Fast In-Place Binning of Laser Range-Scanned Point Sets

BRUCE MERRY, JAMES GAIN, and PATRICK MARAIS, University of Cape Town and Centre for High Performance Computing

Laser range scanning is commonly used in cultural heritage to create digital models of real-world artefacts. A large scanning campaign can produce billions of point samples—too many to be manipulated in memory on most computers. It is thus necessary to spatially partition the data so that it can be processed in bins or slices. We introduce a novel compression mechanism that exploits spatial coherence in the data to allow the bins to be computed with only 1.01 bytes of I/O traffic for each byte of input, compared to 2 or more for previous schemes. Additionally, the bins are loaded from the original files for processing rather than from a sorted copy, thus minimizing disk space requirements. We demonstrate that our method yields performance improvements in a typical point-processing task, while also using little memory and guaranteeing an upper bound on the number of samples held in-core.

Categories and Subject Descriptors: I.3.5 [**Computer graphics**] Computational Geometry and Object Modeling; I.3.6 [**Computer graphics**]: Methodology and Techniques—*graphics data structures and data types*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Binning, point cloud, octree, out-of-core

ACM Reference Format:

Merry, B. Gain, J., and Marais, P. 2013. Fast in-place binning of laser range-scanned point sets. ACM J. Comput. Cult. Herit. 6, 3, Article 14 (July 2013), 19 pages.
DOI: http://dx.doi.org/10.1145/2499931.2499935

1. INTRODUCTION

Laser range scanning is increasingly used in cultural heritage documentation. A large scanning campaign can produce billions of point samples of the physical world, which require further processing such as registration, cleaning, surface reconstruction and hole-filling to be useful [Rüther et al. 2011]. This is too much data to be held in main memory, and so out-of-core techniques are required [Wand et al. 2008].

Processing point clouds out-of-core is usually nontrivial because common operations on a point also depend on the spatial neighborhood of that point. Rather than focusing on a specific task, we consider the general problem of out-of-core point-cloud processing. The goal is to preprocess the data so that it becomes possible to load each point into memory with its neighborhood for processing. Since current

This work was funded by the South African Centre for High Performance Computing (CHPC).

Authors' address: B. Merry, J. Gain, and P. Marais, University of Cape Town, Private Bag, Rondebosch 7701, South Africa; email: {bmerry, jgain, patrick}@cs.uct.ac.za.

DOI: http://dx.doi.org/10.1145/2499931.2499935

disks are 1–2 orders of magnitude slower than RAM, this preprocessing is usually I/O-bound, and reducing the number of I/O passes can substantially improve performance.

The I/O efficiency of a preprocessing algorithm can be measured as an I/O ratio: the number of bytes of I/O traffic per byte of input data. Previous work reorders the data on disk, and so necessarily has an I/O ratio of at least 2: one to read the original data and one to write the reordered version. Our contribution is an approach that exploits coherence in the original data to achieve I/O ratios only slightly greater than 1. This is achieved by always reading the original data rather than creating a reordered copy. Note that, except where otherwise stated, we do not count the I/O operations to load the data again for processing, as they can be overlapped with computations and so have minimal impact on performance.

Our strategy is to partition the samples into *bins* that are small enough to be handled in-core, which can then be processed by a variety of algorithms. Our approach is based on an octree, with each bin being an octree node. For each point, we require a position, and a *radius of influence* that determines the size of its neighborhood. Points whose spheres of influence intersect multiple bins are provided in all of those bins, to ensure that points at the boundaries of bins can be correctly processed. If per-point radii of influence are not known, one can use a conservative global upper bound. We also assume that the scans have already been registered and so the points are all specified in a single coordinate system.

Apart from the much-reduced I/O ratio, our scheme requires minimal temporary disk space (compared to creating a sorted copy) and the number of samples held in memory is bounded. Previous approaches based on a plane sweep [Pajarola 2005] provide no memory bounds. Existing octree-based approaches can provide a bound [Cignoni et al. 2003], but since they store samples in only one, bin it is still possible that hundreds of bins will need to be in memory simultaneously.

To summarize, we provide a method for spatially partitioning point cloud data derived from laser range-scanning campaigns (or other sources that exhibit spatial coherence in the point samples) with the following benefits (and consequent novelty):

(1) an I/O ratio close to 1, with the result that the partitioning preprocess runs in approximately half the time of competing schemes;
(2) a bound on the number of in-core samples required.

Section 2 discusses previous work, and we describe our implementation in Section 3. To illustrate how our approach might be used, we apply it to normal estimation in Section 4, before presenting results and conclusions in Sections 5 and 6.

2. BACKGROUND

Previous work on out-of-core geometric processing generally falls into two categories: sweep-plane approaches and approaches based on tree structures, usually octrees. In sweep-plane approaches, the vertices are first sorted along an axis. During processing, the points are streamed into an *active set*, and points that are no longer required are removed from the active set, and written out to external memory if necessary. This is done in a FIFO manner, so the active set is a slice through the dataset, with the thickness determined by the type of processing being done. The in-core memory usage is determined by the number of points that fall within the active set.

Pajarola [2005] introduces a general framework for geometric processing of point clouds, based on a sweep plane, and demonstrates how it can be used for a number of “local” operators, including finding k nearest neighbors, estimating normals, curvature and density, and smoothing. He also demonstrates that these operations can be pipelined, allowing multiple dependent operators to be applied in a single streaming pass. This is achieved by keeping an active set for each operator, and passing points that are removed from the trailing edge of one set into the leading edge of the next. Boesch and Pajarola [2009] make some refinements, including integration of the sort into the processing. They use an initial pass for principal component analysis (to determine the sweep axis) followed by an external sort, so the preprocessing has an I/O ratio of 3.

Sweep-plane approaches are also used in several surface reconstruction algorithms. A simple form of this is the ball-pivoting algorithm [Bernardini et al. 1999], which is adapted to run out-of-core by dividing space into slices and processing one slice at a time, keeping track of edges that need to be reconsidered in the next slice. Bolitho et al. [2007] implement Poisson-based reconstruction using a "multilevel" streaming approach. A virtual octree covers the entire bounding box, but only those nodes intersected by the sweep plane and their k-neighborhoods are retained in-core. There is thus an active set of octree nodes per octree level. Cuccuru et al. [2009] similarly use an octree for Moving Least Squares (MLS) surface reconstruction, again with only nodes near the sweep plane kept in memory, but they force it to be fully refined to some minimum depth in order to bound the width of the active set.

A significant limitation of sweep-plane approaches is that they do not give any hard bounds on memory usage, as it depends on the distribution of the input samples. Boesch and Pajarola [2009] show that in real-world use cases, the active set can contain up to 9% of the total data, making it unsuitable for extremely large datasets.

The other main approach to out-of-core geometry processing is to use a spatial tree structure such as an octree. This allows octree nodes to be refined until the leaves contain a sufficiently small number of points. Storing relatively large numbers of points in each leaf allows the octree structure to be kept in-core and minimizes the effect of disk latency [Wand et al. 2008].

The octree-based approaches differ substantially in how they initially construct the octree. Wand et al. [2008] perform online construction, adding one point at a time and refining existing nodes where necessary, as well as creating a new root if the point falls outside the bounding box. Richter and Döllner [2010] take a top-down approach: if the point-set is too large to be handled in-core, it is split into 8 smaller point-sets (corresponding to the child nodes of the octree root), and then processed recursively. Both of these approaches have a high I/O ratio, as each sample is read and written each time it is moved deeper in the tree.

Fiorin et al. [2007] have a more elegant solution, wherein the samples are sorted by their position along a space-filling curve, which causes all the samples in any octree node to be contiguous. A single pass is then sufficient to compute the octree structure. However, this still has an I/O ratio of 4 for preprocessing, due to the use of the external sort.

Cignoni et al. [2003] use an octree to index a mesh, with each vertex stored in a single leaf and each face stored in the leaf of one of its vertices. Their octree construction is similar to ours. In a first pass, a maximum octree depth is chosen and the triangles are scanned to determine how many would fall into each maximum-depth leaf node. The octree is then coarsened by merging leaf nodes together, as long as the resulting leaves do not contain too many triangles. A second pass over the data then distributes the triangles into the final set of leaves. Since each element is read twice and written once during this preprocessing, the I/O ratio is 3 (there are further processing steps, but as they are specific to meshes rather than point clouds we do not consider them).

When using an octree to store spheres of influence, one must decide how to store spheres that intersect multiple octree nodes. Most previous work favors storing the sample in only one octree node, which then requires all of a node's neighbors to be loaded before the node can be processed. Cignoni et al. [2003] bound the number of neighbors by constraining the difference in depth between neighbors to 3, but hypothetically this still allows a node to have hundreds of neighbors that must be loaded. It is also unclear how many nodes should be cached to avoid each node being loaded multiple times.

Instead, we reference a sphere of influence from all octree nodes that it intersects. This causes a minor increase in the number of samples stored in a bin of a given volume, but allows bins to store far more samples as only one bin needs to be in memory at a time.

Chiang et al. [1998] use a different approach when bucketing the vertices of a tetrahedral mesh: to produce H^3 bins, they first partition into H equal-sized sets along the X axis, then subdivide each of these into H equal-sized sets along the Y axis, and finally subdivide each of these into H equal-sized sets along the Z axis. This is similar in idea to a kd-tree, but with a breadth of H rather than the usual 2, and a depth of only 3. This approach requires each vertex to take part in three external sorts (one per axis), which creates significant I/O traffic.

All of these approaches have an I/O ratio of at least 2 during preprocessing, and for many of them it is higher.

3. BINNING

We now present our method of placing samples into bins. Each sample i is assumed to have a position p_i, a radius of influence r_i, and possibly other data (such as a normal) that will be needed for later processing, but which do not affect the binning process. The output is a stream of bins, each consisting of a volume of space and the data for all the samples whose spheres of influence intersect that volume. The user specifies an upper bound M on the number of samples in a bin. We assume that M is sufficiently large that no single point in space is overlapped by more than M spheres of influence, and hence the problem can always be solved with sufficiently small bins.

In practice, sphere/box intersection tests are relatively expensive and would significantly complicate some of the code, so in all intersection tests we conservatively approximate the spheres of influence by their axis-aligned bounding boxes. The spheres of influence are generally smaller than the boxes they are tested against, so this does not greatly increase the number of intersections.

Our basic approach is based on an fixed-depth octree and has some similarities to that of Cignoni et al. [2003], although we handle boundaries differently as we are not dealing with a triangle mesh. We start by building a multiresolution histogram to determine how many samples are associated with each octree node. We then select a nonoverlapping set of nodes from this octree to form *buckets*, into which we distribute the samples. Buckets with at most M samples immediately become bins. Some buckets will be leaves that still contain more than M samples, and these are subdivided recursively, similar to Richter and Döllner [2010]. However, this causes additional I/O and so we prefer to choose the leaves to be small enough, or M large enough, that this does not happen too frequently. Figure 1 shows a 2D example of this recursive reprocessing, and Section 3.1 covers this process in detail.

The basic approach requires two passes over the input data, and would thus have an I/O ratio of at least 2. To reduce the I/O ratio to just above 1, we take advantage of the high spatial coherence in input files extracted from range maps: not only are the individual range maps likely to be a coherent subset of the entire point cloud, but sequential points in a single file are likely to be very close together. We exploit this by precomputing *blobs*: contiguous ranges of samples that occupy the same leaves in the octree. Because each blob represents a large range of samples, they take orders of magnitude less storage, and are also more efficient to process. Section 3.2 describes how the blobs are created and used.

Each bin is described by a bounding box and a list of sample IDs. Section 3.3 discusses loading those samples from disk. The entire process is summarized in Algorithm 1 and depicted in Figure 2.

While the primary design goal has been to minimize I/O, we were only able to reach the full I/O bandwidth during preprocessing by utilizing multiple CPU cores. In Section 3.4, we discuss how we have used parallel programming to prevent the CPU from becoming a bottleneck.

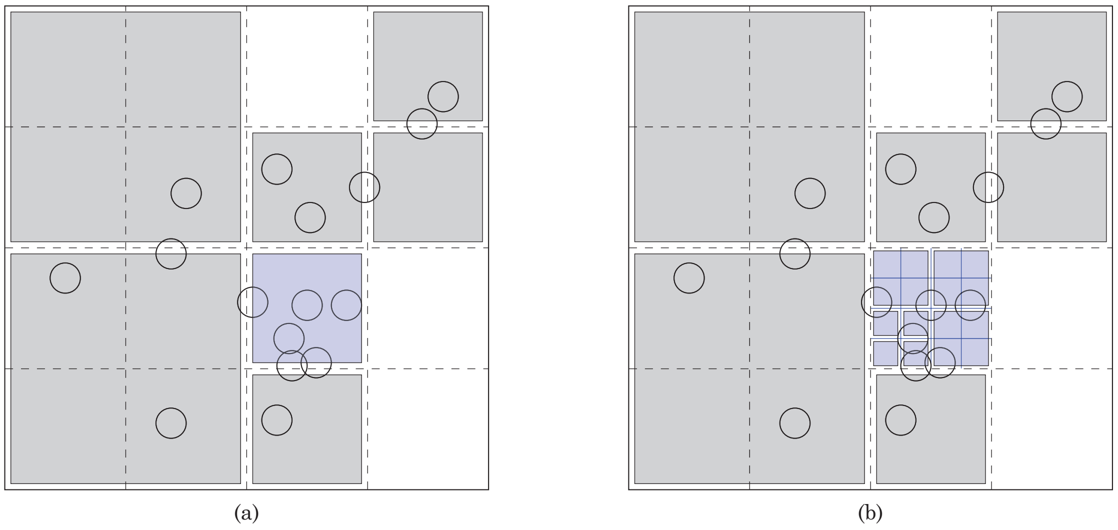


Fig. 1. Example of binning with $M = 4$. (a) The dashed lines delineate the leaves of the initial octree, and the shaded boxes show the buckets. The gray buckets become bins, but the blue bucket contains more than M samples. (b) The blue bucket is recursively processed using a higher-resolution histogram (blue lines). The blue boxes are the bins that result.

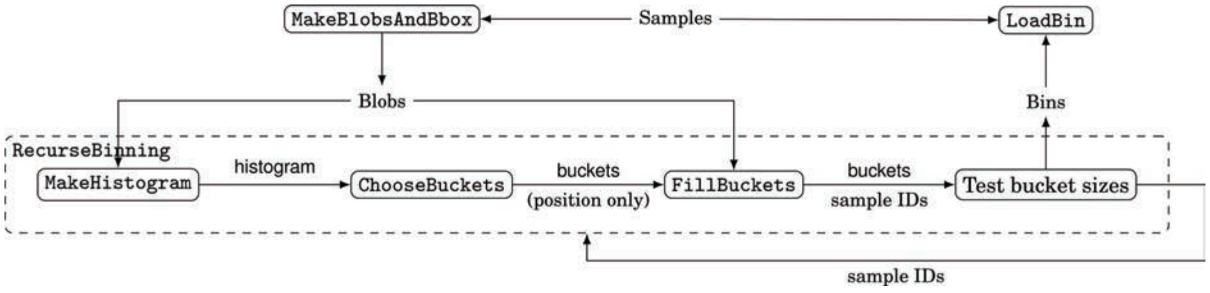


Fig. 2. Overview of the binning process. For simplicity, the bounding boxes have been omitted.

3.1 Histogramming

In this section, we describe the basic algorithm, ignoring the "blob" optimization for the moment. We thus describe streaming passes over the input samples. Section 3.2 will describe how this basic algorithm is modified to use the blobs for acceleration.

We assume that we have a bounding box for the volume of interest. At the top level, this is extracted during the blob-computation pass (see Section 3.2); during recursive binning of an overfull leaf, this is simply the region covered by the leaf.

The octree is represented using a hash table per level. This allows any node to be located in expected O(1) time, and we found that this representation was about 75% faster and also used less memory than a traditional octree with nodes that contain pointers to their children. A dense 3D array per level would be even faster, but use far more memory.

The choice of leaf size is a trade-off. Leaves should ideally be small enough that almost all of them contain no more than M samples, as otherwise they will have to be reprocessed recursively. However, excessively small leaves slow down the histogramming and also make the use of blobs less effective.

ALGORITHM 1: Overview of the binning process. Where a function takes blobs it can also take samples, by using an adapter that converts each sample to one blob.

```
function Binning(samples)
    blobs, bbox = MakeBlobsAndBbox(samples);
    RecurseBinning(blobs, bbox);
endfunction
function RecurseBinning(blobs, bbox)
    histogram = MakeHistogram(blobs, bbox);
    buckets = ChooseBuckets(histogram.root);
    FillBuckets(histogram, blobs);
    foreach bucket ∈ buckets do
        if |bucket.samples| ≤ $M$ then
            LoadBin(bucket);
        else
            RecurseBinning(bucket.samples, bucket.bbox);
        end
    end
endfunction
```

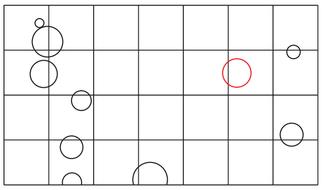
For the top level histogram, we used a leaf size of 1 m for all datasets, although this would need to be adjusted if the sampling density is significantly higher or lower. When reprocessing a volume with side length S and N samples, we heuristically choose a leaf size of $L = \frac{1}{2}S\sqrt{\frac{M}{N}}$. This yields $\frac{S}{L} = 2\sqrt{\frac{N}{M}}$ leaves to a side, and if the nonempty leaves occupy a 2D slice through the volume, then they will contain $\frac{M}{4}$ samples on average.

Each node in the octree contains a count of the samples whose spheres of influence intersect the node. Samples are streamed in sequentially and accumulated into this histogram. Parent counters are not necessarily equal to the sum of their child counters, because a single sphere of influence might intersect more than one child but will only count once towards the parent. Figure 3(b) shows a 2D example with a quadtree, based on the spheres of influence depicted in Figure 3(a).

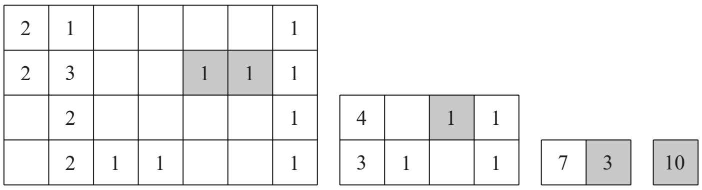
Once all the samples have been accumulated into the octree, we select a subset of the octree nodes to form buckets. These buckets must cover the spheres of influence with no overlaps. We have implemented this using a top-down walk of the octree, as shown in Algorithm 2. The descendant nodes are also updated with the bucket ID, which allows the bucket covering a leaf to be determined in expected O(1) time.

ALGORITHM 2: Selecting buckets. This is a recursive algorithm, which is initially called with the root node of the histogram tree.

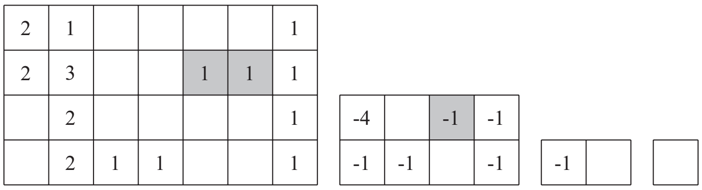
```
function ChooseBuckets(node)
    if node.count ≤ $M$ or node is a leaf then
        if node.count > 0 then
            Append node to the list of buckets;
        end
    else
        foreach child of node do
            ChooseBuckets (child);
        end
    end
endfunction
```



(a) Spheres of influence. This is an artificially sparse example: in real data the spheres would overlap far more. The grid represents the finest level of the histogram.



(b) Multi-resolution histogram. Each level of the tree is shown separately. The grey shading shows entries that were incremented due to the red sphere of influence. The empty cells are implicitly zero, but are not stored in the hash table.



(c) Delta-encoded histogram. In coarser levels, each count has been decremented by the sum of the corresponding counts at the next finer level. The gray cells show the entries that are modified due to the red sphere of influence; note that the two coarsest levels do not need to be modified.

Fig. 3. 2D example of the multi-resolution histogram.

Having selected the buckets, we now pass over the input data again to determine which buckets contain each sample (Algorithm 3). For each bucket, we store a list of sample IDs. To save space, we represent runs of consecutive IDs by the initial ID and the length. We have found this run-length encoding to be so efficient that we are able to keep the lists in-core. To further reduce space, we use a variable-length encoding for the runs: if a run starts within 2^{16} samples from the end of the previous run and has a length of less than 2^{15}, we use a 4-byte encoding, otherwise a 16-byte encoding.

ALGORITHM 3: Distributing samples to buckets

```
function FillBuckets(histogram, samples)
    foreach sample ∈ samples do
        $B := \emptyset$;
        foreach leaf in histogram intersecting sample do
            Insert leaf.bucket into $B$;
        end
        foreach bucket ∈ $B$ do
            Append sample.id to bucket.samples;
        end
    end
endfunction
```

This approach is still somewhat computationally expensive as-is. In considering how to optimize it, we note that in typical usage the leaves are significantly larger than the spheres of influence, and hence the spheres generally intersect only a few leaves. In this case, `FillBuckets` (Algorithm 3) is relatively cheap, as each sample requires time proportional to the number of leaves it intersects. `ChooseBuckets` (Algorithm 2) is also very quick, because it is independent of the number of samples. The histogramming is the slowest part of the algorithm, as it increments at least one counter at each level of the histogram for each sample.

We can improve this by representing the histogram using delta encoding. For each non-leaf node, we replace the count by the count less the sum of the counts of the children, as shown in Figure 3(c). As noted previously, this will be nonzero where a sphere of influence intersects more than one child yet is counted only once in the parent. The advantage of this representation is that adding a sample to the histogram becomes much cheaper in the common case: an internal node is updated only if the sphere of influence intersects more than one of its children. The worst case still requires one update for each level of the tree, but this rarely occurs. In our test cases, the average number of nodes updated per sample (including leaves) was less than 13.

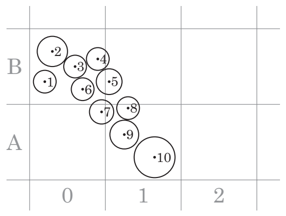
While this encoding is efficient for computing the histogram, it is inconvenient for the top-down walk used to select the buckets. Thus, immediately before selecting the buckets we do a bottom-up propagation to convert the histogram back into the original form shown in Figure 3(b). Algorithm 4 shows the optimized implementation of `MakeHistogram`.

ALGORITHM 4: Constructing the histogram

```
function MakeHistogram(samples, bbox)
    Set node counters to zero;
    foreach sphere of influence do
        foreach level in octree do // Fine-to-coarse
            foreach node ∈ level intersecting sphere do
                $c$ := number of children intersected;
                if node is a leaf then
                    Add $c$ to node.count;
                else
                    Add $1 - c$ to node.count;
                end
            end
            if only one node intersected then
                break;
            end
        end
    end
    // Convert to non-delta-encoded form
    foreach level in octree do // Fine-to-coarse
        foreach node ∈ level do
            Add node.count to node.parent.count;
        end
    end
endfunction
```

3.2 Acceleration through Coherence

A fundamental limitation of the algorithm as described so far is that it requires two passes over all the samples: one to compute the histogram, and one to compute the list of sample IDs in each bucket. This



IDs	Leaves
1–3	B0–B0
4–5	B0–B1
6–6	B0–B0
7–7	A0–B1
8–8	A1–B1
9–10	A1–A1

Fig. 4. 2D analogue of blobs. Left: sample points, their spheres of influence and their IDs, with the grid of cells. Right: blob representation, one per line. Note that although 6 intersects the same cells as 1–3, it is not contiguous with them so cannot be part of the same blob.

gives an I/O ratio of 2, and if the bounding box is not known a priori, a third pass would be required to compute it.

To reduce I/O, we exploit the spatial coherence between sequential points in the input files to accelerate the top level of the recursive binning algorithm (i.e., buckets that have too many samples and need to be further subdivided do not benefit). Specifically, we note that a sample has a high probability of intersecting the same leaf nodes as the previous sample, and hence they can be batched together and handled as a unit.

To take advantage of this, we use a data structure that we term a *blob*. 3D space is divided into an infinite regular grid of cubic *cells* of user-specified size. A blob stores a range of sample IDs, together with the cells that their spheres of influence intersect. Since we approximate the spheres by their bounding boxes, this is simply a cuboid of cells, which we represent by storing two opposite corners. Figure 4 shows an example in 2D.

The blobs are computed by making a pass over the samples, computing the corresponding cells for each sample, and if possible merging the sample into the previous blob. The blobs are written to external storage, but take orders of magnitude less space than the original sample data due to the run-length compression.

In our implementation, a blob is represented using 40 bytes: 64-bit sample IDs for the start and end of the range, and three 32-bit cell coordinates for each corner. Since blobs are always processed serially, we can reduce the storage required by compressing them. We have used a simple variable-length encoding: blobs are specified with either the full 40-byte encoding, or a compact 4-byte differential encoding. The differential encoding contains 19 bits for length, 3 bits per axis to specify the position relative to the previous blob, 1 bit per axis to indicate the extent (either one or two cells), and a 1-bit flag to indicate the encoding. Table II shows that the average bytes per blob is only slightly above 4, implying that the differential encoding is used for the majority of blobs. We have not explored other encoding schemes as this scheme is sufficient to make the blobs effective.

Once the blobs are computed, the original sample data are not needed for the top-level bucketing. Instead, the blobs are used directly. When the multiresolution histogram is created, the leaf size is set to a multiple of the cell size and the octree is aligned to the cell grid, so that each occupied cell is contained in exactly one leaf. In general, the leaves and cells will be the same size, but if necessary we increase the leaf size until the volume of the bounding box is no more than 2^{30} leaves. This helps to keep the histogram performance reasonable even if the user makes a poor choice of cell size.

Algorithm 4 is modified to replace operations on spheres of influence with operations on blobs, with histogram updates scaled by the number of samples represented in the blob. Similarly, Algorithm 3 appends the range of IDs encoded in a blob to the sample ID list for a bucket. Since the sample ID lists are also run-length encoded, the blob is either merged with the previous range (if they abut) or forms a new range. Using the blobs thus not only saves I/O bandwidth, but also reduces CPU load by enabling each blob to be processed as a single unit.

To further reduce the number of passes required, we compute both the bounding box and the blobs in a single pass. Each sample is thus loaded just once during preprocessing while the blobs are written once and read twice. This gives the preprocessing an I/O ratio of $1 + 3r$, where r is the compression ratio for the blobs (the ratio between the size of the blobs and the size of the input). Since r is less than 0.01 for all test cases, this makes the I/O highly efficient in spite of having to process blobs multiple times.

3.3 Retrieving Samples

Once preprocessing is complete, the bins are ready to be processed. For each bin, we have in memory the run-length encoded list of sample IDs for the bin, but the samples themselves are still on disk and need to be retrieved.

Unlike the blob creation pass, this phase requires random access. The spatial coherence is important once again, as each contiguous range of samples can be loaded with a single read. Furthermore, there are frequently small gaps between ranges. We found that merging multiple ranges into one larger `read` request to the operating system improved performance, in some cases by 65%. When we load a range, we combine it with all subsequent ranges that fall within the next 4 MiB. This block size is an arbitrary choice and we have not experimented with tuning it. While this causes a lot of data to be loaded unnecessarily, our results show that the access pattern allows reuse from the operating system cache.

3.4 Parallelization

The use of blobs makes the actual bucketing process very efficient, but we found that in our implementation with a single CPU core and a striped RAID array, the blob creation/bounding box pass was still CPU-limited. Additionally, we wanted to ensure that I/O and processing could be efficiently overlapped. To achieve this, we use multiple threads to improve performance in several areas.

To overlap computation with I/O, the input data are read using a separate thread that enqueues the data to the main computation thread. We have used the STXXL library to manage the external storage of blobs, and it similarly uses a separate thread for asynchronous I/O.

For the bounding box/blob creation pass, we use OpenMP [OpenMP Architecture Review Board 2008] to handle each block of data in parallel. Computing the cells intersected by a sample is expensive, but also trivial to parallelize as it is independent for each sample. Each thread also computes a local bounding box for the samples it examines, and at the end these are used to update the global bounding box. Variable-length encoding of the blobs is also done in parallel: each sub-block is encoded independently to a per-thread buffer, and these buffers are then serially written to external storage. These optimizations were sufficient to make the implementation I/O-bound on our system. Figure 5 shows the flow of data and responsibilities of each thread.

So far, we have only discussed the CPU load of the binning process, but the binning is, of course, merely a preprocess to some computation, such as estimating normals as described in Section 4. This computation is also overlapped with the sample retrieval described in Section 3.3. Once all the samples for a bin have been retrieved, they are placed on a queue, and a consumer thread fetches from the queue and performs computations. It is also possible to parallelize the computations by using multiple consumer threads, at the expense of having more bins resident in memory at a time. Once the sample retrieval thread has enqueued a bin, it can immediately start fetching samples for the next bin without waiting for the computation, as shown in Figure 6.

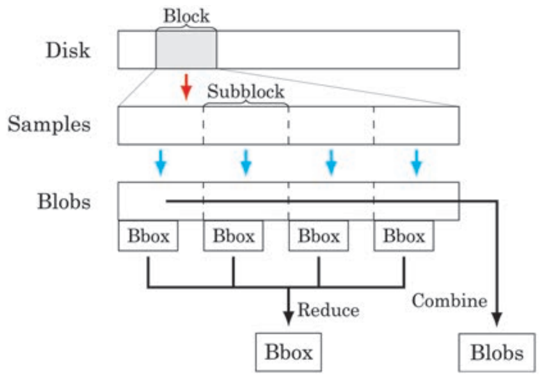


Fig. 5. Blob creation and bounding box pass. The read thread (red arrow) reads a block of data from disk and passes it to the master thread. The samples are divided into sub-blocks, each of which is processed by an OpenMP thread (blue arrows) to produce encoded blobs and a bounding box per sub-block. The master thread (black arrows) then merges the bounding boxes and concatenates the blobs.

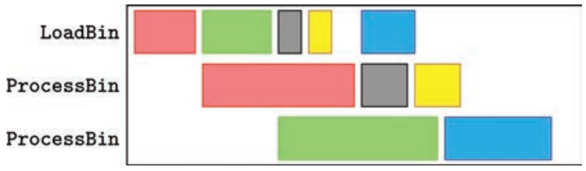


Fig. 6. Overlapping computation and sample loading. Time increases left-to-right, and each row represents a thread. Each colour represents a specific bin and shows how it is first loaded and later processed by a separate thread. The loader thread pauses before loading the blue bin so that no more than four bins are resident in memory at a time.

4. NORMAL ESTIMATION

To illustrate how the binning process might be used in practice, we provide an example application: estimating normals at the sample points from their neighborhoods. It should be emphasized that this is simply one example, and the technique is general and could be applied to other problems such as smoothing, resampling, surface reconstruction, etc.

Our normal estimation is based on the work of Hoppe et al. [1992]: for each sample, the k nearest neighbors are found, and a plane is fitted through these neighbors using principal component analysis (PCA). The normal is then taken perpendicular to the plane. The first step is thus to compute the nearest neighbors of each point, which we describe in Section 4.1. Computing a normal from the neighborhood is outlined in 4.2. For comparison, we also implemented the same normal estimation approach on top of a plane-sweep over the data, as described in Section 4.3.

4.1 Finding k Nearest Neighbors

When a bin is loaded, we compute the normals for all samples that fall inside the bin. To ensure that the correct neighborhood is found, we must set the radii of influence appropriately so that all neighbors will have spheres of influence that intersect the bin. For simplicity, we require the user to specify a single radius R that is used for all samples. We use `libnabo` [Elseberg et al. 2012] to construct a kd-tree over the samples in the bin and to search it for the nearest k neighbors within a ball of radius R.

Not all samples will have k neighbors inside a ball of radius R, and for those we do not compute a normal. We consider this a feature, as these samples are normally outliers that we would like to filter out anyway.

4.2 Fitting Normals to a Neighborhood

A normal is estimated by computing a covariance matrix for the samples in a neighborhood, and taking the eigenvector corresponding to the smallest eigenvalue. We use the `Eigen` library to find this eigenvector [Guennebaud et al 2010].

This only gives an unoriented normal. Hoppe et al. [1992] start by orienting one normal and then propagate this orientation through the neighborhood graph, but this would be complex to implement out-of-core. However, since our samples are acquired by laser range scanning, we have a simpler alternative: the input samples already contain crude normals, estimated by triangulating the grid of samples in a scan, and oriented towards the scanner. We thus choose the orientation that minimizes the angle between the original and our updated estimate.

4.3 Estimating Normals in a Plane Sweep

Pajarola [2005] uses a dynamic balanced kd-tree to index the points in the active set. We found that most kd-tree libraries we examined only support static kd-trees. We did find one library that implements dynamic randomized kd-trees [Duch et al. 1998]. However, the approach in this section using multiple static kd-trees turned out to be significantly faster.

Since the user is already expected to provide an upper bound R for the neighborhood search, we can exploit this to avoid the need for a dynamic data structure. We partition the points into slices with width R, and build a kd-tree on each slice. To find the neighbors of a point within a slice, we first search for neighbors in the same slice, and then search the two adjacent slices if they potentially contain neighbors. We thus need to keep three slices and their kd-trees in memory at a time.

4.4 Parallelization

Each normal can be computed independently of the others, so this is an easily-parallelizable problem. We considered two ways to parallelize the implementation: either a single bin can be processed by multiple threads, or each thread can process a single bin with multiple bins being processed in parallel. The latter gives better performance because it also parallelizes the construction of the kd-trees, but it requires more bins to be resident in memory. We chose a trade-off by processing two bins in parallel with eight threads per bin. This gives twice as many threads as CPU hardware threads on our target system, which helps keep the CPU saturated while the kd-tree is being built for one of the bins.

To overlap computation with I/O, we allow the loader thread to run two bins ahead of the computation. There are thus up to four bins resident in memory at a time.

We used a similar approach to parallelize the sweep-plane version: we process two adjacent slices in parallel, using eight threads per slice. Running the loader one slice ahead is sufficient to keep the computation threads from stalling. There are thus up to five slices resident at a time, as shown in Figure 7.

5. RESULTS

We now present the results of running our normal-estimation code on a number of datasets, and compare it to the version using a plane sweep. We demonstrate the efficacy of our optimizations, and also consider the overall performance and memory usage.

5.1 Experimental Setup

We ran our experiments on a desktop PC with a Core i7-2600 CPU (4 cores, 3.4 GHz), 16 GiB RAM and two 3 TB hard drives using software RAID-0 (striping). We measured the read speed for the RAID setup as 250 MiB/s. We used 64-bit Ubuntu 12.04 and GCC 4.6.

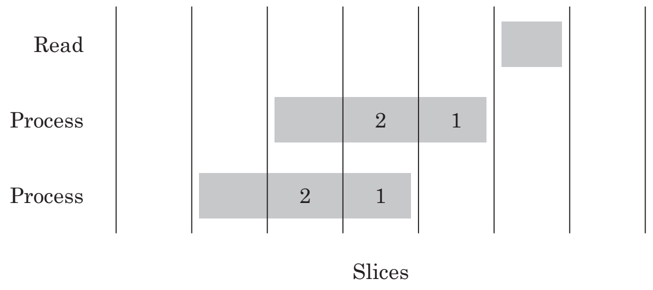


Fig. 7. Active slices. Columns represent slices while each row represents a different thread. Each processing thread computes a kd-tree over one slice (1) and nearest neighbors for another slice (2), taking into account neighbors in the adjacent slices. Note that step (2) requires that all three slices have kd-trees available, so the threads are not completely independent.

Table I. Datasets

Name	Samples	Bytes	Size (GiB)	Bounding box (m)	Radius (m)	Outliers (%)
Amman Tower	$1.26 \cdot 10^7$	31	0.36	$14 \times 14 \times 6$	0.1	0.006
Pisa Cathedral	$1.57 \cdot 10^8$	28	4.11	$118 \times 87 \times 54$	0.2	0.011
Siq	$3.82 \cdot 10^9$	31	110.29	$531 \times 1\,110 \times 157$	0.2	0.019
Songo Mnara	$6.25 \cdot 10^9$	31	180.38	$331 \times 291 \times 24$	0.2	0.008
San Sebastian	$7.25 \cdot 10^9$	28	189.09	$382 \times 329 \times 21$	0.2	0.003
Big	$6.53 \cdot 10^{10}$	28	1 701.80	$1\,146 \times 986 \times 21$	0.2	0.003

Bytes is the average number of bytes per sample. *Outliers* is the fraction of samples that have fewer than *k* neighbors inside the radius limit.

Table I lists the datasets we have used and the chosen radius limit on the nearest neighbor searches. For all datasets, we used $k = 16$ for the nearest neighbor search, limited samples per bin to $M = 10^7$ and used a cell size of 1 meter. The leaf size for the top-level histogram was also 1 meter. The "Big" dataset is a special case: it consists of nine copies of San Sebastian, arranged in a 3×3 grid, to create a test that is an order of magnitude larger than any of our real data. Figure 8 shows the datasets and the corresponding bins.

5.2 Performance

Figure 9 and Table II show the breakdown of time required to compute normals, normalized by the number of samples. Note that we have not implemented write-back of the computed normals to file, since we assume that the normals will be used in further processing such as surface reconstruction. We have separated the time into preprocessing (prior to any normals being computed) and the processing to compute normals. The recursive subdivision of over-full leaves is done in parallel with the normal computations, and hence is not included in the preprocessing time; however, this makes little difference as only 6 buckets are reprocessed across all datasets (5 for Siq and 1 for Songo Mnara).

Our preprocessing performance is close to the theoretical limit: Table III shows that the I/O ratio is only fractionally above 1, and preprocessing time corresponds to a rate of 165–223 MiB/s. The two smallest datasets (Amman Tower and Pisa Cathedral) are entirely cached in memory by the OS, and so their total I/O ratio is only slightly above 1. For the bigger datasets, the total I/O ratio is somewhat larger than the theoretical minimum of 2, and this is reflected in the proportionately longer time spent in loading during processing compared to the sweep-plane approach. However, since the total I/O ratio is less than 3, our in-place approach will still out-perform any approach that rearranges the data out-of-core.

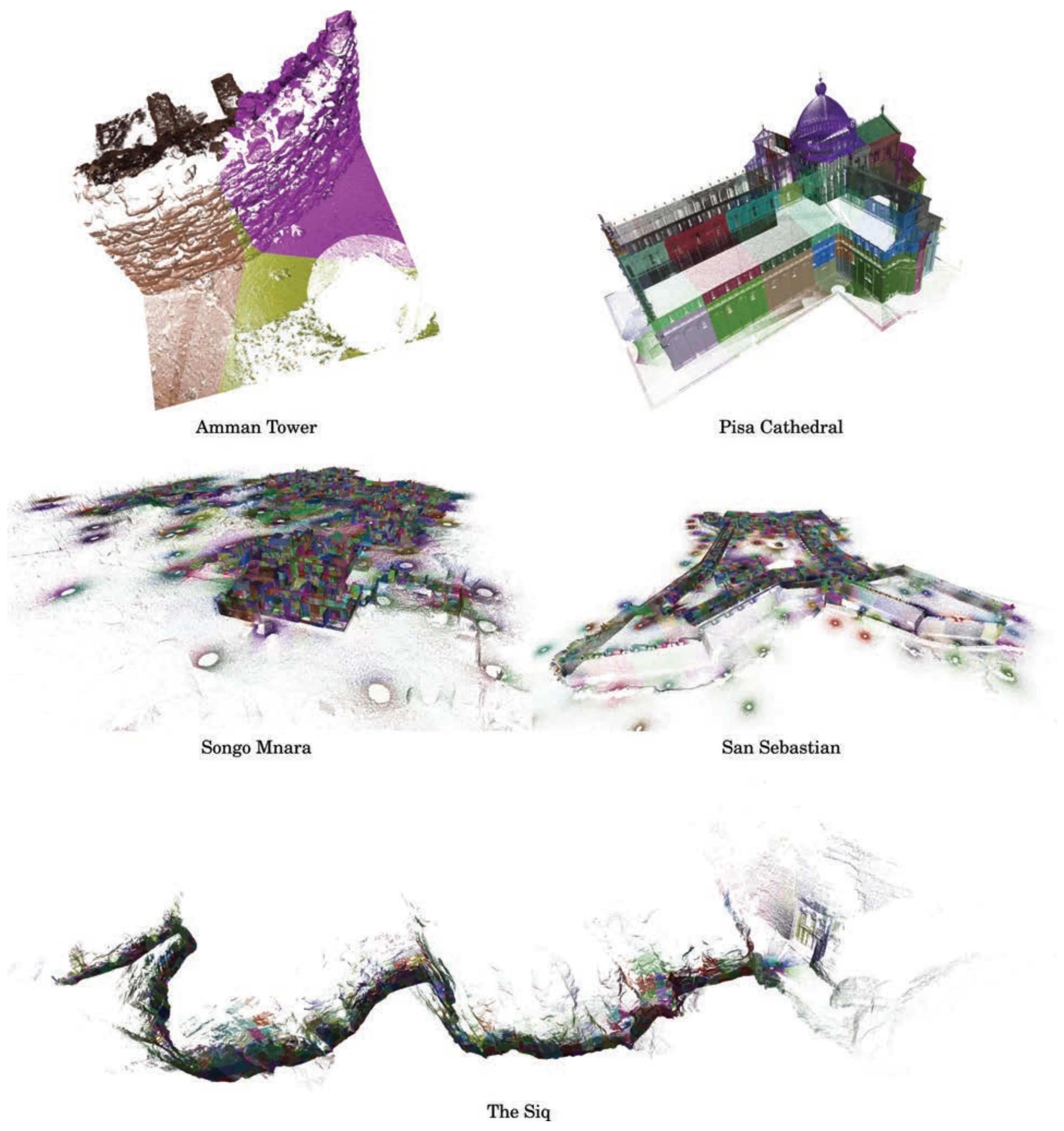


Fig. 8. Datasets and bins. The points in the interior of each bin are shown in a different color. The point clouds have been randomly subsampled to make density variations more apparent. Datasets are © Visual Computing Lab ISTI-CNR (Pisa) and African Cultural Heritage and Landscapes Database (others).

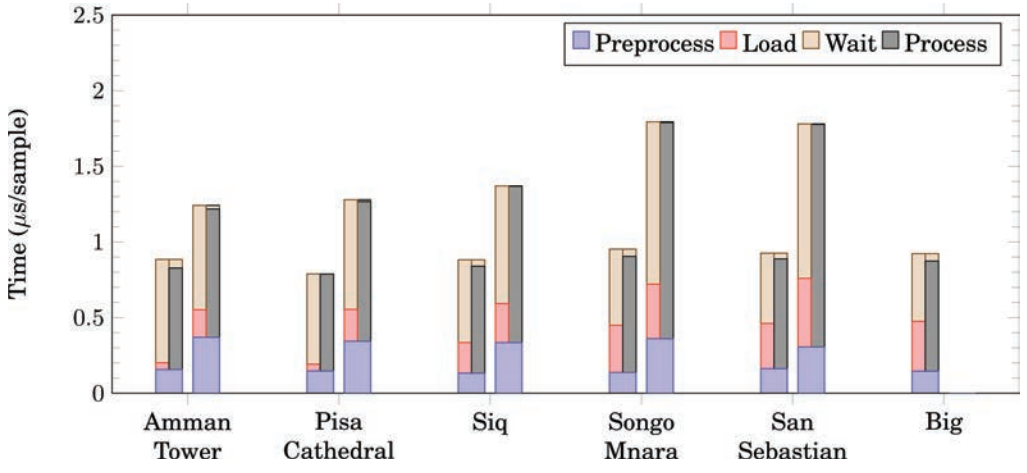


Fig. 9. Time for each phase. In each pair of stacks, the left stack is for our binning approach and the right stack is for the plane-sweep approach. The two substacks in each stack indicate the loading thread and the processing threads, which execute in parallel. The plane-sweep approach was not run for the Big dataset due to insufficient memory.

Table II. Processing times (all in seconds). Where three numbers appear in a column, they are respectively the time for our binning approach, the time for the plane-sweep approach and the ratio between the two. The time for `MakeBlobsAndBbox` is part of the preprocessing time. The total time is greater than the sum of preprocessing and processing times because there are delays while data are loaded. The Big dataset was not run with a plane sweep due to insufficient memory

Name	MakeBlobsAndBbox	Preprocess	Process	Total
Amman Tower	1.8	2.0 / 4.7 / 2.4	8.5 / 10.7 / 1.3	11.2 / 15.7 / 1.4
Pisa Cathedral	20.1	23.2 / 54.1 / 2.3	100.8 / 145.4 / 1.4	124.2 / 201.4 / 1.6
Siq	451.3	506.0 / 1 278.1 / 2.5	2 700.5 / 3 950.7 / 1.5	3 370.3 / 5 236.6 / 1.6
Songo Mnara	774.9	859.6 / 2 253.8 / 2.6	4 793.4 / 8 927.7 / 1.9	5 953.2 / 11 212.8 / 1.9
San Sebastian	1 085.1	1 175.1 / 2 217.0 / 1.9	5 263.7 / 10 673.2 / 2.0	6 715.8 / 12 910.4 / 1.9
Big	8 906.6	9 509.1 / /	47 479.3 / /	60 251.9 / /

Table III. Statistics. *Range length* is the average number of samples in a contiguous range within a bin. *Bin size* is the average number of samples in a bin. *Blob length* is the average samples per blob. The I/O ratios are computed from the kernel's reported low-level disk accesses, and may include disk I/O from unrelated processes

	Range				Blobs	Blob	Bytes/	I/O ratio	I/O ratio
Name	Range length	Ranges	Bins	Bin size	(MiB)	length	Blob	preprocessing	total
Amman Tower	561	$2.33 \cdot 10^4$	4	$3.27 \cdot 10^6$	2.02	28.77	4.83	1.00	1.00
Pisa Cathedral	436	$3.84 \cdot 10^5$	62	$2.70 \cdot 10^6$	38.13	19.48	4.95	1.00	1.00
Siq	373	$1.35 \cdot 10^7$	1 741	$2.88 \cdot 10^6$	582.65	32.98	5.27	1.00	2.25
Songo Mnara	186	$5.18 \cdot 10^7$	3 292	$2.93 \cdot 10^6$	992.21	31.31	5.21	1.01	2.37
San Sebastian	271	$4.04 \cdot 10^7$	3 577	$3.06 \cdot 10^6$	797.25	47.88	5.52	1.00	2.49
Big	269	$3.65 \cdot 10^8$	31 738	$3.10 \cdot 10^6$	7 180.07	47.84	5.52	1.00	2.86

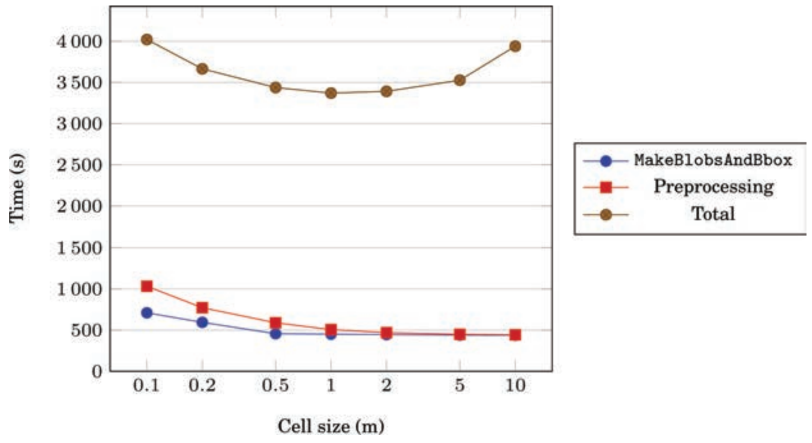


Fig. 10. Effect of cell size on performance for the Siq dataset. For cell sizes of 0.1 m and 0.2 m, the top-level leaf size is automatically increased to 0.4 m. For larger cell sizes, the top-level leaf size matches the cell size.

Figure 9 shows that the I/O ratio corresponds well with actual performance: our preprocessing is roughly twice as fast as the sweep-plane method; and while loading during processing is slower, it is entirely overlapped with computation. Apart from faster preprocessing, it also appears that the processing is more efficient with our method. While we have not investigated the reasons for this, we hypothesize that cube-shaped bins are better for kd-tree searches than thin slices.

Since the cell size is chosen by the user, it would be unfortunate if performance was highly sensitive to this parameter. Figure 10 shows the impact on performance for one of the datasets. When the cell size is very small, preprocessing performance suffers because the blobs become very short and do not compress well. When the cell size is large, there will be many over-full buckets that need to be recursively reprocessed, which affects total time. However, it is clear from the figure that performance remains reasonable across two orders of magnitude in cell size.

To verify the effectiveness of our delta-encoded histogram, we modified the code to skip the delta encoding and to propagate updates all the way up to the root (as in Figure 3(b)) and reran the Songo Mnara dataset. The time spent in `RecurseBinning` increased by 34%, from 92 s to 123 s.

We also tested whether there is any benefit in using adaptively sized bins rather than just a uniform grid, by modifying Algorithm 2 to only select leaves as buckets. On the Pisa dataset, total execution time increased from 124 s to 209 s, the number of bins increased from 62 to 31,738 and the memory required to hold the splat ranges increased from 2 MiB to 50 MiB. While performance may be better had the code been written from scratch for this approach, the huge increase in bins and memory indicate that the adaptive sizing is valuable.

5.3 Memory Usage

The only out-of-core storage we use, other than the original samples, is for the blobs. Table III shows that this is orders of magnitude less than the original data, or the copies that would be generated during an external multiway merge sort.

In-core, the storage is usually dominated by the samples that are loaded per bucket and the kd-trees that are built from them, along with some buffers for inter-thread communication. These are all independent of the size of the input data. For the Big dataset, the histogram takes 99 MiB and the sample ID ranges take 2030 MiB.

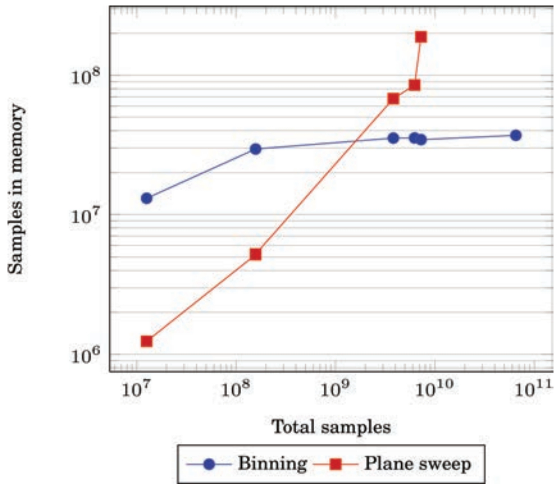


Fig. 11. Peak number of samples in memory. For our binning approach, this asymptotically approaches the upper bound of 4×10^7 (four bins of M elements). With a plane sweep, there is no upper bound, and the largest dataset was not run due to lack of memory.

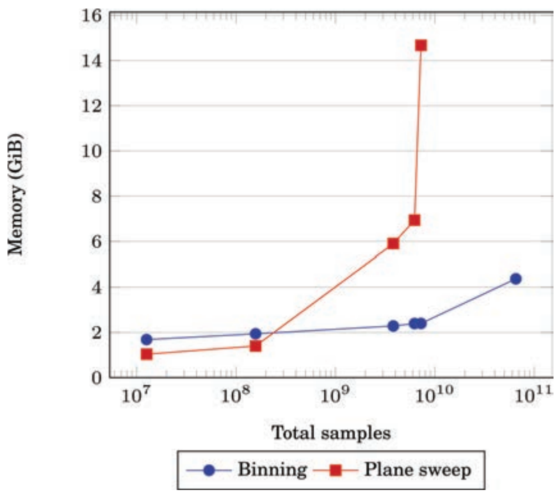


Fig. 12. Peak memory usage. Usage is measured at the `malloc` level, so includes the memory pools used by `std::allocator`. For our approach, the samples account for most of the memory in the smaller datasets, but in the largest set the sample ID ranges contribute 2 GiB.

We found that our binning approach needed a deeper loading pipeline to reduce stalls caused by differently-sized bins (even with this longer pipeline, Figure 9 shows that the processing threads spend some time waiting for the loading thread). In spite of this, both the number of samples allocated (Figure 11) and the maximum total memory (Figure 12) were lower for our binning approach than a slice-based approach on large datasets.

6. CONCLUSIONS

We have shown that as long as the points have a spatially coherent order, our novel blob representation enables spatial binning with a preprocessing I/O ratio that is essentially half that of previous work.

Our overall I/O ratio is also well below 3, which is the lower bound for algorithms that rearrange the data out-of-core.

Our method bounds the number of samples held in-core and uses only a small amount of memory even for very large datasets, and so can scale up further with the current leaf size. Arbitrarily large datasets can be handled by increasing the leaf size to prevent the octree from becoming too large, at the expense of more buckets requiring reprocessing and hence more I/O.

An argument often made for sorting points is that it only needs to be done once, after which the sorted file can be used multiple times. The same can be done in our approach, by storing per-bin information (bounding box and sample ranges) on disk, although it is slightly less flexible as the data structure is dependent on the radii of influence and not just the point positions.

While our framework is more complex to implement than a plane sweep, operations implemented on top of the framework are simpler. Rather than dealing with dynamic data structures and dependencies between slices, each bin can be processed independently as a static point cloud.

As already noted, the algorithm is specifically designed for handling range-scanned point clouds by exploiting spatial coherence. If this order was lost, for example by sorting the points for a plane sweep, the method would perform extremely poorly. Nevertheless, tools based on our method could still be used if the points were first re-sorted along a space-filling curve to restore coherence. It is also necessary for the input to be stored in a file format that allows for random access to samples.

An advantage of our method is that it can use the original data files without modifying them, and minimal extra external storage for data structures. This makes our method particularly well-suited to applications that do not need to make persistent alterations to the original point cloud. Surface reconstruction is a good candidate: for each bin, normals and density estimates can be computed in memory and used to generate an output mesh. Resampling is also likely to perform well. Applications that modify the point cloud and write it back to disk (such as simply computing and storing normals) will also work, but such cases one can no longer avoid creating a copy of the point cloud and so this advantage is lost.

6.1 Future Work

Pajarola [2005] discusses a framework for applying multiple operators to a stream, where each is able to use the results of previous operators. A composition of operators has a larger footprint; for example, it may depend on neighbors' neighbors. Conceptually, it should be possible to achieve a similar effect with binning by increasing the amount of duplication between bins to ensure that all necessary points are loaded. However, this might cause excessive I/O traffic, and it is unclear how to determine this footprint when per-point radii are used instead of a global radius of influence.

Because an octree has a fan-out degree of 8, many bins contain far fewer than $M = 10^7$ points, as seen in Table III. Splitting by a factor of 2 each time would allow bins to be larger, which would reduce the number of samples that are duplicated across bins.

At present, the histogramming occurs after the blob creation pass, but it should be possible to interleave the two, allowing for better overlap between I/O and computation during preprocessing.

ACKNOWLEDGMENTS

We would like to thank the African Cultural Heritage and Landscape Database (particularly Heinz Rüther, Christoph Held and Roshan Bhurtha) and the Visual Computing Lab ISTI-CNR Pisa for providing data. We would also like to thank the anonymous reviewers for their valuable suggestions.

REFERENCES

BERNARDINI, F., MITTLEMAN, J., RUSHMEIER, H., SILVA, C., AND TAUBIN, G. 1999. The ball-pivoting algorithm for surface reconstruction. *IEEE Trans. Visual. Comput. Graph. 5*, 4 (October 1999), 349–359.

BOESCH, J. AND PAJAROLA, R. 2009. Flexible configurable stream processing of point data. In *Proceedings of the 17th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2009)*. 49–56.

BOLITHO, M., KAZHDAN, M., BURNS, R., AND HOPPE, H. 2007. Multilevel streaming for out-of-core surface reconstruction. In *Proceedings of the 5th Eurographics Symposium on Geometry Processing*. Eurographics Association, 69–78.

CHIANG, Y.-J., SILVA, C. T., AND Schroeder, W. J. 1998. Interactive out-of-core isosurface extraction. In *Proceedings of the Conference on Visualization '98 (VIS '98)*. IEEE Computer Society Press, Los Alamitos, CA, 167–174.

CIGNONI, P., MONTANI, C., ROCCHINI, C., AND SCOPIGNO, R. 2003. External memory management and simplification of huge meshes. *IEEE Trans. Visual. Comput. Graph. 9*, 4 (October 2003), 525–537.

CUCCURU, G., GOBBETTI, E., MARTON, F., PAJAROLA, R., AND PINTUS, R. 2009. Fast low-memory streaming MLS reconstruction of point-sampled surfaces. In *Proceedings of Graphics Interface 2009*. Canadian Information Processing Society, 15–22.

DUCH, A., ESTIVILL-CASTRO, V., AND MARTINEZ, C. 1998. Randomized K-dimensional binary search trees. In *Proceedings of the 9th International Symposium on Algorithms and Computation (ISAAC '98)*. Springer-Verlag, Berlin, 199–208.

ELSEBERG, J., Magnenat, S., SIEGWART, R., AND NÜCHTER, A. 2012. Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *J. Softw. Eng. Robot. 3*, 1.

FIORIN, V., CIGNONI, P., AND SCOPIGNO, R. 2007. Out-of-core MLS reconstruction. In *Proceedings of the 9th IASTED International Conference on Computer Graphics and Imaging (CGIM '07)*. 27–34.

GUENNEBAUD, G. AND JACOB, B. AND OTHERS. 2010. Eigen v3. http://eigen.tuxfamily.org. (2010).

HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. 1992. Surface reconstruction from unorganized points. In *Proceedings of the 19th Annual conference on Computer Graphics and Interactive Techniques (SIGGRAPH'92)*. ACM, New York, 71–78.

OPENMP ARCHITECTURE REVIEW BOARD. 2008. OpenMP Application Program Interface Version 3.0. http://www.openmp.org/mp-documents/spec30.pdf.

PAJAROLA, R. 2005. Stream-processing points. In *Proceedings of the Conference on Visualization '05*. 239–246.

RICHTER, R. AND DÖLLNER, J. 2010. Out-of-core real-time visualization of massive 3D point clouds. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa (AFRIGRAPH '10)*. ACM, New York, 121–128.

RÜTHER, H., HELD, C., BHURTHA, R., SCHRÖDER, R., AND WESSELS, S. 2011. Challenges in heritage documentation with terrestrial laser scanning. In *Proceedings of the 1st AfricaGEO Conference*.

WAND, M., BERNER, A., BOKELOH, M., JENKE, P., FLECK, A., HOFFMANN, M., MAIER, B., STANEKER, D., SCHILLING, A., AND SEIDEL, H.-P. 2008. Processing and interactive editing of huge point clouds from 3D scanners. *Comput. Graph. 32*, 2, 204–220.

Received January 2013; revised March 2013; accepted March 2013