ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Doctoral Thesis

High-Performance Fast Iterative Methods for Eikonal Equations

Sumin Hong

Department of Computer Science and Engineering

Graduate School of UNIST

2020

High-Performance Fast Iterative Methods for Eikonal Equations

Sumin Hong

Department of Computer Science and Engineering

Graduate School of UNIST

High-Performance Fast Iterative Methods for Eikonal Equations

A dissertation
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Sumin Hong

12. 10. 2019 of submission

Approved by

[illegible]

Advisor
Won-Ki Jeong

High-Performance Fast Iterative Methods for Eikonal Equations

Sumin Hong

This certifies that the thesis/dissertation of Sumin Hong is approved.

12. 10. 2019

signature [illegible]

Advisor: Won-Ki Jeong

signature [illegible]

Sungahn Ko : Thesis Committee Member #1

signature [illegible]

Young-ri Choi : Thesis Committee Member #2

signature [illegible]

Sam H. Noh : Thesis Committee Member #3

signature [illegible]

Beomseok Nam : Thesis Committee Member #4

Abstract

The eikonal equation has a wide range of applications related to distances or travel time in space, such as geoscience, computer vision, image processing, path planning, and computer graphics. Recently, the research on eikonal equation solvers has focused more on developing efficient parallel algorithms to leverage the computing power of parallel systems, such as multi-core CPUs and graphics processing units (GPUs). However, only a little research literature exists for the massively parallel eikonal equation solver because of its complications related to data and work management. In this dissertation research, I introduce several-fold novel contributions to leverage the high-performance and massive computing platform for a parallel eikonal equation solver.

First, I introduce a novel adaptive domain decomposition method for an efficient multi-GPU implementation of the block-based fast iterative method (FIM). The proposed method expands the sub-domain which is to be processed for each GPU by considering the fair load balancing as the iterative algorithm proceeds. It also provides a locality-aware clustering algorithm to minimize the communication overhead. With this, I solved the parallel performance problems that are often encountered in naive multi-GPU extensions that depend on regular domain decomposition, such as task load imbalance and high communication cost. In addition, it includes several optimization techniques, such as hiding the CPU cost using the CUDA multi-streams and hiding the data transfer costs between multiple GPUs.

Second, I propose an efficient parallel implementation of FIM for a multi-core shared-memory system by using a lock-free local queue approach and provide an in-depth analysis of the parallel performance of the method. In addition, I propose a new parallel algorithm, Group-Ordered Fast Iterative Method (GO-FIM), that exploits the causality of grid blocks to reduce redundant computations, which was the main drawback of the original FIM. The proposed GO-FIM method uses the clustering of blocks based on the updating order where each cluster can be updated in parallel by using multi-core parallel architectures.

Third, I propose a novel algorithm called Causality-Ordered Fast Iterative Method (CO-FIM), that exploits the causality dependency at a node level to reduce redundant computations. Moreover, I propose a new parallel algorithm, Causality and Group-Ordered Fast Iterative Method (CGOFIM), that integrates GO-FIM and CO-FIM. The proposed CGO-FIM determines the updating order at the block level while minimizing the redundancy calculation in the inner block by a node-level causality dependency. The CGO-FIM method has a condition for using both COFIM and FIM interchangeably in the inner block, and it is fully compatible with the lock-free local queue approach, so it can be efficiently implemented for multi-core parallel architectures.

Contents

List of Figures

List of Tables

List of Algorithms

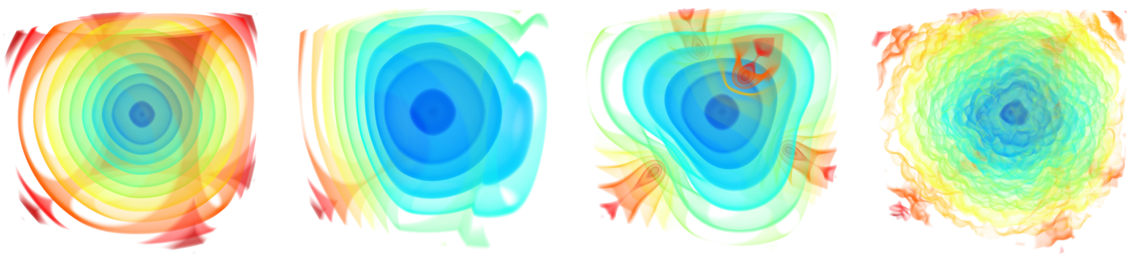
I Introduction

The eikonal equation is a special case of nonlinear Hamilton—Jacobi partial differential equations (PDEs) and is a nonlinear boundary value problem defined by a first-order hyperbolic partial differential equation given as follows:

$$\begin{aligned} H(\mathbf{x}, \nabla\phi) = |\nabla\phi(\mathbf{x})|^2 - \frac{1}{f^2(\mathbf{x})} = 0, \forall\mathbf{x} \in \Omega \subset R^n \\ \phi(\mathbf{x}) = 0, x \in \Gamma \subset \Omega \end{aligned} \tag{1}$$

where Ω is the computational domain in R^n (defined as a rectilinear grid in this paper), Γ is the collection of seed points (i.e., boundary condition), $\phi(\mathbf{x})$ is the travel time or distance from the seed region to location $\mathbf{x}$, and $f(\mathbf{x})$ is a positive speed function defined on $\mathbf{x}$. As one can infer from this definition, the eikonal equation represents the propagation of wave from the seed region where the motion is governed by the speed function, and the solution of the equation represents the *weighted distance* of the shortest path from the nearest seed point by the speed function [1, 2]. Figure 1 shows examples of the iso-contour rendering of the eikonal equation solution (i.e., distance).

The eikonal equation is frequently used in problems related to the distance or the travel time in space, such as geoscience [3], computer vision [4,5], image processing [6,7], path planning [8,9], computer graphics [10–12], seismic travel time computation [13–15], or the computation of brain connectivity maps [16, 17].



(a) Constant (b) Three layers (c) Sinusoidal (d) Correlated random

Figure 1: Color-coded distance map with iso-contours of eikonal equation result. Blue-to-red colors denote the distance to the seed region on each map. Each name describes a feature of the input speed map.

In order to solve the eikonal equation, we need to consider two problems; one is how to accurately discretize the equation on a grid, and the other is how to compute the solution of the nonlinear PDE numerically. For discretization, a Godunov upwind difference scheme is commonly used [4,18–20]. On a 3D Cartesian grid, the first order Godunov upwind discretization

$g(\mathbf{x})$ of the Hamiltonian $\mathrm{H}(\mathbf{x}, \nabla\phi)$ can be defined as follows:

$$g(\mathbf{x}) = \left[\frac{(U(\mathbf{x}) - U(\mathbf{x})^{xmin})^+}{h_x}\right]^2 + \left[\frac{(U(\mathbf{x}) - U(\mathbf{x})^{ymin})^+}{h_y}\right]^2 + \left[\frac{(U(\mathbf{x}) - U(\mathbf{x})^{zmin})^+}{h_z}\right]^2 - \frac{1}{f(\mathbf{x})^2} \quad (2)$$

where $U(\mathbf{x})$ is the discrete approximation to ϕ at node $\mathbf{x} = (i, j, k)$, $U(\mathbf{x})^{pmin}$ is the minimum U value among two adjacent neighbors of $U(\mathbf{x})$ along the axis $p \in \{x, y, z\}$ directions, h_p is the grid spacing along the axis p, $f(\mathbf{x})$ is the speed function at $\mathbf{x}$, and $(n)^+ = \max(n, 0)$. As $U(\mathbf{x})$ is the only unknown in Eq 2, a closed-form solution of U can be found by solving a quadratic equation.

There exists a vast amount of literature on eikonal equation solvers. Most eikonal solvers are directly related to the fast algorithms developed to find the shortest paths in directed graphs [21–23]. The most popular methods are Fast Marching Method (FMM) [20, 24] and Fast Sweeping Method (FSM) [25–27]. FMM is the most representative method for the eikonal equation solver. The core idea of FMM is using an ordered data structure, e.g., Heap, to manage the correct updating order and the narrow list of active points while using an upwind discretization scheme for the finite-difference computation. The algorithm is basically identical to Dijkstra's graph shortest path algorithm [28], but Hamiltonian-based numerical distance computation is used. FSM is another popular method that improves the efficiency of iterative methods by using a pre-defined updating order (i.e., Gauss-Seidel updating with alternating orders [29, 30]). This approach has advantages in parallelization because it does not use a sorted data structure. Recently, many algorithms have been proposed to solve the eikonal equation using modern parallel systems. Most of them are variants of the FMM or FSM; they include parallelization methods for multi-threaded CPUs [31] [32], GPUs [33] [34] and a distributed system [35] [36].

The other type of eikonal solver is the adaptively updating of active points without following a strict update ordering. This type of algorithm is rooted from a *label-correcting* algorithm for the Bellman-Ford shortest path problem on a graph. The Fast Iterative Method (FIM) [17,19,37] is a variant of the label-correcting algorithm and is specifically designed for recent many-core processors, such as GPUs. This algorithm uses an *active list* to manage the grid nodes being updated by the solver. In contrast to traditional eikonal equation solver, this active list is based on a simple queue or list and does not belong to expensive ordered data structures, and all the grid nodes in the list are updated concurrently. Because of this concurrency, the FIM algorithm has an inherent fine-grain parallelism. However, in the original FIM paper, the authors only introduced the main algorithm and its implementation on a single GPU. Even though the FIM algorithm embraces a potential to be applied to any parallel computing systems other than the GPU, it has yet not been fully addressed elsewhere. In addition, because the main design choice for the FIM algorithm mainly focused on increasing parallelism rather than algorithmic optimality, its worst-case performance may vary depending on the complexity of the input speed function.

1.1 Motivation

The main motivation of this dissertation research comes from the following observations.

First, the recent research on eikonal equation solvers has focused more on developing efficient parallel algorithms to leverage the computing poIr of parallel systems, such as multicore CPUs and GPUs. Nevertheless, only a little research literature exists for the massively parallel eikonal equation solver because of its complications related to data and work management. Even though the original FIM algorithm is inherently parallel, a naive parallelization does not guarantee sufficient performance benefits, and the extension of FIM to the modern parallel system has not yet been fully addressed elsewhere.

Second, FIM suffers from several limitations because of the fact that the propagation of the active list does not conform to the solution of the equation. Because of the missing ordered data structure, it allows multiple updates of the node until it converges completely. In terms of complexity, the computational cost of FIM is O(kn), but the actual performance highly depends on the input [19]. The main motivation behind the original FIM is abandoning the ordered data structure that hinders parallelization with an observation that k is usually small. However, this optimism is not sufficient. If the input speed domain is extremely complex, FIM requires a considerable amount of iterative computation and the overall solution cost is significantly increased.

The goal of this dissertation research was to develop novel parallel algorithms for the eikonal equation. It included a study on the efficient implementations of FIM for various modern parallel systems such as multi-core CPUs and multi-GPU systems. In addition, the research targeted the reduction of redundant computations, which was the main drawback of the original FIM, by exploiting causality information.

1.2 Contributions

The goal of this dissertation research was to develop novel parallel algorithms for the eikonal equation. The main contributions of this dissertation research are several-fold. Among these contributions, several methods have been published as my own work [38, 39].

- **A Multi-GPU Fast Iterative Method using Adaptive Domain Decomposition**
  I propose a novel on-the-fly adaptive domain decomposition method for an efficient implementation of the block-based FIM on a multi-GPU system. For the adaptive domain decomposition, I introduce a novel history-based active list prediction algorithm that makes the best effort to predict the future computing domain to distribute tasks evenly across GPUs. I also propose a locality-aware clustering algorithm to minimize the inter-GPU data communication without impairing the load balancing performance. I conducted a rigorous performance analysis on various test cases. The analysis included a comparison of our method to the naive implementation of multi-GPU BlockFIM and other multi-GPU eikonal solvers.

- **A Group-Ordered Fast Iterative Method for Eikonal Equations**
  I propose an efficient parallel implementation of FIM for multi-core shared memory systems. Even though the original FIM algorithm is inherently parallel, a naive parallelization does not guarantee sufficient performance benefits. I propose a local queue-based parallelization approach that can avoid expensive lock synchronization while ensuring good load balancing between threads. In addition, I further improve our parallel FIM algorithm by proposing a novel group-based updating scheme where the updating order is determined by the solution on a coarse level grid, called GO-FIM. This approach can effectively eliminate the drawback of the original FIM without maintaining an expensive global ordered data structure, such as Heap, while providing superior parallel performance by clustering similar blocks for concurrent updating. I present an in-depth analysis of the performance of both the methods on various test datasets and compare them with the state-of-the art eikonal solvers on multi-core CPUs, and finally show how GO-FIM effectively improves FIM on the GPU.

- **A Causality-Ordered Fast Iterative Method for Eikonal Equations**
  Third, I propose a novel algorithm Causality-Ordered Fast Iterative Method (CO-FIM) that exploits the causality dependency at the node level to reduce redundant computations. I propose a novel parallel algorithm, Causality and Group-Ordered Fast Iterative Method (CGO-FIM) that integrates GO-FIM and CO-FIM. The proposed CGO-FIM determines the updating order at the block level while minimizing the redundancy calculation in the inner block by a node-level causality dependency. The CGO-FIM method has a condition for using both COFIM and FIM interchangeably in the inner block and is fully compatible with the lock-free local queue approach; therefore, it can be efficiently implemented for multi-core parallel architectures. I present an in-depth analysis of the performance of both the methods on various test datasets and compare them with the state-of-the art eikonal solvers on multi-core CPUs.

1.3 Document Organization

In Section II, I introduce previous work on serial and parallel eikonal equation solvers. In Section III, I start our discussion by introducing some basic definitions and the original FIM [17, 19] algorithm. A novel multi-GPU implementation of FIM using the on-the-fly adaptive domain decomposition method is discussed in Section IV. An efficient parallel implementation of FIM for multi-core shared memory systems, and Group-Ordered FIM that exploits the causality of grid blocks to reduce redundant computations are introduced in Section V. Section VI details the application of a node-level causality-ordering to FIM. Finally, Section VII wraps up the dissertation and suggests some future research directions.

II Related Work

There exists a vast amount of literature for eikonal equation solvers. Early work had focused on developing numerical methods for computing the viscosity solution of Hamilton-Jacobi equations. Many of them use finite-difference for approximating differential operators and apply a fixed-point iteration method over the entire grid [4, 40–42]. In such methods, the entire grid points must be updated until converged, so the worst-case complexity can be as high as $O(N^2)$. In order to improve inefficiency of such iterative methods, adaptive update schemes have been proposed. One of them is exploiting the causal relationship of the solution for the boundary value PDE problems (i.e., using a strict dependency between neighborhood grid points when computing the solution).

A popular method is Fast Marching Method (FMM) proposed by Sethian [20, 24]. The core idea of this method is using an ordered data structure, e.g., Heap, to manage the correct updating order and the narrow list of active points while using upwind discretization scheme for finite-difference computation. The algorithm is basically identical to Dijkstra's graph shortest path algorithm [28, 43], but Hamiltonian-based numerical distance computation is used. Therefore, the complexity of the algorithm is $O(N\log N)$, which is worst-case optimal [44]. However, when the speed map is not extremely complicated, managing Heap can be a significant overhead. In addition, FMM requires a strict serial updating order to follow, which hinders parallelization on many-core parallel systems. Therefore, FMM may not be the fastest solution when parallel computing is considered.

Another approach to improve efficiency of iterative methods is employing a pre-defined updating order. For example, Fast Sweeping method (FSM) [25] employed Gauss-Seidel updating with alternating iteration order. Since it is well-known that Gauss-Seidel update converges faster than Jacobi update, the proposed method works well on a certain type of problems, i.e., datasets having straight characteristic paths. Due to its simplicity, FSM has been adopted to different Hamiltonian discretization [45], discontinuous Galerkin finite element discretizations [46], distance computation on unstructured grids [47]. Bak *et al.* [48] proposed the Locking Sweeping Method (LSM) to improve the performance of FSM by using boolean flags to skip unnecessary update.

The other type of eikonal solver is adaptively updating of active points without following a strict update ordering. This type of algorithm is rooted from a *Label-correcting* algorithm for the Bellman-Ford shortest path problem on a graph, such as Polymenakos *et al.* [49], Falcon *et al.* [50, 51], and some parallel algorithms by Bertsekas *et al.* [52], which is based on a simple First In First Out (FIFO) queue to store active points only and update points iteratively until the queue becomes empty. This type of solvers show $O(kN)$ complexity where k depends on the input data. In many cases, k could be much smaller than N and there is no overhead to manage ordered data structure, so this type of algorithm runs faster than worst-case optimal algorithms. Adopting a label-correcting algorithm for a general Hamilton-Jacobi equation solver on unstructured grids is introduced by Bornemann *et al.* [53]. Later, Jeong *et al.* [17, 19] introduced Fast Iterative Method (FIM), a variant of label-correcting method specifically designed for massively parallel architecture. FIM manages a list of active points where insertion and removal of points is determined by the *convergence* of the solution, and all the active points in the list can be updated in parallel. In addition, unlike Bornemann *et al.* [53], any active points that are not converged do not leave the active list. FIM also introduced the *BlockFIM* algorithm, where the input grid is split into blocks and each block is treated as a unit of parallel update, which maps well to SIMD parallel architecture such as the Graphics Processing Unit (GPU). Fu *et al.* [54] further extends FIM to compute the geodesic distance on unstructured meshes on the GPU. Bak *et al.* [48] introduced the single queue method, which is similar to Bornemann *et al.* [53] except that causal ordering is used to determine the neighbor nodes to be added to the queue.

Many algorithms have been proposed to solve the eikonal equation for parallel systems. Most of them are variants of the Fast Marching Method (FMM) [24] or Fast Sweeping Method (FSM) [25]. Zhao [31] proposed two parallel implementations of his original FSM, one for shared memory system and the other for distributed memory system. In this paper, the author reported that shared memory version performs better due to the communication overhead of distributed memory version. However, shared memory version also has a inherent limitation that only scales up to 2^d parallel processors (d = dimension of data) because the algorithm relies on parallel Gauss-Seidel update on different iteration orders. Detrixhe *et al.* [32] proposed a parallel sweeping method that overcomes the limitation of Zhao's parallel FSM. It uses the Cuthill-Mckee ordering [55], which clusters grid points on diagonal lines or planes for sweeping, and therefore points on such clusters can be updated concurrently. This approach introduces some overhead of computing non-axis aligned ordering but increases scalability of the parallel performance. A similar idea has been employed to solve the eikonal equation on 2D parameteric surfaces, called the Parallel Marching Method (PMM), by Weber *et al.* [10]. In this work, an alternative discretization and updating sequence are proposed, and their efficient GPU implementation using fine-grain parallelism is also introduced. Gillberg *et al.* [33] proposed a 3D parallel marching method on a single GPU to solve the eikonal equation on a 3D rectilinear grid using a finite difference method by adopting Weber's alternative stencil formulation idea. Later, this method is further extended to multi-GPU systems by Krishnasamy *et al.* [34].

Many parallelization methods for eikonal equations based on domain decomposition methods have been proposed. Herrmann [56] first tried to parallelize the FMM based on the domain decomposition method; Each sub-domain to operate its own min-heap. For synchronizing between sub-domain, it introduced a rollback mechanism. However, this rollback operations introduced significant computation and communication overheads. Breuss *et al.* [57] proposed a shared-memory domain decomposition parallelization of the fast marching method; and Tugurlan [58] suggested a distributed-memory parallelization of the fast marching method based on a domain decomposition approach. Yang *et al.* [36, 59, 60] proposed a parallelization model of the FMM for the distributed system and it achieves high parallel efficiency by reducing the communication frequency between each split sub-domain; In addition, Weinbub *et al.* [61, 62] proposed a shared memory variant of this algorithm. Detrixhe *et al.* [35] suggested a hybrid approach of FSM for a distributed system. It combined the domain decomposition model [31] for coarse-level and their Cuthill-Mckee ordering based FSM algorithm [32] for fine-level. Shrestha *et al.* [63] also proposed a multi-level domain-decomposition strategy approach for a distributed system based on Detrixhe's FSM [32]. This introduced the idea of Cuthill-Mckee ordering on the coarse level either.

Recently, researchers are actively developing hybrid approaches to overcome the limitation of existing methods. Bak *et al.* [48] introduced the two queue method, a variant of a single queue label-correcting update method, to roughly prioritize active points based on its value – high and low – so that the active points having low values are updated before those having high values. This is an inexpensive alternative to FMM because it does not use an expensive ordered data structure but can effectively control the expansion of the active list. Gillberg [3] proposed a similar two-list method using the average distance value as a threshold to restrict the propagation of active points. More recently, Chacon *et al.* [64] introduced a different hybrid technique – instead of splitting the active list into two groups based on distance values, they use two different scales (coarse and fine) so that the propagation of active list is determined by the coarse level grid while the solution is computed on the fine level grid. In this paper, they introduced three different methods – Fast Marching-Sweeping Method (FMSM), Heap-Cell Method (HCM), and Fast Heap-Cell Method (FHCM). FMSM uses Fast Marching for computing ordering on the coarse grid while modified Fast Sweeping is used to compute solutions on the fine level grid. Since Fast Marching on the coarse grid does not capture all cell inter-dependencies, HCM employs an ordered (using Heap) label-correcting method for coarse level ordering while LSM is used to speed up the fine level solution computation. FHCM is an inexact version of HCM to speed up the computation by sacrificing the accuracy. A parallel version of HCM, the Parallel Heap-Cell Method, has been recently proposed by the same authors [65, 66].

As I reviewed in this section, the current research trend in eikonal equation solver is mainly in two directions – one is developing parallel algorithms and the other is improving efficiency of solvers. In this dissertation, I tackle two problems at the same time by proposing a parallel implementation of FIM and a ordered variant of FIM.

III Background

We start our discussion by introducing some basic definitions and the original FIM [17, 19] algorithm first.

3.1 Definitions and Notation

We refer to *node* as a grid point on Ω defined by an n-tuple of numbers (i, j, k). We define an *edge* as a line segment that directly connects two nodes whose length defines a grid length h_p along the corresponding axis $p \in \{x, y, z\}$. We define an *adjacent neighbor* as a node connected by a single edge. For example, the node $y = (i + h_x, j, k)$ is the adjacent neighbor of the node $x = (i, j, k)$ along the positive x direction. In this paper, we focus on the three dimensional case only ($n = 3$).

We use the term *converged* to represent the status when the solution of the eikonal equation computed in the current iteration is not smaller than the solution from the previous iteration, which does not represent the global convergence of the solution.

3.2 Fast Iterative Method for Eikonal Equation

The FIM is an iterative algorithm that adaptively updates the solutions that are currently affected by the wave-front. For them, the FIM maintains a narrow band, called the *active list*, for storing the grid nodes that are being updated. In here, the nodes in the active list have a looser relationship so that they can be updated simultaneously. A node can be removed from the active list only when it is converged; otherwise, it remains in the list and is updated again. In addition, a converged node activates its non-converged adjacent nodes, and any converged node can be reactivated later even though it is inactivated previously.

As shown in Algorithm 1, FIM iteratively updates the solution of the nodes in the active list L until the list becomes empty. FIM is an iterative method – meaning that each node can be updated multiple times. A node can be removed from the active list only when it is converged (otherwise, it remains in the list and is updated again in the following iteration), which is the main difference from conventional label-correcting algorithms that use a FIFO queue to remove the top node immediately. A converged node activates its non-converged adjacent nodes, and any converged node can be reactivated later even though it is inactivated previously.

In FIM, there is no assumption on the updating order of nodes, which allows a straightforward parallelization of the algorithm by splitting the for loop into multiple disjoint sub-loops (line 8 in Algorithm 1) and processing them concurrently using parallel threads, i.e., using OpenMP `parallel for` clause. However, some operations in the algorithm may cause race conditions, such as updating the solution (i.e., $U(\mathbf{x}_{nb}) \leftarrow q$) and adding $\mathbf{x}_{nb}$ to L in `if` $\sim$ `else` block in line (15) in Algorithm 1, because the grid and the active list are shared among different threads and multiple threads may attempt to access them at the same time. A simple solution to avoid this race condition is using a mutex (e.g., lock) to allow only one thread to access shared memory location and active list at any given time. However, lock synchronization is an expensive operation, especially for active list access, and such a naive parallel implementation using locks causes too much overhead, which will result in poor scaling performance for a large number of threads.

Algorithm 1: Fast Iterative Method

```
Input: Grid $\Omega$, Solution $U$, Active list $L$
/* Initialization */
1  forall $x \in \Omega$ do
2      if $x$ is a source node then
3          $U(x) \leftarrow 0$
4          add $x$ to $L$
5      else
6          $U(x) \leftarrow \infty$
/* Compute new solutions for $L$ */
7  while $L$ is not empty do
8      forall $x \in L$ do
9          $p \leftarrow U(x)$
10         $q \leftarrow$ solution of $g(x) = 0$
           /* If not converged */
11         if $p > q$ then
12             $U(x) \leftarrow q$
           /* If converged */
13         else
               /* Check adjacent neighbor nodes for reactivation */
14             forall $x_{nb}$ adjacent to $x$ do
15                 if $U(x_{nb}) > U(x)$ and $x_{nb} \notin L$ then
16                     $p \leftarrow U(x_{nb})$
17                     $q \leftarrow$ solution of $g(x_{nb}) = 0$
18                     if $p > q$ then
19                         $U(x_{nb}) \leftarrow q$
20                         add $x_{nb}$ to $L$
21             remove $x$ from $L$
```

IV MG-FIM: A Multi-GPU Fast Iterative Method using Adaptive Domain Decomposition

4.1 Introduction

The FIM is an iterative algorithm that adaptively updates the solution of the eiknoal equation defined on the grid. The FIM maintains a narrow band, i.e., *active list*, for storing the grid nodes to update. The main idea is that the active list is not constructed based on a strict causal relationship (i.e., dependency) as in the FMM, which allows concurrent updating of multiple nodes. A node can be removed from the active list only when it is converged; otherwise, it remains in the list and is updated again. In addition, a converged node activates its non-converged adjacent nodes, and previously converged node can be reactivated (i.e., added to the active list again) later for further updating.

The BlockFIM is a variant of the FIM, which is specifically designed for SIMD architecture, such as GPUs. The BlockFIM splits the computational domain into disjoint blocks, in which each block consists of multiple nodes (for example, we mainly use an $8 \times 8 \times 8$ block) and treats the block as a basic compute primitive. Therefore, in the BlockFIM, the active list manages blocks instead of nodes. When a block is updated, all the nodes in the block are updated concurrently by the parallel computing cores in the GPU.

The FIM is quite different from other iterative parallel algorithms for the eikonal equation. As shown in Figure 2, other parallelized eikonal equations often have a regular computation sequence - for example, a parallel fast sweeping method (PFSM) presented by Detrixhe *et al.* [32] based on a Cuthill-McKee ordering [67], and a parallel marching method (3DPMM) by Gillberg *et al.* [33], which updates the grid through the in axial directions. However, the shape of active list for the FIM is an irregular. It basically proceeds with a waveform, but it often has a reactive point (red circle). This makes difficult to predict the progress of the FIM, and it further decreases the performance of parallelization.

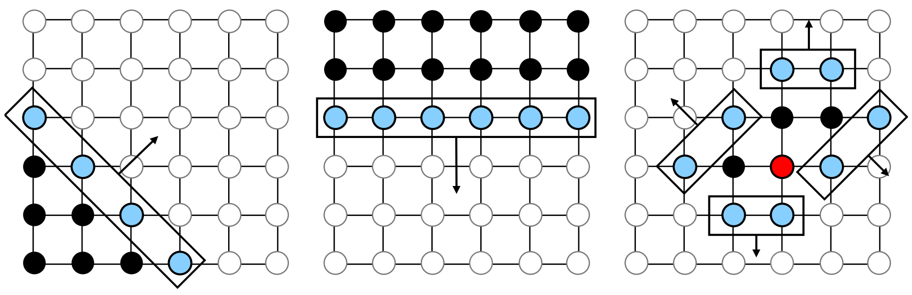


Figure 2: Comparison of different iterative parallel algorithms for eikonal equations. PFSM and 3DPMM have regular update sequences, whereas FIM update order have irregular form.

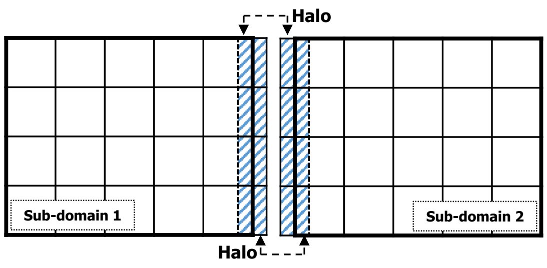


Figure 3: Domain decomposition and Halo communication

4.2 Multi-GPU extension of the BlockFIM

The original BlockFIM is developed only for a single GPU. Therefore, we discuss how the BlockFIM can be extended to multiple GPUs on a shared memory system. A straight-forward extension of the BlockFIM would be using a *domain decomposition* method; the computational domain is split into sub-domains so that each of them can be concurrently updated by a GPU. Each sub-domain is a collection of blocks, and partially overlaps with its adjacent sub-domains around its boundary, called a *halo* (Figure 3). The halo allows each sub-domain to be synchronized via inter-GPU communication while it is independently processed. Since the amount of communication depends on the total size of the halo region, domain decomposition strategies often affect the performance of the method.

Algorithm 2 is a description of the multi-GPU BlockFIM algorithm based on the domain decomposition method. This algorithm is almost identical to the original BlockFIM except the following differences; it uses sub-active lists (i.e., L_i) to accommodate multiple GPUs, and the halo communication step is included to synchronize between sub-domains. First, we assume that block-level domain decomposition has been completed in the initialization phase; For N devices, the total block is grouped into N small groups. Here, any kind of decomposition method can be applied. A initial active list (a set of block containing seed points) is also split into N sub-lists along domain decomposition.

This algorithm can be divided into roughly four parts. Step 1 updates the blocks and the corresponding nodes simultaneously, which belong to the active list. In this time, This computation is repeated n times (Algorithm 2 line 8). C_{node} and C_{block} represent the convergence of the nodes and the block, respectively. When all of the nodes in block b converge, then C_{block} is true; otherwise, it is false.

The next process is the expansion of the active list. It examines the adjacent neighbor blocks based on the converged blocks in step 1. We split this process into two steps; one is indexing of the adjacent neighbor blocks (called as the candidate list L^c) in step 2, and other is examining of converged status the blocks in the candidate list (step 4). As same as the active lists, the candidate list also split into N sub-lists (L_i^c).

For data synchronization between sub-domains, halo communication should be operated between step 1 and step 4; we put this in step 3. For efficient halo communication, we used several

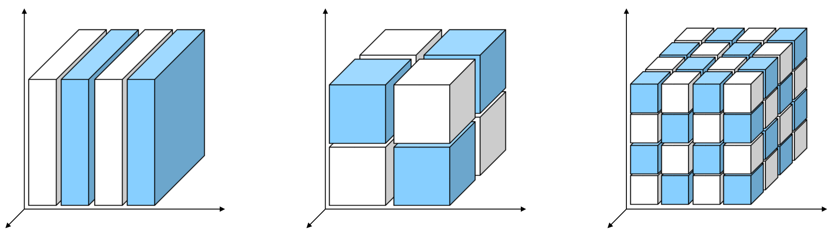

(a) 1-axis, multi-split (b) 3-axis, single-split (c) 3-axis, multi-split

Figure 4: Examples of regular domain decomposition on a 3D rectilinear grid. (a) is multiple splits along one axis, (b) is single split along each axis, and (c) is multiple splits along each axis.

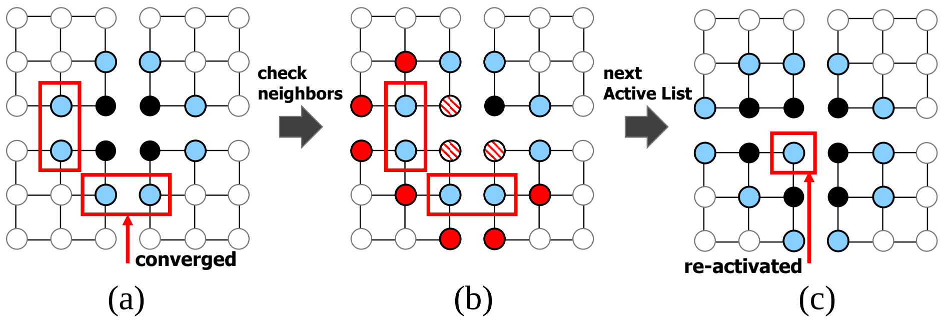


Figure 5: Example of load imbalance in regular domain decomposition. Blue circles are active blocks, black circles are converged blocks, and red circles are candidate blocks to be active in the forward iteration. Due to partial convergence of active list, blocks are not evenly distributed in (c).

tricks. Only the blocks belonging to the active list (that are actually updated) participating to halo communication. Also, the data is grouped as much as possible. For example, a set of blocks which sent from sub-domain i to sub-domain j are packaged in a single continuous array form.

Barrier synchronization between each GPU is required before-after the halo communication and at the end of the loop. This algorithm continues until all sub-active lists become empty, which means that all blocks are converged.

Load imbalance problem

A commonly used domain decomposition strategy is simple axis-aligned grid splitting, as shown in Figure 4. For example, Krishnasamy [34] use 1D axis-aligned splitting and Yang [36] use multiple splits along each axis. This regular domain decomposition works well for most parallel eikonal equation solvers because such methods rely on regular update schemes, e.g., updating the entire plane. Any domain decomposition strategy can be applied to the multi-GPU extension of the BlockFIM introduced earlier. However, the shape of active list can vary depending on the

Algorithm 2: MULTI-GPU FIM ALGORITHM

```
   Input: Input: Set of input X, Set of block V, active list L
1  N ← the number of total device
2  Split $L$ to sublists $L_i$ for all $i \in N$
   /* Parallel Section for devices */
3  while any $L_i \neq \emptyset$ do
4      $i$ ← device id
5      $L_i^c \leftarrow \emptyset$
       /* step 1: Solve active list */
6      forall $b \in L_i$ do
7          for $i$=0 to $n$ do
8              forall $x \in X_b$ do in parallel
9                  $U(x), C_{node}(x) \leftarrow$ solution of $g(x)$
10         $C_{block}(b) \leftarrow$reduction$(C_{node}(x))$
       /* step 2: Collect Adjacent Blocks */
11     forall $b \in L_i$ do
12         if $C_{block}(b) = TRUE$ then
13             remove $b$ from $L_i$
14             for each adjacent block $b_{nb}$ of $b$ do
15                 $j$ ← index of domain for $b_{nb}$
16                 if $b_{nb}$ not in $L_i^c$ then
17                     insert $b_{nb}$ to $L_i^c$
18     Barrier synchronization
       /* step 3: Halo Communication */
19     Halo communication for block in $L$
20     Barrier synchronization
       /* step 4: Check Adjacent Blocks */
21     forall $b \in L_i^c$ do
22         forall $x \in X_b$ do
23             $U(x), C_{node}(x) \leftarrow$ solution of $g(x)$
24         $C_{block}(b) \leftarrow$reduction$(C_{node}(x))$
25         if $C_{block}(b) = FALSE$ then
26             insert $b$ to $L_i$
27     Barrier synchronization
```

input speed map, so distributing active blocks evenly across GPUs from sub-domains generated by static, regular domain decomposition is infeasible in most cases. For example, Figure 5 shows the regular domain decomposition of a 2D grid where circles represent blocks and rectangles represent sub-domains (in this example, the input grid is decomposed into four sub-domains). In Figure 5 (a), the initial active list is evenly distributed across four sub-domains so that two blocks are located in each sub-domain (marked in blue). After one BlockFIM update, only a half of the active list converged (Figure 5 (b), this is marked in a red rectangle) and adjacent blocks are added to the candidate list (marked in a red circle). In the next iteration, the converged blocks will be removed from the active list and some of its adjacent neighbor blocks are activated (Figure 5 (c)). Because the active list expands towards the bottom-left direction in this example, more active blocks are located in the bottom-left region (four blue circles) than in other sub-domains, which causes a load imbalance. This was not the case for other parallel eikonal solvers, especially sweeping algorithms, because they update the entire grid in each sweeping iteration. Therefore, we need a more flexible domain decomposition method to address this issue, which will be discussed in the following section.

4.3 MG-FIM

The load imbalance problem shown above is caused by the fact that the actual amount of task cannot be determined before the FIM runs because the active list changes dynamically. To address this issue, we propose an on-the-fly dynamic domain decomposition method adapting the changes in the active list. The strategy we propose is to dynamically decompose the domain along the propagation of the active list; as the active list expands, we gradually expand the sub-domains so that roughly the same number of active blocks are allocated to each sub-domain to maximize the load balance.

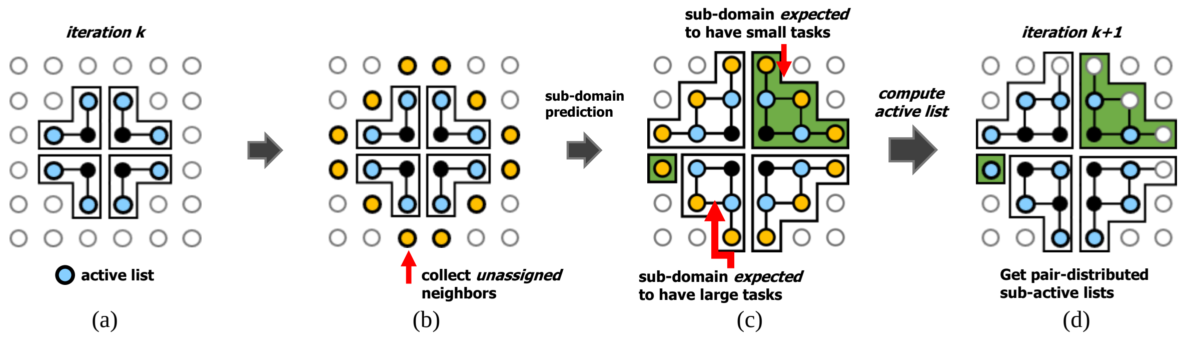

iteration k
active list
(a)
collect unassigned neighbors
(b)
sub-domain prediction
sub-domain expected to have small tasks
sub-domain expected to have large tasks
(c)
compute active list
iteration k+1
Get pair-distributed sub-active lists
(d)

Figure 6: Simple description of our adaptive model. There are four sub-domains, and a block that has never been updated does not belong to any domain (*unassigned*). We assign the surrounding unassigned blocks to the sub-domain before updating the active list.

On-the-fly adaptive dynamic domain decomposition

Our goal is increasing the efficiency of parallelism by optimizing each GPU devices have a similar amount of computing tasks per iteration. This requires that each sub-domain assigned to GPU device have a similar size of the sub-active list. The proposed adaptive domain decomposition algorithm extends the sub-domains considering pair load balancing so that the sizes of the sub-active lists are similar.

In the BlockFIM algorithm, the expansion of the active list occurs after updating distance and checking the status of convergence of blocks. And at this time, the complexity of the input speed map has a large influence on the expansion of the active list. If the input speed map is relatively simple, a block in the active list will converge with a small number of calculations, and will have less chance of reactivation. In the opposite case, a block is updated in many times and frequently reactivated. Therefore, algorithm requires many iterative computation and the overall solution cost increased. Because an iterative update per a block is quite dynamics, the shape of the active list is very irregular. However, if we can learn the features of the input speed map, then we can estimate the size of the next active list from the current active list. Thus, our idea for handling adaptive domain decomposition is clear. We estimate the future size of the sub-active list for each sub-domain, and then extend the sub-domains adaptively so that the sub-domains achieved pair-load balancing.

A brief description of the our approach is is shown in Figure 6. In (a), there are four sub-domains, sub-active lists, and *unassigned* blocks that don't belong to any sub-domain. In (b), we collect all unassigned neighbor blocks (yellow circles), based on the current active lists (blue circle). After that, for each sub-active list, we estimate how many blocks will be remain, and how many blocks will be reactivated in the next iteration. For example, in (c), the upper-right sub-domain is expected to have the lowest number of sub-active lists; so we assign more number of unassigned blocks to the sub-domain. The last step (d) is result of propagation of active list. If the prediction result is appropriate, then each sub-domain will have a similar amount of sub-active list in the next iteration, and we can achieve high compute utilization.

History-based active list prediction

For our adaptive domain decomposition algorithm, the most important thing is an accurate prediction of the change in active list size. Here,to estimate the size of the active list, we need to learn about the feature of the input speed map. However, a high-level and precise learning required expensive steps. Instead, we propose a simple method for predicting the size of the active list based on historical rates. The size of the active list can be decomposed as follows:

$$|L_{next}| \approx |L| * (R^{up} + (1 - R^{up}) * (\alpha + \beta * R^{re})) \tag{3}$$

For an active list, some of the blocks will remain for continuous updating, while others will converge. The rate of update (or remaining to the active list) is R^{up} and the others becomes the rate of convergence $(1 - R^{up})$. A converged block collects the neighbor blocks to proceed active list. There are two types of neighbor blocks, first types are the block which is not computed yet, and second types are the block which calculated more than ones. The ratio between the number of converged blocks and un-computed neighbor blocks is α; the ratio between pre-computed neighbor blocks is β. The un-computed blocks must be included in the next active list, however, the rate of reactivation of the pre-computed blocks (R^{re}) depended on the feature of the speed map.

The size of in the active list can be described using these four variables. Among them, R^{up} and R^{re} are highly related to the feature of the input speed map. However, we found these variables are not changed very rapidly as the iteration proceeds. Therefore, these values can be easily learned by referring to historical values. We learned this value by moving average for each sub-domain. The update rate of moving average we used was 0.3. For α and β, these value are not strongly related to the input speed map. So it is more reasonable to classify these as fixed values. In conclusion, we can easily expect that if a sub-domain has a high value of R^{up} and R^{re}, and then the sub-domain will have large sub-active lists in the next iteration. Our adaptive domain decomposition method using these values is introduced in Algorithm 3.

Algorithm 3 is an extension of Algorithm 2 to use adaptive domain decomposition. Therefore, step 1 to step 4 of the algorithm are the same, and step 0 which is on-the-fly domain decomposition method is additionally attached. In addition, the history-based values R^{up} and R^{re} are managed for each sub-domain for historical based active list prediction (for example, R_i^{up} is value for sub-domain i). This step starts by collecting the unassigned block around the active list into an unassigned list (L^U). The process from line 10 is a calculation of the cost of fair distributing for unassigned blocks. Using R^{up}, R^{re} and the size of the current sub-active list, we calculate the value of *expected*. If a value of $expected_i$ is high, then a sub-domain i has high probability that have more active blocks in the next iteration, then we assigned fewer unassigned blocks to the sub-domain. In this case, we used the value of *total* and *target* to determine how many unassigned blocks to allocate to sub-domains. Our goal is to equalize values that summation of the number of newly assigned block and *expected*. However, this *target* value does not represent the number of sub-active lists in the next iteration. Because there is no guarantee that newly allocated blocks (or unassigned block) in the current process will be activated in the next iteration. However, this *target* fully considers the status of reactivation of each sub-active list, so it can be used as a guideline for block distribution.

Locality-aware clustering

Our dynamic decomposition models generate many fragments in the process of distributing unassigned blocks to the sub-domains. For example, in Figure 6, the blocks highlighted in the green region will be assigned to a disjoint sub-domain for load balancing. However, they become a fragment and halo exchange will be required with the adjacent sub-domain after each updating. Too many of these fragments can increase the cost of halo transmission and cause an overall

Algorithm 3: MULTI-GPU FIM ALGORITHM WITH ADAPTIVE DOMAIN DECOMPOSITION

```
Input: Set of input X, Set of block V, active list L
N ← the number of total device
Split $L$ to sublists $L_i$ for all $i \in N$
/* Parallel section */
while any $L_i \neq \emptyset$ do
    /* step 0: On-the-fly domain decomposition */
    forall $b \in L_i$ do
        for each adjacent block $b_{nb}$ of $b$ do
            $j$ ← number of domain for $b_{nb}$
            if $j$ = unassigned then
                add $b_{nb}$ to $L^U$
    Barrier synchronization
    $total \leftarrow |L^U|$
    forall $i \in T$ do
        $expected_i \leftarrow \gamma * (R_i^{up} + R_i^{re}) * |L_i|$
        $total$ += $expected_i$
    $target \leftarrow total/N$
    forall $i \in T$ do
        $diff \leftarrow target - expected_i$
        from $L^U$, attach $diff$ blocks to sub-domain $i$
    Barrier synchronization
    /* from Algorithm 2 */
    /* step 1 ~ step 4 */
        ......
        ......
    $R_i^{up}$ and $R_i^{re}$ update
```

performance drop. To solve this problem, we consider the connectivity between the block and the sub-domain. Figure 7 is an example of clustering using connectivity. When it is decided to allocate unassigned blocks to the sub-domain by the dynamic domain decomposition algorithm, we can examine the connectivity between these blocks and the sub-domain. This is formed in every edge, and in the case of a three-dimensional grid, it has a total of six adjacent lines per block. For example, we try to assign *diff* blocks to sub-domain i in Algorithm 3. Here, we check all the connectivity between a block in the L^U and the sub-domain i. If the number of edge between a block and a sub-domain, then we consider the connectivity is higher. Then, sort the blocks in L^U by connectivity, then assign the blocks with the highest connectivity to the sub-domain i in first. Figure 7 describes that our method can be easily applied to prevent large number of fragments without complicated steps.

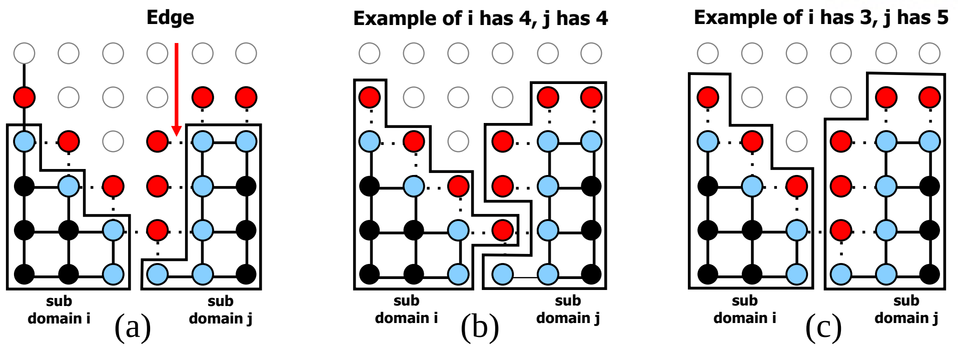


Figure 7: Simple description of clustering. In (a), there are two sub-domains and will assign red circles. At this time, each red circles has an edge with the sub-domain. (b) and (c) are each example of dynamic domain decomposition, respectively. If we decide to allocate n blocks to domain i, we choose n red blocks by sorting the number of adjacent line with domain i.

4.4 Implementation

We built our system using C++, OpenMP, and NVIDIA CUDA [68]. In the execution model of CUDA, one block of the MG-FIM corresponds to one CUDA block, and the nodes in the block are concurrently updated by CUDA threads. The BlockFIM algorithm takes n iterations when a block is included in an active list (see Algorithm 2, line 8). This is to take advantage of the features of the CUDA device which has multiple memory spaces. For example, global memory is large capacity but has relatively slow bandwidth while small size of shared memory has faster bandwidth. When the entire grid is stored in global memory, if we copy a block to shared memory and recycling it multiple times, then get greatly performance improvement. But too large n should be avoided, because it often require unnecessary computation even after the blocks converge. In this paper the value n is fixed to m when we use $m \times m \times m$ block. We also used the shared memory optimization when obtaining the converged status of the block by reducing the status of the node.

Our algorithm implementation is a composite of the GPU compute part and the CPU compute part. In the case of Algorithm 2, a main computation part such as distance solution of block in step 1 is operated on the GPU, but the management of list (e.g. active list) is performed on the CPU. Here, step 2 has a dependency on step 1 at the block level.Thus, we optimized this process by overlapping the GPU computation and the CPU operations using pipelining of multiple CUDA streams. With this, we can almost hide the computation time of the CPU.

OpenMP is used to efficiently use the multiple GPUs on a shared memory system. We created a number of CPU threads equal to the number of GPUs, and one thread was responsible for one GPU. Inter-GPU data communication is optimized using GPU peer-to-peer transfer in the shared memory system. To avoid the overhead caused by many system calls in the halo communication process, the transmission data are packed into one chunk of array and transmitted in a single transaction. To minimize the data transfer, only the adjacent 2D face

(8×8) between two 3D blocks $(8 \times 8 \times 8)$ is transferred (i.e., we do not transfer the entire blocks). At this time, optimization using multiple CUDA streams was also applied to halo communication. In the active list, there are two type of blocks, one is belonging to the halo area, and other are not. Only a block in the halo area participates in halo communication step. Among the block in active list, we calculate blocks in the halo area in first. After that, we can reduce the cost of halo communication by overlapping the data-transfer of the halo data and computation of the remaining block.

4.5 Result

In this section we evaluate the performance of the proposed adaptive domain decomposition method, and compare it with the commonly used regular domain methods for the multi-GPU FIM. We implement the prototype code using C++, NVIDIA CUDA 9.0, and OpenMP with the -O3 level optimization. We evaluate the performances on a computing server equipped with two Intel Xeon CPU E5-2640 v4 deca-core processors sharing 256GB of DDR4 memory and eight NVIDIA GTX 1080 Ti GPUs (Each has 3584 CUDA cores and 11GB device memory). Each GPU is connected to the PCIe 16x interface, with a data transfer bandwidth of 16GB/s, and inter-GPU communication is done via CUDA peer-to-peer(P2P) API to optimize bandwidth between GPU devices. The various initialization costs, such as the loading time of speed maps or the data copying time from host to GPU for input distance domains are independent from algorithm types, thus all running times were measured only for the main computation part.

In the implementation part, we introduced two optimization options; One is to hide the CPU cost using CUDA multiple streams and the other is to reduce the communication cost also using CUDA multiple streams. In this section, OPT1 is implemented without any optimization, OPT2 applied CPU cost hiding. OPT3 is our final model with both of CPU cost hiding and communication reducing. Except the special case (For example, Figure 12 is comparison of optimization of performance), all experiments were performed with the OPT3 level.

The grid dimension is $800 \times 800 \times 800$. All computations are conducted in double precision, and the block width is 8. The performance of the FIM algorithm is heavily depended on the complexity of the input speed function.Therefore, we tested the methods on the various types of speed maps, ranging from a plain constant speed map to a complex maze shape map. The speed maps used in our experiments are defined as follows (All speed maps are defined in the normalized domain $\Omega = [0, 1]$. Distance maps are shown in Figure 8):

Map 1: $f = 1$. Constant speed map.

Map 2: $f = 6 + 5\sin(2\pi x) * \sin(2\pi y) * \sin(2\pi z)$.

Map 3: $f = 1 + 0.5\sin(20\pi x) * \sin(20\pi y) * \sin(20\pi z)$.

Map 4: $f = 1/4$, $1/2$, 1. Three layers of different speeds

Map 5: Spatially coherent random speed map

Map 6: Circular maze speed map with permeable barriers. ($f = 0.01$ on barriers, otherwise 1)

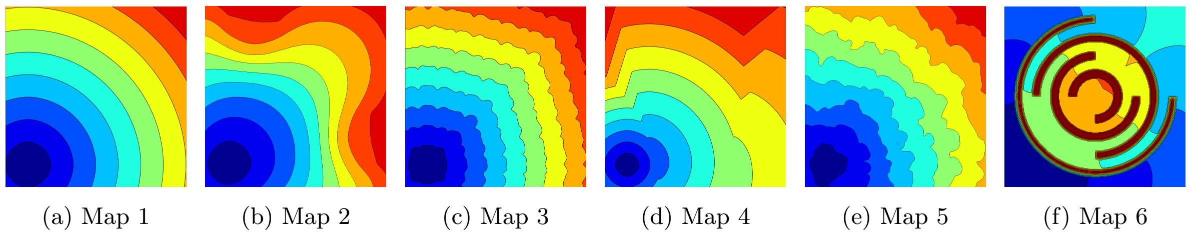


Figure 8: Color-coded distance map with iso-contours of our test datasets visualized in 2D. Blue to red color : from nearest to farthest distances to the seed point. A single source was used here.

We performed experiments in three types of boundary conditions to reflect different environments. Table 1 shows the running times of a single GPU on the given test maps. For corner case, a single seed is located at $(1/8, 1/8, 1/8)$ on the normalized domain $\Omega = [0,1] \times [0,1] \times [0,1]$. The intermediate column is the result of a seed in the center of the domain $(1/2\ ,\ 1/2\ ,\ 1/2)$. In Random 5, we put five sources which randomly arranged in domains. We used this result to demonstrate that our proposed model works well in a variety of environments, regardless of the location or number of sources. According to the result, Map 1 is the case of the simple speed map regardless of the source location. However, other speed map cases that require iterative calculations show different results. For example, in a corner source case, Map 3 requires the most number of computation; but in a center source case, Map 6 is considered as the most complex map.

We tested several regular decomposition methods and our on-the-fly adaptive decomposition method to measure parallelization performances for multi-GPU system. Each decomposition method generates the same number of sub-domains as the number of GPUs so that each sub-domain is assigned to a single GPU.

- *1-axis multi-split (1d-m)* splits the data uniformly only along z-direction (Figure 4 (a)).
- *3-axis single-split (3d-s)* splits the data along the different axis whenever the number of GPUs is doubled (Figure 4 (b)).
- *3-axis mingle-split (3d-m16* and *3d-m32)* splits the data multiple times along each axis to generate more sub-domains than the number of GPUs and assigns GPUs to sub-domains multiple times in a checkerboard fashion (Figure 4 (c)).

Here, *3d-m16* sets the size of the sub-domains to 16^3 and *3d-m32* sets it to 32^3; each the double and quadruple the size of block length 8.

All experiments were undertaken by changing the number of GPUs from one to eight for various domain decomposition methods. However, the case of 3d-s was measured only by two, four, or eight GPU cases depending on the feature of domain decomposition.

	Corner	Center	Random 5
Map1	4.38	4.38	6.31
Map2	15.34	14.29	12.54
Map3	24.83	16.69	18.83
Map4	16.95	6.84	14.27
Map5	18.35	14.47	14.98
Map6	21.98	25.34	18.86

Table 1: Table for single GPU execution time (sec).

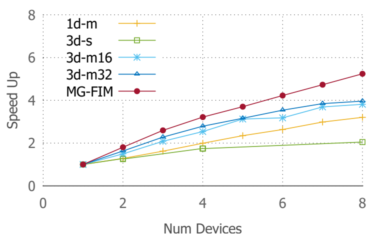
A single corner source

The first experiment is a single corner source example. The seed location is intentionally placed near the corner to test the worst-case of unbalanced task loads. Figure 9 lists the parallel speed up time of each domain decomposition method on different speed maps using a various number of GPUs. Among the regular decomposition methods, the models of 1d-m and 3d-s have a low performance for every speed map case; They achieved less than 4 times speedup even we used 8 GPUs.

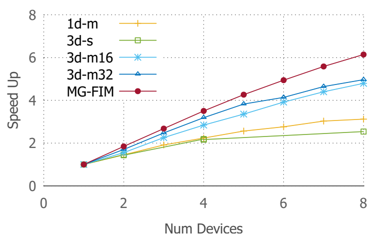
Among them, Map 1, which has a small amount of computation, show low efficiency in every decomposition case.In the case of 3d-m16 and 3d-m32, the parallel speed up on Map 1 is about four times on eight GPUs. However, on other maps with relatively high computational volumes, 3d-m32 achieved about five times speed up on eight GPUs. 3d-m16 showed a slightly lower performance compared to 3d-m32 in every case. Our adaptive model outperformed static domain decomposition methods on all the speed map cases. The parallel speed up is about 5.2 times in Map 1, and up to 6.6 times in Map 3. On average, our model achieved about six times speed up on eight GPUs. On map 6, though, our model showed only a slight performance improvement over 3d-m16 and 3d-m32. A detailed analysis will be given in the following section.

Performance analysis

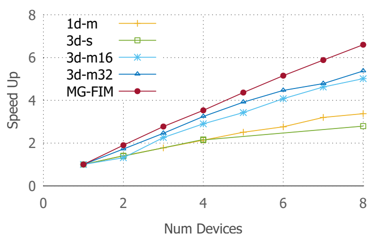
This subsection analyzes the results of a single corner source simulations in detail. There are many factors for the parallel performance drop, but there are two major issues: one is the waiting time of barrier synchronization between GPU devices because of task unbalancing, and the other is the communication time due to halo synchronization. In addition, we should consider the cost of the CPU operation. This CPU operation is mainly used to manage block indexing like active lists. Also the proposed adaptive decomposition is operated in the CPU. Figure 12 details these performance factors in each decomposition model when using eight GPUs. Among the various maps, we select two examples (Map 1 and 3); Map 1 has the smallest and Map 3 has the largest amount of computation. The cost of on-the-fly dynamic decomposition is aggregated in the overhead.



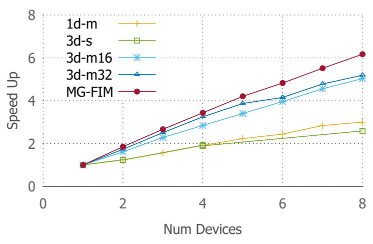
(a) Map 1



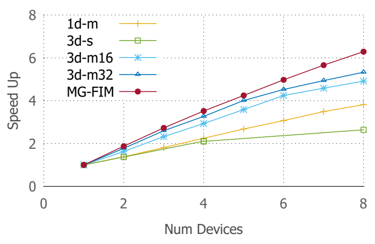
(b) Map 2



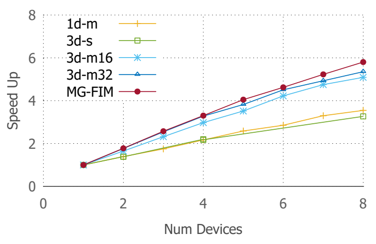
(c) Map 3



(d) Map 4



(e) Map 5



(f) Map 6

Figure 9: Parallel speed up using different number of GPU device (1 to 8) measured in second. A single source which located on corner side.

In both maps, 1d-m and 3d-s have significant performance degradation because of the waiting time of barrier synchronization. In the case of 3d-m16, the cost of waiting is suppressed considerably, but the communication cost becomes the major bottleneck of performance deterioration. In particular, this communication cost cannot be hidden even at OPT3. However, the 3d-m32 can hide about half of the communication cost. So even if 3d-m32 has more performance loss in terms of waiting time compared to 3d-m16, but it finally gives better performance.

Our model has a short wait time, and it hides the communication costs remarkably well in OPT3. The overhead due to dynamic decomposition was 100 to 150 ms in average. This is about 10 percent of the total cost on Map 1 which is most simple case, but it is rarely visible on complex speed map case like Map 3 (less than 4 percent).

To discuss Figure 12 in more detail, we have prepared Figures 10 and 11 each visualize the task load balancing for GPUs. The x-axis is the iteration number, and the y-axis is the task size. If curves are similar, tasks are equally distributed across GPUs; otherwise there is a load imbalancing between GPUs. We used different colors for the eight GPUs.

Figures 10 is the visualization of GPU tasks for Map 3. It shows the models of 1d-m and 3d-s can't handle the pair task distribution between GPUs at all. Therefore, these models will have many waiting times. 3d-m16 has better load balancing than 3d-m32 because it decomposes the domain more tightly. The results also show that our model works near optimally.

We prepared Figure 11 which is result of for Map 6. This map requires about 1150 iterations and it shows a similar pattern to Figure 10, until the iteration 700. However, after 700 iterations, our model doesn't work effectively. This map is an example that belongs to a circular maze. Total computation continues an additional 450 iterations even after the on-the-fly adaptive

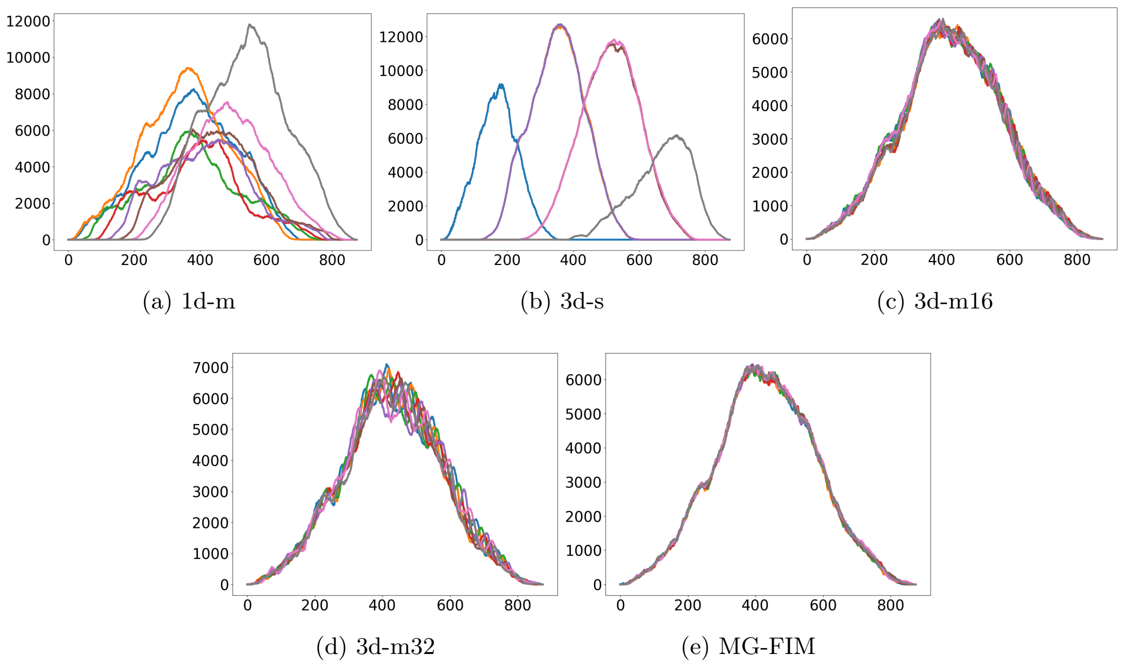


Figure 10: Visualization of GPU tasks in each decomposition model on Map 3 (8 GPUs). The x-axis is the change of the iteration, and the y-axis is the number of tasks.

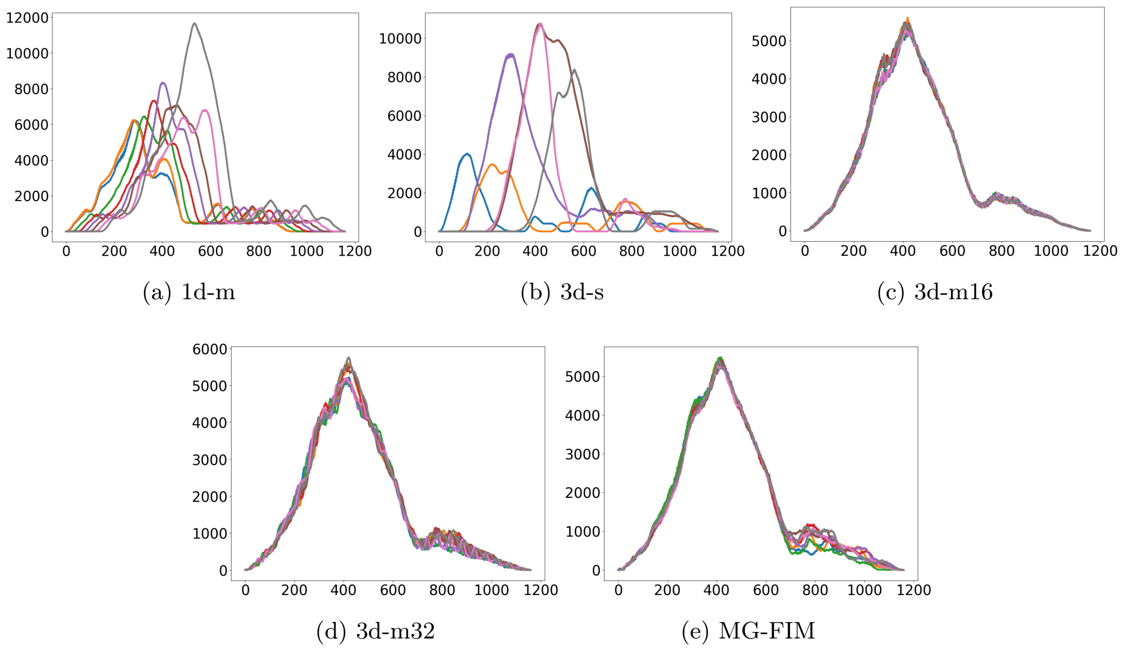
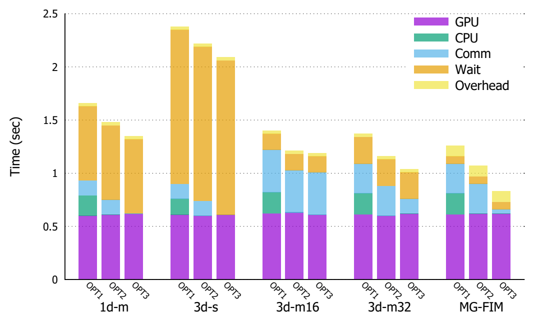
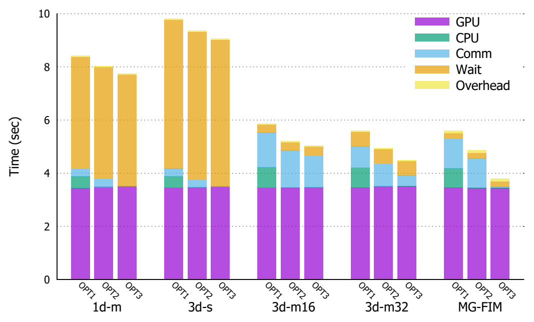


Figure 11: Visualization of GPU tasks in each decomposition model on Map 6 (8 GPUs). The x-axis is the change of the iteration, and the y-axis is the number of tasks.



(a) Map 1



(b) Map 3

Figure 12: Detail time analysis of Map1 and Map3 when using 8 GPUs. OPT1 did not apply any optimization, and OPT3 applied all optimizations.

decomposition is completed in 700 iteration. The core idea of our model is to perform load balancing between GPU devices by an adaptive dynamic decomposition. Thus, we do not take any action after finishing all the dynamic decomposition. Therefore, this is a limitation of our algorithm, and it shows why our model has relatively small performance improvement for Map 6 in Figure 9. Nonetheless, our adaptive model is still superior to any regular domain decomposition model.

Figure 13 further explains result in Figure 12 in terms of communication cost. This shows, how many blocks in the active list participate in the communication for each iteration on average. For optimizing the data transfer costs which used in OPT3, it is important how many blocks are used for communication. The method of OPT3 is hiding the communication cost of halo transmission by overlapping data transfer and the computation of the blocks which not included in halo region. Therefore, if there are many blocks in halo region, the overlapping effect is reduced. 1d-m and 3d-s have very low communication ratios, meanwhile 3d-m16 and 3d-m32 have very high communication ratios because of their tight decomposition. In particular, 3d-m16 can hardly hide the communication cost by computation and data-transfer overlapping because its transmission ratio is near to one in both maps. In addition, 3d-m32 also has a limitation on overlapping due to its high transmission ratio. The communication rate of our proposed model is higher than 1d-m and 3d-s, but smaller than 3d-m32. Because of the relatively low communication rate, our model can hide the transfer cost successfully in OPT3 implementation.

Different source location and block size

This subsection deals with three additional experiments with the same speed map, but they use different source locations and different block sizes. First, Figure 14 is the result of a seed in the center of the domain. The 3d-s decomposition model, which split the domain along its axis, is suitable for this experiment. For Map 1, which required no additional re-activation of the block, 3d-s is the theoretical best decomposition model. Except for Map2, 3d-s works as a relatively good regular decomposition model. However, 1d-m is still the worst model for parallel

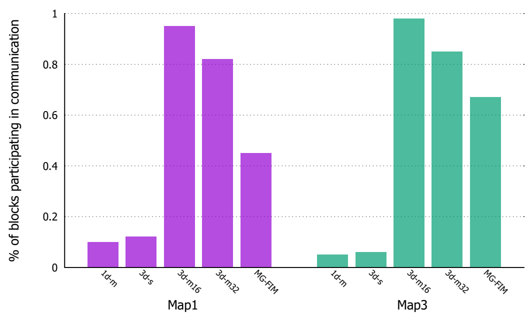


Figure 13: Ratio of the block which participating in communication in active list using 8 GPUs. The ratio is averaged along all iteration.

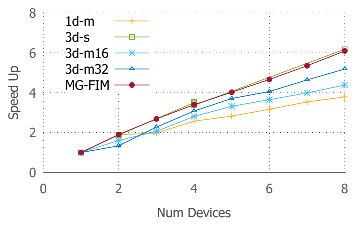
implementation overall. Our model achieved overall good parallel performance. And Figure 15, which use five randomly arranged sources has similar results. These results demonstrate that our proposed model works well in a variety of environments, regardless of the location or number of sources.

Figure 16 is the result of using a single corner source, which is the same as Figure 9. However, it used 16^3 blocks instead of 8^3 blocks, which were used in the other experiments. When the size of a block increases (as a computing unit), more computation is required until one block converges, which leads to an increase in execution time. However, the large length of the blocks takes a long time to converge, whereas the cost of communication is reduced. Thus, in the 16^3 block case, the parallel performance overhead due to the communication cost is relatively small. So this experiment gives more advantageous to 3d-16 and 3d-32 decomposition models. But still our model shows the better performance than other regular decomposition models. Therefore, our proposed model works well in a variety of environments, regardless of the type of sources or block sizes.

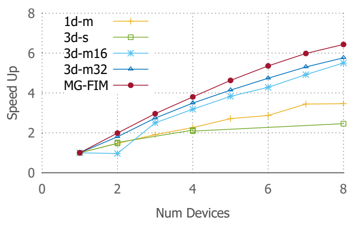
Compare with 3DPMM

In Figure 17, we compare our result with other works concerning the multi-GPU eikonal solver. To the best of our knowledge, the multi-GPU parallel 3d sweeping(3DPMM) by Krishnasamy *et al.* [34] is the only similar work to ours as of today. In the 3DPMM, parallelism lies on a plane. Therefore, we divide the plane as several numbers of rectangles by 1d-m domain decomposition, and the GPUs solve each rectangle in parallel. In this experiment, we used the 640^3 domain size instead of the 800^3 for easier implementation for the 3DPMM algorithm, and we used a single corner example.

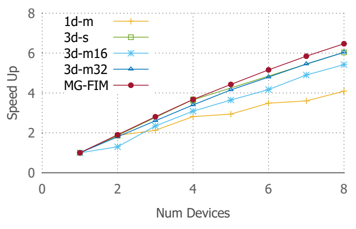
The 3DPMM achieved a parallel speed up of 2.95$\times$ on four GPUs and 4.59$\times$ on eight GPUs. The performance of 3DPMM we tested is little better than the results of the original works, up to 2.86 times the speed up on four GPUs. However, the scaling performance of our methods is better than the 3DPMM - up to 3.4$\times$ on four GPUs and 6.1$\times$ on eight GPUs. This is mainly because the 3DPMM algorithm requires data shuffling between each sweep iteration (For example, after



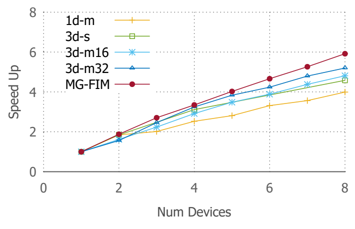
(a) Map 1



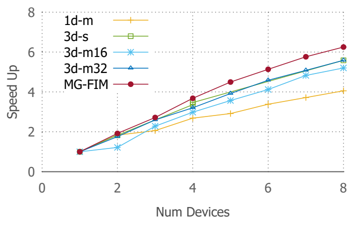
(b) Map 2



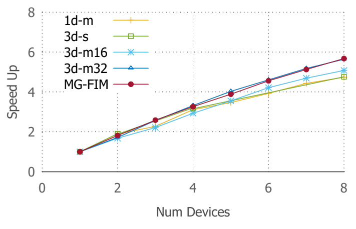
(c) Map 3



(d) Map 4



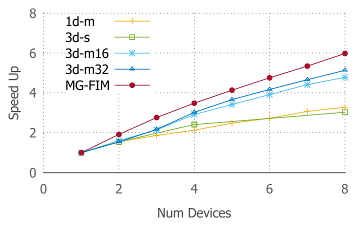
(e) Map 5



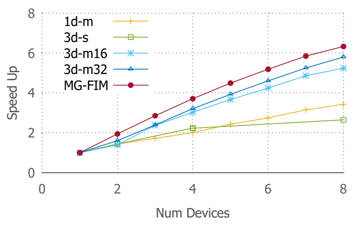
(f) Map 6

Figure 14: Parallel speed up using different number of GPU device (1 to 8) measured in second. A single source which located on center

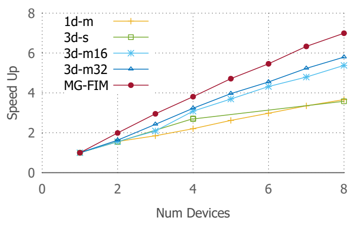
.



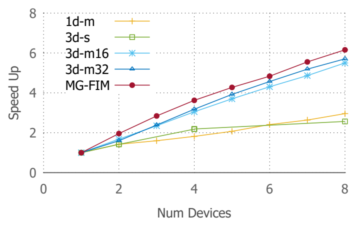
(a) Map 1



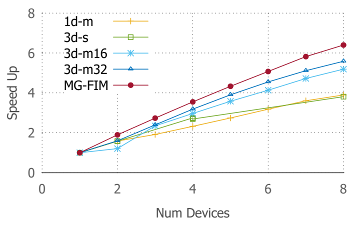
(b) Map 2



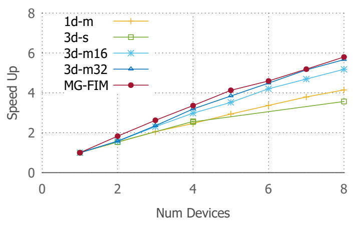
(c) Map 3



(d) Map 4



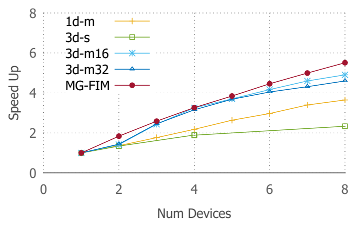
(e) Map 5



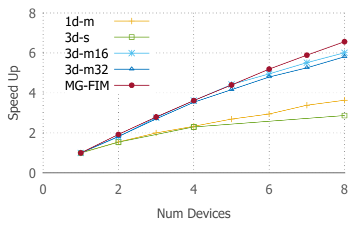
(f) Map 6

Figure 15: Parallel speed up using different number of GPU device (1 to 8) measured in second. Multiple source points are distributed in five places

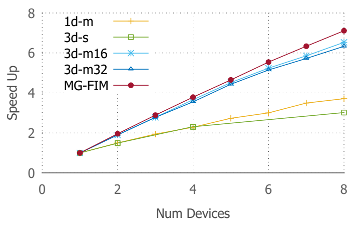
.



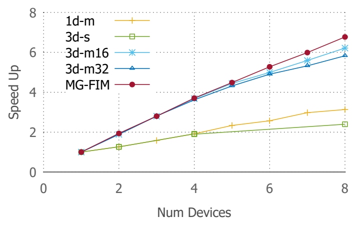
(a) Map 1



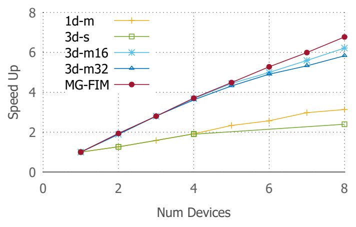
(b) Map 2



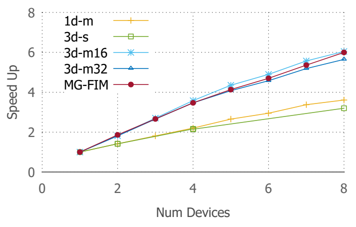
(c) Map 3



(d) Map 4

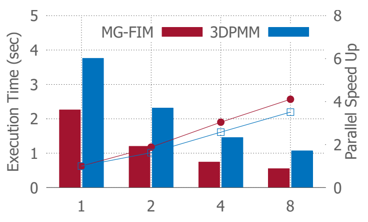


(e) Map 5

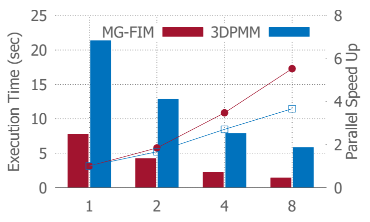


(f) Map 6

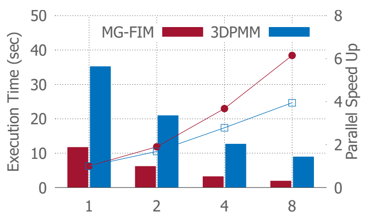
Figure 16: Experiment using diffent size of block (16^3 instead of 8^3). Parallel speed up using different number of GPU device (1 to 8) measured in second. A single source which located on corner side. Running time using different number of GPU device (1 to 8) measured in second.



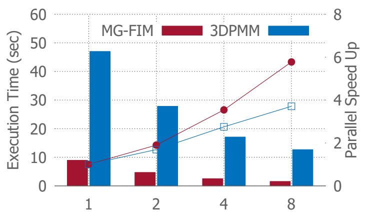
(a) Map 1



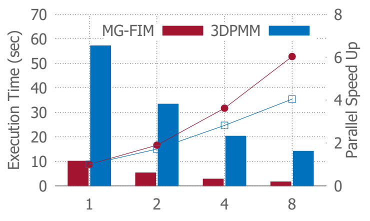
(b) Map 2



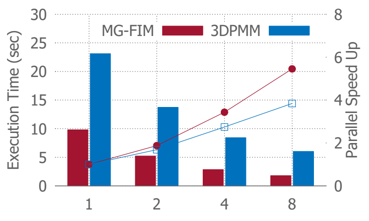
(c) Map 3



(d) Map 4



(e) Map 5



(f) Map 6

Figure 17: Comparison of our adaptive model and Parallel 3D Sweeping method (3DPMM) for various input volume. The bar graph is the execution time (y-left), and the line graph is the parallel speedup (y-right).

sweeping along the x-axis, the entire data should be shuffled in order to proceed to sweep along the y-axis in the next iteration). This data shuffle has a large overhead because it is performed on all domain blocks. Our system not only shows better results in a single GPU results, but also higher efficiency in multiGPU parallelization.

Clustering effect

The final result is the effect of locality-aware clustering. The following results were derived from a single corner experiment. Table 2 shows the changes in the number of communication per block either by using clustering or not for our adaptive domain decomposition model. When a block sends a halo to n sub-domains after the operation, we consider it to have performed n times communications. If we do not use the clustering algorithm, our model requires about 2 times additional communication. This is because there are too many fragments between sub-domains. Our proposal model focused on reducing time in two ways: waiting time due to unbalanced task loads, and communication costs. However, if we do not apply locality-aware clustering, the effect of reducing communication costs will be halved, and the overall performance of our methods will decline.

		Map1	Map2	Map3	Map4	Map5	Map6
adaptive	normal	5.27	21.37	34.53	25.30	24.53	33.50
	clustering	2.91	9.77	16.71	14.36	12.13	16.96

Table 2: Result of locality-aware clustering. The number of communication computation per block. When a block sends a halo to n sub-domains after the operation, we considered to it performed n times communications.

Discussion of Limitation

Our model have good overall performance but have some limitations. The first limitation is shown from the result of Map 6. Our methods are not affected in the further iterative computation, in which all blocks are assigned to a sub-domain. Because it is an on-the-fly algorithm, it has some overhead compared with static domain decomposition. In our experiment, when the computation was small such as Map 1, there was about 10 percent of the overhead time on for the eight GPU cases.

4.6 Summary

In this section, we proposed a novel adaptive domain decomposition method for blockFIM based on a history-based active list prediction method. The proposed model successfully predict the future computing domain and distribute tasks evenly across GPUs. In addition, we also proposed several optimization methods for multiGPU implementation and a locality-aware clustering algorithm to minimize inter-GPU data communication. The experiment results show that the proposed method achieved greater performance improvements in most cases when compared to regular domain decomposition methods. In particular, our model showed the parallel speed up overall 6.2 times and maximum up to 6.6 times on eight GPUs. In the future, we will plan to extend our work to distributed systems. Experiments on extra-large data such as out-of-core are also future research areas. We would like to explore how our results can be applied in real-world applications.

V A Group-Ordered Fast Iterative Method for eikonal Equations

5.1 Introduction

The FIM is an iterative algorithm that adaptively updates the solutions that are currently affected by the wavefront, called *active list*, until they converge. FIM is an inherently parallel algorithm because all nodes belong to the active list can be updated concurrently. However, in the original FIM paper, the authors only introduced the main algorithm and its extension to SIMD parallel architecture, such as the GPU (Graphics Processing Unit). Even though the original FIM algorithm embraces a potential to be applied to any parallel computing systems other than the GPU, it has not been fully addressed yet in elsewhere. In addition, because the main design choice for the FIM algorithm mainly focused on increasing parallelism rather than algorithmic optimality, its worst-case performance may vary depending on the complexity of the input speed function. In this section, we address these issues by proposing a new parallel algorithm that improves the performance on highly-complicated speed functions as well as the efficiency on shared memory systems.

The main contributions of this section are several-fold. First, we propose an efficient parallel implementation of FIM for *multicore shared memory systems*. Even though the original FIM algorithm is inherently parallel, a naive parallelization does not guarantee sufficient performance benefits. We propose a local queue based parallelization approach that can avoid expensive lock synchronization while ensuring good load balancing between threads. Due to the lock-free nature of the method, the parallelization overhead is minimized and the proposed method scales well. Second, we further improve our parallel FIM algorithm by proposing a novel group-based updating scheme where the updating order is determined by the solution on a coarse level grid, called *GO-FIM*. This approach can effectively eliminate the drawback of the original FIM without maintaining an expensive global ordered data structure, such as Heap, while providing superior parallel performance by clustering similar blocks for concurrent update. Last, we show an in-depth analysis of the performance of both methods on various test datasets and compare them with state-of-the-art eikonal solvers on multi-core CPUs, and finally show how GO-FIM effectively improves FIM on the GPU.

5.2 Lock-free Parallel FIM

Fast Iterative Method

As shown in Algorithm 1, FIM iteratively updates the solution of the nodes in the active list L until the list becomes empty. FIM is an iterative method – meaning that each node can be updated multiple times. A node can be removed from the active list only when it is converged (otherwise, it remains in the list and is updated again in the following iteration), which is the main difference from conventional label-correcting algorithms that use a FIFO queue to remove the top node immediately. A converged node activates its non-converged adjacent nodes, and any converged node can be reactivated later even though it is inactivated previously.

In FIM, there is no assumption on the updating order of nodes, which allows a straightforward parallelization of the algorithm by splitting the for loop into multiple disjoint sub-loops (line 8 in Algorithm 1) and processing them concurrently using parallel threads, i.e., using OpenMP `parallel for` clause. However, some operations in the algorithm may cause race conditions, such as updating the solution (i.e., $U(\mathbf{x}_{nb}) \leftarrow q$) and adding $\mathbf{x}_{nb}$ to L in `if` $\sim$ `else` block in line (15) in Algorithm 1, because the grid and the active list are shared among different threads and multiple threads may attempt to access them at the same time. A simple solution to avoid this race condition is using a mutex (e.g., lock) to allow only one thread to access shared memory location and active list at any given time. However, lock synchronization is an expensive operation, especially for active list access, and such a naive parallel implementation using locks causes too much overhead, which will result in poor scaling performance for a large number of threads.

Lock-free Parallel Implementation of FIM

In order to improve the scalability of the method, one can use a temporary local buffer per thread to store new active nodes. The main idea is that since there is no race condition when performing a read access from the active list, we can use a global active list for parallelization but manage a local buffer to collect new active nodes for the next iteration to avoid race condition for a write access to the active list. By doing so, lock synchronization for active list can be effectively reduced, but there still exists an overhead to combine multiple temporary local buffers into a global active list per each iteration, which requires lock synchronization. Therefore, in order to completely remove lock synchronization in list management, we propose a local *and* lock-free parallel implementation of FIM (Algorithm 4).

The proposed lock-free parallel implementation of FIM is using local active list only. Each thread i owns its local active list L_i, and each list is processed simultaneously with other lists. Therefore, read and write access to the list can be performed independently without introducing a race condition and we can completely remove lock synchronization. However, even though each local active list contains equal number of active nodes at the beginning, the size of each list will change over time because each local active list may propagate differently. Therefore, a load-balancing step is required in each outer iteration. In order to reduce the overhead introduced by the load-balancing step, we employ a simple parallel pairwise load-balancing method – every two lists are randomly chosen as a pair, and the size of the lists in each pair is equalized (Algorithm 4 line 14). In our implementation, we randomly generate an odd number (i.e., offset), and add this number to each odd-indexed thread to access an even-indexed list. Since all odd indices are shifted by the same offset, there is no two odd-indexed threads access the same even-indexed lists. In addition, since we select the offset randomly, all even lists will be roughly equally paired with odd threads eventually. In addition, this load-balancing algorithm can be easily parallelized because each thread can access its pair list independently.

Algorithm 4: Lock-free Parallel FIM

```
Input: Grid $\Omega$, Solution $U$, Active list $L$
1  $n \leftarrow$ number of threads
2  $N = \{0, 1, 2, ..., n-1\}$
3  Initialize $U$ and $L$ as line 1 to 6 in Algorithm 1
4  Split $L$ into disjoint sublists $L_i$ for all $i \in N$ so that $L = \cup_{i \in N} L_i$
5  $F$ = flag array, initialized 0
6  foreach $i \in N$ do in parallel
7      forall $x \in L_i$ do
8          $F(x) \leftarrow 1$
9  while $L_i$ is not empty for some $i \in N$ do
       /* Load balancing */
10     offset $\leftarrow$ a randomly selected odd number
11     foreach $i \in N$ do in parallel
12         if $i$ is odd number then
13             $j \leftarrow (i + \text{offset})\%n$
14             Make the size of $L_i$ and $L_j$ equal by stealing nodes from the larger list
15     Barrier synchronization
16     foreach $i \in N$ do in parallel
17         forall $x \in L_i$ do
18             $p \leftarrow U(x)$
19             $q \leftarrow$ solution of $g(x) = 0$
               /* If not converged */
20             if $p > q$ then
21                 $U(x) \leftarrow q$
22             else
23                 forall $x_{nb}$ adjacent to $x$ do
24                     if $U(x_{nb}) > U(x)$ and $F(x_{nb}) == 0$ then
25                         $p \leftarrow U(x_{nb})$
26                         $q \leftarrow$ solution of $g(x_{nb}) = 0$
27                         if $p > q$ then
28                             $U(x_{nb}) \leftarrow q$
29                             if $F(x_{nb}) == 0$ then
30                                 $F(x_{nb}) \leftarrow 1$
31                                 add $x_{nb}$ to $L_i$
32                 Remove $x$ from $L_i$
33                 $F(x) \leftarrow 0$
34     Barrier synchronization
```

Even though we removed expensive lock synchronization by using multiple local active lists, there is a small chance that the same node is accidentally inserted into more than one list at the same time. This happens when a newly activated node is adjacent to multiple converged active nodes that are stored in different active lists. This can be avoided by checking whether a node is currently in *any* active list (Line (24) and (29) in Algorithm 4). We can use a flag per each node to check this, but a special care needs to be taken when implementing this flag-based testing for multiple threads, especially for writing operations. The safest way is using a lock so that only one thread can update or access a flag, but this will violate lock-free implementation. To resolve this issue, we used a *fetch-and-modify* atomic operator (shown in Listing 1) to check the flag in Line (29) so that a node is inserted to one active list only as shown below. By doing this, multiple threads can check the flag but only one of them is allowed to insert a node to its active list and the other threads will simply pass this code block. The performance of lock-free FIM implementation is given in Table 4 (the rows for FIM).

Listing 1: Lock-free code using a fetch-and-modify atomic operator

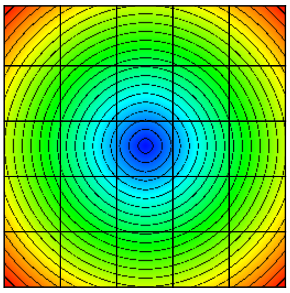
```
// idx : node index
// F[idx] : true if idx is in the list, false otherwise

if(__sync_lock_test_and_set(&(F[idx]), 1) == 0)
{
    // insert idx into active list
    . . . . .
}
```

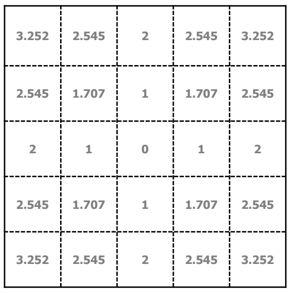
5.3 Group-Ordered FIM

The lock-free parallel FIM introduced in the previous section may reduce the running time of the solver by using multiple threads, but it does not reduce the actual number of computations. This is because the previous parallel implementation focuses only on how to split the task for parallel processing and does not pay attention to how to reduce the total computational cost. In this subsection, we propose a novel numerical algorithm that reduces the computational cost that can be parallelized efficiently as well.

The main drawback of the original FIM [19] is that the propagation of the active list does not conform to the actual distance to the seed points. This is due to the missing ordered data structure – for example, Fast Marching Method (FMM [20]) employs Heap data structure to sort the active nodes based on the distance (i.e., solution), and updates them in the correct order to maintain the causality of the solution, i.e., smaller solutions are computed before larger ones. However, FIM does not use any ordered data structure, but allows multiple updates of the node until it converges completely. In algorithmic point of view, the former, FMM, can be classified as a *Label-Setting* method and the latter, FIM, can be classified as a *Label-Correcting* method. Therefore, in FMM, once the label (i.e., solution) is set, there is no need to re-set the label.

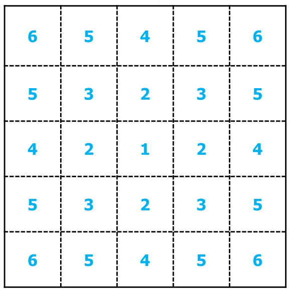


(a)



3.252	2.545	2	2.545	3.252
2.545	1.707	1	1.707	2.545
2	1	0	1	2
2.545	1.707	1	1.707	2.545
3.252	2.545	2	2.545	3.252

(b)



6	5	4	5	6
5	3	2	3	5
4	2	1	2	4
5	3	2	3	5
6	5	4	5	6

(c)

Figure 18: Example of computing the block updating order. (a) Color map of the distance from the seed point (center) on the initial grid (speed map is constant) overlaid by the coarse grid blocks, (b) Distance value per block, and (c) Integer updating order of each block.

However, in FIM, even though a label has been set once, it can be corrected with a new label later. In terms of complexity, label-setting algorithms are worst-case optimal, e.g., $O(n\text{log}n)$ for FMM, and label-correcting algorithms are not worst-case optimal, e.g., $O(kn)$ for FIM, but the actual performance highly depends on the input. The main motivation behind the original FIM is abandoning the ordered data structure that hinders parallelization with an observation that k is usually small unless the input speed function is extremely complicated. Therefore, even though FIM's total number of computations could be higher than FMM, the actual running time could be much shorter due to reducing the significant overhead of managing the ordered data structure and using multiple threads for parallel processing. In this subsection, we propose a novel variant of FIM algorithm, called *Group-Ordered FIM (GO-FIM)*, that can handle the datasets with higher k values as well.

Main Algorithm

The core idea behind GO-FIM algorithm is that we can estimate a rough node dependency that guides the updating sequence without ordered data structures. Specifically, we compute the distance map on a coarser grid, which is later used as a causality map between blocks, and we update the nodes based on this order. For example, assume that the input grid size is n $\times$ n, then we decompose the input grid into $\frac{n}{m} \times \frac{n}{m}$ grid of blocks where each block is of size m $\times$ m (Figure 18 (a), if the input grid size is 40 $\times$ 40 and the block size is 8 $\times$ 8, then the coarse grid size is 5 $\times$ 5). Once we have a coarse grid of blocks, we need to assign a speed value on each coarse grid node by computing the average speed of the corresponding block on the original grid. There could be different approaches to assign speed values on coarse grid nodes, such as using maximum, minimum, or median speed values, but the average speed worked reasonably well throughout our experiments.

When the speed value on each node of the coarse grid is assigned, we run FIM on the coarse grid to compute the distance. Since the coarse grid size is small, execution of FIM on the coarse grid can finish very quickly. Once we finish computing the distance (i.e., solution of eikonal equation) on each coarse grid node, then we reassign the computed distance to the blocks (Figure 18 (b)) and sort them in ascending order. Once a sorted block list is created, we can assign the positive integer order number to each node (Figure 18 (c)) by clustering blocks having a similar distance value. By doing this, we can group multiple blocks that can be updated together during iterations. For example, in the first iteration, all the grid nodes that belong to the block of order number one will become a valid region to run FIM (the other regions will be marked as invalid and FIM will not propagate into that regions). In the second iteration, all the grid nodes that belong to the block of order number one and two will become a valid region. We continue this process until the entire grid becomes a valid region. By doing this, a group of nodes can be processed along the specific update order – therefore, we named the algorithm *Group-Ordered* FIM. Algorithm 5 shows each step of GO-FIM algorithm in pseudocode.

Algorithm 5: GROUP-ORDERED FIM

```
   Input: Grid $\Omega$, Solution $U$, Active list $L$
   /* Coarse grid level */
1  Generate and initialize coarse grid $\widetilde{\Omega}$ from $\Omega$
2  Run FIM on $\widetilde{\Omega}$ to assign distance per block
3  Cluster blocks in $\widetilde{\Omega}$ into $k$ groups ($G_1$ to $G_k$)
   /* Fine grid level */
4  Initialize $U$ and $L$ as line 1 to 6 in Algorithm 1
5  $G = \emptyset$
6  $n \leftarrow$ number of threads
7  $N = \{0, 1, 2, ..., n-1\}$
8  for $v = 1$ to $k$ do
9      $G = G \cup G_v$
10     if $v$ is 1 then
11         $L_{upper} \leftarrow L$
12     else
13         Get $L_{upper}$ from $G$ and $G_v$ (see Equation 4)
14     Split $L_{upper}$ into disjoint sublists $L_i$ for all $i \in N$
15     Barrier Synchronization
16     foreach $i \in N$ do in parallel
17         Get tight active list $\tilde{L}_{tight}$ from $L_i$ (see Algorithm 6)
18         $L_i \leftarrow \tilde{L}_{tight}$
19         while $L_j$ is not empty for some $j \in N$ do
20             Load balancing as line 10 to 14 in Algorithm 4
21             Barrier Synchronization
22             Solve for $L_i$
23             Barrier Synchronization
```

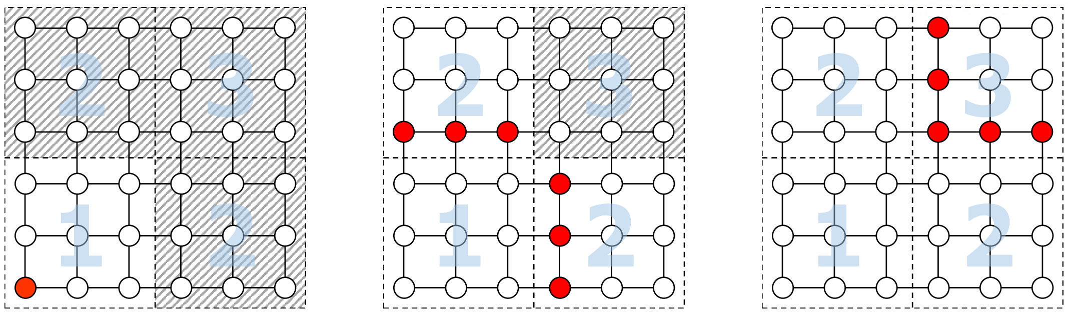


Figure 19: Example of initial active lists for three update groups

Tight Bound for Active List

As briefly explained above, GO-FIM employs a group update scheme – multiple blocks are grouped based on the update order index (1 to k), and the corresponding group is appended to the computational domain per each update pass, i.e., the domain is progressively expanding when the algorithm converges on the current domain. In order to do this, we need to find a proper initial active list for each update group. Let us define G_v is the group of blocks to be newly activated at the v-th update pass and G is the union of groups to be updated together at the v-th update pass, i.e., G_1, G_2, ... to G_v. Then we can define the upper bound (i.e., loose bound) of the initial active list L containing active nodes for the v-th update pass as follows:

$$L_{upper} = \{\mathbf{x} | \mathbf{x} \in B \subset G_v \text{ and } \exists\, \mathbf{x}_{nb} \in B' \subset G \setminus G_v\} \tag{4}$$

where $\mathbf{x}$ is a grid node, $\mathbf{x}_{nb}$ is a neighbor node *adjacent* to $\mathbf{x}$, B and B' are blocks, and $\setminus$ is the set difference operator. Based on this definition, Figure 19 shows an example of three update passes and corresponding initial active list for each pass (drawn in red color). In the first pass (Figure 19 (a)), only a single block (marked with number 1) belongs to the group G and the bottom-left corner node (drawn in red) is the active node of that group because it is the seed point (G=G_1). In the second pass, two additional blocks (marked with number 2), which form the group G_2, are newly activated and added to the group G, and the boundary nodes between the blocks belong to the previous group and newly added blocks (i.e., boundary between $G \setminus G_2$ and G_2) form an active list (Figure 19 (b) red points). Note that even though G_1 is already processed in the previous update pass, it must be included in the next update group G because FIM is a label-correcting method and active nodes can propagate in any direction. You can consider this as the computational domain is gradually expanding as iteration goes. In the same manner, (c) shows the third update pass. G is the union of G_1 , G_2 and G_3 where G_3 is the newly activated group, and the initial active list is a collection of boundary nodes in G_3 adjacent to blocks in $G \setminus G_3$, which is G_1 and G_2 (in this example, G_1 is not adjacent to G_3 but it could be possible in a different setup).

Note that L_{uppper} defined above is simply a collection of all nodes in G_v adjacent to the group $G \setminus G_v = G_1 \cup G_2 ... \cup G_{v-1}$. This implies that there might be unnecessary nodes in L_{uppper}, i.e.,

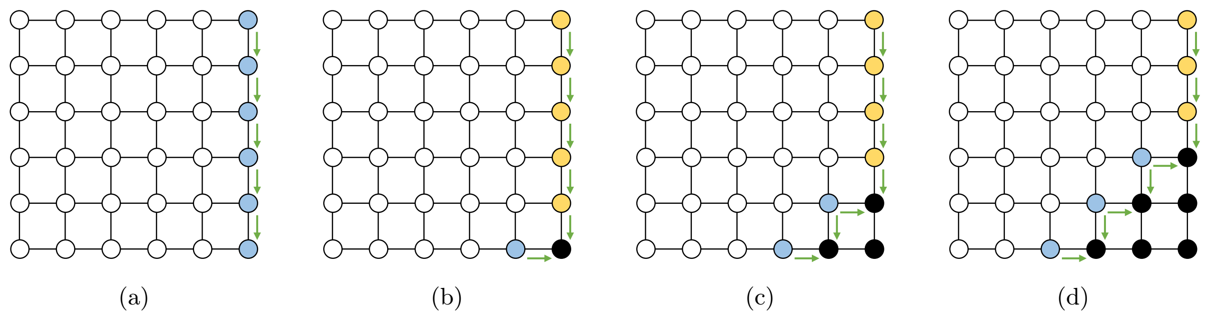


Figure 20: Example of an initial active list defined using an upper bound given in Equation 4. Green arrows represent causal dependency between nodes. Blue points are active nodes updated only once. Yellow points are active nodes unnecessarily updated multiple times due to a non-tight bound. Black points are converged nodes.

nodes that are not true upwind nodes in the group G_v. Therefore, we can define a more tightly bounded initial active list L required for the v-th update step as follows:

$$\begin{aligned} L_{tight} = \{\mathbf{x} | \mathbf{x} \in B \subset G_v \text{ and } \forall \text{ its upwind neighbor} \\ \text{nodes } \mathbf{x}_{nb} \text{ belong to } G \setminus G_v\} \end{aligned} \tag{5}$$

The main idea behind this tight bound is that L_{uppper} may have self-dependency – if we build a directed acyclic graph (DAG) based on the causal relationship between nodes, then some of nodes in L_{uppper} may depend on the other active nodes in the list, which is not the true initial active nodes because those can be activated later by the other *true* initial active nodes.

Figure 20 shows an example of an initial active list based on the upper bound given in Equation 4 on a constant speed map. Green arrows represents causal dependency between nodes, blue points are active nodes that are updated only once, yellow points are active nodes that are updated multiple times, and black points are converged nodes. In this example, an upper bound is used to collect initial active nodes, which are blue points in Figure 20 (a). Since there is causal dependency between the bottom right node and the others, yellow points do not converge after single iteration (only the very bottom node converges after first iteration but the other nodes stay in the active list marked as yellow). That means, the top-right corner node requires six iterations to get the correct solution and first five iterations are extra computations due to a non-tight bound. On the other hand, Figure 21 shows an example of tight bound where only the bottom-right corner node is included in the initial active list. As the figure shows, there is no redundant computation (yellow nodes) as shown in Figure 20.

Even though we theoretically defined a tight bound for the initial active list, it is not easy to derive such a tight list because we must know causal dependency (i.e., upwind neighbors) in advance in order to test whether a node belongs to a tight bound or not, as shown in Equation 5. The difficulty arises from the fact that causal dependency can be resolved only when the solution is already computed, but we need a tight bound to collect initial active nodes to compute

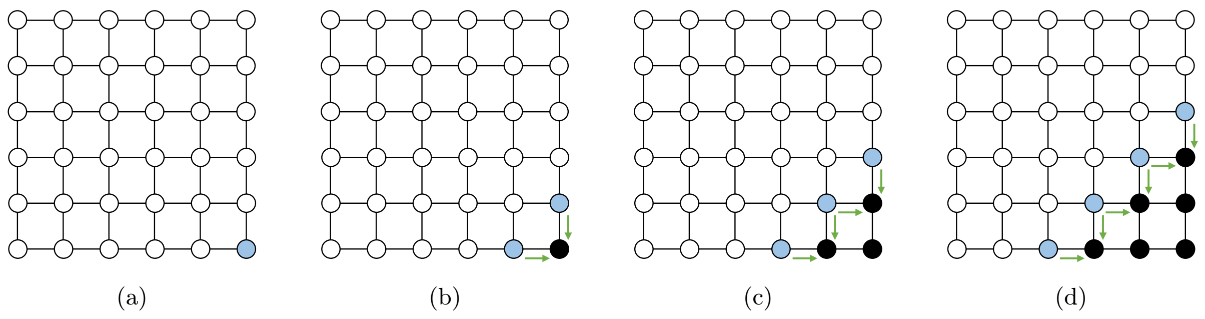


Figure 21: Example of an initial active list defined using a tight bound given in Equation 5. Due to the tightness of the bound, only a single active node is included in the initial active list. There is no unnecessary update in this example.

solutions – a chicken and egg problem.

To solve this problem, we propose a heuristic algorithm to approximately derive a tight bounded active list $\tilde{L}_{tight}$ (see Algorithm 6). The main idea is that even though we cannot define a tight bound without true solutions, we can define a loosely bounded active list using the upper bound given in Equation 4. Even though this is not a tightly bounded list, it is still a valid active list. Therefore, we start from a loosely bounded active list, and extract a more tightly-bounded active list from it. The algorithm is as follows: For a given initial active list, we run FIM update twice using a Jacobi update, and check for convergence. If a node belongs to a tight bound, it must converge after two FIM update iterations. Otherwise, the node must stay in the active list for further computation. Note that this algorithm does not guarantee the tightest initial active list because we cannot determine the complete causal dependency without having the correct solution on the entire domain. However, this simple algorithm works well in practice and can effectively reduce unnecessary computations. As shown in Table 3, there is a significant performance improvement by using a tightly bounded active list, roughly up to 40% of computation is reduced.

Block Clustering

Another problem we need to consider is how to cluster blocks into disjoint groups. In Figure 18, clustering seems relatively easy because blocks having the same distance value are clustered as a single group. However, in real world examples, distribution of distance is nearly uniform, and the difference of distance between adjacent blocks may become small after sorting. Therefore, it might be difficult to draw a clear cut to separate blocks into disjoint groups.

In addition, the total number of groups also affects the performance of the solver. The main (outer) iteration of GO-FIM depends on the grouping strategy because the total number of iterations is equal to the number of groups (in each iteration, the computational domain is expanding by adding the next group to the current domain). If there are too many groups, then it will increase the loop execution overhead as well as extra computation of initial active list

Algorithm 6: COMPUTE TIGHT BOUND

```
  Input: $L_{upper}$, $\tilde{L}_{tight}$
  /* Update distance of $L_{upper}$ using a Jacobi update */
1 foreach $\mathbf{x} \in L_{upper}$ do in parallel
2     $p \leftarrow U(\mathbf{x})$
3     $q \leftarrow$ solution of $g(\mathbf{x}) = 0$
4     $U(\mathbf{x}) \leftarrow q$
  /* Update distance of $L_{upper}$ again and collect converged nodes */
5 foreach $\mathbf{x} \in L_{upper}$ do in parallel
6     $p \leftarrow U(\mathbf{x})$
7     $q \leftarrow$ solution of $g(\mathbf{x}) = 0$
8     if $p \leq q$ then
9         add $\mathbf{x}$ to $\tilde{L}_{tight}$
```

	Ex. 1	Ex. 2	Ex. 3	Ex. 4	Ex. 5
L_{upper}	4.26	4.74	5.33	8.84	8.70
L_{tight}	2.74	3.09	3.53	7.32	7.36

Table 3: Comparison of average number of update per node for different initial active list generation schemes (tested on GO-FIM8).

for each group. On the other hand, if there are only a small number of groups, then it may not reflect the causal dependency of the original grid well and eventually degrades the overall performance.

To address these problems, we employ K-means clustering algorithm [69] to decompose the coarse grid into disjoint groups by clustering blocks having similar distance values. There are some automatic methods to determine K value (i.e., the number of clusters), for example Tibshirani *et al.* [70]. This approach is minimizing the variation of values within each group, i.e., *within-cluster dispersion* (Figure 22). As can be seen in this graph, the dispersion value changes gradually and it is difficult to find the clear cuts to separate the data into groups. Therefore, we have decided to determine the best clustering number empirically.

5.4 Results and Discussion

In this subsection we evaluate the performance of parallel FIM and GO-FIM, and compare them with the most popular serial and parallel eikonal solvers, such as FMM [24], Zhao's FSM [25] [31], Detrixhe *et al.*'s parallel FSM (DFSM) [32] and parallel Heap Cell Method [66]. In cases of GO-FIM and pHCM, we use two different block size configurations – GOFIM4 and pHCM4 for $4 \times 4 \times 4$ and GOFIM8 and pHCM8 for $8 \times 8 \times 8$. Running times are measured using single and multiple threads in order to assess both serial computing performance and parallel scalability of

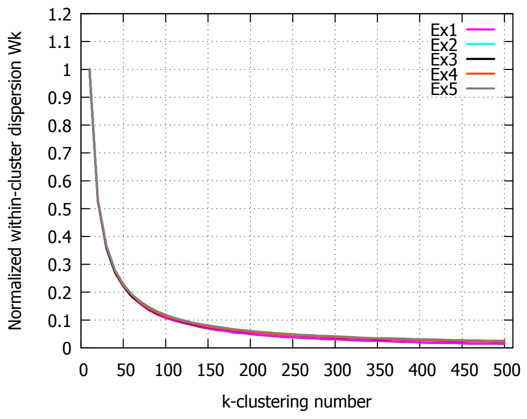
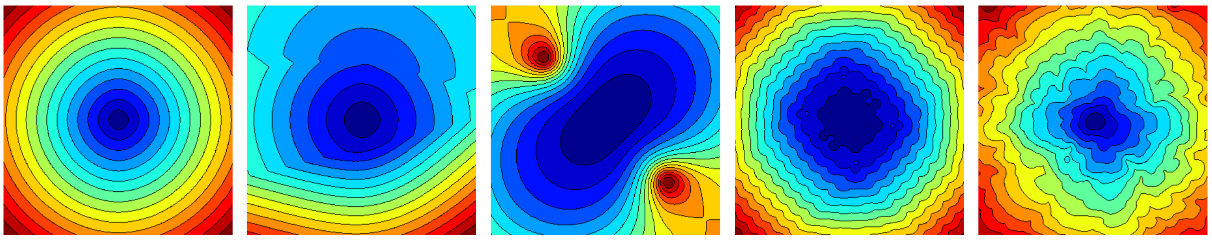


Figure 22: Within-cluster dispersion for different clustering value K.



(a) Example 1 (b) Example 2 (c) Example 3 (d) Example 4 (e) Example 5

Figure 23: Color-coded distance map with iso-contours of our test datasets. Blue to red color : distance to the seed region on each map. For visualization purpose, the center slice of each 3D map is shown here.

each method.

All experiments were conducted on a NUMA(Non Uniform Memory Access)-based Linux server equipped with four AMD Opteron 6128 octa-core processors sharing 64 GB of DDR3 memory. We implemented the experiment code in C++ and OpenMP with the -O3 level optimization with gcc 4.7.0, and all floating point computations are performed in 64 bit double precision. We used OS default thread affinity, which is round-robin thread assigning to an idle processor. For measuring scalability of each method, we tested up to 32 parallel threads except Zhao's parallel FSM [31] that only allows one, two, four or eight threads running concurrently because the algorithm is based on the decomposition of Gauss-Seidel (G-S) update directions (for example, in 2D case, there are only four G-S update directions, ascending and descending directions along x and y axis). To measure overall performance, we check average wall-clock running time of each method including all the initialization/preprocessing time except map generation because speed maps do not depend on the seed/source location.

We tested the methods on the various types of speed maps, ranging from a plain constant speed map to a complex shape map. The definition of each input speed map is as follows.

Example 1. $f = 1$. Constant speed map.

Example 2. $f = \frac{1}{4}, \frac{1}{2}, 1$. Speed map with three layers of different speed values.

Example 3. $f = 6 + 5\sin(2\pi x) * \sin(2\pi y) * \sin(2\pi z)$.

Example 4. $f = 1 + 0.5\sin(20\pi x) * \sin(20\pi y) * \sin(20\pi z)$.

Example 5. $f =$ Spatially coherent random speed map.

where all speed maps are defined in the normalized domain $\Omega = [0, 1]$.

Example 1 is the simplest example that the speed value is identical on every grid node. On this speed map, waves propagate as a circular shape from the seed points and the characteristic paths are straight lines from the seed region. Example 2 has three levels of speed variation, which mimics wave propagation through three different materials. In each layer, characteristic paths are straight lines, but there is a large characteristic direction change at the boundary of two adjacent layers. Example 3 and 4 are sinusoidal speed maps to represent moderate and highly oscillatory isocontours of the distance maps. Example 5 is a spatially correlated random speed map so that speed values are locally homogeneous but varying globally. This dataset is very challenging because characteristic paths frequently turn their directions. These datasets were chosen in order to elaborate the characteristics of each method. In all experiments, we used a 256^3 three-dimensional grid, with the single center source point. Speed maps (input and coarse level) are pre-computed and stored in each grid points. For clustering, we used 240 groups for GOFIM4 and 150 for GOFIM8. Figure 23 shows the color plot and iso-contour rendering of the solution (i.e., distance) of the eikonal equation for each speed map.

Single-threaded result

Although we propose parallel algorithms in this section, it is important to conduct experiments using a single thread because each algorithm's intrinsic characteristics can be revealed by analyzing single-thread performance. The running time of each method on five different datasets are listed in Table 4 (first multi-row), and their average update numbers are listed in Table 5. The datasets are chosen so that different levels of complexity can be tested. Example 1 is the simplest data, and the complexity of data is gradually increasing from Example 2 to 5.

As expected, the performance of FMM did not vary over different examples because FMM is worst-case optimal and less susceptible to speed variation. In contrast, iterative algorithms, such as FSM and FIM, are affected by the complexity of the speed maps. Even though they belong to the same class of algorithm, they behave differently. In Example 2, FSM and DFSM slow down by a factor of four to five compared to Example 1, but FIM is only twice slower. Example

		Ex. 1	Ex. 2	Ex. 3	Ex. 4	Ex. 5
1	FMM	23.50	23.98	23.96	25.00	25.92
	FSM	11.51	54.29	51.71	102.05	94.02
	DFSM	23.41	87.82	86.07	159.57	148.27
	FIM	**5.62**	9.15	44.33	24.95	34.46
	pHCM4	9.26	9.86	11.16	17.90	17.13
	pHCM8	9.16	9.32	11.97	26.78	22.28
	GOFIM4	10.26	10.18	10.85	**16.51**	**16.72**
	GOFIM8	6.76	**7.36**	**8.34**	18.98	20.21
2	FSM	6.84	30.41	37.93	79.12	67.12
	DFSM	16.72	68.45	60.82	138.00	109.85
	FIM	**3.08**	4.93	24.67	14.13	17.63
	pHCM4	5.73	5.68	6.69	10.21	9.76
	pHCM8	4.91	4.99	6.46	13.77	11.83
	GOFIM4	5.49	5.97	5.93	**8.90**	**8.31**
	GOFIM8	3.54	**4.16**	**4.50**	9.94	9.84
4	FSM	4.87	19.68	28.59	52.61	46.63
	DFSM	9.00	35.47	33.48	63.60	60.80
	FIM	**1.68**	2.79	13.12	7.65	9.87
	pHCM4	3.28	3.35	3.83	5.82	5.50
	pHCM8	2.64	2.72	3.33	7.16	6.28
	GOFIM4	2.93	3.16	3.29	**4.68**	**4.53**
	GOFIM8	1.85	**2.18**	**2.49**	5.17	5.27
8	FSM	4.26	16.23	20.22	36.66	35.54
	DFSM	4.65	18.51	18.14	33.15	30.71
	FIM	**0.77**	1.30	6.77	4.04	5.28
	pHCM4	1.75	1.82	2.05	3.12	2.93
	pHCM8	1.37	1.46	1.76	3.69	3.25
	GOFIM4	1.58	1.73	1.82	**2.55**	**2.39**
	GOFIM8	0.97	**1.20**	**1.39**	2.81	2.80
16	FSM	-	-	-	-	-
	DFSM	2.51	9.58	9.16	17.47	17.26
	FIM	**0.44**	**0.70**	3.52	2.23	2.91
	pHCM4	0.97	1.02	1.16	1.74	1.63
	pHCM8	0.75	1.81	0.96	1.96	1.73
	GOFIM4	0.92	1.03	1.09	**1.49**	**1.36**
	GOFIM8	0.57	0.74	**0.87**	1.66	1.64
32	FSM	-	-	-	-	-
	DFSM	1.78	6.10	5.72	11.53	10.35
	FIM	**0.26**	**0.40**	1.96	1.39	1.73
	pHCM4	0.75	0.68	0.92	1.46	1.18
	pHCM8	0.46	0.49	**0.64**	1.15	1.03
	GOFIM4	0.65	0.77	0.83	**1.05**	**0.96**
	GOFIM8	0.43	0.60	0.68	1.13	1.13

Table 4: Running time using different number of threads (1 to 32 threads) measured in second. The fastest time for each dataset is marked in boldface.

3 and 4 show more interesting results – FSM's running time grows about five and nine times respectively (compared to Example 1), but FIM's running time did not follow the similar pattern and Example 3 was much slower than Example 4. This shows that FIM is more susceptible to a large (global) speed variation (as in Example 3) than local variation in Example 4. In our experiments, FIM mostly runs faster than FSM and DFSM because FIM avoids unnecessary computation by only updating active nodes. In Example 1, FIM only requires 2N updates for N nodes because every node converges after a single iteration. In contrast, FSM requires at least 9N computations because a single pass of entire grid update requires eight sweeps per node, one per each axis, and one more sweep to check convergence. We also observed that DFSM is always slower than FSM up to a factor of two for a single thread even though the total number of update is same as FSM. This might be due to better cache-coherency of axis-aligned sweeping of FSM.

pHCM and GO-FIM belong to the class of two-level algorithm, and both perform better than the other methods on all examples except Example 1 where the overhead of both methods outweighs FIM. Other than Example 1, both methods outperform other iterative methods by a large margin, especially this characteristic becomes clearer on complicated data like Example 4 and 5. Note that both pHCM and GO-FIM even outperform FMM on the complicated examples without managing a fine-level priority queue, which we believe is the right approach to improve the performance of label-correcting algorithms. It is also worth noting that performance of GO-FIM is affected by the choice of block size. For simple maps like Example 1, original FIM performs best. If maps are reasonably complex, like Example 2 and 3, then GO-FIM with a larger block size (i.e., GOFIM8) performs well because there is not much variation of characteristic path direction that needs to be covered by fine-grain decomposition of the domain, and therefore using a larger block size will reduce the overhead of GO-FIM algorithm. For highly complicated maps, like Example 4 and 5, a smaller block size works best.

Even though GO-FIM and pHCM share a similar idea, there are also subtle but important differences that make two methods perform differently. pHCM restricts the computational domain to a single cell per thread, but GO-FIM expands the computational domain based on the order of block clusters (this is also different from FMSM that expands the domain one cell at a time). In addition, pHCM uses a dynamic cell ordering using a heap while GO-FIM uses a coarse static ordering based on the clustering. As shown in Table 5, pHCM4 shows less number of updates than GO-FIM4, but for a larger block size GO-FIM8 needs fewer updates than pHCM8. This is because pHCM can find more accurate causal relationship between blocks, so if the block size is small then pHCM may need to update less than GO-FIM. However, if the block size is larger, than it may impair the accuracy of the causal dependency found by pHCM, so the number of updates can be larger than GO-FIM. Note that the affect of block size is smaller in GO-FIM, and sometime a large block size is even better for GO-FIM (for example, in Example 3, the number of update is smaller in GO-FIM8, but that of pHCM8 is higher than pHCM4). It is also worth noting that pHCM has around 5% of overhead for heap maintenance (for pHCM4

	Ex. 1	Ex. 2	Ex. 3	Ex. 4	Ex. 5
FMM	2.99	2.99	2.99	2.99	2.99
FSM	9	55	49	86	79
DFSM	9	55	49	86	79
FIM	2.00	3.73	19.40	9.81	12.91
pHCM4	2.96	3.05	3.38	4.88	4.38
pHCM8	3.49	3.61	4.24	8.41	7.29
GOFIM4	3.46	3.64	3.75	5.42	5.08
GOFIM8	2.74	3.10	3.53	7.32	7.36

Table 5: Average number of update of eikonal solvers on different examples.

	GO-FIM4		GO-FIM8	
	Example 2	Example 5	Example 2	Example 5
1	3.30	4.00	0.45	0.30
2	3.86	4.55	0.58	0.37
4	4.75	5.46	0.81	0.48
8	6.62	7.00	1.34	0.71
16	9.42	9.30	1.86	1.04
32	13.69	13.05	3.13	1.91

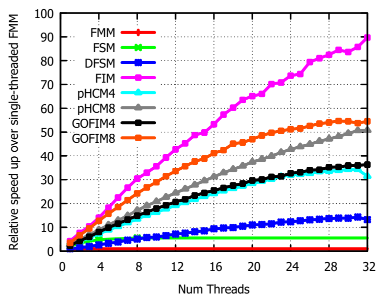
Table 6: The proportion of the preprocessing time of GO-FIM (measured in the percentage over the entire running time)

case, [66]), but GO-FIM shows smaller overhead (e.g., preprocessing cost) around 4% at most (Table 6, thread 1). Therefore, even though pHCM8 requires fewer updates than GO-FIM8 for Example 5, GO-FIM8 is actually faster than pHCM8.

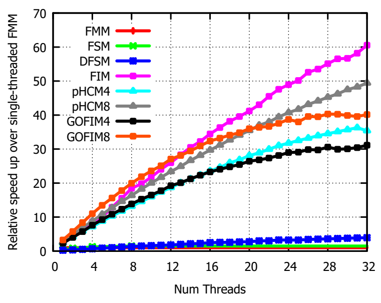
Multi-threaded result

Multi-threaded results are demonstrated as raw running times (Table 4), relative performance over FMM (Fig 24), and parallel scalability of each solver (Fig 25). A popular parallel eikonal solver is Zhao's parallel FSM [31]. The main idea of parallelization in this method is running G-S update concurrently for different sweeping directions. However, it only allows parallelization using two, four, and eight threads because there are only two independent sweeping directions per each axis. This significantly impairs scalability of the method. As you can see in Figure 25, the observed maximum speed up for 32 threads over a single thread is only about a factor of three.

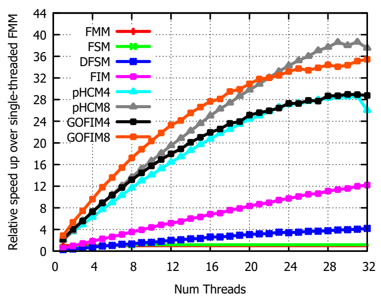
In DFSM [32], the authors proposed a diagonal sweeping approach in order to improve



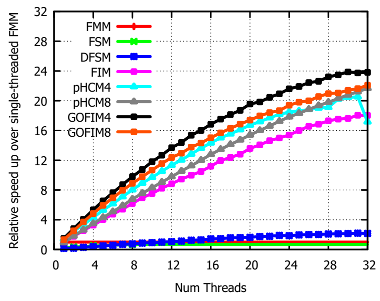
(a) Example 1



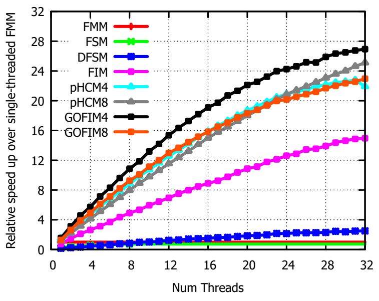
(b) Example 2



(c) Example 3



(d) Example 4



(e) Example 5

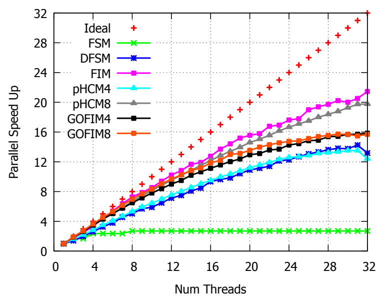
Figure 24: Parallel running time result. The horizontal axis is the number of parallel threads, and the vertical axis is the relative speed up of each solver over the single-threaded FMM

parallel scalability of the sweeping algorithm. We observed that DFSM with 32 threads runs around a factor of $13 \sim 15\times$ faster than a single-threaded DFSM. However, with a small number of parallel threads, the running time of DFSM is longer than that of FSM (see Figure 24 and Table 4) due to the overhead of non-axis aligned sweeping direction. Therefore, DFSM favors the systems with many parallel processors.

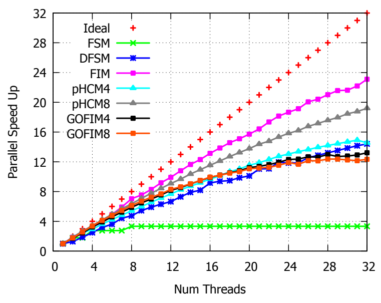
Compared to the two parallel sweeping methods discussed above, our lock-free parallel FIM algorithm runs faster and scales better on multiple threads. We observed that FIM can achieve the best scalability on Example 1, 2 and 3, almost up to 24× speed up for 32 threads, which results in about 80× speed up over a single-threaded FMM. However, FIM did not scale well on complex data like Example 4 and 5, which is the limitation of conventional FIM.

Unlike FIM that directly parallelizes the active list, pHCM concurrently updates multiple blocks in the coarse-level grid by letting each thread handles the each block and updates using the modified LSM method. In our experiments, we observed that pHCM4 and pHCM8 achieved up to $12 \sim 16\times$ and $18 \sim 23\times$ speed up, respectively. Similar to GO-FIM, pHCM scales much better than FSM and DFSM, but pHCM's average computation number increases as the number of threads increases (reported in Chacon *et al.* [66]), which can be a bottleneck for scaling to a large number of threads.

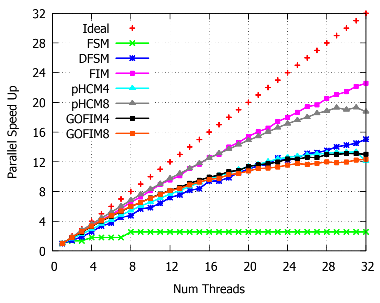
GO-FIM scales reasonably well compared to other parallel solvers except a few cases – FIM



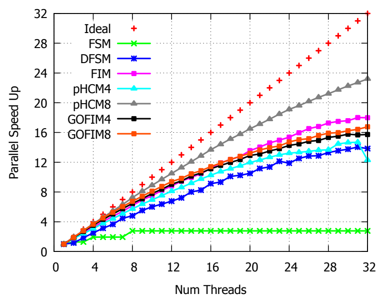
(a) Example 1



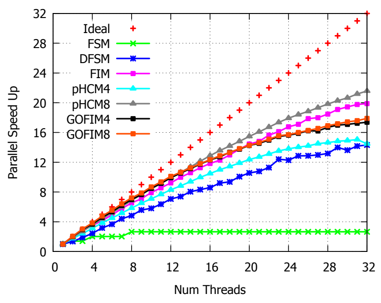
(b) Example 2



(c) Example 3



(d) Example 4



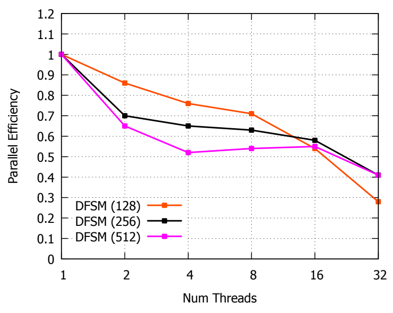
(e) Example 5

Figure 25: Parallel scalability result. The horizontal axis is the number of parallel threads, and the vertical axis is the speed up factor (i.e., scalability) of each solver.

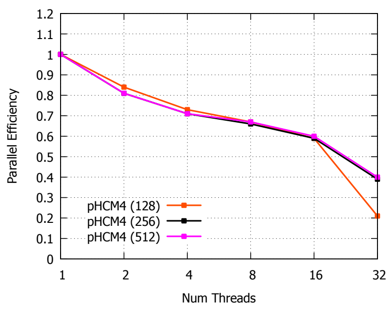
outperforms GO-FIM8 on Example 2 for 13 threads and up (see Figure 24 (b) pink and orange curves cross near 13 threads), and pHCM8 outperforms GO-FIM8 for 22 or more threads on Example 3 (see Figure 24 (c) orange and gray curves cross near 22 threads). However, GO-FIM is practically the best option for most cases under 32 threads because it is not common to have more than 32 cores in a single computing node.

Parallel efficiency on different grid size

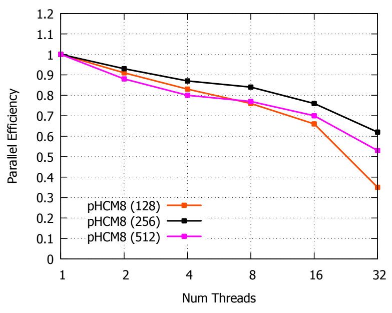
The performance of parallel algorithms is often strongly affected by the data size, so we measured the parallel efficiency of each solver on a constant speed map of three different grid sizes ($128^3, 256^3$, and 512^3). Figure 26 demonstrates how the parallel efficiency varies for different grid sizes and number of threads. DFSM shows better parallel efficiency for a small number of threads (less than 16), which may be due to the cache-coherency effect of specialized sweeping scheme. pHCM was not much affected by the data size and thread counts. However, pHCM4 shows a steep drop of the curves for higher thread counts. FIM and GO-FIM clearly show increasing parallel efficiency as the grid size grows. Unlike pHCM4, GO-FIM4 shows better parallel efficiency for the grid size 512^3. This is partially because pHCM's heap maintenance overhead increases but GO-FIM effectively hides the preprocessing overhead as the data size grows. This implies that GO-FIM suits better for large datasets than the other solvers for higher thread



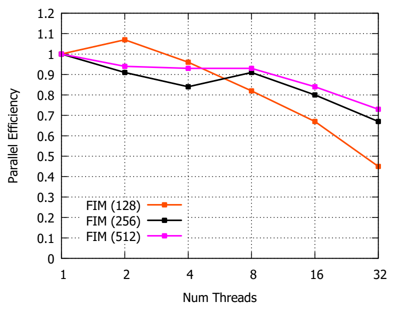
(a) DFSM



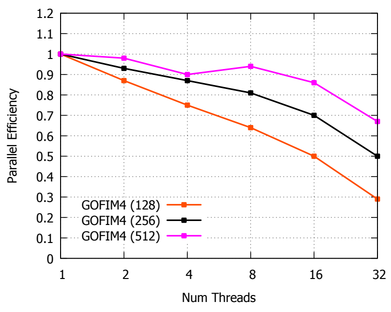
(b) pHCM4



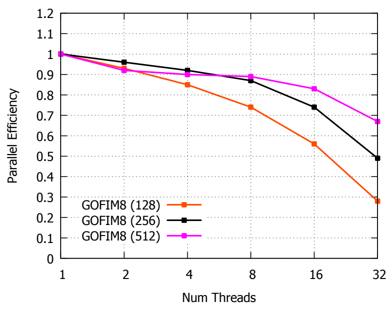
(c) pHCM8



(d) FIM



(e) GO-FIM4



(f) GO-FIM8

Figure 26: Parallel scalability test result of Example 1 on different grid size (N^3). The horizontal axis shows the number of parallel threads by a log scale, and the vertical axis shows the parallel efficiency (parallel speed up / num threads).

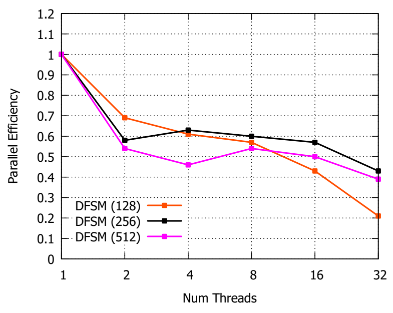
counts.

GO-FIM on the GPU

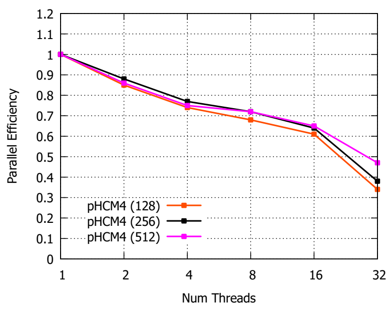
The proposed GO-FIM can be thought of as an extension of BlockFIM [19] because GO-FIM uses rough (i.e., not exact) ordering of block update to reduce unnecessary computation of the original FIM. Table 7 compares BlockFIM and GO-FIM algorithms on the GPU. We used an NVIDIA Tesla K40c for comparing two GPU solvers on a 512^3 grid using two block sizes, 4^3 and 8^3. As shown in this table, GO-FIM effectively reduced running time on the complicated maps (Example 2 to 5). GO-FIM was slower than BlockFIM on Example 1 because the update order is identical on coarse and fine grids so GO-FIM cannot reduce the amount of computation but there exists extra overhead of preprocessing in GO-FIM.

Discussion

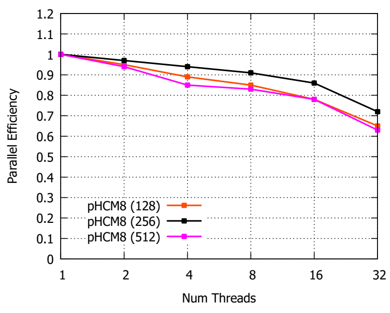
GO-FIM requires pre-processing that is not necessary in the original FIM. In order to assign per-block updating order, GO-FIM must run FIM on the coarse grid to compute distance per block. In addition, proper clustering of blocks is another important pre-processing step to improve the performance. In general, less than one second is spent for preprocessing for single-threaded GO-



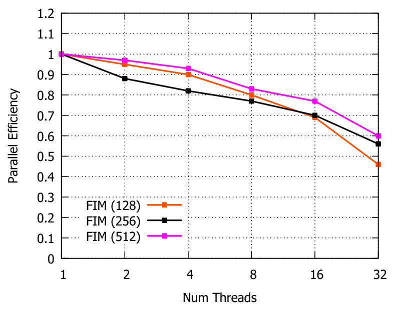
(a) DFSM



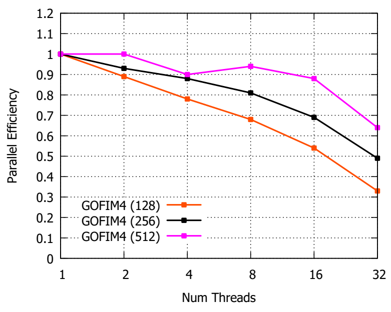
(b) pHCM4



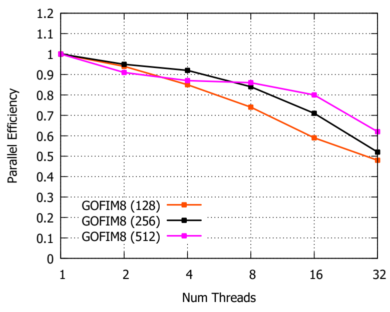
(c) pHCM8



(d) FIM



(e) GO-FIM4



(f) GO-FIM8

Figure 27: Parallel scalability test result of Example 4 on different grid size (N^3). The horizontal axis shows the number of parallel threads by a log scale, and the vertical axis shows the parallel efficiency (parallel speed up / num threads).

FIM8, which is only a small fraction of the total running time. In addition, all computations in preprocessing step can be done in parallel as well, so it does not impact the scalability of the algorithm. One thing we should consider is that preprocessing time varies depending on the block size and inversely proportional to the overall performance – meaning that it is better to have smaller block size to represent underlying speed map more faithfully, but small block size will increase preprocessing time as well. We found that the block size of 4 or 8 works best for 256^3 input data size.

GO-FIM belongs to the class of two-scale hybrid algorithms, such as FMSM proposed by Chacon *et al.* [64], but there also exists important differences between FMSM and GO-FIM. In FMSM, as the authors pointed out, statically computed cell orders do not perfectly capture the correct causal relationship due to the large block size, and that is why the authors later proposed an improved algorithm, HCM, by employing a dynamic ordering of cells using Heap. In GO-FIM, we employ a clustering method to group the subregions based on the static cell ordering, and maximize the parallelism of FIM algorithm by leveraging a larger computational domain. Therefore, we are able to achieve the good performance comparable to that of a dynamic ordering method without ordered data structure as in pHCM while increasing parallel performance. In addition, unlike other hybrid algorithms, it is a natural transition from BlockFIM to GO-FIM

	Ex. 1	Ex. 2	Ex. 3	Ex. 4	Ex. 5
BlockFIM4	0.73	1.37	4.51	6.24	6.21
BlockFIM8	0.87	1.34	2.26	3.11	2.87
GO-FIM4	1.15	1.45	1.33	1.50	1.61
GO-FIM8	1.08	1.25	1.28	2.34	2.24

Table 7: Running Time comparison of BlockFIM and GO-FIM on an NVIDIA Tesla K40c GPU.

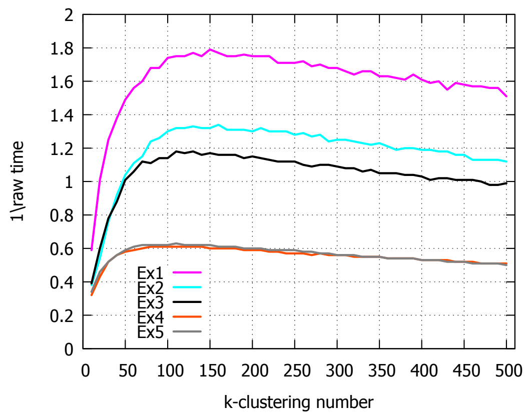


Figure 28: Performance of GO-FIM8 with 16 threads measured using various cluster sizes.

because the algorithm is already using a block-based updating scheme anyway.

GO-FIM's performance is affected by the number of clusters. We observed that too many clusters, for example each block as a single cluster, causes too much overhead for iteration due to thread synchronization, while too few clusters do not represent block orders accurately. We empirically determined the best number of clusters as shown in Figure 28, which is roughly around 150 clusters for the coarse grid size of 32^3 (GO-FIM8) and 240 clusters for coarse grid size of 64^3 (GO-FIM4). Figure 28 is the reciprocal of the raw running time of GO-FIM8 with 16 threads. We noticed that the best results of Example 1 $\sim$ 3 are located around 150 clusters. More difficult cases, like Example 4 and 5, favor small number of clusters, but around 150 clusters is still close to their best results.

One limitation of GO-FIM is that the performance depends on the structure of the input speed map and the layout of coarse blocks. To emphasize this effect, we tested GO-FIM on the maze-like data having permeable barriers with a low speed value (see Figure 29). Dotted lines show the boundary of blocks. The speed value of gray regions is 0.01 and that of while region is 1, and the block size and the thickness of the barrier is 8. We change the location of barriers so that blocks and barriers are overlapping differently. Figure 29 (a) is the case that block boundary and barriers align perfectly so that there is no overlapping of blocks over barriers. In this case, distance on coarse blocks represent the propagation order correctly (the orange arrow

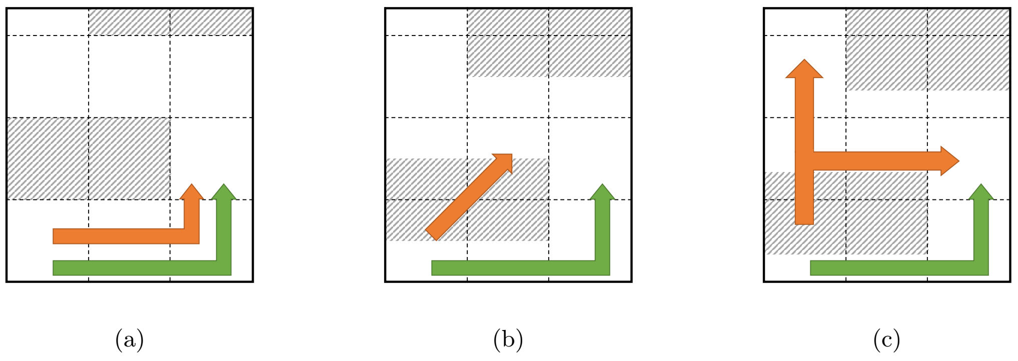


Figure 29: Example of miss prediction on coarse grid. The speed value on white region is 1.0, and that of grey region (permeable barrier) is 0.01. The green arrow represents correct wave propagation direction, and the orange arrow represents wave propagation direction on the coarse grid of GO-FIM.

is the wave propagation direction of coarse blocks in GO-FIM, and the green arrow is the correct wave propagation direction). Figure 29 (b) is the case that some blocks overlap with barriers by half, therefore the average distance on each block is same. In this setting, four blocks in the bottom-left corner have same speed value, so the wave propagates as circular shape (along the orange arrow) while the correct direction should be the green arrow. Figure 29 (c) is the case where the blocks on the bottom row largely overlap with the barrier while the blocks above that row overlap much less with the barrier. In that case, the speed of bottom row is much smaller than that of the row above, so the wave propagates faster along up direction, which increases unnecessary computation. One way to resolve this problem is to make the block size small enough to represent the underlying speed map structure better with the coarse grid, but as we discuss above, using smaller block size will increase the preprocessing time so it will impair the overall performance. Another solution might be using adaptive block size to better represent the speed map, but we leave this for the future work.

5.5 Summary

In this section, we proposed two parallel eikonal solvers, lock-free FIM and Group-Ordered FIM. Lock-free FIM is an extension of original FIM for efficient parallelization on shared memory systems, and GO-FIM further improves the performance of parallel FIM by employing rough ordering of blocks on a coarse grid and clustering of blocks to reduce iteration numbers and to increase the parallelism. The experiment results show that the proposed method maps well on a shared memory system and outperforms popular parallel eikonal equation solvers in many cases.

In the future, we will conduct a theoretical study of GO-FIM algorithm, and plan to extend GO-FIM to distributed systems. Exploring the real-world applications that benefit from the proposed fast parallel eikonal solvers is another future research direction.

VI A Causality-Ordered Fast Iterative Method for Eikonal Equations

6.1 Introduction

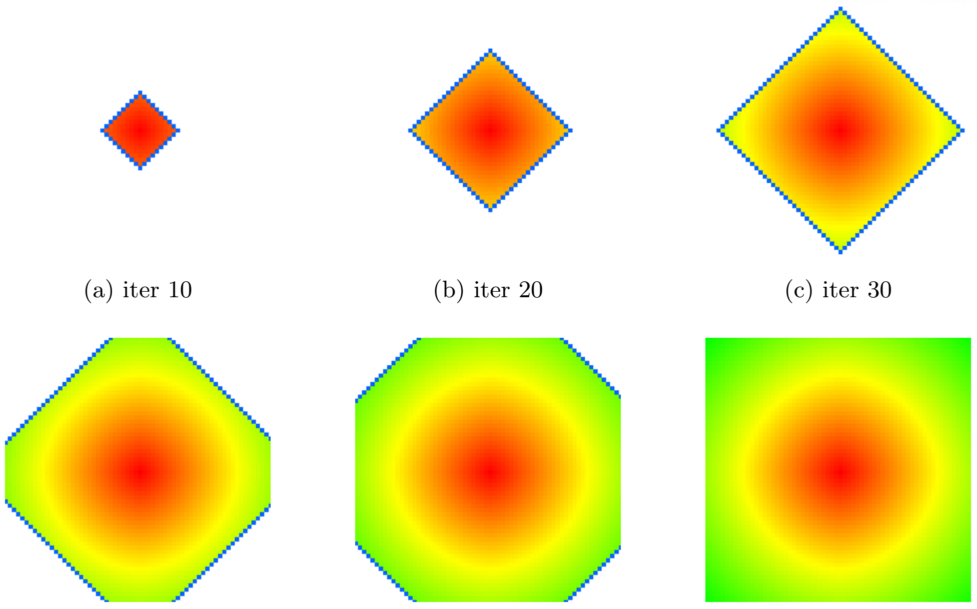
FIM is an iterative algorithm that adaptively updates the solution to the eikonal equation defined on the grid. FIM maintains a narrow band, i.e., *active list*, for storing the grid nodes to update. The main idea is that the active list is not constructed based on a strict causal relationship (i.e., dependency) as in the FMM, which allows concurrent updating of multiple nodes. A node can be removed from the active list only when it is converged; otherwise, it remains in the list and is updated again. In addition, a converged node activates its non-converged adjacent nodes, and previously converged nodes can be reactivated (i.e., added to the active list again) later for further updating. FIM is only based on a simple data structure such as the queue, rather than an ordered data structure, and the lack of an ordered data structure is the main disadvantage of FIM. For example, the Fast Marching Method (FMM [24]), which is a representative method for the eikonal equation, maintains causality information using a heap data structure. However, FIM doesn't have a method to maintain causality of the solution. Instead, FIM performs iterative calculations until all nodes in the active list become converged. In terms of complexity, the cost of FMM is $O(nlogn)$ as a worst-case optimal, while the cost of FIM is $O(kn)$. However, the actual performance of each method highly depends on the input.

Figure 30, and Figure 31 describe the characteristics of FIM. Each example is the result of using FIM for a 64×64 2D domain. Figure 30 is the result of a constant speed map with all speed function values the same to 1, and Figure 31 is the result of an oscillatory continuous speed map. In each figure, (a) to (e) describe the intermediate result of each iteration, and (f) shows the final solution. Red and green each represent the nearest and farthest distances to the seed point, respectively. Blue and black nodes represent nodes in the active list. Among them, blue nodes will be removed from the active list because they have converged, while black nodes are re-computed in the next iteration.

Figure 30 is the best example for FIM. All nodes only include a time to the active list and do not require additional calculations. However, Figure 31 illustrates the disadvantages of FIM. Many nodes belonging to the active list are not converged with a single calculation and must be recalculated several times. In addition, a previously converged node can be reactivated later. In this example, the average number of nodes included in the active list is 3.5.

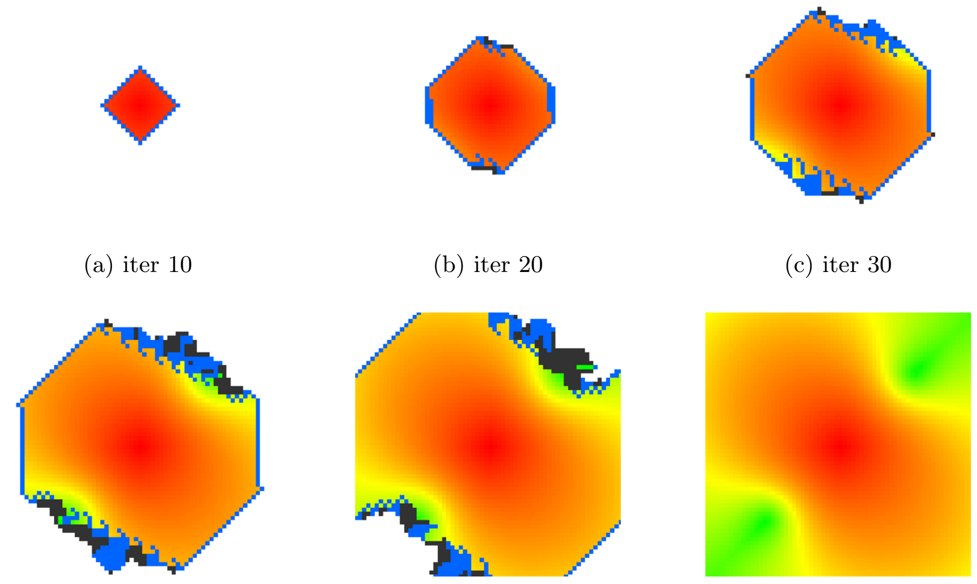
6.2 Causality-Ordered FIM

The propagation of the active list of FIM does not conform to the actual distance to the seed points. Instead, FIM allows for multiple computation; each node is updated several times to get the correct solution. However, we can solve a redundant computation problem by introducing causality information. In this section, we propose a novel eikonal equation solver, a causality-


(a) iter 10 (b) iter 20 (c) iter 30

(d) iter 40 (e) iter 50 (f) final

Figure 30: Color-coded distance to the center on constant domain ($f = 1$). (a)-(e) describe the intermediate result of FIM in the iteration. Red and green represent the nearest and farthest distances to the seed point, respectively. Blue and black nodes are included in the active list. Black nodes are unconverged, so they are re-computed in the next iteration.


(a) iter 10 (b) iter 20 (c) iter 30

(d) iter 40 (e) iter 50 (f) final

Figure 31: Color-coded distance to the center on sinusoidal domain ($f = 6{+}5\sin(2\pi x){*}\sin(2\pi y)$). (a)-(e) describe the intermediate result of FIM in the iteration. Red and green represent the nearest and farthest distances to the seed point, respectively. Blue and black nodes are included in the active list. Black nodes are unconverged, so they are re-computed in the next iteration.

ordered FIM (COFIM), which is an extension of FIM using a node-level causality dependency. The target idea is to prevent the nodes having causality relations from being in an active list at the same time. To achieve this, we have defined a parent and child relationship between the nodes using causality dependency. Moreover, we have proposed a rule to restrict further propagation of the child while the parent nodes are being activated. The main design goals for our new idea can be summarized as follows:

- Proposed method should maintain the active list, which is managing the activated nodes.
- Proposed method should manage the active list with a simple list, so do not use a complex data structure for sorting. The nodes belonging to the active list should be able to update simultaneously.
- Proposed method should prevent the situation in which nodes having dependency on each other are activated at the same time.

The first two criteria are the core design of FIM for easy parallelization. However, these two criteria must be kept in the new algorithm. Therefore, we must achieve the third criterion without harm to the first two.

We solve $U(\mathbf{x})$ after transforming Equation 2 to the associated quadratic equation in closed form. Here, it uses multiple numbers of $U(\mathbf{x})^{pmin}$ to calculate $U(\mathbf{x})$. Among them, U depends only on the neighboring values smaller than itself [11] (i.e., *causal*). Thus, we can define a *parent* of the node $\mathbf{x}$, which the neighbor nodes provided$U(\mathbf{x})^{pmin}$, as the minimum neighbor value. For a node, multiple numbers of a parent can exist (one parent node per axis). The opposite relationship can be defined as *child*. Figure 32 illustrates the parent-child relationship in the Godunov upwind difference scheme. In a 2D grid, node $\mathbf{x}_{i,j}$ computes a distance value using two nodes $\mathbf{x}_{i-1,j}$ and $\mathbf{x}_{i,j-1}$; they are in opposite directions of the characteristic path. These two black nodes are considered as the parent of $\mathbf{x}_{i,j}$. In same time, a node $\mathbf{x}_{i,j}$ is defined as child of the node $\mathbf{x}_{i-1,j}$ and the node $\mathbf{x}_{i,j-1}$. If a parent node has an incorrect value (or not fully converged value), all child nodes that depend on the parent's U value will also have the wrong values.

Algorithm description

To reduce redundant iterative computation of FIM, we focused on a parent and child relationship (i.e, causality dependency) between the nodes. Algorithm 7 explains the function of **Update U(x)** based on Equation 2. This is a modified pseudocode for the solution of the 3D eikonal equation, which commonly uses [25] [19]. This function computes the $U(\mathbf{x})$ value using four values a, b, c, f and gets the parent nodes $P(\mathbf{x})$ for a node $\mathbf{x}$. After that, it returns whether a node $\mathbf{x}$ becomes updated or not.

Based on the causality dependency between nodes, we propose a Causality-Ordered Fast Iterative Method for the eikonal equation (COFIM). This method is an extension of FIM [19]

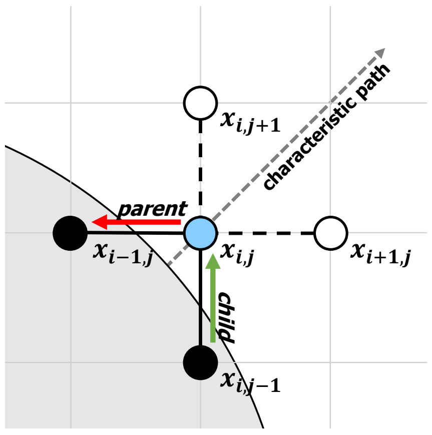


Figure 32: Causality dependency in Godunov upwind difference scheme. The distance of the child node $\mathbf{x}_{i,j}$ (blue node) was calculated from two parent nodes (black nodes).

Algorithm 7: UPDATE $U(\mathbf{x})$ IN 3D DOMAIN

```
$a = U(\mathbf{x})^{xmin}$, $b = U(\mathbf{x})^{ymin}$, $c = U(\mathbf{x})^{zmin}$, $f = F(\mathbf{x})$
$sort\ a, b, c$ for $c \leq b \leq a$
$u, v, w \leftarrow$NULL
$t \leftarrow c + 1/f$
$u \leftarrow U^{-1}(c)$
if $t \geq b$ then
    $t \leftarrow (b + c + sqrt(-b^2 - c^2 + 2bc + 2/f^2))/2$
    $v \leftarrow U^{-1}(b)$
    if $t \geq a$ then
        $t \leftarrow (2(a+b+c) + sqrt(4(a+b+c)^2 - 12(a^2+b^2+c^2-1/f^2)))/6$
        $w \leftarrow U^{-1}(a)$
if $t < U(x)$ then
    $U(\mathbf{x}) \leftarrow t$
    $P(\mathbf{x}) \leftarrow u, v, w$
    return $true$
else
    return $false$
```

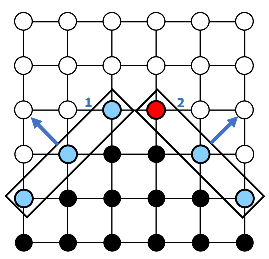
and Group-Ordered FIM (See Section V). The pseudocode description of the proposed algorithm is given in Algorithm 8. The initialization step is almost same as the FIM; however, we add a process to set the parent for each node as empty (line 7). The main iteration part, from line 8 to line 27, shares the same structure with FIM. However, there are two major changes. One is using proposed Algorithm 7 when updating the U value to calculate either parent. The other is the addition of lines 15-22 to check the status of the parent. The key idea of this algorithm is quite simple. If a *parent* of node $\mathbf{x}$ is updated (or not converged yet), the node $\mathbf{x}$ which derived from the *parent* will not have a final solution either. At this time, a *child* node that has node $\mathbf{x}$ as a parent also can't obtain a final solution. Therefore, for a node $\mathbf{x}$ which has converged (or temporarily stabilized), if an update of the parent is detected, node $\mathbf{x}$ does not attempt to activate the adjacent neighbor nodes. This rule is simple, but when a parent node is added to the active list, it prevents a child or grandchild node that depends on the corresponding parent node from being added to the active list at the same time. Therefore, it has provided a computational ordering (or guide) through the causality relationship between nodes. So, we named this algorithm a causality-ordered FIM (COFIM). This does not require any additional sorted data structure such as heap, but it still gives a rough updating order using simple a causality relation between nodes

Figure 33 describes the difference between FIM and COFIM with a simple example. In the figure, both the blue nodes and the red nodes belong to the active list. In iter 1 (Figure 33 (a)), the blue nodes become converged, so they extend the active list to the blue arrow direction. However, the red node is unconverged, so it remains in the active list. In iter 2, all blue and red nodes are converged, and they try to add unconverged neighbor nodes (Figure 33 (b)). In FIM algorithm, there exist three propagation groups on the active list at iter 3 (on Figure 33 (c)). At this time, most of the nodes in group 1 have a causal dependency on group 3, which has been newly created from the red node. These nodes in group 1 can't obtain the correct solution, so they will be updated later through the propagation of group 3. The presence of groups 1 and 3 in the active list at the same time is the main cause of redundant computation in FIM. However, in Figure 33 (d), COFIM solved this problem. The nodes that added their parent nodes to group 3 no longer attempt further propagation, so group 1 will disappeared. This not only reduces the number of nodes in the active list, but also efficiently reduces the number of calls to the update function per node.

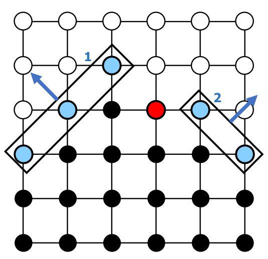
Figure 34 is illustrates the intermediate result of COFIM. This uses the same input data as Figure 31. However, COFIM has effectively controlled the active list, so it has reduced redundant computation. In this example, the average number of nodes in the active list is 1.5 and the value has been reduced to about 40 percent from FIM.

Algorithm 8: CAUSALITY-ORDERED FIM

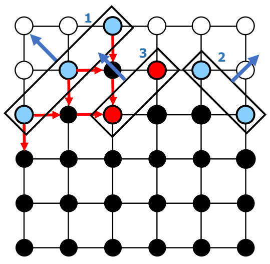
```
Input: Grid $\Omega$, Solution $U$, Active list $L$
/* Initialization */
1  forall $x \in \Omega$ do
2      if $x$ is a source node then
3          $U(x) \leftarrow 0$
4          add $x$ to $L$
5      else
6          $U(x) \leftarrow \infty$
7          $P(x) \leftarrow \emptyset$
/* Compute new solutions for $L$ */
/* Use Algorithm 7 for Update $U$ */
8  while $L \neq \emptyset$ do
9      $L_{next} \leftarrow \emptyset$
10     forall $\boldsymbol{x} \in L$ do
11         if Update $U(\boldsymbol{x}) == true$ then
12             add $\mathbf{x}$ to $L_{next}$
13         else
14             $flag \leftarrow false$
               /* Check parent nodes $u, v, w$ */
15             forall $\boldsymbol{p} \in P(\boldsymbol{x})$ do
16                 if $\boldsymbol{p} \in L_{next}$ then
17                     $flag \leftarrow true$
18                 else
19                     if Update $U(\boldsymbol{p}) == true$ then
20                         add $\mathbf{p}$ to $L_{next}$
21                         $flag \leftarrow true$
22             if $flag == false$ then
                   /* Check other adjacent neighbor nodes for reactivation */
23                 forall $\boldsymbol{x}_{nb}$ adjacent to $\boldsymbol{x}$ do
24                     if $U(\boldsymbol{x}_{nb}) > U(\boldsymbol{x})$ and $\boldsymbol{x}_{nb} \notin L_{next}$ then
25                         if Update $U(\boldsymbol{x}_{nb}) == true$ then
26                             add $\mathbf{x}_{nb}$ to $L_{next}$
27     $L \leftarrow L_{next}$
```



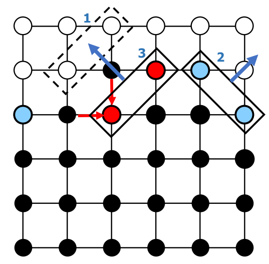
(a) iter 1



(b) iter 2

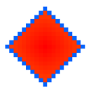


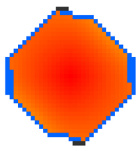
(c) iter 3 on FIM

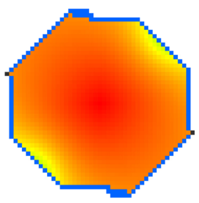


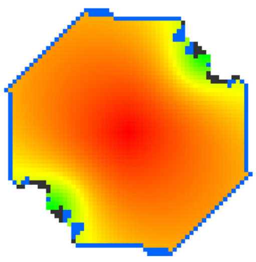
(d) iter 3 on COFIM

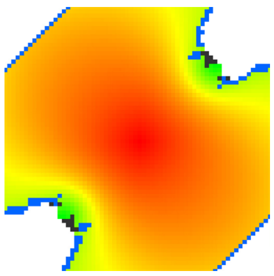
Figure 33: Difference between FIM and COFIM. In FIM, group 1 and 3 exist in the active list at the same time. However, COFIM can disintegrate group 1.


(a) iter 10


(b) iter 20


(c) iter 30


(d) iter 40


(e) iter 50

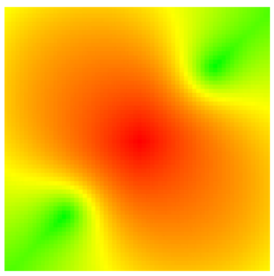

(f) final

Figure 34: Color-coded distance to the center on sinusoidal domain ($f = 6{+}5\sin(2\pi x){*}\sin(2\pi y)$). (a)-(e) describe the intermediate result of COFIM in the iteration. Red and green represent the nearest and farthest distances to the seed point, respectively. Blue and black nodes are included in the active list. Black nodes are unconverged, so they are re-computed in the next iteration.

Integration with Group-Ordered FIM

An approach to solve the redundant computation of FIM using casual dependency has been tried in other studies. Group-Ordered FIM (GOFIM, Section V) is employs a group-based updating scheme where the updating order is determined by the solution on a coarse level grid. GOFIM computes the distance map on a coarser grid, which is later used as a causality map between blocks, and we update the nodes based on this order. Therefore, COFIM and GOFIM suggest a completely different way of using casual dependency. The first is based on a node-level approach and the second on a group-level approach.

GOFIM can effectively reduce redundant iterative cal- culation of FIM in complex speed maps. However, several shortcomings of GOFIM are also suggested in the paper. GOFIM's performance is highly dependent on the size of the block. Smaller blocks are more advantageous to reflect the characteristics of the input speed map. At the same time, however, small blocks create more overhead. So GOFIM requires a trade-off. A user can use large blocks for a relatively simple speed map, and smaller blocks for more complex speed maps. However, the complexity of the input speed map can be difficult to identify in advance.

When using large blocks in GOFIM, it's no surprise that the performance slows down due to complex velocity maps. GOFIM reduces the number of duplicate calculations by using dependencies between blocks, but there is no solution to avoid duplicate calculations inside blocks. However, a new algorithm that integrates GOFIM and COFIM can produce the best results in most cases. In this section, we propose this new algorithm, called Causality and Group-Ordered FIM(CGOFIM) for the eikonal equation.

The proposed method takes advantage of GOFIM, which exploits the causality of grid blocks, to reduce redundant computations, but it also introduces causality order at the node level to further reduce duplicate calculations. This makes it easy to use large blocks when handling coarse maps. This is because CGOFIM can solve duplicate calculation problems in inner blocks caused by large blocks by introducing node-level causality ordering. On the other hand, CGOFIM still can take advantage of the reduction of overhead using large blocks.

Figure 35 is a pipeline description of CGOFIM. This is di- vided into two steps. One is the preliminary task of defining groups using a solution in the coarse grid. The other is the main computation to get a solution to the actual fine grid using this group. The number of the group K also affects the entire performance. Having too many groups generates a lot of overhead, but having too few groups does not apply proper ordering between groups. In Section V, we suggested that the proper K value is about 100 to 250. In this paper, after groups were defined, we created an initial active list for each group. To get an initial active list for each group, we used the tight-bound algorithm. This algorithm is a heuristic method for selecting nodes belonging to *true* upwind nodes among many boundary nodes between groups. It first defines a loose bound list using a node belonging to the boundary between groups. After that, it updates a node belonging to the loose bound twice with the Jacobi scheme and checks whether it has

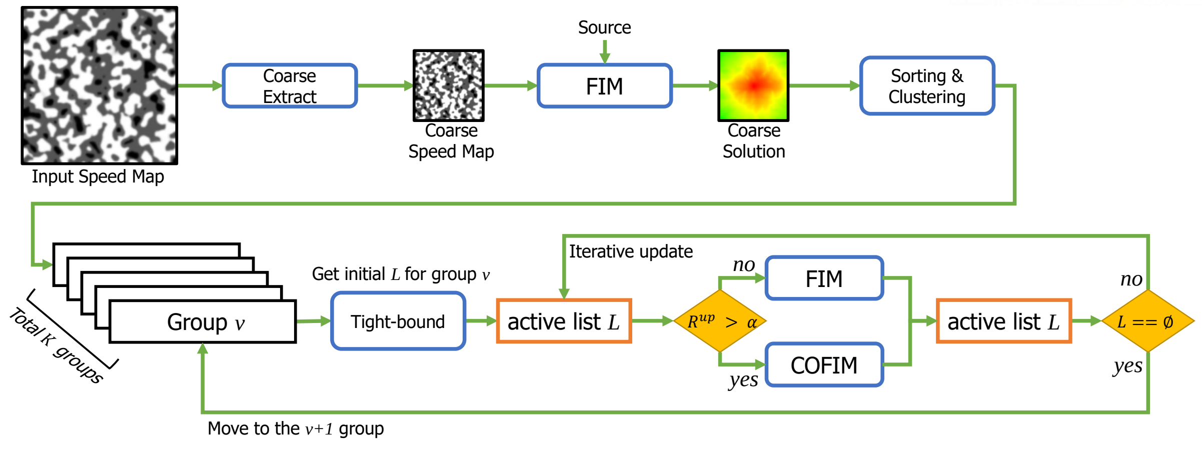


Figure 35: Pipeline of Causality Group Ordered FIM.

converged. If a node belongs in the tight-bound algorithm, it must be converged. So, we selected a node as an initial active list for a group. After obtaining an initial active list for the group, we performed the main algorithm of FIM. (In this time, we prevented an active list from going out of the group.) When all nodes in the group converged (or the active list became empty set), we moved to the next group and repeated the above process.

Here, we can replace the main iteration for solving the active list with COFIM instead of FIM. CGOFIM can complete by simply replacing this part. However, we added one more condition ($R^{up} > \alpha$) so that we could use both of the main loop FIM and COFIM interchangeably. A more detailed discussion regarding the rate is given in the next section.

Reactivation Rate and Casualty Ordering

Causality Ordered FIM is useful to reduce redundant computation by using parent and child relationships between nodes. However, this is ineffective in simple examples such as Figure 1, which required little to no redundant computation. Because COFIM examines the parent node when expanding the active list, this process works as a pure overhead in simple speed maps. Therefore, we propose a condition for using both COFIM and FIM interchangeably.

We defined R^{up} as the rate of updated nodes in the active list L (or re-added to the list L_{next}). If the speed map is complex, many nodes in the active list will be reactivated (so it has high R^{up} value), and it causes a lot of redundant computation. Conversely, an active list in a simple speed map may have a low re-activation rate. So, it doesn't reduce duplicated computation by casualty ordering. Therefore, we use COFIM if the R^{up} value is above a certain threshold. This allows us to avoid the overhead by introducing COFIM. The discussion of the threshold for R^{up} is covered in the results section.

Algorithm 9: Causality Group Ordered FIM

```
   Input: Grid $\Omega$, Solution $U$, Active list $L$
   /* Coarse grid level */
1  Generate and initialize coarse grid $\widetilde{\Omega}$ from $\Omega$
2  Get solution for $\widetilde{\Omega}$ using FIM
3  Cluster blocks in $\widetilde{\Omega}$ into $k$ groups ($G_1$ to $G_k$)
   /* Fine grid level */
4  Initialize $U$ and $L$
5  $G = \emptyset$
6  for $v = 1$ to $k$ do
7      $G = G \cup G_v$
8      if $v$ is 1 then
9          $L_{upper} \leftarrow L$
10     else
11         Get $L_{upper}$ from $G$ and $G_v$ (see Equation 4 in Section IV)
12     Get tight active list $\tilde{L}_{tight}$ from $L_{upper}$ using Algorithm 6
13     $L \leftarrow \tilde{L}_{tight}$
14     while $L$ is not empty do
15         if $R^{up} > \alpha$ then
16             Solve for $L$ using main loop in Algorithm 8
17         else
18             Solve for $L$ using main loop in Algorithm 1
```

6.3 Implementation

We built our system using C++, OpenMP. In this paper, we focus on the reduction of computational cost using casual dependency, so we describe all methods based on a single-threaded system. However, the proposed COFIM and CGOFIM methods inherently have the possibility of parallelization. For efficient parallelization implementation in shared-memory systems, we use the lock-free method, which is proposed in Section V. The feature of a lock-free parallel implementation is that each thread maintains a local buffer (for thread i, make local active list L_i) instead of a global active list shared for all threads. However, if we use multiple local active lists, then we need a method that prevents a node from being inserted into multiple local lists. To avoid this, we can manage a flag for each node to check whether a node is currently in any active list. In lock-free implementation, this flag is managed safely without lock synchronization through a fetch-and modify atomic operation.

6.4 Results and Discussion

We evaluate the performance of our method and compare it with the Fast Marching Method (FMM) [24], the Fast Sweeping Method (FSM) [25], the Fast Iterative Method (FIM) [19], the Group-Ordered FIM (GOFIM) (Section V), and the parallel Heap Cell Method [66]. For FSM implementation, we chose Detrixhe et al.'s parallel FSM (DFSM) [32] among many improved FSM algorithms because of its highest parallel efficiency. We use various block size configurations for hybrid methods such as CGOFIM and pHCM. For example, CGOFIM4 and pHCM4 are implemented on 4^3 block, while GOFIM16 uses 16^3 block. All experiments were conducted on a Non Uniform Memory Access system equipped with two Intel Xeon E5-2630 v4 deca-core processors and 128 GB of DDR4 main memory. We implemented the experiment code in C++ and OpenMP with -O3 level optimization with gcc 5.4.0, and all floating point computations were performed in 64 bit double precision. For measuring serial computing performance and parallel scalability, we tested 1 thread up to 20 parallel threads. We checked the average running time of each method including the initialization step. The input speed map was stored in a hard disk, and its loading time was excluded from the running time measurement.

We tested the methods on the various types of speed maps, ranging from a plain constant speed map to a complex shape map. The definition of each input speed map is as follows.

Map 1. $f = 1$. Constant speed map.

Map 2. $f = \frac{1}{4}, \frac{1}{2}, 1$. Speed map with three layers of different speed values.

Map 3. $f = 6 + 5\sin(2\pi x) * \sin(2\pi y) * \sin(2\pi z)$.

Map 4. $f = 1 + 0.5\sin(20\pi x) * \sin(20\pi y) * \sin(20\pi z)$.

Map 5. $f =$ Spatially coherent random speed map.

where all speed maps are defined in the normalized domain $\Omega = [0, 1]$.

Map 1 is the constant speed map; all the speed values are identical. Map 2 has three speed variations, which represent wave propagation through three different materials. Map 3 and 4 are arbitrary sinusoidal speed maps; the complexity of each map is controlled by the period of the sinusoidal function. Map 5 is a spatially correlated random speed map so that speed values vary globally. This dataset is very challenging because characteristic paths frequently turn their directions. The datasets are chosen so that different levels of complexity can be tested. In all experiments, we used a 512^3 three-dimensional grid. Speed maps are pre-computed and stored in each grid point. In the case of GOFIM and CGOFIM, the performance was affected by the number of clusters; in this section, we used 240 groups for all block sizes. The threshold to employ causality ordering is set to 0.3 for all experiments. Figure 36 shows the color-coded distance map with iso-contours for the solution of the eikonal equation for each speed map.

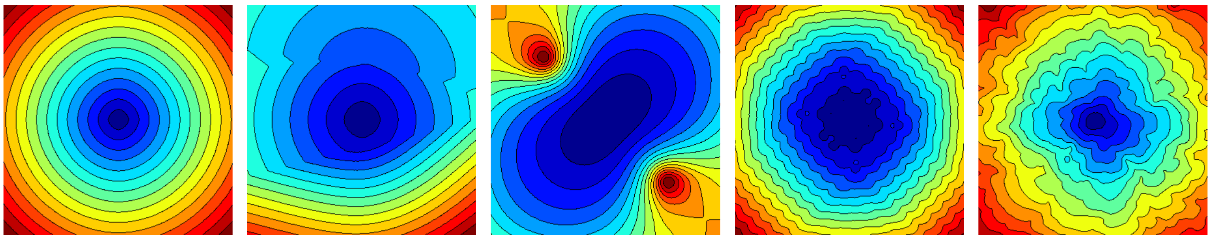

(a) Map 1 (b) Map 2 (c) Map 3 (d) Map 4 (e) Map 5

Figure 36: Color-coded distance map with iso-contours of our test datasets visualized in 2D. Blue and red represent the nearest and farthest distances to the seed point, respectively. A single source was used here.

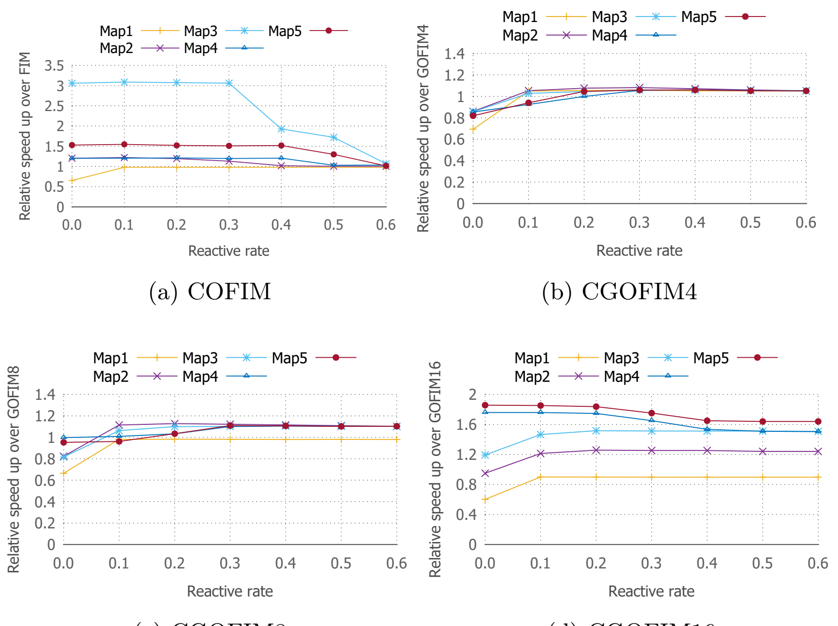
Single-threaded result with a single center source

The first experiment used a center source (in the normalized domain 0.5, 0.5, 0.5). Table 8 demonstrates the running times and average computation number (AvC) per node for each solution.

		FMM	DFSM	FIM	GOFIM4	GOFIM8	GOFIM16	COFIM	CGOFIM4	CGOFIM8	CGOFIM16
Map 1	Time	83.56	92.02	**21.63**	48.54	35.74	24.37	21.70	42.86	36.47	25.21
	AvC	2.99	9.00	**2.00**	3.55	2.76	2.18	2.00	3.52	2.76	2.37
Map 2	Time	84.35	921.17	64.51	53.59	35.30	32.42	55.40	49.23	32.02	**27.05**
	AvC	2.99	103.00	5.39	4.11	3.28	3.49	4.43	3.98	3.20	**3.04**
Map 3	Time	88.09	554.84	244.14	54.69	34.99	42.37	80.43	50.73	32.67	**29.52**
	AvC	2.99	62.00	21.11	4.12	3.31	4.67	6.86	4.10	**3.30**	3.40
Map 4	Time	93.00	1024.23	127.46	74.73	81.03	121.39	106.51	**69.43**	74.32	78.97
	AvC	2.99	110.00	10.51	5.16	6.82	10.29	8.69	**5.14**	6.81	7.97
Map 5	Time	96.15	923.47	169.97	75.29	81.38	145.53	112.37	**69.85**	74.78	89.84
	AvC	2.99	99.00	14.58	4.93	6.61	15.22	9.44	**4.88**	6.56	9.38

Table 8: Running time using different methods measured in seconds. AvC is the average computation number per node. The fastest time and the lowest AvC for each dataset are marked in boldface.

FMM is the worst-case optimal solution and the performance of it is almost not affected by the complexity of the input speed map. In contrast, FSM and FIM are iterative algorithms; their execution times are often affected by the complexity of the speed maps. Map 1 is the simplest case. Here, the iterative update does not occur in both FIM and FSM. FIM requires only two calculations per node and shows the best performance. Furthermore, GOFIM is an algorithm that reduces the redundant calculation of FIM by using coarse-level causality ordering. It uses a coarse level-map that becomes more granular as smaller block sizes are used. Pre-computation for the coarse-level map and clustering for coarse-level causality ordering are all classified as overhead: the smaller the block size, the more overhead. In Map 1, which does not require iterative calculations, GOFIM shows lower performance than FIM because of this overhead.



(a) COFIM (b) CGOFIM4

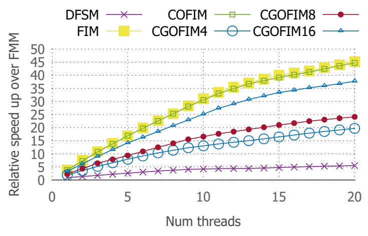
(c) CGOFIM8 (d) CGOFIM16

Figure 37: Experiment using different threshold values to employ causality ordering. (a) to (d) describe the relative speed up when causality ordering is applied to each original method. The x-axis is a list of the threshold values.

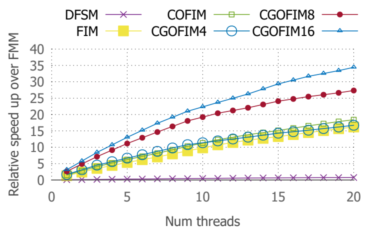
In other maps, GOFIM performs better than FIM. In particular, GOFIM16, which uses large blocks for Map 2, is the better choice, while GOFIM4 is better in Map 4 and Map 5. GOFIM controls coarse-level ordering but does not reduce the iterative calculation inside the block. In the case of Map 2, which is relatively simple, there are not many iterations inside a block even with a large block size. However, in Map 4 and Map 5, which are more complex cases, each block requires a large number of iterations. Therefore, there is not much difference between FIM and GOFIM16. However, GOFIM8, which is halfway between GOFIM4 and GOFIM16, has a lower overhead than GOFIM4m and a better performance than GOFIM16 on Maps 3, 4, and 5.

COFIM applies a node-level causality ordering to FIM. CGOFIM has both a coarse-level ordering in GOFIM and a node-level causality ordering in COFIM. In each case, the method that applies causality ordering usually shows better performance. In particular, COFIM offers a significant performance improvement over FIM. In Map 3, the speed of COFIM is about three times that of FIM. In addition, Map 5 shows 50 percent improvement using COFIM, and Maps 2 and 4 show a 15 percent improvement using COFIM. CGOFIM16 also shows a notable performance improvement over GOFIM16 because it reduces the number of iterations within a block using node-level causality ordering. CGOFIM4 has limited improvement compared to GOFIM4. In the 4x4x4 blocks, there are not many iterations even with complex speed maps. Nevertheless, this is still effective because node-level causality ordering reduces the iteration computation of FIM and has good compatibility with GOFIM.

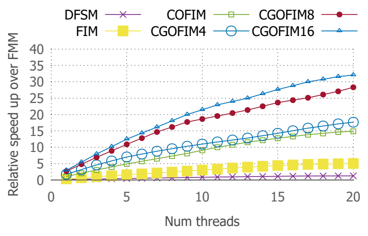
The performance of causality ordering is affected by the frequency of use. Figure 37 describes



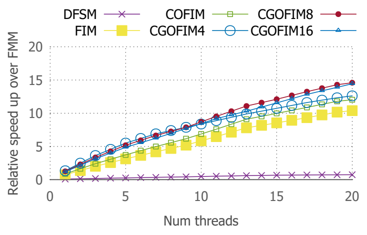
(a) Map 1



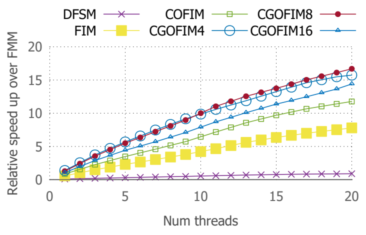
(b) Map 2



(c) Map 3



(d) Map 4



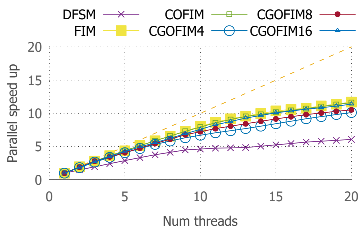
(e) Map 5

Figure 38: Parallel running time results with a single center source. The horizontal axis is the number of parallel threads (up to 20), and the vertical axis is the relative speed up of each solver over the single-threaded FMM.

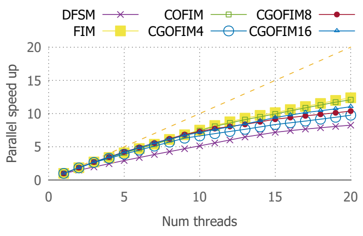
the performance of causality ordering according to the threshold. In Figure 37, each figure describes the relative speed up when causality ordering is applied to each original method. The x-axis is a list of the threshold values. In the case of 0.0, causality ordering is always applied. When the threshold value increases, the frequency of causality ordering is reduced. In the case of Map 1 with a single center source, any node in the active list is not reactivated. Here, recalculation of a parent node for causality ordering is completely unnecessary. So, if we always enable causality ordering, we get only 65 percent performance over the original FIM (Figure 37 (a)). However, performance may also be limited if causality ordering isn't used often enough. For example, if we decided to use causality ordering when at least 30 percent of the nodes in the active list were to be reactivated, then the performance of COFIM would be three times better than the original FIM. However, if we were to use 40 percent reactivate value, the performance gain is only twice that of the original FIM. CGOFIM16 is also affected by the value of the reactivate rate. On the other hand, in the case of CGOFIM4 and CGOFIM8, there is no big difference in performance when using node-level causality ordering. Through many experiments, we found that the 30 percent ratio was appropriate in most cases, so we used this value as the threshold to employ causality ordering.

Multi-threaded results with a single center source

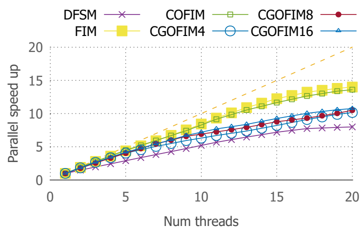
Figures 38 and 39 demonstrate the multi-threaded results and parallel scalability for each solver. We observed that the parallel scalability of DFSM is about $6 \sim 8\times$ with 20 threads. However,



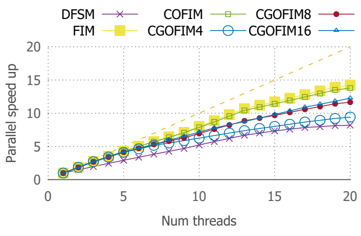
(a) Map 1



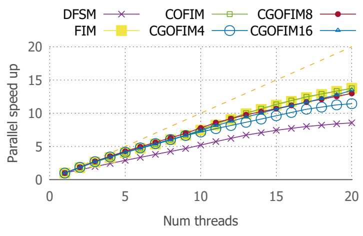
(b) Map 2



(c) Map 3



(d) Map 4



(e) Map 5

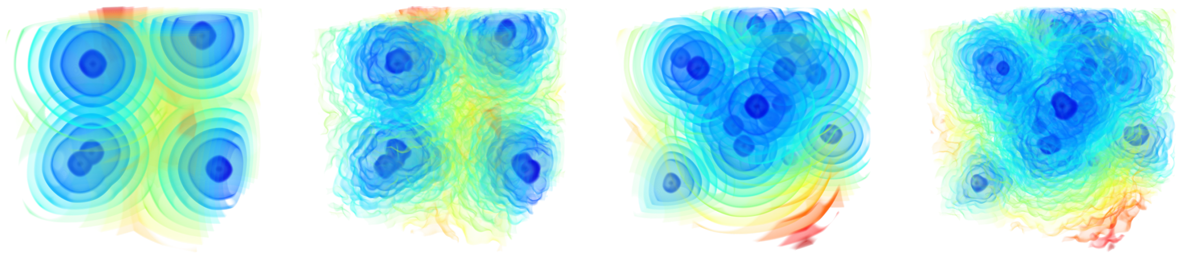
Figure 39: Parallel scalability results with a single center source. The horizontal axis is the number of parallel threads (up to 20), and the vertical axis is the speed up factor (i.e., scalability) of each solver.

DFSM performs much lower than FMM in a single-threaded result. As a result, DFSM shows limited performance improvement over a single-threaded FMM even with parallelization.

A multi-threaded implementation method for FIM and GOFIM is proposed in Section V. The proposed COFIM and CGOFIM share the same parallelization method, so they have almost the same parallel efficiency compared to the original method. The best scalability result for FIM and COFIM is 14 speed up for 20 threads. In Map 1, the performances of FIM and COFIM are about 45 speed up compared to a single-threaded FMM. Similar to the single-threaded result, the parallel performance of COFIM is mostly better than FIM. For CGOFIM, larger block sizes offer better scalability. Especially in the case of Map 4 and Map 5, CGOFIM4 has comparatively less scalability than other versions of CGOFIM. CGOFIM4 was even the best solution for the single-threaded result in Maps 4 and 5. However, CGOFIM8 was the best solution for multi-threaded results. Overall, we observed that CGOFIM16 was the best option for most cases. Not only did it outperform other methods in Maps 2 and 3, which are relatively simple cases, but there was also no significant difference compared to CGOFIM4 and CGOFIM8 in Maps 4 and 5 because of its high scalability.

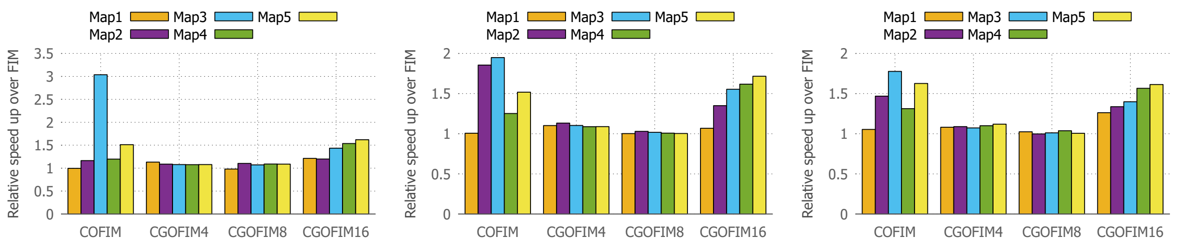
Experimental results with multiple sources

The performance of FIM can be affected by the location of the source or the type of source, in addition to the speed map. As such, we tested our node-level causality ordering method in two more experiments that had a different source location and type. One experiment used



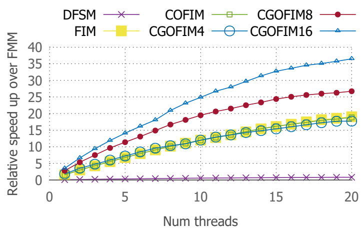
(a) 5 sources, Map 1 (b) 5 sources, Map 5 (c) 20 sources, Map 1 (d) 20 sources, Map 5

Figure 40: Color-coded distance map with iso-contours of our test datasets visualized in 3D. Blue and red represent the nearest and farthest distances to the seed point, respectively. (a) and (b) used five multiple sources, and (c) and (d) used twenty multiple sources.

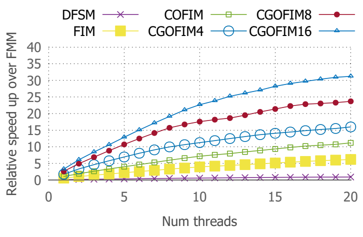


(a) 1 point (b) 5 points (c) 20 points

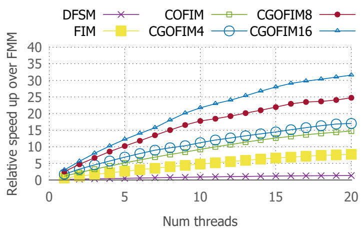
Figure 41: Single-threaded running time results with different sources. (a) used a single center source, and (b) and (c) used multiple sources. The vertical axis is the relative speed up of each solver over the original FIM or GOFIM method.



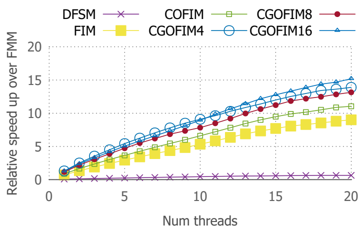
(a) Map 1



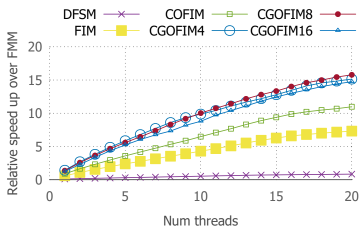
(b) Map 2



(c) Map 3



(d) Map 4



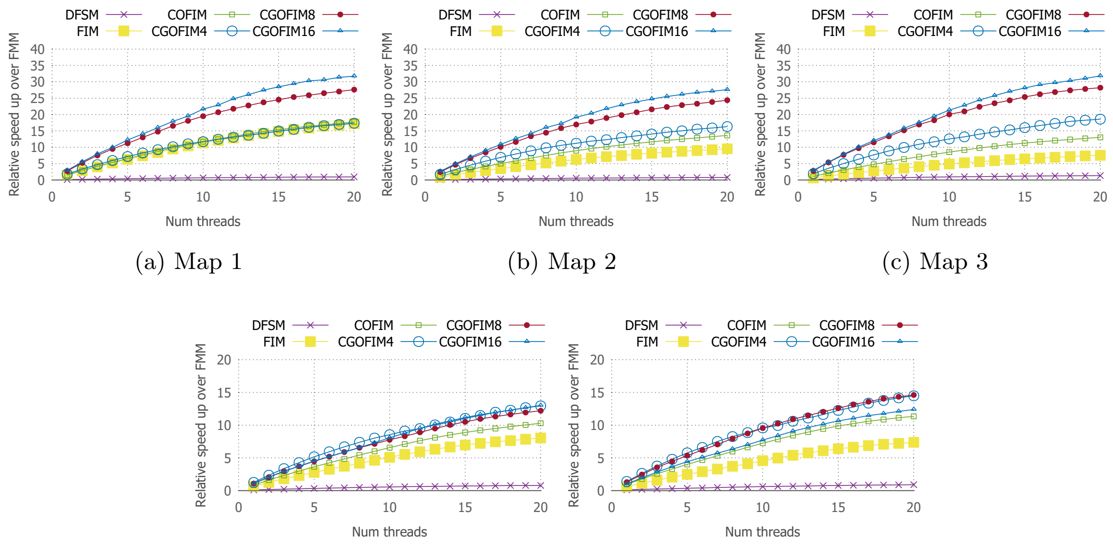
(e) Map 5

Figure 42: Parallel running time results with 5 multiple sources. The horizontal axis is the number of parallel threads (up to 20), and the vertical axis is the relative speed up of each solver over the single-threaded FMM.

five source points and the other used 20 source points. These points are irregularly distributed across the domain. Figure 40 is a 3D visualization for the location of source points in the domain using Maps 1 and 5. Figure 41 describes the performance of causality ordering (compared to original FIM and GOFIM) according to the source types. In Figure 41, COFIM shows significant performance improvements in Map 3 for all source types. Regardless of the source type, COFIM on average offers 50 percent improvement over original FIM. The improvement of CGOFIM16 is also over 40 percent compared to GOFIM16. On the other hand, CGOFIM4 improved by 10 percent and CGOFIM8 improved by 3 percent compared to GOFIM4 and GOFIM8. Overall improvement seems to be largely affected by the characteristics of the map rather than source type. Multi-threaded results are demonstrated as relative performance over FMM (Figure 42, Figure 43). Even in Map 1, which is the simplest speed case, iterative calculations occur in FIM if there exist multiple sources due to interference between different wavefronts. Therefore, when using multiple sources, FIM can no longer be considered as the best solution, even in Map 1. CGOFIM has similar results when using a single source. Overall, we found that CGOFIM16 is the best method for Maps 1, 2, and 3 in every source type. However, CGOFIM8 is considered the better solution for Map 5.

Single-threaded results with different grid size

The performance of both the serial and parallel algorithm is often affected by data size. To check the data-size effect, we measured the running time of each solver on four different grid



(a) Map 1 (b) Map 2 (c) Map 3

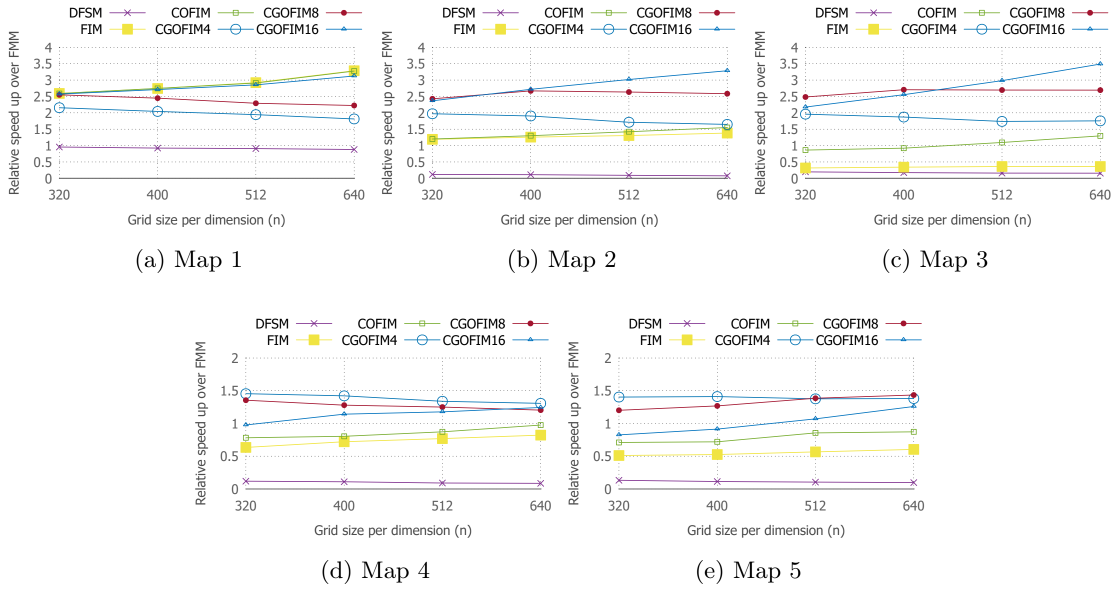
(d) Map 4 (e) Map 5

Figure 43: Parallel running time results with 20 multiple sources. The horizontal axis is the number of parallel threads (up to 20), and the vertical axis is the relative speed up of each solver over the single-threaded FMM.

sizes (320^3, 400^3, 512^3, 640^3). Figure 44 demonstrates a single-threaded running time result for different grid sizes. GOFIM and CGOFIM have an overhead to managing coarse-level solutions and block ordering, and they become larger as the domain size increases. In particular, the performance of CGOFIM4 over FMM reduces when domain size increase. CGOFIM4, which uses 4^3 blocks, has the burden of managing 4 million coarse blocks for 640^3 domains. But CGOFIM16 is more flexible than. Even in the 640^3 domain, it has only 64,000 coarse blocks. Therefore, CGOFIM16 is more competitive when the domain size increases.

Comparison with pHCM

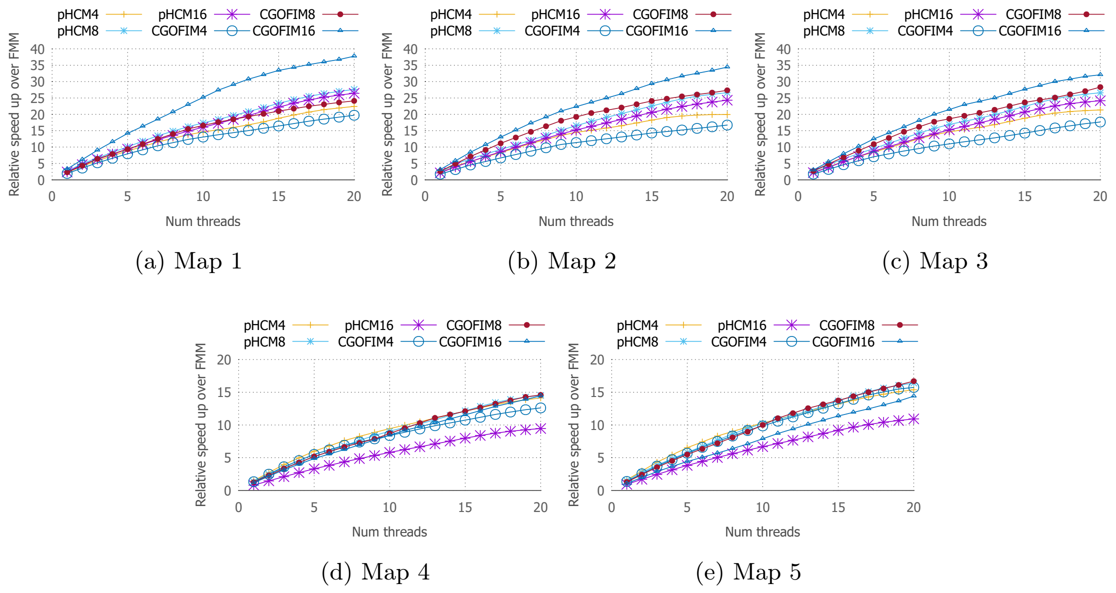
The final experiment compared the parallel Heap Cell Method (pHCM) [66] to the other methods. In this part, we also used a center source (in the normalized domain 0.5, 0.5, 0.5). pHCM is a hybrid solution that is similar to CGOFIM. However, there are important differences that make the two methods perform differently. While CGOFIM uses static coarse ordering, pHCM manages the computation order of cells dynamically using a heap for coarse grids. In addition, when updating a cell in pHCM, the iterative update continues until all nodes in the cell become converged. Therefore, using large-size cells is not a good approach when using pHCM. While only a single node needs to be updated in a cell, the entire node in the cell must be updated too. In our test cases, pHCM shows the best result for a cell size of 8. In Maps 1, 2, and 3, CGOFIM16 outperformed compared to pHCM. In addition, CGOFIM8 has a similar performance to pHCM in Maps 4 and 5.



(a) Map 1 (b) Map 2 (c) Map 3

(d) Map 4 (e) Map 5

Figure 44: Single threaded running time results with different sizes. The horizontal axis is the size of each dimension n (a grid size is n^3. The vertical axis is the relative speed up of each solver over the single-threaded FMM.



(a) Map 1 (b) Map 2 (c) Map 3

(d) Map 4 (e) Map 5

Figure 45: Comparison between pHCM and CGOFIM. Parallel running time results with a single center source. The horizontal axis is the number of parallel threads (up to 20), and the vertical axis is the relative speed up of each solver over the single-threaded FMM.

6.5 Summary

In this section, we proposed a novel extension method of causality ordering for a fast iterative method series that applies causality information at a fine-node level. The proposed method efficiently prevents the situation in which nodes having dependency on each other belong to the narrow band at the same time. Moreover, we propose a hybrid algorithm with the Group-Order approach to reduce computation cost at both the coarse and fine levels. The proposed methods are compatible with lock-free implementation and achieved high parallel efficiency in a shared memory system. In the future, we plan to extend our work to distributed systems. We would like to explore how our results can be applied in real-world applications.

VII Conclusion

In this dissertation, I described novel parallel algorithms for the eikonal equation. The main motivation of this research was to overcome the main drawback of a fast iterative method by exploiting the causality information and apply it to the modern parallel system. The overall performance gain is very impressive. For the multi-GPU extension, the proposed method showed an increase in the parallel speed of an overall 6.2 times and a maximum of up to 6.6 times on eight GPUs. In addition, the proposed parallel methods achieved a speed-up of 15 to 45 times as compared to the single-threaded CPU version. This dissertation has addressed the problems in both science and engineering disciplines, designing a novel algorithm to reduce redundant iterative computation and maximizing the computing potential for modern parallel computing systems.

7.1 Summary of Dissertation Research

In Section IV, I present a novel on-the-fly adaptive domain decomposition algorithm for parallel BlockFIM on multi-GPU systems. It is based on a history-based active list prediction method. The proposed model successfully predicted the future computing domain and distributed tasks evenly across GPUs. In addition, I proposed several optimization methods for a multiGPU implementation and a locality-aware clustering algorithm to minimize the inter-GPU data communication. The experimental results showed that the proposed method achieved greater performance improvements in most cases when compared to the regular domain decomposition methods.

I proposed two parallel eikonal solvers, lock-free FIM and Group-Ordered FIM, in Section V. Lock-free FIM is an extension of the original FIM for efficient parallelization on shared memory systems, and GO-FIM further improves the performance of parallel FIM by using a rough ordering of blocks on a coarse grid and a clustering of blocks to reduce the iteration numbers and to increase the parallelism.

A novel parallel algorithm introducing a node-level causality dependency to FIM is proposed in Section VI. In particular, I propose a novel algorithm called Causality and Group-Ordered FIM (CGO-FIM). The proposed method takes advantage of GOFIM that exploits the causality of grid blocks to reduce redundant computations; it also reduces duplicate calculations by introducing the causality order at the node level. The experimental results showed that the proposed method mapped well on a shared memory system and outperformed the widely used parallel eikonal equation solvers in many cases.

References

[1] S. Geoltrain and J. Brac, "Can we image complex structures with first-arrival traveltime?" *Geophysics*, vol. 58, no. 4, pp. 564–575, 1993.

[2] P. Podvin and I. Lecomte, "Finite difference computation of traveltimes in very contrasted velocity models: a massively parallel approach and its associated tools," *Geophysical Journal International*, vol. 105, no. 1, pp. 271–284, 1991.

[3] T. Gillberg, "A semi-ordered fast iterative method (SOFI) for monotone front propagation in simulations of geological folding," The 19th International Congress on Modelling and Simulation (MODSIM2011), December 2011.

[4] E. Rouy and A. Tourin, "A viscosity solutions approach to shape-from-shading," *SIAM Journal on Numerical Analysis*, vol. 29, pp. 867–884, 1992.

[5] R. Kimmel and J. A. Sethian, "Optimal algorithm for shape from shading and path planning," *Journal of Mathematical Imaging and Vision*, vol. 14, no. 3, pp. 237–244, 2001.

[6] R. Malladi and J. Sethian, "A unified approach to noise removal, image enhancement, and shape recovery," *IEEE Trans. on Image Processing*, vol. 5, no. 11, pp. 1554–1568, 1996.

[7] R. Tsai, S. Osher *et al.*, "Level set methods and their applications in image science," *Communications in Mathematical Sciences*, vol. 1, no. 4, pp. 1–20, 2003.

[8] R. Kimmel and J. A. Sethian, "Computing geodesic paths on manifolds," *Proceedings of the national academy of Sciences*, vol. 95, no. 15, pp. 8431–8435, 1998.

[9] A. Valero-Gomez, J. V. Gomez, S. Garrido, and L. Moreno, "The path to efficiency: Fast marching method for safer, more efficient mobile robot trajectories," *IEEE Robotics & Automation Magazine*, vol. 20, no. 4, pp. 111–120, 2013.

[10] O. Weber, Y. S. Devir, A. M. Bronstein, M. M. Bronstein, and R. Kimmel, "Parallel algorithms for approximation of distance maps on parametric surfaces," *ACM Trans. Graph.*, vol. 27, no. 4, pp. 104:1–104:16, Nov. 2008. [Online]. Available: http://doi.acm.org/10.1145/1409625.1409626

[11] J. A. Sethian, *Level set methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science.* Cambridge university press, 1999, vol. 3.

[12] I. Ihrke, G. Ziegler, A. Tevs, C. Theobalt, M. Magnor, and H.-P. Seidel, "Eikonal rendering: Efficient light transport in refractive objects," *ACM Transactions on Graphics (TOG)*, vol. 26, no. 3, p. 59, 2007.

[13] J. A. Sethian and M. A. Popovici, "3-d traveltime computation using the fast marching method," *Geophysics*, vol. 64, pp. 516–523, 1999.

[14] N. Rawlinson and M. Sambridge, "Wave front evolution in strongly heterogeneous layered media using the fast marching method," *Geophysical Journal International*, vol. 156, no. 3, pp. 631–647, 2004.

[15] S. Li, A. Vladimirsky, and S. Fomel, "First-break traveltime tomography with the double-square-root eikonal equation," *Geophysics*, vol. 78, no. 6, pp. U89–U101, 2013.

[16] C. Lenglet, M. Rousson, R. Deriche, and O. Faugeras, "Statistics on the manifold of multivariate normal distributions: Theory and application to diffusion tensor mri processing," *Journal of Mathematical Imaging and Vision*, vol. 25, no. 3, pp. 423–444, 2006.

[17] W.-K. Jeong, P. T. Fletcher, R. Tao, and R. Whitaker, "Interactive visualization of volumetric white matter connectivity in DT-MRI using a parallel-hardware Hamilton-Jacobi solver," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1480–1487, Nov. 2007. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2007.70571

[18] J. J. Helmsen, E. G. Puckett, P. Colella, and M. Dorr, "Two new methods for simulating photolithography development in 3d," in *Optical Microlithography IX*, vol. 2726. International Society for Optics and Photonics, 1996, pp. 253–261.

[19] W.-K. Jeong and R. T. Whitaker, "A fast iterative method for eikonal equations," *SIAM J. Sci. Comput.*, vol. 30, no. 5, pp. 2512–2534, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1137/060670298

[20] J. A. Sethian, "Fast marching methods," *SIAM Review*, vol. 41, no. 2, pp. 199–235, 1999.

[21] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, "Network flows," 1988.

[22] D. P. Bertsekas, *Network optimization: continuous and discrete models.* Athena Scientific Belmont, 1998.

[23] D. P. Bertsekas, D. P. Bertsekas, D. P. Bertsekas, and D. P. Bertsekas, *Dynamic programming and optimal control.* Athena scientific Belmont, MA, 1995, vol. 1, no. 2.

[24] J. A. Sethian, "A fast marching level set method for monotonically advancing fronts," in *Proc. Natl. Acad. Sci.*, vol. 93, February 1996, pp. 1591–1595.

[25] H. Zhao, "A fast sweeping method for eikonal equations," *Mathematics of Computation*, vol. 74, pp. 603–627, 2004.

[26] Y.-H. R. Tsai, L.-T. Cheng, S. Osher, and H.-K. Zhao, "Fast sweeping algorithms for a class of hamilton–jacobi equations," *SIAM journal on numerical analysis*, vol. 41, no. 2, pp. 673–694, 2003.

[27] J. Qian, Y.-T. Zhang, and H.-K. Zhao, "A fast sweeping method for static convex hamilton–jacobi equations," *Journal of Scientific Computing*, vol. 31, no. 1-2, pp. 237–271, 2007.

[28] E. W. Dijkstra *et al.*, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[29] P. Lesaint and P.-A. Raviart, "On a finite element method for solving the neutron transport equation," *Publications mathématiques et informatique de Rennes*, no. S4, pp. 1–40, 1974.

[30] M. Boué and P. Dupuis, "Markov chain approximations for deterministic control problems with affine dynamics and quadratic cost in the control," *SIAM Journal on Numerical Analysis*, vol. 36, no. 3, pp. 667–695, 1999.

[31] H. Zhao, "Parallel implementations of the fast sweeping method," *Journal of Computational Mathematics*, vol. 25, no. 4, pp. 421–429, 2007.

[32] M. Detrixhe, F. Gibou, and C. Min, "A parallel fast sweeping method for the eikonal equation," *J. Comput. Phys.*, vol. 237, pp. 46–55, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jcp.2012.11.042

[33] T. Gillberg, M. Sourouri, and X. Cai, "A new parallel 3d front propagation algorithm for fast simulation of geological folds." in *ICCS*, ser. Procedia Computer Science, vol. 9. Elsevier, 2012, pp. 947–955.

[34] E. Krishnasamy, M. Sourouri, and X. Cai, "Multi-gpu implementations of parallel 3d sweeping algorithms with application to geological folding," *Procedia Computer Science*, vol. 51, pp. 1494–1503, 2015.

[35] M. Detrixhe and F. Gibou, "Hybrid massively parallel fast sweeping method for static hamilton–jacobi equations," *Journal of Computational Physics*, vol. 322, pp. 199–223, 2016.

[36] J. Yang and F. Stern, "A highly scalable massively parallel fast marching method for the eikonal equation," *Journal of Computational Physics*, vol. 332, pp. 333–362, 2017.

[37] W.-K. Jeong and R. T. Whitaker, "A fast iterative method for a class of hamilton-jacobi equations on parallel systems," 2007.

[38] S. Hong and W. K. Jeong, “A group-ordered fast iterative method for eikonal equations,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 318–331, Feb 2017.

[39] S. Hong and W.-K. Jeong, “A multi-GPU fast iterative method for eikonal equations using on-the-fly adaptive domain decomposition,” *Procedia Computer Science*, vol. 80, pp. 190 – 200, 2016, international Conference on Computational Science 2016, {ICCS} 2016, 6-8 June 2016, San Diego, California, {USA}. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050916306676

[40] J. Vidale, “Finite-difference calculation of traveltimes,” *Bulletin of the Seismological Society of America*, vol. 78, no. 6, pp. 2062–2076, Dec. 1988. [Online]. Available: http://www.bssaonline.org/cgi/content/abstract/78/6/2062

[41] ——, “Finite-difference calculation of traveltimes in three dimensions,” *Geophysics*, vol. 55, no. 5, pp. 521–526, 1990.

[42] M. Falcone, T. Giorgi, and P. Loretti, “Level sets of viscosity solutions: Some applications to fronts and rendez-vous problems,” *SIAM Journal on Applied Mathematics*, vol. 54, no. 5, pp. 1335–1354, 1994.

[43] J. Tsitsiklis, “Efficient algorithms for globally optimal trajectories,” *IEEE Trans. on Automatic Control*, vol. 40, no. 9, pp. 1528–1538, September 1995.

[44] P. A. Gremaud and C. M. Kuster, “Computational study of fast methods for the eikonal equation,” *SIAM journal on scientific computing*, vol. 27, no. 6, pp. 1803–1816, 2006.

[45] C. Y. Kao, S. Osher, and J. Qian, “Lax-friedrichs sweeping scheme for static hamilton-jacobi equations,” *Journal of Computational Physics*, vol. 196, pp. 367–391, 2003.

[46] Y.-T. Zhang, S. Chen, F. Li, H. Zhao, and C.-W. Shu, “Uniformly accurate discontinuous galerkin fast sweeping methods for eikonal equations,” *SIAM Journal on Scientific Computing*, vol. 33, no. 4, pp. 1873–1896, 2011.

[47] J. Qian, Y.-T. Zhang, and H.-K. Zhao, “Fast sweeping methods for eikonal equations on triangular meshes,” *SIAM J. Numer. Anal.*, vol. 45, no. 1, pp. 83–107, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1137/050627083

[48] S. Bak, J. McLaughlin, and D. Renzi, “Some improvements for the fast sweeping method,” *SIAM J. Sci. Comput.*, vol. 32, no. 5, pp. 2853–2874, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1137/090749645

[49] L. C. Polymenakos, D. P. Bertsekas, and J. N. Tsitsiklis, “Implementation of efficient algorithms for globally optimal trajectories,” *IEEE Transactions on Automatic Control*, vol. 43, no. 2, pp. 278–283, 1998.

[50] M. Falcone, “A numerical approach to the infinite horizon problem of deterministic control theory,” *Applied Math. Optim.*, vol. 15, pp. 1–13, 1987.

[51] ——, “The minimum time problem and its applications to front propagation,” in *’Motion by Mean Curvature and Related Topics’, Proceedings of the International Conference at Trento, 1992.* New York: Walter de Gruyter, 1994.

[52] D. Bertsekas, F. Guerriero, and R. Musmanno, “Parallel asynchronous label-correcting methods for shortest paths,” *Journal of Optimization Theory and Applications*, vol. 88, no. 2, pp. 297–320, 1996. [Online]. Available: http://dx.doi.org/10.1007/BF02192173

[53] F. Bornemann and C. Rasch, “Finite-element discretization of static hamilton-jacobi equations based on a local variational principle,” *Computing and Visualization in Science*, vol. 9, no. 2, pp. 57–69, 2006. [Online]. Available: http://dx.doi.org/10.1007/s00791-006-0016-y

[54] Z. Fu, W.-K. Jeong, Y. Pan, R. M. Kirby, and R. T. Whitaker, “A fast iterative method for solving the eikonal equation on triangulated surfaces,” *SIAM J. Sci. Comput.*, vol. 33, no. 5, pp. 2468–2488, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1137/100788951

[55] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[56] M. Herrmann, “A domain decomposition parallelization of the fast marching method,” DTIC Document, Tech. Rep., 2003.

[57] M. Breuß, E. Cristiani, P. Gwosdek, and O. Vogel, “An adaptive domain-decomposition technique for parallelization of the fast marching method,” *Applied Mathematics and Computation*, vol. 218, no. 1, pp. 32–44, 2011.

[58] M. C. Tugurlan, “Fast marching methods-parallel implementation and analysis,” 2008.

[59] J. Yang, T. Michael, S. Bhushan, A. Hanaoka, Z. Wang, and F. Stern, “Motion prediction using wall-resolved and wall-modeled approaches on a cartesian grid,” in *Proc. of the 28th Symposium on Naval Hydrodynamics (USA, Pasadena)*, 2010.

[60] J. Yang, “An easily implemented, block-based fast marching method with superior sequential and parallel performance,” *SIAM Journal on Scientific Computing*, vol. 41, no. 5, pp. C446–C478, 2019.

[61] J. Weinbub and A. Hössinger, “Shared-memory parallelization of the fast marching method using an overlapping domain-decomposition approach,” in *Proceedings of the 24th High Performance Computing Symposium.* Society for Computer Simulation International, 2016, p. 18.

[62] G. Diamantopoulos, J. Weinbub, A. Hössinger, and S. Selberherr, "Evaluation of the shared-memory parallel fast marching method for re-distancing problems," in *2017 17th International Conference on Computational Science and Its Applications (ICCSA)*. IEEE, 2017, pp. 1–8.

[63] A. Shrestha and I. Senocak, "Multi-level domain-decomposition strategy for solving the eikonal equation with the fast-sweeping method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2297–2303, 2018.

[64] A. Chacon and A. Vladimirsky, "Fast two-scale methods for eikonal equations," *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. A547–A578, 2012.

[65] ——, "A parallel heap-cell method for eikonal equations," *arXiv preprint arXiv:1306.4743*, 2013.

[66] ——, "A parallel two-scale method for eikonal equations," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. A156–A180, 2015.

[67] Y. Saad, *Iterative methods for sparse linear systems.* siam, 2003, vol. 82.

[68] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs.* Newnes, 2012.

[69] J. A. Hartigan and M. A. Wong, "A k-means clustering algorithm," *JSTOR: Applied Statistics*, vol. 28, no. 1, pp. 100–108, 1979.

[70] R. Tibshirani, G. Walther, and T. Hastie, "Estimating the number of clusters in a dataset via the gap statistic," *Journal of the Royal Statistical Society, Series B*, vol. 63, pp. 411–423, 2000.

Acknowledgements

I would like to express my sincere gratitude to my advisor, Professor Won-Ki Jeong for the continuous support throughout my study and research. Without his guidance and constant feedback, my PhD course and this dissertation would not have been achievable. I would also like to thank my committee members, Prof. Sungahn Ko (UNIST), Prof. Young-ri Choi (UNIST), Prof. Sam H. Noh (UNIST), Prof. Beomseok Nam (Sungkyunkwan University) for their encouragement, insightful comments toward this work.

I am very grateful to our HVCL lab mates; Hyungsuk Choi, Tran Minh Quan, Gyuhyun Lee, Woohyuk Choi, Jungmin Moon, Junyoung Choi, Kanggen Lee, Inwan Yoo, Haejin Jeong, Nguyen Duc Thanh, Hyungjoon Jang, ChanMin Park, Hyeonsoo Lee, Nguyen Tuan Khoa, Jing Wei Tan, Euibin Bae, Nguyen Mai Huong, Tran Anh Tuan, Sungduk Cho for their collaboration and support in various research areas related to this dissertation.

In addition, my brilliant friends and colleagues inspired me over the many years; Jaewon Kwak, Sangbin Jeong, Subin Han, Moohyun Nam, Wookhee Kim, Jonghyuk Park and many more for making my experience in graduate school exciting and fun.

Last but not least, I would like to express my deepest gratitude to my family, Gyuim Jeong and Jiae Hong. This dissertation would not have been possible without their warm love, continued patience, and endless support.