Instituto Superior Técnico

Parallel and Distributed Computing

Ball Algorithm

(Serial + OMP)

Group 1

Sara Machado	**Rafael Figueiredo**	**Ricardo Grade**
86923	**90770**	**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. OMP Version

2.1. Approach

During the transition to the parallel version, we first tried to parallelize everything in our program that could be done by different threads at the same time, but we quickly realized that this would not be a viable solution to our problem, due to the constant creation and destruction of threads making our program slower than the serial version.
We then understood that a better solution, since we are doing a recursive where each call will execute independently of the others, would be to assign tasks for each subdivision of points.
Also, in order not to create threads that fight each other and therefore delay our program, we only create threads until a certain depth is reached.
Given a depth D and a number of threads N:

- If $2^D < N$: Then, a Task is created to perform the subsequent iterations of the left branch, and the current one will perform the iterations of the right branch.
- Else: The current Task will perform the subsequent iterations of both branches.

2.2. Problem Decomposition

Our first approach was to parallelize the calculation of the projections and also the merge sort, but little or no improvement was achieved with this implementation, the reason for this was because the overhead of creating and destroying threads was happening in each iteration of our recursive.
So we thought about the fact that each recursive call does not depend on another one, and then we could take advantage of that, parallelizing our entire recursive iteration, using tasks, this has brought a significant improvement to our program over the serial version.
After that, we understood that, if tasks were being created for each call, there would be a point where there would be more tasks than threads and this could bring an overhead to our program, where the threads fight each other, and so we introduced the notion of depth, where we track our depth, allowing us to stop creating the task at a certain depth, as we saw in **2.1**.

2.3. Synchronization Concerns

Each recursive call receives arguments that were created only for that specific call, so there is no need for critical synchronization regions, as they are interacting with different data.
Our implementation, however, uses the *taskwait* pragma to wait for the task that performs the subsequent iterations of the left branch, because to create the node that is returned by the recursive, the head node that is returned by the left branch is necessary.
Furthermore, our implementation also uses the *atomic* pragma in the creation of the node, to increase the number of nodes already created, necessary for the program output.

2.4. Load Balancing

In order to take advantage of how many threads have been made available to the program, our implementation tracks its recursion depth, which allows it to identify how many tasks have been created and decide whether a new one should be created to divide up its work or not.

2.5. Performance Analysis

2.5.1. Computer Details

These results were produced in the Lab PC *lab13p9*:

- **Processor:** Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz
- **RAM:** 15.6G
- **CPUs:** 4
- **Threads:** 4

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 threads.
- ***Table 2*** illustrates the obtained speedups of the parallel version with different threads when comparing with the execution using only one thread.

Time (Seconds)					
Arguments	**Serial**	**OMP 1 Thread**	**OMP 2 Threads**	**OMP 4 Threads**	**OMP 8 Threads**
2 5 0	0.0	0.0	0.0	0.0	0.0
2 8 0	0.0	0.0	0.0	0.0	0.0
20 1000000 0	5.7	5.7	3.4	2.3	2.4
3 5000000 0	16.2	16.3	9.3	6.2	6.9
4 10000000 0	39.6	39.7	22.8	15.3	16.4
3 20000000 0	82.2	82.3	46.5	30.9	32.4
4 20000000 0	88.9	88.6	50.9	33.8	35.6

Table 1

Speedup			
Arguments	**OMP 2 Threads**	**OMP 4 Threads**	**OMP 8 Threads**
20 1000000 0	1.7	2.5	2.4
3 5000000 0	1.8	2.6	2.4
4 10000000 0	1.7	2.6	2.4
3 20000000 0	1.8	2.7	2.5
4 20000000 0	1.7	2.6	2.5

Table 2

Note: The computer where the results were collected has only 4 threads. Therefore, increasing the number of threads further than that only results in loss of performance, due to the overhead of threads creation and fighting each other.

As it can be seen, the OMP version has a speedup of approximately 1.8 and 2.7 when executing with respectively 2 and 4 threads. This illustrates that the program has an increasing speedup according to the amount of threads available. However, it does not increase linearly, this can be due to the fact that creating more threads can introduce an overhead to the execution of the program, furthermore, the amount of work assigned to each thread is not exactly the same.