Parallelizing the Ball Algorithm

OpenMP — Group 8

Afonso Ribeiro, 86752 anfonso.ribeiro@tecnico.ulisboa.pt
Baltasar Dinis, 89416 baltasar.dinis@tecnico.ulisboa.pt
Pedro Lamego, 89526 pedrownlamego@tecnico.ulisboa.pt

1 Problem Statement

We aim to parallelize an implementation of the Ball Algorithm, which constructs a ball tree over a set of n-dimensional points. This data-structure is extremely helpful because (when properly instantiated), it allows for logarithmic lookup of which point is closer to a new point. Here, we consider that the set of points is generated from a triple $< D, N, S >$, where S is the value which seeds the random number generator which will then create N D-dimensional points.

2 Serial Solution Overview

Our serial solution has a simple recursive strategy:

1. Find two distant points, A and B, to separate the set;[1]
2. Partition the points based on their projection on the line AB;
   (a) The center is the median of these projections;
3. Compute the radius of the ball (distance from center to point furthest away);
4. Recurse to left and right subsets.

However, we employ several interesting algorithmic optimizations[2].

Computing Projections. The projection of a point p on a line defined by two points a and b is computed as follows:

$$\frac{(p-a)\cdot(b-a)}{\| b-a \|}(b-a)+a \tag{1}$$

However, we only ever need the actual coordinates of the projection in the case of the median. We can skip computing the projection coordinates for all other points, and simply compute the inner product ($(p-a)\cdot(b-a)$). We then use this value to compute the median and partition the points.

Finding the Median. To find the median, we can avoid sorting the whole array, instead using the Quick-Select algorithm to find the median directly. Instead of costing $O(n \log n)$ (from sorting), finding the median becomes a linear operation.

Implicit Leaves. We employ a much more efficient representation of the ball tree, where leaves are implicitly represented in their parents. This is very beneficial, because we can reduce the size of the array by half.

Deterministic IDs. We compute the node IDs deterministically, based on the ID of their parent node. Let that ID be p. The left child id is $2p+1$ and the right child id is $2p+2$, if they are not leaves (leaf ids have an extra offset, due to the implicit construct). Although, This requires allocating more space, but is greatly beneficial because it removes the need for synchronization when assigning node IDs.

3 Parallelization Strategy

In this section we explore our synchronization strategy, exploring the problem decomposition, load balancing and synchronization.

3.1 Decomposition

Considering the problem globally, there is a large serial computation: the generation of points. This is required to be serial because otherwise calls to `random(3)` would not be ordered, and the tree generation becomes non-deterministic. Other than this, it is possible to parallelize the computation of the partition and ball radius (steps $1-3$) and the subtree recursion (step 4).
In fact, it is this subtree recursion which yields the best parallel speedup: if there are still threads available we can recurse into the left subtree and spawn another thread to recurse into the right one. Note that at some point there will not be any more threads available, and the computation becomes serial (for each subtree).
The question becomes how to parallelize the partition and radius computations. These are mainly `for`-based computations, which can be parallelized (in OpenMP) using the `parallel for` directive.
Guided by experimental results, we chose only to parallelize the inner product computation, as further parallelizations yielded slowdowns.

3.2 Load Balancing

We note that, after the first few steps, each thread is running a serial computation on a subtree. This happens for all threads at the approximately the same level (when the number of subtrees of the level is the

[1]These points being distant is a necessary condition for the lookup on the tree to be logarithmic.
[2]We also implemented several interesting implementation optimizations, which we omit for brevity

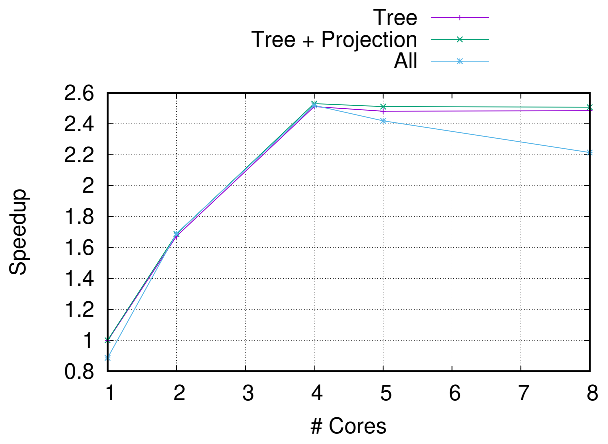

Figure 1: Comparison of three different parallelization approaches.

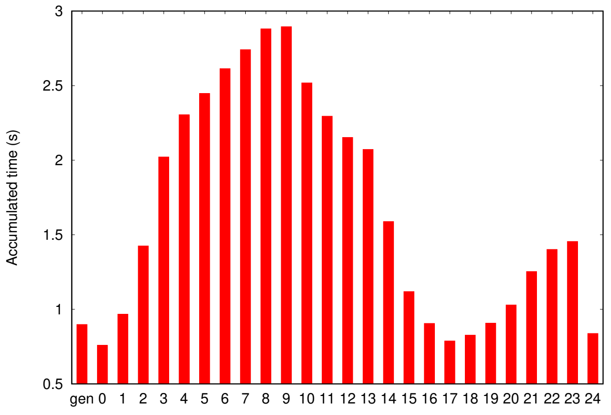

Figure 2: Time spent on each level of the tree.

same as the number of threads), so it is expected that they all have the same load.
Before reaching this saturation point, some computations are still parallelizable. In this case, load balancing is assured based on partitioning the array slices so that each thread has roughly the same amount of work. Again, this is ensured by the OpenMP primitives.

3.3 Synchronization

Since all memory is pre-allocated, and never reallocated subsequently, we are by design protected from pointer invalidation. The deterministic assignment of IDs ensures that we do not need to synchronize to assign IDs. Moreover, when recursing each thread has a disjoint slice of the array of points to work on. This ensures we do not need to synchronize on this vector. Finally, all `parallel fors` are read-only operations, occurring separately from any write operation on the same array slice.

4 Experimental Evaluation

We evaluated our solution, attempting to answer several questions presented in § 4.1. We performed the experiments remotely on the machines of Laboratory 13 (verifying we had exclusive access to the machines). For each datapoint we repeated the experiment 5 times and we present the average.

4.1 Expected Results

Our first observation is that, due to our several optimizations on the serial implementation, it is harder to obtain very large speedups. We remind the reader that there is an unparallelizable computation: the point generation.
Second, we expect our solution to compare much favourably (to other approaches) when there is a high number of dimensions, because the projection optimization is more efficient when there are more dimensions.
We further expect that the time required to achieve the state where each thread is assigned to a different subtree to be quite small (which would explain why trying to further parallelize yields no speedups).
Based on these expected results, we propose three experiments:

Approach Comparison. (§ 4.2) We implement three different approaches to parallelization, to experimentally verify that our approach is the best. The first only parallelizes the tree recursion. The second (our final approach) further parallelizes the computation of inner products. The third parallelizes computing points A and B (step 1) and the maximum radius of the ball (step 3).

Level Discrimination. (§ 4.3) For a given experiment, we discriminate how much time is spent in each level (and on the point generation). This helps corroborate the fact that further parallelizing steps $1-3$ would have low impact, because only the first levels have available threads to do so.

Test Cases. (§ 4.4) For our final solution, we present the different results for several provided test cases.

4.2 Approach Comparison

As Figure 2 shows, trying to further optimize steps $1-3$ actually yields a slowdown. However, parallelizing the inner product computation is still worthwhile. We used the last of the test cases (4 dimensions with 20×10^6 points).

4.3 Level Discrimination

In this experiment, we measure how much time is spent in the non-recursive part of the algorithm (steps $1-3$) at each level of the tree. As we can see in Figure **??**, very little time (relatively) is spent in the few first levels, where parallelizing these steps would be worthwhile. We also note that the point generation is indeed significant (note that its latency is on par with one level of the tree). We used the last of the test cases in this experiment.

4.4 Test Cases

In this experiment we ran our final solution on a variety of test cases, and report both the latency and speedup in Figures 3 and 4.

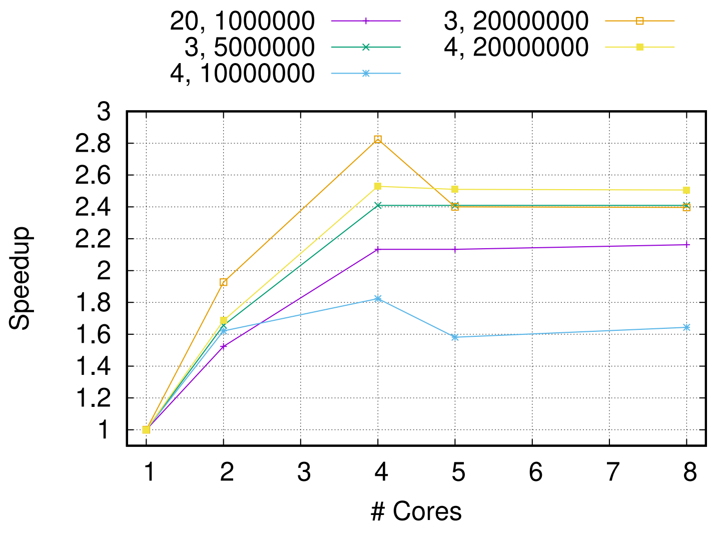

Figure 3: Speedup in multiple tests.

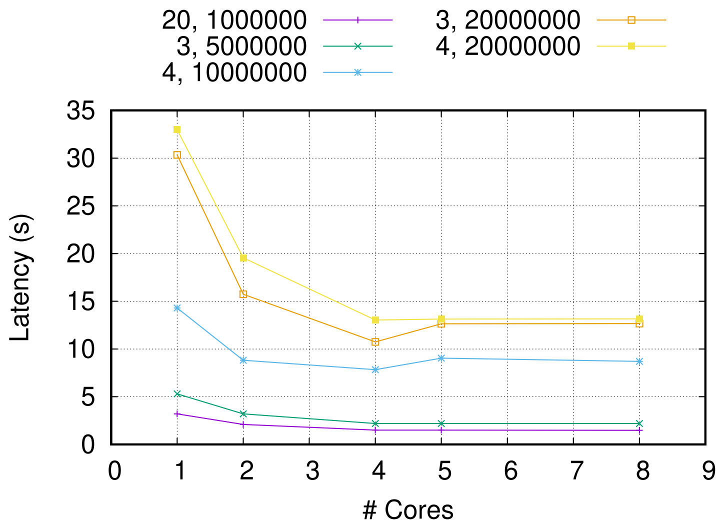

Figure 4: Overall latency in multiple tests.