Report CPD - Group 2

Small problems

The first issue to be tackled was the point generation. We opted to generate the same set of points in every process since making a single process handle point generation and then broadcast to others would present a bigger overhead. Having the same dataset in every process allowed us to minimize the communication data by using indices to refer to the points that were being worked with.

We decomposed our problem into 2 phases as it had the best performance in the first delivery. The first phase would team up different processes to compute the tree levels that had **fewer nodes than the number of processes**. In the second phase, each processor would be responsible for a subtree of the global solution, which was the same problem solved in the first delivery. As such, it used the previously developed algorithm to solve its part of the problem.

The first phase goes as follows: Every process starts in the same team and the process with the lower id of each team is called the *master*. The *master* is the only process that knows the set of points, the working set (*wset*), the team is working with, as we will explain later. The master process starts by finding the furthest points of the current *wset*, *a* and *b*, and sends them to every peer. The orthogonal projections are computed to find the central point of the *wset*, so the *master Scatters* the working set among the team, and each peer proceeds to compute the semi-orthogonal projections of its subset and *Gather* back in the *master*. Being the only process with every orthogonal projection, the *master* calculates the median and center of the *wset* and broadcasts the latter to the team. Each peer would then find the point of its subset that was farthest away from the received center. These distances are then reduced to the *master*, who gets the radius of the current node. After generating the tree node, the team is split into 2 subgroups (by using the *MPI_Comm_split* primitive). Each computes one of the two newly generated *wsets*. The master of the new team receives the new *wset* from its previous master and both teams proceed as described in the beginning of this paragraph. If the team's size is one, proceed to phase 2. After finishing phase 2, each process would then print the locally computed nodes and, due to how MPI handles prints, everything would appear in the same terminal.

This solution only requires synchronization when the master is communicating with his teammates, since the data involved in the computation of nodes is partitioned among the participating processes.

Big problems

We then created another algorithm that solved problems that would not fit into the memory of a single machine. This solution required more communication and coordination, which introduced an overhead in its performance. However, this cost is necessary to solve problems that do not fit into a single machine. Our project chooses the "small" solution if the problem is small enough, in an attempt to minimize unnecessary overheads. We set a static threshold above which the problem would be solved using this second solution.

We started by modifying the *gen_points* function: each process generates every point because they share the same seed for randomization, although only storing a disjoint subset. We used the

distributed approach for block decomposition since it minimized the required communication when partitioning the problem.

To find the 2 furthest points, the *master* broadcasts its first (arbitrary) point and every process calculates the local furthest point in its local *wset*. The team then performs an *All_reduce* to find the furthest point *a*. This required a custom MAX operation. The team repeats this process to find *b*. Then, every process calculates the orthogonal projections for its local *wset*.

The team would then find the median. The quick-select algorithm used in the previous solutions would require too much communication and would create imbalances in point distribution across processes. We opted to use the Odd-Even Transposition Sort algorithm since it provided good scalability and guaranteed the same point distribution. This algorithm required each process to have the capacity to store the points of two processes, which reduced the maximum problem size to half. After having the sorted array, it was trivial to find the median. If the complete array had an even number of points, the owner of the point that defined the upper median value would receive the lower median value from its neighbor and calculate it.

Finally, in order to calculate the radius, we used a similar process to the one in the solution above. A simple *Reduce* to one of the processes allowed the biggest radius to be found. The process can now be repeated by splitting the network in half and repeat from the beginning. Just like above, once all nodes store all of the points necessary for their own partition, they use the solution of the first delivery.

Integration with OpenMP

After both solutions were done, we decided to take advantage of OpenMP and further parallelize the work the processes had to do. More specifically, we applied tasks to the phase where each process was alone in its network and had to solely calculate its own nodes of the tree, just like the solution we presented in stage 1.
This, however, meant that we needed to run our tests with one process per node (using the *--tasks-per-node* flag) and 4 threads per process (*--cpus-per-node*). This, as can be seen in the next section, further optimized the solution, but just for a small number of processes.

Experimental Results and Discussion

During this analysis, we consider each thread of a single machine as a **processing unit** (PU). We also refer to machines as nodes. Do not confuse them with the tree nodes mentioned in previous sections.

After fine-tuning our solutions, we were able to run them in the RNL cluster, which contained several machines with i5-4460 CPU @ 3.20GHz quad-core processors and 16GB RAM. Table 1 shows the measured times and calculated speedup and efficiency for several problem sizes for the smaller solution. Table 2 contains the results of testing the “big” version using 4 threads in each processor.

This first solution got speedups above 1.43 for more than 1 PU, reaching a maximum speedup of 8.39 for 32 nodes with 1 thread each (32 PUs) resulting in an efficiency of 0.26. By looking carefully and comparing the results of different configurations, we noted that the efficiency tends to decrease as we increase the number of PUs, however, the speedup still tends to increase with it.

This means that the computational power that comes with more machines compensates the communication overheads they introduce.

#dims		20			3			4			3			4		
#points		1000000			5000000			10000000			20000000			20000000		
#threads	#procs	t(s)	S	ε	t(s)	S	ε	t(s)	S	ε	t(s)	S	ε	t(s)	S	ε
SERIAL		6.00	1.00	1.00	17.60	1.00	1.00	42.40	1.00	1.00	91.40	1.00	1.00	96.80	1.00	1.00
1	1	7.30	0.82	0.82	19.50	0.90	0.90	49.90	0.85	0.85	98.90	0.92	0.92	107.40	0.90	0.90
	2	4.20	1.43	0.71	10.80	1.63	0.81	27.70	1.53	0.77	56.40	1.62	0.81	62.50	1.55	0.77
	4	2.70	2.22	0.56	6.50	2.71	0.68	17.80	2.38	0.60	35.50	2.57	0.64	40.20	2.41	0.60
	8	1.80	3.33	0.42	4.20	4.19	0.52	10.80	3.93	0.49	21.80	4.19	0.52	23.90	4.05	0.51
	16	1.50	4.00	0.25	3.30	5.33	0.33	7.80	5.44	0.34	15.20	6.01	0.38	16.60	5.83	0.36
	32	1.30	4.62	0.14	2.90	6.07	0.19	6.70	6.33	0.20	10.90	8.39	0.26	14.50	6.68	0.21
	64	1.30	4.62	0.07	3.10	5.68	0.09	6.40	6.63	0.10	11.60	7.88	0.12	12.50	7.74	0.12
4	1	2.80	2.14	0.54	7.10	2.48	0.62	17.30	2.45	0.61	33.40	2.74	0.68	38.90	2.49	0.62
	2	2.00	3.00	0.38	5.20	3.38	0.42	11.90	3.56	0.45	23.10	3.96	0.49	24.80	3.90	0.49
	4	1.60	3.75	0.23	4.00	4.40	0.28	9.00	4.71	0.29	17.00	5.38	0.34	18.20	5.32	0.33
	8	1.40	4.29	0.13	3.50	5.03	0.16	7.50	5.65	0.18	13.90	6.58	0.21	15.00	6.45	0.20
	16	1.70	3.53	0.06	5.60	3.14	0.05	11.70	3.62	0.06	19.30	4.74	0.07	23.40	4.14	0.06
	32	1.50	4.00	0.03	4.50	3.91	0.03	9.70	4.37	0.03	18.70	4.89	0.04	19.20	5.04	0.04
	64	1.40	4.29	0.02	3.80	4.63	0.02	8.00	5.30	0.02	15.70	5.82	0.02	30.20	3.21	0.01

Table 1: Small version

As the problem size gets bigger, the speedup (and consequently the efficiency) slightly increases for the same number of PUs. This happens because the time saved from parallelizing the calculations overcomes the time gained from solving bigger problems (more calculations). This gain was not verified in the leftmost test. This may hint that there is a maximum problem size for these improvements. This can also happen because the leftmost problem operates on the same number of nodes with one extra dimension and our solutions were not designed to scale with increasing dimensions.

We can also observe different speedups for the same number of PUs. For example, the measured speedups for 16 processors with 1 thread differ from the speedups of 4 processors with 4 threads (16 PUs) differ. We can also note that we get higher speedups with fewer threads for several measurements. This result was unexpected since having more threads in fewer machines would reduce the communication overhead imposed by this algorithm and would lead to higher speedups. We hypothesize that this happens because the RNL cluster allocated different processes to the same node when working with a single thread, reducing the communication overheads among some processes.

After running the same set of tests enforcing only 1 process per node, this hypothesis seems to be correct. For example, with 16 or more processes, the times were significantly higher, greatly reducing our speedup. We also observed that for 4 threads there was an abnormal increase in time for every processor. Since this leap is not verified for 1 thread, we suspect that this may have happened due to a different CPU usage while those configurations were running.

The big solution had lower speedups, having a minimum of 0.11 (ironically with the highest number of processors) and a maximum of 4.37 resulting in an efficiency of 0.55 for 2 processors with 4 threads. As with the previous solution, the speedup increases with the problem size, which again indicates that the adopted parallelization was advantageous.

In this version, we notice increasing computation times as we increase the number of processors. Interestingly enough, we found that the times for few processors (up until 4) were

close to the ones obtained with the small solution for every problem size, however, this gap grew as the number of processes increased. These measures suggest that the overhead comes from the extra communication between nodes in the second solution. The time jump that is observable with 32 and 64 processors may be explained by the physical distance of the used computers in the cluster: This number of processors requires different machines of different laboratories and this may create an extra overhead in communication.

#dims		20			3			4			3			4		
#points		1000000			5000000			10000000			20000000			20000000		
#threads	#procs	t(s)	S	ε	t(s)	S	ε	t(s)	S	ε	t(s)	S	ε	t(s)	S	ε
SERIAL		6.00	1.00	1.00	17.60	1.00	1.00	42.40	1.00	1.00	91.40	1.00	1.00	96.80	1.00	1.00
1	1	7.30	0.82	0.82	19.30	0.91	0.91	49.30	0.86	0.86	98.10	0.93	0.93	107.10	0.90	0.90
	2	5.00	1.20	0.60	10.10	1.74	0.87	27.10	1.56	0.78	49.10	1.86	0.93	59.30	1.63	0.82
	4	5.80	1.03	0.26	8.60	2.05	0.51	22.30	1.90	0.48	44.20	2.07	0.52	52.10	1.86	0.46
	8	7.00	0.86	0.11	7.50	2.35	0.29	18.10	2.34	0.29	29.00	3.15	0.39	45.70	2.12	0.26
	16	6.20	0.97	0.06	7.70	2.29	0.14	17.00	2.49	0.16	37.20	2.46	0.15	35.30	2.74	0.17
	32	15.50	0.39	0.01	53.10	0.33	0.01	15.70	2.70	0.08	41.70	2.19	0.07	264.60	0.37	0.01
	64	60.40	0.10	0.00	55.90	0.31	0.00	136.60	0.31	0.00	213.00	0.43	0.01	270.40	0.36	0.01
4	1	2.60	2.31	0.58	6.50	2.71	0.68	17.40	2.44	0.61	29.60	3.09	0.77	32.90	2.94	0.74
	2	3.00	2.00	0.25	5.20	3.38	0.42	12.10	3.50	0.44	20.90	4.37	0.55	23.90	4.05	0.51
	4	4.60	1.30	0.08	6.10	2.89	0.18	14.20	2.99	0.19	28.40	3.22	0.20	29.00	3.34	0.21
	8	5.40	1.11	0.03	6.10	2.89	0.09	14.40	2.94	0.09	26.00	3.52	0.11	29.20	3.32	0.10
	16	5.80	1.03	0.02	6.80	2.59	0.04	14.90	2.85	0.04	29.20	3.13	0.05	30.00	3.23	0.05
	32	6.10	0.98	0.01	6.40	2.75	0.02	15.10	2.81	0.02	26.50	3.45	0.03	30.30	3.19	0.02
	64	54.50	0.11	0.00	52.20	0.34	0.00	129.40	0.33	0.00	207.00	0.44	0.00	257.70	0.38	0.00

Table 2: Big version

Overall we were surprised by the low efficiencies measured by our solutions, especially the ones with many processors. This indicates that the parallelization was not the most efficient and that there are many things that we could improve in future iterations of this project.

We believe communication and synchronization between processes are the main cause of overhead in our solutions. The usage of synchronized MPI calls may cause processes to wait unnecessarily for data to be sent or received. We tried unsuccessfully to use asynchronous calls and believe that an asynchronous approach would greatly improve the solution performance. There could also be ways to minimize the communication between processes.

Another optimization that would be interesting to explore is finding where it is best to divide the work among nodes. Splitting this work has the additional cost of extra communication and this tradeoff has to be well managed. To do this, we would create different versions that would split/not split some parts of the algorithm, benchmark them and pick the best approach.

As with the previous delivery, it could be beneficial to invest in a better parallelization of the computation of the first levels of the tree. These levels have the biggest data sets and every level below them depends on it, making them the most critical points of this algorithm. The results also support this hypothesis since bigger trees have more levels that attenuate this bottleneck, as seen in the previous delivery.

Finally, it is important to note that the built solutions were designed for a number of processors that is a power of 2. This greatly simplified the division of data and partitions since it followed the way the tree was divided itself. For a different number of processes, the results are unpredictable and the algorithm may not work.