Ball Tree Construction

Clara Gil (97070)[1], José Brás(82069)[1], and Miguel Barros(87691)[1]

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.

This document is organized as follows: we start by describing the initial, serial algorithm in Section 2. We then describe the approach used for parallelization in Section 3 and discuss the obtained speedup in Section 4. We present our final remarks in Section. 5.

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 node 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

1. **Initialization**: This stage is only performed once therefore there was no actual benefit in adding overhead to parallelize the initialization of data values.
2. **Calculation of a and b**: The return value of this stage is a unique value that results from a simple computation which means there was little to parallelize. However, this section is called by method 6 which runs multiple times and this one is parallelized as we will see next.
3. **Projection of points onto line a-b**: Just as the previous section this section is not parallelized but it is called by another that it is, hence to avoid overwrites the method has memory space reserved for each working thread (detailed at 3.2)
4. **Calculus of the median projection**: Although this code section is also called by 6, the search for a median value involves sorting the projection which can be a slow job, as we can see in Appendix it consumes 90% of the time. Hence instead of using only quicksort as in the sequential program, we use both quicksort and mergesort in a hybrid way to speedup the process. We decomposed this section in a recursive way. The mergesort will decompose the problem in levels of equal sub-problems until a certain level of granularity is reached. The stopping point is defined by the load balancing mechanism 3.3.
5. **Calculus of the left and right nodes**: This section is decomposed in both data and recursion such that since the code is already recursive in itself each time the program dives into a sub-level it goes as a thread. Tasks are created as defined in 3.3. However this recursion is done twice for each side of the node, thus allowing us to parallelize the computation of each since the values will be independent of each other.

3.2 Synchronicity Concerns

The decomposition used ensures that there is no shared data among the 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 executed has a private point set and private work buffers (for example for storing the orthogonal projections). 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 buffers to work on.

3.3 Load Balancing

In our program the number of threads available and tasks created are tied together. Our load balancing, after the computation of the root node of the tree, follows the following principle:

3.3.1 Thread and Task Usage Each group of points is computed by a single thread. Each thread makes use of a single assigned task unless the maximum number of threads does not equal a power of two number. If that is the case, each thread gets *current_depth* tasks, where *current_depth* is the current depth of the tree. We delve more into this topic in Section 3.3.5.

3.3.2 Binary Tree Assuming we just computed the points for node n with thread x, the following level $n+1$ will be computed by threads x and $x+1$ for the newly spawned left and right branches respectively. Each of these threads will be assigned a single task to compute the next points if the maximum number of threads is a power of two, or more than on task in case the maximum number of threads does not equal a power of two as explained in Section 3.3.5.

3.3.3 Cores and Threads Our intention is the assignment of a single to thread to each core of the processors. This way each core performs the computation for the ball points of its current node. Nonetheless our experimental procedures account for the usage of more threads than the number of cores in the testing machine as can be observed in Section 4.

3.3.4 Threads and Depth The depth of the tree is related to the number of threads generated for each level. Assuming *max_threads* and *current_depth* to be the maximum number of threads and current depth of the tree, at the current level of the tree, if $current\_depth \leq log_2(max\_threads)$, we always have $2^{current\_depth}$ threads running. In case $current\_depth > log_2(max\_threads)$ no more threads will spawn, instead the remaining *max_threads* will compute the nodes of the current and higher depths until the whole tree is computed.

3.3.5 Non-power of two Maximum Number of Threads When we utilize non-power of two *max_threads* the load is balanced in a different way. Each thread will get more than one task until $current\_depth > ceil(log_2(max\_threads))+1$. Each thread gets *current_depth* tasks. This way the task load is more evenly distributed between threads.

4 Experimental Results

Table 2 presents the time obtained for our serial implementation and for our parallel implementation 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 decimals.

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

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

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. it tasks are not IO bound) the speedup with 4 and 8 threads is roughly the same.

From this table we calculated the speedup, which is in Annex 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 sequencial, 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 OMP tasking. When not accounting for the initialization overhead, we get a speedup of 3, as can be seen in the table of annex D.

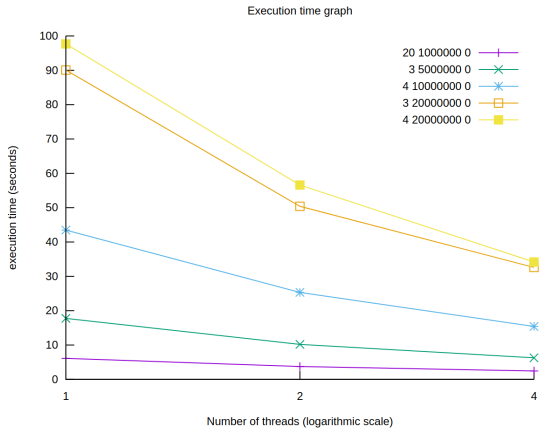
5 Conclusion

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 parents. Our approach allows us to use all available cores to the program. When it is not yet possible to assign one node to each thread, idle threads are used instead to split the work involved in computing one node. Our decomposition achieves a speedup of 2.82 when using 4 threads.

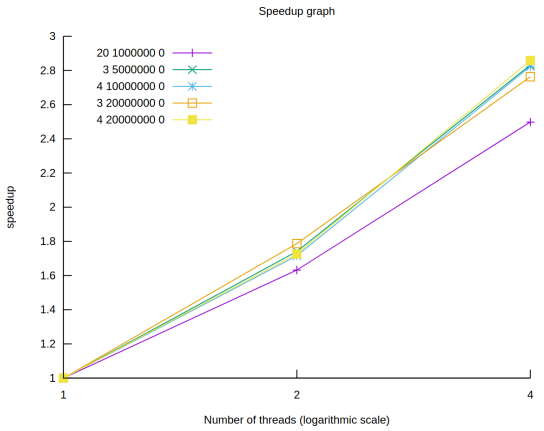
A Call Graph For One Iteration Of The Serial Algorithm

```
                     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
```

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