CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **27**:1639–1657
Published online 4 July 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3320

Fast parallel solver for the levelset equations on unstructured meshes

Zhisong Fu*,†, Sergey Yakovlev, Robert M. Kirby and Ross T. Whitaker

Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT, USA

SUMMARY

The levelset method is a numerical technique that tracks the evolution of curves and surfaces governed by a nonlinear partial differential equation (levelset equation). It has applications within various research areas such as physics, chemistry, fluid mechanics, computer vision, and microchip fabrication. Applying the levelset method entails solving a set of nonlinear partial differential equations. This paper presents a parallel algorithm for solving the levelset equations on unstructured 2D and 3D meshes. By taking into account constraints and capabilities of different computing architectures, the method is suitable for both the coarse-grained parallelism found on CPU-based systems and the fine-grained parallelism of modern massively single instruction, multiple data architectures such as graphics processors. In order to solve the levelset equations efficiently, we combine the narrowband scheme with a domain decomposition that is adapted for several different architectures. We also introduce a novel parallelism strategy, which we call *hybrid gathering*, which allows regular and lock-free computations of local differential operators. Finally, we provide the detailed description of the implementation and data structures for the proposed strategies, as well as performance data for both CPU and graphics processing unit implementations. Copyright © 2014 John Wiley & Sons, Ltd.

Received 12 June 2013; Revised 8 April 2014; Accepted 29 May 2014

KEY WORDS: levelset equations; unstructured meshes; parallel computing; graphics processing unit

1. INTRODUCTION

The levelset method uses a scalar function $\phi = \phi(\mathbf{x}(t), t)$ to implicitly represent a surface or a curve, $S = \{x(t) \mid \phi(\mathbf{x}(t)) = k\}$, hereafter referenced as a *surface*. The surface evolution is captured by numerically solving the associated nonlinear partial differential equation (levelset equation). The levelset method has a wide array of application areas, ranging from geometry, fluid mechanics, and computer vision to manufacturing processes [1] and, virtually, any problem that requires interface tracking. The method was originally proposed for regular grids by Osher and Sethian [2], and the early levelset implementations used finite difference approximations on fixed, logically rectilinear grids. Such techniques have the advantages of a high degree of accuracy and programming ease. However, in some situations, a triangulated domain with finite element-type approximations is desired. Barth and Sethian have cast the levelset method into the finite element framework and extended it to unstructured meshes in [3]. Since then, the levelset method has been widely used in applications that involve complex geometry and require the use of unstructured meshes for simulation. For example, in medical imaging, the levelset method on a brain surface is used for automatic sulcal delineation, which is important for investigating brain development and disease treatment [4]. In computer graphics, researchers have been using levelset methods for feature detection and mesh subdivision via geodesic curvature flow [5]. Yet another application of

*Correspondence to: Zhisong Fu, Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT, USA
†E-mail: zhisong@sci.utah.edu

the levelset method on unstructured domains is the simulation of solidification and crystal growth processes [6].

Many studies have been conducted to develop efficient levelset solvers. In [7], Adalsteinsson and Sethian propose a narrowband scheme to speed up the computation. This approach is based on the observation that one is typically only interested in a particular interface, in which case, the levelset equation only needs to be solved in a band around the interface. Whitaker proposes the sparse field method in [8], which employs the narrowband concept and maintains a narrowband containing only the wavefront nodes and their neighbors to further improve the performance. We should also point to several works that focus on the memory efficiency of the levelset method. Bridson proposes the sparse block grid method in [9] to dynamically allocate and free memory and achieves suboptimal storage complexity. Strain [10] proposes the octree levelset method that is also efficient in terms of storage. Houston *et al*. [11] apply the run-length encoding scheme to compress regions away from the narrowband to adjust their sign representation while storing the narrowband with full precision, which further improves the storage efficiency over the octree approach. A number of recent works [12–17] address parallelism strategies for solving the levelset equations on CPU-based and graphics processing unit (GPU)-based parallel systems. However, these works have been focused on regular grids or coarse-grained parallel systems, and the parallelism schemes proposed do not readily extend to unstructured meshes and fine-grained parallel systems.

Recently, there has been growing interest in floating-point accelerators. These accelerators are devices that perform arithmetic operations concurrently with or in place of the CPU. Two solutions have received special attention from the high-performance computing community: GPUs, originally developed to render graphics, which are now used for very demanding computational tasks, and the newly released Intel Xeon Phi, which employs very wide (512 bit) single instruction, multiple data (SIMD) vectors on the same X86 architecture as other Intel CPUs and promises high performance and little programming difficulty. Relative to CPUs, the faster growth curves of these accelerators in the speed and power efficiency have spawned a new area of development in computational technology. Now, many of the top supercomputers, such as Titan, the current top one [18], are equipped with such accelerators. Developing efficient code for these accelerators is a very important step toward fully utilizing the power of such supercomputers. In this paper, we present efficient parallel algorithms for solving the levelset equations on unstructured meshes on both CPU-based and GPU-based parallel processing systems.

The use of unstructured meshes makes the levelset method more flexible with respect to computational domains. However, solving the levelset equation on an unstructured mesh poses a number of challenges for efficient parallel computing. First, there is no natural partition of the domain for parallelism, and the use of a graph partitioner to decompose the mesh may result in uneven partition sizes, which in turn leads to a load balancing problem. Second, for regular meshes, the valence of the nodes is the same throughout the mesh, and hence, nodal parallelism (assigning each node to a thread) is typically employed. However, for unstructured meshes, the nodes have varying valence and local geometric structure, which leads to irregular data structures and unbalanced workload for nodal parallelism. Third, the inter-partition communication typically requires additional computation and separate data structures to find and store the partition boundary.

In this paper, we present a new parallelism strategy for solving the levelset equation on unstructured meshes that combines the narrowband scheme and a domain decomposition. We propose the narrowband fast iterative method (nbFIM) to compute the distance transform by solving the eikonal equation in a narrowband around the wavefront and the patched narrowband scheme (patchNB) to evolve the levelset. We use a unified domain partitioning for both distance transform and levelset evolution to ensure minimal setup time. For unstructured meshes, the update of the value on each node depends on values of its neighboring nodes, and a variable node valence may lead to load balancing issues. This is especially noticeable for GPUs and other streaming architectures, which employ SIMD-like architecture and prefer regular computations. To address this, we propose elemental parallelism instead of nodal parallelism to mitigate load balancing problems. However, the elemental parallelism approach may result in race conditions, because multiple elements may try to update the value of the same node simultaneously. Atomic operations are typically used to solve this problem. However, atomic operations are expensive, especially on GPUs, and can result

DOI: 10.1002/cpe

in significant numbers of threads blocking while waiting for access to variables. Therefore, we propose a new lock-free algorithm (and associated data structures) to enforce data compactness and locality for both shared memory CPU systems and GPUs. We call this approach *hybrid gathering*. Our algorithm converts the race condition problem to a sorting problem that is efficient on parallel systems, including GPUs [19]. Both the distance transform (part of maintaining the narrowband) and the levelset evolution benefit from this lock-free update scheme. In this paper, we describe the data structures and algorithms and present experimental results that demonstrate the efficiency of the proposed method on both shared memory CPU systems and GPUs.

The paper proceeds as follows. In Section 2, we introduce the levelset equation and the proposed method. In Section 3, we discuss implementation details and data structures. In Section 4, we discuss the performance of both CPU and GPU implementations of the proposed method, using several 2D and 3D examples to measure performance. In Section 5, we summarize the results and discuss future research directions related to this work.

2. MATHEMATICAL AND ALGORITHMIC DESCRIPTION

In general, the levelset equation solver with the narrowband scheme has two main building blocks: the distance transform recomputation (reinitialization) and the interface evolution according to the levelset equation (evolution). In this section, we give the mathematical and algorithmic description of the levelset method. We first introduce necessary notations and definitions and then describe the narrowband scheme and the associated reinitialization algorithm. Finally, we present the numerical scheme for the evolution step and the novel *hybrid gathering* parallelism scheme and lock-free update algorithm.

2.1. Notation and definitions

The levelset method relies on an implicit representation of a surface by a scalar function

$$\phi : \Omega(\boldsymbol{x}) \to \mathbb{R}, \tag{1}$$

where $\Omega \in \mathbb{R}^d, d \in \{2, 3\}$ is the *domain* of the surface model, which can be a 2D plane, a 3D volume, or a manifold. Thus, a surface S is

$$S = \{\boldsymbol{x} \mid \phi(\boldsymbol{x}) = k\}. \tag{2}$$

The choice of k depends on the problem at hand, and we call ϕ the *embedding*. The surface S is referred to as an *isosurface* of ϕ. Surfaces defined in this way partition Ω into *inside* and *outside* parts, and such surfaces are always closed provided that they do not intersect the boundary of the domain. The embedding ϕ is approximated on a tessellation of the domain. The levelset method uses a one-parameter family of embeddings, that is, $\phi(\boldsymbol{x}, t)$ changes over time t with $\boldsymbol{x}$ remaining on the k levelset of ϕ as it moves and k remaining constant. The behavior of ϕ is obtained by setting the total derivative of $\phi(\boldsymbol{x}(t), t) = k$ to zero. Thus,

$$\phi(\boldsymbol{x}(t), t) = k \quad \Longrightarrow \quad \frac{\partial \phi}{\partial t} + \nabla\phi \cdot \boldsymbol{v} = 0. \tag{3}$$

Let $\mathbf{F}$ denote the normal speed, $\mathbf{F} = \boldsymbol{v} \cdot \frac{\nabla\phi}{|\phi|}$. The levelset equation can then be written as

$$\frac{\partial \phi}{\partial t} + \mathbf{F}|\nabla\phi| = 0. \tag{4}$$

In general, $\mathbf{F}$ can be a more complicated function of $\boldsymbol{x}$ and $\nabla\boldsymbol{x}$, $\mathbf{F} = \mathbf{F}(\boldsymbol{x}, \nabla\phi, \nabla^2\phi, \ldots)$. In this paper, we consider the levelset equation with the normal speed function $\mathbf{F}$ of the following form:

$$\mathbf{F} = \boldsymbol{\alpha}(\boldsymbol{x}) \cdot \nabla\phi + \epsilon(\boldsymbol{x})|\nabla\phi| + \beta(\boldsymbol{x})\nabla \cdot \frac{\nabla\phi}{|\nabla\phi|}|\nabla\phi|, \tag{5}$$

DOI: 10.1002/cpe

where $\boldsymbol{\alpha}(\boldsymbol{x})$, $\epsilon(\boldsymbol{x})$, and $\beta(\boldsymbol{x})$ are user-defined coefficient functions. We call these three terms of $\mathbf{F}$ the *advection term*, the *eikonal term*, and the *curvature term*, respectively. This form of a levelset equation is used widely in many applications such as image processing and computer vision.

We approximate the domain Ω by a tessellation Ω_T, which consists of non-overlapping simplices that we call elements. Based upon this tessellation, we form a piecewise linear approximation of the solution by maintaining the values of the approximation on the set of vertices V of the tessellation and employing linear interpolation within each element in Ω_T. The total number of vertices in V is denoted $|V|$, and the total number of elements in Ω_T is denoted $|\Omega_T|$. We use v_i to denote the ith vertex in V. An *edge* is a line segment connecting two vertices (v_i, v_j) in $\mathbb{R}^d, d \in \{2, 3\}$ and is denoted by e_{ij}. The vector from vertex v_i to vertex v_j is denoted by $\mathbf{e}_{i,j} = \mathbf{x}_j - \mathbf{x}_i$.

In this paper, we consider both 2D and 3D cases, and Ω_T consists of triangles or tetrahedra respectively. A triangle, denoted T_{ijk}, is a set of three vertices v_i, v_j, v_k that are pairwise connected by an edge. Similarly, a tetrahedron is denoted T_{ijkl}. The set of vertices adjacent to vertex v_i is called *one-ring neighbors* of v_i and is denoted by $\mathcal{N}_i$, while the set of adjacent elements is called *one-ring elements* of v_i and is denoted by $\mathcal{A}_i$. We denote the discrete approximation of the solution ϕ at vertex v_i by ϕ_i. The area or volume of an element T is denoted meas(T).

2.2. Narrowband scheme and distance transform recomputation

Many applications require only a single surface model. In these cases, solving the levelset equation over the whole domain for every time-step is unnecessary and computationally inefficient. Fortunately, levelsets evolve independently (to within the error introduced by the discrete triangulation) and are not affected by the choice of embedding. Furthermore, the evolution of ϕ is important only in the vicinity of that levelset. Thus, one should perform calculations for the evolution of ϕ only in a neighborhood of the surface expressed by Equation 2. In the discrete setting, there is a particular subset of mesh nodes whose values define a particular levelset. Of course, as the surface moves, that subset of mesh nodes must change according to the new position of the surface.

In [7], Adalsteinsson and Sethian propose a narrowband scheme that follows this line of reasoning. The narrowband scheme constructs an embedding of the evolving curve or surface via a signed distance transform. The distance transform is truncated, that is, computed over a finite number of nodes that lie no further than a specified distance from the levelset. This truncation defines the narrowband, and the remaining points are set to constant values to indicate that they lie outside the narrowband. The evolution of the surface is computed by calculating the embedding only within the narrowband. When the evolving levelset approaches the edge of the narrowband, the new distance transform, and the new embedding are calculated, and the process is repeated. This algorithm relies on the fact that the embedding is not a critical aspect of the evolution of the levelset. That is, the embedding can be transformed or recomputed at any point in time, as long as such transformation does not change the position of the kth levelset, and the evolution will be unaffected by this change in the embedding. Following the strategy in [8], most implementations keep a list of nodes in the narrowband. However, this approach is not efficient for GPUs and unstructured meshes because the nodes in the narrowband will have arbitrary order and the memory accesses are effectively random. We propose the *patched narrowband* (patchNB) scheme to enforce memory locality and improve performance on GPUs. This scheme keeps a list of patches contained by the narrowband instead of nodes, and each patch is assigned to a GPU streaming multiprocessor with values of nodes in each patch being updated in parallel by GPU cores. In this way, the data (geometry information, values, intermediate data) associated with each patch can be stored in the fast shared memory, and global memory accesses are coalesced and reduced. We describe the scheme in more detail in Section 3.

This narrowband scheme requires the computation of the distance transform (reinitialization). We propose a modified version of the patched fast iterative method [20], which we call nbFIM, to compute the distance transform by solving the eikonal equation with the value of speed function set to one. The nbFIM restricts the computational domain to the narrowband around the levelset, which significantly reduces the computational burden. Also, we propose new algorithms and data structures to further improve the performance.

DOI: 10.1002/cpe

Specifically, the nbFIM employs a domain decomposition scheme that partitions the computational domain into patches and iteratively updates the node values of the patches near the levelset until all patches are either converged (not changing anymore) or far away from the levelset. The algorithm maintains an active list that stores the patches requiring an update. The active list initially contains the patches that intersect with the levelset. It is then updated by removing convergent patches and adding their neighboring patches if they are within a certain distance from the levelset. The distance between a patch and a levelset is defined as the minimal value of all the node values in this patch. In this way, the patches that are far from the levelset are not updated at all, which reduces computation. Also, it is guaranteed that all the nodes with values smaller than narrowband width are in the new narrowband list. The details of the implementation will be described in Section 3.

The node value update process calculates the new value of a node by computing potential values from the one-ring elements and take the minimal of the potential values as the new value. We call the computation of the potential value of a node the *local_solver*, which requires geometric information of one of the one-ring elements and the values of other nodes in the element.

2.3. Levelset evolution and PatchNB

The numerical scheme we use to discretize Equation 4 in space is based on [3]. We adopt the positive coefficient scheme for the first-order terms of the levelset equation (the advection and eikonal terms):

$$\mathbb{H}_j(\nabla\phi) = \frac{\sum_{l=1}^{|\Omega_T|} \tilde{\alpha}_j^l \mathbb{H}(\nabla\phi)}{\sum_{l=1}^{|\Omega_T|} \tilde{\alpha}_j^l \text{meas}(T_l)}, \tag{6}$$

where $\mathbb{H}$ denotes wither the advection or the eikonal term, which are first-order homogeneous functions of $\nabla\phi$, and $\tilde{\alpha}_j^l$ are non-negative constant coefficients that are defined as follows:

$$\tilde{\alpha}_j^l = \frac{\max\left(0, \alpha_j^l\right)}{\sum_{k=1}^{d+1} \max\left(0, \alpha_k^l\right)}. \tag{7}$$

The coefficients α_j^l are defined as

$$Q_j = \nabla\mathbb{H} \cdot \nabla N \text{meas}(T), \tag{8}$$

$$Q_j^- = \min(0, Q_j), \tag{9}$$

$$Q_j^+ = \max(0, Q_j), \tag{10}$$

$$\delta_j = Q_j^+ \left(\sum_{l=1}^{d+1} Q_l^+\right)^{-1} \sum_{i=1}^{d+1} Q_i^-(\phi_i - \phi_j), \tag{11}$$

$$\alpha_j^l = \frac{\delta_j}{\mathbb{H}_{T_l}}, \tag{12}$$

where $\mathbb{H}_{T_l} = \int_{T_l} \mathbb{H} d\boldsymbol{x}$, and N_i is the linear basis function satisfying $N_i(\boldsymbol{x}_j) = \delta_{ij}$.

We adopt the finite element method for the curvature term, approximating it at each node v_i as

$$(K|\nabla\phi|)_i = \frac{\sum_{j\in\mathcal{N}_i} W_j^i(\phi_j - \phi_i)}{\sum_{T\in\mathcal{A}_i} \text{meas}(T)} \left| \frac{\sum_{T\in\mathcal{A}_i} \text{meas}(T) \sum_{j=1}^{d+1} \nabla N_j \phi_j}{\sum_{T\in\mathcal{A}_i} \text{meas}(T)} \right|, \tag{13}$$

where $W_j^i = \sum_{\{T|e_{ij}\in T\}} \frac{\nabla N_i \cdot \nabla N_j \text{meas}(T)}{|\nabla\phi_j|}$.

Equations 6–13 show that the computation of the new value ϕ_i at node v_i requires values from its one-ring neighbors and geometric data from the one-ring elements. Algorithm 1 is a typical serial algorithm, and **AH1**, **AH2**, **AV1**, **AV2**, **V**, **PV**, and **CV** are temporary arrays that store the

DOI: 10.1002/cpe

intermediate results. Similar to the reinitialization, we use elemental parallelism for better load balancing and hybrid gathering scheme to avoid race conditions.

Algorithm 1 Evolution(ϕ, B) (ϕ: values of the nodes, B: narrowband)

for all $p \in B$ **do**
  for all $T_{ijk} \in p$ **do**
    $T = T_{ijk}$
    compute meas(T)
    compute $\nabla N_i, \nabla N_j, \nabla N_k$
    $\nabla\phi = \nabla N_i * \phi_i + \nabla N_j * \phi_j + \nabla N_k * \phi_k$
    compute Q for the advection term: Q_i^1, Q_j^1, Q_k^1
    compute Q for the eikonal term: Q_i^2, Q_j^2, Q_k^2
    $(\mathbb{H}^1)_T = Q_i^1 * \phi_i + Q_j^1 * \phi_j + Q_k^1 * \phi_k$
    $(\mathbb{H}^2)_T = Q_i^2 * \phi_i + Q_j^2 * \phi_j + Q_k^2 * \phi_k$
    compute Q^+ and Q^-
    compute $\tilde{\alpha}_i^1, \tilde{\alpha}_j^1, \tilde{\alpha}_k^1, \tilde{\alpha}_i^2, \tilde{\alpha}_j^2, \tilde{\alpha}_k^2$
    $\mathbf{AH1}[i] \mathrel{+}= \tilde{\alpha}_i^1 * (\mathbb{H}^1)_T$, $\mathbf{AH}[j] \mathrel{+}= \tilde{\alpha}_j^1 * (\mathbb{H}^1)_T$, $\mathbf{AH}[k] \mathrel{+}= \tilde{\alpha}_k^1 * (\mathbb{H}^1)_T$
    $\mathbf{AH2}[i] \mathrel{+}= \tilde{\alpha}_i^2 * (\mathbb{H}^2)_T$, $\mathbf{AH}[j] \mathrel{+}= \tilde{\alpha}_j^2 * (\mathbb{H}^2)_T$, $\mathbf{AH}[k] \mathrel{+}= \tilde{\alpha}_k^2 * (\mathbb{H}^2)_T$
    $\mathbf{AV1}[i] \mathrel{+}= \tilde{\alpha}_i^1 * \text{meas}(T)$, $\mathbf{AV}[j] \mathrel{+}= \tilde{\alpha}_j^1 * \text{meas}(T)$, $\mathbf{AV}[k] \mathrel{+}= \tilde{\alpha}_k^1 * \text{meas}(T)$
    $\mathbf{AV2}[i] \mathrel{+}= \tilde{\alpha}_i^2 * \text{meas}(T)$, $\mathbf{AV}[j] \mathrel{+}= \tilde{\alpha}_j^2 * \text{meas}(T)$, $\mathbf{AV}[k] \mathrel{+}= \tilde{\alpha}_k^2 * \text{meas}(T)$
    $\mathbf{V}[i] \mathrel{+}= \text{meas}(T)$, $\mathbf{V}[j] \mathrel{+}= \text{meas}(T)$, $\mathbf{V}[k] \mathrel{+}= \text{meas}(T)$
    $\mathbf{PV}[i] \mathrel{+}= \nabla\phi * \text{meas}(T)$, $\mathbf{PV}[j] \mathrel{+}= \nabla\phi * \text{meas}(T)$, $\mathbf{PV}[k] \mathrel{+}= \nabla\phi * \text{meas}(T)$
    $\mathbf{CV}[i] \mathrel{+}= \frac{\nabla N_i \cdot \nabla N_i * \text{meas}(T)}{|\nabla\phi|} + \frac{\nabla N_i \cdot \nabla N_j * \text{meas}(T)}{|\nabla\phi|} + \frac{\nabla N_i \cdot \nabla N_k * \text{meas}(T)}{|\nabla\phi|}$
    $\mathbf{CV}[j] \mathrel{+}= \frac{\nabla N_j \cdot \nabla N_j * \text{meas}(T)}{|\nabla\phi|} + \frac{\nabla N_j \cdot \nabla N_i * \text{meas}(T)}{|\nabla\phi|} + \frac{\nabla N_j \cdot \nabla N_k * \text{meas}(T)}{|\nabla\phi|}$
    $\mathbf{CV}[k] \mathrel{+}= \frac{\nabla N_k \cdot \nabla N_k * \text{meas}(T)}{|\nabla\phi|} + \frac{\nabla N_k \cdot \nabla N_i * \text{meas}(T)}{|\nabla\phi|} + \frac{\nabla N_k \cdot \nabla N_j * \text{meas}(T)}{|\nabla\phi|}$
  end for
  for all $v_i \in B$ **do**
    $\phi_i \mathrel{-}= \alpha \frac{\mathbf{AH1}[i]}{\mathbf{AV1}[i]} + \epsilon \frac{\mathbf{AH2}[i]}{\mathbf{AV2}[i]} + \beta \frac{|\mathbf{CV}[i]|\mathbf{PV}[i]}{\mathbf{V}[i]^2}$
  end for
end for

2.4. Hybrid gathering parallelism and lock-free update

In both of the reinitialization and the evolution steps, we need to update the values of the nodes in the mesh, and these updates can be performed independently. The natural way to parallelize the computation is to assign each nodal update to a thread. We call this approach *nodal parallelism*, and it can be represented as a sparse matrix-vector operation as shown in Figure 1 (left). The operator $\bigotimes$ denotes a generic operation defined on the degrees of freedom corresponding to $\star$'s. The advantage of this scheme is that it naturally avoids race conditions, because each nodal computation has an associated thread. However, it can introduce unbalanced load when the nodes have widely varying valence. These irregular computations and data structures are not efficient on GPUs, because of the nature of the SIMD streaming architecture. An alternative parallelism scheme is to distribute computations among threads according to the elements. We call such approach *elemental parallelism*; it is more suitable for GPUs, because it gives regular local operators and corresponding data structures. Figure 1 (right) depicts the matrix representation of this approach: the matrix has a block structure in terms of local operators corresponding to the elements. The matrix blocks can overlap each other, and the vector of degrees of freedom is segmented but has overlaps. Each block matrix-vector operation represents a set of local computations that are performed by a thread. This parallelism scheme may result in race conditions as multiple threads may be updating the same degree of freedom because of the overlapping. The conventional solution to this problems is to use atomic operations.

DOI: 10.1002/cpe

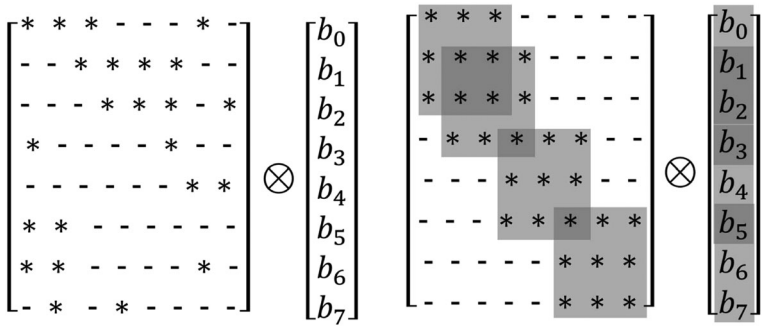

Figure 1. Matrix representations of the parallelism schemes. On the left, we present the nodal parallelism scheme. The ⋆'s denote non-zeros values or some operators. On the right, we present the elemental parallelism scheme. The matrix is blocked, and the blocks can be overlapping each other.

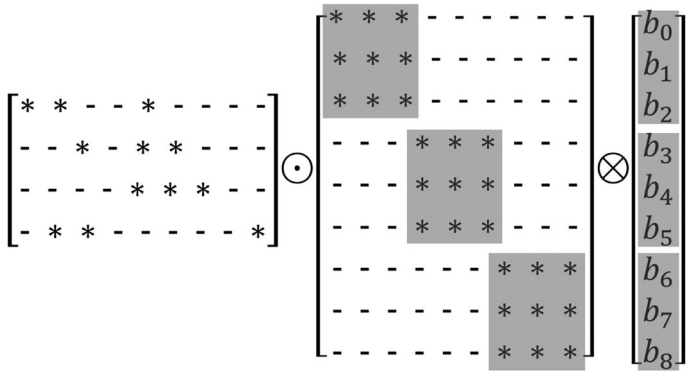

Figure 2. Matrix representations of the elemental gathering scheme.

However, this is not suitable for GPUs as the atomic operations on GPUs are expensive, especially for double-precision floating-point operations, which are widely used in scientific computing.

We have developed a novel parallelism scheme, which we call hybrid gathering, to combine the advantages of both the nodal and elemental parallelism schemes. In the hybrid gathering parallelism scheme, the computation is decomposed into two stages (two matrix-vector operations): (1) performs local operations on the associated element and stores intermediate result and (2) fetches and assembles intermediate result data according to the gathering matrix Λ. The symbols $\otimes$ and $\odot$ in Figure 2 represent the operations in these two stages, respectively. In the first stage, matrix blocks and vector segments are not overlapping, and the matrix-block-vector-segment operations ($\otimes$) can be assigned to different threads, thus avoiding the race condition. This stage decomposes the computations according to the elements with regular local operators, after which, each thread fetches intermediate result data according to the gathering matrix to assemble ($\odot$) the value for the degrees of freedom. In practice, the two stages are implemented in a single kernel function, and fast GPU cache (shared memory or registers) is used to store the intermediate data, which makes the gathering stage very efficient.

The computation of the gathering matrix is a key part of the hybrid gathering parallelism scheme. The degrees of freedom are associated with the nodes, and the computations are performed on elements. Therefore, the gathering matrix should represent a topological mapping from elements to nodes, and this mapping describes the data dependency for each degree of freedom. In practice, the mapping from elements to nodes is typically given as an element list, denoted $\mathbb{E}$, which consists of node indices. The array element indices of this list correspond to memory locations of the data that the threads require. We create a sequence list $\mathbb{S}$ that records the memory locations of $\mathbb{E}$. Then, we sort $\mathbb{E}$ and permute $\mathbb{S}$ according to the sorting. Now, in the sorted list $\mathbb{E}'$, the node indices are grouped, and the permuted sequence list, denoted $\mathbb{S}'$, stores the data memory location in original

DOI: 10.1002/cpe

element list $\mathbb{E}$. The $\mathbb{E}'$ and $\mathbb{S}'$ together indicate the locations of the $\star$'s in the gathering matrix and form the coordinate list (COO) sparse matrix representation [21] of the gathering matrix. In this way, we convert the race condition problem to a sorting problem. Here, list $\mathbb{E}$ has fixed length keys, which allows it to be sorted very efficiently on GPUs with radix sorting [19]. Essentially, sorting allows us to take full advantage of the GPU computing power and avoid the weakness of the architecture in the form of addressing race conditions.

3. IMPLEMENTATION

In this section, we describe the implementation details of our method to solve the levelset equations on both shared memory CPU-based and GPU-based parallel systems. The pipeline consists of two stages: the setup stage and the time-stepping stage. The setup stage includes the partitioning of the mesh into patches, preparation of the geometric data for the following computation and generation of Λ. We choose the METIS software package [22] to perform the partitioning. METIS partitions the mesh into non-overlapping node patches and outputs a list showing the partition index of each node. The data-preparation step permutes the vertex coordinate list and rearranges the element list according to the partitioning. The time-stepping stage iteratively updates the node values until the desired number of time-steps is reached. In each iteration, a reinitialization and multiple evolution steps are performed. We provide implementation details and data structures for the setup stage, the reinitialization step, and the evolution step, respectively, in the following subsections.

3.1. Setup

During the setup stage, the mesh is partitioned into patches, each of which consists of a set of nodes and a set of elements according to METIS output. The sets of nodes are mutually exclusive, and the element sets are one-layer overlapping: the boundary elements are duplicated. The vertex coordinate list and the element list are then permuted according to the partitioning so that the vertex coordinates and the element vertex indices are grouped together, and hence, the global memory access is coalesced.

As described in Section 2, we use the hybrid gathering scheme to decompose the computation. Therefore, we must generate the gathering matrix Λ during the setup stage. Here, we describe a simple example, with a triangular mesh, to demonstrate matrix Λ generation. First, consider the simple mesh displayed in Figure 3. This mesh consists of two triangles, e_0 and e_1, which give us four degrees of freedom to solve for ϕ_0 through ϕ_3. During the solution process, the thread corresponding to element e_0 will be updating the values of ϕ_0, ϕ_1 and ϕ_2, while the thread corresponding to element e_1 will be updating the values of ϕ_0, ϕ_3, and ϕ_1. The corresponding data flow is shown in Figure 4. We store the intermediate data in a separate array that has one-to-one correspondence to the element list. Therefore, the element list indices are the same as the memory locations, from which the degrees of freedom require data. For our example, any computation involving ϕ_0 requires data from the element list memory locations 0 and 3. If we create an auxiliary sequence list $\{0, 1, 2, 3, 4, 5\}$, sort

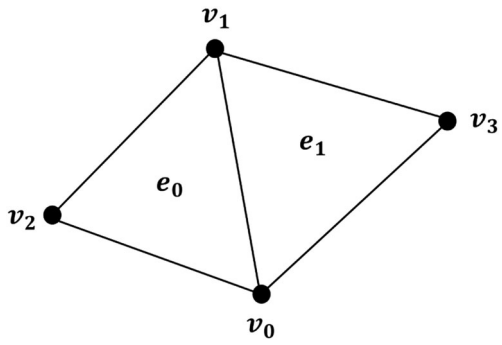

Figure 3. Mesh with two elements: e_0 and e_1.

DOI: 10.1002/cpe

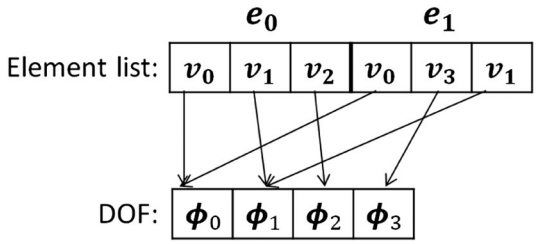

Figure 4. Data flow for the simple mesh example.

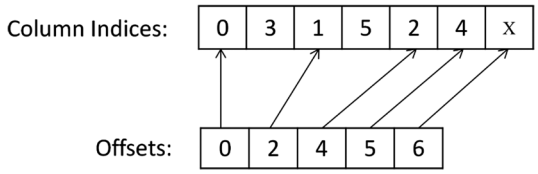

Figure 5. The CSR representation of gathering matrix for the two-triangle example. The box containing 'X' denotes a memory location outside the bounds of the column indices array.

the element list, and permute the sequence list according to the sort, we obtain two new lists: sorted element list $\{0, 0, 1, 1, 2, 3\}$ and permuted auxiliary list $\{0, 3, 1, 5, 2, 4\}$. These two lists now contain the row indices and column indices of the $\star$ entries of matrix Λ. In practice, we convert this to compressed sparse row (CSR) matrix format [21] for storage by a reduction operation and a prefix sum operation on the sorted element list. The $\star$ entries in the matrix actually represent certain operators: minimum and summation in the reinitialization and evolution stages, correspondingly. Hence, we do not have the value array in a typical CSR format. The final CSR representation of the gathering matrix for our two-triangle example is shown in Figure 5. The *Column Indices* array stores the elements of the permuted auxiliary list, which are the column indices of the $\star$ entries in Λ, and the *Offsets* array stores the starting indices of the $\star$ entries for each row.

3.2. Reinitialization

In the reinitialization step, we use Algorithm 2 (nbFIM), to compute the distance transform of the levelset. In this algorithm, the node value update step comprises the bulk of the work. This step updates the values of the nodes in the active patches multiple times (we call these *inner updates*) using the hybrid gathering scheme and records the convergence status of each active node. During each of the inner updates, each thread computes all the values on the nodes of the corresponding element and stores values in an intermediate array in the fast GPU shared memory. Then, we apply the gathering matrix to the intermediate array to compute the new value of each node. In this operation, each row of the gathering matrix corresponds to a node in the mesh, and column indices of the $\star$ entries in each row represent the location of required data in the intermediate array. We assign each row to a thread, which fetches the data from the intermediate array and calculates the minimum that is taken as the new value of the node. In practice, the elemental update and the gathering operation are performed in one kernel so that the intermediate values do not need to be written back to or read from global memory, which would be expensive. As mentioned earlier, the node coordinates and the element list are grouped according to the partitioning and stored in an interleaved linear array so that memory accesses are coalesced. Before the update computation, each thread needs to fetch node coordinates and old values for the corresponding element, and the memory access is virtually random. We use the fast shared memory to hold this data temporarily, and then each thread read data from the shared memory instead of directly from the global memory.

We use an improved *cell-assembly* data structure in the nbFIM, originally proposed in [20]. The new cell-assembly data structure includes five arrays, labeled **COORD**, **VAL**, **ELE**, **OFFSETS**, and **COL**. The **COORD** and **VAL** arrays store per-node coordinates and per-node

DOI: 10.1002/cpe

value, respectively. **ELE** stores the per-element node indices. **OFFSETS** and **COL** form the CSR sparse matrix representation of the gathering matrix, except that we do not require a value array as in general CSR representation.

In summary, assuming that every patch has N nodes and M elements (normally $M > N$), the reinitialization kernel function on the GPU (or SIMD parallelism) proceeds as follows:

(1) If thread index $i < N$, load the coordinates and value of node i into shared memory array **SHARE**.
(2) If thread index $i < M$, load the node indices for element i from **ELE** into registers. Fetch the node coordinates and values from **SHARE** to registers.
(3) If thread index $i < M$, write node values of element i to shared memory **SHARE**.
(4) If thread index $i < M$, call *local_solver* routine to compute the potential values of each node in element i and store these values in **SHARE**.
(5) If thread index $i < N$, load the column indices for the ith row of the gathering matrix, **COL**[**OFFSETS**[i]] through **COL**[**OFFSETS**[$i + 1$]]. Then fetch data from **SHARE**, compute the minimal value, and broadcast the minimal value to **SHARE** according to the column indices.
(6) If thread index $i < N$, if the minimal value is the same as the old value (within a tolerance), node i is labeled convergent.
(7) Repeat steps 4 through 6 multiple times.
(8) If thread index $i < N$, write the minimal value back to global memory **VAL**.

For CPU-based shared memory parallel system, the implementation is generally similar, but we make several modifications from the GPU implementation to suit the CPU architecture. First, we maintain an active node-group list instead of an active patch list. Each node-group in the list stores only the active nodes in its corresponding patch instead of all nodes in the patch. The patched update strategy is suitable for GPU because it provides the fine-grained parallelism desired by the GPU architecture, but it also leads to extra computation [20]. Second, in the computation that updates the active node values, we assign each segment to a thread that updates all the active node values in the segment. In addition, we use the nodal parallelism to avoid any race conditions. Each thread computes the potential values of an active node from the one-ring elements and then calculates the minimal value among the potential values. Specifically, the value update function in the reinitialization stage proceeds in the following steps for each thread t (here, P denotes the number of patches):

(1) Load the coordinates and value ϕ_a of an active node a of the tth patch into registers.
(2) Pick one of the one-ring elements of a and load the coordinates and values into registers.
(3) Call the *local_solver* to compute a potential new value ϕ_{tmp} of node a and perform $\phi_a = \min(\phi_a, \phi_{tmp})$.
(4) Repeat steps 2 and 3 until all one-ring elements of a are processed and write the final ϕ_a back to memory.
(5) Repeat steps 1–4 until all active nodes in patch t are processed.

3.3. Evolution

The evolution step updates the values of the nodes in the narrowband according to the equations presented in Section 2. As shown in Algorithm 1, we compute the approximation of the three terms of **F** and then update the node values. Similar to the reinitialization step, we need to deal with mixed types of parallelism: nodal parallelism and elemental parallelism as elemental computations are suitable for GPUs, but the degrees of freedom we want to solve for are defined on the nodes. The hybrid gathering scheme is also used to solve this problem. The update of a node value depends on multiple elemental computations corresponding to speed function **F**, and the computations for the three terms of **F** all require the same geometric information and node values. Therefore, we want to perform all the computations in one kernel function to avoid repeated global memory access.

The hybrid gathering scheme is based on the assumption that the gathering is performed on fast shared memory, and using a single kernel means that we need to store all the elemental intermediate

DOI: 10.1002/cpe

Algorithm 2 nbFIM(A, L, w, Φ, B, S) (A: set of patches, L: active patch list, Φ: values of the nodes, w: narrowband width, B: narrowband, S: set of source vertices)

```
comment: initialize the active list L
for all a ∈ A do
  for all v ∈ a do
    if any v ∈ S then
      add a to L
    end if
  end for
end for
comment: iterate until L is empty
while L is not empty do
  comment: initialize the active list L
  for all a ∈ L do
    update the values of nodes in a with local_solver
  end for
  for all a ∈ L do
    check if a is converged with reduction operation
  end for
  for all a ∈ L do
    compute the minimum value φ^a_min of the all the nodes in a
  end for
  for all a ∈ L do
    if a is converged && φ^a_min < w then
      add neighboring patches of a into a temporary list L_temp
    end if
  end for
  clear active list L
  for all a ∈ L_temp do
    perform 1 internal iteration for a
  end for
  for all a ∈ L_temp do
    check if a is converged with reduction operation
  end for
  for all a ∈ L_temp do
    if a is converged then
      add a into active list L
    end if
  end for
end while
for all a ∈ A do
  if a is converged then
    add a into narrowband B
  end if
end for
```

results (**AH1**, **AH2**, **AV1**, **AV2**, **V**, **PV**, and **CV** arrays in Algorithm 1) in shared memory, which usually is not large enough to accommodate all this data. We solve this problem by reusing the shared memory space and carefully arranging the memory load order so that the data memory footprint is small enough to fit in the GPU shared memory. Specifically, we store the elemental intermediate results in **SAH1**, **SAH2**, **SAV1**, **SAV2**, **SV**, **PV**, and **SCV** arrays for each patch, which are all in shared memory, and then assemble the intermediate results according to the gathering matrix and

DOI: 10.1002/cpe

store them in fast registers. The evolution kernel function proceeds as follows (assuming every patch has N nodes and M elements):

(1) If thread index $t < N$, load the coordinates and value of node t into shared memory array **SHARE**.
(2) If thread index $t < N$, load the column indices for the tth row of the gathering matrix, **COL**[**OFFSETS**[t]] through **COL**[**OFFSETS**[$t + 1$]] into registers.
(3) If thread index $t < M$, load the node indices for element t from **ELE** into registers. Fetch the node coordinates and values from **SHARE** to registers.
(4) If thread index $t < M$, perform elemental computation of triangle T_{ijk} for $(\mathbb{H}^1)_T$, $(\mathbb{H}^2)_T$ and $\tilde{\alpha}_i^1, \tilde{\alpha}_j^1, \tilde{\alpha}_k^1, \tilde{\alpha}_i^2, \tilde{\alpha}_j^2, \tilde{\alpha}_k^2$.
(5) If thread index $t < M$, compute SAH1[t*M+0] $= \tilde{\alpha}_i^1 * (\mathbb{H}^1)_T$, SAH1[t*M+1] $= \tilde{\alpha}_j^1 * (\mathbb{H}^1)_T$, SAH1[t*M+2] $= \tilde{\alpha}_k^1 * (\mathbb{H}^1)_T$.
(6) If thread index $t < N$, fetch data from **SAH1** according to the column indices for the tth row of the gathering matrix, compute the summation, and store the result in registers.
(7) If thread index $t < M$, compute **SAH2**[t*M+0] $= \tilde{\alpha}_i^2 * (\mathbb{H}^2)_T$, **SAH2**[t*M+1] $= \tilde{\alpha}_j^2 * (\mathbb{H}^2)_T$, **SAH2**[t*M+2] $= \tilde{\alpha}_k^2 * (\mathbb{H}^2)_T$. Note here **SAH2** overlap **SAH1** in the shared memory space and the values of **SAH2** are completely rewritten. In this way, the shared memory footprint is not increased as the size of **SAH2** is the same as **SAH1**.
(8) If thread index $t < N$, fetch data from **SAH1** according to the column indices for the tth row of the gathering matrix, compute the summation, and store the result in registers.
(9) Repeat steps 7 through 8 for **SAV1**, **SAV2**, **SV**, **SPV**, and **SCV** arrays.
(10) If thread index $t < N$, compute the new value of node t.

Similarly, for the CPU implementation of the evolution step, we make certain modifications of the GPU implementation to suit the CPU architecture. We keep a list of elements that are inside the narrowband and perform the computation only on these elements. This is different from the GPU implementation, which updates all elements in a patch as long as any element in the patch is inside the narrowband. We assign the element computations in each patch to a thread instead of each element to a thread to provide a coarse-grained parallelism for CPU. Also, we find that for CPU, atomic operations are sufficiently efficient, so the hybrid gathering scheme is not used.

3.4. Adaptive time-step computation

After each reinitialization step, we perform n update steps for the levelset evolution. In this process, we need to make sure that the evolving levelset does not cross the boundary of the narrowband. According to the Courant-Friedrichs-Lewy condition, the levelset evolution distance of each time-step $\Delta \boldsymbol{x} \leq 2 \max_{i<M}(r_i)$, where r_i denotes the inscribed circle or sphere of the ith element and M is the number of elements in the narrowband. Denoting the narrowband width as w, we make a conservative estimate of the number of steps $n = \frac{w}{4 \max_{i<M}(r_i)}$, so that the levelset evolves at most half of w. Because the narrowband is changing, $\max(r_i)$ is also changing. Hence, we compute the $\max(r_i)$ at the beginning of each reinitialization step: r_i are pre-computed and stored in an array, and the $\max(r_i)$ is computed with a reduction operation. In the evolution step, the time-step Δt is dictated by the three terms of **F** in Equation 4. We define the time-step as

$$\Delta t = \min_{i<M} \left(\frac{2r_i}{d(|\boldsymbol{\alpha}| + |\epsilon|)}, \frac{2r_i^2}{d\,|\kappa|} \right), \tag{14}$$

where d is the dimensionality of the mesh.

4. RESULTS AND DISCUSSION

In this section, we present numerical experiments to demonstrate the performance of the proposed algorithms. We use a collection of 2D and 3D unstructured meshes of variable size and complexity to illustrate the performance of both CPU and GPU implementations. The performance data and

DOI: 10.1002/cpe

implementation related details are provided in the following order: (1) CPU implementation discussion, (2) GPU implementation discussion, and concluded by (3) the comparison of the two. For consistency of evaluation, double precision is used in all algorithms and for all the experiments presented in the following text.

The meshes used for the numerical experiments are

RegSquare: $\Omega = [0, 512]^2$, 524,288 triangles, regular triangulation with maximum node valence of six;
IrregSquare: $\Omega = [0, 512]^2$, 1,181,697 vertices and 2,359,296 triangles, irregular triangulation with maximum node valence of 14;
Sphere: $\Omega =$ sphere surface, 1,023,260 vertices and 2,046,488 triangles, irregular triangulation with maximum node valence of 11;
Brain: $\Omega =$ left hemisphere of human brain cortex surface, 631,187 vertices and 1,262,374 triangles (Figure 6), irregular triangulation with maximum node valence of 19;
RegCube: $\Omega = [0, 63]^3$, 1,500,282 tetrahedra, regular tetrahedralization with maximum node valence of 24; and
IrregCube: $\Omega = [0, 63]^3$, 197,561 vertices and 1,122,304 tetrahedra, irregular tetrahedralization with maximum node valence of 54.

These meshes include 2D planar meshes, manifold (surface) meshes, and 3D meshes. They exhibit different geometrical complexity, mesh quality, and maximum nodal valence, which allows us to assess the effect that mesh properties have on the algorithm performance.

The numerical simulation setup is as follows: we solve the levelset equation

$$\begin{cases} \dfrac{\partial \phi}{\partial t} + \boldsymbol{\alpha}(\boldsymbol{x}) \cdot \nabla \phi + \epsilon(\boldsymbol{x}) |\nabla \phi| + \beta(\boldsymbol{x}) \nabla \cdot \dfrac{\nabla \phi}{|\nabla \phi|} |\nabla \phi| = 0, \\ \phi(\boldsymbol{x}, t = 0) = g(\boldsymbol{x}), \end{cases} \tag{15}$$

where $\boldsymbol{\alpha}(\boldsymbol{x})$ is a user-defined vector function and $\epsilon(\boldsymbol{x})$ and $\beta(\boldsymbol{x})$ are user-defined scalar functions. The initial condition $g(\boldsymbol{x})$ defines the values in the domain at time $t = 0$. The choice of these constant coefficients makes little difference to the computation steps. In the following numerical experiments, we set the constant coefficients α, ϵ, and β to be $(1, 0, 0)$, 0.0, and 0.0, respectively, for non-manifold meshes (RegSquare, IrregSquare, RegCube, IrregCube). Figure 7 shows the result for the RegSquare mesh with these coefficients. The color map indicates the signed distance from the interface. Because the advection term is not well defined on the manifolds, we set the coefficients to be $(0, 0, 0)$, 0.0, and 1.0 for manifold meshes (Sphere and Brain). Solving the levelset equation with these coefficients gives the geodesic curvature flow, which is widely used in many image processing and computer vision applications [4, 5]. Figure 8 shows the geodesic curvature flow on a human

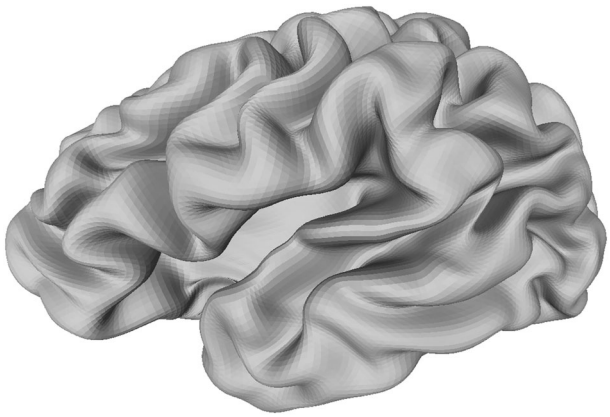

Figure 6. Left hemisphere of human brain cortex surface mesh.

DOI: 10.1002/cpe

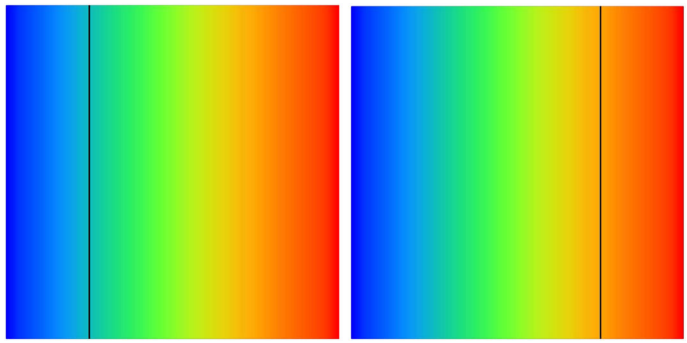

Figure 7. The interface on the RegSquare mesh. The left image shows the initial interface, and the right image shows the interface after evolution.

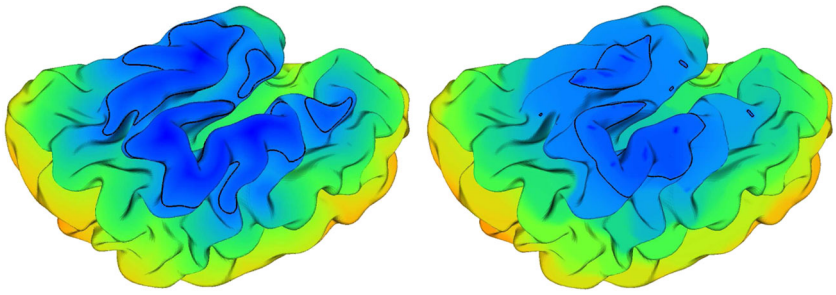

Figure 8. The interface on the brain mesh. The left image shows the initial interface, and the right image shows the interface evolution.

brain cortex. The left image demonstrates the initial interface, and the right image shows the interface after evolution. We use the numerical scheme presented in [5] to discretize the curvature term on manifolds. Computationally, this scheme is almost the same as the numerical scheme we use for 2D and 3D meshes.

4.1. CPU implementation results and performance analysis

We conduct systematic experiments on a CPU-based parallel system to show the effectiveness and characteristics of our proposed method. We test our CPU implementation on a workstation equipped with two Intel Xeon E5-2640 CPU (12 cores in total) running at 2.5 GHz with turbo boost and hyperthreading enabled and 32 GB DDR3 memory shared by the CPUs. The computer is running openSuse 11.4, and the code is compiled with gcc 4.5 using optimization option -O3 (We also used icc as the compiler, and the result were similar). We run our multithreaded CPU implementation as described in Section 3 on a workstation with 12 CPU cores to assess the effect of the patched update strategy on the scalability of the levelset equation solver. We compare the result with a naive parallel implementation without patched update schemes (nbFIM or patchNB). In this naive implementation, the nodal computations in the reinitialization and the elemental computations in the evolution are distributed among threads and performed in parallel without being grouped according to patches.

The plots in Figure 9 shows the strong scaling comparison between the multithreaded CPU implementations with the proposed schemes (Patched) and the naive parallel implementation (Nonpatched). We perform this test with two 2D triangular meshes (RegSquare and IrregSquare) and two 3D tetrahedral meshes (RegCube and IrregCube). As shown from the plots, our proposed multithreaded implementation scales up to 12 cores and achieves up to 7× speedup with 12 cores against the serial implementation (with 1 core). By contrast, the nonpatched implementation scales poorly when running with more than four cores, and it does not scale when running with more than eight cores. We get up to 7× speedup with a 12-core system because the reinitialization step uses the

DOI: 10.1002/cpe

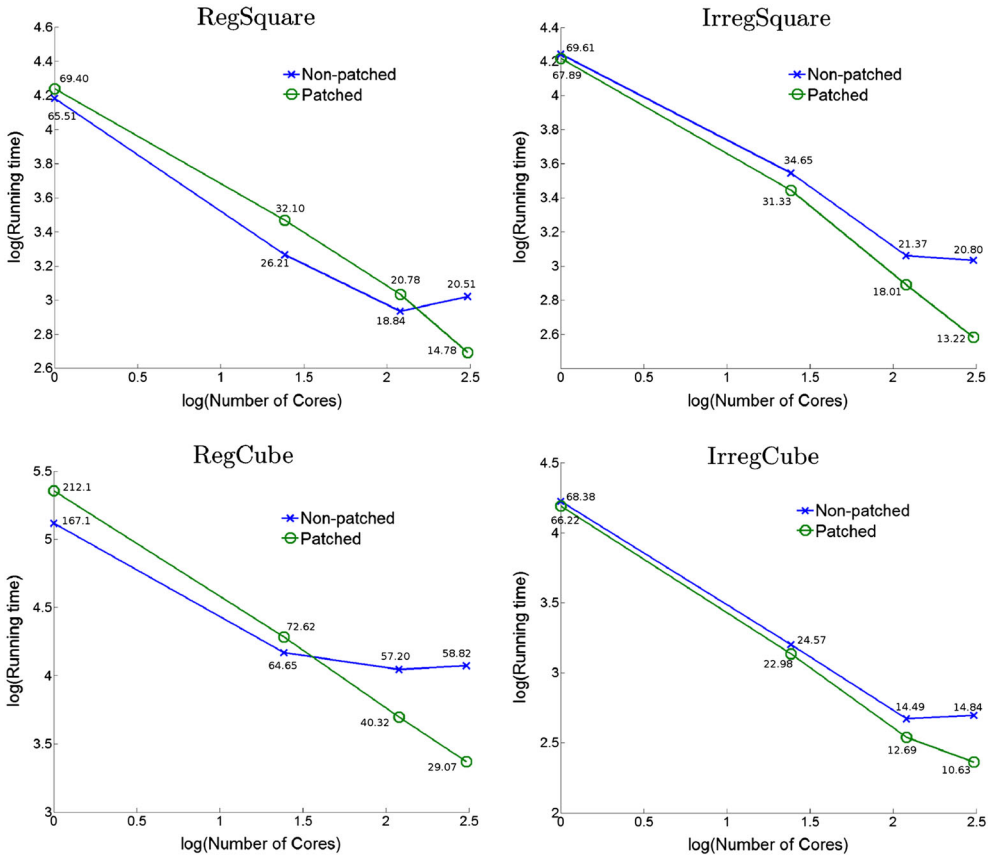

Figure 9. Performance comparison between nonpatched CPU implementation and patched implementation. The numbers in the plot are the running times in seconds, and the log base is e.

fast iterative method, which leads to more computations for more cores, as described in [20]. This supports our claim that with the patched update schemes, each thread accesses data mainly from a single patch (except for boundaries), and in this way, the implementation enforces data locality and achieves better cache performance. In addition, the results show that the proposed implementation scales better on tetrahedral meshes than on triangular meshes. This is because for tetrahedral meshes, the number of active nodes inside each patch is larger in the reinitialization step. The data required by these active node updates are very likely in cache already because each patch is assigned to a thread. Similarly, in 3D cases, the narrowband contains more elements in the evolution step, and each patch within the narrowband has more elements, which leads to more cache hit. Also, for the tetrahedral meshes, the computation is more complicated, and hence, the computational density is higher.

For the patched update scheme, the patch size is a factor that may influence the overall performance, as it affects how the data are loaded into the cache. With larger patch size, each patch has more active nodes in the reinitialization step, and thus, there are more elements inside the larger narrowband for update during the evolution step. These nodes and elements are updated by a single thread, and after a thread updates the first node or element in the patch, the data needed for the following updates are very likely in cache already. However, large patch size may lead to load balancing issue as the workloads of the threads, each updating a corresponding patch, can be very different. Also, if the patch is too large to fit into the cache, the number of cache misses will increase. Table I shows how the patch size affects the performance of our patched multithreaded CPU implementation. It can be seen from the table that there is a sweet spot for the patch size, which achieves the best balance between cache performance and load balancing. In our case, this sweet spot is around

DOI: 10.1002/cpe

Table I. Running times (in seconds) to show patch size influence on performance.

	Size 32	Size 64	Size 128	Size 256
RegSquare	15.40	**14.78**	15.35	18.78
IrregSquare	14.89	**13.22**	15.42	19.22
Sphere	18.15	**15.02**	15.66	17.86
Brain	157.75	142.39	**140.03**	141.89
RegCube	33.08	29.07	**29.05**	30.11
IrregCube	12.89	**10.63**	11.24	11.78

Bold numbers denote the sweet spot for the patch size.

Table II. Running time (in seconds) to show narrowband width influence on performance.

		2	3	5	10	20
IrregSquare	CPU	14.90	14.88	13.22	13.33	15.57
	GPU	3.13	3.14	2.75	2.12	2.46
IrregCube	CPU	13.81	12.90	15.33	55.22	—
	GPU	6.13	5.75	4.94	4.48	7.01

64 for all the test meshes, and the CPU results reported in the following subsection (Section 4.2) are all with patch size 64.

4.2. GPU Performance results

To demonstrate the performance of our proposed schemes on SIMD parallel architectures, we have implemented and tested the levelset method on an NVIDIA Fermi GPU using the NVIDIA CUDA API [23]. The NVIDIA GeForce GTX 580 graphics card has 1.5 GBytes of global memory and 16 streaming multiprocessors, where each streaming multiprocessors consists of 32 SIMD computing cores that run at 1.544 GHz. Each computing core has a configurable 16 or 48 KBytes of on-chip shared memory, for quick access to local data. The code was compiled with NVCC 4.2. Computation on the GPU entails running a kernel function with a batch process of a large group of fixed size thread blocks. This maps well to our patched update scheme that employs patch-based update methods, where a single patch is assigned to a CUDA thread block. In this section, we compare our GPU implementation with the multithreaded CPU implementation with patched update scheme.

As described in Section 2, the narrowband scheme recomputes the distance transform to the zero levelset every few time-steps, and the number of time-steps performed between reinitialization is related to the narrowband width. This width greatly affects the performance of our implementation. When the narrowband width is large, the reinitialization step requires more time to converge, and each evolution step needs to update more nodes that are inside the narrowband. However, with a larger narrowband width, the program needs to perform fewer reinitialization to reach the user-specified total number of time-steps. Table II shows how performance is related to the narrowband width for the IrregSquare and IrregCube meshes. As seen from the table, there is a narrowband width sweet spot for both CPU and GPU performance that achieves the best balance between the narrowband width and the reinitialization frequency. For the CPU, this sweet spot is around five, and for the GPU, it is approximately 10 for the IrregSquare mesh. We obtain similar ideal narrowband width for all other triangular meshes. As described in Section 3, the reinitialization maintains an active patch list instead of an active node list, and this makes our GPU reinitialization efficient for larger narrowband widths. When the narrowband width is smaller than the patch size, those nodes with values larger than narrowband width are not updated in the evolution step, and hence, their distance values need not to be computed in the reinitialization step. This explains why our GPU implementation prefers a larger narrowband width relative to the CPU implementation. For the

DOI: 10.1002/cpe

tetrahedral mesh, although GPU performance sweet spot is the same, five is no longer the optimal narrowband width for the CPU. For 3D tetrahedral meshes, the cost of the reinitialization is dramatically increased, and smaller narrowband width leads to fewer *local_solver* computations. Although with smaller narrowband width, the frequency of reinitialization is increased, performance improvement from fewer *local_solver* calls per iteration outweighs the increased reinitialization frequency. In the following testing results, CPU running times are measured with bandwidth five for triangular meshes and three for tetrahedral meshes, respectively, while GPU running times are measured with narrowband width of 10 for both triangle and tetrahedral meshes.

Table III shows the performance comparison for the time-stepping stage of the levelset equation solver. We present the performance comparison for the reinitialization, evolution, and the total running times separately to demonstrate the performance of the proposed schemes for each step. For the reinitialization with the nbFIM scheme, our proposed GPU implementation performs up to 10× faster than the multithreaded CPU implementation running on a 12-core system. For the evolution, the GPU implementation achieves up to 44× speedup over the same 12-core system. The total running times shown here include CPU–GPU data transfer and time-step computation described in Section 3.4. In addition, from this table and the tables in Section 4.1, we can see that in CPU implementations, the reinitialization step takes a small portion of the total running time, while it takes a large portion in the GPU implementation. This is because in the reinitialization step, the GPU eikonal solver uses the active patch scheme, as described in Section 3, which leads to more computation than in the CPU implementation. Also, the meshes we choose to use in our tests have different maximum valence. Among the triangular meshes, we achieve the greatest GPU over CPU speedup on the RegSquare mesh, because this mesh is regular and has smallest maximum valence. On the other hand, we observe the worst speedup on the IrregSquare and Brain mesh that have larger maximum valences. What is more we can note from Table II is that the performance for the 3D tetrahedral meshes is generally worse for both implementations than that for triangular meshes. This is due to the much higher node valence of the tetrahedral meshes and much more complex computations especially in the reinitialization step. We also observed that for tetrahedral meshes, the kernel functions for the value update in the reinitialization and evolution steps require more registers than are available in hardware, so some locally stored quantities spill into local memory space that has much higher latency than registers. However, overall, our proposed method suits GPU architecture very well, and our GPU implementation achieves large performance speedup compared with an optimized parallel CPU implementation regardless of mesh complexity.

As mentioned in Section 3, lock-free of an algorithm is usually achieved by using atomic operations. However, on GPUs, the atomic operations are expensive. Currently, NVIDIA GPU does not support native double-precision atomic addition and atomic minimum. We have to rely on atomic

Table III. Running times (in seconds) for the reinitialization, evolution, and total, respectively.

	RegSquare	IrregSquare	Sphere	Brain	RegCube	IrregCube
			Reinitialization			
GPU(regular)	0.27(9.7×)	—	—	—	3.03(7.3×)	—
GPU(METIS)	0.65(4.0×)	1.25(3.5×)	1.20(3.8×)	4.91(3.4×)	5.73(3.9×)	3.22(1.4×)
CPU	2.62	4.38	4.53	16.88	22.22	4.53
			Evolution			
GPU(regular)	0.26(44×)	—	—	—	1.02(6.4×)	—
GPU(METIS)	0.28(41×)	0.61(14×)	0.46(17×)	2.32(13×)	2.10(3.1×)	1.26(6.3×)
CPU	11.54	8.46	7.90	30.04	6.54	7.93
			Total			
GPU(regular)	0.69(21×)	—	—	—	4.33(6.7)	—
GPU(METIS)	1.09(14×)	2.02(6.5×)	1.73(7.6×)	7.48(6.5×)	7.83(3.7×)	4.48(2.9×)
CPU	14.78	13.22	13.13	48.39	29.07	12.90

The numbers in the parentheses are the speedups compared against the CPU.

DOI: 10.1002/cpe

Table IV. Running times (in seconds) for the GPU implementations with hybrid gathering and atomic operations.

	RegSquare	IrregSquare	Sphere	Brain	RegCube	IrregCube
Atomic	2.71	6.03	6.03	14.32	17.24	12.25
Hybrid gathering	0.69	2.12	1.73	6.48	7.83	4.48
Speedup	3.9×	2.8×	3.5×	2.2×	2.2×	2.7×

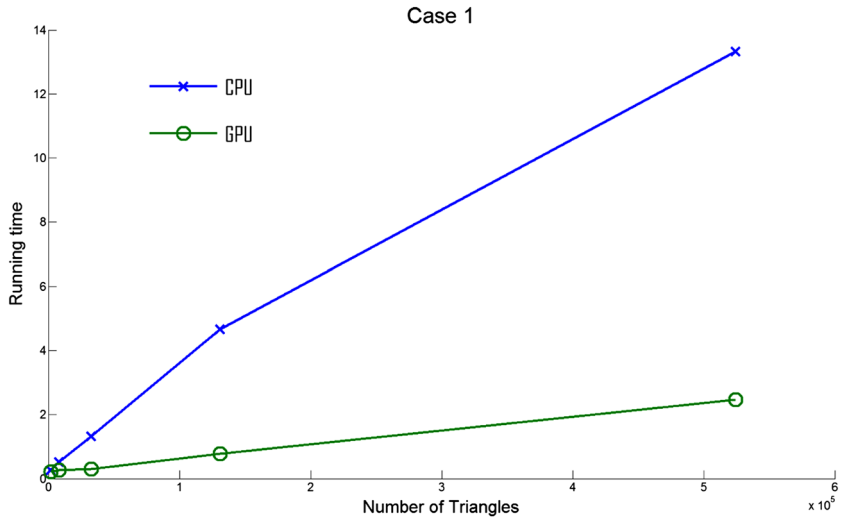

Figure 10. Performance comparison between CPU and GPU implementations for different problem sizes.

compare and swap (atomicCAS) operation to implement the atomic addition and atomic minimum as suggested by Reference [23]. Table IV shows the effectiveness of our lock-free scheme on GPU. This table compares the running times of the GPU implementations using hybrid gathering and atomic operations to solve race condition, respectively. This table shows that the implementation with hybrid gathering gives much better performance than the version with atomic operations.

Finally, Figure 10 shows how the CPU and GPU implementation scale with different mesh sizes. We start from a coarse mesh with 2048 triangles and subdivide it by connecting the midpoints of each edge. We do this four times and obtain five meshes of different sizes. The largest mesh is the RegSquare mesh. It can be seen from the plot that as the mesh size increases, the performance gap widens between the CPU implementation and the GPU implementation. The number of computations per time-step increases with mesh size, which makes the GPU operations more efficient. At lower mesh sizes, the performance difference is not as great because of the low computational density per kernel call.

5. CONCLUSIONS

This work proposes the nbFIM and patchNB schemes to efficiently solve the levelset equation on parallel systems. The proposed schemes combine the narrowband scheme and domain decomposition to reduce the computation workload and enforce data locality. Also, combined with our proposed hybrid gathering scheme and novel data structure, these schemes suit the GPU architecture very well and achieve great performance in a wide range of numerical experiments. The hybrid gathering scheme avoids race conditions without using atomic operations, which are costly on GPUs. This scheme can be applied to more general problems, where data dependency is dictated by a graph structure, and one can choose multiple parallelism strategies corresponding to different graph components to solve these problems. We will explore such problems in our future work. In addition,

DOI: 10.1002/cpe

for many real scientific and engineering applications, a single-node computer does not have enough storage or computing power to perform computations efficiently. We will thus work on extending the levelset equation solver to multiple GPUs and GPU clusters.

REFERENCES

1. Sethian JA. *Level Set Methods and Fast Marching Methods*. Cambridge University Press: Cambridge, 1999.
2. Osher S, Sethian JA. Fronts propagating with curvature dependent speed: algorithms based on Hamilton–Jacobi formulations. *Journal of Computational Physics* 1988; **79**(1):12–49.
3. Barth TJ, Sethian JA. Numerical schemes for the Hamilton–Jacobi and level set equations on triangulated domains, 1997.
4. Joshi A, Shattuck D, Damasio H, Leahy R. Geodesic curvature flow on surfaces for automatic sulcal delineation. *9th IEEE International Symposium on Biomedical Imaging (ISBI), 2012*, Barcelona, 2012; 430–433.
5. Wu C, Tai X. A level set formulation of geodesic curvature flow on simplicial surfaces. *IEEE Transactions on Visualization and Computer Graphics* 2010; **16**(4):647–662.
6. Tan L, Zabaras N. A level set simulation of dendritic solidification of multi-component alloys. *Journal of Computational Physics* 2007; **221**(1):9–40. DOI: 10.1016/j.jcp.2006.06.003.
7. Adalsteinsson D, Sethian JA. A fast level set method for propagating interfaces. *Journal of Computational Physics* 1995; **118**(2):269–277. DOI: 10.1006/jcph.1995.1098.
8. Whitaker RT. A level-set approach to 3D reconstruction from range data. *International Journal of Computer Vision* 1998; **29**:203–231.
9. Bridson RE. Computational aspects of dynamic surfaces. *Ph.D. Thesis*, Stanford University.
10. Strain J. Tree methods for moving interfaces. *Journal of Computational Physics* 1999; **151**(2):616–648.
11. Houston B, Nielsen MB, Batty C, Nilsson O, Museth K. Hierarchical RLE level set: a compact and versatile deformable surface representation. *ACM Transactions on Graphics* 2006; **25**(1):151–175.
12. Lefohn A, Cates J, Whitaker R. Interactive, GPU-based level sets for 3D brain tumor segmentation. In *Medical Image Computing and Computer Assisted Intervention*. MICCAI: Montréal, 2003; 564–572.
13. Jeong WK, Beyer J, Hadwiger M, Vazquez A, Pfister H, Whitaker RT. Scalable and interactive segmentation and visualization of neural processes in EM datasets. *IEEE Transactions on Visualization and Computer Graphics* 2009; **15**(6):1505–1514. DOI: 10.1109/TVCG.2009.178.
14. Fortmeier O, Bücker HM. A parallel strategy for a level set simulation of droplets moving in a liquid medium. *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, Springer-Verlag: Berlin, Heidelberg, 2011; 200–209. (Available from: http://dl.acm.org/citation.cfm?id=1964238.1964261) [Accessed on 22–25 June 2010].
15. Owen H, Houzeaux G, Samaniego C, Cucchietti F, Marin G, Tripiana C, Calmet H, Vazquez M. Two fluids level set: high performance simulation and post processing. *2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*, Salt Lake City, 2012; 1559–1568.
16. Rossi R, Larese A, Dadvand P, Oñate E. An efficient edge-based level set finite element method for free surface flow problems. *International Journal for Numerical Methods in Fluids* 2013; **71**(6):687–716. DOI: 10.1002/fld.3680.
17. Rodriguez JM, Sahni O, Lahey RT, Jr., Jansen KE. A parallel adaptive mesh method for the numerical simulation of multiphase flows. *Computers & Fluids* 2013; **87**:115–131. DOI: http://dx.doi.org/10.1016/j.compfluid.2013.04.004. {USNCCM} Moving Boundaries.
18. Meuer H, Strohmaier E, Dongarra J, Simon H. Top500 supercomputer sites. (Available from: http://www.top500.org/) [Accessed on November 2013].
19. Satish N, Harris M, Garland M. Designing efficient sorting algorithms for manycore GPUs. *NVIDIA Technical Report NVR-2008-001*, NVIDIA Corporation, September 2008.
20. Fu Z, Jeong WK, Pan Y, Kirby RM, Whitaker RT. A fast iterative method for solving the eikonal equation on triangulated surfaces. *SIAM Journal on Scientific Computing* 2011; **33**(5):2468–2488. DOI: 10.1137/100788951.
21. Bell N, Garland M. Efficient sparse matrix-vector multiplication on CUDA. *NVIDIA Technical Report NVR-2008-004*, NVIDIA Corporation, December 2008.
22. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 1998; **20**(1):359–392.
23. NVIDIA. *Cuda programming guide*. (Available from: http://www.nvidia.com/object/cuda.html) [Accessed on February 2014].