Parallel Point Cloud Compression Using Truncated Octree

Naimin Koh, Pradeep Kumar Jayaraman and Jianmin Zheng
School of Computer Science and Engineering, Nanyang Technological University, Singapore
Email: naimin001@e.ntu.edu.sg, pradeep.pyro@gmail.com, asjmzheng@ntu.edu.sg

***Abstract*—Existing methods of unstructured point cloud compression usually exploit the spatial sparseness of point clouds using hierarchical tree data structures for spatial encoding. However, such methods can be inefficient when very deep octrees are applied to sparse point cloud data to maintain low level of geometric error during compression. This paper proposes a novel octree structure called *truncated octree* that improves the compression ratio by representing the deep octree with a set of shallow sub-octrees which can save storage without losing the original structure. We also propose a *variable length addressing scheme*, to adaptively choose the length of an octree's node address based on the truncation level—shorter (resp. longer) address when octree is truncated near the leaf (resp. root) which leads to further compression. The method is able to achieve 40% to 90% compression ratio on our tested models for point clouds of different spatial distributions. For extremely sparse point clouds, the method achieves approximately 7 times higher compression ratio than previous methods. Moreover, the method is designed to run in parallel for octree construction, encoding and decoding.**

***Keywords*-Unstructured point cloud; compression; morton code; octree; parallel processing.**

I. Introduction

As depth sensing technology becomes increasingly advanced and achieves high resolution output, the file size of the resulting point clouds captured by scanners becomes exceedingly large. Unlike structured point clouds where hardware and temporal knowledge can be exploited to achieve higher levels of compression [1], algorithms for unstructured point clouds can only rely on low-level spatial relationship as means of compression. Many algorithms explored this spatial relationship, usually using hierarchical structures, in particular octrees, to represent the spatial relationship. While these methods work well on point cloud that are dense, their compression efficiency decreases significantly on point clouds that are sparse or required deep octrees to minimize quantization error. One must not confuse the sparseness of a point cloud distribution with that of a guaranteed small file size. Large scale scans of entire city blocks can easily cover square kilometers of area that even when sparsely subsampled, can result in a large point cloud file.

This paper presents a novel point cloud encoding algorithm that can flexibly handle point clouds of any sparsity while still accounting for both compression rate and quantization quality. The method is designed to run in parallel on modern multicore CPUs to alleviate the encoding and decoding time for large point clouds leading to 3 times speedup during the encoding phase, and 2 times speedup in the decoding phase. Our encoding scheme uses an octree as the basis. The octree is not only beneficial for downstream techniques such as 3D reconstruction, but also leads to an intuitive compression algorithm where the quantization error can be controlled by the depth of the octree. Different from previous works, we use a parallelized bottom-up octree construction instead of the usual top-bottom construction. By this, we are able to avoid unnecessary traversals from the root to leaf node by early termination after encountering an existing parent node during construction. Moreover, due to the upward propagation of nodes creation from leaf nodes, it is relatively easier to parallelize in a multithreaded implementation.

We introduce a novel concept called *truncated octree* that allows us to reduce the file size significantly compared to a full octree. We observed that sparse point data combined with deep octrees leads to long chains of hierarchies in the octree structure. This spoils the benefits brought about by the octree for compression, since the long chains are dedicated to improve the quantization error of lone points. With a truncated octree, the leaf nodes are not referenced with respect to the root, but from nodes that could be in any chosen truncation level. In effect, this results in the creation of multiple smaller sub-octrees, and each node in the truncation level becomes the root of its corresponding sub-octree. The truncation level is chosen automatically during encoding. This does not lead to information loss, and the original octree can be reconstructed via bottom-up octree construction. Furthermore, we design a *variable length addressing* scheme to address each of the sub-octree. Since the truncation level is dynamically chosen, the addressing space required should also be dynamic to optimally match the requirements of the current selected truncation level. If the truncation level is close to the root level, we need significantly fewer bytes to address the nodes of the sub-octrees, thereby increasing the compression ratio. We demonstrate our method against the state-of-the-art based on the compression ratio and encoding/decoding speed achieved on several models of varying complexity and shapes.

II. Related Work

Largely due to the recent advancements in remote sensing, and high resolution 3D scanning and simultaneous localization and mapping (SLAM) technologies driven by recent growth in the autonomous self-driving car industry, a vast amount of point cloud data is generated leading to increased research interest in the field of point cloud compression for efficient data storage, retrieval and processing.

General floating-point data compression: Since point clouds are floating-point data, it is worth considering methods which perform lossless compression of raw floating-point information. Ratanaworabhan et al. [2] proposed a fast lossless floating-point compression that managed to achieve a peak compression rate of $1.5\times$. Lindstrom [3] proposed a block-based float-point compression scheme that can achieve higher compression rate for both lossy and lossless compression modes. Despite the high 1.5–$4.0\times$ compression ratio, these methods should not be used to compress unstructured data such as triangle mesh geometry or unorganized point sets. This is because they exploit the spatial correlation property of data for high level of block compression. Since our focus is on unstructured point clouds, we do not pursue this line of research.

Image based point cloud compression: Another direction of research adapts image and video compression algorithms to work with point cloud data. This was done by projecting the 3D point cloud into 2D images or frames and then performing video compression on the frames. Chen et al. [4] encoded positional, color and normal information of the point cloud by converting them into 3 space filling curves that are encoded into each of the three channels of an RGB image. Shao et al. [5] and Cohen et al. [6] used block-based intra prediction model for multi-frame range images, where each 2D range image is subdivided into smaller blocks and video compression algorithms are used on these smaller blocks as an encoder. Because image or video compression is used as the encoder, the quality of the point cloud highly depends on the quality of the image compression algorithm used. This makes it difficult to control the tradeoff between file size and point cloud quality. In addition, the restriction of using an image as a storage container has a limiting effect on the maximum precision of the information contained within the channels of an image which makes it difficult to extend the method to obtain higher precision. Furthermore, since image and video compression methods make heavy use of spatial correlation of nearby pixel, the efficiency of these methods will be severely penalized when applied to unstructured data. The lack of accurate control over output precision and unsuitability of working with unstructured point cloud is the main reason why we choose to use the more flexible spatial hierarchy-based representation.

Spatial hierarchy based compression: Spatial hierarchical representations such as octrees and kd-trees have been used in point cloud visualization [7] and compression [8]. Many works exploit the uniform hierarchy structure of the various spatial hierarchy to derive higher level spatial information on the point cloud distribution. Octrees in particular are commonly used. Merry et al. [9] focused on uniformly dense and regular point clouds generated by laser scanner using a spanning tree as the spatial hierarchy which is both computational expensive and inflexible to the distribution of the point cloud. Huang et al. [10] used an octree to guide the compression of their geometry and first proposed the idea of child occupancy code that we use in our encoding scheme. Child occupancy code is a 1 bit code for indicating the existence of each possible child of a given node. The code fits nicely into a single byte since there are exactly 8 possible children under each octree node. As opposed to addressing each child directly, child occupancy code in general is a more compact representation compared to addressing individual children (constant 8 bits vs 3-24 bits).

III. Proposed method

Our method has two inputs: a point cloud model M with a set of 3D vertices, and a user specified maximum allowed quantization error threshold E_{M}, which will be used to determine the required depth of the octree D using Equation 1 below. Let B_{M} be diagonal length of the bounding box of M. Then, the maximum geometric error for octree depth D is defined by the following:

$$E_{\mathrm{M}}(D) = B_{\mathrm{M}} \times 0.5^{D} \tag{1}$$

The output is an encoded binary file that contains the encoded octree representation of the point cloud. This file can be decoded once again to retrieve the quantized point cloud as well as its octree structure which can be reused for further processing. Additional file compression algorithms such as ZIP compression can be applied on the binary file if necessary.

Our method is composed of three main components: bottom-up octree construction, octree truncation with variable length addressing, and encoding/decoding. Bottom-up octree construction allows us to construct the octree representation of the point cloud in parallel. Truncating the octree dynamically allows us to discard unnecessary hierarchical structure in a lossless manner from the upper levels of the octree, and generates a set of sub-octrees such that the total memory usage is less than the original octree. Furthermore variable size addressing adaptively changes the bit length required to represent the octree node address to provide even higher compression rate.

A. Bottom-Up Octree Construction

Using a hierarchical structure allows us to distribute the quantization error of the points across each level of the hierarchy, where each level contributes a little bit more information towards reconstructing the final point coordinates.

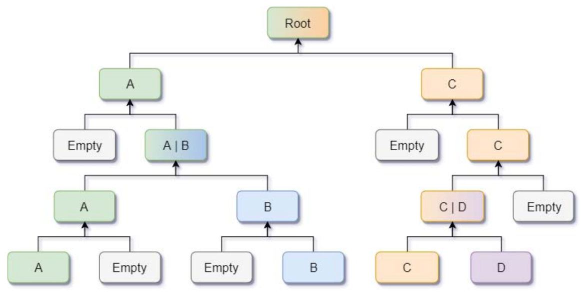


Figure 1: *Early termination of upward propagation in a binary tree. Leaf nodes A, B, C and D are added in order. Nodes B and D had early terminated when they encountered the parent nodes created by A and C respectively.*

Because the information required by each level is small, well structured, and shared by all its children, we can efficiently encode and compress them.

Octree Representation: Octree [11] is a spatial partitioning data structure popular for its simplicity and uniformly structured hierarchy. Using an octree to encode the underlying geometric information of the point cloud provides several benefits. Firstly, the octree nodes can be subdivided only in the presence of points within their bounds, leading to a sparse representation. Secondly, the octree, unlike other hierarchical structures such as bounding volume hierarchy or BSP trees, uses a regular subdivision hierarchy which the encoder and decoder can exploit to more efficiently compress the point cloud without having to store extra partition information. Since there is no need to store explicit floating point positions, all the encoding operations are done using integer representation which is more efficient and minimizes possible floating point rounding errors. We store only the bounding box in floating point representation and use it to reconstruct the point cloud to scale. As shown in Equation 1, E_{M} is the upper bound of geometric error that is introduced by using integer addressing at the specific depth of the octree.

The octree nodes are addressed by the Morton code [12] which is a mapping from n-dimensional space ($n = 3$ in our case) onto a 1D space. Each node of the octree contains a single atomic byte, which represents its *child occupancy code* [10] of this node. Each bit in this code denotes the existence (or absence) of each of the 8 possible children of the octree node. For each level of the octree, we have a hashmap where the keys represent the node address and the values represent the child occupancy code. The full octree itself is then represented as a list of D hashmaps.

The leaf nodes' positional information can be completely derived from its parent's child occupancy code. Hence we do not need to explicitly store leaf nodes in memory. Note that this factor by itself can result in a memory saving of roughly 8 times, which is significant in very deep octrees. With this representation our octree can be considered a sparse octree since it does not explicitly track empty nodes.

Octree Construction: Usually the construction of the octree is done from top-down, where any insertion of new node first starts from the root node followed by recursive traversal towards the bottom. While simple to implement, such a method is inherently serial and inefficient as there is a high number of redundant node traversals from the root to leaf for every insertion.

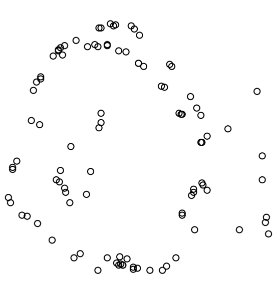
To improving encoding/decoding performance, we instead employ the bottom-up octree construction (see also [13]), i.e., each node insertion starts at the leaf level and recursively creates or updates the immediate parent node until the traversal reaches the root node. With this approach, we can early terminate the upward traversal from the leaf level, once we reach an existing parent node (see Figure 1).

In a parallel implementation, we need only two synchronization points—first when creating a new parent node, and second, when updating the parent node's child occupancy code. In detail, we require a mutex for each octree level when adding new node to that level. Since the child status of the parent node is a single atomic byte (1 bit for each of the 8 children), we can update the child status using atomic `fetch_or` instruction with minimum locking penalty.

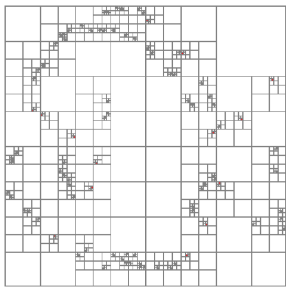
With the synchronization in place, we can insert each point into the octree in parallel. Due to the synchronization, it might be preferable to insert points that are not spatially near to each other to avoid excessive contention on the same parent, since nearby points are more likely share the same parent in higher level. This can be simply achieved by sorting the points randomly before the octree construction.

B. Octree Truncation

Previous methods [9], [10] retain the entire octree in memory during storage and generally perform well on dense and regular point clouds. However, their performance deteriorates when compressing point clouds that are sparse but require deep octrees to minimize the quantization error. Due to sparseness in the point distribution relative to the depth of the octree, the octree structure tends to appear "long legged" where long chains of nodes are created to reduce the quantization error on lone child nodes, which extend all the way to the leaf level (see Figure 2). This class of models is usually created when subsampling a dense point cloud to a lower sampling rate. However, any point cloud represented with a sufficiently deep octree will face this problem, where the leaf cell size becomes much smaller than the minimum distance between two points. When long chains arise in the octree, the memory required for each of the child occupancy codes throughout the chain increases leading to diminishing returns. Hence, we propose the following truncated octree



(a) *Sparse point cloud*



(b) *Deep quadtree*

Figure 2: *A pathological case for existing methods illustrated using 2D points and a quadtree: a sparse point cloud requiring a low quantization error leads to a deep quadtree, where a few lone points cause the creation of long chains, thereby affecting the compression ratio.*

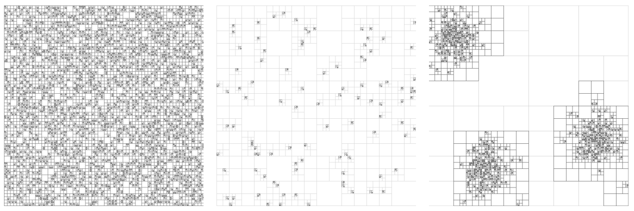


Figure 3: *Example showing truncated quadtrees for point clouds with different distributions. From left to right: dense (untruncated), sparse (truncated before leaf level), clustered (truncated in the middle level) point cloud distributions.*

to alleviate this problem. Rather than storing/encoding the octree nodes with reference to the root node, we instead truncate all the nodes from the root to a determined octree level for a concise representation. This creates a set of sub-octrees with the remaining nodes addressed with reference to each of their corresponding root nodes.

For a dense point cloud, the optimal truncation level will usually be at the root level of the octree, meaning no truncation, because on a dense point cloud, encoding the whole octree in terms of child occupancy codes is already efficient. On the other hand, for extremely sparse point clouds, the optimal truncation level is $D-1$, which is one level above the leaf nodes, i.e., it is more efficient to store the individual points rather than the entire octree. This is because we need a very deep octree for sparse pointclouds which leads to a larger file size if we encode all the nodes. At this truncation level the octree address almost resembles a regular sparse voxel grid. For point clouds that have several isolated clusters of points, the optimal truncation level will likely fall in the middle range, leading to a set of sub-octrees equal to the number of clusters. We show some toy examples illustrating these scenarios in Figure 3.

With the full octree information in memory, it is straightforward to compute the best octree truncation level to achieve the smallest file size by scanning over all the nodes in each octree level and computing their size requirements. Then, we can store the addresses (Morton codes) of the nodes in the truncation level, and the child occupancy codes of all their children. We show the benefits brought about by truncation in Section IV.

Variable Length Addressing Scheme: Careful analysis shows that using a static length for the node address is inefficient for truncated octrees. Based on the truncation level selected, we can optimally determine an efficient length of the address code to use in encoding the sub-root node addresses, similar to the idea of determining the patch size for image completion [14]. The range of addresses represented by the bit length is shown in Table I. Clearly, when we truncate the octree, the depth of each sub-octree is much less than the original octree, thereby requiring less child occupancy codes to be fully represented. Conversely, as the truncation level approaches the leaf level, the number of sub-octrees also increases. The large number of sub-root nodes that need addressing burdens the compression efficiency. Hence we choose this length dynamically based on the truncation level. To have better byte alignment, the variable bit length of the address code is chosen in multiples of 8. By optimally selecting the length of the address, each of the sub-root nodes will only required the minimum bits to correctly address their location on the truncation level. Beyond the obvious benefit of higher compression ratio, optimal address length also helps reduce the encoding/decoding computing requirements.

Table I: *List of octree level and optimal address code length.*

Bit Length	Bit Per Axis	Range Per Axis	Max Octree Level
8	2	0-3	3
16	5	0-31	6
24*	7	0-127	8
32	10	0-1023	11
40	13	0-8191	14
48*	15	0-32767	16
56	18	0-262143	19
64	21	0-2097151	22
128	42	0-4398046511103	43

* Due to the bit flag needed to support offset jump, there is an extra bit used for address code with length in multiple of 3.

Offset Jump Address: It is observed that several nodes in the octree are neighbors. We can hence refer to certain nodes with their original address, and the neighbors with a shorter offset jump instead of the full-length address. Using the previously computed full address and adding the offset jump, we can obtain the full address of the next node.

In our implementation the jump address is set to half the length of the full address in order to attain a good balance between space saving and the range of the offset jump. Instead of the unsigned integer used by the full address, the jump offset is stored as a signed integer to allow for negative jump in any direction. Due to the shorter range of the jump offset compared to the full addressable ranges, there will be cases where the next node address exceeds the range of the offset jump, during which we will revert to using the full address and start over with the jump address once again.

To determine whether the address is a full-length address or a shorter offset jump during encoding/decoding, we use a bit flag encoded into the first bit of the address itself. The improvements by this technique will be shown in Section IV.

C. Encoding

The encoding algorithm (**Algorithm 1**) will encode the truncated octree into a binary blob that can be written directly to a file or further compressed using any file compression algorithm.

Header Information: To support decoding, additional attribute needs to be written to the encoded file's header:

- The bounding box information of M.
- Depth of the octree D.
- Truncated depth of the octree D_t.
- Size of the encoded body in bytes.

Immediately following the header information will be the encoded binary blob.

Body: We have two simple operations in creating the encoded body: writing the sub-root node address addresses of the nodes in the truncation level, and traversing each sub-octree to write the children nodes' child occupancy codes.

Algorithm 1 Encoding Algorithm

```
 1: procedure ENCODE($O, D_t$)
 2:     Write the header information
 3:     $P \leftarrow (0,0,0)$                ▷ Previous sub-root address
 4:     for all sub-root node address $S$ in level $D_t$ do
 5:         if $|PS|$ is within the range of offset jump then:
 6:             Write offset jump address
 7:         else
 8:             Write full sub-root address
 9:         $P \leftarrow$ full address of $S$
10:         for all node $N$ in depth first traversal from $S$ do
11:             Write $N$'s child occupancy code
```

Parallelism of this process can be achieved by parallelizing step 4. Since each sub-octree is completely independent to each other, we can perform the depth first traversal in parallel for each of the sub-octrees. An additional step to aggregate the various results of sub-octree traversal into the final binary blob needs to be performed. At the end of the encoding, the resulting encoded binary blob can be written to file directly or further compressed using file compression algorithm such as Zip compression [15].

D. Decoding

The output of the decoding algorithm is the complete octree representation in memory. Even if the encoded octree was truncated, the decoded octree will still be the full untruncated octree so that it can be used for downstream operations.

Since the header information is written as raw dump, we can easily load and initialize the currently empty octree. The decoding of the encoded body is slightly more involved than the encoding algorithm, since we need to track the status of the child node expansion.

Let P be the previous sub-root node address, initialized to the zero coordinate $(0, 0, 0)$. Let A be the current sub-root node address and N the current sub-root node. Let S be the stack used to track decoding process and O be the output full octree. Let T be the node at the top of S.

Algorithm 2 Decoding Algorithm

```
 1: procedure DECODE($O$)
 2:     Read the header information
 3:     $P \leftarrow (0,0,0)$                ▷ Previous sub-root address
 4:     for all byte $B$ in file do
 5:         isFull $\leftarrow$ first bit of $B$
 6:         if isFull == True then
 7:             $A \leftarrow B$
 8:         else
 9:             $A \leftarrow P + B$
10:         $P \leftarrow A$
11:         $N \leftarrow A$'s node
12:         Recursively add the $N$ to $O$
13:         $C \leftarrow N$'s child
14:         Push $C$ onto $S$
15:         while $S$ is non-empty do
16:             $T \leftarrow S$'s top node
17:             $C \leftarrow$ next child of $T$
18:             Create the node for $C$ in $O$
19:             Remove $C$ from $T$
20:             if $T$ has no child left then
21:                 Pop it from $S$
22:             if $C$ is not leaf then
23:                 Push $C$ onto $S$
```

The decoding algorithm (**Algorithm 2**) reads each byte from the file and decodes the address of the sub-root nodes based on whether it is a full address or an offset jump address. We first reconstruct the upper truncated portion of the octree by recursively upward propagating nodes toward the octree's root. Then we perform depth first traversal of each sub-octree towards the leaf nodes. To enable parallelism for the decoding process, we need extra storage overhead in the form of each sub-octree's encoded size so that the decoder knows when to terminate the process of a sub-octree.

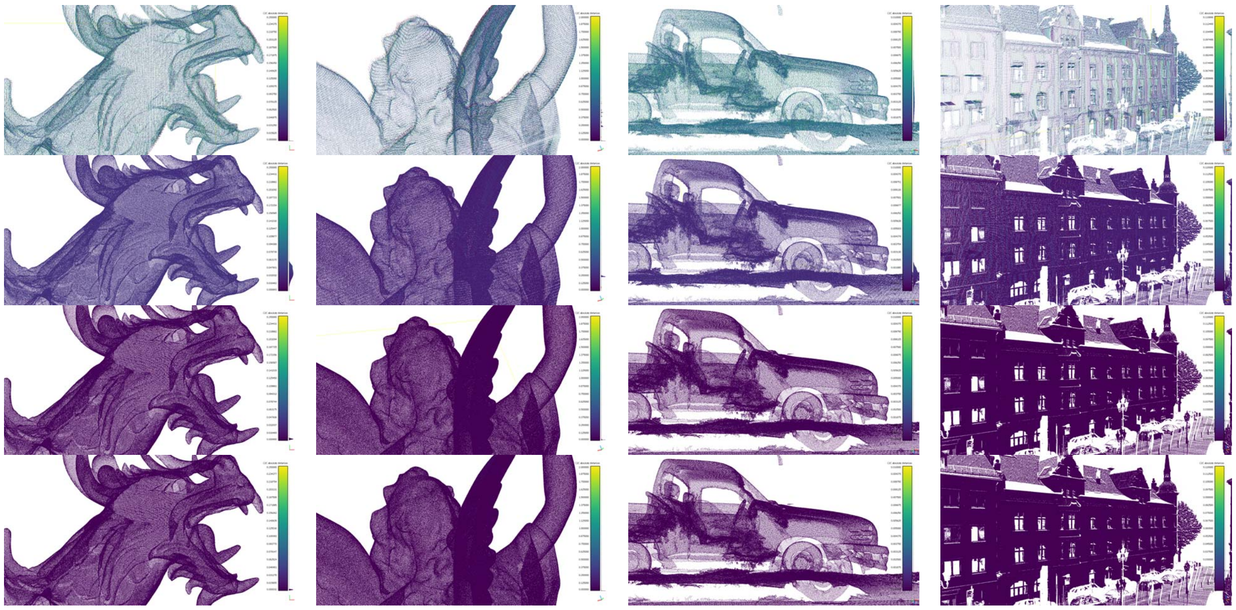


Figure 4: *Visualization of the point cloud distance between the original and decoded point clouds using various user-selected quantization errors. Rows: Octree depth 10, 12, 16, 22 corresponding to quantization errors of 0.0009765%, 0.0002441%, 0.00001525%, 0.0000002384%. Columns: point cloud models* DRAGON*,* LUCY*,* TRUCK*,* FOUNTAIN.

At the end of the decoding, we will have the complete octree in memory. To regenerate the point cloud from the octree, we just need to iterate through each of the leaf nodes, and compute the centroid of the node as the point's position. Alternatively a probabilistic approach can also be used (see [6], [16]) to predict the distribution and position of the points within the octree cell.

IV. EXPERIMENTS

Our method is implemented in C++ and evaluated on a computer with a i7-6700 3.4GHz, 8 core CPU. We support CPU parallelism using the Intel Threading Building Blocks (TBB) library [17]. Some of our test models, namely BUNNY, DRAGON, LUCY, are originally triangular meshes; we remove all attributes except the vertex positions. Other models such as CAR, TRUCK, FOUNTAIN, S28 STATION [18], [19] are real world scanned data. The baseline unstructured point cloud format used in the comparison is the binary version of the Polygon file format (.PLY).

A. Quantization error

We first examine the quality of the decoded point cloud. Given that our method quantizes the point cloud to the size of the smallest octree cell, the maximum error guaranteed will be half of the diagonal length of the smallest cell. In this respect our method's vertex position quantization error is equivalent to that of those proposed by Peng et al. [8]. This implies that the selection of proper octree depth has a large impact on the geometric error of the decoded point cloud. In our experiments, beyond the octree level 12, there is virtually no "visible" difference between the original and compressed point clouds for our test models. We show several results in Figure 4 visualizing the quantization errors for various models and octree depths. The error is fairly evenly distributed throughout the points.

B. Compression results

We next examine the file size compression obtained by our method and others. Table II reports the results.

Table II: *Comparison of file size compression rate. The best compressed size is underlined.*

Model	Points	Size (KB)	Encoded Size (KB)		
			Ours	[3]	[9]
Bunny	2,503	30	17.10	<u>13.0</u>	27
Gargoyle	833,104	9,764	<u>3,020</u>	3,868	3,583
Dragon	3,608,622	42,289	<u>2,196</u>	16,646	2,444
Lucy	11,921,036	139,700	<u>3,122</u>	55,383	3,483
Car	1,763,091	20,662	<u>2,633</u>	8,223	2,795
Truck	2,463,104	28,865	<u>8,721</u>	11,410	10,714
Fountain	28,467,870	333,608	<u>39,342</u>	264,117	43,252
Station	113,898,368	1,334,747	<u>146,382</u>	1,052,707	176,382

The heavily subsampled BUNNY that represents an extremely sparse point cloud, GARGOYLE with average density and point count, and LUCY model with both high point density and point count. The CAR, TRUCK, FOUNTAIN and S28 STATION models are real-world raw laser scan data. We can notice a wide variance in compression rate ranging from 40% to 90% as reported in Table II. This is largely because the encoding algorithm performs much better on a dense point cloud, where both truncated octree encoding and usage of the offset jump are optimal. While sparser model with nodes that are far apart, the sharing of parent nodes is minimal, and usage of the full address code instead of offset jump is also much increased. We wish to highlight the flexibility of the method to handle compression of this wide gamut of point cloud density in comparison to a method like [9] which only works well on regularly sampled point cloud.

The results in Table II demonstrate the improvement when compressing sparse point cloud compared to [9] and [3]. To reflect a sparser point cloud density profile, the experimental models are uniformly subsampled by removing 5% of the original points. Our method improves the compression rate compared to Merry et al. [9] since they must encode the full octree. Improvements are more pronounced as the depth of the octree level increases or as the point cloud density becomes sparser, as demonstrated by the heavily subsampled BUNNY model. Even on denser models, our method has some slight improvement. As mentioned by Lindstrom [3], his method performs poorly for unstructured data that does not exhibit spatial correlation for nearby data points. Notably, our method outperforms both methods on all tests.

C. Performance

We now discuss the computation and parallel processing capability of the method. By using the bottom-up construction of the octree, we are able to gain both the ability to construct the octree in parallel and also upto 90% reduction in the unnecessary nodes traversal as compared to standard top-down octree construction, as reported in Table III.

Table III: *Comparison of the number of traversals performed in generating the octree using different constructions.*

Model	Bottom-Up	Top-Down	Savings (%)
Bunny	33,111	40,048	17.32
Gargoyle	7,656,447	13,811,360	44.56
Dragon	28,173,175	57,753,600	51.22
Lucy	93,597,517	224,445,952	58.30
Car	14,934,133	30,258,960	50.65
Truck	20,497,269	42,769,360	52.07
Fountain	104,576,365	719,850,256	85.47
Station	483,989,488	4,139,535,168	88.31

Due to the simplicity of the encoding and decoding algorithms, our method has a much lower computational complexity compared to method like [5], [8] while still being able to obtain good tradeoff between compression rate and quality. In additional, our method is fully parallelizable and can exploit all available CPU cores which further improve the run-time speed. In particular, the running time required for building octree, encoding and decoding in parallel implementation is about 1/4, 1/3 and 1/2 of that for the counterparts in serial implementation, respectively. Table IV reports the timing statistics of our method.

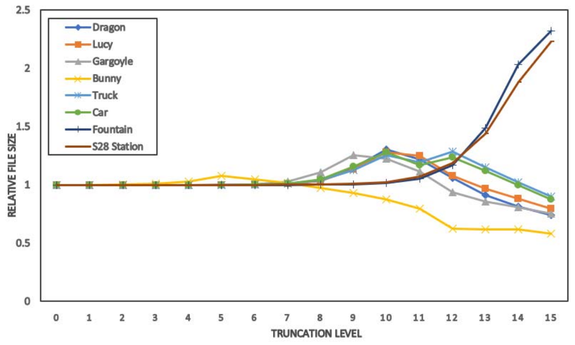


Figure 5: *Effect of truncating the octree at different levels on encoded file size compared to no truncation.*

Table IV: *Timing performance (in seconds) for our method (parallel implementation) and the methods of [3] and [9].*

Model	Our Method (parallel)			[3]	[9]
	Build Octree	Encode	Decode	Encode	
Gargoyle	0.96	0.50	0.90	1.44	5.38
Dragon	4.91	1.06	1.52	6.15	22.84
Lucy	17.99	2.82	3.07	19.95	80.43
Car	2.886	0.89	7.40	3.00	14.22
Truck	3.816	1.24	2.19	4.71	18.99
Fountain	35.25	9.13	10.76	16.67	168.41
Station	145.34	26.25	47.92	62.82	660.12

Table V: *Encoded sizes obtained using different address codes and jump offset lengthes. The octree truncation level of 6 is chosen to fit within the 16 bit full address.*

Model	Bit Length		Encoded Size
	Full Address	Jump Offset	(bytes)
Lucy	64	32	67856
	32	16	37632
	16	8	21802
	16	NA	34882
Gargoyle	64	32	81336
	32	16	47172
	16	8	26411
	16	NA	42784
Dragon	64	32	81336
	32	16	23144
	16	8	13839
	16	NA	21450

Octree truncation significantly helps in reducing the file size as shown in Figure 5, where we explore the relationship between the encoding size against the octree truncation level for dense, sparse and clustered point clouds. Here we explore three point cloud density profiles, the original dense point cloud (FOUNTAIN and S28 STATION), a sparse decimated version (BUNNY, GARGOYLE, DRAGON, LUCY), and clusters of isolated points (CAR and TRUCK). It is clear that sparse and clustered point clouds benefit from truncation when it comes to file compression. While dense point cloud do not have any gain from the truncation, our method is able to fallback to use no truncation and retain the high compression rate from spatial hierarchy based compression. We wish to highlight that due to the adaptive nature of our method, we can achieve the optimal truncation level for any kind of point cloud profile without any manual intervention.

Variable length addressing of the nodes after truncation also reduces the file size significantly. Table V shows approximately 3 times improvement in space-saving can be obtained from using the optimal length address code during encoding. With the use of jump offset address, a further improvement of around 1.5 times reduction in file size was obtained.

V. Conclusions

We have presented a novel parallel compression algorithm for point clouds of varying density profiles. Our compression method is based on a truncated octree and a variable length addressing scheme to obtain higher level of compression rate. Parallelism is also achieved by our method for the main portions of the algorithm during the octree construction, encoding and decoding steps. The experiments have shown that our method can improve compression rate on both sparse and dense point cloud over similar related works.

Acknowledgments

This research was supported by National Research Foundation (NRF) Singapore, under its Virtual Singapore (VSG) Programme (Award No. NRF2015VSG-AA3DCM001-018), and by the Ministry of Education, Singapore, under its MoE Tier-2 Grant (MoE 2017-T2-1-076).

References

[1] H. Yin and C. Berger, “Mastering data complexity for autonomous driving with adaptive point clouds for urban environments,” in *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017, pp. 1364–1371.

[2] P. Ratanaworabhan, J. Ke, and M. Burtscher, “Fast lossless compression of scientific floating-point data,” in *2016 Data Compression Conference (DCC)*, 2006, pp. 133–142.

[3] P. Lindstrom, “Fixed-rate compressed floating-point arrays,” *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, 2014.

[4] X. Chen and S. Zhang, “Three dimensional range geometry and texture data compression with space-filling curves,” *Opt. Express*, vol. 25, no. 21, pp. 26 103–26 117, Oct 2017.

[5] Y. Shao, Q. Zhang, G. Li, Z. Li, and L. Li, “Hybrid point cloud attribute compression using slice-based layered structure and block-based intra prediction,” in *Proc. of the 26th ACM Intl. Conf. on Multimedia*, 2018, pp. 1199–1207.

[6] R. A. Cohen, D. Tian, and A. Vetro, “Point cloud attribute compression using 3-d intra prediction and shape-adaptive transforms,” in *2016 Data Compression Conference (DCC)*, March 2016, pp. 141–150.

[7] P. Rosenthal and L. Linsen, “Image-space point cloud rendering,” *Proceedings of CGI 2008*, 2008.

[8] J. Peng and C.-C. J. Kuo, “Geometry-guided progressive lossless 3d mesh coding with octree decomposition,” in *ACM SIGGRAPH 2005*. ACM, 2005, pp. 609–616.

[9] B. Merry, P. Marais, and J. Gain, “Compression of dense and regular point clouds,” in *AFRIGRAPH ’06*, 2006, pp. 15–20.

[10] Y. Huang, J. Peng, C. . J. Kuo, and M. Gopi, “A generic scheme for progressive point cloud coding,” *IEEE Trans. Vis. Comput. Graph.*, vol. 14, no. 2, pp. 440–453, 2008.

[11] D. Meagher, “Geometric modeling using octree encoding,” *CGIP*, vol. 19, no. 2, pp. 129–147, 1982.

[12] L. Stocco and G. Schrack, “Integer dilation and contraction for quadtrees and octrees,” in *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing. Proceedings*, May 1995, pp. 426–428.

[13] H. Sundar, R. S. Sampath, and G. Biros, “Bottom-up construction and 2:1 balance refinement of linear octrees in parallel,” *SIAM J. Sci. Comput.*, vol. 30, no. 5, 2008.

[14] H. Zhou and J. Zheng, “Adaptive patch size determination for patch-based image completion,” in *Proceedings of the International Conference on Image Processing, ICIP 2010*. IEEE, 2010, pp. 421–424.

[15] P. Deutsch and J.-L. Gailly, “Zlib compressed data format specification version 3.3,” Tech. Rep., 1996.

[16] J. Smith, G. Petrova, and S. Schaefer, “Progressive encoding and compression of surfaces generated from point cloud data,” *Computers & Graphics*, vol. 36, no. 5, pp. 341–348, 2012.

[17] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O’Reilly Media, Inc., 2007.

[18] S. Choi, Q.-Y. Zhou, S. Miller, and V. Koltun, “A large dataset of object scans,” *arXiv preprint arXiv:1602.02481*, 2016.

[19] T. Hackel, N. Savinov, L. Ladicky, J. D. Wegner, K. Schindler, and M. Pollefeys, “Semantic3D.net: A new large-scale point cloud classification benchmark,” *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 41W1, pp. 91–98, 2017.