compute-bound algorithm? Explain.*

Felicity Escarzaga
CS 599: HPC
Assignment 4

Running the range query with an R-tree, which is a memory-bound algorithm, would perform poorly with another memory-bound algorithm running, since the memory access will slow the program. So in this case, it would be better to run with a compute-bound algorithm.