ISOSURFACE VISUALIZATION MINIAPPLICATION

DANIEL BOURGEOIS [*], MICHAEL WOLF [†], AND KENNETH MORELAND [‡]

Abstract. Scientific simulations often make use of increasinly massive, heterogeneous high performance computers. Often, the output from such simulations are too large to fit in main memory. To cope, data visualization experts have had to write parallel algorithms on top of complex, large-scale libraries. The purpose of this paper is to introduce miniIsosurface, a miniapplication that is part of the Mantevo suite of miniapplications that implements two isocontouring algorithms: Marching Cubes and Flying Edges. Miniapplications are small applications that are still complex enough to study larger performance trends. miniIsosurface was explicitly designed to provide software developers and data visualization experts an avenue to explore performance characteristics before full-scale development takes place. Implementations inside of miniIsosurface take advantage of MPI, OpenMP and GPU.

1 Introduction The current era of computing, characterized by increased data and massive heterogeneous parallelism, has brought new challenges to developers of scientific applications. Traditionally, developing scalable applications has focused on the core computational kernel while ignoring the performance of the front and back ends of the simulation pipeline. For applications with smaller output than is now common, scientists could archive simulation results for later interpretation. For extreme-scale applications, however, the output often contains too much data to store in main memory or is limited by IO bandwidth. Hence, there is now a need to develop extreme-scale scientific applications within an ecosystem that includes modeling, simulation, analysis and visualization.

Developing real-world applications that simultaneously integrate the whole scientific simulation pipeline and are scalable remains technically difficult. The increase of heterogeneous, massively parallel architectures has made achieving scalability and portability harder. Scientific computing applications have traditionally relied on OpenMP [14] and MPI [5]. But requiring software implementers to explicitly manage memory hierarchies and limit data movement is becoming more burdensome and less portable. In turn, developing applications that solely rely on OpenMP and MPI for parallelism is becoming less cost-effective. The new status quo is being reflected in the proliferation of tools available to the developer. Many libraries have been developed with the goal of providing developers with the ability to easily and efficiently write performance portable scientific applications. For example, Kokkos [4] has been designed to target complex node architectures with N-level memory hierarchies and multiple types of execution resources. The Thrust [8] library provides a flexible, high-level interface for GPU programming while the same code can utilize OpenMP or Intel Thread Building Blocks [15]. Other parallel programming libraries include HPX [9] and charm [10]. In the realm of visualization, VTK-m [13] “serves as a container for algorithms, provides flexible data representation, and simplifies the design of visualization algorithms on new and future computer architecture.”

Even with new tools, algorithms may not still scale in heterogeneous environments or in different parallel paradigms. Understanding the effects that tool choice will have on performance in real-world applications prior to full-scale development is necessary to developing performance portable applications.

An effort towards understanding performance effects in large-scale applications is the Mantevo Project [7], an open-source effort “for the analysis, prediction and improvement of high performance computing applications” through the use of proxy applications. Mantevo is itself a suite of small applications, miniapps or miniapplications, that are meant to “capture

[*]Louisiana State University, dbour27@lsu.edu
[†]Sandia National Laboratories, mmwolf@sandia.gov
[‡]Sandia National Laboratories, kmorel@sandia.gov

key performance issues that can be exposed in comparatively small, easy to execute, easy to modify code" [2]. Miniapps do not, in general, serve as replacements to real applications but they do provide considerably more information than benchmarks. The miniapp discussed in this paper, miniIsosurface, provides a collection of algorithmic approaches to solving a common visualization task. Having multiple implementations in the same place provides an oppurtunity to not only test comparative performance of the algorithms, but to serve as a baseline for further research into iterative improvements to an algorithm. Another use of miniapps is in the stage before an application is fully developed, developers can explore the suite of miniapps to determine which tools are best suited for the project at hand. To save time and reduce costs, developers can also turn to a miniapp to quickly predict how certain modifications may alter the performance characteristics of a full size application.

How new architectures should be best adapted to is still an open design problem. As is the best way to integrate visualization. To better study the performance behavior of visualization on numerous workflows and numerous systems, this paper examines a miniapp, miniIsosurface. The miniapp was developed to characterize the behavior of isocontouring, a widely used visualization technique.

2 Isocontouring Algorithms Isocontouring algorithms extract a polygonal mesh from a 3D scalar field that is used to approximate the surface at a specified isovalue. For rendering, gradients at the vertices of the polygonal mesh are also computed. These field gradients by isocontour definition give the direction perpendicular to the surface, which is information required by the rendering algorithm to compute how light is reflected. The two algorithms considered here, Marching Cubes and Flying Edges both produce a triangular mesh as well as gradients at the vertex of each triangle.

2.1 Marching Cubes Marching Cubes (MC) is the original isocontouring algorithm, published in 1987 by Lorenson and Cline [12] for the visualization of medical images.

Conceptually, the MC algorithm divides a 3D scalar field into a three-dimensional grid of imaginary cubes, where only the scalar values at the vertices of the cubes are considered. The algorithm determines how the mesh intersects each cube and approximates the mesh going through that cube by a set of zero or more triangles. For output, each triangle consists of the locations and approximate gradient of the three vertices. The final output is all triangles over the entire grid of imaginary cubes.

The insight provided by Lorenson and Cline was how they characterized the part of the mesh passing through each cube. Each vertex of every cube is given a 1 if its value is greater than or equal to the isovalue corresponding to the isosurface and a 0 otherwise. Thus each cube can be given an 8-bit identification. A preconfigured lookup table is created that maps the 256 cube identification numbers to the set of triangles that most matches the isosurface going through cubes with the corresponding identification number. Each vertex of each triangle is located on one of the edges of the cube that has been cut. The precise location for the vertex is calculated by linearly interpolating the ends of the cut edge to the point of intersection. The gradient is calculated in a similar manner, except using the derivative of the scalar field at the ends of the cut edge instead. At the isosurface, the gradient is equal to the derivative of the scalar field. All derivatives are calculated using central differences along the three coordinate axes of the imaginary grid of cubes.

2.2 Flying Edges The Flying Edges (FE) algorithm [17] was designed from the MC algorithm to perform better on multicore processors, to have better cache performance, and to reduce the time to find connections between triangles generated. See Section 3 for a description of how each algorithm was parallelized.

Similarly to MC, FE works on a three-dimensional grid of imaginary cubes and uses a preconfigured lookup table from which to determine the triangle configuration of each cube. The location and gradient of the vertices are computed in the same way as MC.

The difference between the two algorithms lies in how the input data is processed. Whereas MC makes one pass through all cubes, finishing with a list of triangles and associated data, FE uses multiple preprocessing steps.

The first and second preprocessing steps determine which edges are cut. The first preproccesing step determines all edges that are cut along one dimension of the image by accesing the isocontour values stored in the image. Choosing the correct image dimension to work along will maximize cache performance.

Similar to the first preprocessing step, the second step determines which edges along the other two dimensions are cut. However, instead of using the scalar values of the image, the second preprocessing step uses additional information gained in the first preprocessing step to determine which of the remaining edges are cut. By not going back to the image and directly using information in the first preprocessing step, cache performance is again being maximized. In these two steps, additional information is calculated. For a detailed description, see [17].

After the first two preprocessing steps, the number of triangles and cut edges are known. From that information, the size of the output is easily calculated. The third of the preprocessing steps allocates memory to contain the triangles and points. In addition, a prefix sum is calculated so that the location in the output of each point and triangle is known.

After the three preprocessing steps, Flying Edges calculates the intersection point corresponding to each cut edge and populates the polygonal mesh the those points, the triangles and if desired, the gradients.

3 miniIsosurface The purpose of miniIsosurface is to provide a platform for scientific application developers to test performance of isocontouring algorithms on a wide variety of architectures and programming models. The specific use case that application developers will have is unknown and unknowable, so miniIsosurface must be extensible. But at the same time, miniIsosurface must provide points of comparison. To handle this, miniIsosurface has control implementations that are analogous to control treatments in a scientific study. The control implementations serve as the baseline to different and novel implementations that could be introduced by an application developer working on a larger project. To ease the burden of creating new implementations, the control implementations of miniIsosurface are heavily commented and use standard C++ idioms and containers.

As it is impossible to create an implementation of MC and FE with every parallel programming model, for the purpose of this work miniIsosurface focuses on a few canonical control implementations, namely implementations without parallelization and in OpenMP and MPI. Marching Cubes was implemented in serial, with OpenMP and with MPI. Flying Edges was implemented in serial, with OpenMP and on GPU using Thrust. In addition, miniIsosurface implements a wrapper to call VTK [16] which can be specified to use either Marching Cubes or Flying Edges. Researchers are encouraged to add implementations using different algorithms or different programming models for further comparison.

For reference, the rest of this section details how parallelism was achieved in the control implementations and the relative benefits and advantages.

3.1 Parallel Marching Cubes Marching Cubes was implemented in parallel by dividing the 3D scalar image into rectangular sections. Input parameters control the size of each section along the x, y and z dimension. Each section is then processed independently. The output data from each section is then merged into one array.

For OpenMP, see Listing 1 for the parallelization scheme. The MPI implementation is similar in form. Each OpenMP thread fills out local output variables *points*, *normals*

Listing 1: Parallelization of Marching Cubes using OpenMP

```
// declare points, normals, triangles
#pragma omp parallel
{
  // declare threadPoints, threadNormals, threadTriangles
  // declare threadPointMap

  #pragma omp for nowait
  for(int i = 0; i < nSections, ++i)
  {
    // process section i of marching cubes
  }

  #pragma omp critical
  {
    // add threadPoints to points,
    //    threadNormals to normals,
    //    threadTriangles to triangles
  }
}
// output mesh
```

and *triangles*. *points* and *normals* each contain a vector of points of the mesh. *triangles* contains a vector of a triplet of indices where the indices refer to values in *points* and *normals*. Each thread processes the sections that OpenMP assigns to it. Once all sections are processed, the local output needs to be merged into the global output. For readability, a simple, yet performant, way to do this was chosen. As each thread finishes filling out local output, that output is added to the global *points*, *normals* and *triangles* where only one thread at a time is permitted to add to the global output.

In Listing 1, each thread owns a *threadPointMap*. *threadPointMap* maps global edge indices of the lattice formed by the imaginary grid of cubes to an index in *threadPoints*. This is used to make sure that no duplicate points occur in the local output. However, after merging results from different threads, points that occur along the shared boundary of sections will appear multiple times. An OpenMP implementation was created in miniIsosurface to remove duplicate values *points* and *normals*.

Marching Cubes was not designed for the explicit purpose of parallelism and there are a few flaws inhibiting scalability. With MC the the output size is not predetermined so output formed in parallel must be merged, causing a parallel bottleneck and creating duplicate points. Furthermore, output arrays cannot be correctly preallocated and may require dynamic memory allocation.

3.2 Parallel Flying Edges Unlike Marching Cubes, Flying Edges was designed with parallelism in mind. The four steps of FE, the three preprocessing steps and calculating the output mesh, are each run in parallel where each step starts when the previous step finishes. Only the third preprocessing step, which calculates a prefix sum, is not trivially parallelized with a parallel for loop.

The GPU version took advantage of the Thrust library [8] instead of using raw CUDA.

Using Thrust put a layer of abstraction between the GPU and the application, which eased the implementation and readability.

A few modifictations were made to the FE algorithm to implement the GPU version. In the GPU implementation a finer grain of parallelism was used. Insead of parallelizing over each row of cubes indepentently as the other implementations did, the GPU implementation parallelized over each cube. In addition, the GPU version did not calculate trim limits (see [17]).

4 Performance Results All tests were run on Dual socket E5-2698V3 Intel Xeon Processors with 16 cores. GPU tests were run using a single NVIDIA Tesla K80m GPU. To compile, GCC version 4.9.3 was used, Cuda version 7.5.7, OpenMPI version 1.10.0. The datasets [3, 1, 6, 11] are freely available.

Table 4.1: List of data sets used in tests.

Dataset	x-Dim	y-Dim	z-Dim	Isovalue
Supernova [3]	432	432	432	0.07
CT-Angio [1]	512	512	321	100
Stag beetle [6]	832	832	494	1
Christmas Tree [11]	512	499	512	1

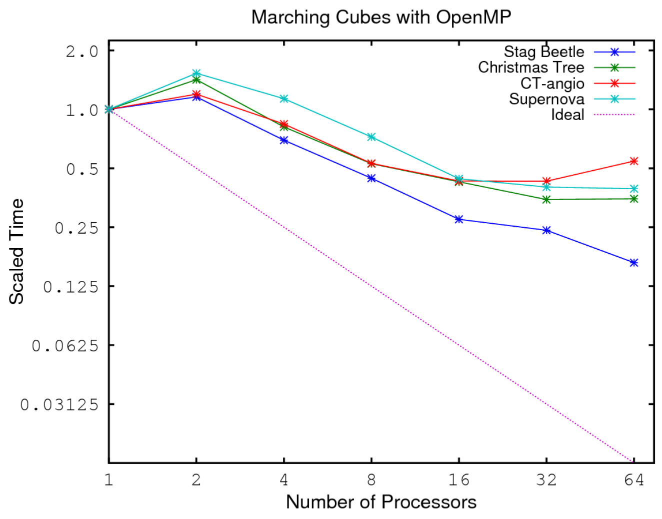

Fig. 4.1: Marching Cubes with OpenMP speedup. The ideal line is perfect speedup.

Figures 4.1 and 4.2 show the speedup of Marching Cubes with the four datasets using OpenMP and MPI, respectively. The idealized line shows the line of a perfect speedup. It is evident that the MPI code scales better than the OpenMP code. Like the OpenMP code, the MPI tests were run on one compute node. As each code works on separate sections of the overall image in parallel in the same way, it is expected that the performance results are similar. Indeed, for one processor, performance results are similar, as seen in Table 4.2. However, after reading in the image data, the MPI code copies image section data to each processor. This is a quick operation on one compute node that may have later add benefits

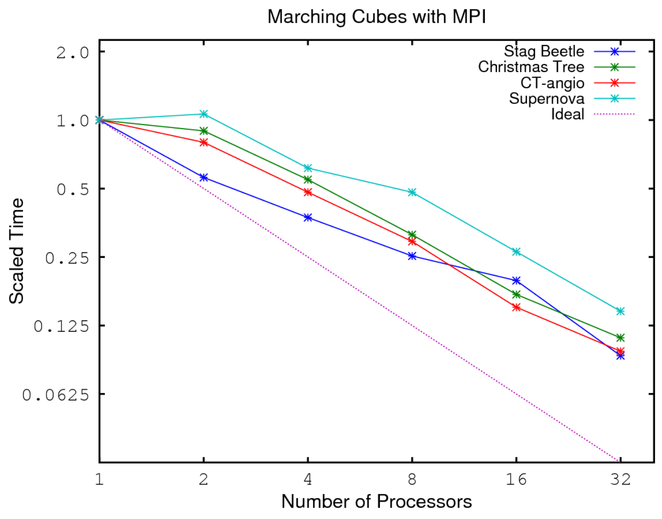

Fig. 4.2: Marching Cubes with MPI speedup. The ideal line is perfect speedup. One compute node was used.

because storing the data in smaller data structures may increase the likelihood of cache-hits. Increased cache hits may then be a potential source of the improved scalability in the MPI code.

Table 4.2: Serial, OpenMP, MPI and GPU times in seconds. *S* stands for serial, *O* stands for OpenMP, *M* stands for MPI and *G* stands for GPU. Results for OpenMP and MPI in this table are with 32 processors.

Dataset	MC S	MC O	MC M	FE S	FE O	FE G
Supernova	3.36	0.810	0.390	1.18	0.248	0.409
CT-Angio	2.74	0.876	0.262	1.08	0.258	0.362
Stag beetle	3.56	0.856	0.331	1.44	0.482	0.987
Christmas Tree	2.34	0.630	0.234	1.52	0.313	0.411

Figure 4.3 shows the speedup of Flying Edges with the four datasets using OpenMP. At 64 cores, roughly 3x speedup is achieved on the CT-Angio dataset. For the GPU, Flying Edges achieved a similar 3x speedup on the CT-Angio dataset.

Figure 4.4 compares the three algorithms on the Stag Beetle dataset. Marching Cubes OpenMP is clearly outperformed by both Marching Cubes MPI and Flying Edges OpenMP. Even though Flying Edges OpenMP is faster on one process than Marching Cubes MPI, they perform similarly with more processors.

The miniIsosurface miniapp also contains a wrapper that can call VTK. The results of calling VTK with the wrapper are shown in Table 4.3. The version of VTK used was 8.0.0. It should be stressed that the comparisons between the wrapper to VTK and miniIsosurface may not be fair because VTK contains overhead to pipe data from input to output.

Table 4.3: Time in seconds using the VTK wrapper in miniIsosurface. These times are not a direct comparison to miniIsosurface implementations.

Dataset	VTK MC	VTK FE
Supernova	21.1	10.1
CT-Angio	14.9	10.3
Stag beetle	11.8	6.60
Christmas Tree	9.20	8.71

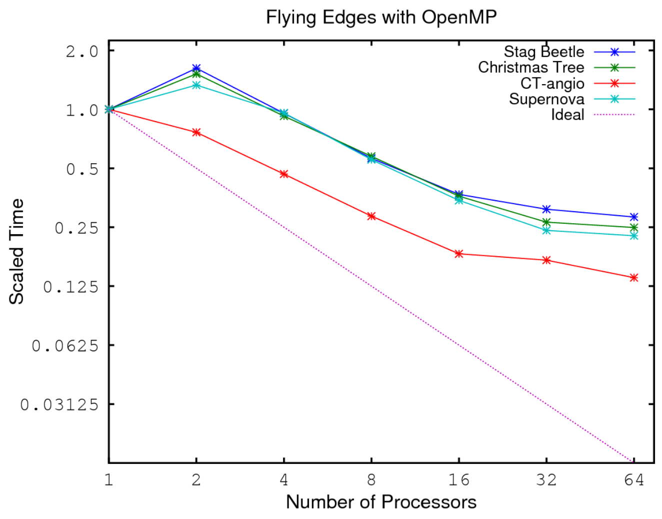

Fig. 4.3: Flying Edges with OpenMP speedup. The ideal line is perfect speedup.

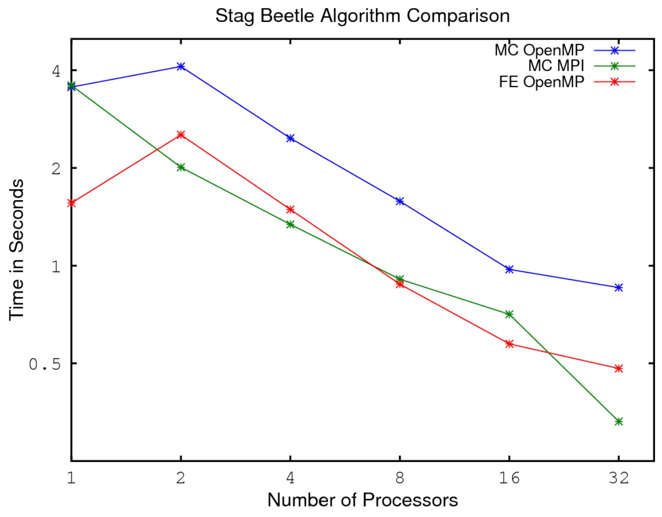

Fig. 4.4: A time comparison of Marching Cubes and Flying Edges parallel implementations.

5 Conclusions As a miniapp, miniIsosurface is designed to bridge the gap from toy code to application code, from easily extendable to reasonably scalable. As a tool, miniIsosurface should be used to adapt and test more complex codes to new architectures.

It has been shown that miniIsosurface does indeed scale. At the same time, the code base is designed so that C++ programmers can understand how the algorithms are parallelized in the various implementations. Not every possible parallel framework was used, but providing implementations in OpenMP, MPI and GPU should pave the way to porting either of the algorithms to other parallel frameworks.

REFERENCES

[1] *3d slicer sample data: Ct-cardio.*

[2] D. Barnette, M. Bettencourt, and M. Hoemmen, *Using miniapplications in a Mantevo framework for optimizing Sandia's SPARC CFD code on multi–core, many–core, and GPU–accelerated compute platforms*, in 51st AIAA aerospace sciences meeting including the new horizons forum and aerospace exposition, 2012, p. 1126.

[3] J. Blondin, *Supernova modeling.*

[4] H. C. Edwards, C. R. Trott, and D. Sunderland, *Kokkos: Enabling many–core performance portability through polymorphic memory access patterns*, Journal of Parallel and Distributed Computing, 74 (2014), pp. 3202–3216.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al., *Open MPI: Goals, concept, and design of a next generation MPI implementation*, in European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 2004, pp. 97–104.

[6] M. E. Groller, G. Glaeser, and J. Kastner, *Stag beetle*, 2005.

[7] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, *Improving performance via mini-applications*, Sandia National Laboratories, Tech. Rep. SAND2009-5574, 3 (2009).

[8] J. Hoberock and N. Bell, *Thrust: A parallel template library*, 2010.

[9] H. Kaiser, B. A. L. aka wash, T. Heller, A. Berg, J. Biddiscombe, A. Bikineev, G. Mercer, A. Schfer, atrantan, A. Serio, J. Habraken, M. Anderson, S. R. Brandt, M. Stumpf, D. Bourgeois, M. Copik, K. Huck, V. Amatya, L. Viklund, Z. Khatami, D. Bacharwar, S. Yang, E. Schnetter, Bcorde5, M. Brodowicz, L. Troska, B. Wagle, S. Upadhyay, Z. Byerly, and H. Brakmic, *STEllAR-GROUP/hpx: HPX V1.0: The C++ Standards Library for Parallelism and Concurrency*, Apr. 2017.

[10] L. V. Kale and S. Krishnan, *Charm++: a portable concurrent object oriented system based on C++*, in ACM Sigplan Notices, vol. 28, ACM, 1993, pp. 91–108.

[11] A. Kanitsar, *Christmas tree*, 2002.

[12] W. E. Lorensen and H. E. Cline, *Marching cubes: A high resolution 3d surface construction algorithm*, in ACM siggraph computer graphics, vol. 21, ACM, 1987, pp. 163–169.

[13] K. Moreland, C. Sewell, W. Usher, L.-t. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, et al., *Vtk-m: Accelerating the visualization toolkit for massively threaded architectures*, IEEE computer graphics and applications, 36 (2016), pp. 48–58.

[14] OpenMP Architecture Review Board, *OpenMP application program interface version 4.5*, 2011.

[15] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*, " O'Reilly Media, Inc.", 2007.

[16] W. Schroeder, K. Martin, and B. Lorensen, *The visualization toolkit 4th edition*, 2006.

[17] W. Schroeder, R. Maynard, and B. Geveci, *Flying edges: A high-performance scalable isocontouring algorithm*, in Large Data Analysis and Visualization (LDAV), 2015 IEEE 5th Symposium on, IEEE, 2015, pp. 33–40.