A Classification of Scientific Visualization Algorithms for Massive Threading

Kenneth Moreland,[‡] Berk Geveci,[*] Kwan-Liu Ma,[†] and Robert Maynard[*]
[‡]Sandia National Laboratories
[*]Kitware, Inc.
[†]University of California at Davis

ABSTRACT

As the number of cores in processors increase and accelerator architectures are becoming more common, an ever greater number of threads is required to achieve full processor utilization. Our current parallel scientific visualization codes rely on partitioning data to achieve parallel processing, but this approach will not scale as we approach massive threading in which work is distributed in such a fine level that each thread is responsible for a minute portion of data. In this paper we characterize the challenges of refactoring our current visualization algorithms by considering the finest portion of work each performs and examining the domain of input data, overlaps of output domains, and interdependencies among work instances. We divide our visualization algorithms into eight categories, each containing algorithms with the same interdependencies. By focusing our research efforts to solving these categorial challenges rather than this legion of individual algorithms, we can make attainable advancement for extreme computing.

1. INTRODUCTION

Processor clock and execution rates have flatlined. Instead, successive generations of processors provide more parallel threading capability [39]. Recent CPUs feature multiple cores and hyperthreading technology to allow each core to run concurrent threads. Furthermore, accelerator type architectures, which have lightweight cores grouped to share control, are becoming increasingly popular for their high ratios of price and power to performance.

High performance computing is also seeing a remarkable increase in the parallelism required on large-scale systems. Consider, for example, the last two generations of leadership-class computers at the Oak Ridge Leadership Computing Facility. The previous Jaguar-XT5 system had a peak performance of about 2 petaflops using about 200 thousand concurrent processes. The current Titan-XK7 system, which incorporates GPU accelerators, has a peak performance of over 20 petaflops but can require 70 to 500 *million* concurrent threads in order to achieve that. Building algorithms that are implemented for new process architectures and programming models and that support the massive parallelization they require is considered one of the top research challenges for scientific visualization [9].

Production visualization products today achieve parallel scalability using a data parallel method that relies on partitioning the data into independent domains for each process [2]. Each domain is processed independently, so ghost regions and overlap are required at domain boundaries so that the interaction of work on parallel processes can be ignored. However, this approach is infeasible when dealing with massive amounts of threads on these emerging architectures. We now need to design new algorithms with a key on data interdependencies to process efficiently in a massively threaded environment.

Unfortunately, to perform scientific visualization with massive threading, we need to redesign our algorithms to work effectively with fine-grained, independent operations. There has already been significant work in making scientific visualization algorithms work well with shared memory threading and accelerator types of architectures [11, 22, 24–26], but all these projects focus on the implementation of a single or select fixed set of algorithms. Our goal in this paper is to present commonalities among various algorithms that share parallel-programming challenges. By addressing these higher level challenges, we can make better progress to ensure that our scalable scientific visualization needs are met.

To identify these high level parallel-programming challenges, we first revisit the key principles on which we base our current parallel visualization algorithms (Section 2) and then categorize our current set of scientific visualization algorithms based on behavior of the fundamental computation (Section 3).

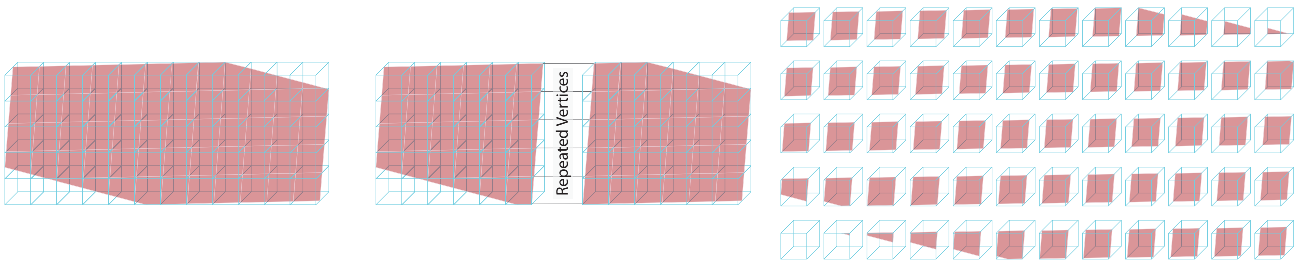
2. KEY PRINCIPLES

Our current scalable scientific visualization relies on a coarse partitioning of data into domains that can often be processed independently. Law et al. [20] provides the following three key principles that must be satisfied for this independent processing to behave correctly.

Data Separability The data can be broken into domains in a way that is simple and efficient. Furthermore, algorithms behave properly on independent domains.

Mappable Input Given an identification for a portion of the output, the input domain responsible for this out-

Ultravis'13 November 17, 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2500-4/13/11 ...$15.00.
http://dx.doi.org/10.1145/2535571.2535591



(a) Serial Partitioning (b) Coarse-Parallel Partitioning (c) Massively-Threaded Partitioning

Figure 1: Partitioning with different levels of parallelism. In serial processing (a) geometry is fully connected. Partitioning for parallel processing introduces disconnected cells and repeated vertices (b), which become unmanageable in massive threading (c).

put can be determined. The output portion can, and often is, identified as simply piece i of N.

Result Invariant The output of the algorithm is equivalent across all possible partitioning.

At first glance, it would appear that any algorithm abiding by these key principles could be partitioned indefinitely. However, there are two problems that arise when building a massively threaded algorithm in which the data is partitioned (potentially) down to domains of single elements.

The first problem is that many algorithms do not really have a clear mapping from an algorithm's output to its input. For example, when computing a contour [23], it is seldom practical *a priori* to know how large the output will be, to know from what domain of the input each piece of the output will originate, and to know what the distribution of elements in the output partitions will be. As long as the data partitioning is coarse enough, managing uneven or empty domains is inconsequential. Load imbalance in downstream processing can either be resolved by dynamic rebalancing or, more commonly, simply tolerating it. However, to achieve efficiency with massive threading, it is important to know the precise elements on which to schedule the working threads. Thus, it is often more practical to determine partitioning by mapping from input to output rather than output to input.

The second problem is that although plenty of algorithms are result invariant in that different partitioning results are *equivalent*, the structures they build are not strictly *isomorphic*. Partitioning data often results in duplicate information on domain boundaries. Consider for example the slice operation demonstrated in Figure 1. In serial (1a) it is straightforward to create a fully connected manifold surface. With coarse level partitioning (1b), as is typically used in MPI processing, partitioning creates gaps between domains and replicated vertices at the boundaries. Massive threading (1c) can result in a completely disconnected mesh. Although all these results are equivalent with respect to the combined results, the different underlying data structures have different properties and associated capabilities.

This duplication of results is a convenient mechanism to avoid communication in parallel processing, and in coarse-level parallelism the duplication is easily managed through ghost regions. (Anecdotally this works best when each process has on the order of 100 thousand to 10 million cells per process [27]) However, if massive threading reduces each domain to every independent element, the duplication is dramatic and creating ghost regions is infeasible. It is therefore not practical to assume result invariance for most algorithms. Instead we have to identify and characterize the *interdependence* of the work where concurrent threads produce coincident data or the threads otherwise require collective operations. It is only then that we can design strategies to manage the work interdependence.

With these issues in mind, we provide an analogous set of key principles for the operation of scientific visualization algorithms for massive threading.

Data Separability The data can be broken down to an elemental level fine enough to provide independent work for sufficient threads.

Discoverable Input Mapping The existence of output elements can be efficiently determined from the input.

Collective Work In the cases where work is not independent, the overlap of responsibility can be resolved through efficient collective operations.

3. CLASSIFICATION OF VISUALIZATION ALGORITHMS

To better understand the effort involved with updating our scientific visualization algorithms to massively threaded processors, we look into the behavior of the algorithms we are currently using in production tools. Specifically, we consider those algorithms available in ParaView, a popular open-source scalable scientific visualization application [4].

For each algorithm in ParaView, we apply the three key principles defined at the end of Section 2: data separability, discoverable input mapping, and collective work. To address data separability, we consider the smallest unit of data on which a single thread can independently operate, which we call the *separable element*. To address discoverable input mapping, we consider how the elements of the input data map to the points and cells of the output geometry and the output fields. Finally, we also identify any *collective work* the algorithm must perform in a multithreaded environment.

From this information we derive classes of scientific visualization algorithms where the algorithms of each class share the same input, output, and interdependence characteristics. This classification is constructive because once we

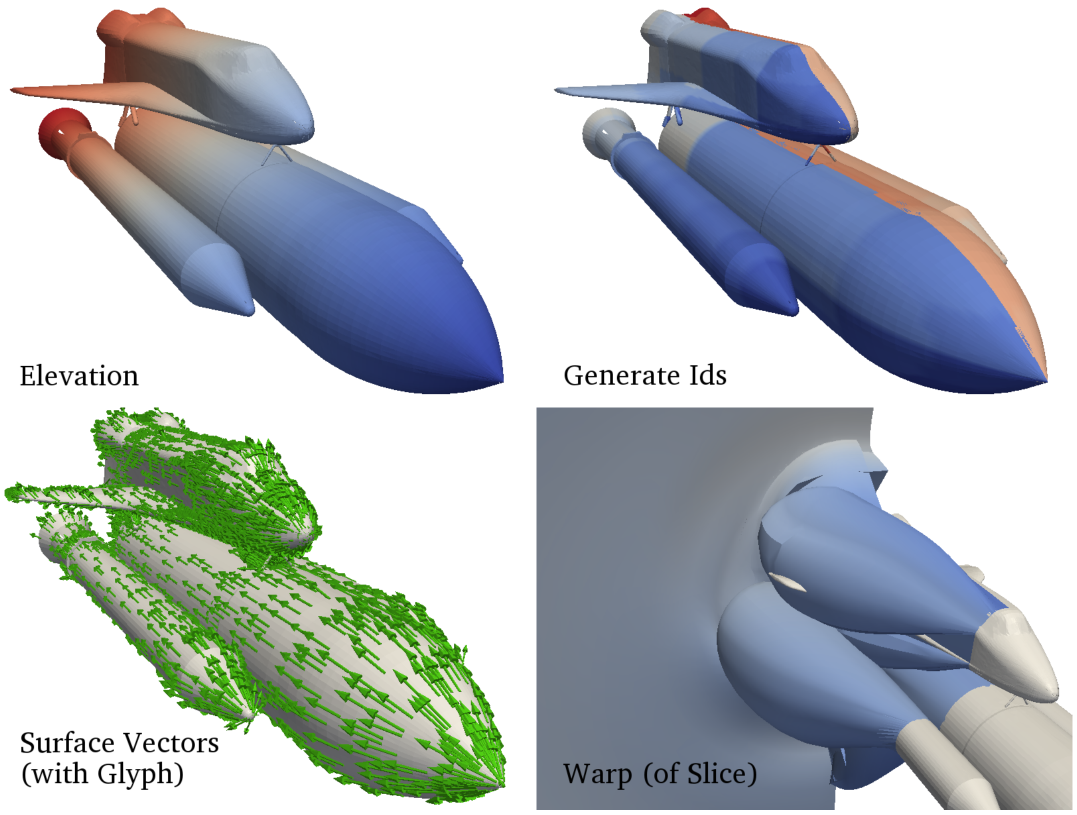


Figure 2: Examples of Basic Mapping algorithms.

resolve the challenges of mapping input to output and managing interdependence with collective operations, the algorithms within each class are straightforward to implement from their serial counterparts. A reference for all these classifications is given at the end of this paper in Table 1.

Basic Mapping

Separable Element	Any
Point Mapping	Identity
Cell Mapping	Identity
Field Mapping	1 to 1
Collective Work	None
Algorithms	Append Attributes, Append Datasets, Calculator, Elevation, Generate Ids, Image/Rectilinear Data to Point Set, Random Vectors, Reflect, Surface Vectors, Texture Map Coordinates, Transform, Warp

The Basic Mapping algorithms operate on an array of field values and generate another array of field values. Each output field value is computed independently using only the associated input value (or values if there is more than one input array).

The typical purpose of this type of algorithm is to generate a derived field of data. Some of these operate on the coordinates of mesh points, for example the Transform filter moves and warps the data with an affine transformation. However, when point coordinates are treated as a field, the behavior characteristics are the same. Some examples of Basic Mapping algorithms are given in Figure 2. The example shuttle data is courtesy of the NASA Advanced Supercomputing Division.

The Basic Mapping class is essentially just a parallel `for` operation over input and output field arrays. As such, basic mapping is essentially built in to many multi- and many-core languages and APIs including OpenMP (parallel for pragma [31]), CUDA (triple chevron notation [35]), Thrust (for_each generic algorithm [7]), and Intel Threading Building Blocks (parallel_for generic algorithm [32]). Although all of these systems use significantly different syntax, Baker et al. [5] show how to simplify porting using a generic programming interface over them.

Map by Cell

Separable Element	Cell
Point Mapping	Identity
Cell Mapping	Identity
Field Mapping	Points on cell to cell
Collective Work	None
Algorithms	Cell Centers, Cell Derivatives, Mesh Quality, Point Data to Cell Data

The Map by Cell class is very similar to Basic Mapping with the exception that the input field arrays are not a one-to-one mapping to the output field arrays. Instead, these operations are performed using the information over a complete cell and attached fields.

Operations of this class involve characterizing the shape of the cell (as in the case of Cell Centers and Mesh Quality) or math operations over continuous function rather than discrete values (as is the case for Cell Derivatives). Some ex-

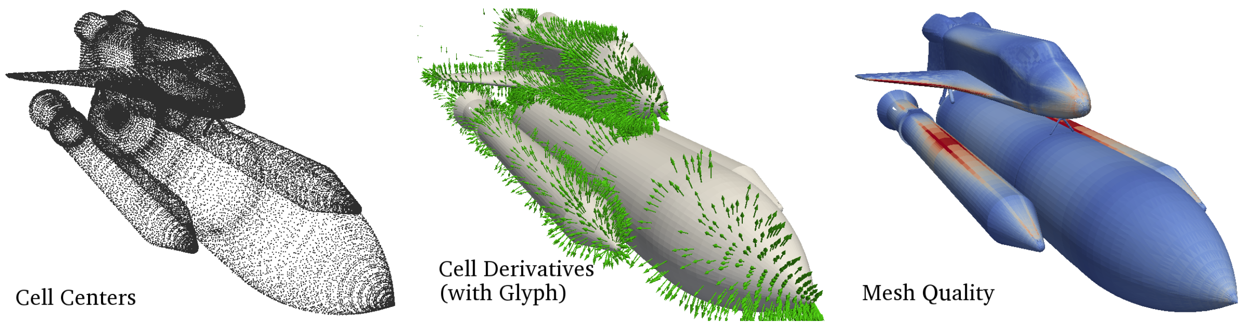


Figure 3: Examples of Map by Cell algorithms.

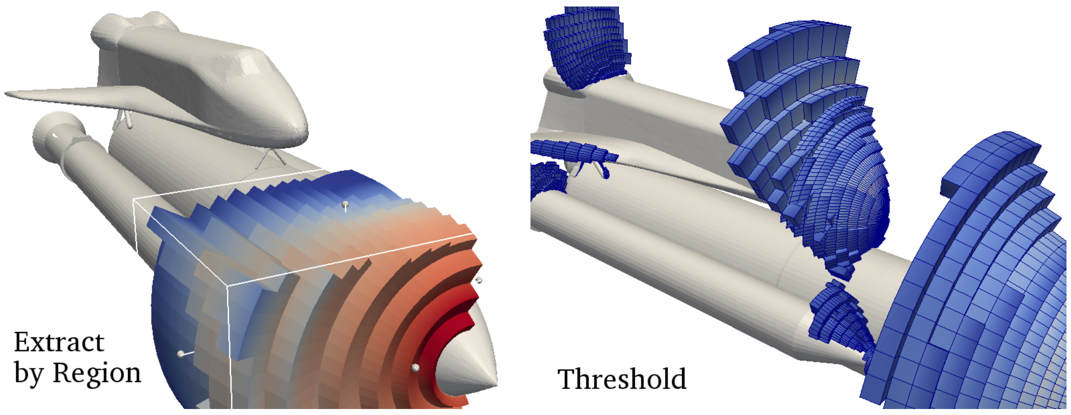


Figure 4: Examples of Reconnect Cell algorithms.

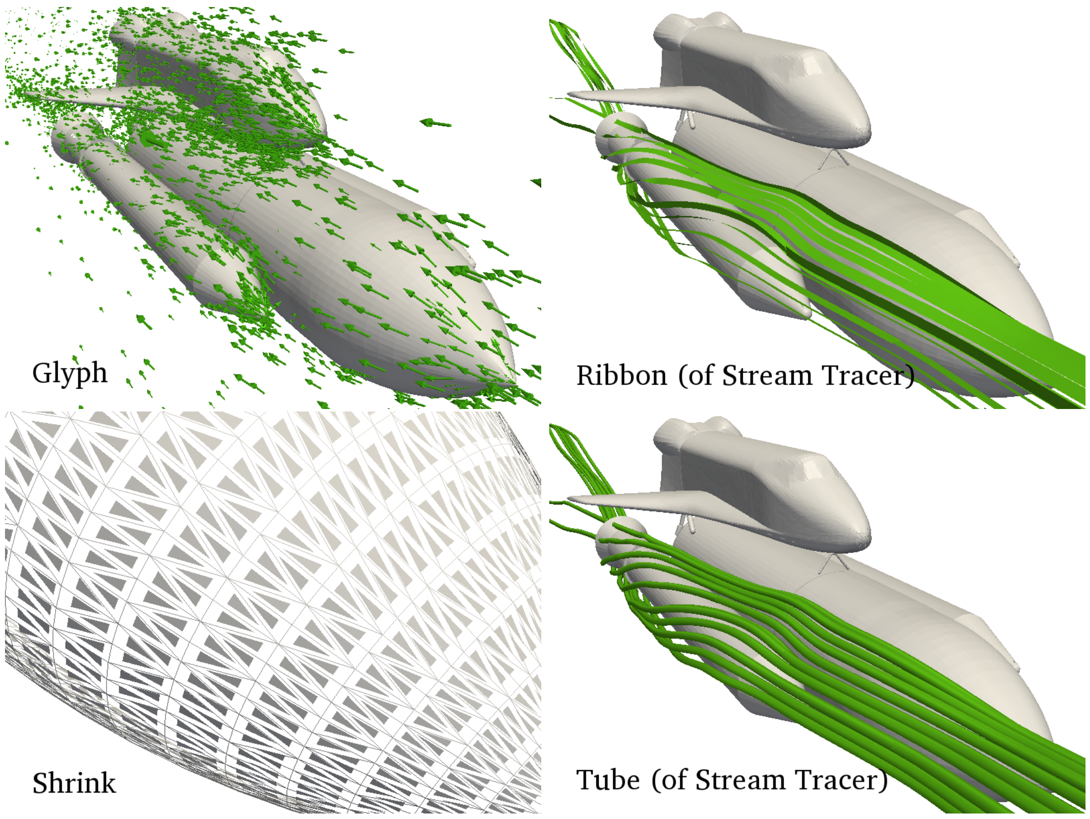


Figure 5: Examples of Build Independent Topology algorithms.

amples of Map by Cell are given in Figure 3.

Although parallel threads generally do access overlapped portions of the input data, the calculations computed by each thread are independent and there is no overlap in the output they produce and are thus easy to parallelize. The EAVL library and Dax toolkit each provide a generic algorithm to perform Map by Cell [26, 29].

Reconnect Cell

Separable Element	Cell
Point Mapping	1 to 0 or 1
Cell Mapping	1 to 0 or more
Field Mapping	Identity
Collective Work	None
Algorithms	Extract Cells by Region, Extract Selection, Mask Points, Tetrahedralize, Threshold, Triangulate

There are several apparent patterns in algorithms that build a topological structure, the first of which is Reconnect Cell. The characteristics of these algorithms are that the output topology uses the same points as (or a subset of the points from) the input topology and that each output cell depends on exactly one input cell. In essence the connectivity of each cell is being redefined. Some examples of Reconnect Cell algorithms are given in Figure 4.

Because each thread creates an independent list of cell connections, Reconnect Cell algorithms have no interdependence. This makes them very similar to Map by Cell algorithms with the important exception that most Reconnect Cell algorithms have a variable number of outputs across the threads that is not known at the onset of execution. Thus, these algorithms must implement some form of *stream compaction* to build efficient packed arrays for the output. Also, since some of these algorithms only use a subset of the input points, it may be desirable to explicitly identify and extract these points.

Each of the three SDAV many-core frameworks for visualization [37] provides an example of Threshold, a Reconnect Cell algorithm, using a different approach. PISTON uses a stream compaction algorithm to determine an efficient output layout and then generates the output cells in parallel [22]. New points are created in the output, which implicitly removes points from the input at the expense of replicating points in the output. Dax uses a similar stream compaction but outputs a more compact array of connection identifiers [24]. Dax can also optionally extract the subset of points used in the output at an added computational expense. EAVL provides a different approach wherein the output data structure references the input structure and the data management makes this behave as an independent data structure [25]. This approach is much faster and memory efficient, but there is no explicit representation of the result for use in other packages that might expect that and it is not possible to remove the input structure from memory as long as the output still references it.

Build Independent Topology

Separable Element	Any
Point Mapping	1 element to many points
Cell Mapping	1 element to many cells (constant number)
Field Mapping	Identity
Collective Work	None
Algorithms	Glyph, Ribbon, Shrink, Tube

Algorithms that Build Independent Topology create new geometry requiring both new points and new cells that connect these points. The geometry created by each thread in this algorithm class is completely independent from that created in any other thread; there are no topological connections between them.

For example, consider the Glyph filter. Each thread in this filter produces a small 3D object, like a scaled sphere or oriented arrow, centered at the point assigned to the thread. Each 3D object is completely disconnected (topologically) from its brethren and thus can be created independently and concurrently. Another example is the Shrink filter, which contracts each cell toward its local center creating new points and intentionally breaking connections. These and other examples of Build Independent Topology algorithms are shown in Figure 5.

The behavioral characteristics of the Build Independent Topology algorithms are very similar to the Basic Mapping and Map by Cell algorithms. The only meaningful difference between them is the interpretation of the output data. Instead of producing a single or set of field values, Build Independent Topology algorithms produce topologies of cells, vertices, and point coordinates; these data must be interpreted and managed as such.

Build Connected Topology

Separable Element	Cell
Point Mapping	1 cell to 0 or more points
Cell Mapping	1 to 0 or more
Field Mapping	Interpolated points
Collective Work	Resolve duplicate points
Algorithms	Clean, Clip, Contour, Extrusion, Isovolume, Slice, Subdivision, Tessellate

The most technically complicated class of algorithms we find that generate geometry are those that Build Connected Topology. Some examples are given in Figure 6. These algorithms create new geometry requiring both new points and new cells that connect these points. The points created by each thread are coincident with points created by another thread. These coincident points represent connections across elements created by different threads; capturing these connections creates an interdependence across threads.

The easiest solution to this problem is to simply ignore it by letting coincident points exist and losing the topological connections. Thus, the output of the algorithm is topologically a set of disconnected cells, often referred to as a *soup*. We find this approach common, particularly in implementations of Contour for GPUs [11, 19, 22, 30].

Producing a soup of cells may be an acceptable solution in some situations, particularly if the result is to be rendered and then forgotten, but can also be a problematic solution. The first problem is that this redundancy causes an inflation of the memory required. The second problem is that subsequent algorithms, such as estimating smooth normals on a contour, might require the lost topological connections. Existing tools, particularly those built on VTK [36], expect chains of algorithms of this nature to work.

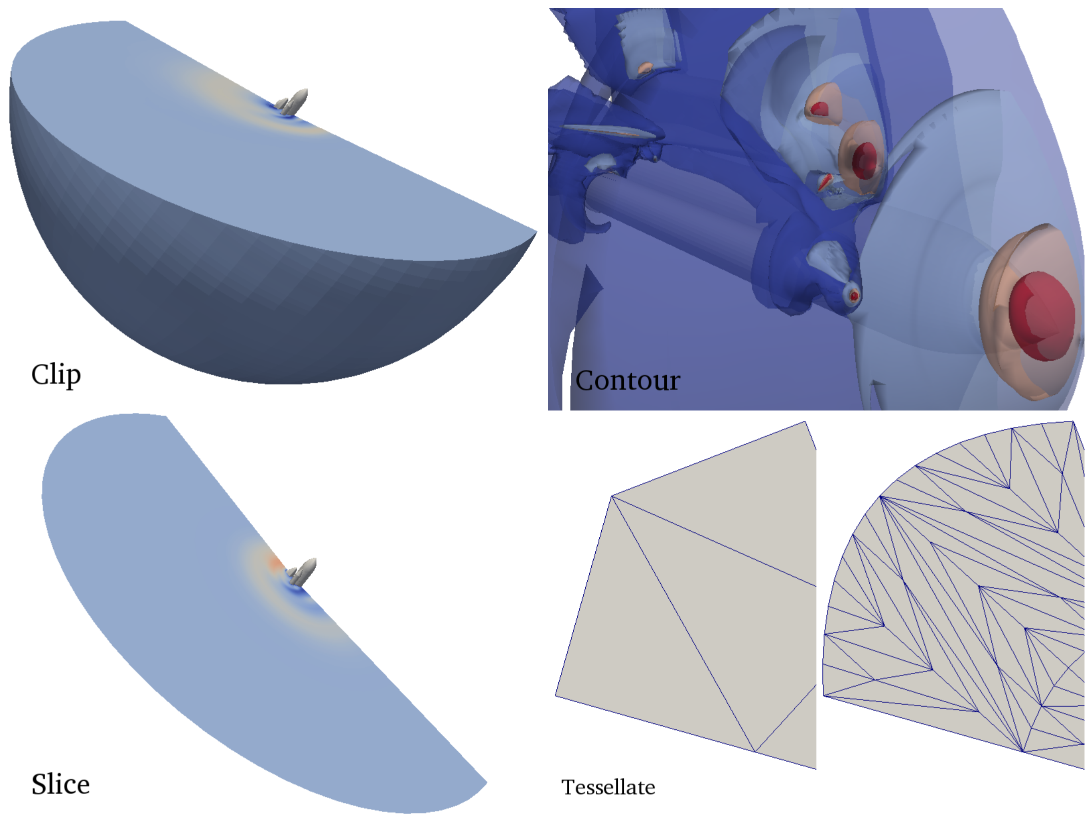


Figure 6: Examples of Build Connected Topology algorithms.

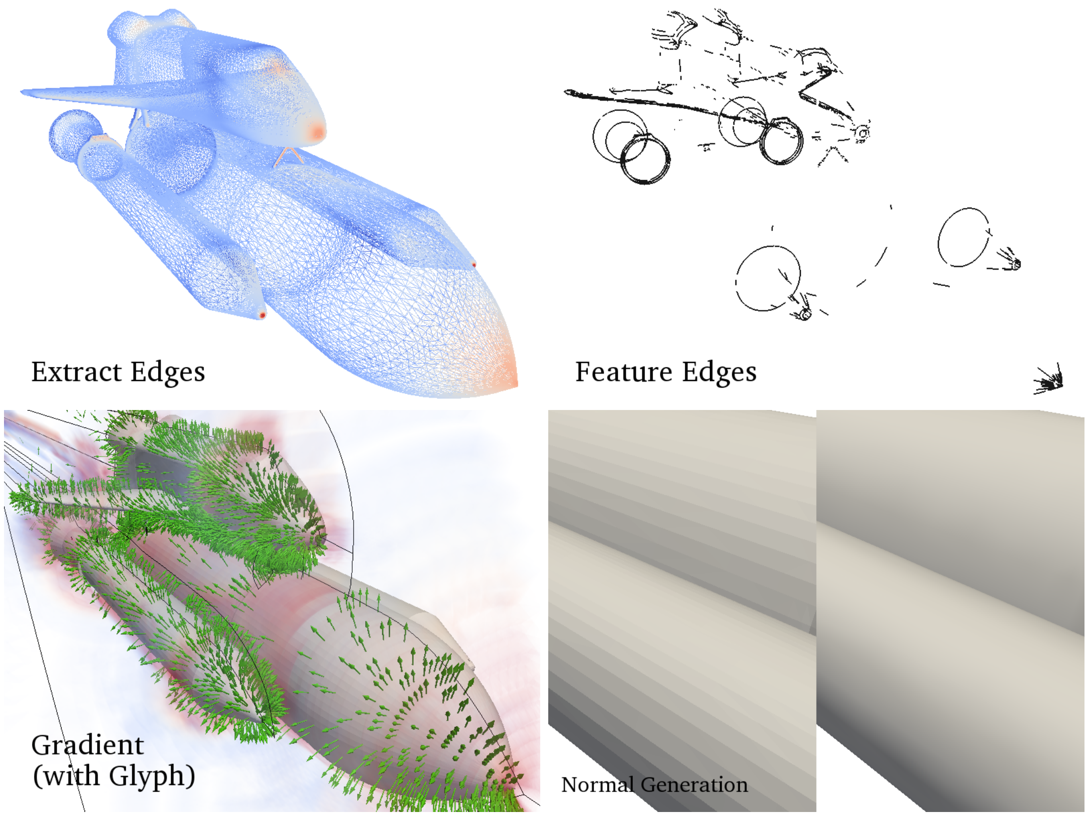


Figure 7: Examples of Capture Cell Adjacencies algorithms.

A typical approach to finding coincident points in a serial algorithm is to use an iterative *locator* structure to find any coincident points that are already created [1]. However, this approach is unlikely to work in a massively threaded environment as constant updates to the locator will require far too much synchronization. Bell [6] informally proposes a *vertex welding* algorithm that can efficiently find these coincident points on massive threading after the fact, but a solution working more closely to the algorithm might have better results.

Capture Cell Adjacencies

Separable Element	Point, edge, or face
Point Mapping	Identity
Cell Mapping	Identity
Field Mapping	Interpolated incident fields
Collective Work	Find incidence relationships
Algorithms	Curvature, Cell Data to Point Data, Extract Edges, Extract Surface (external faces), Feature Edges, Gradient, Normal Generation, Smooth

The previously listed algorithms all operate on a single point in the mesh or on a single cell and its incident features. These data are quickly locatable in general data structures. However, some algorithms need to Capture Cell Adjacencies. They typically operate by considering the points, edges, or faces in the mesh and examining the cells incident to them. Some examples of such algorithms are given in Figure 7.

In some types of data, particularly structured data with implicit topologies, enumerating the mesh elements and finding incident cells is trivial. However, in unstructured data represented by cell connection lists, this information is not explicitly stored. Thus, although the computation of Capture Cell Adjacencies algorithms are completely independent, they might require a collective operation to identify the domain of the input each thread needs.

Serial VTK algorithms support cell adjacencies by building a *links* array that lists which cells each point is incident on [1]. A similar array could be used in a massively threaded environment, but since a links array is not always available, we first need an efficient massively threaded algorithm to build one.

An alternate approach is to use a different cell representation that does explicitly capture these incidence relationships. Such alternate representations include half-edge structures [18], cellular data structures [3], and circular incident edge lists [21]. The drawback to these linked representations is that storing links to every desired incidence relationship is costly. Also, most existing data representations, including CGNS [34], VTK [36], and XDMF to name a few, use cell connection lists, so a links array would likely need to be built to convert between the representations anyway.

Globally Reduce

Separable Element	Any
Point Mapping	None in output
Cell Mapping	None in output
Field Mapping	All to 1
Collective Work	Global reduction
Algorithms	Histogram, Integrate, Outline, Statistics

Some scientific visualization algorithms perform an aggregation on the data, often performing an operation like sum or average on a field or derived field in the data. Thus, these algorithms Globally Reduce data into a single value or small set of values.

Reduction is a common operation in general parallel computing, and a reduction operation is ubiquitously supported across parallel computing environments [7, 31, 32, 35, 38]. Consequently, once the reduction operation can be expressed in terms of an associative binary operation, implementing the reduction is straightforward. The reduction is complicated slightly on a hybrid parallel machine with multiple many-core machines clustered with an interconnect, but the reduction can still be performed by first reducing locally within the shared-memory many-core machine and then reducing those values across the distributed-memory interconnect [10].

Query Data

Separable Element	Point, key, or query
Point Mapping	Identity
Cell Mapping	Identity
Field Mapping	1 query to 1 output
Collective Work	Building query structure
Algorithms	Particle Tracer, Probe Location, Quadric Clustering, Resample with Dataset, Stream Tracer, Streaklines

Most scientific visualization algorithms operate on a domain that is easily identifiable by the enumeration of the work (i.e. the point or cell identifier used to number threads). However, some algorithms must Query Data to find a region of interest. Typically this means finding a cell containing a given coordinate in space. This is part of the fundamental operation of algorithms such as those based on resampling and particle advection. (Note that query only solves the challenge of finding a single integration step in particle advection, shown in Figure 8. It also needs to iterate, to connect successive computations, and to find terminations, but these can be straightforward extensions of existing algorithms.) Although spatial queries are common, queries for a more general set of attributes can also be needed when performing query-based visualization [13–15, 33].

Some queries, such as a spatial query on a uniform rectilinear grid, are straightforward, but many will require a

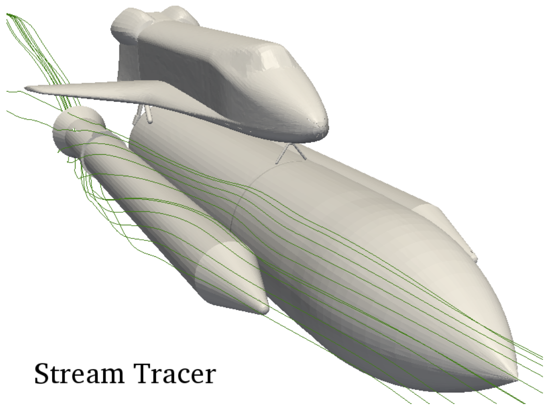


Figure 8: An example of a Query Data algorithm.

Table 1: Overview of visualization algorithm classifications

Name	Separable Element	Point Mapping	Cell Mapping	Field Mapping	Collective Work	Example Algorithm
Basic Mapping	Any	Identity	Identity	1 to 1	None	Generate Ids
Map by Cell	Cell	Identity	Identity	Points on cell to cell	None	Cell Centers
Reconnect Cell	Cell	1 to 0 or 1	1 to 0 or more	Identity	None	Threshold
Build Independent Topology	Any	1 element to many points	1 element to many cells	Identity	None	Glyph
Build Connected Topology	Cell	1 cell to 0 or more points	1 to 0 or more	Interpolated points	Resolve duplicate points	Contour
Capture Cell Adjacencies	Point, edge, or face	Identity	Identity	Interpolated incident fields	Find incidence relationships	Normal Generation
Globally Reduce	Any	None in output	None in output	All to 1	Global reduction	Histogram
Query Data	Point, key, or query	Identity	Identity	1 query to 1 output	Building query structure	Stream Tracer

specialized search structure. Several spatial search structures are proposed [12, 16, 17, 40, 41], most for the purpose of rendering, but there is yet little research on the building and using of query structures on massive threading for the purposes of scientific visualization.

Remaining Algorithms

The previous classification of algorithms contains the majority of algorithms in the ParaView application. There are some algorithms we have left out of this list for a variety of reasons, which we capture here.

A very small number of algorithms have complicated interaction that appears to be unique to the nature of the operation. The Connectivity algorithm needs to traverse large neighborhood regions to find connected components. The Delaunay algorithm needs to find all circles or spheres containing exactly 3 or 4 points, respectively. The Halo Finder algorithm identifies collections of points in mutual proximity. The Decimate algorithm reduces the size of geometry by removing points based on an error metric. The Triangle Strips algorithm finds connected strips in triangle meshes.

There may be commonalities among some of these algorithms, but we are unsure if these similarities extend to shared solutions for interdependence. Regardless, there are few algorithms with unique thread interaction, so addressing these independently is much more tractable than reimplementing every algorithm in existence today.

We have left out one or more classes involving imaging algorithms. These include algorithms involving kernel convolutions and FFT. Although this is generally an important class of algorithms, it is not one that is extensively used by ParaView users. The field of imaging algorithms is very broad and of interest across many disciplines. It deserves its own study, and thus we consider it out of the scope of this paper.

Another algorithm we have left out is Extract Subset, which extracts a region of interest and optionally regularly subsamples structured data. The operation is similar to that of Reconnect Cell except that the basic implementation of copying subarrays is so simplistic we omit it.

There are several algorithms in ParaView that do not translate to running on massive threads. Some of these algorithms do trivial metadata manipulations like extracting a block from an assembly. Other algorithms perform ordinary operations while making temporal adjustments [8] or other changes to pipeline execution management [28]. Also, although the Distributed Data Decomposition (D3) and similar redistribution algorithms can be compute intensive, their data partitioning makes little sense on the fine level dictated by massive threading.

Finally, to avoid being overly pedantic, we omit several algorithms that are variations of algorithms listed or specialized combinations of algorithms listed. For example, ParaView contains seven different versions of Generate Ids where each extracts information from different enumerations. ParaView also has four different versions of Slice, which have some optimizations for different data structures. There are also several algorithms such as Material Interface and Surface Flow that are each basically a conglomeration of other algorithms listed here. It would be unnecessarily distracting to enumerate each of these variations.

4. CONCLUSION

Although we have made progress in designing massively threaded visualization algorithms for scientific visualization, we have a long way to go to support production visualization at extreme scale. Our current scientific and visualization needs incorporate the inclusion of many algorithms including those designed specifically for a specialized scientific domain. We as a community simply do not have the resources to independently address all of these needs.

The development of our current petascale visualization tools is made possible by observations that allow us to design a general parallel capability that is then shared among many visualization categories. If we cannot find a similar way to localize and reuse the parallel portion of our algorithms, we cannot succeed in supporting the next generation of parallel computing.

In this paper we have identified eight categories of shared features among these algorithms summarized in Table 1. By addressing the group of algorithms in each category as one, we can quickly achieve our goal of supporting extreme scale computing.

5. ACKNOWLEDGMENTS

This work was supported in part by the DOE Office of Science, Advanced Scientific Computing Research, under award number 10-014707, program manager Lucy Nowell.

This work was supported in part by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. 12-015215, through the Scientific Discovery through Advanced Computing (SciDAC) Institute of Scalable Data Management, Analysis and Visualization.

Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration.
SAND 2013-7493C

6. REFERENCES

[1] *The VTK User's Guide*. Kitware Inc., 11th edition, 2010. ISBN 978-1-930934-23-8.

[2] J. Ahrens, K. Brislawn, K. Martin, B. Geveci, C. C. Law, and M. Papka. Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications*, 21(4):34–41, July/August 2001.

[3] T. J. Alumbaugh and X. Jiao. Compact array-based mesh data structures. In *Proceedings, 14th International Meshing Roundtable*, pages 485–504, September 2005.

[4] U. Ayachit et al. *The ParaView Guide: A Parallel Visualization Application*. Kitware Inc., 4th edition, 2012. ISBN 978-1-930934-24-5.

[5] C. G. Baker, M. A. Heroux, H. C. Edwards, and A. B. Williams. A light-weight API for portable multicore programming. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 601–606, February 2010. DOI 10.1109/PDP.2010.49.

[6] N. Bell. High-productivity CUDA development with the thrust template library, 2010.

[7] N. Bell and J. Hoberock. *GPU Computing Gems, Jade Edition*, chapter Thrust: A Productivity-Oriented Library for CUDA, pages 359–371. Morgan Kaufmann, October 2011.

[8] J. Biddiscombe, B. Geveci, K. Martin, K. Moreland, and D. Thompson. Time dependent processing in a parallel pipeline architecture. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1376–1383, November/December 2007. DOI 10.1109/TVCG.2007.70600.

[9] H. Childs, B. Geveci, W. Schroeder, J. Meredith, K. Moreland, C. Sewell, T. Kuhlen, and E. W. Bethel. Research challenges for visualization software. *IEEE Computer*, 46(5):34–42, May 2013. DOI 10.1109/MC.2013.179.

[10] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. Hybrid parallel programming with MPI and unified parallel C. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, pages 177–186, 2010. DOI 10.1145/1787275.1787323.

[11] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel. High-speed marching cubes using HistoPyramids. *Computer Graphics Forum*, 27(8):2028–2039, 2008. DOI 10.1111/j.1467-8659.2008.01182.x.

[12] T. Foley and J. Sugerman. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '05)*, pages 15–22, 2005. DOI 10.1145/1071866.1071869.

[13] M. Glatter, J. Huang, S. Ahern, J. Daniel, and A. Lu. Visualizing temporal patterns in large multivariate data using textual pattern matching. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1467–1474, November/December 2008. DOI 10.1109/TVCG.2008.184.

[14] L. J. Gosink, J. C. Anderson, E. W. Bethel, and K. I. Joy. Query-driven visualization of time-varying adaptive mesh refinement data. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1715–1722, November/December 2008.

[15] C. R. Johnson and J. Huang. Distribution-driven visualization of volume data. *IEEE Transactions on Visualization and Computer Graphics*, 15(5), September/October 2009. DOI 10.1109/TVCG.2009.25.

[16] J. Kalojanov, M. Billeter, and P. Slusallek. Two-level grids for ray tracing on GPUs. *Computer Graphics Forum*, 30(2):307–314, April 2011. DOI 10.1111/j.1467-8659.2011.01862.x.

[17] J. Kalojanov and P. Slusallek. A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics*, pages 23–28, 2009. DOI 10.1145/1572769.1572773.

[18] L. Kettner. Designing a data structure for polyhedral surfaces. In *Proceedings of the Fourteenth ACM Symposium on Computational Geometry*, pages 146–154, 1998. DOI 10.1145/276884.276901.

[19] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. In *Proceedings of the 12th Pacific Conference on Computer Graphics and Applications (PG'04)*, pages 186–195, October 2004. DOI 10.1109/PCCGA.2004.1348349.

[20] C. C. Law, K. M. Martin, W. J. Schroeder, and J. Temkin. A multi-threaded streaming pipeline architecture for large structured data sets. In *Proceedings of IEEE Visualization 1999*, pages 225–232, October 1999.

[21] B. Lévy, G. Caumon, S. Conreaux, and X. Cavin. Circular incident edge lists: a data structure for rendering complex unstructured grids. In *Proceedings of IEEE Visualization*, pages 191–198, October 2001.

[22] L.-T. Lo, C. Sewell, and J. Ahrens. PISTON: A portable cross-platform framework for data-parallel

visualization operators. Technical Report LA-UR-12-10227, Los Alamos National Laboratory, 2012.

[23] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics (Proceedings of SIGGRAPH 87)*, 21(4):163–169, July 1987.

[24] R. Maynard, K. Moreland, U. Ayachit, B. Geveci, and K.-L. Ma. Optimizing threshold for extreme scale analysis. In *Visualization and Data Analysis 2013, Proceedings of SPIE-IS&T Electronic Imaging*, February 2013.

[25] J. S. Meredith, S. Ahern, D. Pugmire, and R. Sisneros. EAVL: The extreme-scale analysis and visualization library. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 21–30, 2012. DOI 10.2312/EGPGV/EGPGV12/021-030.

[26] J. S. Meredith, R. Sisneros, D. Pugmire, and S. Ahern. A distributed data-parallel framework for analysis and visualization algorithm development. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5)*, pages 11–19, March 2012. DOI 10.1145/2159430.2159432.

[27] K. Moreland. The ParaView tutorial, version 4.0. Technical Report SAND 2013-6883P, Sandia National Laboratories, 2013.

[28] K. Moreland. A survey of visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):367–378, March 2013. DOI 10.1109/TVCG.2012.133.

[29] K. Moreland, U. Ayachit, B. Geveci, and K.-L. Ma. Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, pages 97–104, October 2011. DOI 10.1109/LDAV.2011.6092323.

[30] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In *Proceedings of the Sixth Joint Eurographics - IEEE TCVG conference on Visualization*, pages 293–300, 2004.

[31] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004. ISBN 978-0-07-282256-4.

[32] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007. ISBN 978-0-596-51480-8.

[33] O. Rübel, Prabhat, K. Wu, H. Childs, J. Meredith, C. G. Geddes, E. Cormier-Michel, S. Ahern, G. H. Weber, P. Messmer, H. Hagen, B. Hamann, and E. W. Bethel. High performance multivariate visual data exploration for extremely large data. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, November 2008.

[34] C. L. Rumsey, D. M. A. Poirier, R. H. Bush, and C. E. Towne. A user's guide to cgns. Technical Report TM-2001-211236, NASA, October 2001.

[35] J. Sanders and E. Kandrot. *CUDA by Example*. Addison Wesley, 2011. ISBN 978-0-13-138768-3.

[36] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*. Kitware Inc., fourth edition, 2004. ISBN 1-930934-19-X.

[37] C. Sewell, J. Meredith, K. Moreland, T. Peterka, D. DeMarle, L.-T. Lo, J. Ahrens, R. Maynard, and B. Geveci. The SDAV software frameworks for visualization and analysis on next-generation multi-core and many-core architectures. In *2012 SC Companion (Proceedings of the Ultrascale Visualization Workshop)*, pages 206–214, November 2012. DOI 10.1109/SC.Companion.2012.36.

[38] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998. ISBN 0-262-69215-5.

[39] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[40] K. Zhou, M. Gong, X. Huang, and B. Guo. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):669–681, May 2011. DOI 10.1109/TVCG.2010.75.

[41] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5), December 2008. DOI 10.1145/1409060.1409079.