Scientific Computing With Rust

Srinath Kailasa

Department of Mathematics
University College London

April 3, 2022

Table of Contents

Table of Contents

Points2Octree

- **Goal**: Go from a set of points distributed across processors to an *unbalanced distributed octree.
- State of the art software tends to use linear trees - i.e. only store leaf nodes [1, 2, 3, 4, 5]
- We implement the scheme outlined in [5] augmented with extra features for use with FMM and other solvers using **Rust** and MPI.

Octrees

- Represent each node uniquely with an anchor and its level in the tree.
- e.g. the anchor of d is (4, 2) and it's at level 3.
- Can represent uniquely with tuple (4, 2, 3).

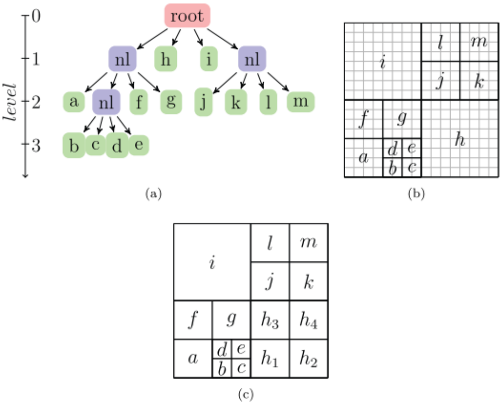

Figure 1: Adapted from [5]

Nice Properties of Morton Encoding

- Sorting leaves in ascending order of Morton keys is equivalent to pre-order traversal of the leaves. Connecting centres, we observe a 'Z' pattern.
- Encoding preserves spatial locality.
- Given octants a b and c with $a < b < c$ and $c \notin \{\mathcal{D}(b)\}$ $\implies a < d < c, \forall d \in \{\mathcal{D}(b)\}$

Briefly on Rust

- Offers memory safety (compile time checks for memory violations).
- Does away with complex programming models (object orientation).
- (Relatively...) Simple API, can feel like writing in an interpreted language.
- Interfaces nicely with other languages, and their libraries. e.g. mature (open source) support for MPI.
- Simple cross platform build system (Cargo).
- 'Zero Cost Abstractions' - performance won't suffer for using handy abstractions (map, filter, fold ...).

Points2Octree - Algorithm

1. Apply Morton encoding to points at each processor.
2. Perform a parallel sort, and remove duplicate Keys, associate points and Keys. Dominates complexity of algorithm [5].
3. Find *coarsest possible* tree that spans the domain defined by the Keys at each processor.
4. Find the *coarsest* node(s) at each processor. Complete the region in between the coarsest node(s) across each processor - known as the **blocks**.
5. Perform some kind of load balancing over block, and redistribute the blocks.
6. Split blocks to find final distributed octree.

Illustration of Blocks

Figure 2: Adapted from [5]

2:1 Balancing

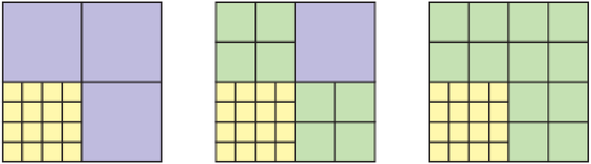

Figure 3: Adapted from [6]

2:1: Balancing

Not always strictly necessary (e.g. particle FMM). However, essential to offer functionality in any octree library. Tends to dominate runtime, run as post-processing step.

1. E.g. evaluating near interaction in volume (continuous medium) FMM $\int_{NearField} K(x, y)f(y)$ is computationally expensive (many singular/near singular integrals), balancing allows one to use precomputed tables for valid interactions [1].
2. Other numerical schemes may take advantage of it to maximise accuracy while minimising memory footprint [6].

Options for Balancing

Two major algorithmic approaches to balancing

1. (older) *Ripple* based algorithms [2, 5].
2. (newer) Sorting based algorithms [1, 3].

2:1 Balancing - (1) Ripple Based

1. Each process enforces balance in its subdomain, must communicate with neighbours to resolve conflicts.
2. Generally require multiple rounds of communication.
3. Algorithmically fairly complex, with no clear advantage of sorting based methods [6].
4. Existing OS software: P4EST (C++)
5. Won't talk about them further.

2:1 Balancing - (2) Sorting Based

1. Each process computes a tree over global domain that's suitably balanced with the octants it controls.
2. These are then sorted in parallel, and duplicates/overlaps are removed (favouring the smaller octants). Guaranteed to be 2:1 balanced.
3. Simple communication pattern (sorting).
4. Existing OS software: Dendro (C++)

2:1 Balancing - Sequential Algorithm

Example local balancing algorithm (there are numerous approaches):

1. Balance Subroutine

for $i \leftarrow L_{max}$ *to* 1 do

   1. For each octant at this level, find coarsest compatible neighbours, add to output.

2. Remove overlaps

for $i \leftarrow L_{max}$ *to* 1 do

   1. For each octant at this level, remove any ancestors that lie in the tree.

Choosing a Sorting Algorithm

This is the most important choice, as both building and balancing require a parallel sort, and in both cases will dominate runtime.

State of the Art - HykSort

First presented in 2013 [4], remains fastest method in 2021 [6].

1. Sample Sort contains a global AllToAll communication. This leads to network congestion for larger problems.
2. Requires selection (and parallel sorting) of $p - 1$ splitters, where p is the number of processors. Can be expensive for large p.
3. HykSort uses a Hypercube style communication pattern, but with k rather than 2 splits for each recursive call, where $k < p$, $k\beta$ splitters are chosen.

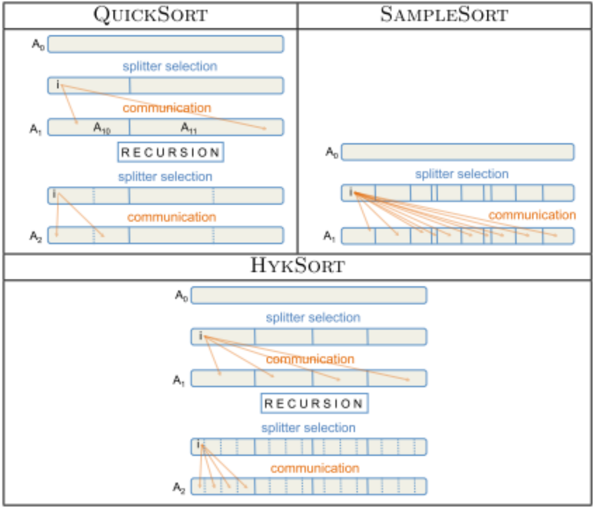

Figure 4: Adapted from [4]

1. In [4] they sort 8 trillion 32 bit integers in 37 seconds on 262,144 cores.
2. Use OpenMP optimised local sorting algorithm based on merge sort, and SIMD optimised merge operation. Don't report similar benchmark for sorting Morton keys though.

Table of Contents

1. Software

Available at github.com/skailasa/distributed-trees

1. Currently only uses sample sort for parallel sort, and balancing not yet implemented.
2. Local sort using Rust's native implementation.
3. Fully unit tested.

TODO:

1. Implement and test HykSort, and whether it matters for our problem sizes.
2. Morton keys are represented as vec of structs for clarity, though would be preferable to represent them as a struct of arrays for speed.
3. Add balancing algorithm.
4. Add API for higher level codes, i.e. easy access to interaction lists etc.
5. Measure strong and weak scaling on cluster.

2. (Local) Benchmarks

Tested on my 6 core i7 laptop.

1. 6e6 randomly distributed particles (1e6 per core), ncrit=150, max depth=7.
2. Unbalanced tree with 260688 leaf nodes.
3. Runtime: 2.41 seconds. Peak Memory Usage: 1.9 GB.

Interesting comparison, single node C++ implementation in ExaFMM-T, current state of the art, accelerated with OpenMP, for same problem has a runtime of 3.94 seconds with peak memory usage of 1.6 GB.

References I

[1] D. Malhotra and G. Biros, "Algorithm 967: A Distributed-Memory Fast Multipole Method for Volume Potentials," *ACM Trans. Math. Softw*, vol. 43, 2016.

[2] T. Isaac, C. Burstedde, and O. Ghattas, "Low-Cost Parallel Algorithms for 2:1 Octree Balance,"

[3] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," 2009.

[4] H. Sundar, D. Malhotra, and G. Biros, "HykSort: A new variant of hypercube quicksort on distributed memory architectures," *Proceedings of the International Conference on Supercomputing*, pp. 293–302, 2013.

References II

[5] H. Sundar, R. S. Sampath, and G. Biros, "Bottom-up construction and 2:1 balance refinement of linear octrees in parallel," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2675–2708, 2007.

[6] H. Suh and T. Isaac, "Evaluation of a minimally synchronous algorithm for 2:1 octree balance," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2020-Novem, no. Section VI, 2020.