Published in final edited form as:
SIAM J Sci Comput. 2011 ; 33(5): 2468–2488. doi:10.1137/100788951.

A FAST ITERATIVE METHOD FOR SOLVING THE EIKONAL EQUATION ON TRIANGULATED SURFACES*

Zhisong Fu[†], **Won-Ki Jeong**[‡], **Yongsheng Pan**[†], **Robert M. Kirby**[†], and **Ross T. Whitaker**[†]

Zhisong Fu: zhisong@sci.utah.edu; Won-Ki Jeong: wkjeong@unist.ac.kr; Yongsheng Pan: ypan@sci.utah.edu; Robert M. Kirby: kirby@sci.utah.edu; Ross T. Whitaker: whitaker@sci.utah.edu

[†]The Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT 84112

[‡]Electrical and Computer Engineering, UNIST (Ulsan National Institute of Science and Technology), 100 Banyeon-ri Eonyang-eup, Ulju-gun Ulsan, Korea 689-798

Abstract

This paper presents an efficient, fine-grained parallel algorithm for solving the Eikonal equation on triangular meshes. The Eikonal equation, and the broader class of Hamilton–Jacobi equations to which it belongs, have a wide range of applications from geometric optics and seismology to biological modeling and analysis of geometry and images. The ability to solve such equations accurately and efficiently provides new capabilities for exploring and visualizing parameter spaces and for solving inverse problems that rely on such equations in the forward model. Efficient solvers on state-of-the-art, parallel architectures require new algorithms that are not, in many cases, *optimal*, but are better suited to synchronous updates of the solution. In previous work [W. K. Jeong and R. T. Whitaker, *SIAM J. Sci. Comput.*, 30 (2008), pp. 2512–2534], the authors proposed the *fast iterative method* (FIM) to efficiently solve the Eikonal equation on regular grids. In this paper we extend the fast iterative method to solve Eikonal equations efficiently on triangulated domains on the CPU and on parallel architectures, including graphics processors. We propose a new local update scheme that provides solutions of first-order accuracy for both architectures. We also propose a novel triangle-based update scheme and its corresponding data structure for efficient irregular data mapping to parallel single-instruction multiple-data (SIMD) processors. We provide detailed descriptions of the implementations on a single CPU, a multicore CPU with shared memory, and SIMD architectures with comparative results against state-of-the-art Eikonal solvers.

Keywords

Hamilton–Jacobi equation; Eikonal equation; triangular mesh; parallel algorithm; shared memory multiple-processor computer system; graphics processing unit

1. Introduction

The Eikonal equation has a wide range of applications. In image analysis, for example, shortest paths defined by image-driven metrics have been proposed for segmentation [16] and the tracking of white-matter pathways in the diffusion weighted images of the brain [10]. In seismology the Eikonal equation is used to calculate the travel time of the optimal trajectories of seismic waves [23]. The Eikonal equation models the limiting behavior of

*This work was funded by NIH/NCRR Center for Integrative Biomedical Computing (P41-RR12553-10) and Department of Energy (DOE NET DE-EE0004449).

Maxwell's equations [8] and is therefore useful in geometric optics. In computer graphics, geodesic distance on surfaces has been proposed for surface remeshing and mesh segmentation [24, 26]. The Eikonal equation also has applications in medicine and biology. For instance, cardiac action potentials can be represented as moving interfaces and Eikonal-curvature descriptions of wavefront propagation [13, 5]. For many of these applications described above, unstructured simplicial meshes, such as tetrahedra and triangles, are important for accurately modeling material interfaces and curved domains. This paper addresses the problem of solving the Eikonal equation on triangulated domains, which are approximations to either flat regions (subsets of $\Re^2$) or curved surfaces in $\Re^3$.

For many of these applications, there is a need for fast solutions to the Eikonal equation—e.g., run times of fractions of a second on large domains. For instance, solvers that can run interactively will allow scientists and mathematicians to explore parameter spaces of complex models and to reconfigure geometries and visualize their relationships to the solutions. In other cases, such as inverse problems and remeshing, the algorithms require multiple solutions of the Eikonal equation as part of the inner loop of an iterative process. Thus, there is a need for fast, efficient Eikonal solvers.

Efficient solutions on state-of-the-art computer architectures place particular constraints on the data dependencies, memory access, and scale of logical operations for such algorithms. The trend in computer architecture is toward multicore CPUs (conventional processors) and massively parallel streaming architectures, such as graphics processing units (GPUs). Thus, parallel algorithms that run efficiently on such architectures will become progressively more important for many of these applications. Of particular interest are the massively parallel streaming architectures that are available as commodities on consumer-level desktop computers. With appropriate numerical algorithms, these machines provide computational performance that is comparable to the supercomputers of just a few years ago. For example, the most recent GPUs, which cost only several hundred US dollars, can reach a peak performance of nearly 10^{12} floating point operations per second (TeraFLOPS); a performance equivalent to a top supercomputer a decade ago [29]. This computing power, however, is for a single-instruction multiple-data (SIMD) computational model, and most of the recent massively parallel architectures, such as GPUs [4], rely heavily on this paradigm. These modern SIMD architectures provide a large number of parallel computing units (up to several hundred cores) in a hierarchical data-sharing structure, rather simple branching circuits, and large memory bandwidth. As such, they place important restrictions on the algorithms that they can run efficiently. Addressing these constraints is an important aspect of this paper.

In the past several decades, many methods have been proposed to solve the Eikonal equation on unstructured grids for both two-dimensional (2D) and three-dimensional (3D) domains. Iterative schemes, for example [21], rely on a fixed-point method that solves a quadratic equation at each grid point in a predefined update order and repeats this process until the solution on the entire grid converges. Some adaptive, iterative methods based on a label-correcting algorithm (from a similar shortest-path problem on graphs [2]) have been proposed [17, 3, 6, 7].

The fast marching method (FMM) by Sethian [22], a form of the algorithm first proposed in [19], is used widely and is the de facto state-of-the-art for solving the Eikonal equation. FMM has an asymptotic worst case complexity of $O(N \log N)$, which is optimal. However, it uses a strict updating order and the min-heap data structure to manage the narrow band which represents a bottleneck that thwarts parallelization. Although the FMM has some parallel variants [9, 28] that use domain decompositions, they rely on a serial FMM within each subdomain, which is not efficient for massively parallel SIMD architectures.

Furthermore, these parallel variants are for regular grids only, and the extension to unstructured, triangular meshes, the topic of this paper, is not straightforward.

For homogeneous speed functions on flat domains, the characteristics of the Eikonal equation are straight lines. In such cases, one can solve the Eikonal equation by updating solutions along specific directions without explicit checks for causality. Based on this observation, Zhao [31] and Tsai et al. [27] proposed the fast sweep method (FSM) which uses a Gauss–Seidel update scheme for the straight (grid-aligned) wavefront and proceeds across the domain in an incremental *sweep*. This method may converge faster than the Jacobi update methods, which update all grid points at once. However, the update scheme, which proceeds simultaneously for all nodes on the wavefront, still presents a bottleneck because it limits updates to a specific set of points in a predefined order. More importantly, previous work [11] has shown that the number of iterations or sweeps grows with the complexity of the *speed function*, and thus the method is efficient only for relatively simple (nearly homogeneous) inputs, where the characteristics are approximately straight. FSM has extensions to 2D and 3D unstructured meshes [18] whose update ordering is based on distances of grid nodes to some selective reference points. However, this extension cannot be easily used for surface meshes (e.g., in $\Re^3$) because Euclidean distances between nodes are not consistent with geodesic distances on the mesh.

Jeong and Whitaker propose the fast iterative method [11, 10] (FIM) to solve the heterogeneous Eikonal equation and anisotropic Hamilton–Jacobi equations efficiently on parallel architectures. The FIM manages the list of active nodes and iteratively updates the solutions on those vertices until they are consistent with their neighboring vertices. Vertices are added to or removed from the list based on a convergence criterion, but the management of this list does not entail an extra burden of expensive ordered data structures or special updating sequences. Proper management of the list ensures consistency of the entire solution. This paper builds on the FIM algorithm, and describes the application to unstructured meshes and an implementation on a streaming SIMD parallel architecture.

In this paper we propose a new computational technique to solve the Eikonal equation on triangulated surface meshes efficiently on parallel architectures; we call it the *mesh fast iterative method* (meshFIM), because it is an extension of the FIM method proposed in [11]. We describe a parallel implementation of meshFIM on shared memory parallel systems and propose a new data structure for the efficient mapping of unstructured meshes for parallel SIMD processors with limited high-bandwidth memory. The contributions of this paper are twofold. First, we introduce the meshFIM algorithms for both single processor and shared memory parallel processors and perform a careful empirical analysis by comparing them to the state-of-the-art CPU-based method, the fast marching method (FMM), in order to understand the benefits and limitations of each method. Second, we propose a patch-based meshFIM solver, specifically for more efficient implementation of the proposed method on massively parallel SIMD architectures. We describe the detailed data structure and algorithm, present the experimental results of the patch-based meshFIM, and compare them to the results of the CPU-based methods to illustrate how the proposed method scales well on state-of-the-art SIMD architectures.

The paper proceeds as follows. In the next section we describe relevant work from the literature. In section 2 we introduce the proposed method and its hierarchical implementation for SIMD parallel architectures. In section 3 we show numerical results, including consistency and convergence, on several different examples with different domains and speed functions, and we compare the performance against the fast marching method. In section 4 we summarize the paper and discuss future research directions related to this work.

2. Fast iterative method (FIM) on unstructured meshes

2.1. Notation and definitions

In this paper, we consider the numerical solution of the Eikonal equation (2.1), a special case of nonlinear Hamilton–Jacobi partial differential equations (PDEs), defined on a 2D manifold with a scalar speed function

$$\begin{cases} H(\mathbf{x}, \nabla\phi) = |\nabla_{\mathscr{S}}\phi(\mathbf{x})|^2 - \frac{1}{f^2(\mathbf{x})} = 0 \forall \mathbf{x} \in \mathscr{S} \subset \Re^3, \\ \phi(\mathbf{x}) = B(\mathbf{x}) \forall \mathbf{x} \in \mathscr{B} \subset \mathscr{S}, \end{cases} \tag{2.1}$$

where $\mathscr{S}$ is a smooth 2D manifold in $\Re^3$, $\nabla_{\mathscr{S}}$ is the gradient operator in the tangent plane to the manifold, $\phi(\mathbf{x})$ is the travel time or distance from the source, $f(\mathbf{x})$ is a positive speed function defined on $\mathscr{S}$ and $\mathscr{B}$ is a set of smooth boundary conditions, which adhere to the consistency requirements of the original equation. Of course, a 2D, flat domain is a special case of this specification, and the proposed methods are appropriate for that scenario as well. The solution of the Eikonal equation with an arbitrary speed function is sometimes referred to as a weighted distance [25] as opposed to a Euclidean distance for a constant speed function on flat domains. We approximate the solution on a triangulation of $\mathscr{S}$ denoted $\mathscr{S}_T$. The solution is represented pointwise on the set of vertices V in S_T, and interpolated across the triangles with linear basis elements. The ith vertex in V is denoted υ_i and its position is a 3-tuple and denoted $\mathbf{x}_i = (x, y, z)$, where $x, y, z \in \Re$. An *edge* is a line segment connecting two vertices (υ_i, υ_j) in $\Re^3$ and is denoted $e_{i,j}$ while the vector from vertex υ_i to vertex υ_j is denoted $\mathbf{e}_{i,j}$ which equals to $\mathbf{x}_j - \mathbf{x}_i$. The angle between $e_{i,j}$ and $e_{i,k}$ is denoted $\angle_i$ or $\angle_{j,i,k}$.

The neighbors of a vertex are the set of vertices connected to it by edges. A triangle, denoted $T_{i,j,k}$, is a set of three vertices υ_i, υ_j, υ_k that are each connected to the others by an edge. We assume the triangulation adheres to a typical criterion for consistency for 2D manifolds, e.g., edges not on the boundary of the domain belong to two triangles, etc. We call the vertices connected to υ_i by an edge the *one-ring neighbors* of υ_i and the triangles sharing vertex υ_i are the *one-ring triangles* of υ_i. For example, in Figure 2.1 left, the vertex υ_1 is the neighbor of vertex υ_2 and vice-versa. Vertices υ_2, υ_3, υ_4, υ_5, υ_6, υ_7 constitute the one-ring of υ_1, and triangles $T_{1,2,3}$, $T_{1,3,4}$, …, $T_{1,2,7}$ (which we will denote with capital letters for multi-indices T_A, …, T_F as in the figure) form the one-ring triangles of υ_1. We define the discrete approximation to ϕ at vertex υ_i to be Φ_i.

2.2. Local solver

In (2.1), domain $\mathscr{S}$ is a manifold for which we have a tessellation $\mathscr{S}_T$ and the numerical solution of the equation $\Phi(\mathbf{x})$ is defined on the vertices of the triangles of the tessellation. The solution at each vertex, sometimes referred to as the *travel time*, is computed from its current value and its one-ring neighbors (see Figure 2.1 left), using a linear approximation of the solution on each triangular facet. The formulation presented here is a constructive form of derivation in [18], which describes a Godunov approximation that picks an upwind direction of travel for the characteristics based on consistency of the resulting solution. For a single update of a single vertex υ_i, a set of n potential solutions ($n = 6$ for υ_1 in Figure 2.1 left) are calculated for the n one-ring triangles. Each of these triangle solutions represents the shortest path across that triangle from the boundary conditions, as described in the following paragraphs. The approximated solution at vertex υ_i, $\Phi_i \approx \phi(\mathbf{x}_i)$, is set to be the minimum among the n values associated with each triangle in the one-ring. From a computational point of view, the bulk of the work is in the calculation of the n temporary or potential solutions from the adjacent triangles of each vertex.

The specific calculation on each triangle is as follows. Consider the triangle $T_{1,2,3}$ in Figure 2.1 right. We use an upwind scheme to compute the solution Φ_3 from values Φ_1 and Φ_2 to comply to the causality property of the Eikonal solution [18]. We consider a local scheme based on piecewise linear reconstructions within the triangle. The characteristics are perpendicular to the gradient of Φ, which is linear, and thus the travel time to υ_1 must be determined by time associated with a line segment lying in the triangle $T_{1,2,3}$.

Because acute triangles are essential for proper numerical consistency [14], we consider only the case of acute triangles here and discuss obtuse triangles subsequently. For a triangle $T_{1,2,3}$ in Figure 2.1 right, we denote the angles formed by the triangular edges as $\angle_1 = \alpha$, $\angle_2 = \beta$, and $\angle_3 = \gamma$, and denote the edge lengths as $\|\mathbf{e}_{1,2}\| = c$, $\|\mathbf{e}_{1,3}\| = b$, and $\|\mathbf{e}_{2,3}\| = a$. We assign a constant speed f to each triangle, $T_{1,2,3}$, which is consistent with a symmetric (isotropic) speed and a linear solution on each element. We denote the difference in travel time between υ_1 to υ_2 as $\Phi_{1,2} = \Phi_1 - \Phi_2$.

If the vertices υ_1 and υ_2 are upwind of υ_3, then there is a characteristic passing through υ_3 that intersects edge $\mathbf{e}_{1,2}$ at position $\mathbf{x}_\lambda = \mathbf{x}_1 + \lambda\mathbf{e}_{1,2}$, where λ is unknown and $\lambda \in [0, 1]$ in order for the characteristic to intersect the edge. The line segment that describes the characteristic across $T_{1,2,3}$ is $\mathbf{e}_{\lambda,3} = \mathbf{e}_{1,3} - \mathbf{e}_{1,\lambda} = \mathbf{e}_{1,3} - \lambda\mathbf{e}_{1,2}$. Thus the travel time from $\mathbf{x}_\lambda$ to $\mathbf{x}_3$ is $\Phi_{\lambda,3} = f\|\mathbf{e}_{\lambda,3}\| = f\|\mathbf{e}_{1,3} - \lambda\mathbf{e}_{1,2}\|$.

Because the approximation of the solution on the triangle $T_{1,2,3}$ is linear, we have $\Phi_\lambda = \Phi(\mathbf{x}_\lambda) = \Phi_1 + \lambda\Phi_{1,2}$. The solution at υ_3 is the solution at $\mathbf{x}_\lambda$ plus the travel time from $\mathbf{x}_\lambda$ to the vertex υ_3, and therefore

$$\Phi_3 = \Phi_\lambda + \Phi_{\lambda,3} = \lambda\Phi_{1,2} + \Phi_1 + f\|\mathbf{e}_{1,3} - \lambda\mathbf{e}_{1,2}\|. \tag{2.2}$$

All that remains is to find λ, and for this we observe that λ should minimize Φ_3 because the characteristic direction is the same as the gradient of the solution. Assigning zero to the derivative (with respect to λ) of (2.2) gives a quadratic equation from which we solve for λ. To satisfy the causality condition, λ must be in the range of [0, 1]. If the solved λ is in [0, 1], we compute Φ_3 from (2.2), else we compute two Φ_3's from (2.2) assuming λ as 0 and 1, and take the smaller one.

Because the computation of the solution for linear, triangular elements have poor approximation properties when applied to obtuse triangles [20], we have to treat obtuse triangles as a special case. For this, we adopt the method used in [14]. As illustrated in Figure 2.2, if $\angle_3$ is obtuse, we connect υ_3 to the vertex υ_4 of a neighboring triangle and thereby cut the obtuse angle into two smaller angles. If these two angles are both acute, then we are done as shown in the left picture of Figure 2.2; otherwise if one of the smaller angles is still obtuse, then we connect υ_3 to the vertex υ_5 of another neighboring triangle. This process is performed recursively, until all new angles at υ_3 are acute as shown in the right image of Figure 2.2. Note that, algorithmically, these added edges and triangles are not considered part of the mesh; they are used only in the solver for updating the solution at υ_3.

2.3. meshFIM updating scheme

The original *fast iterative method* [11] for solving the Eikonal equation was proposed for rectilinear grids. In this section, we extend the method to unstructured triangular meshes, called *meshFIM*, in a way that is appropriate for more general simplicial meshes. We begin with a serial (single-threaded) version of the algorithm, and then describe a parallel (multithreaded) version of meshFIM for shared memory system. Finally, we describe the algorithm for SIMD, streaming architectures with limited (hierarchical) shared memory

capabilities in detail. Here we mention the properties that make FIM suitable for parallel solutions of the Eikonal equation, because they govern some of the subsequent design choices:

1. The algorithm does not impose a particular update sequence.
2. The algorithm does not use a separate, heterogeneous data structure for sorting.
3. The algorithm is able to simultaneously update multiple points.

The strategy of meshFIM is to solve the Eikonal equation on triangular mesh vertices with lightweight data structures for easy mapping to SIMD architectures with fast access to limited amounts of high-speed memory. This is the basic model of state-of-the-art streaming architectures [4]. As in FIM [11], meshFIM maintains a data structure that represents a narrow computational band, a subset of the mesh, called the *active list*, for storing the vertices that are being updated. During each iteration, the list of active vertices/triangles is modified to remove vertices whose solutions are consistent with their neighbors and to include vertices that could be affected by the last set of updates. Thus, a vertex is removed from the active list when its solution is up to date with respect to its neighbors, and a vertex is appended to the list when the value of any potentially upwind neighbor has changed.

Convergence of the algorithm to a valid approximation of the Eikonal equation is provable [11] if three conditions are met:

1. Any vertex whose value may be inconsistent with its neighbors (according to the local solver) must be appended to the active list.
2. A vertex is removed from the active list only when its value is consistent with its neighbors.
3. The algorithm terminates only when the active list is empty.

There are a variety of algorithms that meet these criteria. Indeed, FMM is a special case of this philosophy, which adopts a particular update order that guarantees that once a point is removed from the active list it will never again need to be added (it is upwind of every subsequent update of vertex/grid values). In the remainder of this section we will discuss rules for updating vertex values and managing the active list that are efficient for arbitrary ordering of vertex-value updates, including update schemes that include both synchronous and asynchronous update of the active list.

Before the computation of the solution, any algorithm must compute certain static information about the mesh, including the speed for each triangle and values of the boundary conditions, and initialize the appropriate data structures, in this case the active list L, which is set to be all of the vertices adjacent to the boundary conditions. The computation of the speed function depends on the application, and the initialization of the active list is not a computationally important step; thus we do not treat the initialization as an important aspect of the parallel algorithms presented in this paper.

We begin with the basic algorithm, which assumes synchronous updates of the entire active list, and then introduce alternatives that take better advantage of asynchronous updates. In this context an *iteration* is one loop through the entire active list. In the basic algorithm, for every vertex $\upsilon_j \in L$ we compute the new Φ_j from solutions on the one-ring. This solution puts each vertex into a consistent solution with the values of its neighbors from the previous iteration, and thus all vertices, nominally are removed from the active list. Each updated vertex, however, triggers the activation of neighbors of greater value, which are potentially downwind. The algorithm would continue to update each subsequent active list until the active list is empty.

If we consider asynchronous updates, values that are potentially downwind of others in the active list may take advantage of updated values from the current iteration. Indeed, taken to the limit, the updates are done on individual nodes, one at a time, proceeding from the node of lowest value—which is the FMM algorithm. For parallel algorithms, the approach will be a mixture of synchronous updates among processors and asynchronous updates as each processor proceeds with a particular subset of the active list. The situation becomes more complicated when we consider the limited amount of communication that is available between processors or blocks of processors, which motivates processing multiple iterations on subsets of the domain without exchanging data or updating boundary conditions. In such cases, it is sometimes a more effective use of computational resources to run multiple iterations on the same set of active nodes, not removing each one from the list after updating, so that they can take advantage of updates of neighbors. The particular choice of updating strategy depends on the architecture, and in the sections that follow these choices are described for three different computational scenarios.

2.4. Algorithms for CPU

The criteria for a correct algorithm would suggest that a vertex could be removed from the list and its neighbors activated after a single update—knowing that it will be reactivated as needed. However, in the absence of a strict or approximate sorting of values in the active list, it is efficient to reconcile the values of vertices on the current wavefront (active list), before retiring updated vertices and including new ones. From this insight, we derive the proposed algorithm, which is as follows. Nodes on the active list are updated one at a time. After each node is updated, its value is consistent of its upwind neighbors, and each update is immediately transferred to the solution to be used by subsequent updates. The algorithm loops through the active list, continuously updating values, and when it reaches the last element of the list simply starts again at the beginning—thus, there is effectively no beginning or end to the list. A vertex remains on the active list until the difference between its old value and new value is below some error tolerance—effectively, it does not change from the last update. We refer to a vertex that does not change value (to within tolerance ε) as ε-*converged*. Each ε-converged vertex is removed from the active list. As the converged vertex is removed from the active list, all of its potentially downwind neighbors (neighbors of greater value) undergo one update step. If their values are not ε-converged (i.e., they change significantly), they are appended to the active list. The algorithm continues looping through the active list until the list is empty.

Table 2.1 compares the number of solution updates between FMM, strict synchronous and asynchronous relabeling schemes, and the proposed mesh fast iterative method (meshFIM). The FMM is optimal (although run times will be slightly offset by the time involved in managing the heap), and the synchronous and asynchronous schemes perform very poorly. The asynchronous scheme depends, in principle, on update order, but these results are consistent across a set of experiments with random permutations of the active list. This table also shows that the update strategy of the FIM, while not optimal provides numbers of updates that are much closer to FMM, and showed a robustness to the ordering of the active list.

Because the serial algorithm does not depend significantly on the ordering of updates, the extension to multiple processors is immediate. We simply divide the active list arbitrarily into N sublists, assign the sublists to the N threads, and let each thread use an asynchronous update for the vertices within the sublist. These updates are done by applying the updating step in Algorithm 2.1 to each subactive list.

Algorithm 2.1

meshFIM(*V, B, L*)

comment: 1. Initialization (*V* : all vertices, *L* : active list, *B*: seed vertices)

for each $\upsilon \in V$

do { **if** $\upsilon \in B$ **then** $\Phi_\upsilon \leftarrow 0$ **else** $\Phi_\upsilon \leftarrow \infty$

for each $\upsilon \in V$

do { **if** any 1-ring vertex of $\upsilon \in B$ **then** add υ to *L*

comment: 2. Update vertices in *L*

while*L* is not empty

do { for each $\upsilon \in V$

do { $p \leftarrow \Phi_\upsilon$; $q \leftarrow$ Update(υ); **if** $|p - q| < \varepsilon$

then { **for each** adjacent neighbor υ_{nb} of υ

do { **if** υ_{nb} is not in *L*

then { $p \leftarrow \Phi_{\upsilon_{nb}}$; $q \leftarrow$ Update(υ_{nb}); **if** $p > q$

then { $\Phi_{\upsilon_{nb}} \leftarrow q$; add υ_{nb} to *L* }

remove υ from *L*

2.5. Algorithm for GPU with SIMD parallel architecture

In this section, we describe the implementation of meshFIM for SIMD parallel architecture. First, we will briefly describe the SIMD parallel architecture, which motivates the proposed algorithm. Then we describe a GPU version of meshFIM, we call it patchFIM, to solve the Eikonal equation on the GPU with a SIMD parallel architecture.

2.5.1. SIMD architecture—Single-instruction multiple-datastream (SIMD) is a computational paradigm that decides the data level parallelism that is widely used in today's media processors and graphics processing units (GPUs). The SIMD model relies on a single instruction that is issued repeatedly on data structures that differ only in their memory location. A set of instructions and an associated data-stream (set of memory locations) is called a *thread*.

This SIMD computing strategy has some important implications for modern streaming architectures. First, architectures based on the SIMD model benefit from the efficiency that many arithmetic units share the same control structure, allowing for a dense packing of units on a processor. Second, the architecture provides a set of threads with extremely wide memory bandwidth to a relatively small pool of memory. This bandwidth, however, often relies on coherent memory access, so efficiencies are greatest when threads maintain local memory access patterns. Thus, effective use of this on-chip memory, called *shared memory*

or *local storage* and usually with limited amount, requires careful modification to the traditional programming model. Third, to the extent that instructions in threads can be unrolled, the execution can be organized into a large number of parallel pipelines that are simultaneously fetching data and producing and consuming intermediate results. In this way, this fast memory access is even more highly leveraged, typically to several hundred processors. Finally, SIMD architectures often rely on a hierarchy of computing modules that share a common, significantly slower, set of memory, often called *global memory*. The second tier computing modules (sometimes called *blocks*) consist of a set of individual threads (operating in a SIMD mode), but they can operate independently, and thus they can be considered as a set of parallel, shared memory units. The relatively slow access to the global memory that is shared among blocks means that interblock communication is slow relative to the rate at which threads consume data. For efficiency, algorithms must not rely on excessive communication between blocks.

2.5.2. patchFIM description and algorithm—To make good use of the GPU performance advantage, we propose a variant of meshFIM, called patchFIM, that scales well on SIMD architectures, using a patch-based update scheme. The main idea is splitting the computational domain (mesh) into multiple nonoverlapping patches (sharing only boundary vertices), and treating each patch, which will be processed in a SIMD fashion in a single block, as a computing primitive, corresponding logically to a vertex in the original meshFIM algorithm.

The active list maintains a set of *active patches* instead of active vertices, and a whole active patch is moved from global memory to a block and updated for several SIMD iterations, which we call *internal iterations*. A set of internal iterations comprises a single *iteration* for that patch. Thus for each patch iteration, the data for that patch is copied to the shared memory space, and internal iterations are executed to update the solution on that patch. Of course, multiple computing blocks can process multiple patches simultaneously, while other patches wait in global memory to be swapped out to blocks.

This patch strategy is meant to take advantage of the SIMD parallelism, but it introduces some inefficiencies. For instance, an entire patch must be activated any time a vertex in an adjacent patch gets updated. A patch must remain active as long as any of the vertices are still active. The number of internal iterations is required to offset of the cost of transferring data between memory caches; however, vertices within a patch are updated without communication with adjacent patches, and thus boundary conditions lag and may be out of date as the internal iterations proceed.

These inefficiencies must be justified by an effective SIMD algorithm for the patches. There are two challenges. First is providing SIMD processing on the unstructured mesh, and second is keeping the computational density sufficiently high. The parallelism is obtained by introducing a data structure for SIMD computing on unstructured meshes, which we call the *cell-assembly* data structure (terminology adapted from the finite element method (FEM) literature). Specifically, the *cell-assembly* data structure includes three arrays, labeled mnemonically **GEO**, **VAL**, and **NBH**. **GEO** is the array storing per-triangle geometry and speed information required to solve the Eikonal equation. It is divided into subsegments with a predefined size that is determined by the largest patch among all. Each subsegment stores a set of four floats for each triangle, i.e., three floats for triangle edge lengths and one float for the speed value. **VAL** is the array storing per-triangle values of solution of the Eikonal equation. It is divided into subsegments, similar to **GEO**, but instead of geometric information, solutions on three vertices are stored. We use two **VAL** arrays, one is for input and the other is for output, to avoid memory conflicts. To deal with boundaries across patches, we simply duplicate and store the exterior boundary vertices for each patch and

treat the data on those vertices as fixed boundary conditions for each patch iteration. The **NBH** array stores indices to **VAL** for the per-vertex solution. Figure 2.3 depicts the data structure introduced above.

A single inner iteration on a patch proceeds in two steps. In the first step all of the triangles produce the arrival time for the solution for each vertex of the triangle from the opposite edge, with special values for invalid results, as above. The triangle solutions all undergo the same computation, with some minor branching in the determination of valid solutions. In the second step all vertices are updated by referring back to the appropriate data in triangle solutions and performing a min operation on the valid solutions (assembly). The vertex computation must loop through all of the triangles in the one-ring, and thus the run-time of this step is determined by the vertex with highest valence in the patch. Thus, SIMD efficiency favors meshes with relatively consistent valences.

Preprocessing: The patchFIM algorithm requires some preprocessing before the iterations begin. First, we must partition the mesh into patches. We use the multilevel partitioning scheme described in [12]. It partitions the vertices of a mesh into roughly equal patches, such that the number of edges connecting vertices in different parts is minimized. The particular algorithm for mesh partitioning is not important to the proposed algorithm, except that efficiency is obtained for patches with similar numbers of vertices/triangles and relatively few vertices on the boundaries.

In this step, we also calculate the static mesh information including dealing with the obtuse triangles. We use the same idea as in meshFIM to treat obtuse triangles. However, instead of adding *virtual edge*, we also add *virtual triangles* generated by splitting the obtuse triangle to the corresponding cell-assembly data structures. Figure 2.4 demonstrates this, where $\angle_{1,3,2}$ is obtuse, and adding a *virtual edge* $e_{3,4}$ will generate two "virtual triangles" $T_{1,3,4}$ and $T_{2,3,4}$. If one of $\angle_{1,3,4}$ and $\angle_{2,3,4}$ is still obtuse, the algorithm would split again. The last thing in this step is to initialize values of each vertices and the active list. Instead of keeping a narrow band of active vertices, we maintain a list of active patches. If any of the vertices in a patch is adjacent to a seed point, this patch is added to the initial active list.

Iteration step: In this step, each patch is treated just like a vertex in meshFIM. The main iteration continues until the active list becomes empty. Each patch in the active list is assigned to a SIMD computing unit where all vertices value in this patch are updated several times. After every update, the assembly stage reconciles the different solutions for a vertex. This is done with a loop over the **NBH** to find the minimum value. If a patch is convergent, meaning all vertices in this patch are convergent, it is removed from the active list and its nonconvergent neighbor patches are added to the active list.

Checking the patch convergence can be simply updating the entire patch once and checking if there exists a vertex whose solution has changed by the update. To do this, we use a reduction operator, which is commonly used in the streaming programming model to reduce a larger input stream to a smaller output stream. For SIMD architectures, parallel reduction can be implemented using an iterative method. In each iteration, we adopt a tree-based method in which every thread reads two Boolean values from the convergence array of current patch and writes back the result of the AND operation of two values. The number of the threads to participate in this reduction is halved in the successive iteration, and this is repeated until only one thread is left. In this way, for a block of size n, only $O(\log_2 n)$ computations are required to reduce a block. In the pseudocode to follow, $C(p)$ is a Boolean value representing the convergence status of a patch p (per-patch convergence), and $C_v(p)$ is a set of Boolean values where each value represents the convergence status of the vertices in the patch p (per-vertex convergence). The pseudocode for patchFIM is given in Algorithm

2.2, where the pseudocode for each subroutine in the patchFIM is given in Algorithm 2.3, 2.4, and 2.5, respectively.

Algorithm 2.2

patchFIM($\mathbf{VAL}_{in}$,$\mathbf{VAL}_{out}$, L, P)

comment: L: active list of patches, P: set of all patches
while L is not empty
do
- MainUpdate(L, C_v, $\mathbf{VAL}_{in}$, $\mathbf{VAL}_{out}$)
- CheckNeighbor(L, C_v, C, $\mathbf{VAL}_{in}$, $\mathbf{VAL}_{out}$)
- UpdateActiveList(L, P, C)

Algorithm 2.3

MainUpdate(L, C_v, $\mathbf{VAL}_{in}$, $\mathbf{VAL}_{out}$)

comment: 1. Main iteration
for each $p \in L$ in parallel
do
- **for** i = 1 to n
  do
  - **for each** $t \in p$ in parallel
    do
    - $\mathbf{VAL}_{out}(t) \leftarrow$ LocalSolver($\mathbf{VAL}_{in}(t)$)
    - reconcile solutions in t
  - update $C_v(p)$
  - swap $\mathbf{VAL}_{in}(t)$ and $\mathbf{VAL}_{out}(t)$
  - reconcile solutions in p

Algorithm 2.4

CheckNeighbor(L, C_v, C, $\mathbf{VAL}_{in}$, $\mathbf{VAL}_{out}$)

comment: 2. Check neighbors
for each $p \in L$ in parallel
do {$C(p) \leftarrow$ reduction($C_v(p)$)
for each $p \in L$ in parallel
do
- **if** $C(p)$ = TRUE
  then
  - **for each** adjacent neighbor of p_{nb} of p
    do {add p_{nb} to L

for each $p \in L$ in parallel
do
- **for each** $t \in p$ in parallel
  do
  - $\mathbf{VAL}_{out}(t) \leftarrow$ LocalSolver($\mathbf{VAL}_{in}(t)$)
  - reconcile solutions in t
- update $C_v(p)$
- swap $\mathbf{VAL}_{in}(t)$ and $\mathbf{VAL}_{out}(t)$
- reconcile solutions in p

for each $p \in L$ in parallel

do $\{C(p) \leftarrow \text{reduction}(Cv(p))$

Algorithm 2.5

UpdateActiveList(L, P, C)

comment: 3. Update active list

clear(L)

for each $p \in P$

do { **if** $C(p)$ = FALSE
then insert p to L

3. Results and discussion

In this section we discuss the performance of the proposed algorithms in realistic settings compared to the most popular FMM-based algorithm. We have conducted systematic empirical tests with a set of different meshes with various speed functions. First, we show the result of the single-threaded (serial) CPU implementation of meshFIM and FMM, and discuss the intrinsic characteristics relative to existing algorithms. Second, we provide the result of multithreaded CPU implementation to discuss scalability of the proposed algorithm on shared memory multiprocessor computer systems. Last, we show the GPU implementation to demonstrate the performance of the proposed method on SIMD parallel architectures. Single precision is used in all experiments throughout the entire paper. We have carefully chosen four triangular meshes with increasing complexity to compare the performance of each method. In addition, we used two different speed functions, a constant and correlated random speed, to elaborate how the heterogeneity of the speed function affects the performance of each method.

The meshes for the experiments in this section are as follows:

Mesh 1: A regularly triangulated flat square mesh with 1,048,576 vertices (1,024× 1,024 regular grid),

Mesh 2: An irregularly triangulated flat square mesh with 1,181,697 vertices and 2,359,296 triangles,

Mesh 3: A sphere with 1,023,260 vertices and 2,046,488 triangles (Figure 3.1 left), and

Mesh 4: Stanford dragon with 631,187 vertices and 1,262,374 triangles (Figure 3.1 right).

The speed functions $f(x)$ are: **Speed 1**—a constant speed of one, and **Speed 2**—correlated random noise.

3.1. Serial CPU results

We have tested our CPU implementation on a Windows Vista PC equipped with an Intel i7 920 CPU running at 2.66 GHz. First, we focus only on the performance of FMM and the single-threaded implementation of our method (meshFIM-ST) on different meshes with a constant speed (Speed 1). Rows 1 and 2 of Table 3.1 show the experimental results for the serial implementations.

The Eikonal equation with the speed function of constant one ($f(\mathbf{x}) = 1$) is the simplest test, and the easiest to perform well. However, it is useful in real world applications because the solution is the geodesic distance on a surface to the initial source boundary. In this experiment, we use a single point as the source for all four meshes so that the r-level set of the solution Φ is a curve that is a collection of all points on the surface whose distance to the source point is r. As shown in Table 3.1, FMM outperforms the single-threaded meshFIM slightly on all the test cases. Although FMM has the overhead of managing the heap data structure, the cost related to computing distance becomes the major bottleneck for the Eikonal equation on the mesh. Because meshFIM usually requires more iterations per vertex than FMM (which is optimal in this respect), meshFIM runs slower than FMM for serial execution.

To further elaborate the difference of two methods, we conducted the experiment on Mesh 3 using both speed functions. As shown in Table 3.2, the performance of FMM is not affected by the choice of the speed functions, which clearly demonstrates the advantage of the worst-case-optimal algorithm. On the other hand, the running time for meshFIM increased significantly from Speed 1 to Speed 2 because the total number of iterations (vertex updates) is significantly increased for Speed 2 due to the huge variance of the speed.

The meshFIM algorithm is designed for parallelism, and the results on the multithreaded system bear this out. The third row in Table 3.1 shows the running time of multithreaded meshFIM using four CPU cores. Because FMM is a serial algorithm (a strict ordering of the updates on vertices requires this), there is no benefit of using multiple threads. In contrast, meshFIM scales well on multicore systems. On a quad-core processor, we observed a nearly three times speedup from meshFIM-ST to meshFIM-MT on all cases. This result suggests that meshFIM is a preferred choice for such shared memory systems.

3.2. GPU implementation result

To show the performance of meshFIM on SIMD parallel architectures, we have implemented and tested patchFIM (Algorithm 2.2) on an NVIDIA GT200 GPU using NVIDIA CUDA API [15]. The NVIDIA GeForce GTX 275 graphics card is equipped with 896 MBytes of memory and 30 microprocessors, where each microprocessor consists of eight SIMD computing cores that run at 1404 MHz. Each computing core has 16 KBytes of on-chip shared memory for fast access to local data. The 240 cores run in parallel, but the preferred number of threads running on a GPU is much larger because cores are time-shared by multiple threads to maximize the throughput and increase computational intensity. Computation on the GPU entails running a kernel with a batch process of a large group of fixed size thread blocks, which maps well to the patchFIM algorithm that uses patch-based update methods. A single patch is assigned to a CUDA thread block, and each triangle in the patch is assigned to a single thread in the block. In order to balance the GPU resource usage, e.g., registers and shared memory, and the number of threads running in parallel, we fix the thread block size to result in the maximum occupancy [1] and adjust the maximum number of triangles among all patches to conform that.

Table 3.3 shows the performance comparison of patchFIM with two single-threaded CPU implementations (i.e., FMM and meshFIM) on the same meshes and speed functions, and shows the speedup factors of patchFIM over the CPU methods. Communication times between CPU and GPU, which are only about one tenth of the running times in our experiments, are not included for patchFIM to give a more accurate comparison of the methods. As shown in this result, the patchFIM algorithm maps very well to the GPU and achieves a good performance gain over both the serial and multithreaded CPU solvers. On a simple case such as Mesh 1 with Speed 1, patch-FIM runs about 33 times faster than meshFIM-ST and 25 times faster than FMM. On other more complex cases, patchFIM runs

up to 15 times faster than FMM. In addition, on the heterogeneous media using Mesh 3 with Speed 2, where meshFIM-ST runs roughly half as fast as FMM on the CPU, patchFIM still runs about 14 times faster than FMM.

As shown in this result, SIMD efficiency of the meshFIM algorithm depends on the input mesh configuration, more specifically, the average vertex valence relative to the highest valence. Thus, Mesh 1 is the most efficient set up because almost all vertices have valence six. In contrast, Mesh 2 shows the worst performance due to the highest vertex valence of 11. Meshes 3 and 4 have a maximum valence of eight. Moreover, Mesh 2 has the largest percentage of high valence (greater than six) vertices. Meshes 3 and 4 are commonly found set up where valences follow a tight, symmetric-distribution-centered valence six. In summary, patchFIM implemented on the GPU runs faster than any existing CPU-based solver on all examples we tested, with the effectiveness depending on mesh configuration and distribution of valences of vertices. Many applications based on time-consuming Eikonal equation solvers can run at real-time or interactive rates using the proposed method.

In patchFIM, there are two user-defined parameters: the size of patch and the iteration number within an active patch update. In our experiments, the empirically optimal patch size is 64 vertices, which means the maximum number of vertices among all patches is 64. There is a trade-off here. On the one hand, the smaller patch sizes efficiently concentrate vertex updates on the wavefront. This is because we update all the vertices of a patch each iteration, while only the updates for the vertices on the wavefront are useful. For smaller patch sizes, the average ratio of number of vertices inside the wavefront to the total number of vertices in this patch is higher, hence there is less percentage of useless computation. On the other hand, the SIMD architecture requires the patch size to be large enough to take advantage of the large number of processors and to hide the hardware latency [15] associated with memory transfers. A small parameter study of different patch sizes showed 64 vertices to be an effective compromise and that this parameter is consistent across different meshes.

3.3. Analysis of results

In the previous section, the performance of the Eikonal solvers are compared based on the running time on different architectures. Because running time can be affected by many factors, such as implementation schemes and hardware performance, we measure the number of local solver calls for a more precise performance analysis in this section. We also briefly discuss the accuracy of the proposed method, and introduce parameter optimization techniques for GPU implementation.

3.3.1. Asymptotic cost analysis—The most time-consuming operation for the Eikonal equation solver is the update of the solution on a vertex with its one-ring triangles, and each update includes N local solver calls where N is the valence of this vertex. Table 3.4 compares the average number of local solver calls per vertex on different meshes with different speed functions. As can be seen from Table 3.4, FMM requires approximately 18 local solver calls in all cases. This can be explained as follows. For FMM, the solutions of the vertices on the wavefront may be computed multiple times. Each vertex has six neighbors on the average, and statistically half of the neighbors are potentially upwind. Thus, each vertex is updated roughly three times, and each time requires a solve for the six triangles in the one-ring. This explains the characteristic 18 solves per vertex, independent of the meshes and speed functions. In comparison, the average number of local solver calls for meshFIM depends largely on the speed function, which can be noticed when comparing Speed 2 with Speed 1. In addition, the average number of local solver calls for meshFIM-ST is more than that of FMM on all the experiment settings. This difference in the number of

calls is offset, but only slightly, by the extra work of FMM in maintaining the heap. The multithreaded CPU version (meshFIM-MT) needs more updates because of the extra computation associated with simultaneous updates in the red-black Gauss–Seidel iteration scheme. This explains, to some extent, why we get about three times speedup on a quad-core CPU. The patchFIM method incurs an extra computation associated with patch-based updates. This factor of 5–20 is consistent with the run times we see. Roughly, if we have 200 processors operating at approximately half the clock rate, we would expect, ideally, a 100× advantage. However, with the efficiencies shown in Table 3.4, we would expect a 5–20× speed advantage on the GPU (relative to FMM), which is consistent with data in Table 3.3. These results also provide evidence that the CUDA implementation achieves a computational density that is high enough to offset latency and memory management overhead.

We can asymptotically compare the computational costs of the FMM and mesh-FIM algorithms as follows [11]. Let k_1 and k_2 be the costs for a local solver and a heap updating operation, respectively. Suppose P_{FMM} and P_{FIM} are the average number of local solver calls per vertex in FMM and meshFIM-ST, respectively (as in Table 3.4). Let h be the average heap size. The total costs for FMM and meshFIM-ST on a mesh with N vertices can be defined asymptotically as follows:

$$\begin{cases} C_{FMM}=N(k_1P_{FMM}+k_2P_{FMM}log_2h)=Nk_1P_{FMM}\left(1+\frac{k_2}{k_1}log_2h\right), \\ C_{FIM}=Nk_1P_{FIM}. \end{cases}$$

The value $\frac{k_2}{k_1}$ is empirically measured to be about 0.02. Hence, the ratio for the costs of meshFIM and FMM is $\frac{C_{FIM}}{C_{FMM}}=\frac{P_{FIM}}{P_{FMM}(1+0.02log_2h)}$.

For the setting with Mesh 1 and Speed 1, the average heap size h (which is proportional to the arc length of the expanding wavefront) is 1,302 for FMM and $\frac{P_{FIM}}{P_{FMM}}$ is approximately 1.67, as can be derived from Table 3.4. Therefore, $\frac{C_{FIM}}{C_{FIM}} \approx 1.38$ in this case, which is consistent with the experimental results in Table 3.3.

As shown in the above analysis, $k_1 \gg k_2$ in C_{FMM}, so the impact of the update operations on the performance of FMM is much more significant than that of the heap operations for moderately sized meshes. This is juxtaposed with the lower cost of computing node updates on regular grids, which makes FIM more competitive with FMM in that circumstance, even for serial implementations [11]. It can also be seen that, with a larger mesh (which means larger h), the performance difference between single-threaded meshFIM (C_{FIM}) and FMM (C_{FMM}) will be less. Of course the design goal of meshFIM is that it can be mapped well to parallel architectures. Even with some performance degradation from Gauss–Seidel iteration in meshFIM-ST to red-black Gauss–Seidel in meshFIM-MT, we can still get large performance gain from running on multiple core CPUs.

The performance of meshFIM is determined by the number of updates (or the number of local solver calls), which depends heavily on the heterogeneity of the speed function. The following experiment systematically characterizes how the speed function affects the performance of these algorithms. First, we generate white noise for the initial speed function, and then apply a mesh Laplacian operator [30] N times to the initial speed function

to make the speed function less heterogeneous for increasing N. Figure 3.2 shows the result of this experiment. The x axis is the number of total vertices in the mesh and the y axis is the number of local solver calls. As N becomes bigger, and the speed function more homogeneous or smooth, the plot becomes less steep, and the results become closer to the meshFIM results with a constant speed function. We also see that FMM increases linearly with the number of vertices, as expected.

3.3.2. Error analysis—To show that the proposed algorithm achieves the first-order accuracy we would expect from the linear elements introduced in the solver, we performed a convergence analysis. We use seven regularly triangulated square meshes, representing a 16×16 patch of $\Re^2$, with the number of vertices ranging from 256 to 1,048,576. We considered two cases of boundary conditions. In the first case we used a pair of isolated points and in the second case we used a pair of circles of radius 3, where the domain is 16×16. Boundary conditions were projected onto the grid using the nearest vertices to the circles or points. We then solve for the distances to these boundaries for the entire domain using the patchFIM Eikonal solver and compare against analytical results at the vertices using the average squared error (L_2)—similar plots result from *sup* error. Figure 3.3(a) shows the level sets of a solution to the circular boundary conditions. Finally, we can plot these errors against the size of triangles as shown in Figure 3.3(b). For the circular boundary conditions, the slope of this graph is 1.0, which is consistent to our claim that meshFIM is first-order accurate. For the point boundary conditions, the slope is less—showing the method is not first-order accurate for nonsmooth boundaries, which are inconsistent with the governing equations.

3.3.3. Parameter optimization—As for the iteration number within an active patch update, every active patch is updated multiple times before its convergence is checked. There are two motivations. First, not all the vertices in a patch reach a consistent configuration with a single update. This is clear if we imagine a wavefront of active vertices initiated at one side of a patch propagating to the other side. The check for convergence requires communication with the CPU, and we would like to make maximum use of the fast on-chip shared memory space without communicating with the main memory. However, if the number of iterations per patch n is too large, the algorithm executes useless extra updates after reaching a consistent configuration. Generally n is proportional to the patch diameter, which is related to the number of iterations it takes for a wavefront to propagate across a patch. The optimal choice of n depends not only on the size of the patch but also on the input speed function. In general, according to our experiments, the best n is around 7 for most cases for patches of approximately 64 vertices. The running times for $n < 7$ can be quite good, but are not stable across different data sets and speed functions. However, for $n > 7$ the running time becomes stable and gradually increases as n increases. This is because patches with 64 vertices usually converge in about seven updates, and therefore wavefront propagation is almost identical with $n > 7$ iterations.

4. Conclusions

In this paper we propose a fast and easily parallelizable algorithm to solve the Eikonal equation on unstructured triangular meshes on single core CPU and on parallel, streaming architectures with restrictions on local memory. The proposed algorithms are based on the fast iterative method with modifications to accommodate unstructured triangular grids. The method employs a narrow band method to keep track of the mesh vertices to be updated and iteratively updates vertex values until they converge. Instead of using an expensive sorting data structure to ensure the causality, the proposed method uses a simple list to store active vertices and updates them asynchronously, using an ad-hoc ordering, which can be determined by the hardware. The vertices in the list are removed from or added to the list

based on the convergence, which is a measure of consistency with neighboring vertices. The method is easily portable to parallel architectures, which is difficult or infeasible with many existing methods. We compared the performance of the proposed method with the popular FMM method on a single processor, a shared-memory, multicore CPU, and a SIMD parallel processor.

Acknowledgments

The dragon model is provided by the Stanford University Computer Graphics Laboratory.

REFERENCES

1. CUDA Occupancy Calculator. available online at http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.
2. Bertsekas, D. Dynamic Programming and Optimal Control. Belmont, MA: Athena Scientific; 1995.
3. Bornemann F, Rasch C. Finite-element discretization of static Hamilton-Jacobi equations based on a local variational principle. Comput. Vis. Sci. 2006; 9:57–69.
4. Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P. Brook for GPUs: Stream computing on graphics hardware. ACM Trans. Graphics. 2004; 23:777–786.
5. Colli-Franzone P, Guerri L. Spreading of excitation in 3-D models of the anisotropic cardiac tissue I. Validation of the eikonal model. Math. Biosci. 1993; 113:145–209. [PubMed: 8431650]
6. Falcone M. A numerical approach to the infinite horizon problem of deterministic control theory. Appl. Math. Optim. 1987; 15:1–13.
7. Falcone, M. Motion by Mean Curvature and Related Topics. Berlin: de Gruyter; 1994. The minimum time problem and its applications to front propagation; p. 70-88.
8. Greivenkamp, JE. Field Guide to Geometrical Optics. Bellingham, WA: SPIE Publications; 2003.
9. Herrmann, M. Center for Turbulence Research Annual Research Briefs. Stanford, CA: Stanford University; 2003. A domain decomposition parallelization of the fast marching method; p. 213-225.
10. Jeong WK, Fletcher PT, Tao R, Whitaker RT. Interactive visualization of volumetric white matter connectivity in DT-MRI using a parallel-hardware Hamilton–Jacobi solver. IEEE Trans. Vis. Comput. Graph. 2007; 13:1480–1487. [PubMed: 17968100]
11. Jeong WK, Whitaker RT. A fast iterative method for eikonal equations. SIAM J. Sci. Comput. 2008; 30:2512–2534.
12. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. 1998; 20:359–392.
13. Keener JP. An eikonal-curvature equation for action potential propagation in myocardium. J. Math. Biol. 1991; 29:629–651. [PubMed: 1940663]
14. Kimmel R, Sethian JA. Computing geodesic paths on manifolds. Proc. Natl. Acad. Sci. USA. 1998; 95:8431–8435. [PubMed: 9671694]
15. NVIDIA. NVIDIA CUDA C Programming Guide. available online at http://developer.nvidia.com/nvidia-gpu-computing-documentation.html.
16. Pichon, E.; Tannenbaum, A. ICIP. Vol. Volume 2. Italy: Genoa; 2005. Curve segmentation using directional information, relation to pattern detection; p. 794-797.
17. Polymenakos LC, Bertsekas DP, Tsitsiklis JN. Implementation of efficient algorithms for globally optimal trajectories. IEEE Trans. Automat. Control. 1998; 43:278–283.
18. Qian J, Zhang Y, Zhao H. Fast sweeping methods for eikonal equations on triangulated meshes. SIAM J. Numer. Anal. 2007; 45:83–107.
19. Qin F, Luo Y, Olsen KB, Cai W, Schuster GT. Finite-difference solution of the eikonal equation along expanding wavefronts. Geophysics. 2009; 57:478.
20. Rawlinson R, Sambridge M. Wave front evolution in strongly heterogeneous layered media using the fast marching method. Geophys. J. Internat. 2004; 156:631–647.
21. Rouy E, Tourin A. A viscosity solutions approach to shape-from-shading. SIAM J. Numer. Anal. 1992; 29:867–884.

22. Sethian JA. A fast marching level set method for monotonically advancing fronts. Proc. Natl. Acad. Sci. USA. 1996; 93:1591–1595. [PubMed: 11607632]
23. Sheriff, R.; Geldart, L. Exploration Seismology. Cambridge, UK: Cambridge University Press; 1995.
24. Sifri, O.; Sheffer, A.; Gotsman, C. Proceedings of the IMR. Malaysia: Kuala Lumpur; 2003. Geodesic-based surface remeshing; p. 189-199.
25. Spira A, Kimmel R. An efficient solution to the eikonal equation on parametric manifolds. Interfaces Free Bound. 2004; 6:315–327.
26. Srinark, T.; Kambhamettu, C. Computer Graphics and Imaging. Canada: IASTED/ACTA Press, Calgary; 2003. A novel method for 3 D surface mesh segmentation; p. 212-217.
27. Tsai YR, Cheng LT, Osher S, Zhao H. Fast sweeping algorithms for a class of Hamilton-Jacobi equations. SIAM J. Numer. Anal. 2003; 41:673–694.
28. Tugurlan, MC. Ph.D. thesis. Baton Rouge, LA: Louisiana State University; 2008. Fast Marching Methods-Parallel Implementation and Analysis.
29. Wikipedia. Supercomputer. available online at http://en.wikipedia.org/wiki/Supercomputer.
30. Zhang, H.; Kaick, OV.; Dyer, R. Spectral methods for mesh processing and analysis. In: Schmalstieg, D.; Bittner, J.; Prague, editors. STAR Proceedings of Eurographics 2007. Geneva, Switzerland: Eurographics Association; 2007. p. 1-22.
31. Zhao H. A fast sweeping method for eikonal equations. Math. Comp. 2005; 74:603–627.

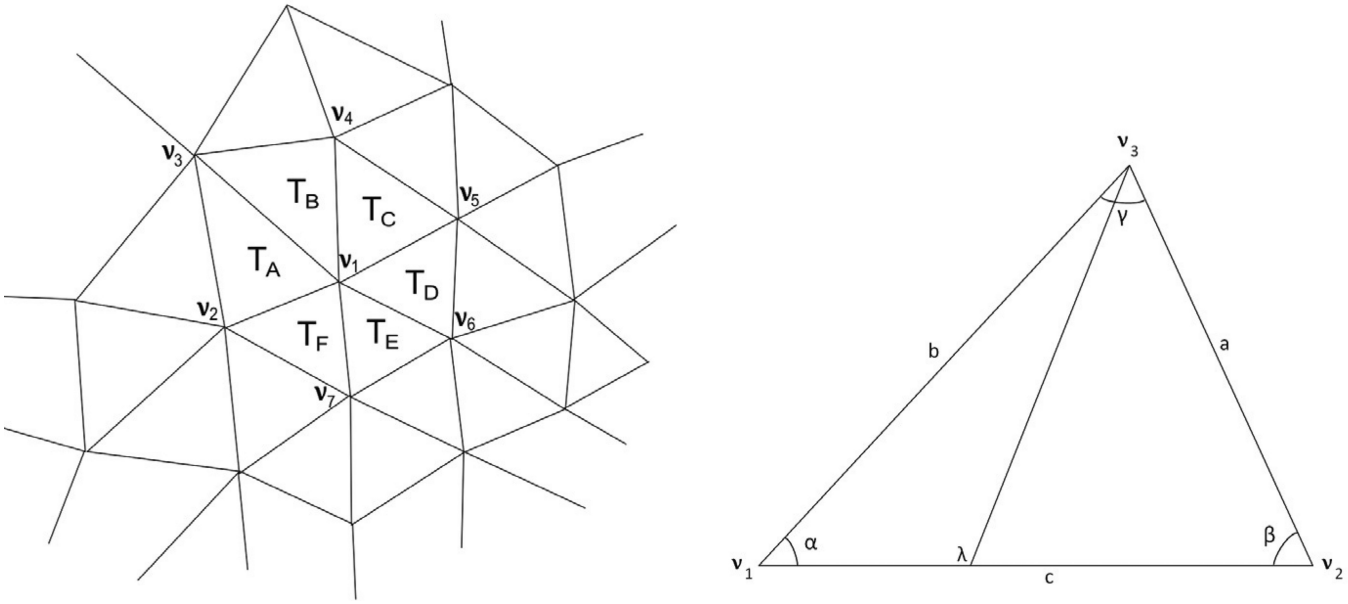

Fig. 2.1.
A triangulation $\mathcal{S}_T$ of surface $\mathcal{S}$ (left) and the local solver: update the value at vertex υ_3 in a triangle (right).

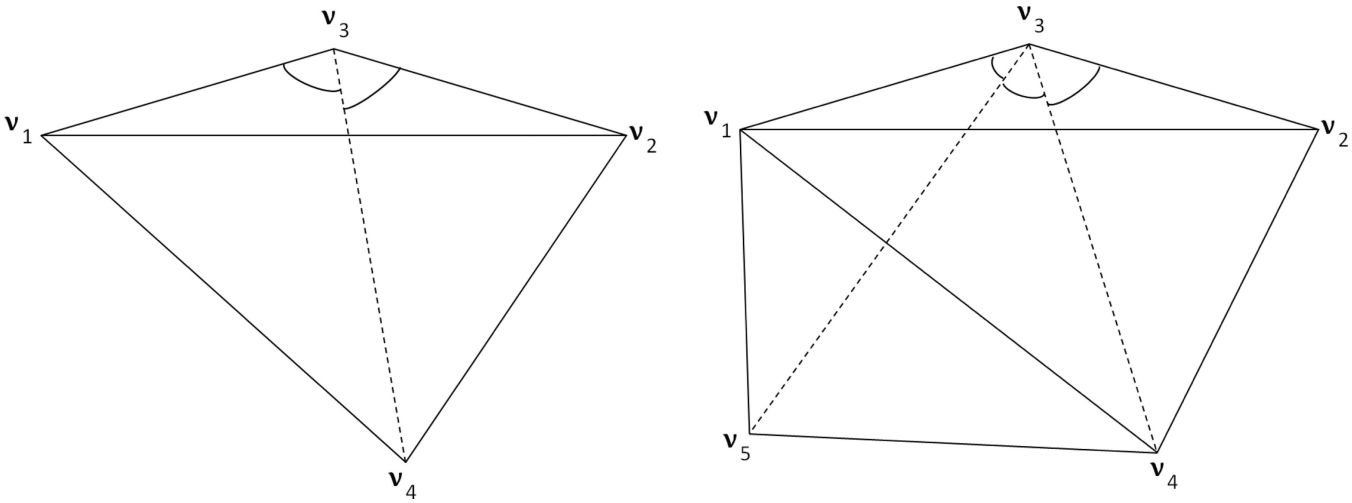

Fig. 2.2.
Strategy to deal with obtuse triangles.

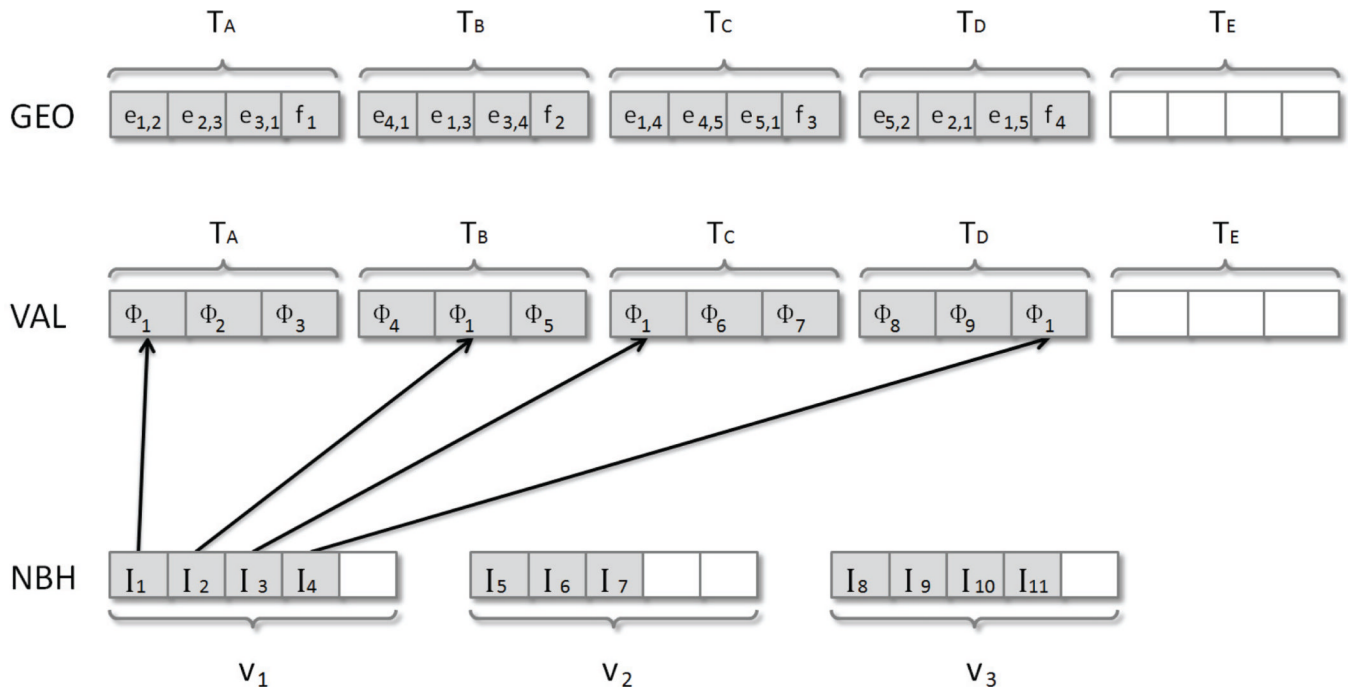

Fig. 2.3.

Data structure: in this figure, T_i is a triangle, $e_{i,j}$ represents the edge length, and f_i is the inverse of speed in a triangle. Φ_i means the value of the i_{th} vertex. I_i in NBH represents the data structure for the i_{th} vertex, each of which has q indices pointing (shown as arrows) to the value array.

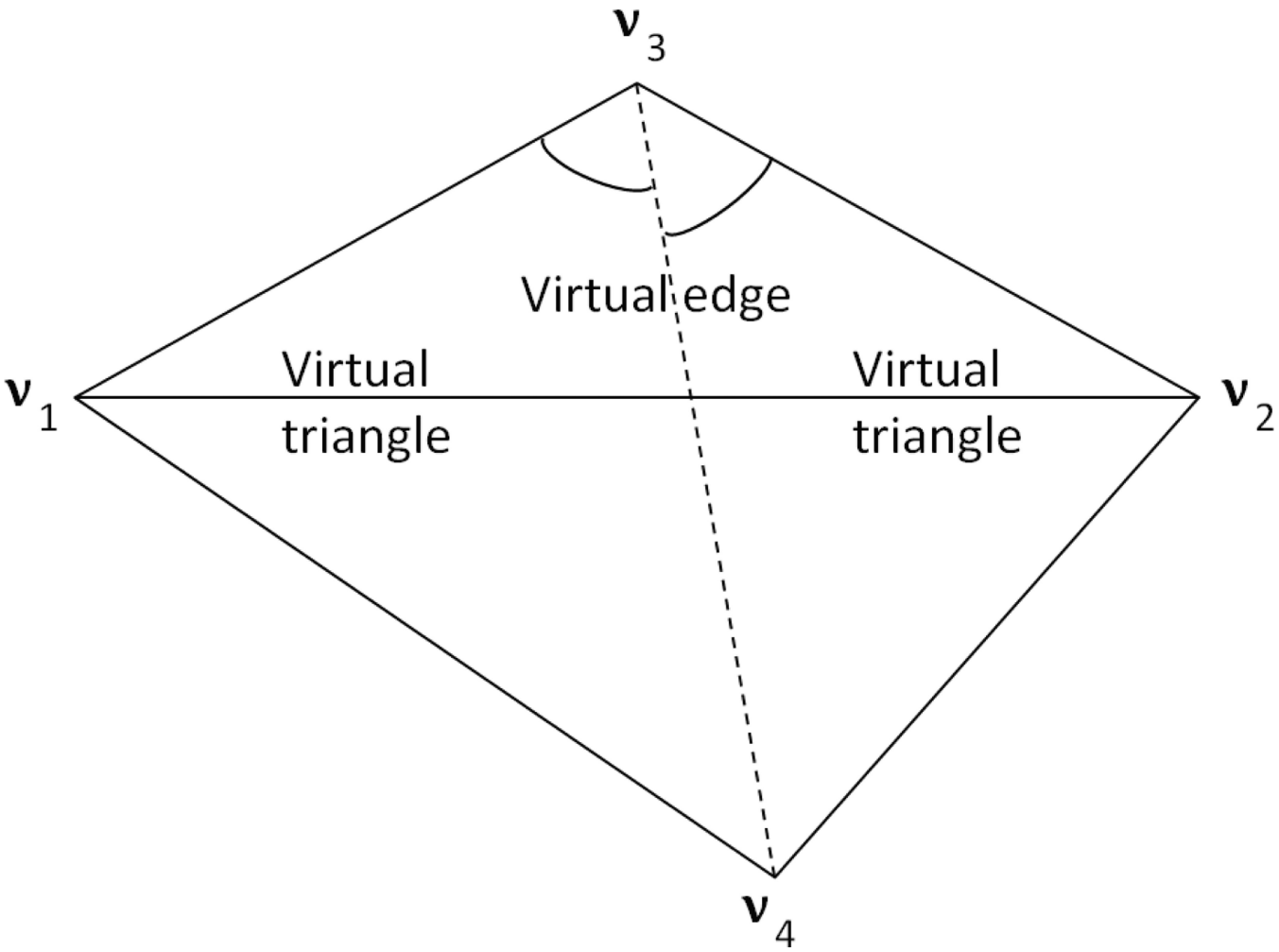

Fig. 2.4.
Virtual edge and virtual triangles.

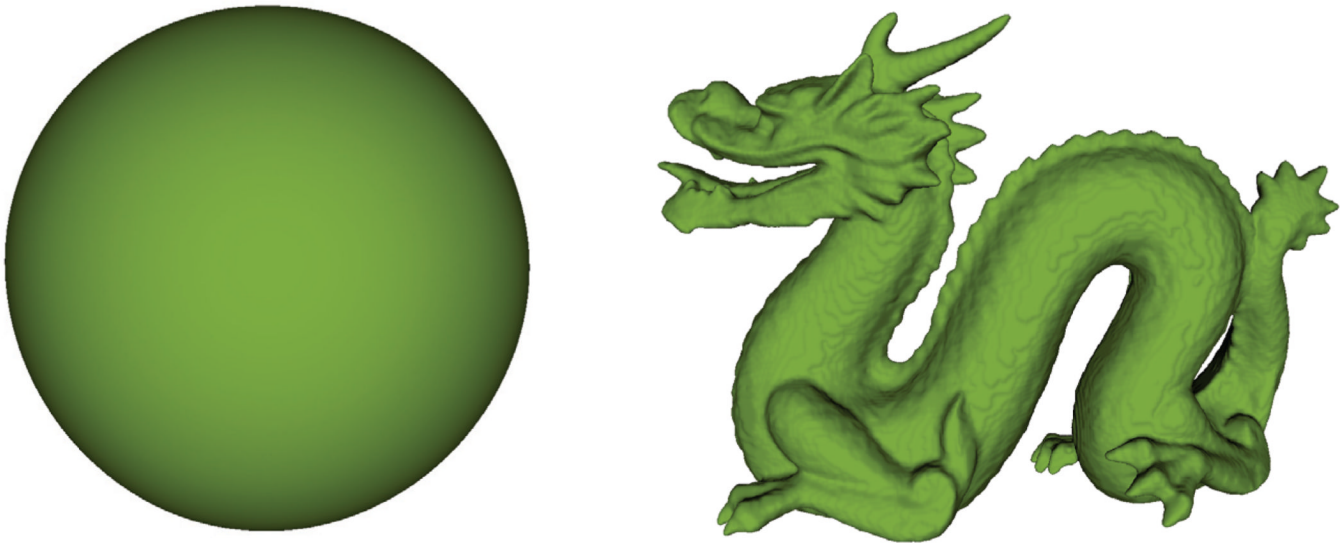

Fig. 3.1.
Sphere and Stanford dragon meshes.

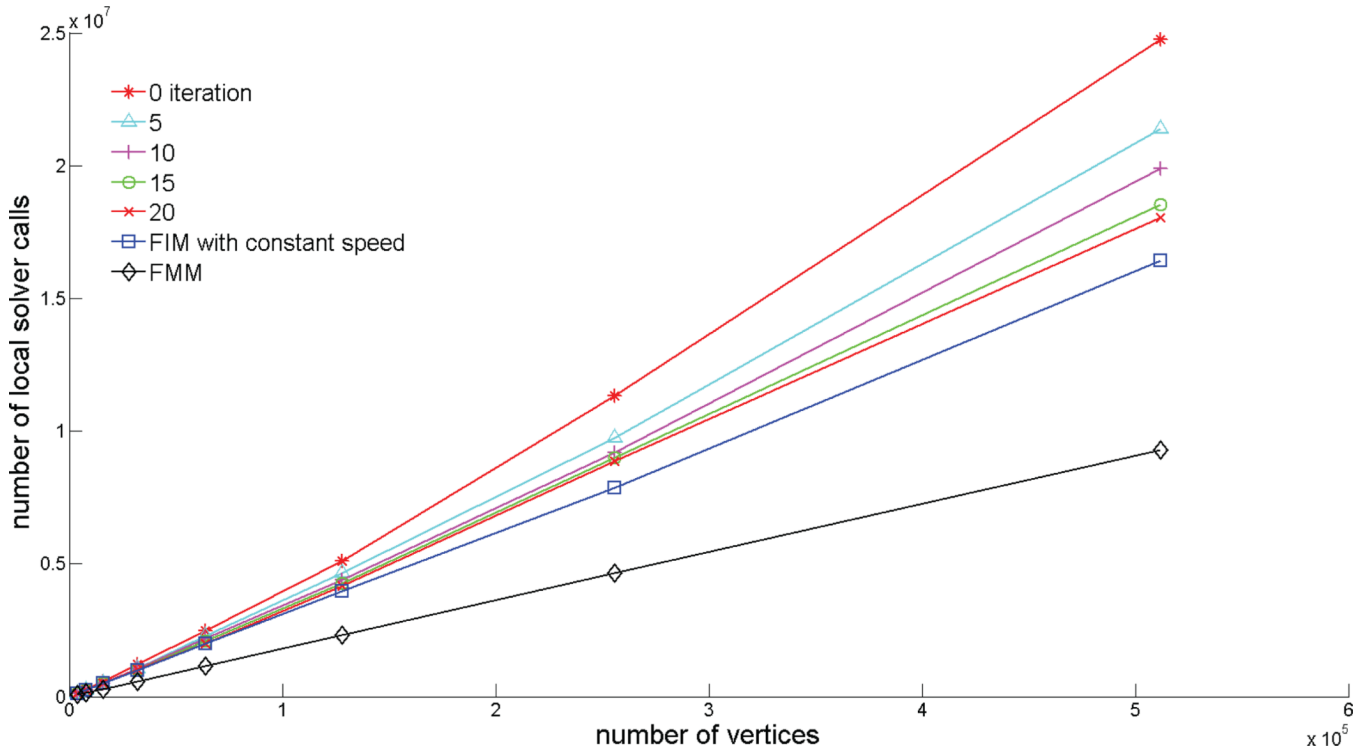

Fig. 3.2.
Laplacian experiment results.

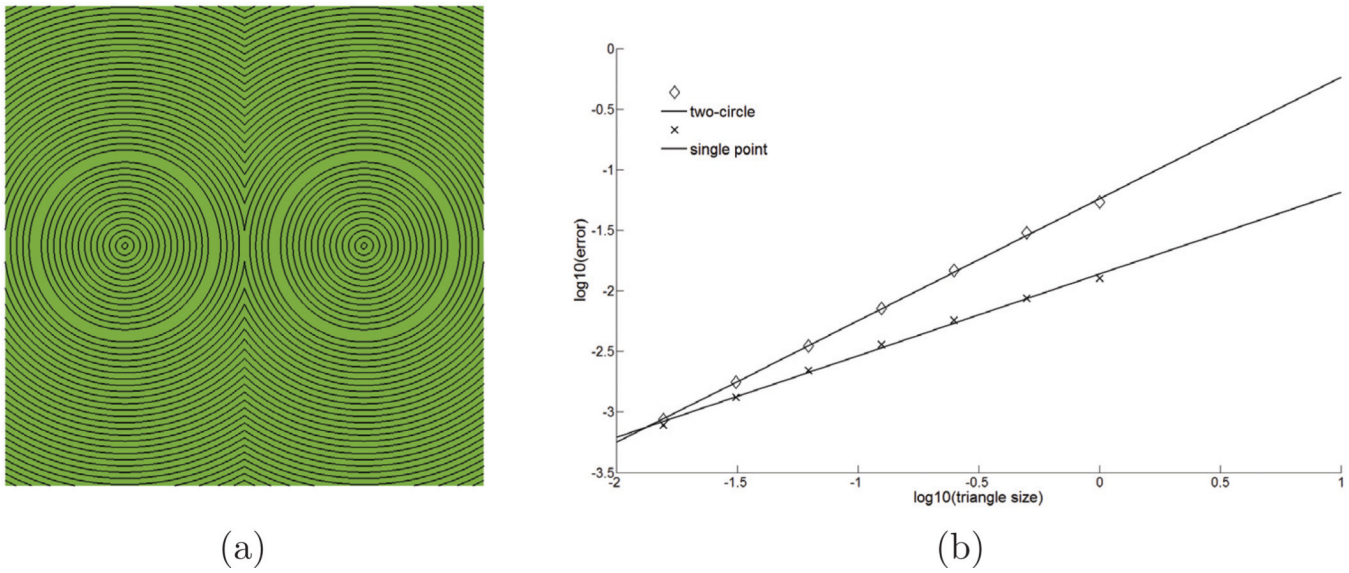

Fig. 3.3.

(**a**) The level sets of the solution of the Eikonal equation which represents distance to two circular boundaries. (*b*) The error as a function of resolution shows first-order convergence for smooth boundary conditions.

Table 2.1

Average number of local solver calls per vertex with the FMM, synchronous relabeling scheme, asynchronous relabeling scheme, and meshFIM for two different meshes—one simple and one complex (sphere and dragon described below).

	FMM	Synchronous	Asynchronous	meshFIM
Simple mesh	18.1	737.8	177.0	19.6
Complex mesh	18.3	671.7	175.2	59.2

Table 3.1

Running time (millisecond) of FMM, single-threaded FIM (meshFIM-ST), and multithreaded FIM (meshFIM-MT) on Meshes 1, 2, 3, and 4 with a constant speed (Speed 1).

	Mesh 1	Mesh 2	Mesh 3	Mesh 4
FMM	5092	7063	6362	3612
meshFIM-ST	6562	9354	8591	4331
meshFIM-MT	2198	3151	2846	1487

Table 3.2

Running time (millisecond) of FMM and meshFIM (single and multithreaded) on Mesh 3 and both speed functions (Speed 1 and 2).

	Speed 1	**Speed 2**
FMM	6362	6435
meshFIM-ST	8591	11960
meshFIM-MT	2846	4362

Table 3.3

Running times (milliseconds) and speedups (factor) for different algorithms and architectures.

	Mesh 1 with Speed 1	Mesh 2 with Speed 1	Mesh 3 with Speed 1	Mesh 4 with Speed 1	Mesh 3 with Speed 2
FMM	5092	7063	6362	3612	6435
meshFIM-ST	6562	9354	8591	4331	11960
patchFIM	201	910	415	287	459
Speedup over FMM	25×	8×	15×	13×	14×
Speedup over meshFIM-ST	33×	10×	21×	15×	28×

Table 3.4

Average number of local solver calls per vertex for different algorithms.

	Mesh 1 with Speed 1	Mesh 2 with Speed 1	Mesh 3 with Speed 1	Mesh 4 with Speed 1	Mesh 3 with Speed 2
FMM	17.9	19.5	18.1	18.3	18.1
meshFIM-ST	18.0	23.3	24.4	19.6	59.2
meshFIM-MT	18.0	26.6	46.1	23.1	83.1
patchFIM (GPU)	105.0	595.5	290.9	251.2	334.1