**Program Summary:**

For activity 1, the implementation of the program was quite straightforward. At each rank, two doubly-nested *for loops* are used. The outer loop reads a search key from the query data set and the inner loop linearly searches the entire local data set for possible matches.

Each rank computes its total number of positive search hits which I used for computing the global sum using an MPI reduction clause.

Similar to activity 1, in activity 2, search key items from the query set were used to query the R-tree that was built using the items from the data set. However, in activity 2, instead of searching using doubly nested for loops, a single for loop was used. In that single for loop, each search key from the local query set was used to construct an MBR object for querying the R-tree. The global sum of total positive search hits was calculated similarly to activity 1.

Activities for Tables 4, 5, 6, 7, and Questions 6,7,8,9 were performed by modifying the job scripts accordingly, using the same source files used in activity 2. These are placed in the folders titled activity\_3 (2 nodes) and activity\_4 (own experiments) respectively.

**Q1: Excluding the fact that the algorithm is brute force, what is one potential inefficiency described in the programming activity?**

**Ans:** One potential inefficiency is the possibly high percentage of cache misses or poor locality. As the data set is much larger than the query set across all the ranks, reading a cell value from the data set will possibly cause the query set to be discarded from the cache. As a result, for reading query values, we might get close to 100% cache misses.

| **# of Ranks (p)** | **Time (s)** | **Speedup** | **Parallel Efficiency** | **Global sum** | **Job script name** |
| --- | --- | --- | --- | --- | --- |
| 1 | 862.732006 | 1.0 | 1.0 | 466380694 | CS599\_HW4\_act1\_nprocs\_1.sh |
| 4 | 215.838894 | 3.997 | 0.999 | 469453172 | CS599\_HW4\_act1\_nprocs\_4.sh |
| 8 | 108.869117 | 7.924 | 0.991 | 468426414 | CS599\_HW4\_act1\_nprocs\_8.sh |
| 12 | 72.959191 | 11.825 | 0.985 | 469588802 | CS599\_HW4\_act1\_nprocs\_12.sh |
| 16 | 54.815149 | 15.739 | 0.984 | 467780256 | CS599\_HW4\_act1\_nprocs\_16.sh |
| 20 | 43.943365 | 19.633 | 0.982 | 467634204 | CS599\_HW4\_act1\_nprocs\_20.sh |

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

**Q2: Describe the performance of the brute force algorithm, does it scale well? Explain.**

**Ans:** The brute force algorithm’s performance in terms of response time and speedup changes significantly as the number of process ranks increases. As the parallel efficiency across different numbers of process ranks always remains close to or above 98%, we can say that the algorithm scales well with an increasing number of process ranks.

As the number of process ranks increases, the total number of queries (which is fixed), gets evenly distributed across ranks, and as a result, the total response time goes down and the parallel efficiency remains considerably high.

| **# of Ranks (p)** | **Time (s)** | **R-tree construction (s)** | **Search time (s)** | **Global sum** | **Job script name** |
| --- | --- | --- | --- | --- | --- |
| 1 | 17.620910 | 2.449388 | 15.171520 | 466380694 | CS599\_HW4\_act2\_nprocs\_1.sh |
| 4 | 6.352711 | 2.485655 | 3.890498 | 469453172 | CS599\_HW4\_act2\_nprocs\_4.sh |
| 8 | 4.624634 | 2.618411 | 2.027492 | 468426414 | CS599\_HW4\_act2\_nprocs\_8.sh |
| 12 | 4.015678 | 2.634290 | 1.401617 | 469588802 | CS599\_HW4\_act2\_nprocs\_12.sh |
| 16 | 3.796811 | 2.725207 | 1.090281 | 467780256 | CS599\_HW4\_act2\_nprocs\_16.sh |
| 20 | 3.620569 | 2.743430 | 0.892530 | 467634204 | CS599\_HW4\_act2\_nprocs\_20.sh |

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

| **# of Ranks (p)** | **Speedup** | **Parallel efficiency** |
| --- | --- | --- |
| 1 | 1.0 | 1.0 |
| 4 | 3.9 | 0.975 |
| 8 | 7.483 | 0.935 |
| 12 | 10.824 | 0.902 |
| 16 | 13.951 | 0.872 |
| 20 | 16.998 | 0.850 |

**Table 3: Parallel efficiency using the index search time (not total response time) from Table 2.**

**Q3: 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.**

**Ans:** The time to construct the RTree/index increases very slightly (in the ranges of 0.13 s - 0.018 s) with an increasing number of p when compared to the significant decreases in search time with increasing p as shown in table 2.

As the dataset sizes get smaller with increasing p values, the changes in index construction time can possibly be attributed to variance in time overheads associated with tree construction and indexing operations. Especially, if the R-tree construction has to deal with a lot of rectangles with overlapping boundaries, it would require more time to create the indexes.

**Q4: Which implementation has better performance, the R-tree (search only time) or the brute force algorithm? Explain.**

**Ans:** The R-tree (search only time0 has better performance than the brute force approach. As the brute force approach has to search a key item linearly in the entire dataset every time, its time complexity is essentially O(N^2). On the other hand, the time complexity of the R-tree is O(logMn). This is why R-tree outperforms the brute force algorithm.

**Q5: Which has the highest parallel efficiency, the brute force algorithm or the R-tree? Why do you think the parallel efficiency varies between algorithms?**

**Ans:** As observed in the data from Tables 1 & 2, the brute force algorithm has the highest parallel efficiency. I think this is because the brute-force algorithm is more parallelizable than the R-tree algorithm. When the brute force algorithm searches a key, it accesses the data array which can take advantage of the caching effect. However, as R-tree nodes are probably stored in non-contiguous memory locations, their access will not probably be able to take advantage of the caching effect. As a result, even though the time complexity of the R-tree algorithm and its response times are better than the brute-force algorithm, the brute force algorithm is a little bit more parallelizable than the R-tree.

| **# of Ranks (p)** | **Time (s)** | **R-tree construction (s)** | **Search time (s)** | **Global sum** | **Job script name** |
| --- | --- | --- | --- | --- | --- |
| 1 | 17.690915 | 2.449334 | 15.241576 | 466380694 | CS599\_HW4\_act3\_nprocs\_1.sh |
| 4 | 6.404741 | 2.465787 | 3.938952 | 469453172 | CS599\_HW4\_act3\_nprocs\_4.sh |
| 8 | 4.460015 | 2.527613 | 1.968709 | 468426414 | CS599\_HW4\_act3\_nprocs\_8.sh |
| 12 | 4.050019 | 2.594082 | 1.487537 | 469588802 | CS599\_HW4\_act3\_nprocs\_12.sh |
| 16 | 3.623462 | 2.605505 | 1.034253 | 467780256 | CS599\_HW4\_act3\_nprocs\_16.sh |
| 20 | 3.514779 | 2.667933 | 0.864584 | 467634204 | CS599\_HW4\_act3\_nprocs\_20.sh |

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

| **# of Ranks (p)** | **Speedup** | **Parallel efficiency** |
| --- | --- | --- |
| 1 | 1.0 | 1.0 |
| 4 | 3.870 | 0.968 |
| 8 | 7.742 | 0.968 |
| 12 | 10.246 | 0.854 |
| 16 | 14.737 | 0.921 |
| 20 | 17.629 | 0.881 |

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

**Q6: 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 do the speedup and parallel efficiency compare? Is there anything interesting about performance?**

**Ans:** The speedup and parallel efficiencies between Table 3 and Table 5 remain close to each other with slight differences. As the value of p increases, the parallel efficiencies in Table 3 decrease monotonically. However, the parallel efficiencies in Table 5 do not decrease/increase monotonically.

Specifically, when p =20, the speedup and parallel efficiency in Table 5 (two nodes) are higher than the corresponding entry values in Table 3 (one node). The interesting observation here is that the parallel efficiencies in Table 5 do not change monotonically as p increases monotonically.

**Q7: Explain what parameters you used in your job script. Report the results in the Tables below.**

**Ans:** In my job script, I changed the number of nodes from 2 to 4 and divided the processes evenly across each node (e.g., for nprocs = 12, –ntasks-per-node=3 as -nodes=4).

| **# of Ranks (p)** | **Time (s)** | **R-tree construction (s)** | **Search time (s)** | **Global sum** | **Job script name** |
| --- | --- | --- | --- | --- | --- |
| 1 | 17.877473 | 2.507371 | 15.370098 | 466380694 | CS599\_HW4\_act4\_nprocs\_1.sh |
| 4 | 6.392881 | 2.482491 | 3.910381 | 469453172 | CS599\_HW4\_act4\_nprocs\_4.sh |
| 8 | 4.436299 | 2.468244 | 1.968050 | 468426414 | CS599\_HW4\_act4\_nprocs\_8.sh |
| 12 | 4.503628 | 2.575576 | 1.928047 | 469588802 | CS599\_HW4\_act4\_nprocs\_12.sh |
| 16 | 3.906770 | 2.554111 | 1.374356 | 467780256 | CS599\_HW4\_act4\_nprocs\_16.sh |
| 20 | 3.407620 | 2.574159 | 0.838918 | 467634204 | CS599\_HW4\_act4\_nprocs\_20.sh |

**Table 6: Total response time, index construction time, and search time (four nodes, 1 node when p=1)**

| **# of Ranks (p)** | **Speedup** | **Parallel efficiency** |
| --- | --- | --- |
| 1 | 1.0 | 1.0 |
| 4 | 3.931 | 0.983 |
| 8 | 7.810 | 0.976 |
| 12 | 7.972 | 0.664 |
| 16 | 11.183 | 0.699 |
| 20 | 18.321 | 0.916 |

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

**Q8: How does performance compare to your experiment in Table 7 to Tables 3 and 5? Did your new experiment outperform the 2 node experiment?**

**Ans:** The performance in Table 7 outperforms the performance in Table 5 for p = 4,8,20. However, for p = 12,16, performance in Table 5 outperforms performance in Table 7.

Between Table 3 and Table 7, the performances follow a similar pattern. For p = 4,8,20, new experiment outperforms the Table 3 version. However, for p =12, 16, the Table 3 version outperforms the new experiment version.

The new experiment outperformed the 2 node experiment for some values of p, not for all values of p. I think this probably happens due to the overheads associated with cross node communications as well as the randomness of the communication network situation.

**Q9: 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.**

**Ans:** I would rather want the other user to be running a memory-bound algorithm instead of a compute-bound algorithm.

If the other user runs a compute-bound algorithm, it will be highly likely that the algorithm will be using available CPUs extensively and possibly causing a lot of inter-process/cross-core communication - thus possibly dominating the communication network. Such a situation would negatively affect the performance of the R-tree algorithm in a distributed cluster environment.