The EMU dbscan implementation uses a k-d tree to speed up nearest neighbor computations – the tree provides a spatial decomposition that allows quickly ruling out large selections of points (subtrees) far from the search point, while constraining exhaustive searches to a (hopefully) small region around the search point. While simple in theory, efficient implementation on a distributed memory system like EMU proves not so easy, especially when considering parallelization.

We’ve found that while increasing the number of threads does improve performance, debug outputs (.cdc) show much of the work falling onto a single nodelet. While many factors could be at play, threads migrating to the single nodelet holding the k-d tree presents an obvious issue, especially considering nearest neighbor searches make up the bulk of the dbscan local computational load.

As a first stab, we initially replicated the k-d tree across all nodes – for now not worrying about the increased memory footprint – to test our intuition. The observed preprocessing cost was otherwise quite small in the single threaded case, as creating deep copies of the k-d tree only requires pointer updates. Then as we increased the number of threads, preliminary simulator timings indicate significant speed up, roughly on par with that of replicating the point data. Replicating both the tree and point data further compounds this speed up.

Additionally, the replicated k-d tree implementation demonstrates effective scaling to a larger number of threads (we observed a minimum runtime at 32 threads versus 16) before thread overhead outweighs additional parallelization benefits. This deferral of thread saturation is a key observation indicating the replication does in fact improve thread distribution, alleviating traffic to any single nodelet. Otherwise we would expect to see this bottlenecking effect earlier.

However, the cdc output still shows much of the work on node 0, which shouldn’t be the case if \*everything\* is replicated… [if I’m interpreting correctly].

[That issue aside] However, this solution begs the question of memory usage. Afterall, the motivation behind PIM is to keep the memory size small per nodelet to speed up access. For larger problems, we cannot expect full replication of the k-d tree (and maybe not even allocation on a single node), and especially not for the point data.

That said, we can still alleviate traffic to nodelet 0 with *partial* replication of the tree, namely the highly trafficked top near the root. For n nodelets, imagine *only* replicating the top log n levels (n total nodes) of the tree, with the remaining subtrees distributed across nodes (one each). Even if the tree grows with the number of points, the replicated memory overhead is bounded by the number of nodes. Furthermore, point queries can then be run on *any* node up until the point of a single migration and termination on the node corresponding to the correct subtree. In this way, no single node is responsible for the tree bottleneck.