Adaptive Octrees, Algorithms and Implementations

Srinath Kailasa [*]

University College London

October 14, 2020

Abstract

Octrees find a wide variety of applications across computational science as they allow for the spatial decomposition of a three dimensional domain. The adaptivity of an octree refers to the potential to have neighbouring tree nodes of non-uniform size at a given level of the octree. This allows for a better description of non-uniformly distributed data, allowing the octree to 'adapt' to the data. In this document we describe the main theoretical underpinnings of adaptive octrees, with special reference to their application in Fast Multipole Methods. We move on to a description of major sequential and parallel implementations of adaptive octrees, and describe the strategy being undertaken by the new Python/SYCL implementation being developed as a part of the Excalibur exascale software collaboration, AdaptOctree [3].

Motivation

The motivation behind the AdaptOctree software is to create:

- A Python library for adaptive octrees, pluggable into the PyExaFMM interface.

[*] srinath.kailasa.18@ucl.ac.uk

- SYCL acceleration for data parallelism.
- MPI acceleration for process parallelism.

Few generic oct/quadtree implementations seem to exist in the open source. A cursory google search found a single example of a well maintained library, that makes use of shared memory parallelism via OpenMP and OpenCL acceleration for data parallelism maintained by the Klöckner group at UIUC [1] - the home of PyOpenCL. The overarching goal in creating this software is a further modularity and acceleration of PyExaFMM.

Octrees

It is preferable to to work with *linear* representations of Octrees. In this context linearity refers to just storing the Morton encoded keys of octants at the lowest level of granularity, or leaves. This contrasts with traditional implementations which make use of pointers, which can be hard to synchronise in parallel implementations [6]. Furthermore, this seems to have been adopted as a standard way of representing octrees judging by recent literature on the subject [5, 4]. Furthermore space filling curves based on Morton encodings seem to be a standard [5, 6], due to their relative simplicity.

In the context of FMMs, the original implementation using an adaptive octree [2], constructed the octree in a sequential manner. The algorithm for this is simple, starting from the root node, we recurse post order (top-down) the octree, refining a child box if it contains more than S particles, which is a user determined constant.

Approaches to parallelise this begin by assigning equal numbers of particles to be worked on by each processing unit in a parallel scheme, however this results in the following difficulties, which we quote from [6]

1. If the number of particles in a given octant is less than S for a given processing unit, this doesn't mean that this is true when that octant's particles are accumulated over all processors. Thus we may end up with coarser octants than intended.
2. Generally, we'd like unique octants in each processor, to reduce communication overheads. This makes it imperative to remove duplicates across processing units.

3. Since there are guaranteed to be a number of duplicates and overlaps in terms of the locally constructed trees ('local' in the sense of the processing unit) this adds to the amount of work that can't be considered 'useful' i.e. solving the problem.

Therefore Sundar et. al propose an alternative 'bottom-up' approach in which octrees are represented by the Morton ids corresponding to their leaf nodes. This is a linear data structure, and therefore far easier to work with in parallel implementations. It doesn't take much to imagine partitioning schemes where leaf nodes are chunked and distributed across processing units. If we take care to chunk based on Morton id ranges, we can preserve locality of data and ensure that each processor is able to construct a local subtree. This is the approach taken in available parallel octree implementations [4, 5]. This approach incurs a cost in that the algorithms with which to construct adaptive trees become correspondingly more complex.

Before moving on to algorithms, we describe some necessary facts about the approach taken to Morton encoding by Sundar et al that aid their implementation [6]. They form Morton ids via an *interleaving* of bits corresponding to the index of an octant at a given level, finally they append $\lfloor \log_2 D_{\max} \rfloor + 1$ bits corresponding to the level of the octant, where $D_{\max}$ is the maximum depth of the octree. This is best illustrated through an example, consider an octant at octree level 2, with the following index tuple $(1, 2, 3)$, corresponding to it's x y and z indices. Then its Morton ID is constructed as,

$$(\underbrace{001}_{x}, \underbrace{010}_{y}, \underbrace{011}_{z}) \tag{1}$$

$$\rightarrow \text{interleaving, } 000011101 \tag{2}$$

$$\rightarrow \text{append level (2), } 000011101010 \tag{3}$$

Constructing Morton ids in this fashion imbibes them with some useful properties for our purposes of tree construction.

...

Algorithms

Sundar et. al break up their approach into two main algorithmic contributions.

1. Constructing a linear octree from a set of points spanning a three dimensional domain.
2. Enforcing an optimal 2:1 balance constraint over the linear octree.

They have tested implementations involving the linearisation and balancing of octrees containing 1×10^9 nodes in ≈ 60s on the Pittsburgh Supercomputing Center's TCS-1 AlphaServer containing 1024 processor units. However, as we shall see, the methods described in [6] overlap for each goal, therefore can't really be considered separately.

The strategy can be discretised into the following steps

1. *Distribute Workload* - Ensure that each processing unit has roughly the same amount of computation to get through.
2. *Balance Subtrees* - For each subtree, enforce the 2:1 balancing constraint.
3. *Merge* - Remove duplicates.

1. *Distribute Workload*

- Begin by sorting based on Morton id, after refinement of domain into uniform grid at the maximum level of refinement $D_{\max}$.
- Distribute these ids evenly across processor nodes.
- Between 'first' and 'last' Morton ids for a given processor node, compute the minimal octants required to span the region.
- Use this as an input to algorithm to find *complete* octree spanning the domain.
- Refine each 'block' - the terminology for a leaf in this final linear octree, to comply with the constraint of fewer than S particles per node

Blocks can then be assigned a weight, corresponding to particles they contain, ready to be distributed to each processor node.

In order to construct a complete linear octree from a set of octants, you just have to complete the region between all subsequent pairs of octants (see alg. 1),

Algorithm 1 Construct Complete Linear Octree From a Set of Octants (Sequential)

```
    Input: A list of Octants $L$
    Output: $R$ the sorted linear octree encompassing those octants.
function CompleteOctree($L$)
RemoveDuplicates($L$)          ▷ Can be achieved with set-like data structure
$L \leftarrow$ Linearise($L$)                             ▷ Remove all overlaps
$L$.pushback($\mathcal{FC}(\mathcal{A}_{finest}(\mathcal{DFD}(\text{root}, L[1])))$)
$L$.pushfront($\mathcal{LC}(\mathcal{A}_{finest}(\mathcal{DLD}(\text{root}, L[\text{len}(L)])))$)
    for $i \leftarrow 1$ to $\text{len}(L) - 1$ do
        $A \leftarrow \text{CompleteRegion}(L[i], L[i+1])$
        $R \leftarrow R + L[i] + A$
    end for
end function
```

Algorithm 2 works by checking sequentially whether a given octant is contained within the ancestors of it's adjacent octant, if not then it is appended to the final non-overlapping octree. The octants are sorted by Morton id, which allows this strategy to work.

Algorithm 2 Remove Overlaps from a Sorted List of Octants

```
    Input: A sorted list of Octants $W$
    Output: $R$, an Octree with no overlaps
function Linearise($W$)
    for $i \leftarrow 1$ to (len($W$)-1) do
        if $W[i] \notin \mathcal{A}(W[i+1])$ then
            $R \leftarrow R + W[i]$
        end if
    end for
    $R \leftarrow R + W[\text{len}(W)]$
end function
```

Finally to ensure that the condition of having fewer than S particles per leaf octant is met, we iterate through the resulting linear octree, and split octants wherever this condition is not met.

Algorithm 3 Construct Linear Octree From a Set of Points (sequential)

Input: A list of points L over a 3D domain, and a parameter for the maximum number of points per leaf octant S.

Output: A complete linear Octree, B.

```
1: function PointsToOctree($L$)
2:     $F \leftarrow$ [Octant($p$, $D_{\max}$), $\forall p \in L$]
3:     for each $b \in B$ do
4:         if NumberOfPoints($b$) $> S$ then
5:             $B \leftarrow B - b + \mathcal{C}(b)$
6:         end if
7:     end for
8: end function
```

2. *Balance Subtrees*

Step (1) results in an balanced linear octree, for many applications it's useful to impose a degree of smoothness over the domain by imposing a balance constraint. A '2:1' balance constraint refers to the fact that two adjacent octree nodes (i.e. which share an edge, a face or a corner) are no more than a level apart in terms of their size. For FMM implementations this constraint leads to fewer operator precomputations.

Once we've obtained a balanced linear octree, balancing it is done via algorithm (4). This works by starting an iteration at the finest level of the octree possible $D_{\max}$, generating the coarsest possible neighbours for octants at this level, and seeing if there is an intersection with octants already present. If there is they are retained in the final tree, then we iterate to the next coarsest level and so forth. At the end of the iteration we'll be left with a balanced tree by definition.

AdaptOctree Software

Foo Bar

Algorithm 4 Balance a Complete Linear Octree (sequential)

Input: An octant N and a partial list of its descendants L
Output: A complete balanced octree R.

```
 1: $W \leftarrow L,\ P \leftarrow \emptyset,\ R \leftarrow \emptyset$
 2: function BalanceOctree$(a, b)$
 3:     for $l \leftarrow D_{\max}$ to $(\mathcal{L}(N) + 1)$ do
 4:         $Q \leftarrow \{x \in W | \mathcal{L}(x) = l\}$
 5:         Sort$(Q)$
 6:         $T \leftarrow \{x \in Q | \mathcal{S}(x) \notin T\}$
 7:         for each $t \in T$ do
 8:             $R \leftarrow t + \mathcal{S}(t)$
 9:             $P \leftarrow P + \{\mathcal{N}(\mathcal{P}(t), l - 1) \cap \{\mathcal{D}(N)\}\}$
10:         end for
11:         $P \leftarrow P + \{x \in W | \mathcal{L}(x) = l - 1\}$
12:         $W \leftarrow \{x \in W | \mathcal{L}(x) \neq l - 1\}$
13:         RemoveDuplicates$(P)$
14:         $W \leftarrow W + P,\ P \leftarrow \emptyset$
15:     end for
16:     Sort$(R)$
17:     $R \leftarrow$ Linearise$(R)$                         ▷ Remove overlaps
18: end function
```

References

[1] Andreas Klöckner et. al. *Boxtree*. URL: https://github.com/inducer/boxtree.

[2] J. Carrier, L. Greengard, and V. Rokhlin. "A Fast Adaptive Multipole Algorithm for Particle Simulations". In: *SIAM Journal on Scientific and Statistical Computing* 9.4 (1988), pp. 669–686. ISSN: 0196-5204. DOI: 10.1137/0909044. URL: http://epubs.siam.org/doi/10.1137/0909044.

[3] *Exascale Computing for System-Level Engineering*. URL: https://excalibur-sle.github.io/.

[4] Ilya Lashuk et al. "A massively parallel adaptive fast multipole method on heterogeneous architectures". In: *Communications of the ACM* 55.5 (2012), pp. 101–109.

[5] Dhairya Malhotra and George Biros. "A Distributed Memory Fast Multipole Method for Volume Potentials". In: *ACM Transactions on Mathematical Software* 43.2 (2016), pp. 1–27. ISSN: 0098-3500. DOI: 10.1145/2898349. URL: http://dx.doi.org/10.1145/0000000.0000000https://dl.acm.org/doi/10.1145/2898349.

[6] Hari Sundar, Rahul S Sampath, and George Biros. "Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel". In: *SIAM Journal on Scientific Computing* 30.5 (2008), pp. 2675–2708.