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 Shared-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 shared memory architecture, using OpenMP.

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 variables. This is the only memory allocation done by the program. It is also in this stage were 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 from the current point set as follows: point a is the most distant point from the first point in the set, point b is the most distant point from a in the set.
3. **Projection of points onto line a-b** : We compute the orthogonal projection of each point in the current point set onto the line defined by a-b
4. **Calculus of the median projection**: We compute the median of the orthogonal projections obtained in step 3. To do this, we sort the orthogonal projections by their x coordinate. This point will be the center of the current node in the ball-tree.
5. **Calculus of the radius of the current node**: We compute the furthest away point from the center (computed in step 4). The distance between it and the center will be the radius of the current node in the ball tree.
6. **Calculus of the left and right nodes**: We repeat the previous steps (starting at step 2) two times with the points whose orthogonal projection are to the left and right of the center (using the x coordinate) respectively. 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

In our OpenMP parallel solution we use data and recursive decomposition depending on the functions of the program.

3.1 Decomposition

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 2.1. Tree nodes are dependent on their predecessor but are not dependent on the same level nodes. Therefore, we can decompose the output data to be parallelizable. Once there is a division of the left and right side of a node an *omp task* is created to delegate the computation of the descendant left node to another thread available, while the current one will continue to compute the descendent right node.

The number of *omp tasks* created is limited. Although the code is recursive, we only make a recursive decomposition up to a certain level of granularity to avoid *task* creation overhead. We set a maximum level of tree depth to which the current thread will create new tasks, if the current depth of the node is equal or bigger than this maximum, the current thread will compute both left and right child nodes. The calculation of this maximum level is explained in 3.3.

In the top of the tree most threads are free since there are not enough nodes that can be computed. As we can see in Appendix A, 84.9% of the time spent computing a node is in sorting the orthogonal projections to find the median value which will be the center of the node. Thus, to speed up this process, we divide the projections in equal sized partitions that are sorted in parallel and then merged similarly to the merge sort algorithm. Like in step 6 this is only done up to a certain depth of the tree, otherwise the default single-threaded quicksort is used (*qsort*).

3.2 Synchronicity Concerns

The decomposition used ensures that there is no shared data among tasks. Each tasks work on a completely different set of points and there is no shared data, except for a global counter. This counter is used to determine the id of each node in the ball tree, and each thread that increments it does so while holding an exclusive lock. Each task has a private point set and private work buffers. Like in the serial implementation, all of the work buffers are allocated once at the beginning of the program, and each task gets an exclusive partition of the global work buffers to work on.

As mentioned above, for the topmost nodes the sort algorithm is parallelized: the orthogonal projections are partitioned and a task is created to sort each partition in a procedure similar to merge sort. Each of these tasks are independent, and there is no shared data among them. Once two adjacent partitions have been sorted, they must be merged (just like in merge sort). The thread responsible for merging a pair of partitions must wait that the two corresponding tasks to finish. This is done using the *taskgroup* directive of OpenMP.

3.3 Load Balancing

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

The maximum depth of the binary tree after which no more tasks are created is defined as follows:

1. If the number of threads available corresponds to a power of two, we create as many tasks as the number of threads. In this case each thread gets a primitive task (compute a node/branch of the tree) and the work is evenly split. The maximum depth will thus be $max\_depth = log_2(number\_of\_threads)$
2. If the total number of threads is not a power of two it is impossible to divide the tasks (and thus the work) evenly between every thread. In this case we use formula $max\_depth = ceil(log_2(max\_threads)) + 1)$ that in our experiments provided good results with 3 and 6 threads available.

The total number of created tasks of the parallel merge sort algorithm for a node with depth $current\_depth$ is $2^{max\_depth - current\_depth}$. This guarantees that if the total number of threads equals a power of 2 we are always using the total number of threads in each level of the tree. A simple example would be the case where we utilize 4 threads. In this case for $current\_depth = 0$, we will use 4 threads for the ordering process. For $current\_depth = 1$, 2 nodes are computed in parallel and each use 2 threads for their ordering process.

4 Experimental Results

Table 1 presents the time obtained for the serial and parallel implementations using 1, 2, 4 and 8 threads. The benchmark was ran 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	Serial	Parallel 1 threads	Parallel 2 threads	Parallel 4 threads	Parallel 8 threads
20 1000000 0	6.22	6.12	3.75	2.45	2.46
3 5000000 0	19.73	17.76	10.2	6.27	6.23
4 10000000 0	44.62	43.47	25.33	15.4	15.23
3 20000000 0	89.54	90.07	50.39	32.6	30.4
4 20000000 0	96.28	97.68	56.59	34.18	36.29

Table 1. Benchmark results

Since the CPU only has 4 cores, and each thread of the program makes full use of the CPU time available to it (i.e. tasks are not I/O bound) the execution time with 4 and 8 threads is roughly the same.

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

From this table we calculated the speedup, which is in Appendix C. The speedup obtained with 2 threads is around 1.74 and with 4 threads is around 2.82.

The ideal speedup would be 4, but achieving this is very difficult. Some parts of the program are inherently sequential, namely the generation of the points and memory allocation in the first step of the algorithm. The parallel implementation also has additional overhead of using a lock to access and increment the global counter used for the current node id, as well as the overhead of the OMP runtime related to tasks. When not accounting for the initialization overhead, we get a speedup of 3, as can be seen in the table of Appendix D.

5 Conclusion

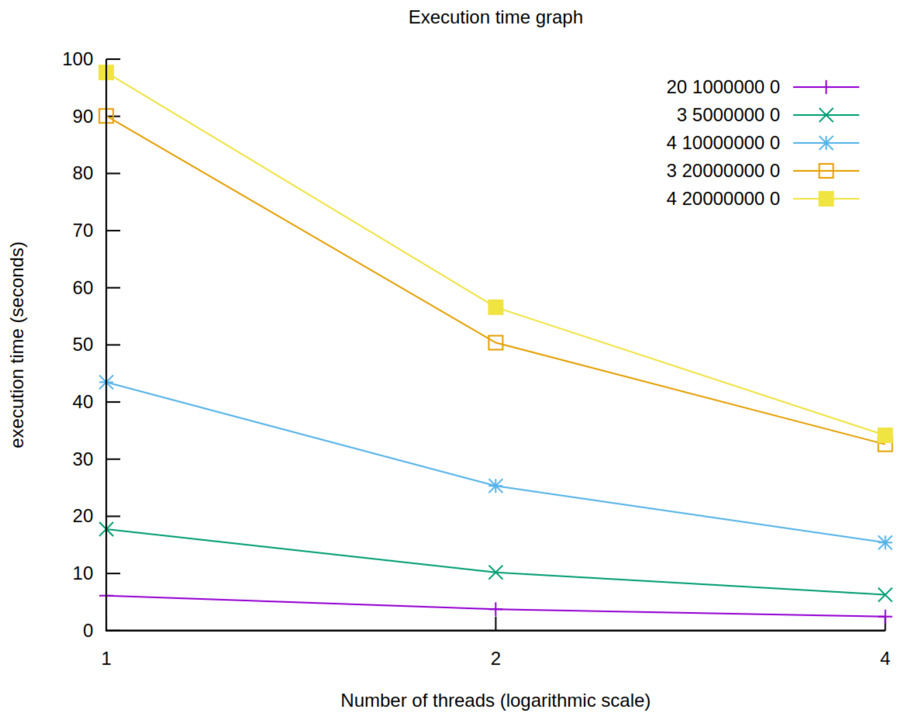
We present a parallel implementation of the ball tree construction algorithm. This problem is hard to parallelize because to compute each node in the ball tree you need to compute its parent. Our approach allows us to use all available cores to the program. When it is not yet possible to assign one subtree to each thread, idle threads are used instead to split the work involved in computing one node.

A Call Graph For One Node Of The Serial Algorithm

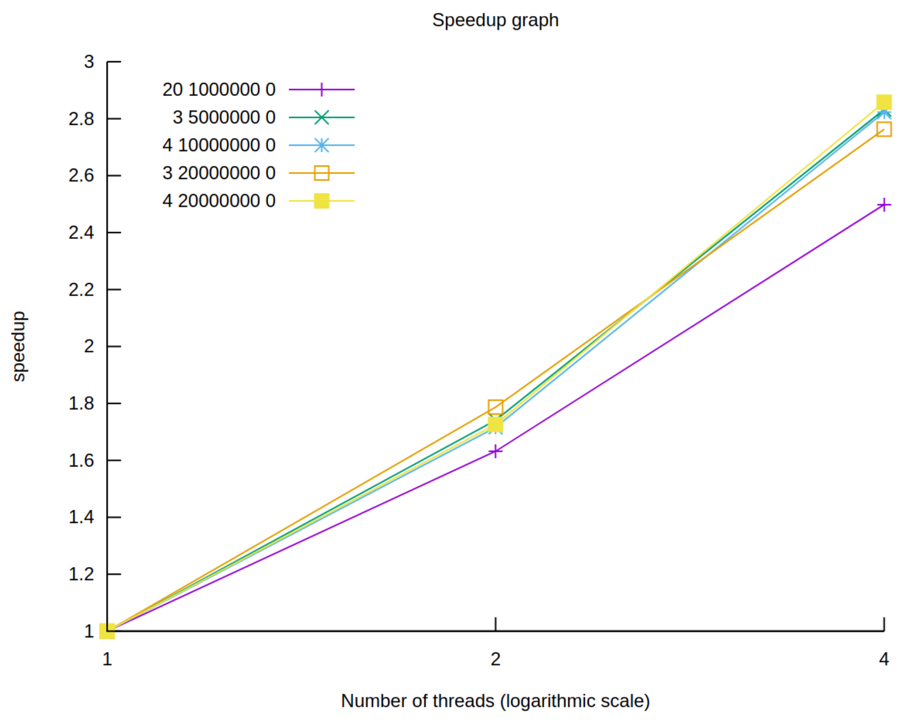
```
                     Call graph (explanation follows)


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)

B Parallel Time Plot



C Speedup Plot



D Benchmark Results Excluding Initialization

Arguments	Serial	Parallel 1 threads	Parallel 2 threads	Parallel 4 threads	Parallel 8 threads
20 1000000 0	5.86	5.85	3.36	2.07	2.08
3 5000000 0	17.5	17.59	9.75	5.94	5.76
4 10000000 0	42.66	42.47	23.97	14.29	14.19
3 20000000 0	88.22	88.09	48.51	28.81	29.01
4 20000000 0	95.45	94.84	54.39	31.72	31.87

Table 2. Benchmark results excluding initialization