Flying Edges: A High-Performance Scalable Isocontouring Algorithm

Conference Paper · October 2015

DOI: 10.13140/RG.2.1.3415.9609

3 authors:

William J. Schroeder
Kitware, Inc.
65 PUBLICATIONS **7,279** CITATIONS
SEE PROFILE

Robert Maynard
Kitware, Inc.
9 PUBLICATIONS **10** CITATIONS
SEE PROFILE

Berk Geveci
Kitware, Inc.
41 PUBLICATIONS **723** CITATIONS
SEE PROFILE

Flying Edges: A High-Performance Scalable Isocontouring Algorithm

William Schroeder* Kitware, Inc.
Rob Maynard† Kitware, Inc.
Berk Geveci‡ Kitware, Inc.

Abstract

Isocontouring remains one of the most widely used visualization techniques. While a plethora of important contouring algorithms have been developed over the last few decades, many were created prior to the advent of ubiquitous parallel computing systems. With the emergence of large data and parallel architectures, a rethinking of isocontouring and other visualization algorithms is necessary to take full advantage of modern computing hardware. To this end we have developed a high-performance isocontouring algorithm for structured data that is designed to be inherently scalable. Processing is performed completely independently along edges over multiple passes. This novel algorithm also employs computational trimming based on geometric reasoning to eliminate unnecessary computation, and removes the parallel bottleneck due to coincident point merging. As a result the algorithm performs well in serial or parallel execution, and supports heterogeneous parallel computation combining data parallel and shared memory approaches. Further it is capable of processing data too large to fit entirely inside GPU memory, does not suffer additional costs due to preprocessing and search structures, and is the fastest non-preprocessed isocontouring algorithm of which we are aware on shared memory, multicore systems. The software is currently available under a permissive, open source licence in the VTK visualization system.

Index Terms: I.3.1 [Computing Methodologies]: Computer Graphics—Parallel processing; I.3.5 [Computing Methodologies]: Computational Geometry and Object Modeling—Geometric algorithms

1 Introduction

The current computing era is notable for its rapidly increasing data size, and the rapid evolution of computing systems towards massively parallel architectures. The increasing resolution of sensors and computational models has driven data size since the earliest days of computing, and is now reaching the point where parallel approaches are absolutely essential to processing the resulting large data. However, despite the widespread acknowledgement that parallel methods are essential to future computational advances, many important visualization algorithms in use today were developed with serial or coarse-grained data parallel computing models in mind. Such algorithms are generally not able to take effective advantage of emerging massively parallel systems, and therefore struggle to scale with increasing data size. Towards this end, we are challenging ourselves to rethink approaches to visualization algorithms with the goal of better leveraging emerging parallel hardware and systems. In this paper we revisit isocontouring, one of the most important and well-studied visualization techniques in use today.

*e-mail: will.schroeder@kitware.com
†e-mail:robert.maynard@kitware.com
‡e-mail:berk.geveci@kitware.com

1.1 Considerations

Modern computational hardware is moving towards massive parallelism. There are a number of reasons for this including physical constraints from power consumption and frequency scaling concerns [13, 25] which limit the speed at which a single processor can run. At the same time chip densities continue to increase, meaning that manufacturers are adding hardware in the form of additional computing cores and supporting infrastructure such as memory cache and higher-speed data buses. The end result, however, is that computing systems are undergoing dramatic change, requiring algorithmic implementations to evolve and reflect this new computational reality.

Taking advantage of massive parallelism places particular burdens on software implementations. Typically data movement is discouraged, requiring judicious use of local memory cache and management of the memory hierarchy. Computational tasks and data structures are simplified so that they may be readily pushed onto pipelined hardware architectures. Conditional branching is discouraged as this tends to result in idle processing cores and interrupts concurrent execution; and bottlenecks due to I/O contention significantly inhibit potential speed up. In general, an algorithm performs best if it is implemented with simple computational threads (even if this means extra work is performed); executes independently across local, cached memory; and reads and writes data from pre-allocated memory locations avoiding I/O blocking whenever possible.

We believe that it is important to reimagine essential visualization algorithms with this new computing reality in mind. While efforts are underway to develop and extend programming languages which can automatically parallelize algorithms, despite decades of effort only limited success has been demonstrated [26], and today's computational challenges require solutions now. One especially important visualization algorithm is isocontouring, which reveals features of underlying scalar fields. However, despite years of research, many implementations of this technique have not been designed in the context of massive parallelism, and are hobbled by excessive data movement and related bottlenecks, or serial execution dependencies. For example, naive implementations of the popular Marching Cubes (MC) [18] algorithm process interior voxel edges four times, and visit interior data points eight times. In addition, merging coincident points often introduces a serial bottleneck (due to the use of a spatial or edge-based locator), and the output of the algorithm is data dependent, meaning that arrays must be dynamically allocated and that there is no clear mapping of input to output data, resulting in yet another parallel processing bottleneck.

1.2 Motivation and Related Work

As isocontouring is one of the most useful and important visualization techniques, there is a vast collection of literature addressing a variety of different techniques. The publication of the MC algorithm produced significant productive research of which [3, 32] are representative examples; more recently the introduction of GPGPU computation has produced many excellent highly parallel methods [8, 11, 15, 20, 21, 22]. Despite these many advances, however, we desired to create a general algorithm amenable to parallelization on a variety of hardware systems (e.g., CPU and GPU) with the ability to handle large data, especially given that we often

encounter data that is too large to fit on a single GPU (given current technology). We also wanted to investigate methods to remove parallel computing bottlenecks in MC, which while easily parallelized, suffers from the inefficiencies noted previously.

Our visualization workflow is also different than what is often assumed in much of the literature. While many existing isocontouring algorithms are used in an exploratory manner to reveal the structure of scalar fields, our usual approach is to process very large data using known isovalues, with the goal of generating the corresponding isosurfaces as fast as possible, avoiding the extra costs of building and storing supplemental acceleration structures. For example, medical CT imaging techniques produce data in which known isovalues correspond to different biological structures such as skin and bone. Similarly, in computational fluid dynamics various functions such as mass density or Q-criterion are well understood and previous experience typically suggests appropriate isovalues. It is not uncommon for our datasets to exceed GPU memory, thereby incurring significant transfer overhead across the associated PCIE bus, as we often process large volumes on the order of 2048^3 resolution with double precision scalar values (e.g., ∼9GB for 1024^3 doubles, or ∼68GB for 2048^3 doubles). Thus many of the exploratory isocontouring techniques, which are typically based on preprocessing the data to build a rapid search structure such as an octree [30], interval tree [7], or span space [17], are not suitable to our workflow. Such preprocessing steps may add significant time to what is a non-exploratory workflow, while also introducing complex, auxiliary data structures which often consume significant memory and/or disk resources.

Out-of-core isosurfacing techniques [6, 29] may be problematic too. While these algorithms are designed to process data that is much larger than machine memory, the cost of I/O (e.g., writing out entire solutions) at extreme scale can be prohibitive [19, 25]. Instead, we often revert to in situ processing [2], meaning that visualization algorithms are executed alongside the running simulation or data acquisition process. The benefit of this approach is that expensive I/O can be significantly reduced by extracting key features such as isosurfaces, slices, streamlines, or other solution features, each of which is significantly smaller than the full dataset. Using such methods, researchers can intelligently instrument (and conditionally program) their simulation runs to produce meaningful feature extracts that focus in on key areas of interest, avoiding the need to write out an entire output dataset [1]. Hence simple isocontouring algorithms that can be easily executed in situ with the simulation are essential for larger data.

Given these considerations, the motivating goal of this work was to develop a fast, scalable isocontouring algorithm requiring no preprocessing or additional I/O. We also challenged ourselves to develop simple, independent algorithmic tasks that would scale well on a variety of different types of massively parallel hardware.

2 Algorithm

In this section we begin by providing a high-level overview of the Flying Edges (FE) algorithm, including introducing descriptive notation. Next we address key algorithmic features. Finally we address key implementation details.

2.1 Notation

The algorithm operates on a structured grid with N scalar values arranged on a topologically regular lattice of data points of dimensions $(l \times m \times n)$ with values s_{ijk}. Grid cells are defined from the eight adjacent points associated with scalar values in the $i+(0,1), j+(0,1), k+(0,1)$ directions. Grid edges E_{ijk} run in the row E_{jk}, column E_{ik}, and stack E_{ij} directions; and are composed of the cell edges e_{ijk}; grid cell rows R_{ijk} consist of all cells v_{ijk} touching both E_{ijk} and $E_{(i+1)(j+1)(k+1)}$. So for example, an x-row edge

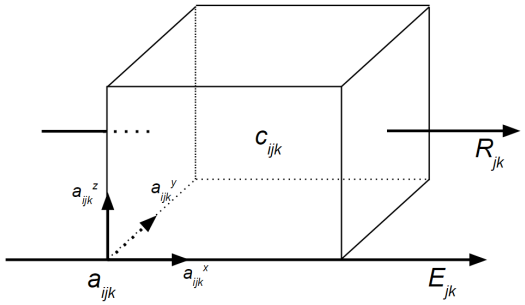

Figure 1: The cell axes a_{ijk} coordinate system. The algorithm processes the a_{ijk} in parallel along x-edges E_{jk} to count intersections and the number of output triangles. First and last intersections along E_{jk} are used to perform computational trimming when generating output in parallel over the grid cell rows R_{jk}.

of length n with $0 \leq i < n$:

$$E_{jk} = \bigcup_{i=0}^{n-1} e_{ijk} \quad \text{and} \quad R_{jk} = \bigcup_{i=0}^{n-1} v_{ijk}. \tag{1}$$

Each v_{ijk} has an associated cell axes a_{ijk} which refers to the three cell edges emanating from the point located at s_{ijk} in the positive row x, column y, and stack z directions. Refer to Figure 1.

The purpose of the algorithm is to generate an approximation to the isocontour Q. The isocontour is defined by an isovalue q : $Q(q) = \{p \mid F(p) = q\}$ where $F(p)$ maps the point $p \in R^n$ to a real-valued number (i.e., the isovalue q). The resulting approximation surface S is a continuous, piecewise linear surface such as a triangle mesh (in 3D). Note also that we say that Q intersects e_{ijk} when some p lies along the cell edge e_{ijk}. We assume that the scalar value varies linearly over e_{ijk} so Q may intersect at most at only one point along the edge.

2.2 Overview

Highly-organized structured data lends itself to a variety of parallel approaches. Our method, which is based on the independent processing of grid edges, provides a number of benefits:

- The algorithm takes advantage of cache locality by processing data in the fastest varying data direction (i.e., assuming that an $i-j-k$ grid varies fastest in the i-direction, the edges oriented along the i data rows or based on the notation above, the voxel cell x-edges e_{jk});
- it separates data processing into simple, independent computational steps that eliminate memory write contention and most serial, synchronous operations;
- it reduces overall computation by performing a small amount of initial extra work (i.e., computational trimming) to limit the extent of subsequent computation;
- it ensures that each voxel edge is only intersected once by using the cell axes a_{ijk} to control iteration along the grid edges E_{jk};
- it eliminates the need for dynamic memory reallocation, output memory is allocated only once;
- and the algorithm eliminates the parallel bottlenecks due to point merging.

While others have used edge-based approaches to characterize the quality of isocontour meshes [10], or to ensure watertight meshes using octree edge-trees [16], we use our edge-based approach as an organizing structure for parallel computation; both as a means of traversing data as well as creating independent computational tasks.

One of the most challenging characteristics of any isocontouring algorithm is that it produces indeterminate output, i.e., the number

of output points and primitives (e.g., triangles) is not known a priori. This presents a challenge to parallel implementations since processing threads work best when output memory can be preallocated and partitioned to receive output. So for example algorithms such as MC–which on first glance seem embarrassingly parallel–suffer significant performance degradation when inserting output entities into dynamic output lists due to repeated memory reallocation. A natural way to address this challenge is to use multiple passes to first characterize the expected output by counting output points and primitives, followed by additional passes to actually generate the isocontour. The key is to minimize the effort required to configure the output. Our approach also takes advantage of the multi-pass approach to guide and dramatically reduce subsequent computation, thus the early passes can be considered as a form of preprocessing to set up later passes for optimized computation.

The algorithm is implemented in four passes: only the first pass traverses the entirety of the data, the others operate on the resulting local metadata or locally on rows to generate intersection points and/or gradients. At a high level, these passes perform the following operations:

1. Traverse the grid row-by-row to compute grid x-edge cases; count the number of x intersections; and set the row computation trim limits.
2. Traverse each grid cell row, between adjusted trim limits, and count the number of y- and z-edge intersections on the a_{ijk}, as well as the number of output triangles generated.
3. Sum the number of x-, y-, and z-points created across the a_{ijk} of each row, and the number of output triangles; allocate output arrays.
4. Traverse each grid cell row, between adjusted trim limits, and using the a_{ijk} generate points and produce output triangles into the preallocated arrays.

The first three passes count the number of output points and triangles, determine where the contour proximally exists and set computational trim values, and produce the metadata necessary to generate output. In the fourth and final pass, output points and primitives are generated and directly written into pre-allocated output arrays without the need for mutexes or locking. In the following we describe each pass in more detail and then follow with a discussion of key implementation concepts.

Pass 1: Process grid x-edges $\mathbf{E_{jk}}$. For each grid x-edge E_{jk}, all cell x-edge intervals (i.e., cell edges e_{jk}) composing E_{jk} are visited and marked when their interval intersects Q (i.e., an edge case number is computed for each e_{jk}). The left and right trim positions xL_{jk} and xR_{jk} are noted, as well as the number of x-intersections along E_{jk}. Each E_{jk} is processed independently and in parallel. (Note that the trim position xL_{jk} indicates where Q first intersects E_{jk} on its left side; and xR_{jk} indicates where Q last intersects E_{jk} on its right side. Additional trimming details will be provided shortly.)

Pass 2: Process grid rows $\mathbf{R_{jk}}$. For each grid row R_{jk}, the cells v_{ijk} between the adjusted trim interval $[\bar{x}L_{jk},\bar{x}R_{jk})$ are visited and their MC case numbers are computed from the edge-based case table from Pass 1 (the original s_{ijk} values are not reread). Knowing the cell case value c_{ijk} it is possible to perform a direct table lookup to determine whether the y- and z-cell axes edges a^y_{ijk} and a^z_{ijk} intersect the isocontour, incrementing the y- and z-cell intersection count as appropriate. In addition, the c_{ijk} directly indicates the number of triangles generated as the contour passes through v_{ijk}. Again, each R_{jk} can be processed in parallel. At the conclusion of the second pass, the number of output points and triangles is known and is used to allocate the output arrays in the next pass. Note that because the adjusted trim interval may be empty, it is possible to rapidly skip data (rows and entire slices) via computational trimming (see Figure 2).

Note that while the algorithm is designed to operate across x-edges due to it being the (typically) fastest varying data direction, it can be readily be recast using y- or z-edges. In such cases, the results remain invariant, as the generated MC cases (after combining the edge cases in the proper order) are equivalent.

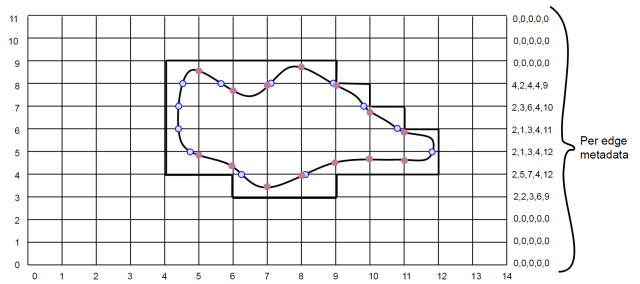

Figure 2: A trimmed contour after the second pass (in 2D). On the right side is a metadata array that tracks information describing the interaction of the contour with the E_{jk} edges, including the number of x- and y-edge intersections, the number of output primitives generated, and trim positions. Trimming can significantly reduce computational work.

Pass 3: Configure output. At the conclusion of Pass 2, for each R_{jk} the number of x-, y-, and z-point intersections is known, as well as the number of output triangles. The third pass accumulates this information on each E_{jk}, assigning starting ids for the x-, y-, and z-points and triangles to be generated in the final pass along each R_{jk}. Note that this accumulation pass can be performed in parallel as a prefix sum operation in $O(m/t+\log(t))$ where the m is the number of E_{jk} and t is the number of threads of execution [4]. (Using the scan or prefix sum operation is reminiscent of [11, 12] which use this operation to generate output offsets. Here the scan is used to accumulate offsets on a per-edge basis.)

Pass 4: Generate output across $\mathbf{R_{jk}}$. In the final pass, each R_{jk} is processed in parallel by moving the a_{ijk} along the row within the adjusted trim interval $[\bar{x}L_{jk},\bar{x}R_{jk})$, computing x, y, z edge intersection points on the cell axes as appropriate. This requires reading some of the s_{ijk} again to compute gradients (if requested) and interpolate edge intersections with Q. Triangles are also output (if necessary) as each cell is visited. Again, the adjusted trim edge interval is used to rapidly skip data, with the cell case values c_{ijk} indicating which (if any) points need be generated on the a_{ijk}. Because the previous pass determined the start triangle and point ids for each R_{jk}, an increment operator (based in edge use table $U(c_i)$) is used to update point and triangle ids as each cell is processed along the row (Figure 5). This eliminates the need to merge points, and points are only generated once and written directly into the previously allocated output arrays without memory write contention. It should be noted that the generation of intersection points and gradients, and the production of triangle connectivity, can proceed independently of one another for further parallel optimization.

Assuming that the grid is of dimension n^3, the algorithm complexity is driven by the initial pass over the grid values $O(n^3)$ invoking n^2/t total threads of execution. The second pass traverses the $x-$edge classifications, generally of size less than $O(n^3)$ due to computational trimming, also with n^2/t total thread executions. The third pass sums the number of output primitives as described previously in $O(n^2/t+\log(t))$. The fourth and final pass processes up to n^2/t threads, although trimming often reduces the total workload significantly.

3 Implementation

In the following section we highlight and discuss some of the implementation features of the Flying Edges algorithm.

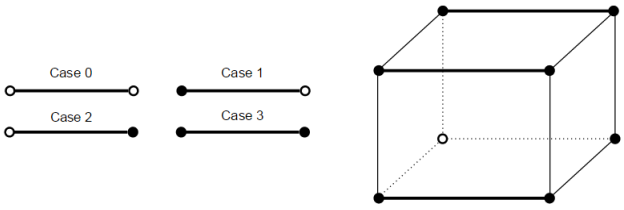

Figure 3: An edge-based case table combines four x-edge cases to produce an equivalent Marching Cubes vertex-based case table. Edge cases can be computed independently in parallel and combined when necessary to generate a MC case when processing R_{jk}.

3.1 Edge-Based Case Tables

Voxel-based contour case tables were popularized by MC and have been extended to other topological cell types such as tetrahedra [28]. Computing a case number c_{ijk} for a cell involves comparing the scalar field value at each cell vertex against the isocontour value q: $c_{ijk} = 1$ if $s_{ijk} \geq q$;0 otherwise; and then performing a shifted logical OR operation to determine a case value $0 \leq c_{ijk} < 256$. In naive implementations of MC, many unnecessary accesses to and comparisons against a particular vertex scalar value $\bar{s}$ occur as the algorithm marches from cell to adjacent cell. In Flying Edges, typically s_{ijk} are accessed only once unless it is necessary that a particular $\bar{s}$ is required for subsequent computation (e.g., gradient computation or edge interpolation). This is because the total number of data values N is usually much greater than the number of s_{ijk} which actually contribute to the computation of the isocontour.

Case computation is performed in parallel along each grid edge E_{jk}. The edge case for each e_{jk} that compose the E_{jk} is determined from the scalar values of its two end points, with 2^2 total states possible. The resulting case value e^c_{jk} indicates whether each cell x-edge intersects the isosurface, and if so (as described previously in Pass 1), the number of x-intersections is incremented by one. During subsequent processing, the four edge case values $\{e^c_{jk}, e^c_{(j+1)k}, e^c_{j(k+1)}, e^c_{(j+1)(k+1)}\}$ can be combined to determine the cell case value c_{ijk} (see Figure 3). The edge-based case table is equivalent to the standard MC table which considers the states of eight vertices as compared to four parallel cell x-edges. We use an edge-based case table because it removes computational dependencies which degrade parallel performance, allowing the computation of edge cases to proceed completely independently.

Further efficiencies in the algorithm result from exploiting the implicit topological information contained in each MC case c_i. Obviously, given a particular case number, the c_i defines the numbers of points and triangles that will be generated, and exactly which edges are involved in the production of the output primitives. An important construct is the edge use table $U(c_i)$ which for each c_i indicates which of the 12 voxel edges intersects Q. By using this information efficiently it is possible to rapidly perform operations such as counting the number of $y-$ and $z-$ edge intersections, and incrementing point and triangle ids across cell rows (during Pass 4) to generate crack-free isocontours without a spatial locator.

3.2 Computational Trimming

As used here, computational trimming is the process of precomputing information or metadata in such a way as to reduce the need for subsequent computation, thereby reducing the total computational load. More specifically, in the Flying Edges algorithm, computational trimming is used to significantly reduce the total effort necessary to generate the isocontour. It is possible to rapidly skip portions of rows, entire rows, and even entire data slices while computing. This is accomplished by taking advantage of the topological characteristics of the continuous, piecewise linear contour surface S, and using the first x-edge interval pass to bound the computations necessary to perform later passes (including interpolation and primitive generation). In practice, computational trimming significantly accelerates algorithm performance.

Basically computational trimming takes advantage of the information available from the initial pass in which x-edge intervals indicate something about the location of the isocontour. For example (Figure 4), if there are no e_{jk} intersections along any of the four E_{jk} that bound a particular cell row R_{jk}, then an isocontour exists in those cells along R_{jk} if and only if it passes through some y-edge (2D) and/or z-edge (3D). In such circumstances, because Q is continuous, a single intersection check with any y-edge (in 2D) or y-z cell face (3D) is enough to determine whether the isocontour intersects any part of R_{jk}. A similar argument can be made over contiguous runs of cells in R_{jk} in a manner reminiscent to run-length encoding. In our implementation, to reduce algorithmic complexity and memory overhead we chose to keep just two trim positions for each E_{jk}: the trim position on the left side where the isocontour first intersects E_{jk}; and the trim position on the right side where the isocontour last intersects E_{jk}.

While the first pass identifies intersected x-edges (i.e., generates edge case values) and determines trim positions along E_{jk}; subsequent passes process cell rows R_{jk} which (in 3D) are controlled by the four grid edges $\{E_{jk}, E_{(j+1)k}, E_{j(k+1)}, E_{(j+1)(k+1)}\}$. Thus to process cell rows, an adjusted trim interval $[\bar{x}L_{jk}, \bar{x}R_{jk})$ is computed where $\bar{x}L_{jk}$ is the leftmost trim position, and $\bar{x}R_{jk}$ is the rightmost trim position along the four edges defining R_{jk}. Additionally, intersection checks are made with the cell faces at the positions $\bar{x}L_{jk}$ and $\bar{x}R_{jk}$ to ensure that the isocontour is not running parallel to the four E_{jk}. If any intersection is found, the adjusted trim positions are reset to the leftmost ($\bar{x}L_{jk} = 0$) and/or rightmost ($\bar{x}R_{jk} = (n-1)$) locations (assuming the grid x-dimension is n), as the isocontour must run to the grid boundary.

3.3 Point Merging

In many isocontouring algorithms such as MC and its derivatives, each interior cell edge is visited four times (since in a structured grid four cells share an edge). If the edge intersects the isocontour Q, four coincident points will be generated, requiring additional work and resources to interpolate and then store the points. In implementation, it is common to use some form of spatial locator or edge-based hash table to merge these coincident points to produce a crack-free polygonal surface. While there are algorithmic concerns from the use of the locator (e.g., extra memory requirements) probably the bigger impact is due to the computational bottleneck introduced by such an approach in a parallel environment. Multiple threads feeding into a single locator requires data locking which negatively impacts performance. Some implementations address this issue by instantiating multiple locators, one in each subsetted

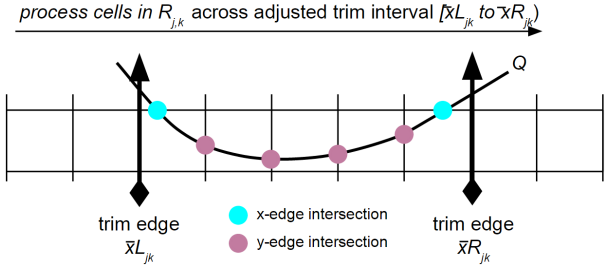

Figure 4: A 2D depiction of an adjusted trim edge interval. Shown are the trim position of intersection with the isocontour Q to the furthest left $\bar{x}L_{jk}$ and furthest right $\bar{x}R_{jk}$ on the cell row R_{jk}. The trim position, in combination with topological reasoning about the location of the continuous Q, is used to eliminate unnecessary computation.

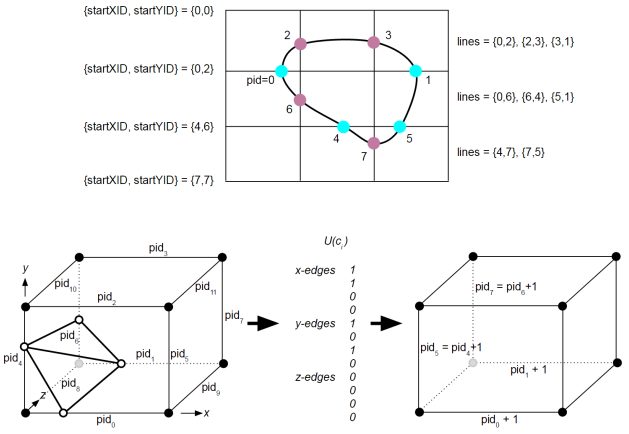

Figure 5: Processing a cell row R_{ijk} to generate output, eliminating point merging. At the beginning of each cell row, the edge metadata–determined from the prefix sum of Pass 3–contains the starting output triangle ids, and x-, y-, and z-point ids, and is used to initialize a cell row traversal process (illustrated at the top of the figure and shown in 2D). Then to traverse to the next adjacent cell along R_{ijk} the edge use table $U(c_i)$ is applied to increment the point ids appropriately. The a_{ijk} are used to generate the new points, while the other points are simply referenced as they will be produced in a separate thread. Newly generated points and triangles are directly written into previously allocated memory using their output ids.

grid block which are processed independently, which can then be merged at the conclusion of surface generation at the additional cost of a parallel, tree-based compositing operation.

Flying Edges eliminates all of these problems. First, because edges are processed by traversal of the cell axes a_{ijk} across R_{jk}, each of the x, y, z cell edges is interpolated only once, generating at most one point per cell edge. Secondly, because the number of triangles, and the number of x-edge, y-edge and z-edge intersection points (and hence point ids) is known along all cell rows at the outcome of the third pass, data arrays of the correct size can be preallocated to receive the output data. Further, as cell rows R_{jk} are processed in the fourth pass, the c_{ijk} implicitly defines how to increment each of the point ids as traversal moves from cell to neighboring cell along R_{jk}(Figure 5). Geometry (interpolated point coordinates) and topology (triangles defined by three point ids) can be computed independently and directly written to the preallocated and partitioned output. Thus a locator or hash table is not required, thereby eliminating the point merging bottleneck while producing a crack-free S across multiple threads.

3.4 Cell Axes on the Grid Boundary

The a_{ijk} are key to efficient iteration over the grid since they ensure that every cell edge is processed only one time. However, on the positive x, y, and z grid boundaries, the a_{ijk} are incomplete, requiring special handling. For example, when an x-edge terminates on the positive y-z grid face, the last point along the E_{jk} is not associated with any cell, and hence the a_{ijk} is undefined. Conceptually we create a partial a_{ijk} consisting of just the y and z edges and process them. Similar, but more complex situations occur for the E_{jk} located near the x-y and x-z grid boundaries. In implementation, dealing with these special cases adds some complexity to what is a relatively simple algorithm. Alternative ways to deal with this complexity is to pad the volume on the positive grid boundaries, or skip processing the last layer of cells on the positive grid boundaries.

3.5 Degeneracies

Degeneracies occur when one or more cell vertices take on the value of the isocontour. Much of the literature ignores degenerate situations due to the assumed rarity of such occurrences, but in practice they are common. For example, integral-valued or limited precision scalar fields are finite in extent, and/or the choice of isovalue frequently is set to key field values which produces degeneracies.

Degeneracies typically result in the generation of non-manifold isocontours, or the creation of zero length (line) or zero area (triangle) primitives. This is typically due to the isocontouring "nicking" adjoining edges, at a common vertex, to produce such degenerate primitives. If the purpose of contour generation is strictly visual display, then such situations matter little as these primitives are generally handled properly during rendering. However, if the isocontour is to be processed in an analysis pipeline (e.g., topological analysis, subsequent interior mesh generation), degeneracies can introduce difficulties into the workflow. In many implementations of MC, degenerate primitives are culled prior to insertion into dynamic output arrays. However in FE, the initial passes count the number of output points and triangles and then allocates memory for them; once degenerate primitives are identified it is not possible to modify the output.

In the algorithm outlined here, the treatment of degeneracies can be expanded through modification of the edge-based case table and the way in which it is calculated. As described previously, edge bits are set when a cell edge intersects the isosurface Q using a semi-open interval check; in the expanded case table an open edge interval is used (i.e., meaning that s_{ijk} are classified in one of three ways: equal to, less than, or greater than q). Thus the number of entries in a vertex-based case table for voxel cells would increase to a total size of 3^8 entries (versus 2^8 for standard MC). While we have experimented with such alternative case tables in 2D, our workflow is such that an expanded case table is not needed at this time (a topic for future research).

3.6 Load Balancing

Many parallel isocontouring algorithms devote significant effort towards logically subdividing and load balancing the computation. Typically approaches include organizing volumes into rectangular sub-grids or using a spatial structure such as an octree to manage the flow of execution [33, 9]. The challenge with isocontouring is that a priori it is generally not possible to know through which portion of the volume the isosurface will pass, meaning that some form of pre-processing (evaluating min-max scalar region within sub-volumes for example), or task stealing is used to balance the workload across computational threads. In the Flying Edges algorithm, the basic task is processing of an edge, in which each edge (or associated cell row) can be processed completely independently. In our implementation, we chose to use the Thread Building Blocks (TBB) library using the `parallel_for` construct to loop in parallel over all edges [14]. Behind the scenes, TBB manages a thread pool to process this list of edges, and since the workload across an edge may vary significantly, new edge tasks are stolen and assigned to the thread pool in order to ensure continued processing. Note that the algorithm does not require mutexes or other locks as the output data is allocated and partitioned prior to data being written.

3.7 Memory Overhead

The algorithm computes and stores information in the first two passes. X-edge cases are described using two bits due to the four possible states defined from the two end vertex classifications. Thus with total grid size N, then approximately $2N$ bits of additional memory are consumed to represent the x-edge cases (compared to our typical grids with a 32-bit float or integer per s_{ijk}). In addition, edge metadata is stored for each E_{jk}, which consists of six integer values: the number of x-, y-, and z-cell axes edge intersections; the number of triangles produced along the associated R_{jk}; and the left and right trim positions. Assuming that the grid is of dimensions n^3, then the metadata requires $6n^2$ integer values (compared to the

$n^3 = N$ grid size). Thus the total memory overhead in bytes, assuming eight byte integers, is $8 \cdot 6n^2 + 2 \cdot n^3/8$.

4 RESULTS

In the following we characterize the performance of the algorithm against four different datasets. First we compare the performance across typical implementations. Then we examine the internal performance of the algorithm, quantifying the time to execute each pass of the algorithm, and the benefit of computational trimming.

4.1 Performance

The performance of three algorithms: Marching Cubes (MC), Synchronized Templates (ST), and Flying Edges (FE) is compared in this section as datasets, the number of threads of execution, and data sizes are varied. Note that the generated results are not controlled to produce identical results; rather they reflect what is typically done in practical application. MC and ST are executed using a data parallel approach, meaning that each thread executes a pipeline across a subsetted block of data, performing point merging within the piece but not across piece boundaries (hence there are topological seams between sub-volumes). In comparison FE does not produce such seams but can produce degenerate triangles on output (in other words, we used conventional MC-type case tables that do not address degenerate conditions). Also note that in this work we focused on threaded, multi-core implementations as the data size exceeds the capacity of GPUs (2048^3 with double precision scalars).

(A quick note regarding Synchronized Templates. This algorithm has long been the fastest non-preprocessed algorithm in the VTK system, initially implemented by K. Martin in VTK [23] in the year 2000. Similar to FE, ST uses a_{ijk} to ensure that each cell edge is processed at most only once and does not use a spatial locator. It also uses efficient, but serial, programming constructs to ensure high performance computing.)

To characterize the performance of FE, we ran it against four different datasets (Figure 6). The test system was an Intel E5-2699 Xeon Workstation, with $2 \times 18 = 36$ hardware cores for a total of 72 threads, and 64Gbyte memory. These datasets are as follows, listed in order from smallest to largest in size:

CT-angio. This dataset is part of the 3D Slicer (slicer.org) data distribution. It is a anonymized CT angiography scan at resolution of $(512 \times 512 \times 321)$, with 16-bit scalar values. We used $q = 100$. Find the data at [27].

Supernova. This supernova simulation data is sized 432^3 with a 32-bit float scalar type. The isovalue was set to $q = 0.07$ to run the tests. Find the data at [5].

Nano. The third test dataset is a scanning transmission electron microscopy dataset of dimensions $(471 \times 957 \times 1057)$ with $q = 1800$ and 16-bit unsigned scalars. The data is a tomographic reconstruction of a hyper-branched Co_2P nanocrystal. 3D isosurface contours show the particle size and shape. See [31] for more information.

Plasma. This $(2048 \times 2048 \times 2048)$ volume dataset is one time step from a kinetic simulation of magnetic reconnection in plasmas. The scalar data is 32-bit float values. We choose $q = 0.30$.

Table 1 compares the MC, ST, and FE algorithms in serial operation across the four datasets. The run times are normalized against the basic MC algorithm, with the actual run time of MC shown in parentheses. Note that an optimized, threadable version of MC is also run (MC-opt) which uses an edge-based locator and makes two passes to discard empty cells. Table 2 compares the MC-opt, ST, and FE algorithms in parallel operation across the four datasets using a constant 36 threads (the basic MC was not threadable and hence not used). Again, the run times are normalized against the optimized MC-opt algorithm, with the actual run time of MC-opt shown in parentheses. Table 3 compares ST and FE side-by-side on the full plasma 2048^3 dataset, and records FE overall parallel efficiency. (All reported run-times are an average time across five separate runs.)

Table 1: Comparison of selected serially executed isocontouring algorithms. Shown are normalized speedup factors (normalized against the standard MC algorithm). The plasma dataset was downsampled to 1024^3 because MC implementations were unable to process the full dataset.

Algorithm	CT-angio	Supernova	Nano	Plasma
MC	1 (2.10s)	1 (2.667s)	1 (3.88s)	1 (69.86s)
MC-Opt	1.49	1.92	1.28	1.79
ST	1.44	1.90	0.54	3.51
FE	5.22	7.35	3.51	8.58

Table 2: Comparison of selected parallel executed isocontouring algorithms (the number of threads=36). Shown is normalized speedup factors (normalized against optimized MC algorithm). The plasma dataset was downsampled to 1024^3 because MC implementations were unable to process the full dataset.

Algorithm	CT-angio	Supernova	Nano	Plasma
MC-Opt	1 (0.266s)	1 (0.266s)	1 (0.310s)	1 (4.56s)
ST	3.67	3.20	1.10	4.35
FE	8.26	9.45	4.49	11.05

Finally, Figure 7 plots the parallel efficiency of FE as the number of processors is varied from one to 72 (Intel Workstation) and one to 80 (IBM Power System). The IBM Power System S822 has two POWER8 processors with 10 cores each, each core has 8 threads running at 3.42 GHz (total of 160 threads) with 160 GB of RAM.

The results show superlinear scalability for FE at ≤ 4 threads on the Intel Workstation. Also note that FE outperforms all versions of MC, optimized MC, and ST in both serial and parallel execution. The actual performance gain would be even greater if the threaded ST and MC algorithms were required to merge coincident points across piece boundaries. Note also that MC was unable to run any data greater than approximately 1024^3 due to the use of 32-bit integral offsets in its implementation.

4.2 Analysis

Table 4 shows the time to execute each pass of FE on the four datasets described previously. The time is expressed as a percentage of the total algorithm execution time, computed from the Intel Workstation using 32 cores. Also shown in the table is a performance factor f capturing the effect of enabling computational

Table 3: Comparison run times and speed up factors, ST vs. FE. Plasma 2048^3 dataset used on Intel Workstation. Note that this system has 36 physical cores and 72 total threads.

Threads	ST (seconds)	FE (seconds)	FE Speed Up (factor)	FE Efficiency (percent)
1	157.95	53.07	2.98	100.0
2	76.82	25.14	3.06	105.6
4	39.90	12.99	3.07	102.1
8	22.32	7.2	3.10	92.13
16	12.79	4.15	3.08	79.99
24	9.013	2.96	3.04	74.60
36	6.76	2.39	2.83	61.61
72	5.57	2.06	2.71	35.85

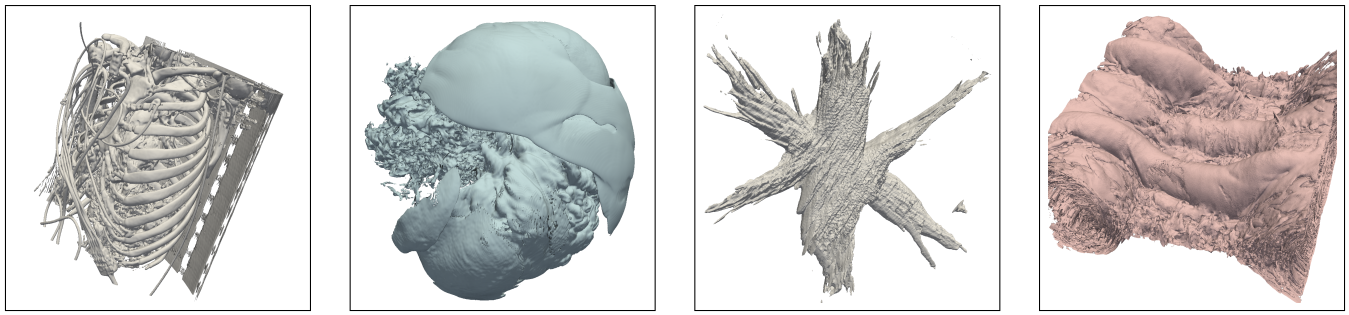

Figure 6: The four datasets used for testing. In reading order, the CT-angio, Supernova, Nano, and Plasma datasets.

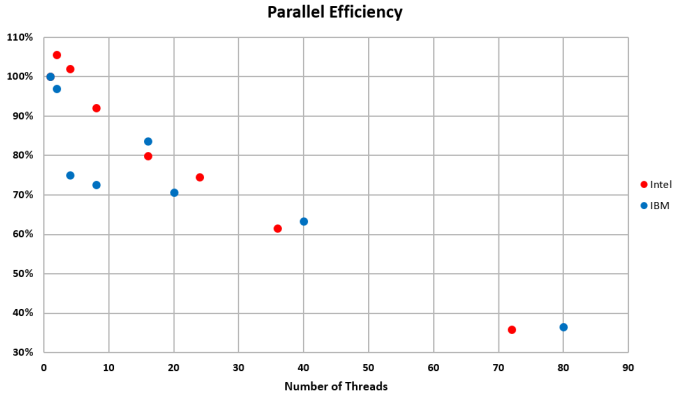

Figure 7: Parallel scaling efficiency across a representative 2048^3 dataset (the Plasma dataset described previously). Two plots are shown for a 72-thread (36 cores) Intel Workstation and IBM Power System with 160-threads (20 cores).

Table 4: Comparison of the four passes of the algorithm across the four datasets in parallel execution (32 threads). The numbers are expressed as a percentage of total execution time. The last column captures the effect of enabling computational trimming by showing a performance factor gain (i.e., numbers > 1 are faster).

Dataset	Pass 1 %	Pass 2 %	Pass 3 %	Pass 4 %	Trimming f
CT-angio	27.4	20.0	4.2	48.5	1.39
Supernova	81.9	5.6	12.1	0.4	1.15
Nano	79.0	6.3	14.5	0.2	1.16
Plasma	91.0	2.0	6.9	0.1	1.15

trimming; e.g., a factor greater than one $f > 1$ indicates that the algorithm is faster when trimming is enabled.

In general, a significant amount of time is spent traversing the dataset in the initial Pass 1. Indeed as the data becomes larger, increasing time is spent in this initial pass, as the amount of work to produce the isosurface relative to traversing the entire volume decreases, depending on the extent of the isosurface through the volume. Unfortunately in a single pass algorithm this initial traversal cannot be avoided; however it does affirm the benefits of preprocessing and search structures when performing exploratory isosurfacing. While computational trimming has a modest impact, its effect increases as the amount of work devoted to producing output (Passes 2-4) increases as well. While we were initially surprised by the amount of time taken by the metadata prefix sum operation (Pass 3), this operation is expected to scale suboptimally. Further note that the Nano dataset is slab shaped, with its smallest dimension in the $x-$direction. This suggests that much work is going into processing many, relatively shorter edge runs compared to the total dataset size. It may be that in some situations it may be better to process data in the y- or z-edge directions, depending on the effect of processing data in non-contiguous order, a topic for future research.

5 Conclusion and Future Work

We have developed a high-performance, scalable isocontouring algorithm for structured scalar data. To our knowledge it is currently the fastest non-preprocessed isocontouring algorithm on threaded, multi-core CPUs available today. The development of this algorithm has been motivated by the emergence of new parallel computing models, with the recognition that many current and future algorithms require rethinking if we are to take full advantage of modern computing hardware. Each pass of the Flying Edges is independently parallelizable across edges, resulting in scalable performance through task-stealing, edge-based computational tasks. Our results demonstrate this, although the actual computational load of any isocontouring algorithm based on an indexed lookup is relatively small, requiring just a few floating point operations to interpolate intersection points along edges. As a result, for very large data, much of the computing effort involves loading and/or accessing data across the memory hierarchy. Thus with large numbers of threads, memory bounds may limit the overall scalability.

To address memory constraints, distributed, hybrid parallel approaches may perform better as separate machines (and therefore memory spaces) can process a subset of the entire volume. Along similar lines, we have envisioned a parallel, hybrid implementation that performs the initial, two FE counting passes across all distributed, grid subsets, and then communicates point and triangle id offsets to each subsetted grid. Then each machine can process its data to produce seamless isocontours without the need to merge points across distributed grids.

Another way to view the Flying Edges algorithm is that it is a form of data traversal, with geometric constraints (continuous surfaces) informing the extent to which data is processed. Similar geometric constraints exist for related visualization algorithms such as scalar-based cutting and clipping. Also, the algorithm may be readily adapted to any topologically regular grid. In the future we plan on extending the algorithm to process rectilinear and structured grids, and clipping and cutting algorithms, based on an abstraction of the edge-based traversal process, with appropriate specialization required in the way edge interpolations (i.e., point coordinates and interpolated data) are generated.

We have already begun implementing this algorithm on GPUs. While our initial experiments are promising, there is some uncertainty as to whether the extra work required to track computational trimming is warranted. On one hand, processing entire edges E_{jk} without early termination simplifies computation and reduces the likelihood of stalling the computational cores. On the other hand, computational trimming can significantly reduce the amount of data

to be visited, and consequently the total computation performed. Future numerical experiments will provide guidance, although it is possible that differing hardware may produce contrary indications based on the particulars of the computing architecture. Finally, GPU processing of large volumes requires complex methods to stream data on and off the GPU when card memory is exceeded. The speed of Flying Edges is such that with very large data, the cost of data transfer to and from the GPU often exceeds the time to actually process the data on a multi-core CPU. (Refer to Table 2 assuming PCIE transfer throughput of 8GB/sec.)

In the spirit of reproducible science, we have implemented the algorithm in both 2D and 3D and contributed them to the VTK system with a permissive BSD license. These open source implementations are available as vtkFlyingEdges2D and vtkFlyingEdges3D.

Acknowledgements

This effort is part of an overall vision which we refer to as VTK-m, in which critical algorithms in the VTK system are extended towards massive multi-core parallelism. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program under Award Number(s) DE-SCOO07440. We have been supported by many members of the DOE National Labs, particularly the SDAV [24] effort. Additional support has been provided by nVidia and Intel. We'd also like to thank the 3D Slicer community; H. Karimabadi, W. Daughton, and Yi-Hsin Liu for providing the Plasma dataset; Rob Hovden for the nanoparticle data; and C. Atkins and S. Philip for their help generating results.

References

[1] J. Ahrens. Implications of numerical and data intensive technology trends on scientific visualization and analysis. Plenarry Presentation at SIAM CSE15, March 2015.

[2] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of Supercomputing 2014*, 2014.

[3] C. Bajaj, V. Pascucci, D. Thompson, and X. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, pages 97–104. IEEE Computer Society, 1999.

[4] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[5] J. Blondin. Supernova modelling. http://vis.cs.ucdavis.edu/VisFiles/pages/supernova.php.

[6] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *Proceedings of IEEE Visualization '98*, pages 167–174, 1998.

[7] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, Apr-June 1997.

[8] M. Ciżnicki, M. Kierzynka, K. Kurowski, B. Ludwiczak, K. Napierała, and J. Palczyński. Efficient isosurface extraction using marching tetrahedra and histogram pyramids on multiple gpus. In *Parallel Processing and Applied Mathematics*, pages 343–352. Springer, 2012.

[9] D. D'Agostino, A. Clematis, and V. Gianuzzi. Parallel isosurface extraction for 3D data analysis workflows in distributed environments. *Concurrency and Computation: Practice and Experience*, 23(11):1284–1310, 2011.

[10] C. Dietrich, C. E. Scheidegger, J. L. Comba, L. P. Nedel, C. T. Silva, et al. Edge groups: An approach to understanding the mesh quality of marching methods. *Visualization and Computer Graphics, IEEE Transactions on*, 14(6):1651–1666, 2008.

[11] C. Dyken, G. Zieglar, C. Theobalt, and H. Seidal. High-speed Marching Cubes using HistoPyramids. *Computer Graphics Forum*, 27(8):2028–2039, 2008 2008.

[12] D. Horn. Stream reduction operations for gpgpu applications. In M. Pharr and R. Randima, editors, *GPU gems 2: programming techniques for high-performance graphics and general-purpose computation*, page 573V589. Addison-Wesley Professional, 2005.

[13] C. Hsu and W. Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the ACM/IEEE SC 2005 Conference*, November 2005.

[14] Intel. Threading Building Blocks. https://www.threadingbuildingblocks.org/.

[15] B. Jeong, P. A. Navrátil, K. P. Gaither, G. Abram, and G. P. Johnson. Configurable data prefetching scheme for interactive visualization of large-scale volume data. In *IS&T/SPIE Electronic Imaging*, pages 82940K–82940K. International Society for Optics and Photonics, 2012.

[16] M. Kazhdan, A. Klein, K. Dalal, and H. Hoppe. Unconstrained isosurface extraction on arbitrary octrees. In *Symposium on Geometry Processing*, volume 7, 2007.

[17] Y. Livnat, H. Shen, and C. R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, March 1996.

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

[19] Y. Lu, Y. Chen, and R. Thakur. Memory-conscious collective I/O for extreme scale HPC systems. In *Proceedings of High Performance Computing, Networking, Storage and Analysis (SCC)*, 2012.

[20] S. Martin, H.-W. Shen, and P. McCormick. Load-balanced isosurfacing on multi-gpu clusters. *EGPGV*, 10:91–100, 2010.

[21] NVIDIA Developer Zone. CUDA implementation of isosurface computation. http://docs.nvidia.com/cuda/cuda-samples/index.html#marching-cubes-isosurfaces.

[22] L. Schmitz, L. F. Scheidegger, D. K. Osmari, C. A. Dietrich, and J. L. D. Comba. Efficient and quality contouring algorithms on the gpu. *Computer Graphics Forum*, 29(8):2569–2578, 2010.

[23] W. J. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object Orient Approach to Computer Graphics, Fourth Edition*. Kitware, Inc., 2006.

[24] DOE scalable data management, analysis, and visualization program. http://www.sdav-scidac.org/.

[25] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, pages 1–25, 2010.

[26] J. P. Shen and M. H. Lipasti. *Modern processor design : fundamentals of superscalar processors (First Edition)*. Waveland Press, 2005.

[27] 3D slicer sample data: CT-cardio. http://www.slicer.org/slicerWiki/images/0/00/CTA-cardio.nrrd.

[28] G. Treece, R. Prager, and A. Gee. Regularised marching tetrahedra: improved iso-surface extraction. *Comp. & Graphics*, 23(4):593–598, 1999.

[29] Q. Wang, J. JaJa, and A. Varshney. An efficient and scalable parallel algorithm for out-of-core isosurface extraction and rendering. *Journal of Parallel and Distributed Computing*, 67:592–603, 2007.

[30] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. In *Proceedings of the 1990 workshop on Volume visualization*, pages 57–62, 1990.

[31] H. Zhang, D.-H. Ha, R. Hovden, L. F. Kourkoutis, and R. D. Robinson. Controlled synthesis of uniform cobalt phosphide hyperbranched nanocrystals using tri-n-octylphosphine oxide as a phosphorus source. *NANO Letters*, 11(1):188–197, 2011.

[32] X. Zhang, C. Bajaj, and W. Blanke. Scalable isosurface visualization of massive datasets on cots clusters. In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 51–58. IEEE Press, 2001.

[33] X. Zhang and B. Chandrajit. Scalable isosurface visualization of massive datasets on commodity off-the-shelf clusters. *Journal of parallel and distributed computing*, 69(1):39–53, 2009.