Point Cloud Compression Optimization

Milind Kulkarni
Purdue University, USA
milind@purdue.edu

Benjamin Gottfried*
Purdue University, USA
bg@purdue.edu

Kirshanthan Sundararajah
Purdue University, USA
ksundar@purdue.edu

Abstract

This paper improves upon previous methods to encode and compress point cloud streams by exploiting spatial and temporal redundancy in point data. When implemented, a point cloud stream can be represented by a series of octrees that represents the 3-dimensional objects in the scene captured in each frame. The effectiveness of the previously presented encoding method is examined, and its computational efficiency is analyzed and improved. This paper presents methods of exploiting task and data parallelism that found in various procedures used in the serial implementation of the encoding algorithm.

1 Introduction

Point clouds are almost ubiquitously used in any application involving representing spatial data in three dimensions. Hence, they are paramount to many applications in graphics, for example, representing a football field, on which to perform calculations and draw additional objects, such as the line of scrimmage. One of the most prolific and intuitive uses of point clouds is in the sensor data generated by the Microsoft Kinect.

The Kinect continuously scans a room and represents the scene as a 3-dimensional space at a rate of 30 frames per second. Each frame captured can be represented as a set of points, where each point (x, y, z) represents whether or not some object is occupying the space centered at that point at that time frame. The set of points at each time frame make up a point cloud, and when multiple, consecutive point clouds are chained together, the resulting data structure is referred to as a point cloud stream.

1.1 Data Representation

There already exists a convenient file format (.pcd) for defining point clouds. This file consists of a small header, which we will mostly ignore, along with a set of n (x, y, z) coordinates that make up the cloud. We can represent this point cloud as an octree, where each octree node corresponds with a bounded, 3-dimensional space (a voxel). In this paper, "voxel," "octree," and "octree node," will be used almost interchangeably.

The root of the octree is the entirety of the 3-dimensional space represented by the scene, and each child maps to a partitioned subvoxel of its parent voxel. The 8 children of this octree correspond to 8 subvoxels that equally partition the space (not contained points) contained by the parent. These subvoxels are recursively partitioned until the leaf nodes are at the predefined maximum recursion-depth. Hence, all leaf nodes (resp. voxels) are at the same depth (resp. size). This is necessary for the correctness of our algorithms. Each subvoxel is bounded by its parent voxel and the midpoints between each maximum and minimum x, y, and z coordinate that bounds the parent voxel.

Each octree node exists if and only if there exists at least one point from the original point set that is contained within its voxel's bounds, otherwise that specific node is null. Each octree node contains an array of pointers to its children, which are null if their subvoxels are unpopulated. Hence, the data contained at each octree node can be efficiently represented as an 8-bit integer, where each bit is set if and only if the corresponding child pointer exists. The struct representing an octree node is as follows:

```
typedef struct _OctreeNode {
  struct _OctreeNode* children[8];
  char data;
} OctreeNode;
```

Let d be the maximum recursion-depth. Given a point set, an octree can be constructed in $O(nd)$ time, and $O(8^d)$ space. This construction is performed by recursively inserting each point p into the subvoxel that bounds p, setting the data bit in the current node that corresponds to the newly populated subvoxel.

*Ben Gottfried performed this work as an undergraduate at Purdue University. He is seeking admission to CS PhD programs, as well as internships for Summer 2021.

Algorithm 1 Create an octree by recursively inserting each point, creating nodes and setting bits as necessary

```
1:  procedure CreateOctree(PointSet P)
2:      procedure InsertPoint(OctreeNode T, Point p)
3:          if T is not a leaf node then
4:              ▷ Find the suboctant c that contains p
5:              ▷ Set the bit in T.data corresponding to c
6:              InsertPoint(c, p)
7:      OctreeNode T
8:      foreach (Point p : P) do
9:          InsertPoint(T, p)
10:     return T
```

If the only important data associated with a point cloud is whether or not a given point in space is populated, there exists an efficient way to compress a set of (x, y, z) coordinates. The compression will lose some small details, like how many points exist in each leaf voxel, or other metadata associated with a point, like color, but in the case of our Kinect example, these losses are acceptable, provided our recursion depth is deep enough to allow a sufficiently granular resolution to make decisions based on the positions of objects.

In **Section 3**, we will argue why it is most efficient to maintain the relative bounds of each captured scene dynamically, rather than keeping maximum and minimum bounds fixed. Thus, the bounds of the scene associated with each point cloud are defined by the maximum and minimum x, y, and z coordinates, plus and minus approximately 5%, respectively, so that no point lies exactly on the borders defining the bounds.

2 Octree Serialization

The octree representing the point cloud can be serially encoded using the following algorithm: Informally, first append the scene's bounds $\vec{B}$ to a byte list S, then perform a breadth-first traversal of all non-leaf nodes, appending the data byte of each node visited to S.

Algorithm 2 Serialize an octree into a byte list

```
1:  procedure Serialize(OctreeNode T)
2:      ByteList S
3:      S.append(B⃗)
4:      Queue Q
5:      Q.enqueue(T)
6:      while (¬Q.empty()) do
7:          OctreeNode t ← Q.dequeue()
8:          if (t is not a leaf node) then
9:              S.append(t.data)
10:             foreach (OctreeNode c : t.children) do
11:                 Q.enqueue(c)
12:     return S
```

The serialized octree can then be decoded by essentially performing the inverse of **Algorithm 2**. Each byte b creates a new tree node t with b as its data byte, which gets traversed breadth-first.

Any octree can then be restored into a sufficiently accurate point cloud (.pcd file) by assigning a point to either the centroid or a specific corner of every populated leaf voxel.

Algorithm 3 Decode an octree from a byte list

```
1:  procedure Deserialize(ByteStream S)
2:      Octree T
3:      Queue Q
4:      Q.enqueue(T)
5:      foreach (Byte b : S) do
6:          OctreeNode t ← Q.dequeue()
7:          t.data ← b
8:          foreach (set bit i : b) do
9:              OctreeNode t_i
10:             Q.enqueue(t_i)
11:             t.children[i] ← t_i
12:     return T
```

If defining a point cloud as an octree with a maximum resolution has sufficient precision, and if the only pertinent data is the presence of an object at a location, then serializing the octree as described is almost always a more efficient use of space. We will argue this claim in the general case, but first, consider the format of point cloud data in applications like the Kinect, or a football game. These intuitions will further justify why a compression scheme like this is optimal.

Most of the space in any scene is vacuous. There are only 1-4 players standing in a room, or moreover, 2 small teams taking up an entire football field. This leads to fairly sparse data, but for the objects that do exist, they are usually contiguous, uniform chunks of mass. This implies that the overwhelming majority of the voxels for the data we are compressing fall into one of two cases:

- The voxel is completely empty, so a large volume of space, or an octree node with shallow depth, is empty, so it is written off and ignored.
- There exists an object in a mid-level octree node, whose suboctants are nearly all populated. A contiguous object implies that the leaf voxel for the center of that object is populated, as well as a sizable number of adjacent leaf voxels are all populated. The data bits for parent voxels containing this object are comprised of almost entirely '1' bits, and the least common ancestor, or the smallest containing subvoxel, of the populated leaf nodes would be relatively small.

Using this intuition, we will now prove the claim that even the uncompressed serialization is more space efficient than the standard point cloud format (.pcd), which uses a set of floating point x, y, and z coordinates. Let C_P denote a point cloud represented as a .pcd file, and let C_S denote the same file, but serialized using **Algorithm 2**. The size of a single point in C_P is always a constant 12 bytes, assuming the size of a float is 32 bits, but the total size of the set of points in C_S varies. We assume that our scene is comprised of two types of objects: larger objects of contiguous mass, like humans or furniture, and isolated objects that only populate a small number of adjacent leaf voxels, like balls in a game.

For a single, isolated point in C_S, there are d bytes that recursively partition the space until the leaf voxel is defined as populated. In the worst case, if our scene consisted of exclusively small, isolated points, $|C_P| = 12n$ and $|C_S| = dn$, so our encoded octree would be larger than the .pcd file by a factor bounded by $\frac{d}{12}$. If $d < 12$, it is already more efficient to store C_S, regardless of how the data is laid out.

But for a larger, contiguous object, the overhead of the bytes representing internal nodes is amortized because the least common ancestor of adjacent points is just their parent, so the bytes in their serialization are only written once. Let x be a larger, contiguous object. C_P must contain at least as many points to comprise x as there are populated leaf voxels in C_S. Suppose to the contrary that C_P uses fewer points to represent x than C_S. Then there is at least one subvoxel in C_S that contains a point not in C_P. However, we defined C_S to be the serialization of C_P, and using **Algorithm 2**, a leaf voxel is never created unless there exists a point bounded exactly by that leaf voxel; a contradiction. Hence, the number of points in C_P that compose x $|x \subset C_P| \geq |x \subset C_S|$, and by our assumption that x is a larger object of contiguous points, the overhead to get the least common ancestor of the points in $x \subset C_S$ is shared by those points, so the factor $\frac{d}{12}$ is amortized.

As d increases, the overhead for an isolated point increases, but by our assumption that x exists in C_P, which implies x exists in C_S, as d increases, C_P must contain even more points to achieve the same level of granularity. Since x is contiguous and it must have as many as points C_S, the depth of the least common ancestor of $x \subset C_S$ is constant, so the compression ratio only increases.

3 Stream Compression

In this section, we will present a serial algorithm for encoding a point cloud stream [4], rather than sending each point cloud as either a .pcd file, or even a serialized octree as presented in **Section 2**.

We introduce the notion of taking the symmetric difference between two octrees. Intuitively, the symmetric difference between two trees is the set of nodes that exist in only one of the two trees. Voxels that are either vacant or populated in both are not included in the symmetric difference. Why this operation is useful will become clear shortly. First, we present an algorithm that takes two octrees T_i, T_j and calculates the symmetric difference between the two. The result returned is a list of bytes, where a '0' bit represents no change from the previous tree, and a '1' bit represents a tree node that in T_j that was not in T_i, or vice versa. Notationally,

$$T_i \triangle T_j = \delta_{i,j}$$

Algorithm 4 Calculate the symmetric difference between 2 octrees

```
1:  procedure CalcSymDiff(OctreeNode T_i, OctreeNode T_j)
2:      ByteList δ
3:      Queue Q
4:      Q.enqueue((T_i, T_j))
5:      while (¬Q.empty()) do
6:          t_i, t_j ← Q.dequeue()
7:          if (t_i, t_j are not leaf nodes) then
8:              δ.append(t_i.data ⊕ t_j.data)
9:              C⃗ ← t_i.children OR t_j.children
10:             foreach (OctreeNode (c_i, c_j) : C⃗) do
11:                 Q.enqueue((c_i, c_j))
12:     return δ
```

This algorithm is correct in its computation of $\delta_{i,j}$ since every populated voxel in both trees T_i, T_j is visited during the breadth-first traversal exactly once. Upon visitation, the data bytes of the current nodes in both trees are `XOR`ed, yielding the change between those nodes, which is the essence of symmetric difference between octrees. Furthermore, if one tree node does not exist in either T_i or T_j, then the data byte of the tree node that does exist is `XOR`ed with 0, yielding itself. This is intuitively correct since if a subtree did not previously exist, or a subtree newly exists, that subtree will need to change completely from the previous tree. Hence, by our definition of symmetric difference, **Algorithm 4** is correct.

The key intuition behind the algorithm for encoding a series of point clouds is that the objects in the scene are changing very little between two consecutive frames. Hence, the bits that are set, respectively cleared, in T_i are mostly the same as in T_{i+1}. From this observation, we can deduce that the byte list $\delta_{i,j}$ will be almost entirely '0' bits. This lends itself extremely well to run-length encoding to further compress the size.

Let us also revisit the idea touched upon at the end of **Section 1.1** that it is beneficial to dynamically update the bounds of the scene for every point cloud. A pattern that might frequently occur in these 3-dimensional data-capturing sensors is translations. A sensor might pan around a football game, or a person playing Kinect might jump. The relative locations of all objects in the scene are changing with respect to the sensor, but their orientations are mostly static. Consequently, by storing the maximum and populated coordinate points in each dimension, a box can be maintained around the objects in focus, and relative to that focus box, the objects are not changing. This means that despite the scene changing, we store a constant value with our encoded scene (2 floats per dimension), but our symmetric difference focus boxes between adjacent frames is still mostly '0' bits, which maintains the effectiveness of RLE compression.

The following is the complete algorithm for encoding and compressing a stream of point clouds. Informally, we store the encoded initial scene T_0 as a byte list, then for each subsequent point set P_i, create an octree T_i, write its bounds $\vec{B}_i$, calculate the symmetric difference between the previous and current trees $\delta_{p,i} = T_p \triangle T_i$, compress $\delta_{p,i}$ using RLE, then write it.

Algorithm 5 Compress an incoming point cloud stream

```
1:  procedure StreamCompress(PointSet[ ] P)
2:      OctreeNode T_0 ← CreateOctree(P_0)
3:      ByteList S_0 ← Serialize(T_0)
4:      Write(S_0)
5:      OctreeNode T_p ← T_0
6:      foreach (PointSet P_i : P) do      ▷ as new P_i stream in
7:          OctreeNode T_i ← CreateOctree(P_i)
8:          Write(B⃗_i)
9:          δ_{p,i} ← CalcSymDiff(T_p, T_i)
10:         Write(RLE(δ_{p,i}))
11:         T_p ← T_i
```

3.1 Decompression

We have shown that this algorithm is a space-efficient way to encode a point cloud stream, but the encoding is useless if we cannot recover the original point sets. Accordingly, we need to show that the original point sets can be reconstructed using only our encoded stream. Let the number of additional point sets contained in the stream after P_0 be n. Since, any octree can be restored into a sufficiently accurate point cloud, to show that each points et $P_i, i \leq n$ can be reconstructed, it suffices to show that each corresponding octree $T_i, i < n$ can be recovered from the encoding. We prove this by strong induction:

For the base case, we will show that T_1 can be reconstructed from T_0. First, reacquire T_0 by performing **Algorithm 3**. Next, decode $\delta_{0,1}$ by undoing the RLE. Now, assume there exists an algorithm to reconstruct an octree T_i given the previous octree $T_p, p = i - 1$ and the symmetric difference between the two octrees $\delta_{p,i} = T_p \triangle T_i$. If this algorithm does exist, then we have everything we need to use it to reconstruct T_1. Therefore, our base case holds.

For the inductive step, we assume that we can reconstruct all trees $T_r, 1 \leq r \leq k$. To discharge our inductive hypothesis, we must show that it is possible to reconstruct T_{k+1}. As long as $k + 1 \leq n$, $\delta_{k,k+1}$ is somewhere in our encoded byte stream. Since we assumed T_k can be reconstructed, we can use the aforementioned algorithm to reconstruct T_{k+1} using T_k and $\delta_{k,k+1}$. Hence, our inductive case holds and any tree T_i can be reconstructed $\forall i, i \leq n$.

In order to complete this proof, we need to show that the previously utilized, but undefined, adjacent reconstruction algorithm exists and is correct. Let T_p, T_i be the previous octree and the octree currently being constructed, respectively. Informally, iterate across each byte $b \in \delta_{p,i}$, while performing a breadth-first traversal of both T_p and T_i. Let the currently visited node of T_p (resp. T_i) be t_p (resp. t_i). Then t_i.`data` is $b \oplus t_p$.`data`. Next, iterate across each bit in b and the data byte of t_p. If only one of the bits is set at a position j in both bytes, then t_i has a child at position j. Otherwise, the child of t_i at position j is null. Irrespective of the nullity of both the previous and current tree's children at j, enqueue the children, or null in its place, then continue the traversal for all $b \in \delta_{p,i}$.

Algorithm 6 Reconstruct an Octree given the previous octree and their symmetric difference

```
1:  procedure Reconstruct(OctreeNode T_p, ByteList δ_{p,i})
2:      OctreeNode T_i
3:      Queue Q
4:      Q.enqueue((T_p, T_i))
5:      foreach (byte b : δ_{p,i}) do
6:          t_p, t_i ← Q.dequeue()
7:          t_i.data ← b ⊕ t_p.data
8:          byte c ← b OR t_p.data
9:          foreach (set bit j : c) do
10:             OctreeNode t_j ← b[j] ⊕ t_p.data[j]
11:             t_i.children[j] ← t_j
12:             Q.enqueue((t_p.children[j], t_j))
13:     return T_i
```

Before we prove **Algorithm 6** to be correct in the general case, let us consider the special case of where the previous tree is null. This is a non-issue, since a null octree is treated the same as an octree with no

populated voxels. This would lead to every set bit in the symmetric difference byte list to correspond with a populated node in the new tree. Hence, it can be seen that $\delta_{\emptyset,k} = S_k = \textbf{Serialize}(T_k)$. We now prove this algorithm in the general case to be correct by induction:

For the base case, consider two octrees T_i, T_j of depth $d = 1$, such that, by our definition of depth, all of T_x's children, $x \in \{i, j\}$ are leaf voxels. This implies that their full serializations using **Algorithm 2** are just T_x.data, $x \in \{i, j\}$. Hence, $\delta_{i,j} = T_i \triangle T_j = T_i$.data $\oplus T_j$.data, where $|\delta_{i,j}| = 1$ byte. Let that byte be b. By definition of **Algorithm 4**, the only set bits in b are the positions j, such that T_x.children$[j]$ exists for one and only one $x \in \{i, j\}$. Accordingly, running **Algorithm 6** with T_i and $\delta_{i,j}$ as input yields a single octree node T_k, such that T_k.data $= T_i$.data $\oplus\, b$.
but $b = T_i$.data $\oplus\, T_j$.data, so
T_k.data $= T_i$.data $\oplus\, T_i$.data $\oplus\, T_j$.data $= T_j$.data
which implies $T_k = T_j$. Therefore, the base case holds.

For the inductive case, assume a tree t_j of depth $d = k$ can be reconstructed, given a tree t_i, also of depth $d = k$, and $\delta_{i,j} = t_i \triangle t_j$. We will use this assumption to then show that given a tree T_i of depth $k + 1$ and its respective $\delta_{i,j} = T_i \triangle T_j$, we can reconstruct T_j. By our inductive hypothesis, a tree of depth $d = k$ can be reconstructed. The subtree of T_j consisting of only the internal nodes T_x is a an octree of depth $d = k$, so we can we reconstruct that. During the execution of **Algorithm 6**, for all nodes considered, their children are enqueued, so there exists a bijective mapping of all of the remaining bytes in $\delta_{i,j}$ not used to reconstruct T_x to the tree nodes in $T_i \setminus T_x$. All of the nodes in $T_i \setminus T_x$ are all of depth $d = k + 1$, so they are all parents of leaf nodes of T_i. These individual parents of leaf nodes nodes are also subtrees of depth $d = 1$, so using their mapped byte, their individual subtrees can be reconstructed in the same manner as the base case. Since T_x can be reconstructed, and all of the subtrees coming from the leaves of T_x can be reconstructed, then T_j can be reconstructed, discharging our inductive hypothesis. Therefore $\forall d, d \geq 0$, trees of depth d can be reconstructed, so **Algorithm 6** is correct.

This completes the proof for the feasibility and correctness for our stream decompression algorithm. □

4 Parallel Optimizations

The majority of the paper thus far has been discussion and formal proof of previously known and tested serial algorithms. We now look toward throughput-oriented hardware to extract more performance from the spatial and temporal redundancies in point cloud streams, but we must first redesign our algorithms accordingly.

4.1 Parallel Compression

Since our compression algorithm is a streaming algorithm, it will always be bottlenecked by the rate at which new data points enter the compression system. If one had all the entire series of point sets at the beginning, then obviously the entire encoding and compression process could happen simultaneously for all point sets, since all of the data sets are independent. But we will not assume that impracticality, thus maintaining the streaming capabilities of our compression algorithm. Consequently, our high-level compression procedure is identical to that presented in **Algorithm 5**, but we can take advantage of our additional threads to further optimize the subroutines utilized.

The first potential optimization I would like to explore is constructing the octree in parallel. I argue this is possible by first stably sorting all points in x, y, then z order, partitioning them into each subvoxel of the root in constant time, setting the data bit for subvoxel with at least one point, then recursing. Each thread would be responsible for performing the same task on exactly one subvoxel, so this exhibits great data parallelism.

Algorithm 7 Create an octree in parallel by sorting and partitioning all points, setting data bits, then recursing

```
1: procedure ParCreateOctree(PointSet P)
2:     OctreeNode T
3:     ▷ Stably sort all p ∈ P by (x, y, z) coordinates
4:     P⃗_c ← Partition(P)
5:     T.data ← |P⃗_c|
6:     foreach (PointSet P_c : P⃗_c) do
7:         T.children[c] ← spawn ParCreateOctree(P_c)
8:     sync
9:     return T
```

This parallel elision of **Algorithm 1** is still correct because each after being partitioned, each thread operates on a disjoint set of points, since each point can exist on only one partition. This algorithm would undoubtedly be less efficient if executed sequentially, but since each subvoxel can be considered simultaneously, this is a performance upgrade.

The next improvement made is one to the subroutine that encodes the serialization of an octree, **Algorithm 2**. The optimization takes advantage of parallel breadth-first traversal paradigm first designed by Jo et al [3]. The essence of the paradigm is each computation performed at every level of the tree happens simultaneously, syncing all threads before recursing to the next level. Each thread locally executes in a depth-first manner, which further improves upon the spacial locality often missing in serial breadth-first traversals.

Since the same operations are all performed in each computation, this optimization exploits great data par-

allelism. The soundness of the following algorithmic paradigm is implicit because it has been shown that the parallel execution of breadth-first search holds using this technique. Since **Algorithm 2** is just a form of breadth first traversal over an octree, the paradigm can be applied here. Since the following algorithm is non-trivial and relies on a few newly presented concepts, the following paragraphs will outline **Algorithm 8** before pseudocode is given.

First, we must define the new notion of a position sequence. The position sequence is the string of indices required to get from the root of the octree to the node for which the position sequence corresponds. Thus, the position sequence of the root is the empty string λ, and the number of characters in the position sequence is its depth. Let us also define the serialization Z_l for an individual level l of the octree. Since **Algorithm 2** executes breath-first, it is impossible to have a serialization that contains a data byte from a level x prior to a byte from a level y, such that x is deeper in the octree than y. Z_l can also be defined as the data bytes of all tree nodes on level l, sorted with respect to each node's position sequence. Hence, the full serialization Z of an octree of depth d is simply $\sum_{l=0}^{d} Z_l$, where concatenation is used instead of addition. This notion of Z_l will become important later.

Algorithm 8 accepts a thread block as input, where each thread contains an octree, a position sequence (which will be soon be defined), and a set containing bytes and position sequences. Before handling the individual threads, the algorithm must create a new thread block B_n to contain the threads for all of the nodes in the next deepest level of the tree. It also creates a new set, called the FlatSet F, which contains the sets of bytes for all of the threads in the current thread block.

Algorithm 8 then executes all of the threads in the thread block simultaneously for all of the nodes in the thread block's level of the tree, performing the following additional computations: Let t be an arbitrary thread, and let T, S, P be t's octree, byte set, and position sequence, respectively. First, T's data and P are added to S. Then, S is added to F. Note that all set add operations must be synchronized, so a mutex lock or another synchronization primitive must be used. Next, t creates its own empty byte set S_t. For each populated child c of T, a new thread is added (but not yet executed) to B_n that consists of c, S_t, and the concatenation of P with the index used to get c.

After performing computations on each thread in the previous thread block, **Algorithm 8** computes in parallel the following: It recurses, calling itself again with B_n as its input. Additionally, it computes the result of a "flatten" operation on F. Let this operation be **Flatten**(F) and let it return Z_l, where l is the level of the octree on which F is defined. We will prove this to be feasible in the following paragraph. **Flatten** returns Z_l, so the result of **Flatten** can be concatenated with the output of the next recursive call to the algorithm to yield $\sum_{l=0}^{d} Z_l = Z$ which is the full serialization of T.

In order to prove the correctness of **Algorithm 8**, we must define the internals of **Flatten**. F is a set of sets, such that the union of all its members contain all of the data bytes for the octree level l for which F is defined. Sorting those data bytes in order of their position sequences yields Z_l. Therefore,

$$\bigcup_{S \in F} S = \{b | b \in Z_l\} \rightarrow \textbf{sorted}(\bigcup_{S \in F} S) = Z_l$$

which implies, concatenating a series of calls to **Flatten**, one for each level of the octree, yields the full serialization of the octree:

$$\sum_{l=0}^{d} Z_l = Z$$

Algorithm 8 Serialize an octree in parallel

```
1:  ▷ Each thread contains: OctreeNode, PositionSeq, ByteSet
2:  procedure ParSerialize(ThreadBlock B)
3:      ThreadBlock B_n
4:      FlatSet F
5:      foreach (Thread t : B) do
6:          if (t.T is not a leaf node) then
7:              t.S.add(t.T.data, t.P)
8:              F.add(t.S)
9:              ByteSet S_t
10:             foreach (OctreeNode c : t.T.children) do
11:                 B_n.add(Thread(c, S_t, t.P + c.pos)
12:     return Flatten(F) + ParSerialize(B_n)
```

An analogy can be drawn between **Algorithm 2** and **Algorithm 4**, since the internals of the algorithms for serialization and symmetric differentiation are very similar — the key difference being in **Algorithm 2**, only the raw data byte is appended, whereas in **Algorithm 4**, the two tree nodes' data bytes are `XOR`ed. The same analogy can be drawn between **Algorithm 8**. Since **Algorithm 4** is also a breadth-first traversal, we can apply this parallelized breadth-first traversal paradigm here, as well, to further optimize our speed of computing the symmetric difference between two octrees. **Algorithm 9** is correct for the same reasons **Algorithm 8** is correct, along with the same reasons why **Algorithm 4** works as a serial, breadth-first traversal.

Using our parallelized subroutines, the improved complete algorithm for encoding and compressing a stream of point clouds, which is **Algorithm 5** with its serial subroutines replaced, is **Algorithm 10**.

Algorithm 9 Calculate the symmetric difference between 2 octrees in parallel

1: ▷ Each thread contains: OctreeNode (i, j), PositionSeq, ByteSet
2: **procedure ParSymDiff**(ThreadBlock B)
3: ThreadBlock B_n
4: FlatSet F
5: **foreach** (Thread $t : B$) **do**
6: **if** ($t.T_i$ and $t.T_j$ are not a leaf nodes) **then**
7: $t.S$.add($t.T_i$.`data` $\oplus$ $t.T_j$.`data`, $t.P$)
8: F.add($t.S$)
9: ByteSet S_t
10: $\vec{C} \leftarrow t.T_i$.`children` `OR` $t.T_j$.`children`
11: **foreach** (OctreeNode $(c_i, c_j) : \vec{C}$) **do**
12: B_n.add(Thread($(c_i, c_j), S_t, t.P + c$.`pos`)
13: **return Flatten**(F) + **ParSymDiff**(B_n)

Algorithm 10 Compress an incoming point cloud stream in parallel

1: **procedure StreamCompress**(PointSet[ ] P)
2: OctreeNode $T_0 \leftarrow$ **ParCreateOctree**(P_0)
3: ThreadBlock $B \leftarrow$ThreadBlock$(T_0, \lambda, \emptyset)$
4: ByteList $S_0 \leftarrow$ **ParSerialize**(B)
5: **Write**(S_0)
6: OctreeNode $T_p \leftarrow T_0$
7: **foreach** (PointSet $P_i : P$) **do** ▷ as new P_i stream in
8: OctreeNode $T_i \leftarrow$ **ParCreateOctree**(P_i)
9: **Write**$(\vec{B}_i)$
10: $B \leftarrow$ThreadBlock$((T_p, T_i), \lambda, \emptyset)$
11: $\delta_{p,i} \leftarrow$ **ParSymDiff**(B)
12: **Write**(**RLE**$(\delta_{p,i})$)
13: $T_p \leftarrow T_i$

4.2 Parallel Decompression

Unfortunately, the other subroutines called upon during stream decompression cannot be efficiently parallelized. Namely, **Algorithm 3**, the subroutine to decode a serialized octree, and **Algorithm 6**, the subroutine to reconstruct an octree given the previous octree and their symmetric difference. The inability to extract parallelism from these subroutines can be attributed to the nature of their most essential operations: performing computations while traversing a linked list. This operation's inefficacy to be parallelized is due to its interdependence of data from past, present, and future tasks' operations. Thus, those subroutines must execute sequentially.

However, when decompressing and an encoded point cloud stream, it is much more reasonable to make the assumption that the entirety of the stream is initially available. This allows for more sophisticated algorithms to be used during decompression. The algorithm we will propose to optimize point cloud stream decompression makes use of a classic algorithm in parallel programming: Prefix sum [1]

Before diving in to the complete algorithm, we must first define a new operator $+_M$ which will be used to merge symmetric differences between octrees. Although in set theory, the symmetric difference operation is a binary operator $\triangle : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S}$ that maps the relation of two sets to another set, but the way we defined the symmetric difference operator with respect to octrees does not map the relation two octrees to another octree, but rather an abstract δ. This is because in the calculation of symmetric difference, **Algorithm 4**, the mapping between octrees is not surjective because unpopulated octree nodes are ignored to same space in the encoding. Consequently, δ does not carry meaning on its own and must be used to build off of an existing octree to yield another octree. Let Δ be the set of all possible symmetric differences δ. Therefore, let $+_M$ be a binary operator such that $+_M : \Delta \times \Delta \rightarrow \Delta$, and $\delta_{i,j} +_M \delta_{j,k} = \delta_{i,k}$.

We now present an algorithm to compute $\delta_{i,j} +_M \delta_{j,k} = \delta_{i,k}$. Informally, the algorithm consists of walking four byte lists; two copies of $\delta_{i,j}$ and two copies of $\delta_{j,k}$. For each δ, there is a parent list p responsible for enqueuing the appropriate byte, and a child list c responsible for keeping a pointer at the next byte in the byte list to be merged. When a bit is set in either parent byte, if they have a difference byte in there child list corresponding to that set bit in the parent byte, a new byte is created which is the `XOR` of any existing, corresponding child byte, which is then appended to the resulting byte list.

Algorithm 11 Merge two symmetric difference results

1: **procedure Merge**(ByteList $\delta_{i,j}$, ByteList $\delta_{j,k}$)
2: ByteList $\delta_{i,k}$
3: ByteList $p_0, c_0 \leftarrow \delta_{i,j}$
4: ByteList $p_1, c_1 \leftarrow \delta_{j,k}$
5: **foreach** (byte $b_0, b_1 : p_0, p_1$) **do**
6: **foreach** (set bit $x : b_0$ `OR` b_1) **do**
7: byte $r \leftarrow 0$
8: **if** ($b_0[x]$ is set) **then**
9: $r \leftarrow r \oplus c_0$
10: $c_0 \leftarrow c_0$.`next`
11: **if** ($b_1[x]$ is set) **then**
12: $r \leftarrow r \oplus c_1$
13: $c_1 \leftarrow c_1$.`next`
14: $\delta_{i,k}$.append(r)
15: **return** $\delta_{i,k}$

We now argue why **Algorithm 11** correctly merges the symmetric differences $\delta_{i,j}$ and $\delta_{j,k}$ together to form $\delta_{i,k}$. Both byte lists are walked sequentially in the order they were formed, meaning in the order an octree is

traversed. Since two separate pointers are used to walk each linked list, when a set bit is encountered, a new byte is inserted into the resulting byte list in its proper order, because the octree is being traversed in the correct order. This new byte either consists of a difference byte of a single difference list, if that bit was only set in one parent list or it is the XOR of the two child difference bytes because if the tree is being traversed in order, and a bit is set in the same place, then the child difference bytes must both correspond to the octree node. Furthermore, if T_i and $\delta_{i,k}$ were used to reconstruct T_k, then the reconstruction would match the original tree, because the bits set in $\delta_{i,j}$ first toggle octree nodes in T_i yielding T_j, then $\delta_{j,k}$ toggles bits T_j finally yielding the correct T_k. Thus, $\delta_{i,j} +_M \delta_{j,k} = \delta_{i,k}$.

As explained in the beginning of this section, algorithms that mainly consist of walking linked lists like **Algorithm 11** are woefully condemned to sequential execution. However, this is not a concern because a large amount of parallelism will be extracted via the next algorithm we present.

The final step before $+_M$ can be used in prefix sum is that the binary operator used in place of addition must be associative. Hence, we must prove the associativity of $+_M$. An operator $\circ$ is associative if $\forall a, b, c \in S$, we have that $a \circ (b \circ c) = (a \circ b) \circ c$. Let $\circ$ be $+_M$, S be Δ, a, b, c be $\delta_{i,j}, \delta_{j,k}, \delta_{k,l}$, respectively. Based on the correctness of **Algorithm 11**, we have that:

$$\delta_{i,j} +_M (\delta_{j,k} +_M \delta_{k,l}) = (\delta_{i,j} +_M \delta_{j,k}) +_M \delta_{k,l}$$

$$\delta_{i,j} +_M \delta_{j,l} = \delta_{i,k} +_M \delta_{k,l}$$

$$\delta_{i,l} = \delta_{i,l}$$

Therefore, $+_M$ is associative.

We now present the Prefix Merge algorithm, which is prefix sum with its operator being $+_M$ over the set Δ. Our end goal with Prefix Merge is to reclaim the most accurate representations of the original point clouds given an encoded T_0 and $\forall i, 1 \leq i \leq n$, we have $\delta_{i-1,i}$.

Let $D[i], 1 \leq i \leq n$ be an array of symmetric differences $\delta_{i-1,i}$. By using the prefix sum algorithm with $+_M$, we can transform D such that $\forall i, D[i] = \delta_{0,i}$.

Algorithm 12 Compute the parallel prefix sum over D using the binary operator $+_M$

```
1: procedure PrefixMerge(ByteList[ ] D)
2:     Integer n ← D.length
3:     for stride ← 1 to n by stride do
4:         for i ← stride to n by stride×2 do
5:             for j ← 1 to stride do
6:                 spawn D[i + j] ← Merge(D[i], D[i + j])
7:         sync
8:     return D
```

Using D, every T_i, and subsequently P_i, can be reconstructed in parallel by simultaneously invoking n instances of **Algorithm 6**, octree reconstruction, followed by assigning a point to the centroid of each populated octree node in T_i, and writing the new point set as a .pcd file. Therefore, our complete stream decompression algorithm is the following:

Algorithm 13 Decompress a compressed octree stream in parallel using Prefix Merge

```
1: procedure ParDecomp(OctreeNode T_0, ByteList[ ] D)
2:     D ← PrefixMerge(D)
3:     for i ← 1 to n do
4:         spawn Write(ToPcd(Reconstruct(T_0, D[i])))
5:     sync
```

5 Experimental Evaluation

This will be completed once I nail down the cilk [2] and CUDA implementations this summer to compare benchmarks to my serial elision...

6 Conclusion

Parallel execution runs faster than serial. More details to follow after the experimental evaluation is completed.

References

[1] Blelloch, Guy E. "Programming Parallel Algorithms." *Communications of the ACM*, vol. 39, no. 3, 1996, pp. 85–97.

[2] Frigo, Matteo, et al. "The Implementation of the Cilk-5 Multithreaded Language." *ACM SIGPLAN Notices*, vol. 33, no. 5, 1998, pp. 212–223.

[3] Jo, Youngjoon, et al. "Efficient Execution of Recursive Programs on Commodity Vector Hardware." *ACM SIGPLAN Notices*, vol. 50, no. 6, 2015, pp. 509–520.

[4] Kammerl, Julius, et al. "Real-Time Compression of Point Cloud Streams." *2012 IEEE International Conference on Robotics and Automation*, 2012