Parallel Poisson Surface Reconstruction

Matthew Bolitho[1], Michael Kazhdan[1], Randal Burns[1], and Hugues Hoppe[2]

[1] Department of Computer Science, Johns Hopkins University, USA
[2] Microsoft Research, Microsoft Corporation, USA

Abstract. In this work we describe a parallel implementation of the Poisson Surface Reconstruction algorithm based on multigrid domain decomposition. We compare implementations using different models of data-sharing between processors and show that a parallel implementation with distributed memory provides the best scalability. Using our method, we are able to parallelize the reconstruction of models from one billion data points on twelve processors across three machines, providing a nine-fold speedup in running time without sacrificing reconstruction accuracy.

1 Introduction

New scanning and acquisition technologies are driving a dramatic increase in the size of datasets for surface reconstruction. The Digital Michelangelo project [1] created a repository of ten scanned sculptures with datasets approaching one billion point samples each. New computer vision techniques [2] allow three dimensional point clouds to be extracted from photo collections; with an abundance of photographs of the same scene available through online photo sites, the potential for truly massive datasets is within reach. Processing such large datasets can require thousands of hours of compute time. Recent trends in microprocessor evolution show a movement toward parallel architectures: Multi-core processors are now commonplace among commodity computers, and highly parallel graphics hardware provides even higher performance per watt. Traditional single threaded algorithms will no longer benefit from Moore's law, introducing a new age in computer science in which efficient parallel implementations are required.

This paper presents an efficient, scalable, parallel implementation of the Poisson Surface Reconstruction algorithm [3]. The system is designed to run on multi-processor computer systems with dsitributed memory, allowing the reconstruction of some of the largest available datasets in significantly less time than previously possible. We begin our discussion with a brief review of both serial and parallel surface reconstruction algorithms in Section 2. We then provide a more in-depth review of the Poisson Surface Reconstruction algorithm on which our work is based, presenting a review of the original implementation in Section 3, and its adaptation to a streaming implementation in Section 4. We describe our parallel reconstruction algorithm in Section 5 and evaluate its effectiveness in terms of both accuracy and efficiency in Section 6. Finally, we conclude by summarizing our work in Section 7.

2 Related Work

Surface reconstruction has been a well studied problem within the field of Computer Graphics. The work can be roughly divided into two categories: Computational geometry based methods; and function fitting methods.

Computational Geometry: Computational geometry based methods use geometric structures such as the Delaunay triangulation, alpha shapes or the Voronoi diagram [4–9] to partition space based on the input samples. Regions of space are then classified as either 'inside' or 'outside' the object being reconstructed and the surface is extracted as the boundary between interior and exterior regions. As a consequence of using these types of structures, the reconstructed surface interpolates most or all of the input samples. When noise is present in the data, the resulting surface is often jagged and must be refit to the samples [7] or smoothed [5] in post-processing.

Function Fitting: The function fitting approaches construct an implicit function from which the surface can be extracted as a level set. These methods can be broadly classified as global or local approaches.

Global fitting methods commonly define the implicit function as the sum of radial basis functions centered at each of the input samples [10–12]. However, the ideal basis functions, poly-harmonics, are globally supported and non-decaying, so the solution matrix is dense and ill-conditioned. In practice such solutions are hard to compute for large datasets.

Local fitting methods consider subsets of nearby points at a time. A simple scheme is to estimate tangent planes and define the implicit function as the signed distance to the tangent plane of the closest point [13]. Signed distance can also be accumulated into a volumetric grid [14]. For function continuity, the influence of several nearby points can be blended together, for instance using moving least squares [15, 16]. A different approach is to form point neighborhoods by adaptively subdividing space, for example with an octree. Blending is possible over an octree structure using a multilevel partition of unity, and the type of local implicit patch within each octree node can be selected heuristically [17]. Since local fitting methods only consider a small subset of the input points at a time, the solutions are more efficient to compute and handle large datasets with greater ease than global methods. The greatest challenge for local fitting methods is how to choose the subset of points to consider at any given point in space. These heuristic partitioning and blending choices make local fitting methods less resilient to noise and non-uniformity in the input samples.

Parallel Surface Reconstruction Despite the increasing presence of commodity parallel computing systems, there has been comparatively little work on parallel surface reconstruction. The work of [18] implements the Poisson method on the GPU, achieving significant speedups for small datasets. A significant limitation of the implementation is that it requires the entire octree, dataset and supplemental lookup tables to reside in GPU memory, limiting the maximum size of reconstructions possible. To simplify the lookup of neighbor nodes in the octree and reduce the total number of node computations required, the implementation also uses first-order elements. While this allows a more GPU-friendly implementation, the lower-degree functions make the method more susceptible to noise and other anomalies in the input data.

Some other surface reconstruction algorithms lend themselves to efficient parallel implementations. Many local implicit function fitting methods can be at least partially parallelized by virtue of the locality of most data dependencies. Global implicit function fitting methods often have complex data dependencies that inhibit parallelism. Finally, computational geometry approaches can leverage parallel processing by computing structures such as the Delaunay triangulation in parallel (e.g. [19]).

3 Poisson Surface Reconstruction

The Poisson Surface Reconstruction method [3] uses a function fitting approach that combines benefits from both global and local fitting schemes. It is global and therefore does not involve heuristic decisions for forming local neighborhoods, selecting surface patch types, and choosing blend weights. Yet, the basis functions are associated with the ambient space rather than the data points, are locally supported, and have a simple hierarchical structure that allows the resulting linear system to be solved efficiently.

The Poisson Idea: To solve the surface reconstruction problem, the Poisson method reconstructs the indicator function, a function that has value one inside the surface and zero outside. The key idea is to leverage the fact that an oriented point set can be thought of as a sampling of the gradient of the indicator function. This intuition is formalized by using the discrete point set to define a continuous vector field V representing the gradient field of the (smoothed) indicator function. Solving for the indicator function χ then amounts to finding the scalar function whose gradient best matches V, a variational problem that is optimized by solving the Poisson equation: $\Delta\chi = \nabla \cdot V$.

Function Representation: Since the indicator function (and therefore its gradient) only contains high-frequency information around the surface of the solid, an adaptive, multi-resolution basis is used to represent the solution. Specifically, an octree $\mathcal{O}$ is adapted to the point samples and then a function space is defined by associating a tri-variate B-spline F_o to each octree node $o \in \mathcal{O}$. The B-spline F_o is translated to the node center and scaled by the size of the node, and the span $\mathcal{F}$ of the translated and scaled B-splines defines the function-space over which the Poisson equation is solved.

Solving the Poisson Equation: To solve the Poisson equation, a finite-elements approach is used, with the system discretized by using the elements F_o as test functions. That is, the system is solved by finding the function χ in $\mathcal{F}$ such that: $\langle \Delta\chi, F_o \rangle = \langle \nabla \cdot V, F_o \rangle$ for all $o \in \mathcal{O}$.

4 Streaming Implementation

To enable the reconstruction of models larger than memory, the algorithm was adapted to operate out-of-core, using a streaming implementation to maintain only a small subset of the data into memory at any given time [20]. The key observation in performing Poisson Surface Reconstruction in an out-of-core manner is that the computations required for each step of the process are local, due to the compact support of the basis functions F_o. Specifically, each step can be described by evaluating a *node function* N across all nodes in $\mathcal{O}$, where the result of $N(o)$ for each $o \in \mathcal{O}$ is dependent only on the nodes $o' \in \mathcal{O}$ where $\langle F_o, F_{o'} \rangle \neq 0$ (i.e whose base functions overlap). Spatially, this corresponds to a small neighborhood immediately surrounding o and all its ancestors in $\mathcal{O}$.

In a one dimension, the most efficient streaming order is a simple left to right traversal of the tree nodes: in this configuration, neighborhoods are always compact and contiguous in the data streams. This ordering is generalized to higher dimensions by grouping nodes into 2D slices (i.e. all nodes at a certain depth with the same x-coordinate), and streaming the data on a slice-by-slice basis. Since a node must remain in main memory from the first to the last time it is referenced by other nodes, and since coarse levels of the tree have longer lifetimes than finer levels of the tree, a multi-level streaming approach is used.

5 Parallel Surface Reconstruction

When designing the streaming implementation, one of the primary concerns was minimizing the effect of the I/O required for out-of-core processing. In particular, this motivated the streaming approach (since streaming I/O is highly efficient) and the minimization of the number of passes required through the data (minimizing the total amount of I/O performed). When considering a parallel implementation, a different set of design concerns prevail: minimizing data sharing and synchronization.

5.1 Shared Memory Implementation

In this section, we consider a simple, shared-memory parallelization of the reconstruction algorithm. Although this is not the model we use for our final implementation, analyzing the shared-memory implementation provides valuable insight regarding the key properties a parallel solver must satisfy to demonstrate good speed-up when parallelized across numerous processors.

The most straightfoward parallelization of the serial streaming implementation is to evaluate each node function N_i in parallel, since the restrictions placed on data dependencies for efficient streaming allow the function to be evaluated in any order within a slice. With slices in large reconstructions typically containing tens or hundreds of thousands of nodes, there appears to be ample exploitable parallelism. A data decomposition approach can be used, with each processing slice in the octree partitioned into a coarse regular grid. Since the data dependencies of a node function are compact, the only shared data between partitions within the same level of the tree resides around the perimeter of each partition. To execute a node function across a slice, each processor is assigned a number of data partitions for processing in a way that best balances the workload. Although this approach provides a straightforward implementation, it has two significant scalability issues.

First, for distributive functions, data are shared not only within a level of the tree, but across all depths of the tree. At the finest levels, the contention for shared data is very low – since a very small portion of each partition is shared, the probability of two processors needing concurrent access to a datum are low. At the coarser levels of the tree, however, the rate of contention becomes very high – and the data associated with the coarsest levels of the tree are updated by each processor for every computation. We found that this problem persisted even when we used an optimistic, lock-free technique that implemented an atomic floating-point accumulation (i.e. A+ = x) using the compare-and-set (CAS) instruction found in most modern processor architectures.

Second, scalability is limited by the frequency of global synchonization barriers required to evaluate multiple functions correctly. Each streaming pass, P, across the octree is a pipeline of functions $P = \{N_1, N_2, ..., N_n\}$ that are executed in sequence. Although the data dependencies are such that the evaluation of $N_i(o)$ cannot depend on the result of $N_i(o')$, it is possible that N_i may depend on N_j if $i > j$. The implication of this in a parallel setting is that function N_i cannot be evaluated for a particular slice until N_j has completed processing in dependent slices *on all processors*, requiring a global synchronization barrier between *each function*, for *each processing slice*. For all but the very largest reconstructions, the resulting overhead is prohibitive. Although this synchronization frequency can be reduced by processing functions over slabs of data formed from multiple octree slices, the associated increase in in-core memory usage results in an undesirable practical limitation on the reconstruction resolution.

5.2 Distributed Memory Implementation

To address the scalability issues that arise from using a shared memory, multithreaded architecture, we instead implement our solver using a distributed memory model. In this model, each processor shares data explicitly through message passing, rather than implicitly through shared memory. The advantages of this model over the shared memory approach are as follows:

1. Each processor maintains a private copy of all data it needs. Thus, data writes during computation can be performed without the need for synchronization. Data modified on more than one processor can be easily and efficiently reconciled at the end of each computation pass.
2. Without the need for shared memory space, the system can be run on computing clusters, offering the potential for greater scalability, due to the increased memory and I/O bandwidth, as well as number of processors.

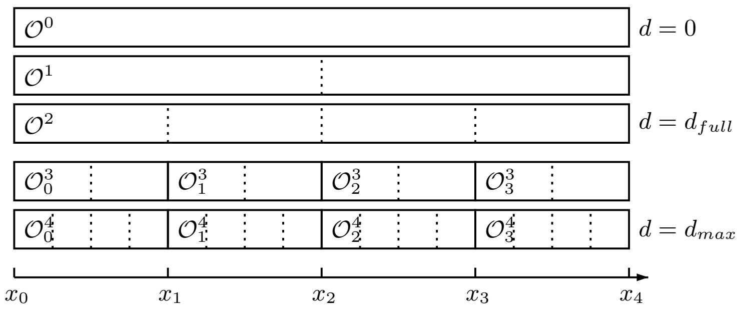

Fig. 1: An illustration of the way data partitions are formed from the tree with $p = 4$ processors. All nodes in $\mathcal{O}^0$, $\mathcal{O}^1$ and $\mathcal{O}^2$ are shared amoungst all processors, and together form the data partition $\mathcal{O}^{full}$. The nodes in remaining depths are split into spatial regions defined by the x-coordinates $\{x_0, x_1, x_2, x_3, x_4\}$, to form the partitions $\mathcal{O}^d_i$.

Furthermore, we adapt the streaming implementation by implementing each of the functions as a separate streaming pass through the data. While this increases the amount of I/O performed, it alleviates the need for global, inter-slice, synchronization barriers that are required to allow multiple functions to be evaluated correctly.

Data Partitioning Instead of fine-grained, slice-level parallelism, the distributed system uses a coarse-grained approach: given p processors, the reconstruction domain is partitioned into p slices along the x-axis given by the x-coordinates $X = \{x_0, x_1, x_2, ..., x_p\}$. The nodes from depth d in the octree are split into partitions $\mathcal{O}^d = \{\mathcal{O}^d_1, \mathcal{O}^d_2, ..., \mathcal{O}^d_p\}$ where $\mathcal{O}^d_p$ are all nodes $o \in \mathcal{O}$ such that $x_p \leq o.x < x_{p+1}$ and $o.d = d$.

Since the coarse nodes in the tree are frequently shared across all processors, we designate the first d_{full} levels in the tree to be part of its own data partition $\mathcal{O}^{full}$, which is not owned by a particular process, and whose processing is carried out *in duplicate* by all processors. Since the total data size of $\mathcal{O}^{full}$ is small, the added expense of duplicating this computation is significantly less than the cost of managing consistent replication of the data.

Figure 1 summarizes the decomposition of the octree into partitions. A processor P_i is assigned to own and process the nodes in $\mathcal{O}^*_i$ in a streaming manner. To allow for data sharing across slabs, processor i has a copy of data in partitions $\mathcal{O}^*_{i-1}$ and $\mathcal{O}^*_{i+1}$ from the result of the previous pass through the data, as well its own copy of $\mathcal{O}^{full}$. Since only a very small portion of data in $\mathcal{O}^*_{i-1}$ and $\mathcal{O}^*_{i+1}$ are ever read or written from P_i (only the data in slices immediated adjacent to $\mathcal{O}^*_i$), the neighboring data partitions are sparsely populated minimizing the amount of redundant storage required.

Since each function is implemented in a separate streaming pass, the execution of a function N_i in one data partition can no longer depend on the execution of a function N_j in another partition, and a global synchronization is only required between the different streaming passes. In practice, we have found that the arithmetic density of most functions is such that the I/O bandwidth required to perform a streaming pass is more than an order of magnitude less that the bandwidth that modern disk drives can deliver, so processing only a single function per pass does not noticeably affect performance.

Load Balancing Because the octree is an adaptive structure, its nodes are non-uniformly distributed over space. This presents a challenge when choosing the partition bounds X in order to most optimally allocate work across all processors. To minimize workload skew, each partition $\mathcal{O}_i^d$ should be approximately the same size (assuming that the processing time of each node is, on average, constant).

Because we wish to perform the allocation of nodes to partitions before the tree has been created, we use the input point-set to estimate the density of nodes in the tree. Since an octree node may not straddle two data partitions, the partition bounds X must be chosen such that each x_i is a multiple of $2^{-d_{full}}$ (i.e. the width of the coarsest nodes in the high-resolution tree). We use a simple greedy algorithm to allocate X: Given an ideal partition size of $N_{ideal} = \frac{N}{p}$ we grow a partition starting at $x = 0$ until the partition size would exceed N_{ideal}. We then over-allocate or under-allocate the partition depending on which minimizes $|N_i - N_{ideal}|$. The procedure is continued along the x-axis until all partition sizes have been determined.

Replication and Merging of Shared Data Once data have been modified by a processor, the changes need to be reconciled and replicated between processors. A majority of the shared updates performed by the reconstructor are of the form $o.v = o.v + v$; that is, accumulating some floating point scalar or vector quantity into tree nodes. The merge process for a process P_i is as follows:

1. If P_i has written to $\mathcal{O}_{i-1}$ and $\mathcal{O}_{i+1}$, send data to P_{i-1} and P_{i+1} respectively.
2. If P_{i-1} and P_{i+1} have modified data in $\mathcal{O}_i$, wait for all data to be received.
3. Merge the received data blocks with the data in $\mathcal{O}_i$ (an efficient vector addition operation).

Once data has been reconciled, the updated data can then be redistributed to other processes as follows:

1. If P_i has been updated and is needed by P_{i-1} or P_{i+1} in the next pass, send $\mathcal{O}_i$ to the neighboring processors.
2. If P_{i-1} and P_{i+1} have modified data needed for the next pass, wait for all updated data blocks.

Because each processor streams through the data partitions, changes made to data can be sent asynchronously to other processing nodes as each block in the stream is finalized, rather than after the pass is complete, thereby hiding the latency involved in most message passing operations.

In addition to the accumulation-based data reconciliation, there are two important steps in the reconstruction process that cannot be merged and replicated as efficiently.

Tree Construction To maximize the parallel processing capability of our system, the construction of the octree itself is performed in parallel. The input point-set P is partitioned during pre-processing into segments $\mathcal{P} = \{\mathcal{P}_1, ..., \mathcal{P}_p\}$ where $\mathcal{P}_i$ contains all points $x_i \leq p.x < x_{i+1}$ (where x_i is the partitioning bounds separating the domain of process P_{i-1} from process P_i).

The first challenge presented in the construction of the tree is the different topological structure created in $\mathcal{O}^{full}$ by each processor. To facilitate efficient merging of data in later steps, it is desirable to have a consistent coarse resolution tree. Although it is possible to merge each of the coarse resolution trees after the first pass, we take a simpler approach: because the coarse resolution tree is small, we pre-construct it as a fully refined octree of depth d_{full}.

The second challenge is that in the initial phases of the reconstruction, a point in partition P_i may affect the creation of nodes outside of $\mathcal{O}_i$ (since the B-splines are supported within the 1-ring of a node). Although this problem could be resolved by allowing processors to generate nodes outside their partition and then merging the nodes at the end of the streaming pass, we have opted for a simpler solution. Recognizing that the points that can create nodes and update data in O_i are in the bounds $x_i - \delta_x \leq p.x < x_{i+1} + \delta_x$, (where $\delta_x = 2^{-d_{full}}$ is the width of the finest-level nodes in the full octree $\mathcal{O}^{full}$) we have processor P_i process this extended subset of points and only perform the associated updates of nodes in $\mathcal{O}_i$. In practice, this adds a small computational cost by processing overlapping point data partitions, but greatly simplifies the creation of the tree.

Solving the Laplacian To solve the Poisson equation correctly in a parallel setting, we use an approach inspired by domain decomposition methods [21]. In the serial implementation, the linear system is solved in a streaming manner using a block Gauss-Siedel solver, making a single pass through the data. Although we can still leverage this technique within each data partition, the regions of the linear system that fall near the boundaries need special treatment to ensure that the solution across partitions is consistent and correct. To avoid the need for the solver in $\mathcal{O}_i$ to depend on a solution being computed in $\mathcal{O}_{i-1}$, each processor P_i solves a linear system that extends beyond the bounds of $\mathcal{O}_i$ by a small region of padding, and once solutions have been computed by all processors, the solution coefficents in overlapping regions are linearly blended together to give a solution which is consistent across partition boundaries.

6 Results

To evaluate our method, we designed two types of experiments. In the first, we validate the equivalence of our parallel implementation to the serial one, demonstrating that correctness is not sacrificed in the process of parallelization. In the

Procs.	Verts.	Tris.	Max δ	Average δ
1	320,944	641,884	-	-
2	321,309	641,892	0.73	0.09
4	321,286	641,903	0.44	0.06
8	321,330	641,894	0.98	0.12

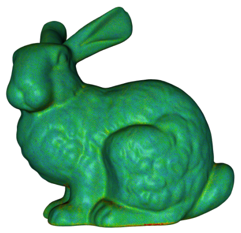

Fig. 2: An analysis of correctness: A comparison of several different reconstructions of the Bunny dataset at depth $d = 9$ created with the distributed implementation. The table summarizes the size of each model, and the maximum and the average distance of each vertex from the ground-truth. The image on the right shows the distribution of error across the $p = 8$ model. The color is used to show δ values over the surface with $\delta = 0.0$ colored blue and $\delta = 1.0$ colored red.

second, we evaluate the scalability of our parallel implementation.

Correctness We wish to ensure that the surface generated by the parallel implementation is equivalent in quality to the serial implementation. In particular, we want to ensure that the model doesn't significantly change as the number of processors increases, and that any differences that do exist do not accumulate near partition boundaries. To test this, we ran an experiment using the distributed implementation, reconstructing the Stanford Bunny model at depth $d = 9$ using 1, 2, 4, and 8 processors. We then compared the model generated with only one processor M_{serial}, to the models generated with multiple processors M_i by computing an error value δ at each vertex of M_i that is the Euclidean distance to the nearest point on the triangle mesh of M_{serial}. The units of δ are scaled to represent the resolution of the reconstruction so that $1.0\delta = 2^{-d}$ (the width of the finest nodes in the tree).

The table in Figure 2 presents the results of this experiment. Some differences in the output are expected between different numbers of processors because of the lack of commutativity of floating point arithmetic. The results show that in all cases, the average error is low, and the maximum error is bounded within the size of the finest tree nodes. It also shows that error does not change significantly as the number of processors increases. The image in Figure 2 shows the distribution of error across the mesh for $p = 8$, and is typical of all multiple processor results. The image highlights that error is evenly distributed across the mesh, and that the only significant error occurs along the shape crease along the bottom of the bunny's back leg. These errors are the result of a different choice in triangulation along the edge.

Scalability One of the most desirable properties of a parallel algorithm is scalability, the of an ability algorithm to run efficiently as the number of processors increases. Table 1 shows the running times and Figure 3 shows the speedup of both the shared memory and distributed memory implementations on up to 12 processors when reconstructing the Lucy dataset from 94 million points, and the

	Shared Memory		Distributed Memory					
	Lucy		Lucy			David		
Processors	Lock Time	Lock-Free Time	Time	Disk	Memory	Time	Disk	Memory
1	183	164	149	5,310	163	1,970	78,433	894
2	118	102	78	5,329	163	985	79,603	888
4	101	68	38	5,368	164	505	81,947	901
6	102	61	26	5,406	162	340	84,274	889
8	103	58	20	5,441	166	259	86,658	903
10	-	-	18	5,481	163	229	88,997	893
12	-	-	17	5,522	164	221	91,395	897

Table 1: The running time (in minutes), aggregate disk use (in MB), and peak memory use (in MB) of the shared memory and distributed memory implementations of the Parallel Poisson Surface Reconstruction algorithms for the Lucy dataset at depth $d = 12, d_{full} = 6$ and the David dataset at depth $d = 14, d_{full} = 8$, running on one through twelve processors. It was not possible to run the shared memory implementation on more than eight processors.

David dataset from 1 billion points. The shared memory implementations were run on a dual quad core workstation. The distributed memory implementation was run on a three machine cluster with quad core processors and a gigabit ethernet interconnect. Two variations of the shared memory implementation are examined: one which uses fine-grained spatial locking to manage concurrent updates, and the other using the lock-free update procedure. The lock-free technique is faster and offers greater scalability than the spatial locking scheme, but the scalability is still limited when compared to the distributed implementation. One significant factor affecting the performance is the way in which both the spatial locking and lock free techniques interact with architectural elements of the underlying hardware. When locking shared data between processors, data that was kept primarily in fast on-chip memory caches has to be flushed and shared through main memory each time it is modified to keep separate caches coherent. This forces frequently shared data to be extremely inefficient to access, with no cache to hide high latency memory access. Because the distributed implementation doesn't need to coordinate writes to the same data, the computation is far more efficient, and cleanly scales with increasing numbers of processors. The reduced scalability as the number of processors increases is due to the complete occupancy of all processors on each machine, causing the algorithm to become memory bandwidth bound. Table 1 also lists the peak in-core memory use and aggregate disk use of the distributed algorithm. Since the in-core memory use is related to the size of the largest slices and each data partition is streamed independently, peak memory use is consistent across all degrees of parallelism. Because of the replication of across processors, the disk use grows as the number of processors increases. A majority of the extra data storage is from $\mathcal{O}^{full}$, whose size grows as d_{full} is increased. For the Lucy model, with $d_{full} = 6$, the size of $\mathcal{O}^{full}$ is 18MB, whereas for the David model, with $d_{full} = 8$, it is 1160MB.

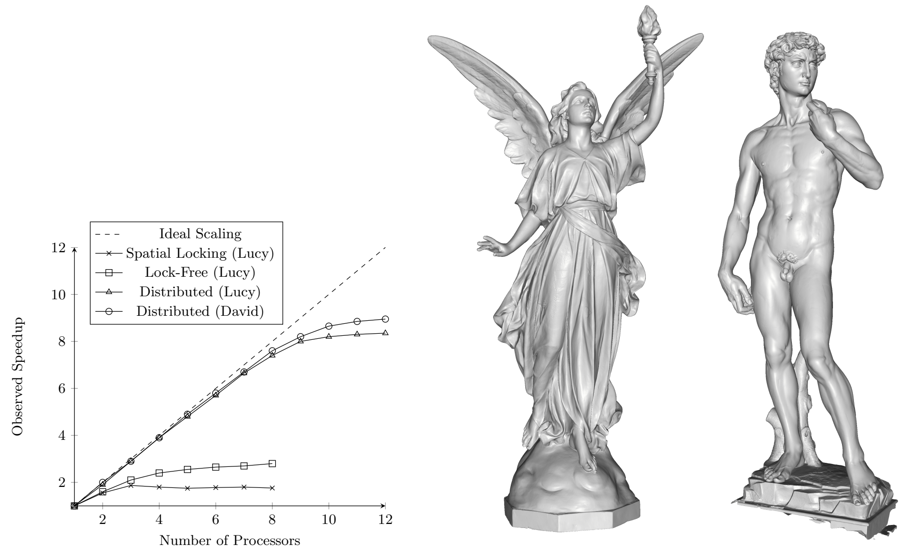

Fig. 3: Analysis of scalability: The speedup of three different parallel Poisson Surface Reconstruction algorithms for the Lucy dataset at depth $d = 12$ and the David dataset at depth $d = 14$ running on one through twelve processors. The Spatial Locking and Lock-Free methods use a shared memory based implementation with two different locking techniques to resolve shared data dependencies. The distributed method uses data replication and message passing to resolve shared data dependencies.

7 Conclusion

We have presented an implementation of the Poisson Surface Reconstruction algorithm that is specifically designed for parallel computing architectures using distributed memory. We have demonstrated both its equivalence to the serial implementation and efficient execution on commodity computing clusters with a nine-fold speedup in running time on twelve processors. One avenue we intend to persue in future work is support for parallel processing on a GPU-based cluster.

8 Acknowledgements

We would like to acknowledge the Stanford 3D Scanning Repository for generously distributing their data. The authors would also like to express particular thanks to Szymon Rusinkiewicz and Benedict Brown for providing non-rigid body aligned Lucy and David scan data [22]. This work was supported in part by an NSF CAREER Award. The authors would also like to thank NVIDIA Corporation for support.

References

1. Levoy, M., Pulli, K., Curless, B., Rusinkiewicz, S., Koller, D., Pereira, L., Ginzton, M., Anderson, S., Davis, J., Ginsberg, J., Shade, J., Fulk, D.: The Digital Michelangelo project: 3D scanning of large statues. In: SIGGRAPH 2000. (2000)
2. Snavely, N., Seitz, S.M., Szeliski, R.: Modeling the world from Internet photo collections. Int. J. Comput. Vision **80** (2008) 189–210
3. Kazhdan, M., Bolitho, M., Hoppe, H.: Poisson surface reconstruction. In: SGP. (2006) 61–70
4. Boissonnat, J.D.: Geometric structures for three-dimensional shape representation. ACM TOG **3** (1984) 266–286
5. Kolluri, R., Shewchuk, J.R., O'Brien, J.F.: Spectral surface reconstruction from noisy point clouds. In: SGP. (2004) 11–21
6. Edelsbrunner, H., Mücke, E.P.: Three-dimensional alpha shapes. ACM TOG **13** (1994) 43–72
7. Bajaj, C.L., Bernardini, F., Xu, G.: Automatic reconstruction of surfaces and scalar fields from 3d scans. In: SIGGRAPH. (1995) 109–118
8. Amenta, N., Bern, M., Kamvysselis, M.: A new voronoi-based surface reconstruction algorithm. In: SIGGRAPH. (1998) 415–421
9. Amenta, N., Choi, S., Kolluri, R.K.: The power crust, unions of balls, and the medial axis transform. Comp. Geometry **19** (2000) 127–153
10. Muraki, S.: Volumetric shape description of range data using "blobby model". In: SIGGRAPH. (1991) 227–235
11. Carr, J.C., Beatson, R.K., Cherrie, J.B., Mitchell, T.J., Fright, W.R., McCallum, B.C., Evans, T.R.: Reconstruction and representation of 3d objects with radial basis functions. In: SIGGRAPH. (2001) 67–76
12. Turk, G., O'brien, J.F.: Modelling with implicit surfaces that interpolate. ACM TOG **21** (2002) 855–873
13. Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., Stuetzle, W.: Mesh optimization. In: SIGGRAPH. (1993) 19–26
14. Curless, B., Levoy, M.: A volumetric method for building complex models from range images. In: SIGGRAPH. (1996) 303–312
15. Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., Silva, C.T.: Point set surfaces. In: IEEE VIS. (2001) 21–28
16. Shen, C., O'Brien, J.F., Shewchuk, J.R.: Interpolating and approximating implicit surfaces from polygon soup. In: SIGGRAPH. (2004) 896–904
17. Ohtake, Y., Belyaev, A., Alexa, M., Turk, G., Seidel, H.P.: Multi-level partition of unity implicits. ACM TOG **22** (2003) 463–470
18. Zhou, K., Gong, M., Huang, X., Guo, B.: Highly parallel surface reconstruction. Technical Report 53, Microsoft Research (2008)
19. Hardwick, J.C.: Implementation and evaluation of an efficient parallel delaunay triangulation algorithm. In: Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures. (1997) 23–25
20. Bolitho, M., Kazhdan, M., Burns, R., Hoppe, H.: Multilevel streaming for out-of-core surface reconstruction. In: SGP. (2007) 69–78
21. Smith, B., Bjorstad, P., Gropp, W.: Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations. (Cambridge University Press)
22. Brown, B.J., Rusinkiewicz, S.: Global non-rigid alignment of 3-D scans. ACM TOG **26** (2007)