EG UK Computer Graphics & Visual Computing (2022)
M. Turner and P. Vangorp (Editors)

Real-time indexing of point cloud data during LiDAR capture

P. Bormann[1] and T. Dorra[1] and B. Stahl[2] and D. W. Fellner[1,3]

[1]TU Darmstadt & Fraunhofer IGD, Germany
[2]Department of Sustainable Systems Engnineering INATECH, Albert Ludwigs University Freiburg & Fraunhofer IPM, Germany
[3]Graz University of Technology, Institute of Computer Graphics and Knowledge Visualization, Austria

Abstract

We introduce a software system that is capable of indexing point cloud data in real-time as it is being captured by a LiDAR (Light Detection and Ranging) sensor. Our system extends the popular MNO (modifiable nested octree) structure so that it can be built progressively without knowing the bounding box of the point cloud. Using a task-based parallel algorithm incoming points are continuously processed and distributed to the octree nodes using grid-based sampling. Different task priority functions enable prioritization of either high point throughput or low latency. We provide a reference implementation of this system and evaluate it using both a synthetic and a real-world test scenario. The synthetic test demonstrates good scalability up to 16 threads, with maximum point throughputs of up to 1.8 million points per second. These numbers are verified on a sensor system using a Velodyne VLP-16 LiDAR sensor, where our system is able to index all data produced by the scanner in real-time.

CCS Concepts

• ***Information systems*** → *Geographic information systems; Mobile information processing systems;* ***Data structures;*** • ***Computing methodologies*** → *Point-based models;* ***Vector / streaming algorithms;***

1. Introduction

Point clouds are an important tool for many applications that require precise spatial information over potentially large areas. Use-cases for point clouds include object recognition tasks [QSMG17], such as those used by self-driving cars [Qi,17], risk assessment and disaster management [LVM*21,WZM*14], or the preservation and visualization of cultural heritage sites [PNVW*17].

The two major ways of obtaining point cloud data are the usage of LiDAR (Light Detection And Ranging) sensors and photogrammetry. Since both technologies are continuously improving in their precision, point cloud data sets are constantly getting bigger, with many of the aforementioned use cases requiring large scale point clouds with billions of points. This ultimately leads to challenges in data handling, as the size of point clouds grows beyond what regular machines can handle trivially. To make even the largest point clouds usable in visualization and analysis applications, point clouds are typically preprocessed and an index structure is created. The type of the index structure varies depending on the specific use case, but typically some form of a spatial acceleration structure such as an octree or kd-tree is used. Creating a suitable index structure for a point cloud with billions of points is a time- and resource-intensive process, which often cannot be performed in-core on most machines and requires specialized software. As a result, preprocessing times of many hours or even days are not uncommon, and the resulting index structures are often stored as separate files, resulting in duplication of the original data.

In this paper, we present an approach for indexing point cloud data in soft real-time during the capturing process with a LiDAR sensor. Our approach allows for data that is being captured to immediately be inserted into a high-quality octree structure that is similar to those generated by tools such as *PotreeConverter* [SOW20], *Schwarzwald* [BK20], or *Entwine* [ent], continuously updating the index structure with new points generated by the LiDAR sensor. In contrast to these tools, which are batch-based and require full bounding box information upfront, our approach indexes the point cloud in a streaming manner using parallel task-based processing, enabling it to run in parallel to the capturing process. A multi-root octree structure enables indexing without knowing the full bounding box, and a variety of task priority functions make it possible to prioritize either high point throughput or low latency. To demonstrate our approach, we implemented a point cloud server that performs the indexing and allows clients to query data during the indexing process, notifying them of any changes in the index structure that are relevant to the queries. We are able to achieve indexing rates of about 1.8 million points per second on commodity hardware, which is high enough to keep up with the point output of most current LiDAR scanners. Our approach supports the same grid-based sampling as *Entwine* and *Schwarzwald* and thus achieves similar visual quality.

The key contributions of this paper are:

- To the best of our knowledge, this is the first approach that is

DOI: 10.2312/cgvc.20221173

able to perform point cloud indexing in real-time during LiDAR capturing

- A stream-based indexing algorithm based on the popular MNO-structure, using a priority queue to keep the latency between capturing a point at the scanner and inserting that point into the index low
- A detailed evaluation of the effect that different priority functions have on expected point throughput and latency
- An evaluation of our indexing algorithm on a real-world scanner system, using a Velodyne VLP-16 sensor and running on an nVidia Jetson AGX Xavier system

2. Related Work

Point cloud indexing can be grouped into two major categories: Visualization-optimized indices and general-purpose indices. Wimmer and Scheiblauer [WS06] were the first to introduce a nested octree structure to enable point rendering without knowing sampling densities or point normals. Based on their approach, the *modifiable nested octree (MNO)* structure was developed, which stored points as a regular grid inside the octree nodes [Sch14]. While initially developed to enable easy modification of the points within the octree, it is mainly used for representing different *levels-of-detail (LOD)* of a point cloud due to the work by Schütz [Sch16], who came up with a variety of sampling strategies for generating LODs. While the sampling gives visually pleasing results, it introduces a non-trivial runtime overhead for the construction of the MNO, resulting in a variety of tools that aim to generate the MNO index as fast as possible. Notable works are *PotreeConverter* [SOW20], *Schwarzwald* [BK20], and *Entwine* [ent], which run on a single machine, as well as the Cloud-optimized approach by Kocon et al. [KB21].

While these systems focus heavily on the visual aspect by answering view frustum queries quickly, there are also general-purpose systems that can perform a broader set of queries on point cloud data. Van Oosterom et al. [VMRI*15] evaluated several database management systems (DBMS) for their usage with point cloud data, and proposed grouping points by space-filling curves. This approach was generalized to higher-dimensional indices with the *HistSFC* system by Liu et al. [LVMV20]. With the *Point Cloud Server* [CPP17], Cura et al. developed a system that unifies requirements from many different application domains into a single point cloud DBMS.

All these systems require some form of preprocessing to generate the respective index structure, sometimes with additional overhead for importing data into the DBMS. In terms of interactive indexing of point clouds, GPU-based approaches for generating accelerators such as bounding volume hierarchies [JG21] or kd-trees [ZHWG08] have been studied extensively. While these approaches can index multi-million point datasets in a few milliseconds, the resulting accelerators lack the LOD support that is crucial for efficiently visualizing larger point clouds.

3. Approach

Any point cloud index that is geared towards visualisation must support efficient spatial queries based on the view frustum of the virtual camera with multiple levels of detail. In order to handle large point clouds, both indexing and querying must be possible using out-of-core methods. For real-time indexing of point clouds, two additional requirements apply:

High indexing throughput Points need to be processed on-the-fly, as they arrive from the sensor. While fast indexing is desirable for batch-processing, with real-time indexing the index must be able to keep up with the incoming points from the sensor. The number of points per second generated by current LiDAR sensors is typically in the magnitude of 10^5 to 10^6 points per second.

Query result updates In order to provide a real-time view of the point cloud during capturing, there needs to be a mechanism that keeps query results up to date when new points are inserted. The delay between receiving a point from the sensor and it being visible to clients should be low enough to still give a real-time impression.

Our implementation ensures the first requirement by using an indexing scheme that parallelizes well and by issuing a number of optimisations to reduce the number of load and store operations. We are also using a modified octree structure that can be built without bounding box information. The second requirement is satisfied by keeping an open socket connection between the point cloud server and the client visualization applications, through which the server can send updates to the clients whenever an octree node that is in use by a client changes.

3.1. Index structure

The index structure described in this paper is based on the MNO datastructure by Scheiblauer [Sch14]. The MNO is an octree variant that stores a grid of points in every node. The grid spacing halves in every tree level, so that the tree levels form the different levels of detail.

When indexing points in real-time, the bounding box of the full point cloud is not known up front. However, the vanilla MNO needs this information to determine the correct size of the root node. Instead of a single root node, we use a regular grid of fixed sized nodes in the first tree level. New root nodes are created on demand when the first point falling into its area is indexed. This structure is similar to the hybrid hash-grid/MNO approach by Kocon et al. [KB21]. Figure 1 shows an example of this datastructure.

The index is stored on disk using the LAS file format [Ame13], or optionally the compressed LAZ format. Unlike the PotreeConverter [SOW20], we cannot store the full index as a continuous point buffer in a single file. This is because we do not know in advance how much space to reserve for each node. Instead, each node is stored as a separate file. The files are named based on the LOD of the node and the position in the grid that the nodes of this level of detail form.

In memory, a node is represented as a hash map that maps each occupied cell in the inner node grids to the contained point.

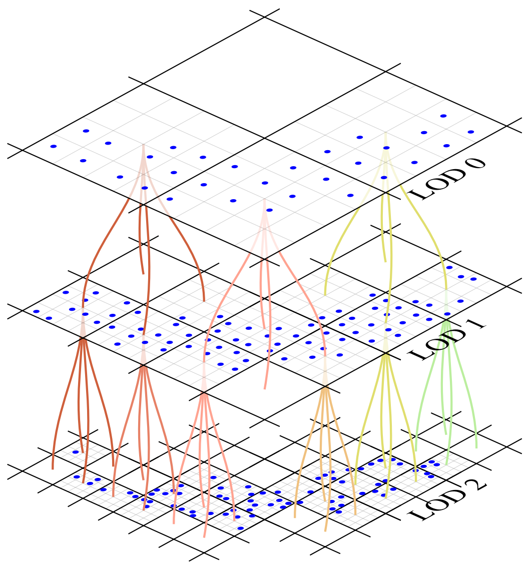


Figure 1: *Example of our datastructure. Each LOD consists of a sparse grid of nodes, seperated using black lines. The nodes contain an inner grid of points, that is drawn using grey lines. The example shows three root nodes in LOD 0.*

3.2. Indexing process

In batch-process indexing, the order in which points are processed is important for the indexing performance, because more locality in the point data leads to nodes having to be swapped between memory and disk less frequently [Lei13]. While batch-process indexers can pre-sort [Sch14] or tile [SOW20] the raw point data, we do not have control over the order in which points are added in the real-time indexing use case. Our indexer uses an in-memory point buffer for each node that collects incoming points local to this node's subtree. We call this the *inbox* of the node. It enables us to insert multiple points in one step, having to load and store the updated node only once, rather than for each incoming point seperately.

The indexer works using a top-down approach: New points are initially added to the inbox of their corresponding root node. From there, they descend into the tree until a matching node is found. This is done by repeatedly performing the following steps:

1. Choose a node with a non-empty inbox and take the contained points, leaving it empty.
2. Load the corresponding node from disk. If the node does not already exist, a new empty node is created.
3. Try inserting the points into the node using grid center sampling. This will accept points for grid cells that are still empty. For occupied cells, whichever point is closer to the center of the grid cell is accepted and the other one is rejected.
4. Store the modified node to disk.
5. Split the rejected points into eight groups depending on which child node they fall into.
6. Add each group to the inbox of the corresponding child node.

The indexer parallelizes trivially: There are multiple worker threads that execute these indexing steps independently from each other. Synchronisation is only needed when accessing the inboxes, which are shared between all workers, and when choosing the next node to process, to make sure that a node is never picked for processing while it is already being processed by another thread.

The performance of the indexer depends on how the next node to be processed is chosen in step 1. We implemented and evaluated several task priority functions that assign a priority value to each node. Out of all available nodes we always choose the one with the highest priority:

NrPoints We always select the node with the highest number of points in its inbox. This puts a strong emphasis on the core idea of processing many points together, so that we have to swap between disk and memory less frequently. For a node to get chosen for processing, it will have to wait until enough incoming points have been collected for that node.

TaskAge This priority function selects nodes based on the time for which their inbox has been non-empty. Old inboxes are preferred over new inboxes. While the $NrPoints$ priority function has no guarantee on how long it lets a node with incoming points wait, the $TaskAge$ priority function tries to achieve that all nodes with incoming points will be processed in a timely manner. It ensures a relatively steady flow of points down the tree for all parts of the data structure, which leads to good caching properties.

NrPointsTaskAge A combination of both previous priority functions.

If we have $NrPoints$, the number of incoming points for a node, and $TaskAge$, the number of time steps for which the inbox was non-empty, then equation 1 shows how to calculate the priority of a node:

$$Priority = NrPoints \cdot 2^{TaskAge} \tag{1}$$

The $NrPoints$ priority function has the disadvantage that at higher tree levels less points fall into a node and the inbox grows extremely slowly, because the side length of the nodes shrinks exponentially with the tree level. This leads to very high latencies in the higher tree levels, because it takes a long time for enough incoming points to accumulate for the node to be picked for processing. Also, it leads to many cache misses, because a node is often evicted from cache before it is picked for processing again. In the worst case, nodes are never selected for processing at all, because their inbox stopped growing completely after the LiDAR sensor moved away. To alleviate this, we added the additional factor $2^{TaskAge}$ to the $NrPoints$ priority function, that boosts the priority of nodes which have been waiting for processing for a long time.

TreeLevel Among all candidates with a non-empty inbox, this priority function always selects the node with the largest tree level. This leads to new points being collected in the root nodes only. Once a root node has been picked, its points are fully inserted into the tree before the next root node can be picked.

An overview of the whole indexing process can be seen in fig. 2.

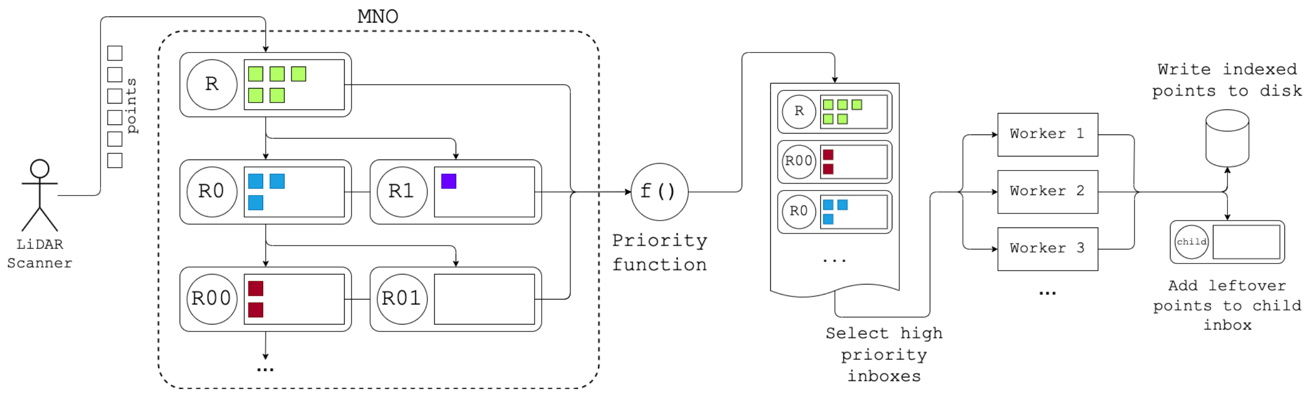


Figure 2: *Overview of the indexing process starting at the LiDAR scanner and ending at the indexed points on disk*

3.3. Optimizations

The index structure already makes an effort at keeping the number of load and store operations low. The cost for loading or storing a node can be further reduced by an additional caching layer. All nodes are accessed via a global LRU (least-recently-used) cache.

During indexing, loading and storing of frequently accessed nodes is less expensive, because they can be accessed from the cache rather than disk. As long as the sensor does not move, points will mostly fall into the same set of nodes. If the cache is large enough to fully cover this set, the disk only needs to be accessed for nodes entering or leaving this set as the sensor moves. The cache is shared between the indexing process and query execution. This allows the point data of updated nodes to be made available to the query without requiring an extra round trip to disk. For each node, the cache can store the point data either as LAS encoded binary data, or in its decoded form, or both. The point data is lazily (de)serialized into the representation that is required when accessing it. This avoids unnecessary (de)serialisation operations which is especially important if compression is enabled, which makes LAS/LAZ encoding and decoding expensive.

Small insertion operations with only a handful of points do not contribute much to the overall indexing progress, but the average cost for loading and storing the point data is the same as for any other insertion operation. We implemented an optional optimisation that tries to avoid such insertion operations.

In addition to the sampled points, each node can store up to n points that have been rejected in the grid center sampling step. We call these *bogus points*. When enabled, we add the rejected points to the node's bogus points after the grid center sampling step. If this list contains less than n points, we store the node as is. Only if the number of bogus points exceeds the threshold n, we empty the bogus points list and let the points further descend down the tree.

In leaf nodes, bogus points help to avoid excessive tree heights. Here, they are equivalent to how the first version of PotreeConverter [Sch16] only expands a leaf node into eight child nodes if a sufficient number of points is exceeded. Another advantage of bogus points is that, when used together with the *NrPoints* priority function, they help counteract its problems with slowly growing inboxes in the higher tree levels.

The disadvantage of using bogus points is that they lead to visible irregularities in the point density when rendering a query result. Figure 3 shows an example. Since the bogus points lead to a local increase of the point density, these artefacts are only visible when rendering with a relatively small point size compared to the average point distance in the selected LOD. Therefore, we deem this not to be an issue in most practical visualisation applications. Alternatively, excluding bogus points from the query result hides these irregularities. However, from a correctness perspective this means that some of the captured points will never be visible in query results, as if they are not part of the point cloud. Considering this, the threshold n has to be picked carefully, acting as a trade-off between render quality and indexing performance.

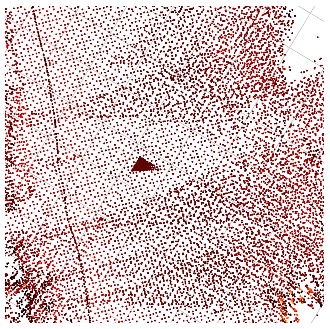


Figure 3: *Artefact caused by bogus points*

3.4. Implementation

In order to show that the previously described index structure also works well in practice, it was implemented as a part of a proof of concept. The system consists of three network-connected components, as shown in fig. 4.

The *LiDAR Server* is the core of the system. As the name suggests, it plays the role of a central server. Both the point source and the viewer connect to the server as clients. It is responsible for indexing and storing the point data that it receives from the point source. Multiple clients can connect and subscribe to queries. The LiDAR server will execute these queries and send the query result

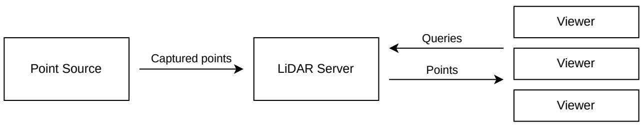


Figure 4: *Architecture of the proof of concept implementation*

Dataset	Capture duration	Points	Size
Indoor 1	1 m 31 s	6.39M	216 MB
Indoor 2	2 m 32 s	11.94M	404 MB
Outdoor 1	5 m 23 s	89.8M	2.7 GB

Table 1: *Properties of the point clouds used in the evaluation*

back to the clients, while also keeping the query result up to date after new points have been received from the point source.

The *Point Source* is a small application that connects the LiDAR server to the physical capturing setup. It receives point and trajectory data from the sensors and forwards it to the LiDAR server, while transcoding it to the expected data format. Keeping the point source as an independent service from the LiDAR Server has the advantage that it is easy to connect our system to a new capturing setup by just swapping out the point source, which acts as an adapter between LiDAR Server and the specific LiDAR sensor used.

The *Viewer* visualizes a live view of the point cloud as it is being captured and indexed. The user can freely navigate in 3D space and the viewer will always issue a query for the currently visible subset of the point cloud to the server. An unlimited number of querying clients and at most one point source are allowed to connect to the server in the current implementation.

We released the source code for all three components under an open-source license on GitHub [Fra22].

4. Evaluation

First we evaluated the limits of the index structure in terms of point throughput and scalability using a synthetic test scenario. Additionally, we performed a real-world end-to-end test where our software was run live on a sensor system during point data acquisition.

Table 1 gives an overview of the point clouds used for the evaluation. The indoor datasets *Indoor 1* and *Indoor 2* were generated during the real-world test, the *Outdoor 1* dataset was generated upfront and used for the synthetic test.

We used three different devices for the evaluation. Table 2 gives an overview of their respective hardware configurations. The *Virtual Server* and the *Laptop System* were used for the synthetic tests. The *Nvidia Jetson* device is part of the capturing setup that we ran the indexer on during the real-world test.

For data acquisition, the sensor system is equipped with a Velodyne VLP-16 hi-res LiDAR system combined with a 360-degree RGB-camera ring. Additionally, a further Intel RealSense stereo camera is integrated. The positioning system contains a GNSS-Receiver (global navigation satellite system) combined with an

Name	Storage	CPU cores	Memory
Virtual Server	HDD	16	64 GiB
Laptop System	SSD (NVMe)	8	16 GiB
Nvidia Jetson	SSD (USB-C)	8	32 GiB

Table 2: *Overview of the used test systems*

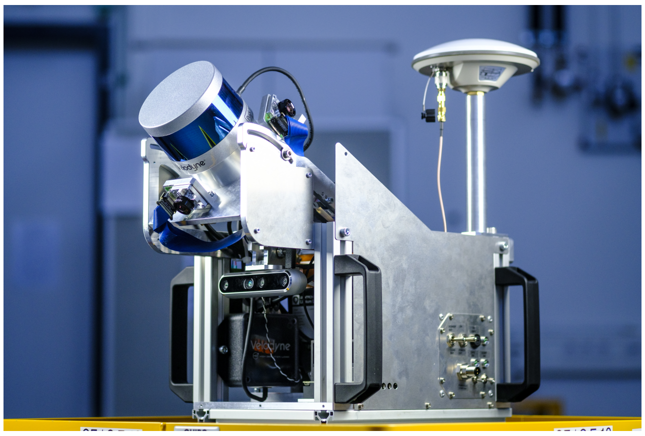


Figure 5: *Image of the scanner system that is used for the evaluation*

IMU (inertial measurement unit). Furthermore, the prediction of the position is supported by a visual odometry algorithm. For the localization of the point cloud a SLAM (Simultaneous Localization and Mapping) algorithm is used which is based on the rtabmap [LM19]. The communication of the individual sensors as well as the referencing of the data runs via the Robot Operating System (ROS) on a Nvidia Jetson Xavier as embedded GPU to provide a fast data processing pipeline. The individual sensor data streams are each provided with a time stamp, which is made available by the GNSS module. The scanner is synchronised via the NMEA (national marine electronics association) protocol and a PPS (pulse per second) signal. For the indexing of the data, only the data of the Velodyne VLP16 hi-res without colour information was used. The scanner produces 300,000 points per second with a vertical field of view of ± 10 ° and has a range of 100 m.

4.1. Synthetic test scenario

The goal of the synthetic tests was to evaluate the indexing process and data structure. Therefore, all measurements were performed in isolation and directly on the index structure, without the surrounding LiDAR server. The results do not account for additional overhead needed by the client/server approach, e.g. for transmitting the point data over the network.

For the synthetic test we used the *Outdoor 1* dataset, which was obtained from the sensor platform driving through an outdoor scene. This point cloud was repeatedly replayed to the indexer, emulating a physical capturing setup, while giving us reproducible results. By speeding up or slowing down the replay, we could also test different point rates.

The most important metric for real-time point cloud indexing is

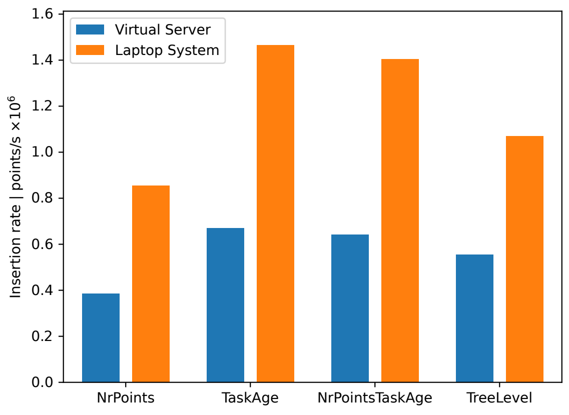


Figure 6: *Insertion rate measured for the different priority functions*

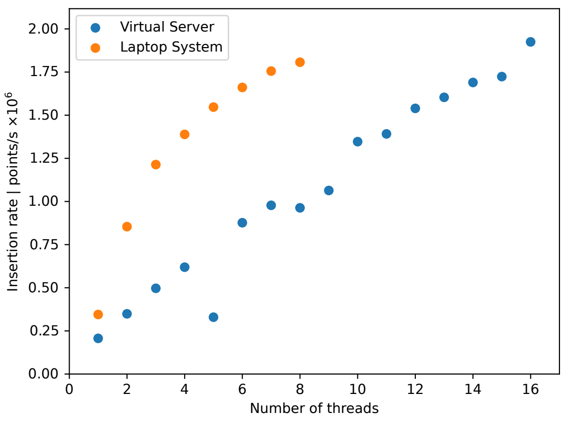


Figure 7: *Insertion rate in relation to the number of threads used*

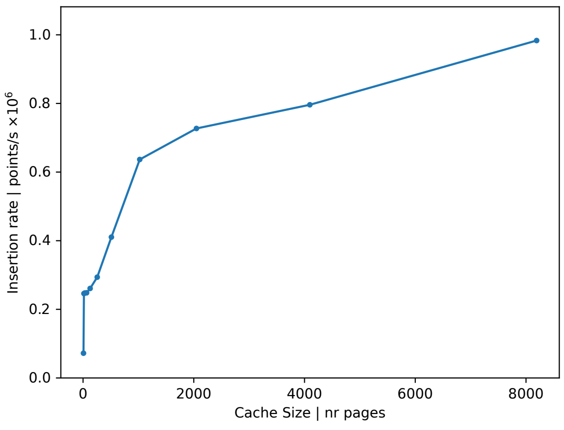


Figure 8: *Insertion rate in relation to the size of the LRU cache*

the point insertion rate, which defines how many points per second our system is able to process. If this value exceeds the point output from the LiDAR sensor, real-time indexing is possible. We measure the insertion rate by monitoring the overall number of points in all inboxes. In regular intervals, we insert enough new points to fill up the inboxes to a fixed number of points. Like this, new points are inserted as fast as previously added points are processed. From the time that it takes to index the full test dataset in that manner, the insertion rate is calculated.

Figure 6 shows the insertion rates that we measured for the different priority functions. The *TaskAge* function comes out as the winning one. However, we observed it suffering from insufficient cache sizes. Therefore, *NrPointsTaskAge* or *TreeLevel* might be a better choice in cases where the cache size is limited, for example due to memory constraints. Finally, the *NrPoints* priority function performs relatively weak, due to its issues that we already described in section 3.2.

Next we measured the insertion rate for different numbers of worker threads. The results in fig. 7 show that the parallelisation works well and the insertion rate increases with the number of threads.

We also analyzed how the size of the LRU cache and the number of bogus points affect the insertion rate. Data for this analysis can be seen in fig. 8 and fig. 9.

Our system is able to run a query and keep the query result up to date while the indexing process is running. This is used in the viewer to render a live view of the point cloud during the capturing process, but also applications like real-time analysis could benefit from this. We measured the latency of these query result updates in order to evaluate how well the index structure is suited for these kind of applications.

For the latency measurement, the test dataset is replayed and indexed at a fixed point rate. Concurrently to the indexing process, a query is executed and the query result is kept up-to-date. For each point, we record the timestamp when it was passed to the indexer for adding to the point cloud, and when it was first seen in the query result. The delay between both timestamps is the point latency. The latency values of all points are aggregated to the median latency, as well as the interval between the 10% and 90% percentiles.

Figure 10 shows the measured latencies for different insertion rates and priority functions. The latencies are stable, as there is no visible effect of the choice of priority function and also a higher insertion rate only yields a very slight increase in latency. The spike in the *TreeLevel* priority function is most likely due to fluctuating performance on the virtual server that the test ran on. In general, the latency values are around 0.1s, which is fast enough to give users a real-time impression in the viewer.

4.2. Real-world test on the sensor system

To demonstrate our system in a real-world scenario, we ran the LiDAR Server on the Nvidia Jetson of the sensor system while driving the sensor system through two different indoor scenes. Figure 11 shows a visualization of one of the datasets that were generated during this test run.

Both test runs used identical parameters:

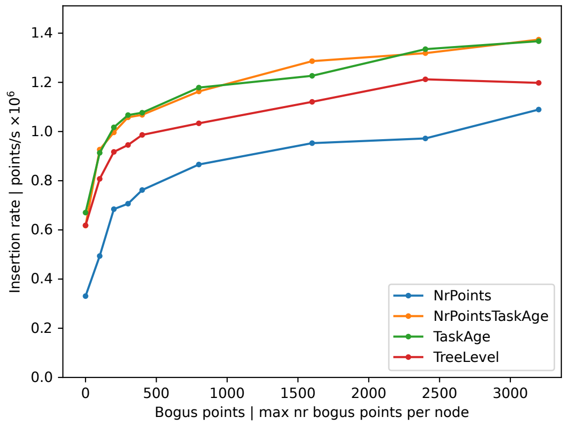


Figure 9: *Insertion rate in relation to the number of bogus points that are allowed*

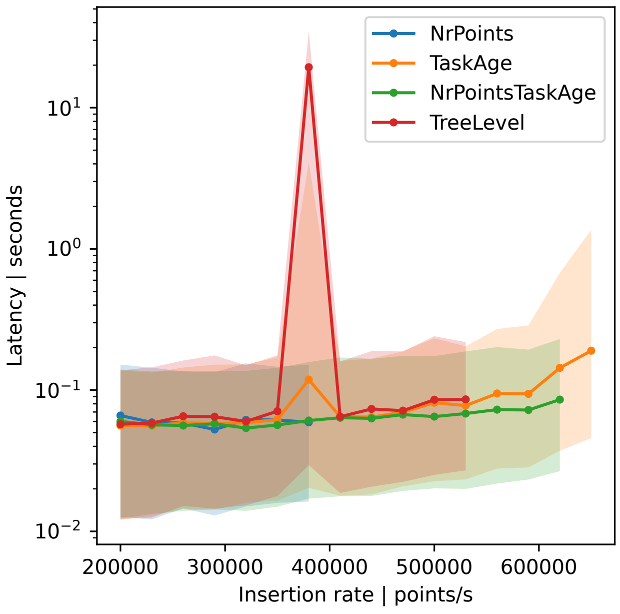


Figure 10: *Latency of the query result updates during capture*

- Worker threads: 6
- Priority function: *NrPoints*
- Maximum cache size: 500 nodes
- Maximum LOD: 10
- LasZip compression: disabled
- Bogus points: disabled

During the test runs, we recorded the number of non-empty inboxes, as well as the number of points in the inboxes. The results in fig. 12 show that the indexer can keep up with the incoming points from the sensor. If it was too slow, the inboxes would grow indefinitely, which is not the case here. This is also shown by the detail view in fig. 12c, where it can be seen that the indexer manages to completely process all points before the next batch of points arrives from the sensor.

The bump in number of tasks at the beginning and end of the measurements is because at this point the capturing system was not yet moving or already had stopped to move. With the LiDAR sensor not moving, large amounts of points were captured at similar coordinates. Once the system started moving, the indexer was able to catch up.

5. Discussion

Both the synthetic and the real-world test scenarios showed that our system is able to perform real-time indexing of point cloud data during LiDAR capturing. In the synthetic tests, the peak insertion rate reaches 1.8 million points per second, while on the sensor system, the average insertion rate exceeds the maximum scanner output of 300,000 points per second. Overall point latency is also low, on average less than 0.1 seconds with all priority functions.

While these values are promising, there are some important caveats. Similar to other point cloud indexing tools, the parallelization of our system depends on the point distribution. In the average case, points are distributed over many octree nodes, allowing good parallelization. However, there are edge cases where many points fall into the same node, such as when the sensor system is standing still, continuously generating duplicated points. If too many points accumulate and new tasks keep building up faster than the worker threads can process them, memory usage will increase continuously. In the limit, our system will run out of memory and cease to function. Potential mitigations for this will have to be implemented in the future, for example by detecting an overflow of points and writing these points to disk unindexed. This data could be processed at a future point when the system has idle time.

As mentioned in section 3.1, the stream-based processing forces us to write each node as an individual file, which is less efficient than the single-file approach of the PotreeConverter. For maximum efficiency, we chose grid-based sampling instead of the often more visually pleasing blue-noise sampling. Lastly, like all point cloud indexing systems, ours is heavily I/O dependent, running 2-3x faster on an SSD than on an HDD. Since SSDs have become quite affordable they can be found even in most consumer-grade devices nowadays. Hence, we consider this to be a marginal limitation.

6. Conclusion

We introduced a system that is capable of indexing point clouds in real-time during LiDAR capture. An index based on the state-of-the-art MNO data structure is generated and updated on the fly when new data arrives from the sensor. The index supports LOD with similar visual quality to that obtained from batch-processing tools such as *Schwarzwald* or *Entwine*. Our system achieves point throughput rates of up to 1.8 million points per second, allowing it to keep up with many current LiDAR scanners that produce in the order of a few hundred thousand points per second. This is possible through task-based parallel processing, where points are inserted into a priority queue. We provide different task priority functions

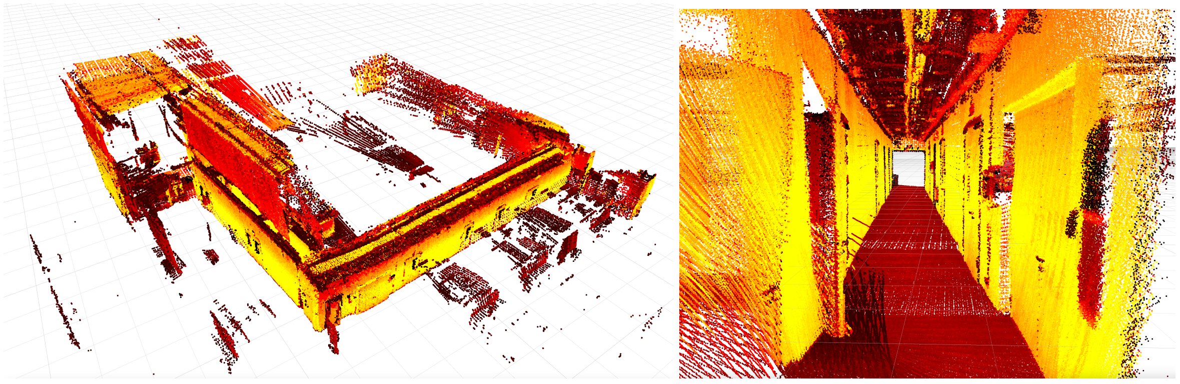
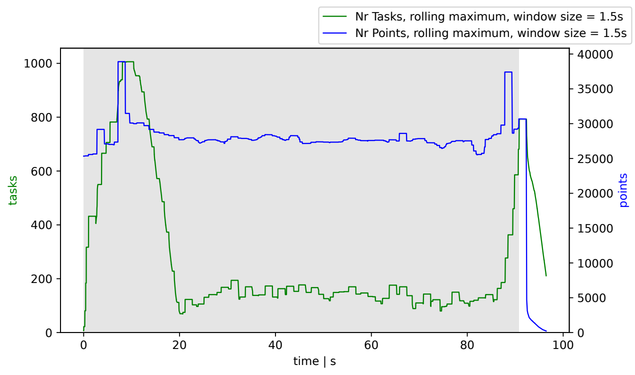
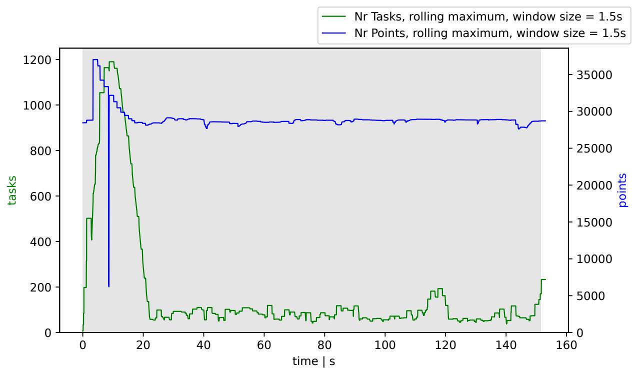


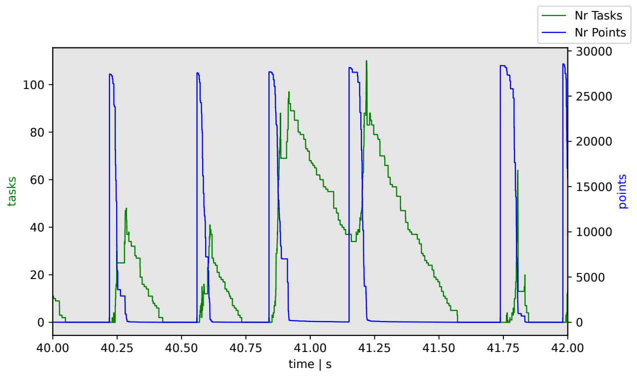
Figure 11: *The generated dataset* Indoor 2



(a) *Indoor 1*



(b) *Indoor 2*



(c) *Detail view into* Indoor 2

Figure 12: *Capturing performance during the two real-world test cases. Grey areas indicate the time for which the LiDAR sensor was active.*

that allow prioritizing either higher point throughput or lower latency. We demonstrate the usability of our system in a series of experiments, both synthetic and on a real-world sensor system.

In its current state, our system requires a sensor setup that outputs localized points, for example by using SLAM. This makes it unusable for sensor systems that perform point localization as a post-process using a high-precision trajectory. Compared to state-of-the-art point cloud indexing tools such as *PotreeConverter*, our tool has to write each node as an individual file, resulting in significantly more files. Lastly, the performance of our system deteriorates when many adjacent or duplicate points are inserted, as is the case when the sensor system is standing still.

In the future we plan to use our tool as the basis for real-time object detection on the captured point cloud, which we estimate will benefit from the voxel-like structure of the MNO datastructure. We also want to improve robustness of the system in situations where the indexer cannot catch up with incoming points to prevent out-of-memory errors.

References

[Ame13] AMERICAN SOCIETY FOR PHOTOGRAMMETRY AND REMOTE SENSING (ASPRS): LAS specification, version 1.4 - R13. https://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf, 2013. Accessed: 2022-04-06. 2

[BK20] BORMANN P., KRÄMER M.: A System for Fast and Scalable Point Cloud Indexing Using Task Parallelism. In *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference* (2020), Biasotti S., Pintus R., Berretti S., (Eds.), The Eurographics Association. doi:10.2312/stag.20201250. 1, 2

[CPP17] CURA R., PERRET J., PAPARODITIS N.: A scalable and multi-purpose point cloud server (PCS) for easier and faster point cloud data management and processing. *ISPRS Journal of Photogrammetry and Remote Sensing 127* (may 2017), 39–56. doi:10.1016/j.isprsjprs.2016.06.012. 2

[ent] Entwine. URL: https://entwine.io/. 1, 2

[Fra22] FRAUNHOFER IGD: lidarserv - implementation of lidar server, point source, and viewer. https://github.com/igd-geo/lidarserv, 2022. Accessed: 2022-05-25. 5

[JG21] JAKOB J., GUTHE M.: Optimizing LBVH-Construction and Hierarchy-Traversal to accelerate kNN Queries on Point Clouds using the GPU. *Computer Graphics Forum 40*, 1 (2021), 124–137. doi:10.1111/cgf.14177. 2

[KB21] KOCON K., BORMANN P.: Point cloud indexing using big data technologies. In *2021 IEEE International Conference on Big Data (Big Data)* (2021), IEEE, pp. 109–119. 2

[Lei13] LEIMER K.: External sorting of point clouds, 2013. 3

[LM19] LABBÉ M., MICHAUD F.: Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics 36*, 2 (2019), 416–446. doi:10.1002/rob.21831. 5

[LVM*21] LIU H., VAN OOSTEROM P., MAO B., MEIJERS M., THOMPSON R.: An efficient nd-point data structure for querying flood risk. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives 43*, B4-2021 (2021), 367–374. doi:10.5194/isprs-archives-XLIII-B4-2021-367-2021. 1

[LVMV20] LIU H., VAN OOSTEROM P., MEIJERS M., VERBREE E.: AN OPTIMIZED SFC APPROACH for ND WINDOW QUERYING on POINT CLOUDS. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences 6*, 4/W1 (2020), 119–128. doi:10.5194/isprs-annals-VI-4-W1-2020-119-2020. 2

[PNVW*17] POUX F., NEUVILLE R., VAN WERSCH L., NYS G.-A., BILLEN R.: 3d point clouds in archaeology: Advances in acquisition, processing and knowledge integration applied to quasi-planar objects. *Geosciences 7*, 4 (2017), 96. 1

[Qi,17] QI, CHARLES R AND SU, HAO AND MO, KAICHUN AND GUIBAS L. J.: Multi-label semantic 3D reconstruction using voxel blocks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), pp. 652–660. doi:10.1109/3DV.2016.68. 1

[QSMG17] QI C. R., SU H., MO K., GUIBAS L. J.: Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), pp. 652–660. 1

[Sch14] SCHEIBLAUER C.: *Interactions with Gigantic Point Clouds*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, 2014. URL: https://www.cg.tuwien.ac.at/research/publications/2014/scheiblauer-thesis/. 2, 3

[Sch16] SCHÜTZ M.: *Potree: Rendering Large Point Clouds in Web Browsers*. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, Sept. 2016. URL: https://www.cg.tuwien.ac.at/research/publications/2016/SCHUETZ-2016-POT/. 2, 4

[SOW20] SCHÜTZ M., OHRHALLINGER S., WIMMER M.: Fast out-of-core octree generation for massive point clouds. *Computer Graphics Forum 39*, 7 (Nov. 2020), 1–13. URL: https://www.cg.tuwien.ac.at/research/publications/2020/SCHUETZ-2020-MPC/, doi:10.1111/cgf.14134. 1, 2, 3

[VMRI*15] VAN OOSTEROM P., MARTINEZ-RUBI O., IVANOVA M., HORHAMMER M., GERINGER D., RAVADA S., TIJSSEN T., KODDE M., GONÇALVES R.: Massive point cloud data management: Design, implementation and execution of a point cloud benchmark. *Computers and Graphics (Pergamon) 49* (jul 2015), 92–125. doi:10.1016/j.cag.2015.01.007. 2

[WS06] WIMMER M., SCHEIBLAUER C.: Instant points: Fast rendering of unprocessed point clouds. In *PBG@ SIGGRAPH* (2006), pp. 129–136. 2

[WZM*14] WANG Z., ZLATANOVA S., MORENO A., VAN OOSTEROM P., TORO C.: A data model for route planning in the case of forest fires. *Computers and Geosciences 68* (2014), 1–10. URL: http://dx.doi.org/10.1016/j.cageo.2014.03.013, doi:10.1016/j.cageo.2014.03.013. 1

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG) 27*, 5 (2008), 1–11. 2