Volume xx (200y), Number z, pp. 1–12

Distributed Poisson Surface Reconstruction

M. Kazhdan[1†] and H. Hoppe

[1]Johns Hopkins University, USA
[†]Corresponding author: misha@cs.jhu.edu

Abstract

Screened Poisson surface reconstruction robustly creates meshes from oriented point sets. For large datasets, the technique requires hours of computation and significant memory. We present a method to parallelize and distribute this computation over multiple commodity client nodes. The method partitions space on one axis into adaptively sized slabs containing balanced subsets of points. Because the Poisson formulation involves a global system, the challenge is to maintain seamless consistency at the slab boundaries and obtain a reconstruction that is indistinguishable from the serial result. To this end, we express the reconstructed indicator function as a sum of a low-resolution term computed on a server and high-resolution terms computed on distributed clients. Using a client-server architecture, we map the computation onto a sequence of serial server tasks and parallel client tasks, separated by synchronization barriers. This architecture also enables low-memory evaluation on a single computer, albeit without speedup. We demonstrate a 700 million vertex reconstruction of the billion point David statue scan in less than 20 minutes on a 65-node cluster with a maximum memory usage of 45 GB/node, or in 14 hours on a single node.

CCS Concepts

• ***Computing methodologies*** → ***Mesh geometry models; Reconstruction;***

1. Introduction

Reconstructing surfaces from 3D points is a well studied problem in computer graphics and computer vision, with diverse applications (e.g., LIDAR scenes, biomedical imaging, cultural heritage capture). Numerous surface reconstruction approaches have been explored, including those leveraging computational geometry and machine learning techniques, as reviewed in Section 2. Among these, the screened Poisson Surface Reconstruction algorithm (sPSR) [KBH06,KH13] has been commonly used in the community because it is both global (providing robustness in the presence of noise and missing data) and efficient (with time and space complexity linear in the input size).

However, as the rate at which 3D point clouds grow outpaces the processing and memory capacities of commodity PCs, the Poisson surface reconstruction algorithm has begun to lag in its ability to generate surfaces at large scales. Although recent implementations have leveraged the parallelization available on modern CPUs, this has only afforded a small (e.g. 4× or 8×) speedup and fails to address the issue of memory bottleneck.

For faster execution, the algorithm must be **distributed** among several parallel processors. In typical compute clusters, each node has local memory and storage and the nodes communicate via fast networks. Supercomputers often offer large memory spaces, but these are usually Non-Uniform Memory Access (NUMA) architectures in which memory is partitioned among multiple processors such that access to nonlocal memory is significantly more costly. In either case, efficient solutions involve partitioning the problem into distributed computations that focus on local data.

We present an efficient distributed version of the screened Poisson surface reconstruction algorithm. The distributed algorithm results in near-linear speedups on a compute cluster with 65 client nodes. Additionally, the same algorithm can be executed serially on a single processor, e.g., allowing the reconstruction of a 700 million-vertex mesh from 1 billion points on a 64 GB PC.

2. Related work

Surface reconstruction Several computational geometry methods create triangle meshes that interpolate all or a subset of the data points [ABK98, BMR*99]. For resilience to noisy data, it is common to define an implicit surface that only approximates the points, often in the form of a signed-distance function [HDD*92,CL96a,CT11] or an indicator function [KBH06,KH13]. Recent work defines implicit functions with the aid of machine learning [WSS*19,BGKS20,RGA*21,CTFZ22].

Out-of-core geometry processing Many techniques are able to operate on models larger than main memory by partitioning and traversing space using cubical cells or slices [IG03, NNSM07, AGL06]. In particular, *streaming* computations advance through space using a sliding in-core window [IL05, ILS05, VCL*06, ILSS06, BKBH07].

Out-of-core surface reconstruction Some reconstruction approaches are naturally adaptable for out-of-core evaluation because their data access patterns are localized. The ball-pivoting algorithm of [BMR*99] is implemented out-of-core by partitioning the domain into slices. The VRIP accumulated signed distance field of [CL96b] is applied to large volumes by independently processing blocks of the volume and stitching together the reconstructed pieces [LPC*00]. Similar strategies have been used to process models on the limited memory available to GPUs [CBI13, NZIS13]. Reconstruction schemes based on local neighborhood fitting such as [HDD*92, ABCO*01, OBA*03] can be adapted to operate out-of-core [Paj05]. Floating scale surface reconstruction [FG14] and field-aligned online surface reconstruction [STJ*17] are other local formulations that are well suited for out-of-core, distributed computation. In contrast, methods that cast surface reconstruction as a global minimization [KH13, UB15] are challenging to evaluate as a distributed computation.

Performant Poisson surface reconstruction Focusing specifically on Poisson surface reconstruction, there has been early work on adapting the implementation to be more time and/or memory efficient. The key idea behind these approaches is to leverage the locality of the computation performed by the Poisson surface reconstruction algorithm, allowing different processors to solve for independent solution variables simultaneously [ZGHG08, BKBH09] and supporting streaming computation so that only a small subset of the data-structure needs to be memory-resident at any given time [BKBH07].

Our proposed approach is most similar to the work of Bolitho *et al.* [BKBH09] which distributes the reconstruction problem among multiple clients by separately considering the low- and high-resolution components of the problem. The entire low-resolution problem is solved by each of the clients, while the high-resolution problem is spatially partitioned along a 1D axis, with each client refining the solution within its own volumetric "slab". Consistency across slab boundaries is realized in two ways. First, slabs are defined to be overlapping. And second, data near the partition boundaries is synchronized and blended between adjacent clients.

While our approach borrows some of these ideas, it differs in a number of fundamental ways.

- It has space and time complexity $O(N/C)$, with N the number of points and C the number of clients.
- It supports the use of an adaptive octree at the coarse resolution, allowing the coarse octree to be have finer depth, thereby supporting more clients and finer-granularity load balancing.
- It minimizes the communication between the clients, so that only high-resolution information immediately at the slab boundaries needs to be shared.
- It is guaranteed to produce a watertight surface, even in the presence of machine-precision errors that result in arithmetic operations failing to be associative or commutative.

As source code for the method of Bolitho *et al.* is unavailable we built our distributed implementation from the ground up. This has the advantage of allowing us to integrate more recent developments in the Poisson reconstruction algorithm (e.g. incorporating a screening energy for better fit, using a linear-time implementation of the multigrid solver instead of the initial log-linear implementation, and discretizing the problem using the sparser system derived from first-order B-splines instead of the initial second-order B-spline discretization). Unfortunately, it also prevents a direct comparison of the two methods.

3. Review

Distributed Poisson Surface Reconstruction (DPSR) adapts both

1. screened Poisson Surface Reconstruction (sPSR) — to compute the implicit function over *distributed* clients, and
2. adaptive octree isosurfacing — to ensure that slab-adjacent clients create isosurfaces whose level-set curves are *identical* at the shared boundary.

We briefly review the implementations of both techniques.

3.1. Screened Poisson surface reconstruction

Given a set of oriented points $\{(p_i, n_i)\}$ with positions p_i and normals n_i, sPSR [KH13] proceeds by interpreting the points as a vector field $\vec{V} : \mathbb{R}^3 \to \mathbb{R}^3$ and finding the indicator function χ minimizing the energy:

$$E(\chi) = \int_{\mathbb{R}^3} \left\| \nabla\chi - \vec{V} \right\|^2 + \alpha \sum_i (\chi(p_i) - 0.5)^2 .$$

Discretizing using a B-spline basis defined over an octree $\mathcal{O}$, this reduces to solving a linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b},$$

where both the solution vector $\mathbf{x}$ and the constraint vector $\mathbf{b}$ are elements of $\mathbb{R}^{|\mathcal{O}|}$. This system is solved efficiently in a coarse-to-fine manner using the hierarchical structure of the octree.

The implementation has (roughly) the following steps:

sPSR.1 The input points are inserted into an octree of a specified reconstruction depth D. Each octree node $o \in \mathcal{O}$ stores the weighted sample $\{p_o, n_o, w_o\}$ consisting of the average sample position p_o, average sample normal n_o, and a weight w_o equal to the sample count.

sPSR.2 The weighted samples are used to construct a density estimator.

sPSR.3 Each weighted sample $\{p_o, n_o, w_o\}$ contributes to the vector field $\vec{V}$ by "splatting" the weighted vector $w_o n_o$ at position p_o, using a kernel whose width falls off with estimated local sampling density. Simultaneously, the sum of sample weights is accumulated.

sPSR.4 The value of the screening parameter α is defined in terms of the sum of the sample weights.

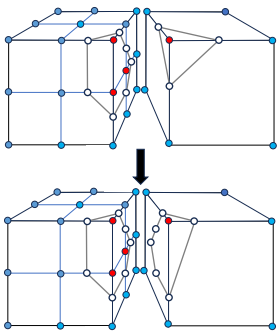
sPSR.5 The constraint vector $\mathbf{b} \in \mathbb{R}^{|\mathcal{O}|}$ is computed using both the vector field $\vec{V}$ and the weighted samples.

sPSR.6 Proceeding in a coarse-to-fine manner, a Gauss-Seidel solver relaxes the B-spline coefficients $\mathbf{x} \in \mathbb{R}^{|\mathcal{O}|}$, which define an implicit function χ.

sPSR.7 The average of the implicit function over the samples, $\bar{x}$, is computed. (Though we expect this value to be approximately 0.5, it may deviate when the screening weight α is small.)

sPSR.8 A polygonal mesh approximating the isosurface $\chi^{-1}(\bar{x})$ is computed as described below.

3.2. Unconstrained isosurface extraction



Given an unconstrained octree and an implicit function, Kazhdan *et al*. [KKDH07] compute a discrete approximation of the isosurface in the form of a polygonal mesh whose vertices and edges lie on the octree leaf cells. The approach processes the octree leaf nodes in a fine-to-coarse order, defining the intersection of the surface with a leaf node by induction on the dimension of the cells. The inset shows a visualization, for two face-adjacent octree nodes, where the node on the left is more refined than the one on the right.

Iso.0 The implicit function is evaluated at the corners of the octree leaf nodes (blue and red circles, indicating the sign).

Iso.1 Iso-vertices are computed at leaf-node edges (hollow circles). These are obtained by computing the level-set crossing given the edge's corner values (and gradients) when there is no finer node incident on that edge. Otherwise the iso-vertices from the incident finer leaf nodes are associated with each edge (bottom row).

Iso.2 Iso-edges are computed at leaf-node faces (gray edges). These are obtained by linking iso-vertices assigned to the edges when there is no finer node incident on the face. Otherwise the iso-edges from the incident finer leaf nodes are associated with the face (bottom row).

Iso.3 Iso-polygons are computed at leaf nodes. These are obtained by linking the iso-edges from the faces into polygons.

4. Approach

To distribute the computation of both Poisson reconstruction and isosurface extraction among multiple clients, it is natural to divide the problem spatially, assigning each client a region of the 3D space. The main challenges are (1) that the surfaces reconstructed in the individual client computations must be identical along the region boundaries so as to not exhibit visible artifacts (e.g., topological cracks or shading discontinuities) and (2) that the union of the clients' reconstructions should match the surface reconstructed using the traditional non-distributed approach.

4.1. Slab-based partition

As in [BKBH09], we partition the 3D space using adaptively sized slabs along a single axis, as illustrated in Figure 2. The partitioning axis is selected to be the "long axis" of the input points. Specifically, we measure the extent of the point set along multiple directions and rotate the samples so that the direction of longest extent aligns with the z axis. We also translate and uniformly scale the point set to fit into the unit cube $[0,1]^3$.

To distribute the computation among C clients, we partition space along the z axis into C slabs, each containing a similar number of input points (Figure 1(a)). We constrain the location of these slab boundaries to align with the cells of a coarse octree of depth d, where $C < 2^d$. Thus, we start by dividing space

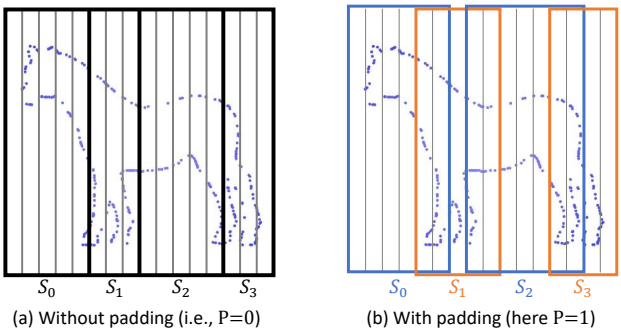


Figure 1: *2D illustration of assigning input points to 16 regular intervals and partitioning the intervals into 4 adaptively sized slabs.*

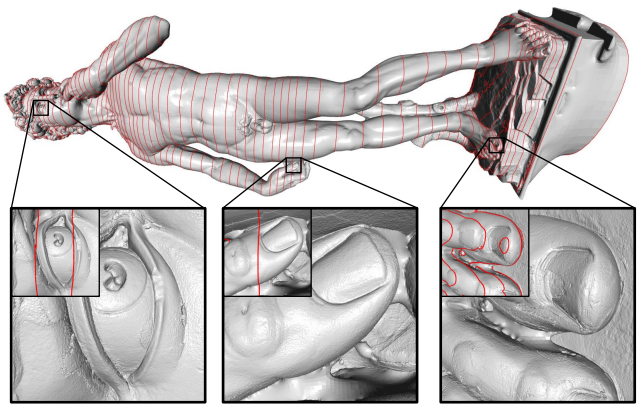


Figure 2: *Reconstruction of the "David" at depth $D = 15$ using $C = 64$ clients. The red curves reveal the partition of the domain space into adaptively sized slabs using a dynamic-programming scheme. A high density of clients is assigned to the head despite its small surface area because that region is more densely sampled.*

along the z axis into 2^d regular *intervals*, $\frac{i}{2^d} \leq z \leq \frac{i+1}{2^d}$, for $0 \leq i < 2^d$. We then determine a partition of the intervals into C *slabs* $S_c = \left\{ (x,y,z) \middle| z \in \left[\frac{i_c}{2^d}, \frac{i_{c+1}}{2^d} \right] \right\}$, with $0 \leq c < C$ and $0 = i_0 < i_1 < \cdots < i_c < \cdots < i_C = 2^d$, such that the number of points falling within each slab is balanced. Ultimately, client c is responsible for reconstructing the surface $\mathcal{S}_c$ within its own slab, and we require that successive surfaces are continuous across the *slice* corresponding to their shared boundary plane:

$$\mathcal{S}_c \cap \left\{ (x,y,z) \middle| z = \frac{i_{c+1}}{2^d} \right\} = \mathcal{S}_{c+1} \cap \left\{ (x,y,z) \middle| z = \frac{i_{c+1}}{2^d} \right\}.$$

All large results in this paper use a coarse octree of depth $d = 8$. Thus, there are 256 regular z intervals. Figure 2 shows an adaptive partition of these intervals into $C = 64$ slabs.

4.2. Connected isosurface

The implicit functions generated by each pair of successive clients may not match precisely along their shared boundary slice. If iso-

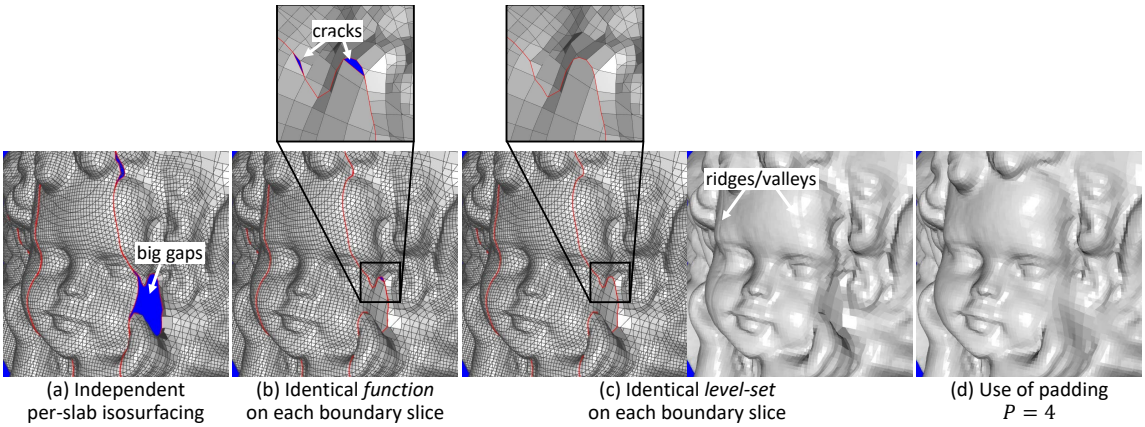


Figure 3: *Challenges in obtaining a connected, seamless surface using a computation distributed over slabs (shown in red): (a) Extracting isosurfaces independently per slab leaves large gaps between reconstructed mesh parts; (b) Defining a shared 2D slice function corrects the gaps but leaves topological cracks (where the octree is refined to different levels on either side of the slab boundary); (c) Defining shared iso-vertices and iso-edges guarantees a connected surface mesh, but which may still exhibit ridge artifacts along slab boundaries; (d) Padding the slab extents to include nearby points improves the consistency of the implicit functions, leading to a seamless reconstruction.*

surfaces are computed independently within each client, large gaps become evident (Figure 3a). We present a two-part approach that guarantees a watertight connection.

(1) Consistent function values The values of the two implicit functions meeting at a slice must agree on the slice. Conveniently, because isosurfacing samples functions discretely, it is unnecessary for the 3D function to be continuous. Our strategy is to define a single 2D function over the slice and use it to *override* the 3D implicit function solely on the slice plane.

We achieve this by noting that the restriction of each client's implicit function to the slice can be represented using bivariate B-splines defined over a quadtree. Fusing the quadtrees defined by the two clients (i.e. computing the coarsest quadtree containing the two quadtrees as subtrees) and averaging the B-spline coefficients, we obtain a single 2D function defined on the shared slice that is close to the restrictions of the two 3D implicit functions to that slice. Figure 4 shows the fused quadtree and implicit function for points non-uniformly sampled from a cylinder and Figure 3b shows the resulting improvement.

(2) Consistent isocurves The shared quadtree and B-spline coefficients ensure a common function on the slice, but we also need the two clients to sample the function in the same way. If the clients only restrict their octrees to the shared slice, they may obtain different quadtrees, which results in mesh cracks (Figure 3b).

We address this by performing level-set extraction over the 2D slice (using the shared quadtree). In **Iso.2**, for each octree face that lies in the slice plane, we replace the iso-edges that would have been computed from the octree cell by the iso-edges computed directly on the quadtree faces. As shown in Figure 3c, the reconstructed surface parts are now guaranteed to be connected.

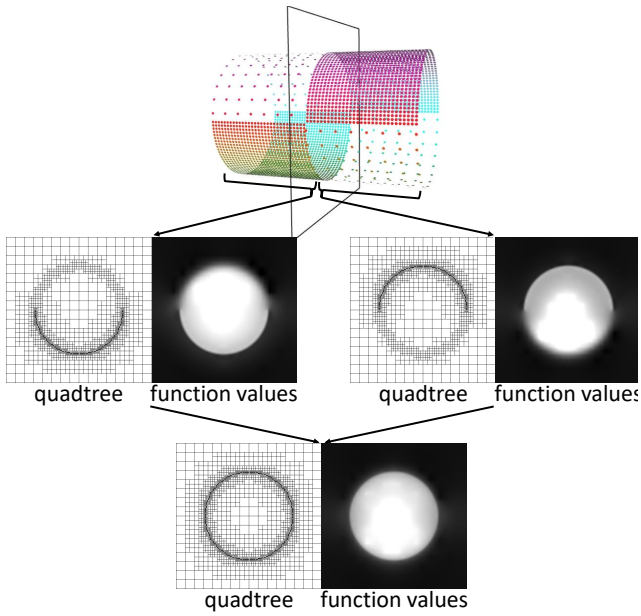


Figure 4: *Non-uniform point sampling of a cylinder, with a two-slab partition (top): The different sampling densities on either side of the shared boundary slice results in the clients defining different quadtrees and different functions on the slice (middle). Fusing defines a single, consistent quadtree and function (bottom).*

4.3. Accurate implicit function

Even with guaranteed connectedness of the isosurfaces, the resulting surface often exhibits ridge artifacts along the slab boundaries, which are especially evident with flat shading (Figure 3c). The challenge is that the Poisson reconstruction is a global problem, and attempting to solve it over distributed clients by disjointly par-

titioning the points leads to implicit functions that fail to agree near the slices. In particular, points within a slab S_c should affect the value of the implicit function not just in S_c but, at least in principle, in all slabs. We address this in two ways:

1. We leverage the hierarchy to separately solve the low- and high-resolution problems. The low-resolution problem is solved over a single octree of depth d by the server. The high-resolution problem is distributed across the C clients.
2. For each client slab S_c, we extend the set of considered points to include those within a padded range of the slab, $\left[\frac{i_c-P}{2^d}, \frac{i_{c+1}+P}{2^d}\right]$, where P denotes the padding size (see Figure 1(b)).

The key idea is that while a point outside slab S_c affects the reconstructed implicit function within the slab, its contribution to the implicit function becomes smoother the further it lies from the slab, and hence its long-distance effect is well-captured by the low-resolution solution. Thus we choose a padding size $P > 0$ to enable capturing the high-frequency effects of points sufficiently close to the slab, and use the lower-resolution global solve to account for points further away.

Figures 3c and 3d compare reconstructions of the "Angel" dataset obtained without padding ($P = 0$) and with padding ($P = 4$), showing that the surface becomes seamless when a small amount of padding is introduced. Experimental results in Section 6 indicate that the root mean squared error introduced due to the distributed computation is significantly smaller than the width of a leaf node.

5. Implementation

Our implementation follows a server-client architecture, in which clients communicate only with the server. We begin by describing the distributed preprocessing step that partitions the samples into 2^d volume intervals. Next, we describe the implementation of the reconstruction algorithm that takes the partitioned samples and outputs a surface for each slab. Finally, we describe the postprocessing step that fuses the surfaces generated by the clients. We discuss the differences between our implementation and that of [BKBH09] and conclude by discussing implications for serial reconstruction.

5.1. Preprocessing

We partition the input samples into volume intervals, transforming a single input file into 2^d files. We assume that the size of an input sample on disk is fixed (e.g., binary PLY format) so that the cost of seeking to the position of a sample within the input file is constant, and that the clients have a shared file system so that a file written by one client can be read by another. The preprocessing consists of six successive steps, alternating between serial processing by the server and parallel distributed processing by the clients. It requires three reading passes through the samples and one writing pass. Synchronization is necessary only between successive phases, i.e., in the form of a barrier.

Pre.1

The server determines the number of points in the input file and assigns each client a subsequence of points for processing. Concretely, if there are N points and C clients, the subsequence associated to each client c is $N_c = [n_c, n_{c+1})$, with $n_c = \lfloor N \cdot c/C \rfloor$.

Synchronization: The server sends each client c the subsequence range N_c.

Pre.2

Each client c computes the extent of the points in the subsequence N_c along several directions. (We use 9 directions. The first three correspond to the x, y, and z axes and the remainder correspond to the two diagonals in each of the xy, xz, and yz planes. These nine directions have the property that any single direction can be completed to a triplet of directions forming an orthonormal frame.)

Synchronization: Each client c sends the extents of the points in subsequence N_c along each of the directions to the server.

Pre.3

The server consolidates the extents, computes the direction with maximal extent, completes that direction to an orthonormal frame, and gets the similarity transformation taking the samples' bounding box into the unit cube. (The transformation is chosen so that the direction with maximal extent is mapped to the z axis.)

Synchronization: The transformation from the oriented bounding cube to the unit cube is sent to each of the clients.

Pre.4

Each client c reads the subset of samples in subsequence N_c, applies the similarity transformation to each sample, and writes the transformed sample to one of 2^d volume interval files. (A sample with transformed position (x,y,z) is mapped to the $\lfloor z \cdot 2^d \rfloor$-th file.)

Synchronization: Each client c sends back the number of samples from the subsequence I_c falling into each of the 2^d volume intervals.

Pre.5

The server accumulates the number of samples to get the total number of samples falling within each volume interval. It then solves a dynamic programming problem to partition the 2^d volume intervals into C slabs such that the number of samples falling in each slab is as balanced as possible. Specifically, given a partitioning π of intervals to slabs, we define the cost $\mathcal{C}_\pi$ of the partition to be the vector of the point counts falling within each slab, sorted from largest to smallest. We then solve a dynamic programming problem to find the "smallest" partition, where for two partitions π' and π'' we define $\mathcal{C}_{\pi'} \leq \mathcal{C}_{\pi''}$ using lexicographic ordering. (The smallest partition minimizes the number of points falling in the largest slab. And, of all such partitions, it minimizes the number of points falling in the second largest slab. etc.)

Synchronization: The server sends each client c the range of the c-th interval of the partition, $\{i_c, i_{c+1}\}$.

Pre.6

Each client c considers the volume interval in the range $[i_c, i_{c+1})$ and consolidates the C volume interval files generated by the different clients in step **Pre.4** into a single volume interval file.

Note An alternate implementation of step **Pre.2** could compute the extent along the principal axes of the samples' covariance matrix, so that the streaming direction aligns with the direction of maximal covariance, as in [BKBH07]. The drawback of such an approach is that it requires two passes over the samples – one to compute the covariance matrix and a second to compute the extent.

5.2. Reconstruction

The reconstruction algorithm has seven successive phases, alternating between serial server processing and parallel client processing. The phases are separated by synchronization barriers.

Recon.1 [sPSR.1-3]

Each client c considers the partition interval $\{i_c, i_{c+1}\}$ from step **Pre.4**. Client c reads in the points falling within the interval $\left[\frac{i_c - P}{2^d}, \frac{i_{c+1}+P}{2^d}\right]$, constructs the octree, computes the density estimator, constructs the vector field, and accumulates the sum of sample weights. We modify the implementation of sPSR in two ways.

First, to reduce the overhead of the padding points on the size of the system, we observe that an oriented point (p,n) in the padding region only needs to be introduced at the finest depth D if its z coordinate p_z is close to the range $\left[\frac{i_c}{2^d}, \frac{i_{c+1}}{2^d}\right]$ in units of 2^{-D} (the width of a node at depth D). Thus, to each padding sample (p,n) we associate the finest depth δ_p at which the neighborhood of p still intersects the slab:

$$\left[p_z - \frac{P}{2^{\delta_p}}, p_z + \frac{P}{2^{\delta_p}}\right] \cap S_c \neq \emptyset.$$

Then, when constructing the octree, computing the density estimator, and constructing the vector field, we insert a sample in the padding region at a depth no larger than δ_p.

Second, to ensure that the constraints $\mathbf{b}$ shared with the server are identical to those that would have been constructed if a single client were used, we separately compute the vector field $\vec{V}^{int}$, generated by samples in the interior of the slab, and the vector field $\vec{V}^{pad}$, generated by samples in the padding region. For a similar reason, when computing the sum of sample weights we consider only the contributions of interior points.

Synchronization: Each client sends its sum of sample weights to the server.

Recon.2 [sPSR.4]

The server accumulates the sum of sample weights from the clients and computes the screening weight α.

Synchronization: The server sends the screening weight α back to the clients.

Recon.3 [sPSR.5]

Each client constructs the constraint vector $\mathbf{b}$ using the screening weight α, the vector fields $\vec{V}^{int}, \vec{V}^{pad}$, and the weighted samples. As in step **Recon.1**, the right-hand-side is computed separately as $\mathbf{b}^{int}$ using $\vec{V}^{int}$ and the interior samples, and as $\mathbf{b}^{pad}$ using $\vec{V}^{pad}$ and the padding samples.

Synchronization: Each client sends its low-resolution subtree and associated subvector of $\mathbf{b}^{int}$ (i.e. the subset of the tree and associated coefficients, for nodes at depth no greater than d) to the server, as well as the average positions and counts of the samples sitting over nodes at depth d (required by the server to construct the coefficients of the system matrix associated with the screening).

Recon.4 [sPSR.6]

The server merges the trees and accumulates the constraints from the different clients, as well as the average of the samples. It then constructs and solves the linear system to obtain the low-resolution solution $\mathbf{x}$.

Synchronization: For each client c, the server extracts the subtree and subvector of $\mathbf{x}$ corresponding to nodes whose z coefficients are in the range $\left[\frac{i_c - P}{2^d}, \frac{i_{c+1}+P}{2^d}\right]$ and sends those to the client.

Recon.5 [sPSR.6-7]

Each client begins by growing its octree so as to contain the server's subtree. (This is necessary if a neighboring client was more refined near the slice.) Next, each client combines the interior and padding constraints, $\mathbf{b}^{int}$ and $\mathbf{b}^{pad}$, and using the low-resolution solution obtained from the server continues to solve the linear system starting at depth $d+1$ and continuing to depth D. Then, the client computes the sum of the values of the implicit function at the samples interior to the slab.

Synchronization: To ensure a connected reconstructed mesh, each client communicates to the server the restriction of its implicit function on its two boundary slices. Using the B-spline structure, each of these implicit functions is represented by a quadtree with bivariate B-spline coefficients associated with the nodes. (As non-linear interpolation is used to determine the position of isovertices [FKG15], we also send the B-spline coefficients encoding the partial derivatives of the implicit function along the z direction.) The client also sends the sum of the values of the implicit function at the interior samples.

Recon.6 [sPSR.7]

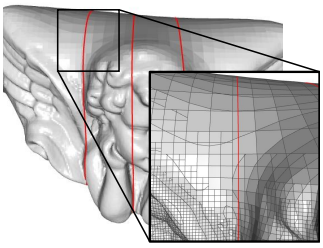
The server averages the values of the implicit functions to get the isovalue x. Then, for each boundary slice between successive clients $c-1$ and c, the server merges the quadtrees, averages the coefficients generated by the two clients to the plane $z = \frac{i_c}{2^d}$, and computes the level-set curve at x.

Synchronization: For every slice s, the server sends the merged quadtrees and averaged coefficients to clients $s-1$ and s. It also sends the level-set curves computed over each quadtree to ensure that the surfaces $\mathcal{S}_{s-1}$ and $\mathcal{S}_s$ connect along shared vertices and edges, even when the octrees are refined differently at the slice. The server also sends the isovalue x to each client.

Recon.7 [sPSR.8, Iso.0-2]

Using the solution computed in step **Recon.5**, each client extracts the isosurface S_c with isovalue x. The isosurface extraction of [KKDH07] is modified in three ways. (1) When considering octree corners on the boundaries of the slab, values are assigned using the merged quadtree and averaged coefficients received from

the server. (2) When computing iso-vertices along leaf node edges that lie on the boundary, we use the iso-vertices received from the server. (3) When computing iso-edges along leaf node faces that lie on the boundary, we use the iso-edges received from the server.



As we do not require the coarse tree to be complete, it is possible for client leaf nodes to straddle the boundary slices. In this case we "trim" the leaf nodes to the boundary slices. The values at slice corners (the intersection of the leaf's edges with the boundary slices) as well as iso-vertices on slice edges (the intersections of the leaf's faces with the boundary slices) and iso-edges lying on slice faces (the intersection of the leaf's volume with the boundary slices) are obtained from the merged quadtrees in step **Recon.6**. The inset shows an example for the reconstruction of the "Angel" model, with the slab boundaries highlighted in red. (Note that the trimming effectively splits the iso-surface associated to the leaf node, and can result in skinny faces.)

An alternative approach would be to have the server compute the subset of the iso-surface defined by leaves straddling the boundary slices. This would avoid the introduction of small faces, but would make fusing (described next) more complicated as shared vertices would no longer be at the beginning/end of the vertex list.

5.3. Postprocessing

Isosurface extraction is performed in each client by streaming across the z axis. For each client c, vertices along slice $z = \frac{i_c}{2^d}$ are the first ones created and vertices along slice $z = \frac{i_{c+1}}{2^d}$ are the last. Moreover, each slice's vertices have a consistent ordering in the two adjacent clients. Thus, topologically merging the vertices from the successive clients' reconstructions is straightforward as it only requires knowing the count of the slice vertices – a value already determined by the server when it explicitly constructs the level-set curve in step **Recon.6**.

5.4. Relation to [BKBH09]

As discussed in Section 2 our approach differs from the earlier method of [BKBH09] in several ways.

Complexity In that approach, every client constructs and solves the low-resolution system. While the complexity of solving is negligible (since it is done over a coarse grid), the generation of the system constraints requires having each client stream over *all* of the points. Thus, for an input of N samples, the runtime complexity will be at least $O(N)$. In our approach, the construction of the low-resolution system (steps **Recon.1–4**) is designed such that each client only needs process $O(N/C)$ points.

Coarse octree depth The earlier approach requires the coarse octree to be complete. This limits the maximum value of d – the depth of the coarse octree. That, in turn restricts the total number of clients and the granularity at which the workload is partitioned among clients. Our approach uses an adapted coarse octree, avoiding these limitations.

Communication The earlier approach has clients synchronize high-resolution constraint and solution coefficients in a padding region near the slab boundaries. In our approach, we only share the restricted quadtrees between neighboring clients (between steps **Recon.5** and **Recon.6** and between steps **Recon.6** and **Recon.7**).

Surface continuity By having the geometry of the surface at the slab boundary computed only once (step **Recon.6**) we guarantee that the level-set curves for the two surfaces extracted on either side of a slab boundary are identical, independent of machine precision.

5.5. Implications for serial reconstruction

The client-server design of our distributed solver involves only a small number of synchronization barriers. Consequently, in addition to providing an efficient distributed solution, our implementation trivially enables a serial realization of the traditional sPSR algorithm with a significantly lower memory footprint. Specifically, we can run the clients serially on a single machine by storing state to disk when transitioning between clients. (In comparison, the streaming implementation [BKBH07], while having smaller peak memory utilization, is markedly more challenging to implement.)

6. Evaluation

We evaluate our approach by considering the quality of the generated reconstructions, the empirical and practical efficiency of the algorithm, and the effects of solving a dynamic programming problem on client workload. Open-source code implementing the distributed reconstruction is available at https://github.com/mkazhdan/PoissonRecon.

6.1. Reconstruction quality

The slab padding introduced in Section 4.3 improves the accuracy of the distributed solver, avoiding reconstruction artifacts along slab boundaries. To quantify this improvement, we consider two related questions.

(1) What is the discrepancy between the implicit functions generated by the individual clients on the boundary slice shared by successive clients? This discrepancy is measured by comparing the restrictions of the implicit functions computed by each of the clients to the boundary slices.

(2) How different is the DPSR reconstruction obtained with distributed clients (which each see only a subset of the points) from a reference sPSR reconstruction obtained with a single client (which sees all points)? We compare both the implicit (3D) functions and the extracted level-set polygon meshes.

Table 1 shows the L_2 difference between the restrictions of clients' implicit functions to the slices, the L_2 difference between the implicit functions computed using a single client and the implicit functions obtained using $C = 4$ clients, and the root mean

Angel	$P=0$	$P=2$	$P=4$	$P=6$	$P=8$
Slice	$3.8\cdot 10^{-2}$	$4.1\cdot 10^{-3}$	$1.7\cdot 10^{-3}$	$8.6\cdot 10^{-4}$	$4.5\cdot 10^{-4}$
Implicit	$1.6\cdot 10^{-2}$	$9.8\cdot 10^{-4}$	$3.9\cdot 10^{-4}$	$1.9\cdot 10^{-4}$	$1.2\cdot 10^{-4}$
Geometric	$1.9\cdot 10^{-3}$	$3.8\cdot 10^{-5}$	$2.1\cdot 10^{-5}$	$9.0\cdot 10^{-6}$	$6.0\cdot 10^{-6}$

David Head	$P=0$	$P=2$	$P=4$	$P=6$	$P=8$
Slice	$9.1\cdot 10^{-2}$	$1.2\cdot 10^{-2}$	$8.4\cdot 10^{-3}$	$4.2\cdot 10^{-3}$	$3.6\cdot 10^{-3}$
Implicit	$3.6\cdot 10^{-2}$	$4.0\cdot 10^{-3}$	$2.4\cdot 10^{-3}$	$1.3\cdot 10^{-3}$	$9.2\cdot 10^{-4}$
Geometric	$1.6\cdot 10^{-4}$	$8.0\cdot 10^{-6}$	$4.0\cdot 10^{-6}$	$2.0\cdot 10^{-6}$	$2.0\cdot 10^{-6}$

Table 1: *Slice difference (L_2), implicit function difference (L_2), and geometric difference (RMS) for reconstructions with different padding sizes P. "Angel" is obtained at depth $D=8$ with coarse depth $d=5$ and $C=4$ clients. "David Head" is obtained at depth $D=10$ with coarse depth $d=5$ and $C=4$ clients.*

squared distance (in units of bounding box width) between the vertices on the surfaces reconstructed using a single client and the surfaces reconstructed using $C=4$ clients for the "Angel" (24K points) and "David Head" (215M points) datasets.

Examining the table, we see the expected trend of improved reconstruction quality with increased padding size P. Examining rows "Slice" and "Implicit" We also find that for a padding size of $P=4$, the differences are on the order of 10^{-3}. We believe this is negligibly small as the implicit function is approximately an indicator function taking on values in the range $[0,1]$ and in regions of fine sampling density we expect the gradient of the implicit function to be large near the surface. Thus a change on the order of 10^{-3} should correspond to an imperceptible difference in the isosurface. This is corroborated in the "Geometric" rows of the table which show that using a padding size of $P=4$, the root mean squared distance is on the order of 10^{-5}.

We also visualize the distances between the reconstructions obtained using a single client and reconstructions obtained using $C=4$ clients in Figure 5, for padding sizes of $P=0$ and $P=4$. With a padding size of $P=0$ there are significant errors at the slices between clients. In contrast reconstructions with a padding size of $P=4$ only exhibit error in regions of low sampling density.

In all subsequent evaluations, we fix the padding size at $P=4$.

Note We observe that the choice $P=4$ matches the back-of-the-envelope calculation for a "reasonable" padding size, given the finite-elements discretization of the PDE (i.e. and is independent of the point sampling). Concretely, in choosing P we would like to ensure that the reconstruction generated by multiple clients is close to the reconstruction that would be generated by a single client.

At a minimum, this requires that the right-hand-side $\mathbf{b}^{int}$ generated by client c matches the restriction to slab S_c of the right-hand-side generated by sPSR. In the implementation of sPSR, an oriented sample falling into octree node $o\in\mathcal{O}$ contributes to the coefficients of the vector field $\vec{V}$ at the corners of the nodes in the one-ring neighborhood of o. Furthermore, since the right-hand-side constraints are computed by integrating the vector field against the gradients of first-order B-splines centered at octree corners, an oriented sample will affect the value of $\mathbf{b}^{int}$ if and only if it is within a distance of $P=2$ of slab S_c.

We further extend the padding size because solving the Poisson equation diffuses the constraints into neighboring nodes – a process that can be realized as a convolution with the Green's function. Formally, the Green's function is not compactly supported so the constraint at one node affects the solution everywhere. However, as the long-range effects of the convolution are low-frequency, we find that they are well captured by the server's solution, and extending to a padding size of $P=4$ works well in practice, regardless of how the points are sampled.

6.2. Theoretical complexity analysis

We recall that our implementation is parameterized by the depth of the reconstruction (D), the depth of the coarse octree used by the server (d), the number of clients (C), and the padding size (P).

Redundancy We measure the redundancy in terms of the additional octree nodes introduced by points in the padding region. This reflects the computational overhead of setting up and solving the system, as well as extracting the isosurface. As derived in Appendix A, the expected redundancy overhead due to padding is:

$$R_N(P)=\frac{2\cdot P\cdot C}{2^d}$$

for the naive implementation (in which padding samples are introduced up to the finest depth), and

$$R_A(P)=\frac{3\cdot P\cdot C}{2^D}$$

for the adaptive implementation (in which the depth at which a padding sample is introduced is determined by its proximity to the slab). We validate these estimates by measuring the fraction of additional octree nodes introduced by points in the padding region. This reflects the added overhead of setting up and solving the system, as well as the overhead of extracting the isosurface.

Distributed complexity For the distributed implementation of the Poisson Surface Reconstruction we expect the runtime of the client (resp. server) to scale as $O(4^D\cdot R/C)$ (resp. $O(4^d+2^D\cdot C)$), with $O(4^d)$ the complexity of the coarse octree, $O(4^D)$ the complexity of the fine octree, $O(2^D)$ the complexity of the quadtree storing the shared solution on a slice, and R the redundancy factor. Similarly, we expect the peak memory usage of the client to scale as $O(4^D\cdot R/C)$ and that of the server to scale as $O(4^d+2^D)$.

Serialized distributed complexity Similar to the distributed implementation, the expected peak memory of the client (resp. server) is $O(4^D\cdot R/C)$ (resp. $O(4^d+2^D\cdot C)$). However, in this case the expected running time of the client is $O(4^D\cdot R)$.

6.3. Empirical complexity results

Redundancy Table 2 compares the measured redundancy factors when reconstructing the "Angel" (24K points) and "David Head" (215M points) datasets for different padding sizes P. Reconstruction depth D, coarse octree depth d, and number of clients C are chosen so that the expected redundancy for the two datasets are the same. As expected, for both approaches the redundancy factor

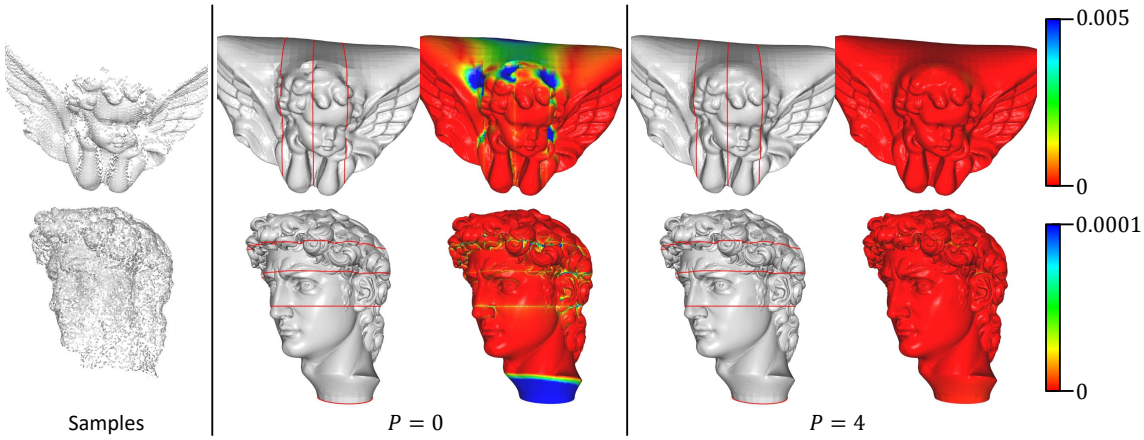


Figure 5: *Visualizations of the closest distance from the surfaces reconstructed from the "Angel" and "David Head" datasets (left) with $C = 4$ clients to surface reconstructed with a single client, using padding sizes of $P = 0$ (center) and $P = 4$ (right). Using a padding size of $P = 0$ there are noticeable error at the client boundaries. These go away With a padding size of $P = 4$.*

	Naive					Adaptive				
$P =$	0	2	4	6	8	0	2	4	6	8
Expected	0.0	0.5	1.0	1.5	2.0	0.00	0.09	0.19	0.28	0.38
Angel	0.0	0.2	0.4	0.6	0.8	0.00	0.04	0.08	0.12	0.15
David Head	0.0	0.5	1.1	1.6	2.2	0.00	0.10	0.20	0.30	0.40

Table 2: *Comparison of the expected and measured redundancy factors when reconstructing two datasets, as a function of padding size P. The table compares the Naive implementation (in which all points in the padded region are introduced at the finest depth) and our Adaptive implementation (in which the depth at which a point is introduced is chosen based on its proximity to the slab interior). "Angel" uses $C = 4$ clients, reconstruction depth $D = 8$, and coarse depth $d = 5$; "David Head" uses $C = 16$, $D = 10$, and $d = 7$.*

grows linearly with P, with the naive approach incurring a significant cost for larger padding values. In contrast, the overhead for the adaptive approach remains small even for larger values of P.

We note that the redundancy factor does not consider the I/O cost of padding, which is the same for both the naive and adaptive implementations: $\frac{2 \cdot P \cdot C}{2^d}$.

Distributed complexity Table 3 shows the running time and peak memory usage when reconstructing the full "David" point-set (roughly one billion points) and the "Safra Square" point-set (roughly 2.5 billion points) as a function of reconstruction depth D and number of clients C. (For the "David" datasets, which exhibits misalignment error, we disabled screening. For the "Safra Square" we used the default screening weight.) Reconstructions were run on a distributed cluster using a SLURM workload manager.

Visualizations of the reconstructed "David" at depth $D = 15$ and "Safra Square" at depth $D = 16$ can be seen in Figures 2 and 6. The figures show zoom-ins on the partition boundaries, demonstrating the seamlessness of both the geometry and color reconstructions.

As expected we see a reciprocal relation between the number of clients and the peak memory usage and an (almost) reciprocal relation between the number of clients and the running time. We also see that for a given number of clients, increasing the depth increases the running time and peak memory usage by a factor of four. (This model starts to fail at higher depths, as sPSR assigns a point's depth dynamically, only refining the octree where the sampling density is high enough.)

While we would hope that doubling the number of clients would reduce the running time and peak memory usage by a factor of two, the tables show that this does not hold for larger values of C. This is because for fixed values of coarse octree depth d, larger numbers of clients make it more difficult to load balance the workload. (In the limit, as $C = 2^d$, each client is assigned a single interval, reducing the workload distribution to a regular spatial partition.)

The tables also show the runtime for the preprocessing step. (The peak memory usage is negligible since the samples are not stored in memory.) As expected, we see that empirical performance matches the $O(N/C)$ complexity, particularly for smaller C where load balancing is easier. And, importantly, we see that the distribution of the preprocessing ensures that the cost of partitioning remains negligible, relative to the cost of reconstruction.

Serialized complexity Table 4 compares the running times and peak memory usage of the serialized distributed implementation with the traditional sPSR on the "David" point-set. As the table shows, the serialized distributed computation incurs a running time overhead of 30–50%, but its required memory is an order of magnitude smaller, enabling a reconstruction at depth $D = 15$ within a 64GB memory budget.

Comparing the performance of the serialized implementation to the performance of the parallel implementation, we note that the serialized implementation is slower (compare to Table 3: row $D = 12$, column $C = 1$, left), likely due to the additional disk I/O, and requires more memory (compare to Table 3: column $C = 64$, left).

We stress that while our distributed implementation makes it

		David							Safra Square						
		$C=1$	$C=2$	$C=4$	$C=8$	$C=16$	$C=32$	$C=64$	$C=1$	$C=2$	$C=4$	$C=8$	$C=16$	$C=32$	$C=64$
Preprocess time (s)		458	231	112	65	41	28	28	1383	729	376	238	127	72	60
$D=12$	Time (s)	433	308	174	94	56	47	48	536	318	184	108	77	59	60
	Memory (GB)	17	10	6	3	2	1	1	11	8	5	3	1	1	1
$D=13$	Time (s)	-	1190	572	343	208	146	106	1254	776	461	263	162	126	113
	Memory (GB)	-	46	27	15	9	5	3	41	27	17	10	6	4	3
$D=14$	Time (s)	-	-	-	1955	893	641	410	-	-	1472	814	479	307	247
	Memory (GB)	-	-	-	71	45	22	14	-	-	59	34	21	14	9
$D=15$	Time (s)	-	-	-	-	-	-	1187	-	-	-	-	1536	856	619
	Memory (GB)	-	-	-	-	-	-	44	-	-	-	-	66	41	26
$D=16$	Time (s)	-	-	-	-	-	-	-	-	-	-	-	-	-	1301
	Memory (GB)	-	-	-	-	-	-	-	-	-	-	-	-	-	58

Table 3: *Performance and output complexity for the "David" and "Safra Square" datasets, (with 1,005,395,738 and 2,364,268,059 points, respectively), for different numbers of clients and at different depths. We set the coarse octree depth to $d=8$. For the "David" reconstruction, the output mesh has 13, 54, 216, and 727 million vertices, respectively. For the "Safra Square" reconstructions, the output mesh has 12, 48, 171, 554, and 1610 million vertices, respectively.*

		Traditional sPSR	Serial DPSR
$D=12$	Time (s)	764	985
	Memory (GB)	12	2
$D=13$	Time (s)	2211	3234
	Memory (GB)	58	6
$D=14$	Time (s)	-	13,728
	Memory (GB)	-	20
$D=15$	Time (s)	-	48,962
	Memory (GB)	-	56

Table 4: *Performance comparison for the traditional reconstruction and the serial implementation of the distributed reconstruction using the "David" dataset at different depths. For the serial implementation we set $C=64$ and $d=8$.*

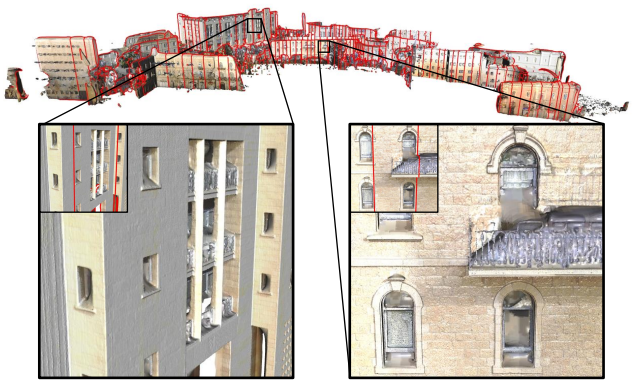


Figure 6: *Visualizations of the "Safra Square" reconstruction at depth $D=16$ using $C=64$ clients. Boundary slices are drawn in red. Parts of the reconstruction generated in regions of low sampling density are trimmed off.*

easy to obtain a low-memory serial implementation, the streaming implementation in [BKBH07] has significantly lower memory.

6.4. Benefits of load balancing

Our implementation uses dynamic programming to assign point intervals to clients so as to make the distribution of work as balanced as possible. The slabs are visualized in Figure 2 and 6 which show the reconstructed models with boundary edges drawn in red.

Tables 5 and 6 show the effects of using dynamic programming for load balancing on the clients' memory utilization when reconstructing the "David" model. Table 5 compares the performance of regular partitioning to load balancing for varying number of clients C, while Table 6 shows the effects of increased coarse octree depth on memory utilization for varying coarse octree depth d. (Recall that the the input point set is partitioned into 2^d intervals, so that larger values of d allow for more fine-grained load balancing.)

Since the complexity of sPSR is linear, the average memory utilization remains roughly the same, regardless of whether "regular" or "balanced" partitioning is used and regardless of the depth of the coarse octree depth. Examining the maximum memory utilization we see that load balancing improves performance, reducing the peak memory utilization by $12-65\%$. We also find that increasing the coarse octree depth reduces the peak memory utilization by an additional 10%, 17%, and 6% respectively.

We make two observations about load balancing: First, while increasing d allows for finer load balancing, it comes at the cost of having the server solve a finer system. For $d \in \{8,9,10,11\}$, the respective running times were 369, 355, 376, and 425 seconds. Second, the distribution of points to clients is only a prediction of the amount of work each client needs to do, and this prediction becomes less precise as the point sampling is becomes non-uniform. This is reflected in the $d=11$ column of Table 6 which shows larger peak memory utilization than $d=9$ or $d=10$, despite the smaller peak number of samples per slab.

	$C=8$		$C=16$		$C=32$		$C=64$	
	Reg.	Bal.	Reg.	Bal.	Reg.	Bal.	Reg.	Bal.
Average	55.5	56.2	29.0	29.2	14.9	15.3	8.0	8.2
Max	81.3	70.8	50.4	44.8	36.8	22.2	20.2	14.2

Table 5: *Per-client memory usage (in gigabytes) with a regular slab partition and with adaptive load-balancing, on the "David'' dataset reconstructed at depth $D = 14$ with coarse tree depth $d = 8$, and using $C \in \{8, 16, 32, 64\}$ clients.*

	$d=8$	$d=9$	$d=10$	$d=11$
Average Mem.	8.2	7.9	7.7	7.6
Max Mem.	14.2	12.9	12.2	13.4
Max Samples	18.6	17.0	16.6	15.9

Table 6: *Per-client memory usage (in gigabytes) and maximum number of samples within a slab (in millions) on the "David'' dataset reconstructed at depth $D = 14$ with $C = 64$ clients, and using coarse tree depth $d \in \{8, 9, 10, 11\}$,*

7. Conclusion

We presented a novel implementation of the screened Poisson surface reconstruction algorithm that allows the reconstruction to be distributed across multiple clients using a client-server system, with only a small number of synchronization barriers. By decomposing the solution of the linear system into low-frequency (global/server) and high-frequency (local/client) components, leveraging padding, and enforcing a connected isosurface, we obtain a solution that exhibits no artifacts at client boundaries and is indistinguishable from the single-client solution.

This distributed implementation can leverage modern hardware to reconstruct surfaces from huge (e.g. billion-point-scale) datasets with running time and memory usage that are an order of magnitude smaller than those required by the traditional screened Poisson surface reconstruction implementation. In addition, the small number of synchronization barriers lets us reduce the peak memory usage for a single-client machine, enabling the reconstruction of surfaces at higher depth (i.e. more precisely).

In the future, we would like to explore several extensions of our approach. We would like to incorporate Streaming Poisson Surface Reconstruction [BKBH07] within our client/server model to further reduce the peak memory utilization. We would like to consider approaches for decoupling the granularity of point-partitioning from the coarse octree depth. And we would like to incorporate mesh compression, as in the works of Schertler *et al*. [STJ*17] and Maggiordomo *et al*. [MMT23], to reduce the size of the output generated by the clients.

Acknowledgmenets

The authors would like to thank the Digital Michelangelo Project [DMP] and Resonai [Res] for generously sharing their data.

References

[ABCO*01] ALEXA M., BEHR J., COHEN-OR D., FLEISHMAN S., LEVIN D., SILVA C.: Point set surfaces. In *Proceedings of the Conference on Visualization '01* (2001).

[ABK98] AMENTA N., BERN M., KAMVYSSELIS M.: A new voronoi-based surface reconstruction algorithm. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), pp. 415–421.

[AGL06] AHN M., GUSKOV I., LEE S.: Out-of-core remeshing of large polygonal meshes. *IEEE transactions on visualization and computer graphics 12*, 5 (2006), 1221–1228.

[BGKS20] BADKI A., GALLO O., KAUTZ J., SEN P.: Meshlet priors for 3d mesh reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2020), pp. 2849–2858.

[BKBH07] BOLITHO M., KAZHDAN M., BURNS R., HOPPE H.: Multilevel streaming for out-of-core surface reconstruction. In *Symposium on geometry processing* (2007), pp. 69–78.

[BKBH09] BOLITHO M., KAZHDAN M., BURNS R., HOPPE H.: Parallel poisson surface reconstruction. In *Advances in Visual Computing* (Berlin, Heidelberg, 2009), pp. 678–689.

[BMR*99] BERNARDINI F., MITTLEMAN J., RUSHMEIER H., SILVA C., TAUBIN G.: The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics 5* (1999).

[CBI13] CHEN J., BAUTEMBACH D., IZADI S.: Scalable real-time volumetric surface reconstruction. *ACM Transactions on Graphics (ToG) 32*, 4 (2013), 1–16.

[CL96a] CURLESS B., LEVOY M.: A volumetric method for building complex models from range images. *Computer Graphics (Proceedings of SIGGRAPH 96)* (1996).

[CL96b] CURLESS B., LEVOY M.: A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), pp. 303–312.

[CT11] CALAKLI F., TAUBIN G.: Ssd: Smooth signed distance surface reconstruction. *Computer Graphics Forum 30*, 7 (2011), 1993–2002.

[CTFZ22] CHEN Z., TAGLIASACCHI A., FUNKHOUSER T., ZHANG H.: Neural dual contouring. *ACM Transactions on Graphics (TOG) 41*, 4 (2022), 1–13.

[DMP] The digital Michelangelo project. `https://graphics.stanford.edu/papers/digmich_falletti/`.

[FG14] FUHRMANN S., GOESELE M.: Floating scale surface reconstruction. *ACM Transactions on Graphics 33*, 4 (2014), 1–11.

[FKG15] FUHRMANN S., KAZHDAN M., GOESELE M.: Accurate isosurface interpolation with hermite data. In *2015 International Conference on 3D Vision* (2015), pp. 256–263.

[HDD*92] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Surface reconstruction from unorganized points. *Computer Graphics 26* (1992).

[IG03] ISENBURG M., GUMHOLD S.: Out-of-core compression for gigantic polygon meshes. *ACM Transactions on Graphics 22*, 3 (2003), 935–942.

[IL05] ISENBURG M., LINDSTROM P.: Streaming meshes. In *IEEE Visualization* (2005), IEEE, pp. 231–238.

[ILS05] ISENBURG M., LINDSTROM P., SNOEYINK J.: Streaming compression of triangle meshes. In *Symposium on Geometry Processing* (2005).

[ILSS06] ISENBURG M., LIU Y., SHEWCHUK J., SNOEYINK J.: Streaming computation of delaunay triangulations. *ACM Transactions on Graphics 25*, 3 (2006), 1049–1056.

[KBH06] KAZHDAN M., BOLITHO M., HOPPE H.: Poisson surface reconstruction. *Symposium on Geometry processing* (2006), 61–70.

[KH13] KAZHDAN M., HOPPE H.: Screened Poisson surface reconstruction. *ACM Transactions on Graphics 32*, 3 (2013), 1–13.

[KKDH07] KAZHDAN M., KLEIN A., DALAL K., HOPPE H.: Unconstrained isosurface extraction on arbitrary octrees. In *Symposium on Geometry Processing* (2007), pp. 125–133.

[LPC*00] LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINZTON M., ANDERSON S., DAVIS J., GINSBERG J., ET AL.: The digital michelangelo project: 3d scanning of large statues. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), pp. 131–144.

[MMT23] MAGGIORDOMO A., MORETON H., TARINI M.: Micro-mesh construction. *ACM Transactions on Graphics 42*, 4 (2023), 1–18.

[NNSM07] NIELSEN M. B., NILSSON O., SÖDERSTRÖM A., MUSETH K.: Out-of-core and compressed level set methods. *ACM Transactions on Graphics 26*, 4 (2007), 16–es.

[NZIS13] NIESSNER M., ZOLLHÖFER M., IZADI S., STAMMINGER M.: Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics 32*, 6 (2013), 1–11.

[OBA*03] OHTAKE Y., BELYAEV A., ALEXA M., TURK G., SEIDEL H.: Multi-level partition of unity implicits. *ACM Transactions on Graphics* (2003), 463–470.

[Paj05] PAJAROLA R.: Stream-processing points. In *Proceedings of the Conference on Visualization '05* (2005).

[Res] Resonai. `https://www.resonai.com/`.

[RGA*21] RAKOTOSAONA M.-J., GUERRERO P., AIGERMAN N., MITRA N. J., OVSJANIKOV M.: Learning delaunay surface elements for mesh reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2021), pp. 22–31.

[STJ*17] SCHERTLER N., TARINI M., JAKOB W., KAZHDAN M., GUMHOLD S., PANOZZO D.: Field-aligned online surface reconstruction. *ACM Transactions on Graphics 36*, 4 (2017), 1–13.

[UB15] UMMENHOFER B., BROX T.: Global, dense multiscale reconstruction for a billion points. In *Proceedings of the IEEE international conference on computer vision* (2015), pp. 1341–1349.

[VCL*06] VO H. T., CALLAHAN S. P., LINDSTROM P., PASCUCCI V., SILVA C. T.: Streaming simplification of tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics 13*, 1 (2006), 145–155.

[WSS*19] WILLIAMS F., SCHNEIDER T., SILVA C., ZORIN D., BRUNA J., PANOZZO D.: Deep geometric prior for surface reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2019), pp. 10130–10139.

[ZGHG08] ZHOU K., GONG M., HUANG X., GUO B.: *Highly parallel surface reconstruction*. Tech. rep., Microsoft Research Asia, 2008.

Appendix A: Estimating redundancy

With the introduction of padding regions, each sample point can affect the octree construction for multiple clients, resulting in additional octree nodes. In this appendix, we analyze this redundancy overhead. We begin by recalling that the distributed reconstruction is parameterized by several variables:

- the depth D of the reconstruction,
- the depth d of the coarse octree,
- the number of clients C, and
- the padding size P.

Naive approach

A naive implementation is to introduce all points in the padded region at the finest depth of the tree. Because the expected number of samples in the padding region added to a slab is proportional to $2 \cdot P$ and the expected number of points within the interior of the slab is proportional to $2^d/C$, we obtain a redundancy factor

$$R_N(P) = \frac{2 \cdot P}{2^d/C} = \frac{2 \cdot P \cdot C}{2^d}. \tag{1}$$

Adaptive approach

In the adaptive approach, padding points are only introduced up to a depth δ where they are within a distance of $\frac{P}{2^\delta}$ from the interior of the slab. That is, they are within a distance of P from the interior, in units measured by the size of octree nodes at depth δ. Thus, while all points in the padding region are introduced at depth d, we expect that half of those will be introduced at depth $d+1$, and only half of those will be introduced at depth $d+2$, and so on. Furthermore, we note that if, on average, the points in a slab contribute to κ nodes at the finest depth D, they will contribute to $\frac{\kappa}{4}$ nodes at depth $D-1$, $\frac{\kappa}{4^2}$ nodes at depth $D-2$, and so on.

Combining these observations, we estimate the redundancy factor by first computing the expected number of nodes generated by samples in the padding region:

$$\begin{aligned}\sum_{i=0}^{D-d} \frac{2 \cdot \kappa \cdot P}{4^i \cdot 2^{D-d-i}} &= 2 \cdot \kappa \cdot P \cdot \frac{\sum_{i=0}^{D-d} 2^{D-d-i}}{4^{D-d}} \\ &= 2 \cdot \kappa \cdot P \cdot \frac{2^{D-d+1}-1}{4^{D-d}} \approx \frac{4 \cdot \kappa \cdot P}{2^{D-d}}.\end{aligned} \tag{2}$$

Then, using the fact the expected number of nodes generated by interior points is $\frac{\kappa \cdot 2^d}{C} \cdot (1+1/4+1/4^2+\cdots)$, the redundancy factor becomes:

$$R_A(P) = \frac{4 \cdot \kappa \cdot P/2^{D-d}}{\kappa \cdot 2^d/C \cdot 4/3} = \frac{3 \cdot P \cdot C}{2^D}. \tag{3}$$