Out-of-Core Multigrid Solver for Streaming Meshes

Xiaohan Shi

Hujun Bao

Kun Zhou

State Key Lab. of CAD&CG, Zhejiang University*

Abstract

We present an out-of-core multigrid for solving the Poisson equation defined over gigantic meshes. This enables gradient-domain operations on out-of-core meshes with irregular connectivity. Taking a streaming mesh and boundary constraints as input, our solver builds a multigrid hierarchy and refines the multigrid solution progressively by performing all operations as streaming computations. A set of rules are carefully designed to make neighboring multigrid nodes perform tasks cooperatively and efficiently. With a sublinear memory growth with respect to the number of mesh vertices, our approach handles meshes with 14M vertices using merely 84MB of memory, while an equivalent in-core multigrid implementation fails to fit into 2GB memory space.

Keywords: out-of-core multigrid solver, irregular domain, Poisson equation, gradient domain mesh editing

1 Introduction

Many problems in computer graphics can be formulated as solutions to a Poisson equation. Among the various techniques developed to solve the Poisson equation, multigrid solvers utilize hierarchies to achieve quick convergence for large systems while maintaining linear memory requirement [Brandt 1977; Briggs et al. 2000]. On regular domains like images, multigrid solvers can be adopted without much customization. Shi et al. developed a fast multigrid algorithm for *irregular* domains (*i.e.*, meshes) [Shi et al. 2006]. Kazhdan and Hoppe presented a streaming multigrid algorithm for gigapixel (i.e., out-of-core) images [Kazhdan and Hoppe 2008]. While these approaches target irregular connectivity and out-of-core data respectively, neither of them allows both out-ofcore data and irregular connectivity. In this paper, we propose an out-of-core multigrid approach for solving the Poisson equation on gigantic meshes with irregular connectivity. Our approach processes signals defined over a mesh in a streaming fashion, allowing gradient-domain operations on out-of-core meshes.

1.1 Related Work

Our work is mostly related to Poisson-based methods, multigrid solvers and out-of-core algorithms.

Poisson-Based Methods In image processing, Poisson Image Editing [Pérez et al. 2003] edits the texture, the illumination and the color of objects in images by solving the Poisson equation. A variety of tools are introduced to specify the desired gradient field of images. In geometry processing, surface and volumetric meshes

are deformed, smoothed, and merged by manipulating the gradient field and position constraints of meshes in the Poisson equation [Yu et al. 2004; Sorkine et al. 2004; Zhou et al. 2005]. Surface reconstruction from oriented points can also be cast as a Poisson problem [Kazhdan et al. 2006].

In-Core Multigrid Algorithms Since images have a *regular* domain, multigrid solvers can be directly used for image processing [Pérez et al. 2003]. Such solvers have also been implemented efficiently on the GPU [McCann and Pollard 2008]. However, meshes differ from images in that they usually have *irregular* connectivity.

Shi *et al.* [2006] developed a fast multigrid algorithm for mesh editing by customizing prolongation and restriction operators. Their method assumes that the solution vector (the mesh itself) fits in memory. Our streaming multigrid algorithm follows their multigrid operators (on residual equations), but solves very large Poisson equations on out-of-core meshes.

Georgii and Westermann [2006] presented a multigrid framework for constructing implicit, yet interactive solvers for the governing equation of motion of deformable volumetric bodies. They also assume the volumetric model fits in memory. While this framework solves the more general elliptic PDEs (*e.g.*, the governing equation), our approach focuses on the Poisson equation. It is possible to extend our approach to solve sparse elliptic PDEs in the future.

Out-of-Core Multigrid Algorithm for Regular Domains Kazhdan and Hoppe developed a streaming multigrid for gradient domain-operations on out-of-core images [Kazhdan and Hoppe 2008]. While traditional multigrid implementations perform restriction, prolongation, and relaxation iterations as separate passes, their streaming multigrid performs all operations as streaming computations and groups together as many computations as possible. Specifically, they create a window spanning entire rows of the image and tall enough to account for all data dependencies. As the window sweeps down the image, streaming operations are performed on the pixels in the window. Since only the pixels in the window are resident in memory, gigapixel images can be handled. Extending Kazhdan and Hoppe's idea to irregular domains, however, is highly non-trivial, as discussed in Section 1.2.

Other Out-of-Core Algorithms As dataset sizes increase, many external memory techniques have been developed for a wide variety of computer graphics problems. The reader is referred to the survey by Silva *et al.* [2002] for early out-of-core algorithms in computer graphics. More recent algorithms include surface reconstruction [Bolitho et al. 2007], Delaunay triangulation [Isenburg et al. 2006], and I/O-efficient file formats [Isenburg et al. 2005]. Among them, Streaming Meshes [Isenburg et al. 2005] is mostly related to our work. Our approach takes a streaming mesh as input, and focuses on solving the Poisson equation on such an out-of-core mesh.

1.2 Approaches and Challenges

We aim to develop a streaming multigrid solver for out-of-core meshes with *irregular* connectivity. Following Kazhdan and Hoppe's idea, we sweep a window over the mesh while performing streaming operations on vertices in the window. To develop such a streaming multigrid solver for streaming meshes, we need to address two challenges. First, unlike image hierarchies which

^{*}Email: {shixiaohan,bao}@cad.zju.edu.cn, kunzhou@acm.org



Figure 1: Our streaming multigrid pipeline, showing three different streaming passes.

are naturally and implicitly defined (*e.g.*, the indices of neighbors, parent and child nodes can be directly calculated from the node index), the hierarchies of meshes are very complicated and must be built explicitly. Hierarchy construction must also be performed as streaming operations so that the hierarchy can be built progressively during the window sweep. Second, it is unclear in what order streaming operations should be performed on hierarchies with irregular connectivity. Mechanisms must be developed to help multigrid nodes perform streaming operations correctly and efficiently.

1.3 Contributions

We present an out-of-core multigrid for solving the Poisson equation on out-of-core surface and volume meshes with irregular connectivity. The solver's significance lies in two aspects. First, it proves that a multigrid hierarchy can be created as a streaming process (or more precisely, in a single streaming pass). It thus enables gradient-domain operations on out-of-core meshes. Second, it shows that multigrid operations can be carried out on irregular meshes as streaming computations without too much overhead – it serves as a means of trading performance for memory when solving the Poisson equation, *e.g.*, for users who want to reserve memory for other applications in the operating system.

Our approach has two technical contributions: (1) we convert the construction process of multigrid hierarchy into a set of streaming computations (*i.e.*, *tasks*) to allow progressive execution of the whole solving process, and (2) we define a set of rules that make neighboring nodes perform tasks cooperatively and efficiently.

To demonstrate the potential of our solver, we use it to perform detail-preserving mesh deformation on out-of-core data. An interactive user interface is designed to allow easy specification of user constraints and preview of deformation results on a low-resolution mesh. Our solver then deforms the out-of-core mesh with small memory requirement in reasonable time. We analyze our solver and show that it has a sublinear memory growth with respect to the mesh size. Note that although the deformation of the low-resolution mesh is performed interactively, the whole deformation of the outof-core mesh is not interactive (nor is it expected to be).

2 Out-of-Core Multigrid Solver

We assume that the mesh on which the Poisson equation is defined is represented as a streaming mesh. Arbitrarily formatted meshes can be converted into streaming meshes in a preprocess via spectral sequencing as described in [Isenburg et al. 2005]. The input to our streaming multigrid is a vertex stream which contains for each vertex its reference position, its position constraint (if the vertex is a handle), its number of neighbors and the indices of neighbors that already appeared in the stream. The output of our streaming multigrid solver is a vertex stream in the same vertex order, consisting



Figure 2: Streaming mesh (a) and the in-memory working set of multigrid hierarchy (c). A window sweeps over the streaming mesh (b1-b3), leading to small in-memory working sets. Different colors in (b1-b3) and (c) show the in-memory working set at different time in one streaming pass.

of the deformed position of each vertex – the mesh's vertex and triangle layout (*i.e.*, order) is left unchanged.

Our general purpose streaming multigrid allows the user to provide problem-specific residual equations and customized multigrid operators (*i.e.*, relaxation, restriction and prolongation operators). In other words, the user may formulate these equations and operators for his/her very own purpose (e.g., for solving harmonic field over meshes), and use them in Section 2.2.2 and 2.2.3. We assume that the operators have a local stencil (*i.e.*, the operators only take neighboring vertices as input.), which is usually the case in solving the Poisson equation.

We use a V-cycle multigrid algorithm with multi-level streaming (similar to Kazhdan and Hoppe's pipeline) to solve the Poisson equation with multiple streaming passes (see Figure 1). Taking a single V-cycle as an example, the first pass of the streaming data builds the multigrid hierarchy and performs restriction in all levels from fine to coarse. Meanwhile, the updated data for each node (*i.e.*, neighborhood, current solution, and residual equation) is streamed back to disk, one stream for each level. After the residual equation at the coarsest level is solved by a direct solver (*e.g.*, Intel MKL), a second pass streams node data back from disk and performs prolongation in all levels from coarse to fine. For multiple V-cycles, the prolongation phase of a first V-cycle and the restriction phase of a second V-cycle can be handled in one streaming pass. Therefore, k + 1 streaming passes are sufficient for k V-cycles.

From a mesh point of view, each streaming pass can be thought of as a window sweeping over the mesh, building the in-memory working set of the out-of-core multigrid hierarchy, and performing multigrid operations within. See Figure 2 for a visualization.

V-Cycle Multigrid *vs.* **Full Multigrid** We choose V-cycle multigrid instead of full multigrid because the latter requires additional streaming passes, which slows down the performance. For example, full multigrid on an *l*-level hierarchy requires l - 1 streaming passes before the initial solution reaches the finest level. In constrast, our heavily relaxed (*i.e.*, $n \sim 10$ in our tests) V-cycle multigrid usually requires only 3 V-cycles (4 passes) to reach a reasonably accurate solution.



Figure 3: Streaming operations on a corner of a streaming mesh. (a) Mesh connectivity and streaming order. (b) Hierarchy construction: numbers show the order in which the nodes are created. Nodes #1, #5 and #7 are lifted as soon as all their 1-ring neighbors have been created. (c) Restriction: numbers show the order in which the nodes create their residual equations. Node #7 creates its residual equation as soon as all 2-ring neighbors of its child has created theirs. (d) Prolongation: numbers show the order in which the nodes obtain their correction vector. Nodes #2, #3, #5 and #7 copy correction vector from their parents asap. Other nodes interpolate correction vector from the parent of their 1-ring neighbors.

2.1 Multigrid Hierarchy

The multigrid hierarchy we build is a set of fine-to-coarse graphs. The graph at the finest level is the input mesh itself, including all vertices and edges. During graph coarsening, our algorithm greedily chooses a maximal independent vertex set as the vertices of the coarser level graph. This is done by sweeping mesh vertices in streaming order and lifting any unconstrained vertex that does not have a lifted 1-ring neighbor. Using this algorithm, approximately one third of the vertices will be lifted at each level. Edges of the coarser level graph are built by connecting each pair of lifted nodes that are 2-ring neighbors at the finer level. Please see Figure 3 for an illustration and Section 2.2.1 for detailed streaming construction process.

2.2 Tasks & Rules

The whole multigrid solving process is carried out by several streaming passes over the data. Each pass goes through different *phases: hierarchy construction, restriction*, and *prolongation*. Each of these phases is further divided into several tasks, which will be detailed shortly.

For each task, we define (1) operations that a node should perform, (2) data dependencies that a node must respect before performing the operations, and (3) conditions that must be satisfied to complete the task. In order to make neighboring nodes perform tasks cooperatively and efficiently, nodes must notify each other of their progress. We further define for each task (4) which nodes to notify when a node has performed its operations, and (5) which nodes to notify when a node has just completed a task. We summarize the above definitions in Table 1. See Figure 3 for a vivid illustration of how tasks are performed on a typical piece of mesh.

Given the defined conditions, a straightforward implementation might repeatedly check for nodes that can start performing or have just completed a task. In our tests we found that such brute-force checking is very slow. We manage the checking process as a simple count-and-compare mechanism. Specifically, each task defines its *start condition target* (SCT) and *complete condition target* (CCT), which describe the number of conditions that must be satisfied before the task can be started and completed, respectively. Correspondingly, two counters, the *start condition counter* (SCC) and the *complete condition counter* (CCC), keep the number of already-satisfied conditions. Whenever a node is notified to start a task, it increases its SCC. A node actually starts a task only if the increased SCCmatches its SCT. The same applies when a node is notified to complete a task. See Table 1 for appropriate values of SCT and CCT for each task.

We call a lifted node *child*, and its corresponding node in the coarser level *parent*. Unless mentioned otherwise, by *neighbor* we mean 1-ring neighbor. We call the neighbors that come before a vertex in streaming mesh the vertex's *preceding neighbors*, and the neighbors that come after a vertex the vertex's *succeeding neighbors*. For clarity, we assume that the multigrid operators have 1-ring stencil. Rules and counters for multigrid operators with larger stencils can be derived similarly.

2.2.1 Hierarchy Construction

When a node is created, it must build the multigrid hierarchy around it before any multigrid operations can be performed. This is done in two steps: first the node discovers all its neighbors (*registration*), and then the node decides whether it can be lifted to a coarser level (*lifting*).

Note that the hierarchy building process is done only once, during the very first streaming pass. Later in that pass the connectivity information is streamed to disk. Successive passes stream the same information back from disk, so no more hierarchy construction is needed.

Registration When created, a node only knows its preceding neighbors and the number of its neighbors. Moreover, its preceding neighbors are unaware of the node's existence. In *registration*, a node notifies its preceding neighbors of its presence.

A node can start registration as soon as it has been created in memory (*i.e.*, SCT = 0). To complete registration, a node must know all its neighbors. For a node at the finest level, the number of neighbors is directly obtained from the streaming mesh data (*i.e.*, CCT = |succeeding neighbors|). However, for a node at a coarse level, determining the number of neighbors is more complex. First, the 2-ring neighbors of its child should decide their lift/retain status. Then, the number of neighbors is computed by counting the number of lifted nodes in its child's 2-ring neighborhood (*i.e.*, CCT = |child's 2-ring neighbors| + |child's lifted 2-ring neighbors|).

To review, after a node is created in memory it notifies its preceding neighbors first. Then, if it is in a coarse level, it awaits notification from all 2-ring neighbors of its child (which means they have all decided their lift/retain status). Finally, it waits for notification from all its succeeding neighbors (which means they have been loaded into memory).

Lifting After a node has discovered all its neighbors, it may decide whether it can be lifted. According to the sweeping algo-

Phase	Task	Start Condition Target (SCT)	Complete Condition Target (CCT)	Post-Operation Notification	On-Complete Notification
Hierarchy Construction	Registration	0	in finest level: succeeding neighbors in other levels: child's 2-ring neighbors + child's lifted 2-ring neighbors	proceding neighbors	
	Lifting	$ preceding\ neighbors $	neighbors	succeeding neighbors	neighbors + parents of 2-ring neighbors
Restriction	Building Equation	child's 2-ring neighbors +1	0		preceding neighbors
	Pre-Solve Relaxation	neighbors	n	neighbors	parent of 2-ring neighbors
Prolongation	Correction Interpolation	lifted: 1 retained: <i>lifted neighbors</i>	0		preceding neighbors
	Post-Solve Relaxation	neighbors	n	neighbors	child + child's neighbors

Table 1: A summary of the phases, the tasks, and the rules. The rules define for each task: the number of conditions that must be satisfied before starting and completing the task, and the nodes that must be notified after the operations in the task have been performed and the task is completed.

rithm described in Section 2.1, a node waits until all its preceding neighbors have decided their lift/retain status (*i.e.*, SCT = |preceding neighbors|).

A node is lifted *if and only if* it is unconstrained and it does not have a lifted preceding neighbor. Otherwise, it is retained. If a node is lifted, a new node is created to be its parent and is added into the coarser level. After a node determines its lift/retain status, it notifies its succeeding neighbors, so that they may start the lifting task.

A node completes *lifting* whenever all its 1-ring neighbors have decided their lift/retain status (*i.e.*, CCT = |neighbors|), and notifies its neighbors so they may complete *lifting*. This complete condition implies that the node knows its 2-ring neighbors, its child, and its parent (if lifted). On completing *lifting*, a node notifies all parent of its 2-ring neighbors, so that they may complete registration.

2.2.2 Restriction

After the local structure of the multigrid hierarchy has been built, *restriction* formulates the residual equations at all levels from fine to coarse. Each node first builds the residual equation, then relaxes the current solution n times. After this phase, a direct solver is used to solve for the correction vector at the coarsest level.

Building Equation For a node at the finest level, the residual equation can be built directly (*i.e.*, SCT = 0). For a node at a coarse level, the residual equation is built according to the restriction operator [Shi et al. 2006]. Note that its child and the child's 2-ring neighbors must have relaxed their current solutions (*i.e.*, SCT = |child's 2-ring neighbors| + 1).

A node is free to complete *Building Equation* (*i.e.*, CCT = 0). But before that, it must notify its preceding neighbors so that they can start relaxing their current solutions.

Pre-Solve Relaxation Before the residual equation is restricted to a coarser level, a node smoothes its current solution n times. Our relaxation scheme is inspired by Kazhdan and Hoppe's idea of *temporally blocked relaxation*. In our case, a node is allowed to perform the k^{th} relaxation *if and only if* all its preceding neighbors have been relaxed k + 1 times *and* all its succeeding neighbors have been relaxed k times (*i.e.*, SCT = |neighbors|).

After a node's SCC matches SCT, it performs relaxation once (say, the k^{th} relaxation), and notifies its neighbors so they may perform their next round of pre-solve relaxation. The node's CCC is increased to track the number of relaxations performed. If the node

Function StreamingPass(Stream s) // Initialize a ready queue q.Clear() // Read through the whole stream while s.HasNextNode() do // Fetch the next node and put it into the queue n = s.ReadNextNode()CheckNode(n)// Flush the ready queue while q.ContainsNodes() do // Fetch a node and let it perform its task n = q.PopFront()n.PerformCurrentTask() n.NotifyNeighborhood() // Check each notified node for each m just notified by n do CheckNode(m) **Function** CheckNode(Node *n*) // Check if the node can advance to its next task. if n.ACC == n.ACT then n.AdvanceToNextTask() n.NotifyNeighborhood()

for each m just notified by n do
 CheckNode(m)
// Check if the node can start its current task.
if n.SCC == n.SCT then
 g.PushBack(n)

Figure 4: Pseudocode of a streaming pass.

has relaxed *n* times, it has completed *pre-solve relaxation* (*i.e.*, CCT = n). Otherwise, its *SCC* is reset to 0, so that the node waits for its 1-ring neighbors to relax once before performing the $(k + 1)^{th}$ relaxation. Nodes at the coarsest level do not need to perform *pre-solve relaxation*.

After a node completes *pre-solve relaxation*, it must notify all parent of its 2-ring neighbors, so they may build their residual equations.

2.2.3 Prolongation

After a direct solver solves for the correction vector at the coarsest level, *prolongation* corrects current solutions at all levels from coarse to fine. Each node first corrects the current solution according to the correction vectors interpolated from a coarser level, and then smooths the current solution n times.

Member	Туре	Size (Bytes)		
# neighbors	unsigned byte	1		
neighbors	pointer[# neighbors]	4 neighbors		
# preceding neighbors	unsigned byte	1		
mask for lifted neighbors	unsigned int	4		
child	pointer	4		
parent	pointer	4		
\mathbf{v}_{i}^{l}	double[3]	24		
w_{ji}^l	double[# neighbors]	8 neighbors		
\mathbf{g}_{ji}^l	double[# neighbors][3]	24 neighbors		
α_i^l	double	8		
ξ_i^l	double[3]	24		
current task	unsigned byte	1		
SCT, SCC, CCT, CCC	unsigned byte[4]	4		
smooth count	unsigned byte	1		
Total	74 + 36 neighbors			

 Table 2: A node's data structure.

Correction Interpolation The correction vector at the coarsest level is solved using a linear system solver, so it does not require any interpolation. In other levels, lifted and retained nodes interpolate correction from a coarser level using different prolongation operators [Shi et al. 2006].

A lifted node directly copies the correction vector from its parent as soon as its parent has done relaxation n times (SCT = 1). On the other hand, a retained node interpolates the correction vector from the parent of its lifted neighbors (SCT = |lifted neighbors|).

After a node obtains its correction vector, it corrects its current solution and completes the task (*i.e.*, CCT = 0). Before relaxing its current solution, the node notifies its preceding neighbors so that preceding neighbors can start relaxing their current solutions.

Post-Solve Relaxation *Post-solve relaxation* is nearly the same as *pre-solve relaxation*. The only difference is the nodes that must be notified after relaxation completes. A node in the finest level notifies no one. A node in a coarse level notifies its child and the child's neighbors, so they may start prolongation.

We outline the workflow of a streaming pass in Figure 4. A *ready queue* is created to hold all the nodes that can start performing their current tasks. Nodes are streamed from disk and are checked if they are ready. Meanwhile, the ready queue is flushed by repeatedly taking out a node, letting the node perform its task, and putting in any node that becomes ready. When the ready queue is emptied, either successive node data needs to be streamed from disk, or, in the case that the streaming data comes to the end, a whole streaming pass completes.

3 Application: Out-of-Core Detail-Preserving Mesh Deformation

In this section, we demonstrate our streaming multigrid technique by implementing a detail-preserving mesh deformation algorithm for out-of-core data. We follow the formulation of mesh deformation, residual equations and multigrid operators described in [Shi et al. 2006]. Please refer to Appendix for the detailed formulation and equations.

Interactive User Interface Although the whole deformation is not interactive, we still provide the user with interactive feedback by deforming a coarse mesh for preview purpose. The coarse mesh is an approximation of the out-of-core mesh. Specifically, the out-of-core mesh is first down-sampled to a mesh with 5M vertices by remeshing (*e.g.*, Poisson Surface Reconstruction [Kazhdan et al.

2006]). The resulting mesh is further simplified into a mesh with 10K vertices via mesh simplification (*e.g.*, QSlim [Garland and Heckbert 1997]).

The user directly manipulates the coarse mesh, and an in-core multigrid algorithm [Shi et al. 2006] solves for the deformed coarse mesh interactively. Once the user is satisfied with the deformation preview, the user-specified deformation constraint on each coarse mesh vertex is automatically projected onto the nearest out-of-core mesh vertex, and the streaming multigrid solver is launched to solve for the final out-of-core deformation result.

3.1 Implementation Details

Memory Management Our implementation manages memory usage itself to avoid system overhead when frequently allocating and releasing small pieces of memory for each node. Specifically, a memory pool is pre-allocated for the efficient creation of nodes in all hierarchy levels. Note that nodes do not have to be released – the memory pool just marks the corresponding memory block free. The size of the memory pool can either be user-defined (when the user can afford a block of memory that is large enough), or automatically increased as needed (if the user wants to use as little memory as possible). In the latter case, performance is affected slightly when the memory pool size is increased due to the system memory allocation.

Data Structure of A Node A node contains all information necessary for its computation, including information about its neighbors, its numeric variables and its task (see Table 2). The data members are compactly stored, and minimal data types (*i.e.*, byte) are used wherever possible. Notice that the pointers to 1-ring neighbors are only stored once in the array 'neighbors', with preceding neighbors stored at the beginning followed by succeeding neighbors. The first succeeding neighbor can be found by the member '# preceding neighbors', which contains the number of preceding neighbors. The member 'mask for lifted neighbors' is a bit-wise mask indicating all lifted neighbors.

3.2 Performance and Scalability Analysis

To analyze the performance and scalability of our solver, we compare to an in-core multigrid implementation which keep everything in memory and perform multigrid operations as separate passes. See Table 3 for timing and memory statistics.

Performance Our streaming multigrid algorithm is roughly two to three times slower than the in-core algorithm. The performance drop mainly comes from the overhead of streaming tasks like no-tifications and neighborhood checks in order to start or complete tasks. For both the hierarchy construction and multigrid operations (*e.g.*, relaxation, prolongation and restriction), the overhead takes as much time as the numerical computations. This is the price for handling out-of-core data. Note that doing pre-fetching and lazy-write-backs could keep the I/O non-blocking, thus could further reduce the running time of our method (allowing the implementation to hide I/O behind computation so that the running time is just the sum of hierarchy construction and multigrid operation times).

Unlike Kazhdan and Hoppe's algorithm [2008], which is I/Obound, our streaming multigrid algorithm is CPU-bound. There are two reasons. First, more CPU instructions are performed in our algorithm (*i.e.*, building hierarchy and the overhead mentioned above) relative to the amount of I/O. Second, although input streaming meshes have spatially coherent vertex layout, the irregular connectivity of meshes leads to disordered memory access patterns, causing more cache misses than images. Specifically, the 1-ring

Model	# Vert. / # Levels	Solver	I/O Time	H.C. Time	M.O. Time	Total Time	Peak # Nodes	Memory
Bunny	25.047.74	Streaming	1.08s	1.00s	3.48s	5.56s	15.2K	13MB
	55,94774	In-Core		0.65s	1.58s	2.23s		23MB
Elephant	42 221 / 4	Streaming	1.28s	1.15s	4.03s	6.45s	14.6K	14MB
	42,32174	In-Core		0.74s	1.83s	2.56s		27MB
Armadillo	172 074 / 5	Streaming	5.26s	2.59s	16.67s	24.52s	33.2K	20MB
	172,9747.5	In-Core		1.13s	8.02s	9.14s		82MB
Dragon	437,645 / 6	Streaming	13.33s	6.42s	40.59s	60.35s	53.6K	25MB
		In-Core		3.07s	20.19s	23.26s		191MB
Lucy	026 440 / 7	Streaming	28.57s	11.76s	86.04s	126.37s	66.4K	27MB
	930,44077	In-Core		4.68s	42.61s	47.29s		397MB
Lucy	14,027,872 / 9	Streaming	428.08s	153.71s	1306.90s	1888.69s	257.0K	84MB
		In-Core		N/A^{\dagger}	N/A [†]	N/A^{\dagger}		>2GB
XYZ RGB	14 438 400 / 0	Streaming	440.60s	156.50s	1355.39s	1952.50s	239.7K	79MB
Dragon	14,430,40079	In-Core		N/A [†]	N/A [†]	N/A [†]		>2GB

Table 3: Timing and memory statistics of our streaming multigrid and an in-core multigrid, including the time spent for disk I/O, hierarchy construction (H.C.) and multigrid operations (M.O.), and the memory used. All timings were measured on a 2.0GHz Intel Xeon workstation with 2GB RAM. [†]Obtaining statistics on large meshes was impractical because the in-core multigrid requires more than 2GB memory.

neighborhood of a pixel forms a constant number (three) of continuous blocks in memory. However, 1-ring neighborhood a vertex may form variable numbers of continuous blocks in memory, causing more cache misses. We do not have an accurate analysis of the optimal patch size. For a more detailed discussion, the reader is referred to [Sander et al. 2007], which deals with a similar problem.

Scalability With our compact per-node data storage, large meshes with tens of millions of vertices can be deformed with a memory cost affordable by common PCs. For example, deforming the Lucy model (14,027,872 vertices) requires only 84MB of memory. As shown in Table 3, the memory requirement of our approach grows sublinearly (*e.g.*, $O(N^{\frac{1}{2}})$) for surface mesh with respect to the number of mesh vertices. This is easy to understand by imagining a sphere - the perimeter of a sphere (the size of the in-memory working set) is linear with respect to the radius, while the area (the total number of vertices) is quadratic. For volumetric meshes, although not as efficient as for surface meshes, the memory growth of our approach still stays sublinear (*e.g.*, $O(N^{\frac{2}{3}})$).

Limitations Since we assume that the multigrid operators have local stencils, our approach obviously cannot solve dense equations. Even for multigrid operators with *k*-ring stencils (k > 1), calculation time will increase quadratically. Additionally, unlike algorithms that use a fixed-size memory buffer, the memory requirement of our approach grows as the number of mesh vertices increases. It is possible that an extremely large mesh could require more than 2GB of memory in order to be handled, which makes our approach inapplicable. Fortunately, based on our statistics we estimate that such a surface mesh will contain about 500T vertices, far beyond the size of out-of-core data available today.

4 Conclusions

We have presented an out-of-core multigrid for solving the Poisson equation defined over streaming meshes. This enables gradientdomain operations on out-of-core meshes with irregular connectivity. Our key contribution is the streaming multigrid solver for arbitrary meshes, which accepts user-defined problem-specific residual equations and multigrid operators. To make the whole streaming process possible, all operations (including hierarchy construction) are converted into streaming computations. A set of rules have been designed to make neighboring nodes perform different streaming computations cooperatively and efficiently. To this end, our approach builds a multigrid hierarchy and refines the multigrid solution progressively, with a sublinear memory growth with respect to the number of mesh vertices. We have demonstrated the potential of our approach by performing detail-preserving mesh deformation for out-of-core data.

Discussion We assume that arbitrarily formatted meshes are first converted into streaming meshes in a preprocess via spectral sequencing as described in [Isenburg et al. 2005]. We choose spectral sequencing for its smallest front width, because the goal of our paper is to provide a means of trading performance for memory – to handle meshes as large as possible in limited memory space with reasonable performance overhead. Other streaming orders (*i.e.*, breath-first) can be used but might require more (approximately 1.5x) memory space despite possible (approximately 2x) performance improvement. While the number of cores, and thus the computational power of today's CPUs, is increasing at a rate governed by the Moore's Law, the memory size is increasing at a much lower rate. Therefore, minimizing memory consumption is at least as important as maximizing performance in algorithm design.

Future Work It would be interesting to extend our approach to solving sparse elliptic PDEs (*e.g.*, the governing equations for physically-based simulation of elastic deformable models, which are usually large in order to simulate physical effects in fine details). If deformable models are adaptively simulated, mesh topology and multigrid hierarchy change between frames. It will be a challenge to properly transfer data (defined on a time-varying domain) between frames via streaming I/O. It is also interesting to investigate the parallelization of our approach to exploit the processing power of many-core CPUs and GPUs. One challenge might be that there is not enough parallelism in the streaming operations. For example, the relaxation operation of all nodes are executed sequentially in our approach. A possible solution is to perform relaxation by patch, one thread for each patch, with nodes on patch boundaries relaxed at the end.

Acknowledgements

This work is partially supported by the 973 Program of China (No. 2009CB320804), the NSF of China (No. 60825201) and NVIDIA. The authors would like to thank Bennett Wilburn for his help on writing. We are also grateful to the anonymous reviewers for their helpful comments.



Figure 5: Lucy, with her pets around looking at the torch in her hand. (The Lucy model is down-sampled for rendering purpose.)

References

- BOLITHO, M., KAZHDAN, M., BURNS, R., AND HOPPE, H. 2007. Multilevel streaming for out-of-core surface reconstruction. In *Proceedings of ESGP'07*, 69–78.
- BRANDT, A. 1977. Multi-level adaptive solutions to boundaryvalue problems. *Mathematics of Computation* 31, 333–390.
- BRIGGS, W. L., HENSON, V. E., AND MCCORMICK, S. F. 2000. A Multigrid Tutorial. Society for Industrial and Applied Mathematics.
- GARLAND, M., AND HECKBERT, P. S. 1997. Surface simplification using quadric error metrics. In *Proceedings of SIG-GRAPH*'97, 209–216.
- GEORGII, J., AND WESTERMANN, R. 2006. A multigrid framework for real-time simulation of deformable volumes. *Comput*ers & Graphics 30, 3, 408–415.
- ISENBURG, M., LINDSTROM, P., AND LIVERMORE, L. 2005. Streaming meshes. In Proceedings of Visualization'05, 231–238.
- ISENBURG, M., LIU, Y., SHEWCHUK, J., AND SNOEYINK, J. 2006. Streaming computation of delaunay triangulations. ACM Trans. Graph. 25, 3, 1049–1056.
- KAZHDAN, M., AND HOPPE, H. 2008. Streaming multigrid for gradient-domain operations on large images. ACM Trans. Graph. 27, 3, 21.
- KAZHDAN, M., BOLITHO, M., AND HOPPE, H. 2006. Poisson surface reconstruction. In *Proceedings of SGP'06*, 61–70.
- MCCANN, J., AND POLLARD, N. S. 2008. Real-time gradientdomain painting. ACM Trans. Graph. 27, 3, 93.
- PÉREZ, P., GANGNET, M., AND BLAKE, A. 2003. Poisson image editing. *ACM Trans. Graph.* 22, 3, 313–318.

- SANDER, P. V., NEHAB, D., AND BARCZAK, J. 2007. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Trans. Graph.* 26, 3, 89.
- SHI, L., YU, Y., BELL, N., AND FENG, W.-W. 2006. A fast multigrid algorithm for mesh deformation. ACM Trans. Graph. 25, 3, 1108–1117.
- SILVA, C. T., CHIANG, Y.-J., EL-SANA, J., AND LINDSTROM, P. 2002. Out-of-core algorithms for scientific visualization and computer graphics. In *Course Notes for IEEE Visualization* 2002.
- SORKINE, O., COHEN-OR, D., LIPMAN, Y., ALEXA, M., RÖSSL, C., AND SEIDEL, H.-P. 2004. Laplacian surface editing. In *Proceedings of SGP'04*, 175–184.
- YU, Y., ZHOU, K., XU, D., SHI, X., BAO, H., GUO, B., AND SHUM, H.-Y. 2004. Mesh editing with poisson-based gradient field manipulation. ACM Trans. Graph. 23, 3, 644–651.
- ZAYER, R., RÖSSL, C., KARNI, Z., AND SEIDEL, H.-P. 2005. Harmonic guidance for surface deformation. In *Eurographics* 2005, 601–609.
- ZHOU, K., HUANG, J., SNYDER, J., LIU, X., BAO, H., GUO, B., AND SHUM, H.-Y. 2005. Large mesh deformation using the volumetric graph laplacian. ACM Trans. Graph. 24, 3, 496–503.

Appendix: Formulation of Mesh Deformation and Multigrid Operators

Our formulation of mesh deformation follows the two-step method described in [Shi et al. 2006]. The first step computes the harmonic field [Zayer et al. 2005] of the user-specified rotations and scalings and modifies the Laplacian coordinates of the mesh vertices accordingly. The second step solves the deformed mesh according to the modified Laplacian coordinates and user-specified position constraints. Both steps are formulated as solutions of a Poisson equation. Specifically, the per-vertex signal $\bar{\mathbf{x}}_i$ (*i.e.*, rotation, scaling, and vertex position) is governed by

$$\sum_{j \in N(i) \setminus C(i)} w_{ji}(\bar{\mathbf{x}}_i - \bar{\mathbf{x}}_j - \mathbf{d}_{ji}) + \sum_{j \in C(i)} w_{ji}(\bar{\mathbf{x}}_i - \mathbf{c}_j - \mathbf{d}_{ji}) = 0$$
(1)

where N(i) is vertex *i*'s 1-ring neighborhood, C(i) is vertex *i*'s constrained 1-ring neighbors, \mathbf{d}_{ji} is the expected difference between vertex *i* and *j*, and \mathbf{c}_i is a boundary condition.

When an initial solution \mathbf{x}_i is present, a residual equation can be derived by replacing $\bar{\mathbf{x}}_i$ with the initial solution plus a correction $\bar{\mathbf{x}}_i = \mathbf{x}_i + \Delta_i$,

$$\sum_{j \in N(i) \setminus C(i)} w_{ji} (\Delta_i - \Delta_j - \mathbf{g}_{ji}) + \alpha_i \Delta_i = \xi_i$$
 (2)

where $\mathbf{g}_{ji} = \mathbf{d}_{ji} - (\mathbf{x}_i - \mathbf{x}_j)$ is the error in the current solution, $\alpha_i = \sum_{j \in C(i)} w_{ji}$, and $\xi_i = \sum_{j \in C(i)} w_{ji} (\mathbf{d}_{ji} - (\mathbf{x}_i - \mathbf{c}_j))$ is a boundary condition. The following are the relaxation, prolongation and restriction operators derived from the residual equation.

The relaxation operator simply smoothes the current solution of a vertex according to its 1-ring neighbors

$$\mathbf{x}_{i} = \frac{\sum_{j \in N(i) \setminus C(i)} w_{ji}(\mathbf{x}_{j} + \mathbf{d}_{ji}) + \sum_{j \in C(i)} w_{ji}(\mathbf{c}_{j} + \mathbf{d}_{ji})}{\sum_{j \in N(i)} w_{ji}}.$$
 (3)

The prolongation operator interpolates the correction Δ_l^l from level l + 1. For a lifted node, the correction is copied directly from its corresponding node in level l + 1

$$\Delta_i^l = \Delta_i^{l+1}.\tag{4}$$

For a retained node, the correction is smoothly interpolated from its lifted 1-ring neighbors

$$\Delta_{i}^{l} = \frac{\sum_{j \in R^{l}(i)} w_{ji}^{l} (\Delta_{j}^{l+1} + \mathbf{g}_{ji}^{l}) + \xi_{i}^{l}}{\sum_{j \in R^{l}(i)} w_{ji}^{l} + \alpha_{i}^{l}}$$
(5)

where $R^{l}(i)$ is vertex *i*'s lifted 1-ring neighbors in level *l*.

The restriction operator builds the residual equation in level l + 1 according to level l. We want the solution of the newly built residual equation in level l + 1 to satisfy the residual equation in level l as much as possible. Therefore, we take the residual equation in level l and substitute all unknowns in level l into unknowns in level l + 1 using the aforementioned prolongation operators Eq. (4) and Eq. (5)

$$\sum_{j \in K^{l}(i)} w_{ji}^{l} \left(\Delta_{i}^{l+1} - \frac{\sum_{k \in R^{l}(j)} w_{kj}^{l} (\Delta_{k}^{l+1} + \mathbf{g}_{kj}^{l}) + \xi_{j}^{l}}{\sum_{k \in R^{l}(j)} w_{kj}^{l} + \alpha_{j}^{l}} - \mathbf{g}_{ji}^{l} \right) + \alpha_{i}^{l} \Delta_{i}^{l+1} - \xi_{i}^{l} = 0$$

$$(6)$$

where $K^{l}(i)$ is vertex *i*'s retained 1-ring neighbors in level *l*. Rearranging the above equation into the format of Eq. (2) yields the following restriction operators

$$w_{ki}^{l+1} = \sum_{j \in K^{l}(i) \bigcap K^{l}(k)} \frac{w_{kj}^{l} w_{ji}^{j}}{Z_{j}^{l}},$$
(7)

$$\mathbf{g}_{ki}^{l+1} = \frac{1}{w_{ki}^{l+1}} \sum_{j \in K^{l}(i) \cap K^{l}(k)} \frac{w_{kj}^{l} w_{ji}^{l}(\mathbf{g}_{kj}^{l} + \mathbf{g}_{ji}^{l})}{Z_{j}^{l}}, \quad (8)$$

$$\alpha_i^{l+1} = \alpha_i^l + \sum_{j \in K^l(i)} \frac{w_{ji}^l \alpha_j^l}{Z_j^l}, \tag{9}$$

$$\xi_{i}^{l+1} = \xi_{i}^{l} + \sum_{j \in K^{l}(i)} \frac{w_{ji}^{l}(\xi_{j}^{l} + \alpha_{j}^{l} \mathbf{g}_{ji}^{l})}{Z_{j}^{l}}.$$
 (10)

where $Z_i^l = \sum_{s \in R^l(i)} w_{si}^l + \alpha_i^l$.