Ball Tree Construction

Clara Gil (97070), José Brás(82069), and Miguel Barros(87691)

Instituto Superior Técnico

Abstract. We describe the implementation of a parallel Ball Tree construction algorithm in a Distributed-Memory environment. We outline the approach used for parallelization, the problem decomposition, the synchronization concerns and how load balancing was addressed. We conclude by presenting and discussing the performance results obtained.

1 Introduction

The Ball Tree is a space partitioning data structure for organizing points in a multi-dimensional space. It's particularly useful for a nearest neighbor search. Building a ball tree for a given set of points is an expensive computation if the number of points is large. This report describes the implementation of a parallel algorithm for constructing a ball tree in a distributed memory architecture, using OpenMPI.

2 Overview

This section describes the baseline serial implementation of the ball tree construction algorithm. The algorithm takes as input a set of **N** points, each with **D** dimensions and prints to *stdout* the ball tree constructed from the point set using the algorithm described in Section 2.1.

2.1 Algorithm

The stages of the algorithm are:

1. **Initialization**: We start by allocating the memory to store the ball tree, input points and all auxiliary buffers. It is also in this stage where we call the routine provided by the faculty to get the initial point set.
2. **Calculation of a and b**: We obtain points a and b, the two most distant points in the point set.
3. **Projection of points onto line a-b** : We compute the orthogonal projection of each point onto the line defined by a-b
4. **Calculus of the median projection**: We compute the center of the current node (median of the orthogonal projections obtained in step 3).
5. **Calculus of the radius of the current node**: We compute the radius of the current node.
6. **Calculus of the left and right nodes**: We repeat the previous steps (starting at step 2) for the points whose orthogonal projection are to the left and right of the center (using the x coordinate). The resulting nodes will be the left and right child of the current node respectively. If during this we obtain a point set with only one point, we stop recursing and simply create a node with the point as it's center and radius 0.

3 Parallelization

Since the MPI solution handles larger data sets, it is required that this data is distributed among the available memory of each process, as we will present in the following sections.

3.1 Decomposition

One of the benefits of distributed memory programming is the ability to run larger instances of the problem, whose size does not fit on a single machine. In our implementation, the point set is split among the available processes. This means that there is a need for communication for most steps of the algorithm. The changes made to the algorithm to cope with this are:

1. **Initialization**: Each process only generates it's respective portion of the point set. Each process stores approximately n/p points. All work buffers also have the necessary size only.
2. **Calculation of a and b**: To compute the furthest away point, each process first computes the furthest point local to them and then an *MPI_Allgather* is used to gather all local furthest points. The furthest of these points is the global furthest point.
3. **Calculus of the median projection**: Since the points are distributed amongst the processes, a distributed sorting algorithm is necessary. We use the Parallel Sorting by Regular Sampling Algorithm. This algorithm is best at keeping each processes point list balanced. This algorithm does not work if $n >= p^2$. If this isn't true, surely the machine can store all the points and sort them. We use *MPI_Alltoallv* to get all projections and then sort them locally at each process.
4. **Calculus of the left and right nodes**: In this stage, there is the extra work of splitting the processes in two teams (using the *MPI_Group_incl* and *MPI_Comm_create* functions), with each team computing the left and right partition respectively. This process is further detailed in Section 3.3. Care is taken to preserve the order of points in the original set.

The primitive task of the Ball Tree Algorithm is the calculation of a tree node given a set of points, this is performed by a composition of steps 2-6 detailed in Section 2.1.

At the top of the tree, the work of the primitive task is split among the processes of the respective team. After the task is done, processes are split in two distinct teams, and each team will compute one of the child nodes. After a certain depth, all teams will have only one process and so that process can compute that node and all child nodes sequentially.

3.2 Synchronization Concerns

Processes only synchronize at MPI calls (i.e. MPI_Alltoall, MPI_Bcast, etc). The computation of the left and right child nodes of the current node are performed by completely different group of processes, that only communicate in the beginning to transfer the points of each partition to the respective team.

To minimize the cost of communication, the most expensive MPI call is done asynchronously: the transfer of points of each partition to the respective team. This allows the two transfers to occur at the same time.

Due to dependency of each step of the primitive task on the previous steps, most other calls are synchronous (since they need to be finished before advancing to the next step of the algorithm)

3.3 Load Balancing

Since the binary tree is balanced, the work of computing the left and right branch of a node is roughly the same. Due to the properties of the median projection, the child nodes of the same parent node also have the same number of points (give or take 1). As such, the time required to compute both nodes is the same.

After the top most node is computed by p processes, the two child nodes can be computed simultaneously. To do this, we split the processes in two teams. The left team, that has the first $p/2$ processes (i.e. the $p/2$ processes with the smallest rank) computes the left child node. The right team, that has the rest of the processes, computes the right child node. This repeats until the current team has only one process, at which point the single process computes the full current sub-tree sequentially.

If p is a power of 2, the work assigned to each process is perfectly balanced. For any depth of the tree, the number of processes in each team will be same. In this case, each process will participate in the computation of the top most node of the tree in a team of p processes, one of it's child nodes in a team of $p/2$ processes and so on until he is the only process in his team (which happens at a node with depth $\log_2 p$, in which case he computes the full sub-tree of the current node.

If p is not a power of 2, the work assigned to each process is uneven (it is clearly impossible to evenly split the work). For example, assume $p = 3$. A team of 3 processes compute the top most node. Then a team of 1 process (process with rank 0) computes the left most node (and since there is only 1 process in that team, it will compute the entire left sub-tree) and a team of 2 processes (processes with rank 1 and 2) compute the right node. Afterwards, the right team is further split, and 1 process computes the left sub-tree of the right node (process with rank 1), while the other computes the right sub-tree (process with rank 2). In this particular case the process with rank 0 has double the work of the other processes, and the execution time is expected to be the same as the execution time with 2 processes.

4 Experimental Results

Table 1 presents the time obtained for the serial and parallel implementations using 1, 2, 4, 8, 16, 32 and 64 nodes. The benchmark was ran on the RNL Cluster, on machines using an Intel(R) Core(TM) i5-4460 CPU with 4 cores running at 3.20GHz.

The time was measured using the OpenMP *omp_get_wtime()* function. We present only the times for very big inputs. For smaller inputs the obtained times were all too small to represent with only two decimal points.

Arguments	1	2	4	8	16	32	64
20 1000000 0	5.9	3.3	4.8	3.9	2.3	2.4	4
3 5000000 0	17.5	10.8	6.4	4.1	2.5	2.5	4.3
4 10000000 0	40.7	24.9	15	9.5	6	5.8	6.7
3 20000000 0	82.3	49.5	28.6	17.4	10.2	8.7	10
4 20000000 0	84.7	52.3	32.3	19.1	12.3	12.1	11.4

Table 1. Benchmark results

When comparing our sequential program time with the sequential times provided by the faculty, our program is roughly 36% faster for the larger inputs.

The parallel speedup for the larger instances is approximately $p/2$ (with p the number of processes) up to $p = 16$, at which point the performance gains plateau, as the overhead of communication increases at a larger rate than the work assigned to each process decreases. The number of processes at which point this plateau is reached depends on the problem size, so for larger instances we would not see this plateau.

The main parallel overhead of our program is in splitting the processes into two teams to compute the left and right child node of the current node and in *Alltoall* phase of the Parallel Sorting by Regular Sampling algorithm, since it involves a large transfer of data.

5 Conclusion

We present a parallel implementation of the ball tree construction algorithm in a distributed memory environment. Our approach splits the point set by the cluster machines, allowing instances that do not fit in the memory of a single machine, and splits the computation of sub-trees in parallel by different groups of processes, improving the computation time.

A Call Graph For One Node Of The Serial Algorithm

```
                     Call graph (explanation follows)


index % time    self  children    called     name
                                               <spontaneous>
[1]     84.9    8.06    0.00                 compare_node [1]
-----------------------------------------------
                                               <spontaneous>
[2]      5.1    0.48    0.00                 get_points [2]
-----------------------------------------------
                                               <spontaneous>
[3]      3.8    0.36    0.00                 orthogonal_projection [3]
-----------------------------------------------
                                               <spontaneous>
[4]      3.1    0.29    0.00                 build_tree [4]
                0.00    0.00       1/1           get_center [8]
-----------------------------------------------
                                               <spontaneous>
[5]      1.5    0.14    0.00                 create_array_pts [5]
-----------------------------------------------
                                               <spontaneous>
[6]      1.3    0.12    0.00                 distance [6]
-----------------------------------------------
                                               <spontaneous>
[7]      0.4    0.04    0.00                 frame_dummy [7]
-----------------------------------------------
                0.00    0.00       1/1           build_tree [4]
[8]      0.0    0.00    0.00       1         get_center [8]
-----------------------------------------------

Index by function name

   [4] build_tree              [6] distance                [2] get_points
   [1] compare_node            [7] frame_dummy             [3] orthogonal_projection
   [5] create_array_pts        [8] get_center
```

Fig. 1. Call graph for the serial computation of one node of the ball tree (20 million points, 4 dimensions)

A Execution Time Graph

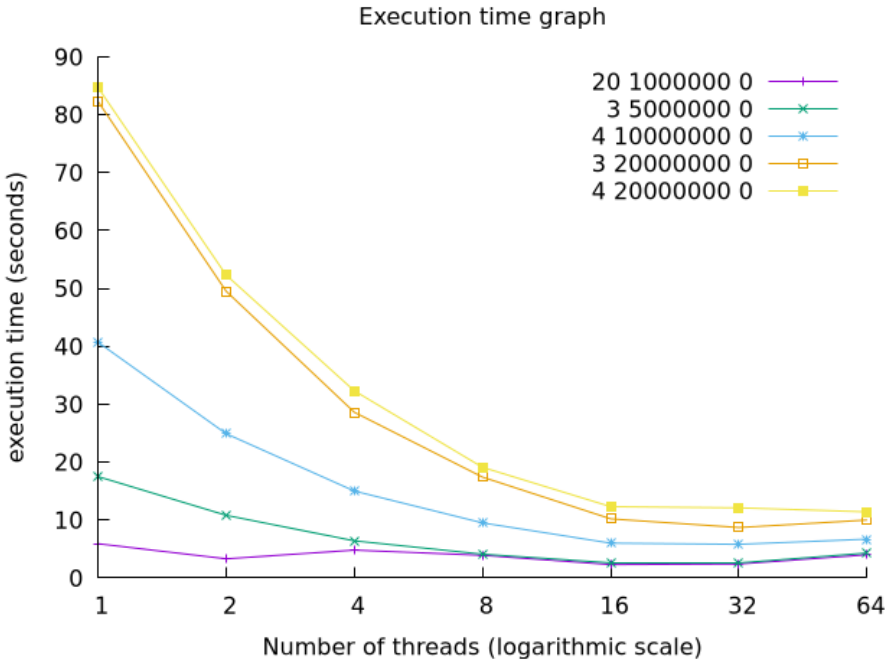

Fig. 2. Execution time graph