Parallelizing the Ball Algorithm

MPI + 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.

3 Deterministic IDs

Previously, we employed a well known ID assignment scheme for binary trees:

$$left(p) = 2 \times p + 1$$
$$right(p) = 2 \times p + 2$$

Where $left(p)$ is the ID of the left child of node with ID p (respectively for $right(p)$. This scheme requires only $2^k - 1$ allocated nodes, where k is the smallest value such that $\frac{N}{2} \leq 2^k$ (the $\frac{1}{2}$ factor is due to the implicit leaves optimization).
However, it is unsuitable for the distributed solution since IDs are grouped by level, causing sub-trees to be spread out in the node array and overlapping with other (disjoint) sub-trees. When each process should compute a particular sub-tree, this becomes problematic: it becomes difficult to link a node's ID with its index in the pre-allocated array of nodes, especially if the process cannot hold all points in memory.
To solve this problem, we devised an ID assignment scheme, which satisfies the following properties:

P1. Two disjoint sub-trees have disjoint ID ranges;

P2. The ID space is compact.

We say the ID space is compact if a tree with n nodes requires at most a n-sized array of pre-allocated nodes to be represented.
Our scheme is the following:

$$left(p) = p + 1$$
$$right(p, N) = p + \frac{N}{2}$$

Where N is the number of points the sub-tree rooted at p represents. Although we believe this scheme to generalize, it is rooted in the assumption that the binary tree is right-heavy (if a node has only one child, it is always the right child), which is the case in our solution. Furthermore, the scheme requires knowledge of how many points a given sub-tree represents, which is true in our problem space. We note that given a node with ID p which is the root of a sub-tree representing N points, will occupy the ID range $[p, p+N)$, simultaneously satisfying P1 and P2. It satisfies P2 because it requires only $N-1$ nodes, which 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

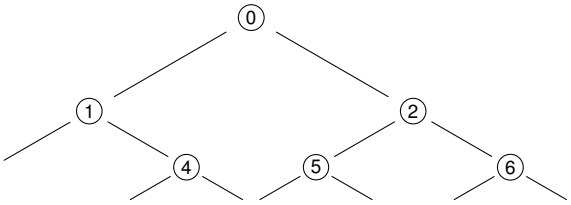

Figure 1: Original ID assignment for a tree representing 7 points.

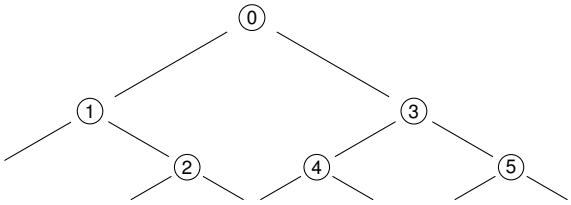

Figure 2: Improved ID assignment for a tree representing 7 points.

minimum number of nodes required for representing N points (a binary tree with N leaves has $N-1$ inner nodes). Since it is compact, it is impossible for a disjoint sub-tree to overlap this range, satisfying P1.
As an example, Figures 1-2 show ID assignments for trees rooted at 0 representing 7 nodes, both in the original scheme and in our scheme.
To the best of our knowledge, this ID scheme is novel, as we know of no prior work which achieves these properties. Given the wide applicability of binary trees and the natural way they support parallel computation, we believe this can lead to interesting generalizations and applications.

4 Parallelization Strategy

In this section we explore our synchronization strategy, exploring the problem decomposition, load balancing and synchronization. In the context of MPI, there are two cases we must consider when tackling the problem: 1) the case where at least one node can have the entire set of points (*single process* version) and 2) when each node can only retain a portion of the points (*distributed* version). Unless manually overridden, the solution decides autonomously which case to do, by pre-allocating all required memory (if this allocation fails to have all the points in all nodes, the system falls back on the distributed version).
Due to its simplicity, we first describe the parallelization strategy for the single process version (§4.1) and then the distributed version (§4.2). In both cases, at some point the computation will reach a stage where a node has sole responsibility for computing a given sub-tree. Here, we employ the solution from the first part of the project (referred to as the *exclusive sub-tree solution*), which uses thread-based parallelism. Since it was already explained in detail in the previous submission, we will halt our explanation at that point.

4.1 Single process version

Decomposition. The decomposition in this case is straightforward: the nodes organize themselves in (implicit) teams. Each node in the team computes *the same node* of the tree. Crucially, the team then divides itself in two, and the first team only computes the left sub-tree and the other the right sub-tree. Although this does require some duplicate work, any other solution (where only one or some nodes computed the node) would introduce a data dependency, slowing down the solution, whilst all the other cores would go under-utilized. Eventually, each node will be alone in computing the sub-tree, where the exclusive sub-tree solution is applied.

Load Balancing. We note that, after a few steps (approximately $\log(P)$, where P is the number of MPI processes), each process will be in exclusive sub-tree mode, all with approximately the same number of points in its care (P is not a power of 2, some processes start the exclusive mode at given level and others at the next level). We make the assumption that all processes have the same computational power, so there is no optimization to be made. If this were not the case, we could make sure that, if such asymmetries in the solution, then the weaker nodes could enter the exclusive mode at the lower levels.

Synchronization. This solution requires little to no synchronization between processes. Each node computes on its local data, and need not wait for the rest of the team before moving on to other levels. However, since in the beginning multiple processes are computing the same tree nodes, the system must consider which processes stores (and prints) the node. This is decided trivially (and without explicit synchronization): each team has an implicit leader (with id 0), and that is the node which stores the value.

4.2 Distributed version

Decomposition. There are three stages to the distributed computation of the tree: 1) finding the two most distant points; 2) finding the center and computing the radius of the node and 3) partitioning the points and dividing the computation. The first stage is quite straightforward: the leader (process with ID 0) sends the first point to all nodes, which compute their most distant point a_p, synchronize to compute a and then repeat the process to compute b. To compute the center, we compute the local projections and then find the median of the projections. This is done using a distributed variant of QuickSelect. The algorithm is as follows: at every round, a pivot is chosen from the available points (the process which chooses keeps changing); all processes partition their points on this value: if the sum of all partition indices is $\frac{N}{2}$ then we have found the median (the pivot). Otherwise, we can check whether we overestimated or underestimated the median, and shrink the target interval at each

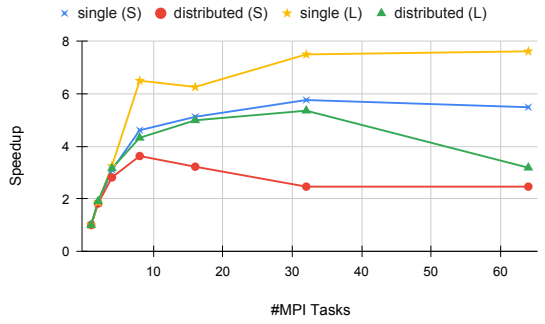

Figure 3: Scalability of the single process and distributed versions. Shown as speedup of running the workloads with 1, 2, 4, 8, 16, 32 and 64 processes.

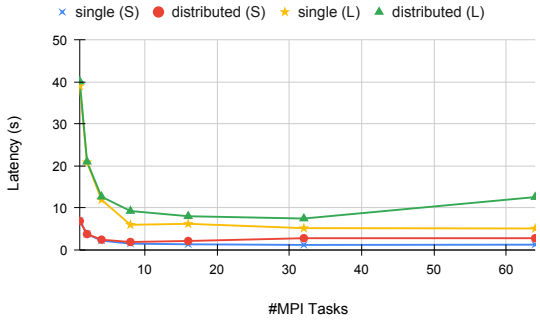

Figure 4: Latency results of the single process and distributed versions.

node accordingly. This is a linear operation. Computing the center and the radius is then straightforward. Partitioning the points is one of the most convoluted parts of the process. The problem is that a given node will either belong to the left or right groups (after the computation is divided) and has to send all points which it is not going to compute on to nodes that need this data. Moreover, we would like this data to be contiguous, so we can use the `MPI_Sendrecv_replace` primitive, allowing the two arrays to be swapped between processes in-place. However, our representation has two arrays: one of points (which until now remained static) and one of pointers to points (which point to the first array). The problem now is that we cannot have pointers "crossing" the partition index p (ie: a pointer in position $i < p$ pointing to a point with index $j \geq p$). So, we first *untangle* the arrays, finding symmetrical crossings and swapping the underlying points and pointers. Afterwards, each node receives the information pertaining to how much each node receives. Each node then can autonomously compute which ranges it has to swap with which processes.

Load Balancing. As with the single process solution, when the computation reaches the exclusive mode, each process has roughly the same amount of points to process. In the cooperative phase, all processes have equal sized ranges of points, so they are expected to have the same load. Again, if processes are expected to have uneven computational power, this can be tuned.

4.3 Synchronization

In this version of the protocol, there are several points where processes need to synchronize. However, this synchronization is guaranteed trivially by using MPI's primitives.

5 Experimental Evaluation

We evaluated our solution, attempting to answer the following questions:

Q1. How does the single process version perform (in terms of scalability);

Q2. How does the distributed version perform (in terms of scalability);

Q3. What is the latency of the single process version compared with the latency of the distributed version.

We performed all experiments remotely on the machines of the RNL cluster. For each datapoint we repeated the experiment 10 times and present the average.
To answer Q1 and Q2, we exercised the system (in both modes) with a varying number of MPI processes ($1, 2, 4, 16, 32$ and 64, with 1 serving as a baseline) and always with 4 OpenMP threads. We designed 2 workloads to exercise the system with: a small (S, 3 dimensions and 5×10^6 points) and large (L, 4 dimensions and 20×10^6 points). As we can observe in Figure 3, the single process approach yields a better speedup (which does not deteriorate). The experiment also indicates (unsurprisingly) that the bigger the problem, the more scalable the system. Regarding the latency experiment, we observe that the single process variant fares better in comparison with the distributed setting. Both this and the comparatively smaller speedup is expected, since the distributed version incurs a high synchronization overhead, while the single process has no synchronization.