Instituto Superior Técnico

Parallel and Distributed Computing

Ball Algorithm

(Serial + MPI)

Group 1

Sara Machado

86923

Rafael Figueiredo

90770

Ricardo Grade

90774

1. Serial Version

1.1. Implementation Details

Our implementation builds the Ball Tree recursively.
First, it must be discovered all Points Projections in the Line ab, to calculate them in the most efficient way possible, we figured that the Point $b - a$ is required for the calculation of all Projections, therefore we keep it for reuse in the following ones.
Then the Center needs to be identified, the way it is being reached is by ordering the Projections by their first coordinate, choosing the one in the middle, or making the midpoint of the two Projections in the middle. For this calculation, our implementation is using the MergeSort algorithm, the reason for not using QuickSort already built into C, is because MergeSort seemed to be more efficient in our tests, it makes sense because MergeSort has a worst-case complexity of $O(n * log(n))$, which is better than QuickSort's $O(n^2)$.
So, it is time to calculate the Radius, for that it is necessary to find the point furthest from the Center, to make this discovery more efficient, instead of comparing real distances, our implementation compares square distances, doing this, the calculation of the square root is avoided.
After that, the Points are separated into two subgroups that will give rise to two other iterations of this recursive.
Also, the allocation of points to an array is being carried out in a single malloc, to minimize malloc calls, and all resources that are no longer needed are freed up in order not to overload the computer's main memory.

2. MPI Version

2.1. Approach

To fit more points than a computer can load, our implementation distributes the points evenly across all processes. In addition, all processes also calculate the initial point that is required for the calculation of Point a. Each process calculates its furthest point from the initial point, then exchanges it with each other, and finally, they adopt the furthest between them. The same process is applied to Point a to calculate Point b.
After that, each process calculates its Projection Points in relation to Line ab, orders them by their first coordinate and sends all the first coordinates of the projections already ordered to the team leader. That, in turn, collects all these partial ordered vectors, finds its median linearly, transmits it to each team member and requests the point(s) it requires from them to calculate the center. Each team member divides his set of points into left and right subsets according to the median, the leader calculates the center with the requested points, transmits to each team member, each team member calculates the distance between the center and his point further away, sends to the leader, and the leader takes the longest distance as the radius.
At this stage, it is necessary for this team to split up so that one team is in charge of the left subset and another of the right subset, the way they do it will be explained in the next section.

2.2. Problem Decomposition

In order to divide the workload between the processes, the system establishes teams. When the program starts, all processes that have been assigned points will belong to a single team, and each process on that team will have a specific id that is provided in ascending order from its own process number. The team will then iterate the algorithm to their respective points and instead of continuing their left and right iterations, the processes will exchange points in order to better parallelize the work between the teams, making them independent of each other. Processes with even ids will send their right subset to their successor, and processes with odd ids will send their left subset to their predecessor. If the total number of cases is odd, the last one will not receive the left side and will send the right side to its predecessor. Each process exchanges points with its neighbor in order to maintain the initial order necessary for the algorithm correctness.
After the exchange of points, two new teams are created, one for the left subset and another for the right subset. This division will continue until the team has only one member, at this stage, the omp will be used to divide subsequent subsets into different threads.

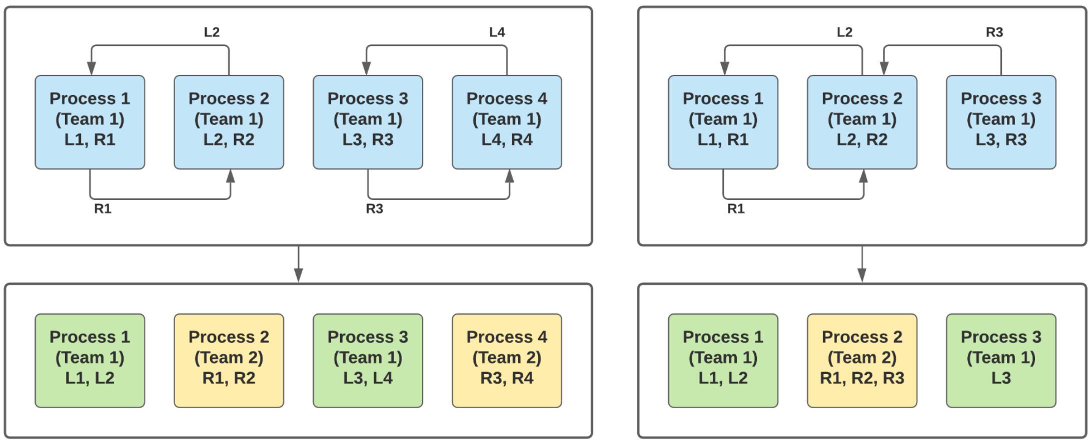
L2
L4
Process 1 (Team 1) L1, R1
Process 2 (Team 1) L2, R2
Process 3 (Team 1) L3, R3
Process 4 (Team 1) L4, R4
R1
R3
Process 1 (Team 1) L1, L2
Process 2 (Team 2) R1, R2
Process 3 (Team 1) L3, L4
Process 4 (Team 2) R3, R4
L2
R3
Process 1 (Team 1) L1, R1
Process 2 (Team 1) L2, R2
Process 3 (Team 1) L3, R3
R1
Process 1 (Team 1) L1, L2
Process 2 (Team 2) R1, R2, R3
Process 3 (Team 1) L3

Figure 1*: Teams division with an even and odd number of teammates respectively*

2.3. Synchronization Concerns

Before each recursive call, the left and right subsets are switched to their respective processes and the new teams for the next recursion are calculated. Since some processes may have completed their calculation, that is, they have no more points to calculate, a verification message is sent to all team members. This message aims to inform the team which processes are still running, that is, they still have points to be computed. Those that are no longer working will be removed from the team, as they will no longer be part of the next iteration of the algorithm.
At the end of the recursion, a tree node is created. As only one process can create the tree node in each iteration, a problem arises, as the information needed to create a node can be spread across other processes. To solve this problem, the leader of the left team will be responsible for the generation of the new node and for that, the right team leader waits to send the necessary information after finishing computing his side of the tree.

When a process becomes a single member of a team, it no longer has other processes to communicate with and will take advantage of the available threads to speed up the tree calculation. Using OMP to manage the available threads, each new recursive call to the left side of the tree will be executed in a new task, as long as there are still threads available. As mentioned above, creating a node requires information from both sides of the tree and so our implementation uses the *taskwait* pragma to wait for that task to finish and provide the necessary information for creating the node.
After the construction of the tree, it is necessary to print it. Since different nodes will be created by different processes, it is necessary to use a synchronization mechanism that allows all processes to print nodes created neatly. The process with a rank of 0 prints its nodes and notifies the node with the next rank after it finishes printing. The following nodes will wait for the rank below them to finish and then follow the same process, right down to the last process, which no longer needs to notify anyone.

2.4. Load Balancing

In case the number of processes assigned is a power of 2, the number of points present in each process will be perfectly balanced, thanks to our implementation decomposition explained previously. Otherwise, there will eventually be decompositions between an odd number of processes, which will result in one more subset being assigned to one process and one less to another, compared to the others.
In addition, in order to take advantage of how many threads have been made available to the program, our implementation tracks the number of threads available in each recursion, which allows us to reason whether a new thread should be created to divide its work, or whether the creation would only cause the threads to struggle with each other.

2.5. Performance Analysis

2.5.1. Computer Details

These results were produced in the Técnico's cluster, which contains several laboratories with several computers with 4 threads each.

2.5.2. Results

- Below ***Table 1*** illustrates the time that our program took to execute with the following arguments in both versions with a varying number of processes and threads.
- ***Table 2*** illustrates the obtained speedups of the parallel version with different number of processes and threads when comparing with the execution using only one process with the same amount of threads.

When running with processes 32 and 64, it was expected that they would always cause the greatest speedup, however in our results this is not the case.
This can be justified by the fact that communication between computers in different laboratories is slower than between computers in the same laboratory. As we use more processes, Técnico's cluster will have to supply more computers from different laboratories, since each laboratory has a different and finite number of computers. The increase in overhead caused by this communication will be greater than the time that these processes take to calculate the tree by themselves. This effect can also be observed, to a lesser extent,

when running the program with 16 processes in the cluster, depending on how the processes are distributed among the laboratories, the speedup will be variable.

In addition to these cases, as you can see in the tables below, when the number of points and the number of processes increases, the speedup also increases, which suggests that the more points are used in the calculation, the more work will be divided between processes, resulting in increased speed.

When comparing execution with 1 thread versus 4 threads, the results below show that as the number of points and the number of processors increases, speedups also increase, as each processor has more threads to speed up the final calculation phase, when there are no more team members.

Time (Seconds)															
Arguments	Serial	OMP 1 Thread							OMP 4 Threads						
		MPI Number of Processes							MPI Number of Processes						
		1	2	4	8	16	32	64	1	2	4	8	16	32	64
2 5 0	0.0	0.1	0.1	0.1	0.1	0.2	0.2	0.3	0.1	0.1	0.1	0.1	0.1	0.2	0.3
2 8 0	0.0	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.1	0.1	0.1	0.1	0.2	0.2	0.2
20 1000000 0	5.7	5.9	3.8	2.4	1.6	1.4	1.4	2.2	6.1	3.9	1.5	1.4	1.1	1.4	2.5
3 5000000 0	16.2	18.6	10.0	4.7	3.6	2.4	2.7	4.1	19.0	8.6	3.4	2.5	2.0	2.4	3.8
4 10000000 0	39.6	41.2	22.1	12.1	7.5	4.9	5.4	8.2	42.1	22.6	7.1	5.2	4.2	4.8	8.1
3 20000000 0	82.2	91.6	47.8	25.4	15.2	9.8	10.0	14.5	93.3	48.7	14.1	9.9	7.6	8.9	14.6
4 20000000 0	88.9	92.1	48.5	26.3	15.7	10.4	10.5	14.9	94.5	49.6	12.8	9.4	7.8	9.6	15.0

Table 1

Speedup												
Arguments	OMP 1 Thread						OMP 4 Threads					
	MPI Number of Processes						MPI Number of Processes					
	2	4	8	16	32	64	2	4	8	16	32	64
20 1000000 0	1.6	2.5	3.7	4.2	4.2	2.6	1.6	4.1	4.4	5.5	4.4	2.4
3 5000000 0	1.9	4.0	5.2	7.8	6.9	4.5	2.2	5.6	7.6	9.5	7.9	5.0
4 10000000 0	1.9	3.4	5.5	8.4	7.7	5.0	1.9	6.0	8.1	10.0	8.8	5.2
3 20000000 0	1.9	3.6	6.0	9.3	9.2	6.3	1.9	6.6	9.4	12.3	10.5	6.4
4 20000000 0	1.9	3.5	5.9	8.9	8.8	6.2	1.9	7.4	10.1	12.1	9.8	6.3

Table 2