Chris Keefe
Assignment 4 - Range Queries

Activity 1: Code description

The code for this part of the exercise is quite straightforward - for each query, iterate over all points in the data checking whether they are within the query range. I took advantage of short-circuiting logical operators to reduce the number of checks required: if the point is outside of range for either dimension, the point is not counted. Hits are counted in an array indexed by query number.

Local sums are reduced into a global sum using MPI_SUM, and the longest local timing is selected using an MPI_MAX reduction.

Jobscript/run description

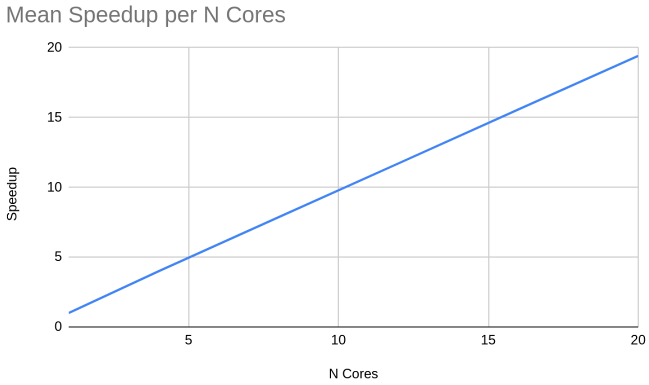
As previously, I used BASH to coordinate the running of all jobs. The sbatch.command script passes job-specific output parameters, including --job-name, --ntasks, and --output. I've included the `sbatch.command` script with my code.

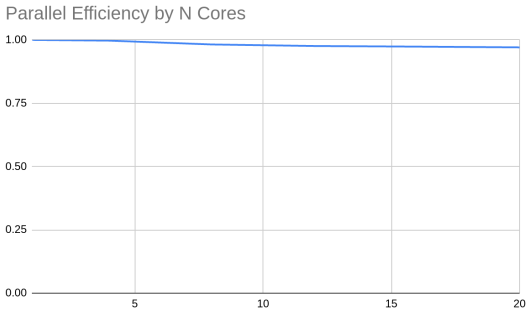
Q1: Excluding the fact that the algorithm is brute force, what is one potential inefficiency described in the programming activity?

Honestly, for a naive implementation, this seems pretty reasonable. Depending on cache size, we might be able to exploit locality by "tiling" smaller groups of queries and points. This is the only semi-reasonable idea I've come up with, though. Sorting on one coordinate would allow us to drop many queries, but that would take us away from brute force. If our system were limited by insufficient memory, we could reduce memory use by splitting data across ranks, but that would probably cost dearly in communication overhead.

	A1 Total Time (s)	A1 Speedup	A1 Par Effic	Global Sum	Jobscript
1	844.0645	1	1	466380694	jobscript.sh + sbatch.command
4	211.6208	3.9886	0.9972	469453172	jobscript.sh + sbatch.command
8	107.4345	7.8565	0.9821	468426414	jobscript.sh + sbatch.command
12	72.0658	11.7124	0.976	469588802	jobscript.sh + sbatch.command
16	54.1879	15.5766	0.9735	467780256	jobscript.sh + sbatch.command
20	43.4818	19.4119	0.9706	467634204	jobscript.sh + sbatch.command

Q2: Describe the performance of the brute force algorithm, does it scale well? Explain.
Though the algorithm is inefficient and quite slow at P=1 ranks, it scales linearly and almost perfectly through P=20. (Parallel efficiency near 1). Performance is bad at the scale tested, but scalability is excellent.


Mean Speedup per N Cores


Parallel Efficiency by N Cores

Activity 2: Code description

Here we construct an R-tree, populate it with data “rectangles” (xmin=xmax, ymin=ymax), and then query the tree once for each query, adding the number of hits returned to an array of results. Timings are slightly more complex than in activity one, but all three use the same MPI_MAX reduction pattern described as above.

Jobscript/run description

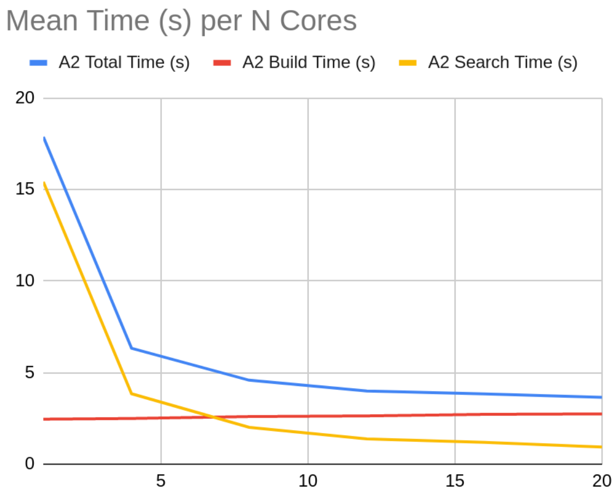
Again, sbatch.command coordinates running jobs, and helper scripts summarize the data

	A2 Total Time (s)	A2 Build Time (s)	A2 Search Time (s)	Global Sum	Jobscript
1	17.8808	2.4569	15.4239	466380694	jobscript.sh + sbatch.command
4	6.3306	2.4953	3.8481	469453172	jobscript.sh + sbatch.command
8	4.584	2.5993	2.0109	468426414	jobscript.sh + sbatch.command
12	3.9929	2.6352	1.375	469588802	jobscript.sh + sbatch.command
16	3.8354	2.7195	1.1875	467780256	jobscript.sh + sbatch.command
20	3.6474	2.7448	0.932	467634204	jobscript.sh + sbatch.command

	A2 Search Time (s)	A2 Speedup	A2 Par Effic
1	15.4239	1	1
4	3.8481	4.0082	1.0021
8	2.0109	7.6701	0.9588
12	1.375	11.2174	0.9348
16	1.1875	12.9885	0.8118
20	0.932	16.5492	0.8275

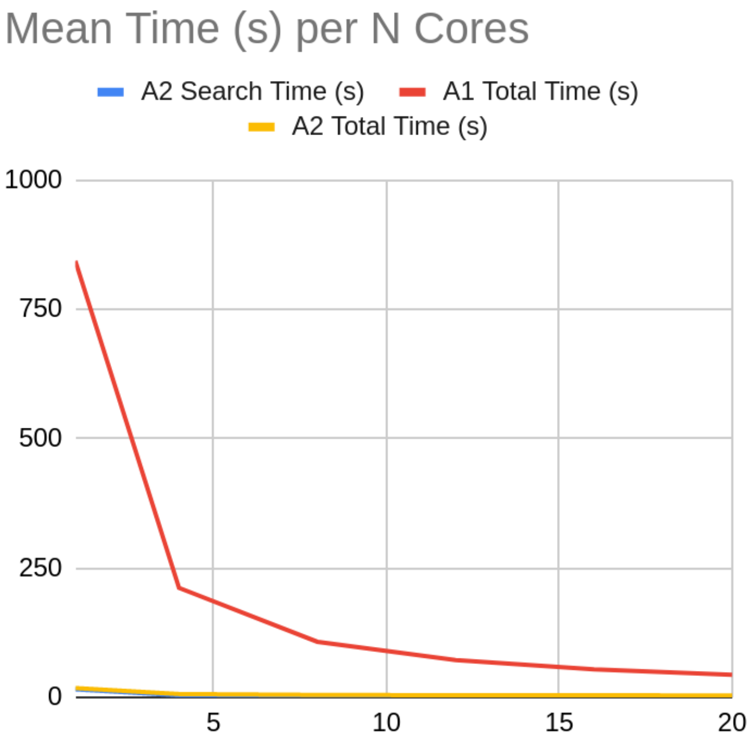
Q3: Each rank constructs an identical R-tree but searches the index for different range queries. How does the time to construct the index change with increasing p? Explain this performance behavior.

Build time is nearly constant across P cores, because each core is constructing an identical tree concurrently. The small but steady increase in overall build time as P increases may be the result of cores competing for access to memory-related resources (e.g. minor delays caused by multiple cores sharing a memory device controller or bus).



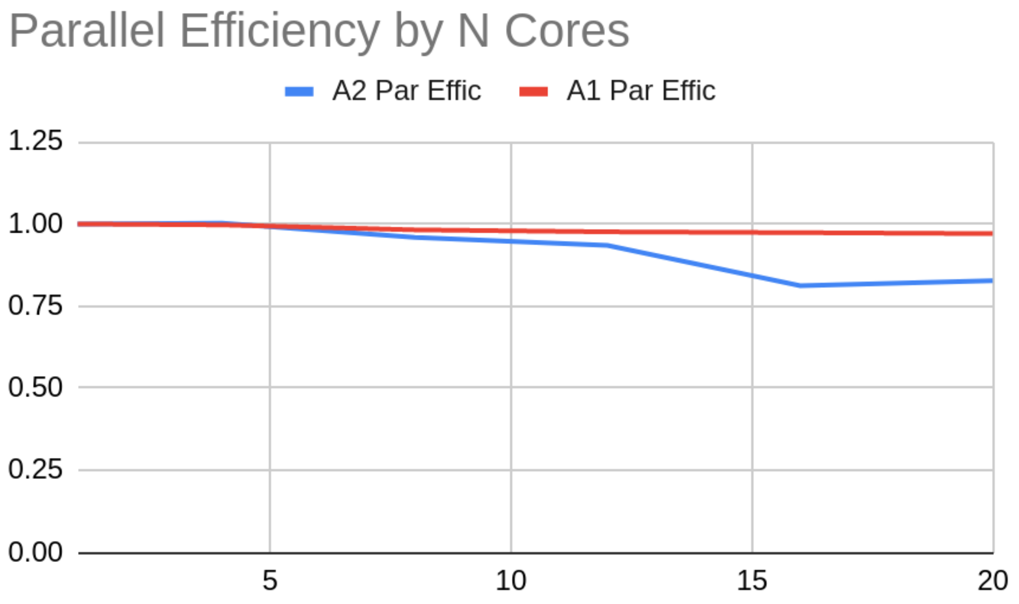
Q4: Which implementation has better performance, the R-tree (search only time) or the brute force algorithm? Explain.

The R-tree has massively better performance, with single-core speeds 2.4x faster than the naive approach's speed on 20 cores. Brute force scales well, but the baseline performance from which it scales is comparatively terrible.



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?

The brute force algorithm has better parallel efficiency. The R-tree approach reduces the overall number of range checks made enormously, but I believe it does so by requiring more indexing operations per query on average. I suspect parallel efficiency is worth with R-tree, because both algorithms perform little computation per query, and the increased number of memory reads per query creates a bottleneck.



Activity 3:

Code identical

Jobscript/run description

For this run, I had the sbatch.command script calculate the number of tasks per node, and let it pass that parameter to the jobscript's srun call.

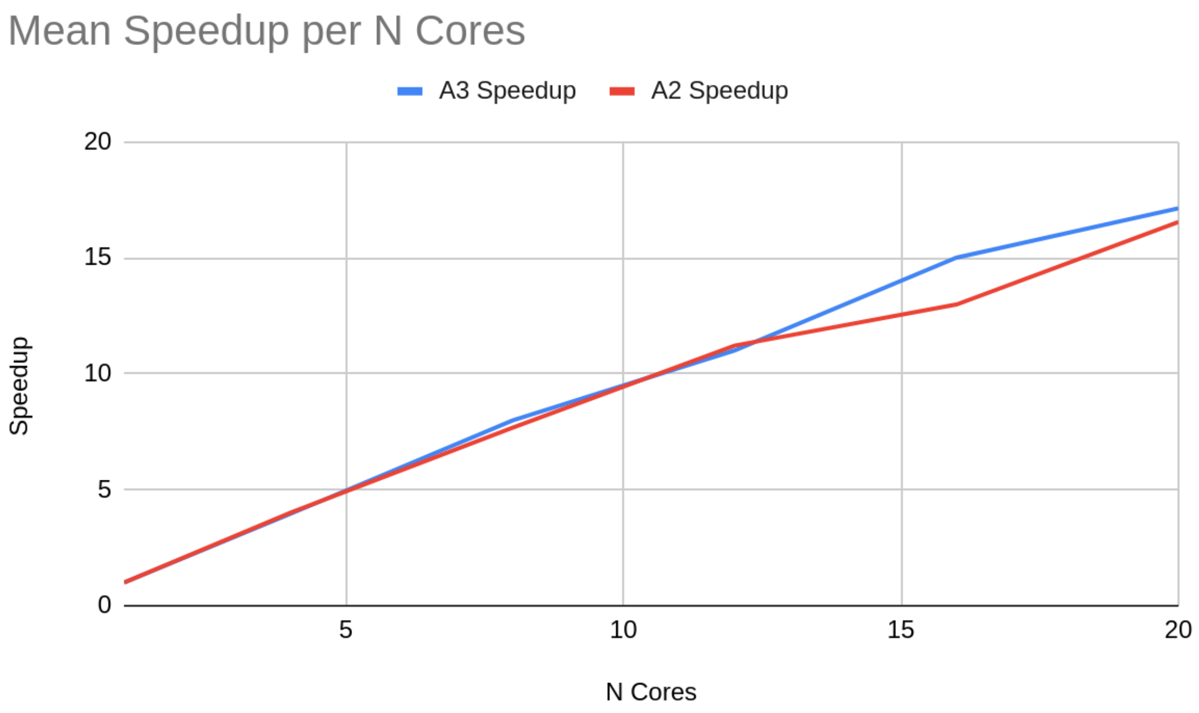
It is not possible to run on both one core and two nodes simultaneously, so I dropped P=1 from this and all following timing tables. Speedup and parallel efficiency require a baseline value, and I've chosen to use timings from activity 2's single-node single-core run for that (in orange).

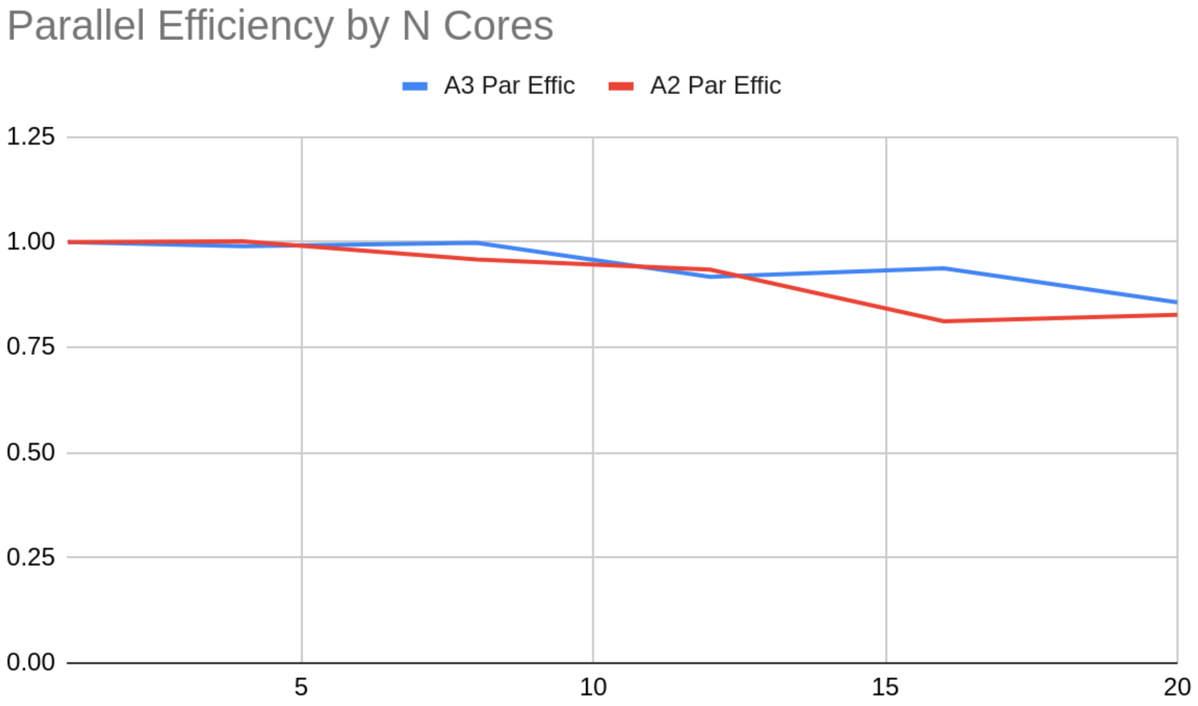
	A3 Total Time (s)	A3 Build Time (s)	A3 Search Time (s)	Global Sum	Jobscript
4	6.3494	2.4746	3.8939	469453172	jobscript.sh + sbatch.command
8	4.4159	2.5011	1.9316	468426414	jobscript.sh + sbatch.command
12	3.9547	2.5844	1.4012	469588802	jobscript.sh + sbatch.command
16	3.6117	2.6017	1.0277	467780256	jobscript.sh + sbatch.command
20	3.5397	2.6472	0.9	467634204	jobscript.sh + sbatch.command

	A3 Search Time (s)	**A3 Speedup**	**A3 Par Effic**
1	15.4239	1	1
4	3.8939	3.961	0.9903
8	1.9316	7.985	0.9981
12	1.4012	11.0076	0.9173
16	1.0277	15.0082	0.938
20	0.9	17.1377	0.8569

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 does the speedup and parallel efficiency compare? Is there anything interesting about performance?

My results for speedup and efficiency on the one-node and two-node runs were nearly identical, with each run “winning” in a roughly alternating fashion. Because the search phase doesn’t actually require any communication between ranks, I found this unsurprising. The query section timing doesn’t reflect setup or communications overhead, just same-generation cpus crunching numbers, and I suspect variation is just a product of “natural” variability in machine performance in the field.





Activity 3: Experiment description

I chose to test the hypothesis that node exclusivity has limited impact on performance, and does not provide sufficient benefits at this scale of compute to justify potential costs in fairshare and wait time.

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

I did so by dropping the --exclusive flag from all of my jobs, increasing the number of iterations called in the `sbatch.command` script from 3 to 10 to reduce the impact of outlier jobs, and considering both mean and median search times in an effort to similarly reduce the impact of outlier values.

Jobscript/run description

As above, I used BASH to coordinate the running of all jobs. The sbatch.command script passes job-specific output parameters, including --job-name, --ntasks, and --output. I've included the `sbatch.command` script with my code.

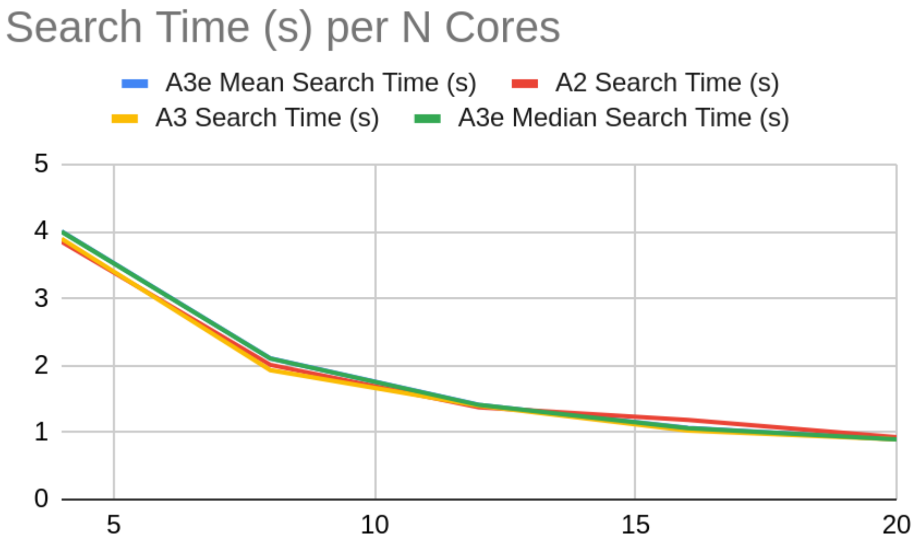
Mean and median timings were very similar, indicating there was limited outlier effect in these timings.

	A3e Total Time (s)	A3e Build Time (s)	A3e Mean Search Time (s)	A3e Median Search Time (s)	Global Sum
4	6.5423	2.5426	4.004	3.998	469453172
8	4.7424	2.6393	2.1075	2.1062	468426414
12	4.0533	2.6436	1.4146	1.4116	469588802
16	3.7427	2.6772	1.0719	1.0671	467780256
20	3.6493	2.7627	0.8998	0.9003	467634204

	A3e Search Time (s)	A3e Speedup	A3e Par Effic
1	15.4239	1	1
4	4.004	3.8521	0.963
8	2.1075	7.3186	0.9148
12	1.4146	10.9034	0.9086
16	1.0719	14.3893	0.8993
20	0.8998	17.1415	0.8571

- **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?**

Performance under these experimental conditions was nearly identical to performance under exclusive conditions with the experimental treatment outperforming the single-node exclusive at p = 16, likely due to natural variation in performance across cores. Mean and median measurements were nearly identical for the experimental group (Δ < 0.004 S for all P), indicating outlier effects were small. For this scale of computation, this seems to bear out the idea that the costs in fairshare (and possibly wait time as well - all test runs started immediately so I don't have data on this) makes node exclusivity unnecessarily costly for non-benchmarking tasks. More robust sampling would be needed (especially during periods with higher load on the system) in order to come to any robust conclusions.



- **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.**

Compute bound competition would likely be preferable. This algorithm has limited compute costs, and though it seems to scale well across all of the tests I've done, I suspect it is more likely to be memory-bound than compute-bound. If we assume that both programs must be running at the same time for this question, adding load to a node's memory could slow down our run, which would be unfortunate.

This question is somewhat hard to answer, though. If we drop the assumption that both users always get the same resources, a second user with a compute-bound algorithm would probably increase the number of nodes they request, making it difficult for us to scale up our algorithm, and potentially negating its very good parallel efficiency by increasing our wait time to access compute resources.