STAG: Smart Tools and Applications in Graphics (2020)
S. Biasotti, R. Pintus and S. Berretti (Editors)

A system for fast and scalable point cloud indexing using task parallelism

Pascal Bormann[1,2] and Michel Krämer[1,2]

[1]Fraunhofer Institute for Computer Graphics Research IGD, Fraunhoferstr. 5, 64283 Darmstadt, Germany
[2]Technical University of Darmstadt, 64289 Darmstadt, Germany

Abstract

We introduce a system for fast, scalable indexing of arbitrarily sized point clouds based on a task-parallel computation model. Points are sorted using Morton indices in order to efficiently distribute sets of related points onto multiple concurrent indexing tasks. To achieve a high degree of parallelism, a hybrid top-down, bottom-up processing strategy is used. Our system achieves a 2.3x to 9x speedup over existing point cloud indexing systems while retaining comparable visual quality of the resulting acceleration structures. It is also fully compatible with widely used data formats in the context of web-based point cloud visualization. We demonstrate the effectiveness of our system in two experiments, evaluating scalability and general performance while processing datasets of up to 52.5 billion points.

CCS Concepts

• ***Computing methodologies*** → ***Parallel algorithms;*** • ***Information systems*** → ***Extraction, transformation and loading;***

1. Introduction

Point clouds are an important tool for the representation of objects from the real world. With the advent of cheap, high-quality laser scanners and the rising popularity of drones for data capturing, point cloud data has found its way into an increasing number of applications, both for data analysis and visual exploration. The nature of point clouds as a volumetric geometry representation often results in significantly larger datasets compared to polygon-based geometry representations. This increased data size poses several challenges for systems that visualize point clouds: Since only a fraction of the full dataset can usually be held in memory, relevant data has to be fetched from disk or a remote server dynamically using out-of-core techniques. In this context, spatial acceleration structures play a crucial role in enabling interactive rendering of large point clouds. The best spatial acceleration structures for point clouds provide fast access to specific sections of the data when viewing it up close, while at the same time enabling access to subsampled versions when viewing it from far away. This process is known as *Level of detail (LOD)* and has been studied in the context of point cloud visualization by Scheiblauer [Sch14] and later by Schütz [Sch16] who adopted it into the widely used web-visualization tool *Potree* [pota].

The creation of a good spatial acceleration structure that supports LOD, a process called *indexing*, is computationally expensive due to two main factors: First, point cloud datasets are often very large: With billions of points in a single dataset, it is common to deal with hundreds of Gigabytes or even Terabytes of data. While the size of such datasets is the reason for creating out-of-core capable acceleration structures, it also means that any process that creates such an acceleration structure has to be out-of-core capable as well. Since out-of-core algorithms use slow external memory to store intermediate results, their runtimes are usually slower than in-core algorithms. Second, to enable LOD, lower resolution versions of the point cloud have to be computed from the full resolution point cloud. This is often realized through sampling procedures that select points based on a minimum distance to each other, resulting in uniform subsamples with good visual quality. This subsampling process is challenging because raw point cloud data is most often unsorted, containing no spatial relationships between individual points. Identifying points within a minimum distance to each other therefore is equal to a search in unsorted data. Using spatial acceleration structures thus shifts the computational burden away from the rendering process and towards the preprocessing system. While this shift is a beneficial one, seeing that the creation of an acceleration structure has to be done just once for a static point cloud, it still has drawbacks: The necessary preprocessing step is time- and resource-consuming and introduces a delay before the data is ready to be used.

In this work, we describe a system for creating visualization-optimized acceleration structures for point cloud data that addresses these challenges of data volume and delay before the data can be used. For the indexing process, the system utilizes a *novel data-parallel algorithm*, which is based on a *hybrid approach combining top-down and bottom-up processing*. Compared to existing systems, this approach achieves a *higher performance and scalabil-*

DOI: 10.2312/stag.20201250

ity because it allows individual tiles to be processed independently by multiple threads in parallel.

Existing systems tend to be either fast but tailored to a specific rendering system, or slow but usable with popular and standardized rendering systems. We therefore set out to create a system that covers both use cases and satisfies the following goals:

1. **Performance**: The system must be able to achieve significantly better performance than the state of the art, while maintaining comparable visual quality.
2. **Compatibility**: The system has to support common data formats. In particular, it has to support both the widely-used *Potree* system as well as the standardized file format *3D Tiles* [ces]. A noteworthy non-goal is the optimization of system performance through usage of bespoke file formats. Ideally, our system should be a drop-in replacement for other point cloud indexing tools.
3. **Hardware-agnostic**: The system should make best use of the available hardware. We want to achieve significant performance benefits both on modern systems with dozens of threads and fast solid-state-drives (SSDs) as well as on systems with slower hard-disk-drives (HDDs).

The remainder of this paper is structured as follows: In section 2 we give an overview of related work in the area of spatial acceleration structures for point clouds. Section 3 describes the design of our system, covering the three main observations that enable our systems performance. We also cover several important implementation details in section 4. In section 5 we evaluate the performance and quality of our system compared to other widely used point cloud indexing systems and then discuss these results in section 6. Section 7 concludes this paper and gives an outlook for future work.

2. Related work

The two main acceleration structures used for out-of-core point cloud visualization are kd-trees and octrees. These structures are common and well-understood in literature but have their own unique benefits and challenges when applied to point cloud data. While this paper focuses on octrees, it is worth understanding the applicability of both acceleration structures, which motivates our decision to generate octrees over kd-trees.

kd-trees are generally able to adapt better to irregular data, whereas octrees are simpler to create and handle due to their regular nature. One challenge while building kd-tree accelerators is the selection of the split plane position, which either requires a full scan of all points for each split plane, or spatial sorting of the points to quickly find the median position along an axis. These approaches are not applicable to large point clouds due to their sheer size. Multiple linear searches through billions of points are slow, and sorting is not trivially possible because the whole data very often does not fit completely into memory. To solve this problem, histogram-based approaches have been used to find approximate positions for the split planes [RDD15, BGM*12]. Goswami et al. used a different approach by introducing a kd-tree variant that has multiple split planes along each axis, which allows the creation of accelerators with uniform sized nodes [GZPG10]. The more uniform nature of kd-trees has benefits for out-of-core rendering on a single machine, as it enables fast selection of visible subsets from the point cloud, which can be swapped in and out of GPU memory, enabling good visual quality even for highly interactive applications like VR [DMS*18].

The situation is different when the point cloud data does not reside on the local machine but instead on a remote server. Here, octrees are used more prominently, which is in part due to the work by Scheiblauer [Sch14] where he introduced the *modifiable nested octree* data structure. It is based on partitioning the point cloud into an octree with each node containing a subsample of the original point cloud, where nodes closer to the root contain sparser subsamples and deeper nodes contain more dense subsamples. This works well with web-based point cloud visualization, as LOD is already part of the data structure and as such, selecting a representative visual subset for a given view can be realized by simply requesting octree nodes of different sizes and streaming the full content of these nodes to the client. Based on this work, the widely used *Potree* [pota] system was developed by Schütz [Sch16], which also contains the *potree-converter* tool [potb] for generating the required octree acceleration structure from raw point cloud data. Schütz also illustrated different possible strategies for subsampling the point cloud with different visual characteristics [Sch16, section 3.3.1]. *potree-converter* uses a form of Poisson-disk sampling [Bri07], which yields good visual quality but is computationally expensive. Another system for generating octree acceleration structures from point cloud data is *Entwine* [ent]. It uses a similar approach to *potree-converter* but selects point samples based on their distance to cells in a grid, favoring points closer to the center, which is more computationally efficient compared to Poisson-disk sampling. It is worth noting that there is significant development of the *potree-converter* tool currently ongoing, with the aim of increasing performance and optimizing the resulting file structure [SOW20].

An efficient way to build octrees in general is through the usage of the Z-order curve, a space-filling curve that emerges as the result of sorting points based on their Morton index [Mor66]. Using this technique, Karras derived fully parallel algorithms for creating octrees, kd-trees and bounding volume hierarchies for arbitrary primitives [Kar12]. While powerful, these algorithms do not take LOD into consideration, making them less useful in the context of point cloud visualization. Here, Elseberg et al. [EBN13] provide a simple algorithm based on sorting points by Morton index to create an octree in log-linear time. While they do not consider parallelizing the computation, the concept of their algorithm - recursively subdividing a range of sorted points - is one of the building blocks of the underlying indexing algorithm that our system uses.

In the context of parallelized point cloud processing, the work by Alis et al. is particularly noteworthy, as they identified the potential for sorting point clouds using Morton indices to distribute processing over multiple worker nodes in a network [ABL16]. Partitioning points based on spatial locality allowed Alis et al. to speed up a k-nearest-neighbour algorithm. We build upon this approach and extend it to the parallelized creation of spatial acceleration structures for point clouds.

3. Design of our system

At the highest level, our system processes the whole point cloud in fixed-sized chunks called *batches*, so that each batch can be held

and processed entirely in memory. A single batch is made up of a fixed number of points read sequentially from the source files. The results of processing a batch are immediately stored on disk and can be temporarily loaded from disk during the processing of subsequent batches, thus enabling out-of-core processing of arbitrarily large point clouds. Within a batch, each point is assigned to one specific octree node. From all points that fall into the bounding box of the node, the sampling method selects a representative subset of points for this node. The sampling methods in our system are the ones present in both *Entwine* and *potree-converter*, namely *grid-center sampling* and *Poisson-disk sampling*. Since the sampling process is computationally expensive, the main design principle of our system is to aim for a high degree of data parallelism, so that each logical core on the target machine can always be saturated with sampling points. We are able to achieve this by making three observations:

1. The indexing process can be modeled as a recursive task graph
2. Processing does not have to be exclusively top-down or bottom-up
3. Sorting points by Morton indices enables fast identification of independent points

The next three sections explain these observations and their importance in detail.

3.1. Modeling the indexing process as a recursive task graph

Task-parallel programming is a common method for achieving processing speedup by using concurrency, and there are powerful frameworks that handle the scheduling of task graphs onto parallel systems [HLLL20]. By modeling the indexing process as a task graph, we can distribute the work over a larger number of threads. The actual amount of achievable parallelism depends on the structure of the task graph, in particular the maximum number of independent tasks at any point during processing. Our system aims for high degrees of data parallelism, the way we process the points plays an important role in enabling this goal.

The two main sets of data are the points and the nodes of the acceleration structure. This naturally results in two different ways of indexing the point cloud: Either iterate over all points, checking each point against all nodes until a matching node is found (figure 1 (a)), or iterate over all nodes, selecting all matching points for the current node from the set of all points (figure 1 (b)). Since each node can contain multiple points, but each point only belongs to a single node, parallelizing over the points would require extensive synchronization, as two points in different tasks might fall into the same octree node. Instead, we chose to create one task per octree node and check all possible points for their affiliation to this node. Care has to be taken in the case of point cloud indexing because the resulting data structure will contain points in both leaf nodes and interior nodes. The sampling for LOD introduces dependencies between a node and each of its direct or indirect children, and the result of sampling methods like the *Poisson-disk sampling* depends on the order in which points are processed. As a consequence, sibling nodes in the octree can be processed independently, but nodes in a parent-child relationship have to be processed with the parent node first and then the child nodes. In addition to that, the full structure of the task graph for a single batch cannot be known in advance because each node has to first be processed, sampling all points for this node, before it is clear how many - if any - points remain to be sorted into the node's child nodes.

Putting this structure into a task graph form yields a recursive task graph illustrated in figure 2. Processing starts at the root node of the tree, followed by up to eight independent tasks for each of the root node's children, which in turn are again followed by up to eight independent tasks each, and so on until each point is assigned to a node. In this way, the deeper the processing progresses into the octree, the more independent tasks are present in the task graph and thus the higher the degree of achievable parallelism is.

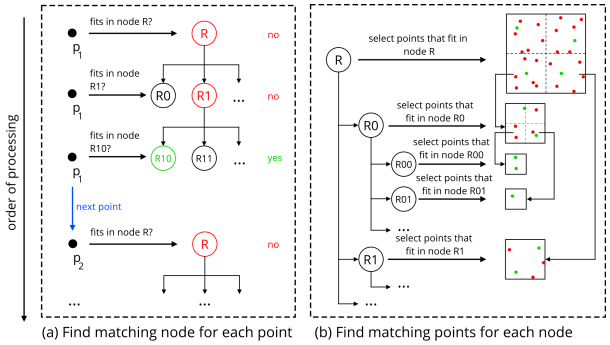

Figure 1: *Indexing a point cloud can be achieved by either finding a matching node for each individual point (a) or finding all matching points for each individual node (b)*

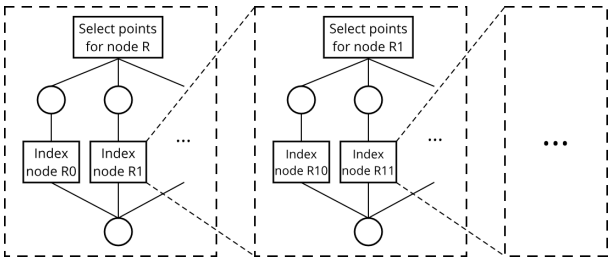

Figure 2: *Processing the point cloud starting from the root node results in a recursive task graph*

3.2. A hybrid top-down, bottom-up processing scheme

The task graph introduced in the previous section models a full top-down processing scheme. Top-down processing always starts at the root node and progresses down into the tree. It is used in both the *Entwine* and *potree-converter* systems, where each point is first checked against the root node, then passed on to the appropriate child node if it does not fit into the root node, and so on. In contrast, bottom-up processing starts at the leaf nodes of the tree and moves up towards the root node, reconstructing the upper nodes from the lower nodes. This is the approach that Goswami et al. use [GZPG10] to add LOD to their multi-way kd-tree. Both approaches have their respective advantages and disadvantages: Top-down processing touches each node only once, whereas bottom-up processing may access a single node multiple times, resulting in

more I/O load. In contrast, every point in top-down processing has to be checked against the root node, with a very high probability that the point will not fit, resulting in a high number of checks that cannot easily be parallelized. Bottom-up processing immediately starts at the leaf nodes of the tree, allowing for trivial parallelization over all leaf nodes.

Our system uses a hybrid approach where we start at some level l_{par} in the octree where the number of non-empty nodes - nodes whose bounding boxes contain some of the points of the current batch - is at least as big as the desired level of parallelism. This way we guarantee that we immediately saturate all threads with indexing tasks. Since every node will - on average - have more than one non-empty child node, the number of available tasks for scheduling will remain constantly higher than the number of available threads. As the workload of each of these tasks may vary drastically, having a large number of tasks to schedule helps counteracting these variations. This hybrid processing scheme skips all octree levels above l_{par}, we reconstruct these nodes from the nodes at level l_{par} in a final postprocessing step after all batches have been processed.

3.3. Quickly identifying independent points using Morton indices

The remaining problem is how we can quickly identify how many non-empty nodes there are at a given level in the octree, and how we can gather all points that are contained within the bounding box of a specific node. This can be achieved efficiently by sorting all points in a batch by their three-dimensional Morton index. Once sorted in this way, all points that belong to any specific node in the octree are stored sequentially in memory. Since at each level, no point belongs to more than one node, this yields a series of disjunct memory regions that can be processed independently. The split positions that indicate where the range for one node ends and the range for the next node begins can be identified in logarithmic time using a binary search. Sorting primitives by Morton index in this way enabled Lauterbach et al. to create a massively parallel algorithm for fast bounding-volume-hierarchy (BVH) construction on GPUs [LGS*09], which was later extended to octrees by Karras [Kar12], proving the effectiveness of this approach.

Figure 3 illustrates the indexing process visually, using full top-down processing for brevity. The resulting task graph is illustrated in figure 4 and contains the following steps:

1. Morton-Index calculation and sorting, trivially parallelized using the fork-join pattern
2. Merging the sorted ranges of step 1 into one range for each node at level l_{par}
3. Processing each non-empty node at level l_{par}

   3.1 Processing a single node, which is comprised of the following steps:

   (a) Loading points from disk that were selected for this node in previous batches
   (b) Merging new points in this batch with previous points
   (c) Applying the sampling function to each point to select all points that belong to the current node
   (d) Storing selected points on disk

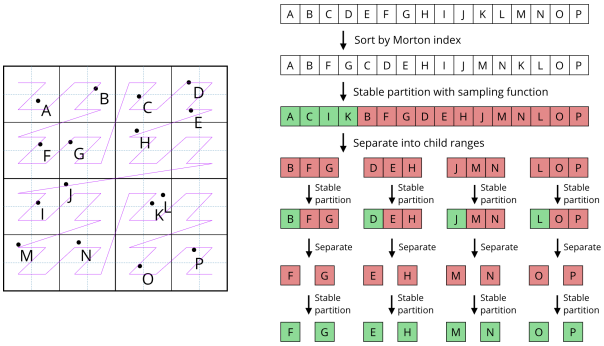

Figure 3: *Overview of the process of sampling points for each node (illustrated in 2D and with full top-down processing for brevity). On the left, example points and the resulting Z-order curve are shown. On the right, a step-by-step overview illustrates how points are sampled at each node, with selected points in green and remaining points in red.*

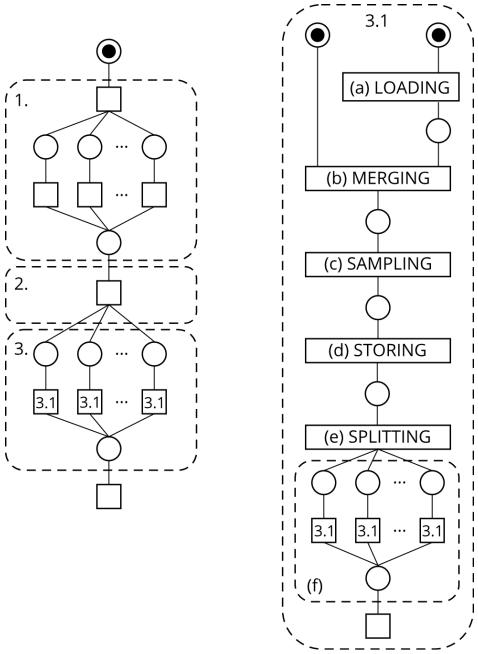

Figure 4: *The task graph for the indexing process in our system. The recursive nature of processing can be seen with task 3.1, which processes a single node and calls itself recursively. Labeled tasks are explained at the end of section 3.3.*

   (e) Splitting remaining points into up to eight disjunct ranges containing all points that fall into each of the child nodes' bounding boxes
   (f) Processing each of the child nodes as a new task, starting again from step 3.1

Step (d) writes the points for each node into a separate file, which matches the expected file structure that *Potree* uses, while at the same time being fully compatible with the 3D Tiles file format. This fulfills our goal of file format compatibility.

4. Implementation

In this section, we go over some implementation details that were important in achieving the performance results that our system exhibits. Our implementation is loosely based on the source code of the *potree-converter* tool in version 1.7. The source code for our implementation is available at https://github.com/igd-geo/schwarzwald.

4.1. Computing Morton indices for points in $\mathbb{R}^3$

We briefly state a formal definition for a Morton index in $\mathbb{R}^3$ before illustrating how it is calculated in our implementation. Morton indices were first proposed by Guy M. Morton [Mor66], and while they are widely used, we feel it is important to clearly state their definition in relation to point cloud data in order to understand some of their inherent challenges in this context.

Given a bounding box $B \subset \mathbb{R}^3$, subdivide it into a regular grid of k^3 cells. Let $c_i = (x_i, y_i, z_i)^T$ be the index of cell i, with $x, y, z \in [0; k-1]$. The Morton index m_i for cell c_i is an integer number obtained by interleaving the bits of x_i, y_i and z_i. In $\mathbb{R}^3$, there are six possible orders for interleaving the bits from most significant to least significant bit (XYZ, XZY, YXZ, YZX, ZXY, and ZYX), in our implementation we use the order XYZ. Figure 5 illustrates the bit-interleaving process.

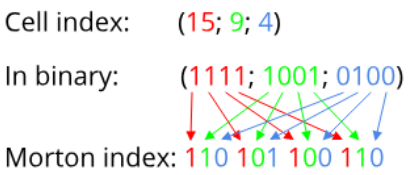

Figure 5: *A 3D Morton index is calculated from X (red), Y (green) and Z (blue) coordinates through interleaving of the bit representations of the coordinates.*

In order to compute a Morton index for a point $p \in \mathbb{R}^3$ in a point cloud, we have to subdivide the bounding box of the point cloud into a regular grid and then find the grid cell that contains p. Since our algorithm generates octrees, we first calculate the cubic bounding box of the point cloud and use it as a reference for Morton index calculation. We then have to choose an appropriate value for the subdivision factor k, which is a tradeoff between precision and the number of bits required to store the Morton index. In our implementation, we chose $k = 21$, requiring $3k = 63$ bits per Morton index, which fits into a single 64-bit integer value. The subdivision factor k determines the maximum depth of the octree in our implementation - 21 levels in this case - and thus also determines the minimum side length of any node in the octree, which equals $2^{-k} = \frac{1}{2097152}$ times the side length of the cubic bounding box. For a dataset with a bounding box side length of 20km, the smallest node can have a side length of about 1cm. While this suffices for the datasets that we encountered during our tests (the most dense dataset having an average distance of 5cm between points at 4.5km side length), this is a limitation in terms of precision that is not shared by the reference tools *Entwine* and *potree-converter*. Extending our system to use 128-bit Morton indices would increase the ratio of bounding box side length to minimum node side length to about $4.4 * 10^{12}$, yielding millimeter precision for an earth-sized point cloud. This would come at the expense of increased memory usage and decreased performance since 128-bit arithmetic is currently not natively supported on general-purpose processors and has to be emulated using multiple 64-bit operations.

To find the cell c_p for a point p, we first translate p into the local reference frame of the cubic bounding box B, whose origin is the minimum vertex of B. This yields a point $p_B \in [0; l_B]^3$, where l_B equals the side length of B. Multiplying p_B by $\frac{2^k}{l_B}$ yields a point $p_{norm} \in [0; 2^k]^3$. The coordinates for the indices of c_p are then computed as $x_p = min(\lfloor p_{norm_x} \rfloor, 2^k - 1)$ and correspondingly for y_p and z_p. A C++ implementation for the bit interleaving procedure can be found in appendix A.

4.2. Internal point representation

During each batch, our system reads points from the input files into an internal point buffer. A point is defined as a tuple $p = (a_1, a_2, ..., a_k)$ of attributes a_i, such as the position in $\mathbb{R}^3$, an integer-valued intensity or a classification number. The interested reader is referred to the LAS file specification [las] which defines a series of common point attributes. Most tools that we found store collections of points in interleaved format, that is for a collection of points $(p_1, p_2, ..., p_n)$, all attributes for each point are stored continuously in memory:

$$(a_1(p_1), a_2(p_1), ..., a_k(p_1), a_1(p_2), a_2(p_2), ..., a_k(p_2), \\ ..., a_1(p_n), a_2(p_n), ..., a_k(p_n))$$

An alternative representation is to store the data sequentially per attribute:

$$(a_1(p_1), a_1(p_2), ..., a_1(p_n), a_2(p_1), a_2(p_2), ..., a_2(p_n), \\ ..., a_k(p_1), a_k(p_2), ..., a_k(p_n))$$

The first representation is called *Array of Structures (AoS)*, whereas the second representation is called *Structure of Arrays (SoA)*, referring to the way that these representations would be implemented in a C program. While AoS memory layouts are often easier to work with and more extensible, SoA layouts exhibit better cache locality because similar attributes are stored together in memory. For our system, we chose the SoA memory layout for the internal point buffer. The reasoning behind this decision is that during the indexing process, many of the computationally expensive steps (Morton index calculation, sampling) only require the positions of points and no other attributes. Storing all positions in a contiguous memory region exhibits good cache locality, thus increasing performance.

4.3. Parallel file reading

Using the task graph introduced in section 3, we are able to saturate a large number of threads with indexing tasks, reaching indexing speeds of up to 20 million points per second. To sustain this high point throughput, our system has to be able to read points from the

source files into the internal point buffer with a similar throughput. Depending on the file type and the target system, points can be read with throughputs ranging from less than one million points per second to almost 20 million points per second on a single thread. In particular, we found that reading points from compressed LAS files (LAZ) can be up to an order of magnitude slower than reading from uncompressed LAS files due to the computational overhead of the LAZ decompression algorithm. We use a second task graph that consist of several read tasks that concurrently fill the internal point buffer with points read from disk. This task graph is executed once per batch in parallel to the indexing task graph. To ensure that these two task graphs do not stall each other we limit the fork factors of both task graphs so that their sum does not exceed the total number of threads. Choosing the actual fork factors for reading and indexing is either done through a fixed allocation of available threads at program startup, or using an adaptive strategy that calculates reading and indexing throughput factors for each batch and adjusts the fork factors accordingly. Similar to *Entwine*, parallel file reading is implemented with file-level granularity. The maximum fork factor for the reading task graph is thus limited by the number of files. This can have detrimental effects on the runtime performance, which is discussed in section 5.

4.4. Supporting lossy file formats

During indexing, we write the selected points at each node immediately to disk in the desired output file format (*Potree*-format or 3D Tiles). These files serve as a way for caching data during processing, while at the same time being the final output data once processing is finished. Using the same file format for caching and output data is efficient as no postprocessing step is required to convert intermediate results into the final data format. Depending on the output file format, some additional work is required to make our system work correctly. If during a batch a node is being processed for which data exists on disk, our system loads these points from disk and calculates the Morton indices for these points again. When using lossless file formats, such as the PNTS file format of 3D Tiles, the Morton indices are recomputed exactly. Since we use a stable partitioning function during sampling, all selected points for each node are still sorted by Morton index when being stored to disk, and thus remain sorted after retrieval. For lossy file formats like LAS, which quantizes floating point values as 32-bit integers, there might be rounding errors, resulting in slight differences between the recalculated and original Morton indices. This can break the sorting of the points. The only way to fix this is by sorting the points again after reading from disk. As a consequence, the indexing process is slightly slower when using the LAS file format as compared to the 3D Tiles format.

5. Evaluation

In this section, we compare the results of our system to those of the *potree-converter* and *Entwine* tools. For this, we conducted two experiments:

1. *Tiling performance*: Here, we analyze the runtime performance, quality of the resulting octree, as well as visual quality based on four publicly available datasets ranging from 854 million points to 52.5 billion points
2. *Scalability*: Here, we analyze the scalability of our system in relation to the number of used threads and compare it to the scalability of *Entwine*

The following datasets were used for testing:

- *Wellington*: The Wellington, New Zealand 2013 open dataset [wel] consisting of approximately 52.5 billion points stored in 9405 LAZ files with a total size of 248 GiB
- *DCPP*: The PG&E Diablo Canyon Power Plant open dataset [ca1] consisting of approximately 17.7 billion points stored in 2337 LAZ files with a total size of 85 GiB
- *Utah*: The High Resolution Topography of House Range Fault, Utah open dataset [uta] consisting of approximately 2 billion points stored in 16 LAZ files with a total size of 15 GiB
- *DoC*: The District of Columbia open dataset [dis] consisting of 854 million points stored in 320 uncompressed LAS files with a total size of 24 GiB

For the *Tiling performance* experiment, we used a virtual machine in an OpenStack cloud with 8 virtual CPUs, 16 GB RAM, and a 3 TB volume residing on an HDD. For the *Scalability* experiment, we used a virtual machine in Amazon AWS, using the m5a.16xlarge flavor, which has 64 virtual CPUs, 256 GB RAM and block storage of type General Purpose SSD.

In order to guarantee a similar workload for all tested tools, we ran our tool and *potree-converter* with `-d 111` and *Entwine* with `--span 64` as the spacing parameters. The parameter `-d 111` for our tool and *potree-converter* determines the maximum number of points on the diagonal of the root bounding box and is a close approximation to `--span 64`, the latter dictating the maximum number of points on a single axis. Additionally, we ran our tool twice, once with *Poisson-disk sampling* to compare to *potree-converter*, and once with *grid-center sampling* to compare to *Entwine*. All tools generated their data as compressed LAS files in the *Potree* structure.

We used *Entwine* version 2.1 and *potree-converter* version 1.7 for all tests and ran all tools using Docker on Ubuntu 18.04. As mentioned in section 2, *potree-converter* version 2.0 has been released at the time of writing this paper. Due to its novelty, at the time of writing there was no support for running inside a Docker container on a Linux environment, which prevented us from obtaining comparable performance data. Preliminary results obtained on a Windows-based desktop machine show promise, with *potree-converter* version 2.0 being around 30% faster than our tool, and indicate that the performance of *potree-converter* version 2.0 should be thoroughly analyzed and compared to our system once it is possible to do so in a common environment. Disregarding the technical hurdles, *potree-converter* version 2.0 uses a different, non-backwards-compatible file format and focuses heavily on SSD-based systems, which differs from the goals that we set out for our system in section 1. The implications of this are discussed in section 6.

5.1. Tiling performance

Table 1 shows the runtimes of the tested tools with the four datasets. Our tool consistently outperforms both *potree-converter* and *Entwine*, achieving a 4.9x to 9x speedup over *potree-converter* and a

2.3x to 3x speedup over *Entwine*. The slow performance of *potree-converter* compared to both *Entwine* and our system is mostly due to the fact that *potree-converter* always uses only two threads. On the largest dataset with 52.5 billion points, *Entwine* aborted early without an error message, likely due to insufficient memory. Compared to *Entwine* and *potree-converter*, our system does not keep any state in memory between multiple batches, so as long as each batch fits into memory, our system will be able to compute point clouds of arbitrary size. In particular, it is thus very unlikely that our system will run out of memory after prolonged processing.

In addition to the runtimes, we also analyzed the size and node counts of the resulting octrees, which can be seen in table 2 and table 3. Here, our system creates significantly more nodes than both *potree-converter* and *Entwine*, with *Entwine* producing octrees with the least nodes. This is because Entwine combines multiple related nodes if they all contain only a few points. Our system is very strict and fully samples all nodes but the leaf nodes, resulting in more internal nodes with very few points. The consequence of this is also increased memory usage for compressed files, as compression is less effective for files with only a few points. In terms of data sizes, it is also worth noting that *potree-converter* version 1.7 does not support the full range of attributes defined by the LAS standard, thus discarding some data.

Lastly, we performed a visual comparison of the resulting indexed point clouds, which can be seen in table 4. The results of our tool are very similar to those of *Entwine*, with the *Entwine* data being slightly more dense. Both our tool and *potree-converter* exhibit artifacts on the edges between nodes, which are a result of the *Poisson-disk sampling* implementation which does not guarantee a minimum point distance between adjacent nodes. The artifacts are more prominent for our tool, likely due to the sorting of the points which results in more regular sampling patterns.

5.2. Scalability

Figure 6 shows the results of the scalability experiment, illustrating how runtime performance scales in relation to the number of threads used. We compared the runtime performance of *Entwine* with that of our system for up to 64 threads. Since *potree-converter* uses a fixed number of two threads, we excluded it from this experiment. We chose the two datasets *DoC* and *Utah* because they are sufficiently different to illustrate the effects of file format and number of files on the runtime performance. For processing performance and scalability, the *DoC* dataset is more favorable, as it contains a large number of small, uncompressed files. Uncompressed files are significantly faster to read, and the large number of files enables reading many files in parallel. In contrast, the *Utah* dataset consists of a small number (16) of compressed files, with extensive variation in the file size (from 57 MiB to 2.8 GiB).

Both *Entwine* and our system benefit from using more threads, but to different degrees. For the *DoC* dataset, doubling the number of threads reduces the runtime of our system by a factor of about 2 up until 16 threads are used. The largest gain can be seen between two and four threads, where the runtime drops by a factor of 2.73. This is due to the fact that with only two threads, one thread is loading points from disk while one thread is indexing and it is very likely that one of these threads will perform its work faster than the other, resulting in one thread being idle for a significant portion of the runtime. The larger the number of threads, the less significant this effect will be. With the highest number of threads (64), synchronization effects start to become apparent, as the runtime with 32 threads is about equal to that with 64 threads. For the *Utah* dataset, our system scales less well but still better than *Entwine*, which is mostly due to the fact that the structure of the dataset puts a natural limit on the maximum number of parallel reading threads. For the *DoC* dataset with up to 16 threads, *Entwine* also scales with a factor of about 2, the exception being the jump from two to four threads, where the runtime is equal. The reason for this is that *Entwine* never uses less than four threads, even when called with `--threads 2`. For the *Utah* dataset, *Entwine* also scales worse than for the *DoC* dataset, which we again attribute to the less favorable file structure of this dataset.

6. Discussion

Based on the performance results, our approach for increasing data parallelism was warranted. While our system does benefit from running with an SSD - as it is very I/O heavy - it outperforms the reference systems *Entwine* and *potree-converter* significantly while running with an HDD. Compared to *Entwine*, our system is also able to write output files directly into 3D Tiles format without any intermediate steps. *Entwine* does support 3D Tiles, but only as the result of a separate processing tool that has to be run after indexing. We do not see any technical limitations why a point cloud indexing tool should not be able to directly output indexed data into multiple common formats. Concerning input file formats, our system currently only supports LAS/LAZ files but is easily extensible to other file formats. The only important limitation in terms of performance is the availability of bounding box information in the file format, for example in the file header like in the LAS file format. Any file format that does not provide this information will require an initial scan over the whole point cloud to compute the full bounding box, which may introduce a significant performance penalty.

While the resulting acceleration structures that our system produces are visually very similar to those of *Entwine* and *potree-converter*, it falls short to these systems in terms of node count and file size. The very high file count that point cloud indexing systems produce is a general problem that we will have to address in future work. Indeed, version 2.0 of *potree-converter* has the reduction of file counts as one of its main goals [SOW20]. At the same time, standardized file formats such as 3D Tiles are inherently hierarchical and often realized through large file hierarchies. The multi-file approach in particular makes requesting data from web applications much easier as less logic is required in both the client and server. We see significant potential for further research in this area.

In terms of scalability, our system meets expectations for up to 32 threads. At higher thread counts, the synchronization points in the fork-join parallelization model that we use prevent further scalability. This is a critical hurdle that has to be overcome before our system will be usable in a large distributed environment. Running in such an environment seems highly desirable, as it gets exceedingly challenging to handle very large datasets with millions of files on a single machine. Indeed, while indexing the largest dataset with

	potree-converter	*Entwine*	Ours (*Poisson-disk sampling*)	Ours (*grid-center sampling*)
Wellington	70h 39m	N/A	8h 26m	**7h 27m**
DCPP	21h 16m	9h 29m	4h 21m	**4h 8m**
Utah	1h 47m	52m 17s	17m 26s	**17m 23s**
DoC	45m 43s	13m 35s	6m 25s	**5m 26s**

Table 1: *Runtime for indexing each of the test datasets, compared between potree-converter, Entwine and our system*

	potree-converter	*Entwine*	Ours (*minimum distance*)	Ours (*grid center*)
Wellington	**314 GiB**	N/A	317 GiB	322 GiB
DCPP	**100 GiB**	123 GiB	226 GiB	227 GiB
Utah	15 GiB	16 GiB	**11 GiB**	**11 GiB**
DoC	**4.1 GiB**	4.5 GiB	7.6 GiB	7.8 GiB

Table 2: *Data sizes of the resulting datasets after indexing, compared between potree-converter, Entwine and our system*

	potree-converter	*Entwine*	Ours (*minimum distance*)	Ours (*grid center*)
Wellington	**10.85M**	N/A	13.28M	13.66M
DCPP	3.1M	**1.78M**	7.65M	7.58M
Utah	630k	**257k**	485k	560k
DoC	144k	**87k**	186k	200k

Table 3: *Resulting number of octree nodes, compared between potree-converter, Entwine and our implementation*

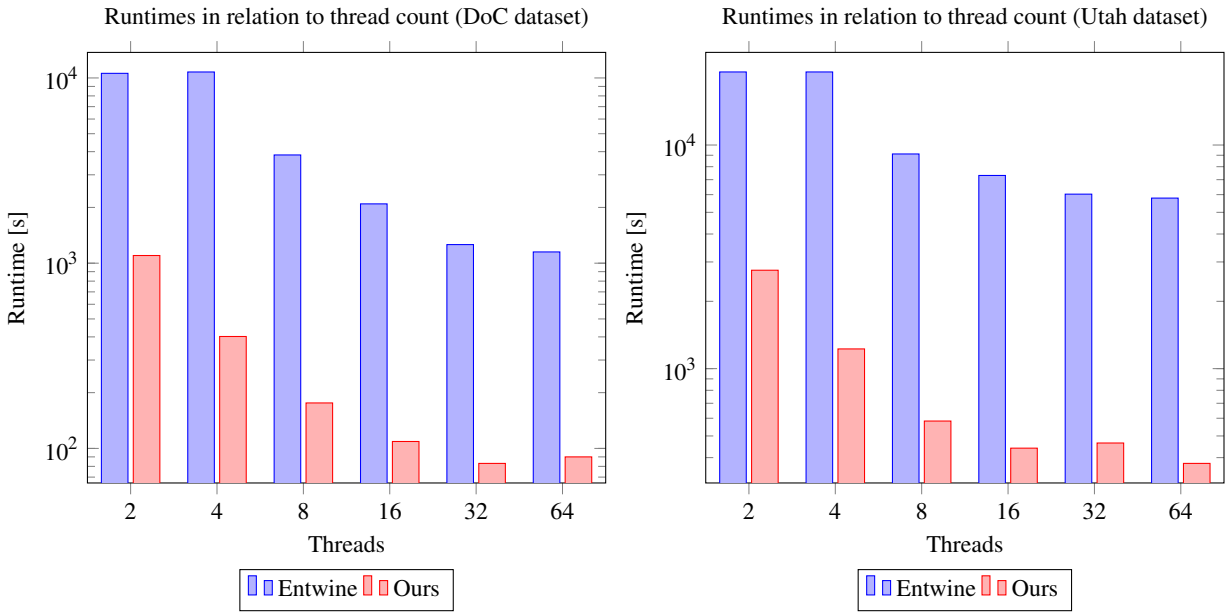

Figure 6: *Runtime performance in relation to number of threads used for Entwine and our system. Tested on the DoC dataset (left) and the Utah dataset (right)*

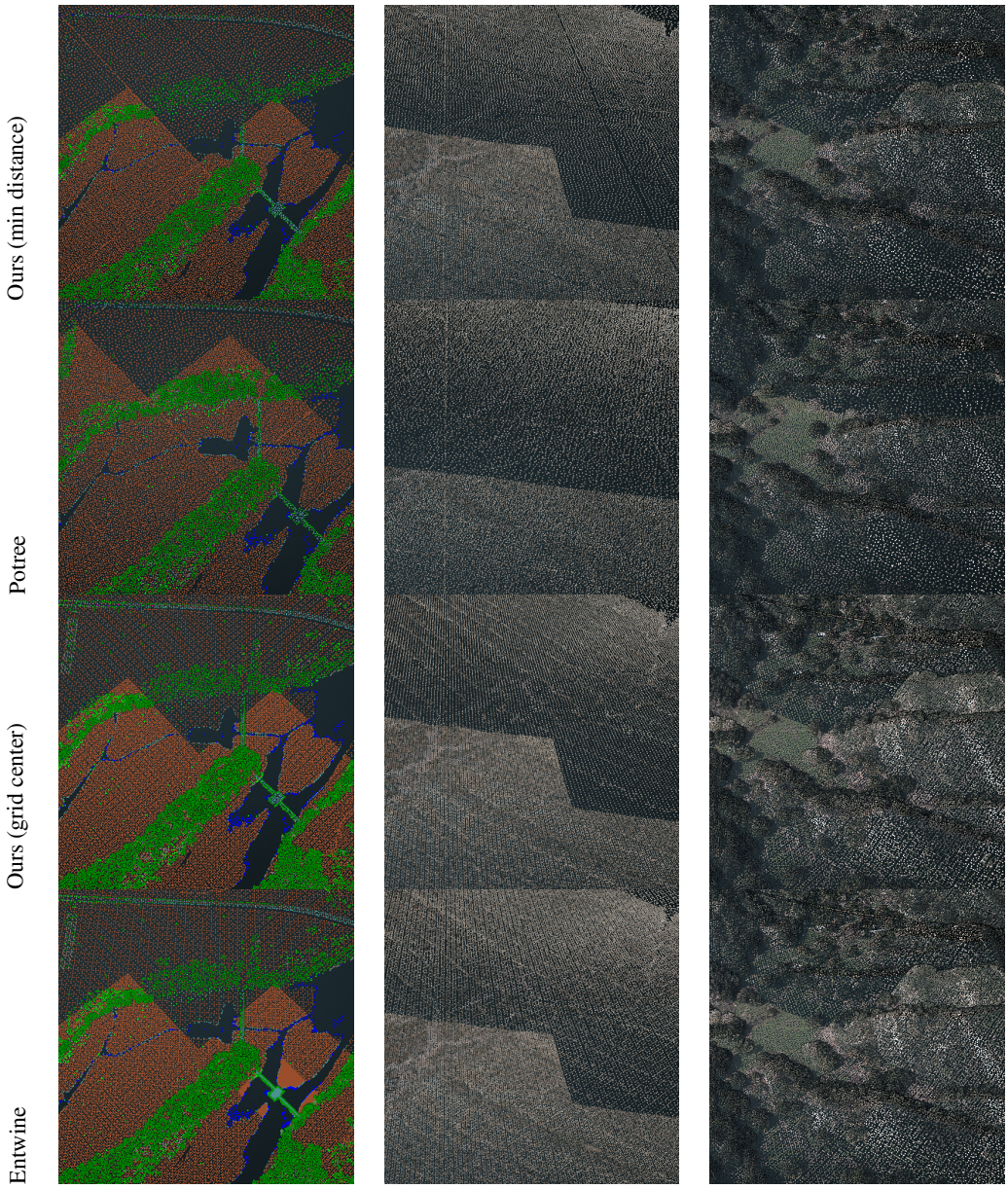

Table 4: *Visual comparison of the resulting point cloud octrees for the DoC (left), Utah (middle) and DCPP (right) datasets*

52.5 billion points in our experiments, we started to see problems with hash collisions of file names due to a faulty implementation in the ext4 file system.

7. Conclusion

We introduced a system for the creation of visualization-optimized acceleration structures for point cloud data. Our system achieves significantly better performance than other widely used systems, both on HDD and SSD systems, while at the same time produces compatible output data for both the *Potree* system and the standardized 3D Tiles file format. To achieve this, we modeled the point cloud indexing process as a recursive task graph, which gets scheduled to a high number of concurrent threads. Points are sorted by 3D Morton indices to quickly distribute points onto multiple concurrent tasks, while at the same time retaining good data locality. We demonstrated the efficiency of our system in two experiments and compared the performance, data quality and visual quality to the two indexing tools *Entwine* and *potree-converter*.

For future work, we aim to adapt our system for usage in a large-scale distributed environment, to efficiently process even larger datasets. We also would like to add the same node-collapsing capabilities that *Entwine* exhibits to our tool to reduce the total node counts.

References

[ABL16] ALIS C., BOEHM J., LIU K.: Parallel processing of big point clouds using Z-Order-based partitioning. In *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences-ISPRS Archives* (2016), vol. 41, International Society of Photogrammetry and Remote Sensing (ISPRS), pp. 71–77. 2

[And] Bit Twiddling Hacks. https://graphics.stanford.edu/~seander/bithacks.html#InterleaveBMN. Accessed: 2020-08-27. 10

[BGM*12] BALSA RODRIGUEZ M., GOBBETTI E., MARTON F., PINTUS R., PINTORE G., TINTI A.: *Interactive exploration of gigantic point clouds on mobile devices*. Tech. rep., 2012. URL: http://www.crs4.it/vic/. 2

[Bri07] BRIDSON R.: Fast Poisson disk sampling in arbitrary dimensions. *SIGGRAPH sketches 10* (2007), 1278780–1278807. 2

[ca1] PG&E Diablo Canyon Power Plant (DCPP): San Simeon, CA Central Coast. http://opentopo.sdsc.edu/lidarDataset?opentopoID=OTLAS.032013.26910.2. Accessed: 2020-08-27. 6

[ces] Specification for streaming massive heterogeneous 3D geospatial datasets. https://github.com/AnalyticalGraphicsInc/3d-tiles. Accessed: 2020-08-27. 2

[dis] District of Columbia - Classified Point Cloud LiDAR. https://registry.opendata.aws/dc-lidar/. Accessed: 2020-08-27. 6

[DMS*18] DISCHER S., MASOPUST L., SCHULZ S., RICHTER R., DÖLLNER J.: A point-based and image-based multi-pass rendering technique for visualizing massive 3D point clouds in VR environments. *Journal of WSCG 26*, 2 (2018), 76–84. doi:10.24132/JWSCG.2018.26.2.2. 2

[EBN13] ELSEBERG J., BORRMANN D., NÜCHTER A.: One billion points in the cloud - An octree for efficient processing of 3D laser scans. *ISPRS Journal of Photogrammetry and Remote Sensing 76* (2013), 76–88. doi:10.1016/j.isprsjprs.2012.10.004. 2

[ent] Entwine. https://entwine.io/. Accessed: 2020-08-27. 2

[GZPG10] GOSWAMI P., ZHANG Y., PAJAROLA R., GOBBETTI E.: High quality interactive rendering of massive point models using multi-way kd-trees. In *18th Pacific Conference on Computer Graphics and Applications* (2010), IEEE, pp. 93–100. 2, 3

[HLLL20] HUANG T.-W., LIN D.-L., LIN Y., LIN C.-X.: Cpp-Taskflow v2: A General-purpose Parallel and Heterogeneous Task Programming System at Scale. URL: http://arxiv.org/abs/2004.10908, arXiv:2004.10908. 3

[Kar12] KARRAS T.: Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. *High-Performance Graphics 2012, HPG 2012 - ACM SIGGRAPH / Eurographics Symposium Proceedings* (2012), 33–37. doi:10.2312/EGGH/HPG12/033-037. 2, 4

[las] LAS SPECIFICATION, VERSION 1.4 - R13. https://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf. Accessed: 2020-08-27. 5

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. *Computer Graphics Forum* (2009). doi:10.1111/j.1467-8659.2009.01377.x. 4

[Mor66] MORTON G. M.: *A computer oriented geodetic data base and a new technique in file sequencing*. Tech. rep., International Business Machines Company New York, 1966. 2, 5

[pota] Potree. http://potree.org/. Accessed: 2020-08-27. 1, 2

[potb] potree/PotreeConverter at 1.7. https://github.com/potree/PotreeConverter/tree/1.7. Accessed: 2020-08-27. 2

[RDD15] RICHTER R., DISCHER S., DÖLLNER J.: Out-of-core visualization of classified 3d point clouds. In *3D Geoinformation Science*. Springer, 2015, pp. 227–242. 2

[Sch14] SCHEIBLAUER C.: *Interactions with Gigantic Point Clouds*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2014. URL: https://www.cg.tuwien.ac.at/research/publications/2014/scheiblauer-thesis/. 1, 2

[Sch16] SCHÜTZ M.: Potree: Rendering large point clouds in web browsers. *Technische Universität Wien, Wiedeń* (2016). 1, 2

[SOW20] SCHÜTZ M., OHRHALLINGER S., WIMMER M.: Fast out-of-core octree generation for massive point clouds. *Computer Graphics Forum 39*, 7 (aug 2020), 1–2. URL: https://www.cg.tuwien.ac.at/research/publications/2020/SCHUETZ-2020-MPC/. 2, 7

[uta] High Resolution Topography of House Range Fault, Utah. https://portal.opentopography.org/datasetMetadata?otCollectionID=OT.102019.6341.1. Accessed: 2020-08-07. doi:{https://doi.org/10.5069/G9348HH6}. 6

[wel] Wellington, New Zealand 2013. https://portal.opentopography.org/datasetMetadata.jsp?otCollectionID=OT.042017.2193.2. Accessed: 2020-08-07. doi:{https://doi.org/10.5069/G9CV4FPT}. 6

Appendix A: Bit interleaving for 3D Morton index calculation

The Morton index calculation in our system depends on a function that interleaves the bits of three 32-bit numbers into a single 64-bit number. To achieve this, we use the following C++ function that inserts two zero-bits between every bit of the input number:

```
uint64_t expand_bits_by_3(uint64_t val) {
  val &= 0x1FFFFF; // Truncate to 21 bits
  val = (val | (val << 32)) &
   uint64_t(0x00FF00000000FFFF);
  val = (val | (val << 16)) &
   uint64_t(0x00FF0000FF0000FF);
  val = (val | (val << 8)) &
   uint64_t(0xF00F00F00F00F00F);
  val = (val | (val << 4)) &
   uint64_t(0x30C30C30C30C30C3);
  val = (val | (val << 2)) &
   uint64_t(0x1249249249249249);
  return val;
}
```

This function is an adaptation of the bit interleaving function for two 16-bit values by Sean Eron Anderson [And], extended to three 64-bit values. The final bit interleaving procedure for the order XYZ is then quite simple:

```
uint64_t interleave_bits_xyz(uint64_t x,
    uint64_t y, uint64_t z) {
    uint64_t ex = expand_bits_by_3(x);
    uint64_t ey = expand_bits_by_3(y);
    uint64_t ez = expand_bits_by_3(z);
    return ez | (ey << 1) | (ex << 2);
}
```