A Group-Ordered Fast Iterative Method for Eikonal Equations

Sumin Hong and Won-Ki Jeong, *Member, IEEE.*

Abstract—In the past decade, many numerical algorithms for the Eikonal equation have been proposed. Recently, the research of Eikonal equation solver has focused more on developing efficient parallel algorithms in order to leverage the computing power of parallel systems, such as multi-core CPUs and GPUs (Graphics Processing Units). In this paper, we introduce an efficient parallel algorithm that extends Jeong *et al.*'s FIM (Fast Iterative Method, [1]), originally developed for the GPU, for multi-core shared memory systems. First, we propose a parallel implementation of FIM using a lock-free local queue approach and provide an in-depth analysis of the parallel performance of the method. Second, we propose a new parallel algorithm, *Group-Ordered Fast Iterative Method (GO-FIM)*, that exploits causality of grid blocks to reduce redundant computations, which was the main drawback of the original FIM. In addition, the proposed GO-FIM method employs clustering of blocks based on the updating order where each cluster can be updated in parallel using multi-core parallel architectures. We discuss the performance of GO-FIM and compare with the state-of-the-art parallel Eikonal equation solvers.

Index Terms—Eikonal equation, GPU, Parallel Computing

1 Introduction

The Eikonal equation has been widely used in diverse fields, such as geoscience and geophysics, computer vision, image processing, computer graphics, etc [2]. It 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 an uniform rectilinear grid in this paper), Γ is the collection of seed points (i.e., boundary condition), $\phi(\mathbf{x})$ is the *travel time* or the *distance* from the seed region to the grid 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 wave propagation from the seed region where the motion is governed by the speed function, and the solution of the equation represents the geodesic distance of the shortest path from the nearest seed point. Therefore, the Eikonal equation is frequently used in the problems related with the distances or travel time in space, such as seismic travel time computation [3] or finding the minimum cost path for tracing neural fiber tracts in the brain [4].

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 (more details can be found in [1], [5], [6]), and we simply borrow the same scheme in this paper. 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:

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

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 neighbor 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 $(n)^+ = \max(n, 0)$. Since $U(\mathbf{x})$ is the only unknown in Eq 2, a closed-form solution of U can be found by solving a quadratic equation.

Therefore, the main focus of this paper is to introduce a new parallel algorithm to efficiently solve the Eikonal equation. Specifically, we introduce two parallel numerical algorithms that extend the *Fast Iterative Method (FIM)* by Jeong *et al.* [1] for multi-core shared memory systems. 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 paper, we address these issues by proposing a new parallel algorithm

• *The authors are with the School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Ulsan 689-798, Republic of Korea.*
E-mail: {sumin246,wkjeong}@unist.ac.kr

that improves the performance on highly-complicated speed functions as well as the efficiency on shared memory systems.

The main contributions of this paper 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.

The rest of the paper proceeds as follows. In Section 2, we overview the previous work on serial and parallel Eikonal solvers. In Section 3 we introduce our lock-free implementation of FIM. In Section 4 we introduce our novel group-ordered FIM algorithm in detail. Experimental results and their detailed analysis and discussions will be given in Section 5. Finally, Section 6 wraps up the discussion and suggests some future research directions.

2 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 [5], [7], [8], [9]. 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. A popular method is Fast Marching Method (FMM) proposed by Sethian [6], [10]. 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, but Hamiltonian-based numerical distance computation is used. Therefore, the complexity of the algorithm is $O(N\log N)$, which is worst-case optimal. 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) [11] 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 [12] and distance computation on unstructured grids [13]. Bak *et al.* [14] proposed the Locking Sweeping Method (LSM) to improve the performance of FSM by using boolean flags to skip unnecessary update. Since FSM does not rely on a global ordered data structure, several parallel variants have been proposed. Zhao [15] 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. Recently, Detrixhe *et al.* [16] proposed a parallel sweeping method that overcomes the limitation of Zhao's parallel FSM. It uses the Cuthill-Mckee ordering [17], 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. In fact, a similar idea of parallelizing Gauss-Seidel sweeping order has been proposed earlier for solving Eikonal equation on parametric surfaces by Weber *et al.* [18].

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.* [19], Falcon *et al.* [20], [21], and some parallel algorithms by Bertsekas *et al.* [22], 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.* [23]. Later, Jeong *et al.* [1], [4] 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.* [23], 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.* [24] further extends FIM to compute the geodesic distance on unstructured meshes on the GPU. Bak *et al.* [14] introduced the single queue method, which is similar to Bornemann *et al.* [23] except that causal ordering is used to determine the neighbor nodes to be added to the queue.

Recently, researchers are actively developing hybrid approaches to overcome the limitation of existing methods. Bak *et al.* [14] 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 [25] 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.* [26] 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 [27].

As we 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 paper, we tackle two problems at the same time by proposing a lock-free parallel implementation of FIM and a group-ordered variant of FIM.

3 Lock-free Parallel FIM

3.1 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.

Algorithm 1: 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          $U(x) \leftarrow \infty$
/* 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 $L$
```

3.2 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 2).

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 2 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.

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 2). 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 2 (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
    . . . . .
}
```

Algorithm 2: Lock-free Parallel FIM

```
Input: Grid $\Omega$, Solution $U$, Active list $L$
/* Initialization */
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
   /* Flag array for node state */
6  foreach $i \in N$ do in parallel
7      forall $x \in L_i$ do
8          $F(x) \leftarrow 1$
   /* Parallel update of sublists $L_i$ */
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
       /* Solve for $L_i$ */
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                 $U(x) \leftarrow q$
               /* If converged */
22             else
                   /* Check adjacent neighbor */
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                             $U(x_{nb}) \leftarrow q$
                               /* atomic operator */
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
```

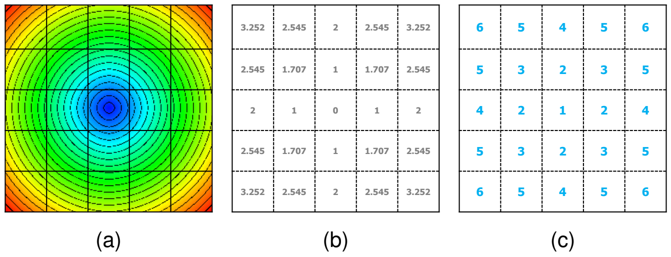
4 Group-Ordered FIM

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 section, 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 [1] 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 [6]) 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. 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., $O(n\log n)$ 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 section, 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.

4.1 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 × 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 × m (Figure 1 (a), if the input grid size is 40 × 40 and the block size is 8 × 8, then the coarse grid size is 5 × 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



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

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

Fig. 1. 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.

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 1 (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 1 (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 3 shows each step of GO-FIM algorithm in pseudocode.

4.2 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:

$$L_{upper} = \{\mathbf{x} | \mathbf{x} \in B \subset G_v \text{ and } \exists\, \mathbf{x}_{nb} \in B' \subset G \setminus G_v\} \quad (3)$$

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 2 shows an example of three

Algorithm 3: 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 3)
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 4)
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 2
21             Barrier Synchronization
22             Solve for $L_i$
23             Barrier Synchronization
```

update passes and corresponding initial active list for each pass (drawn in red color). In the first pass (Figure 2 (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 2 (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., nodes that are not true upwind nodes in the group G_v. Therefore, we can define a more tightly

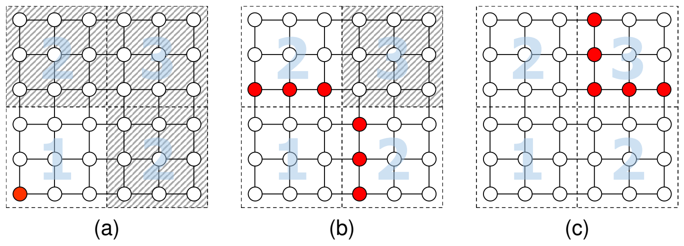


Fig. 2. Example of initial active lists for three update groups

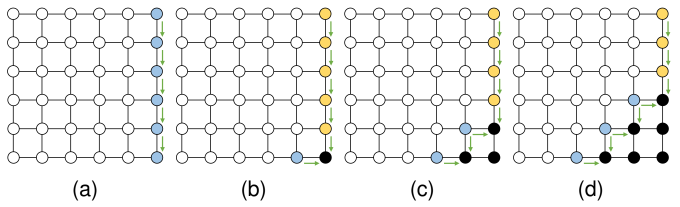


Fig. 3. Example of an initial active list defined using an upper bound given in Equation 3. Green arrows represent causal dependency between nodes. Blue points are active nodes updated only once. Yellow points are active nodes unnecessarily updated multiple times due to a non-tight bound. Black points are converged nodes.

bounded initial active list L required for the v-th update step as follows:

$$L_{tight} = \{\mathbf{x} | \mathbf{x} \in B \subset G_v \text{ and } \forall \text{ its upwind neighbor nodes } \mathbf{x}_{nb} \text{ belong to } G \setminus G_v\} \tag{4}$$

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 3 shows an example of an initial active list based on the upper bound given in Equation 3 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 3 (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 4 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 3.

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 4.

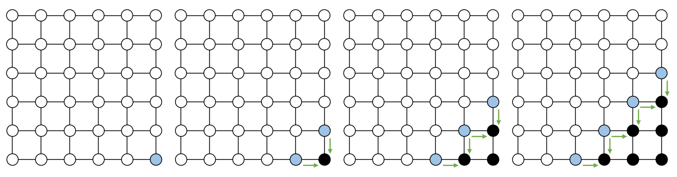


Fig. 4. Example of an initial active list defined using a tight bound given in Equation 4. Due to the tightness of the bound, only a single active node is included in the initial active list. There is no unnecessary update in this example.

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 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 4). 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 3. 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 1, there is a significant performance improvement by using a tightly bounded active list, roughly up to 40% of computation is reduced.

Algorithm 4: 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 1
Comparison of average number of update per node for different initial active list generation schemes (tested on GO-FIM8). More details about the datasets can be found in Section 5.

4.3 Block Clustering

Another problem we need to consider is how to cluster blocks into disjoint groups. In Figure 1, 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 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 [28] 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.* [29]. This approach is minimizing the variation of values within each group, i.e., *within-cluster dispersion* (Figure 5). 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. More detailed discussion regarding the clustering is given in Section 5.5.

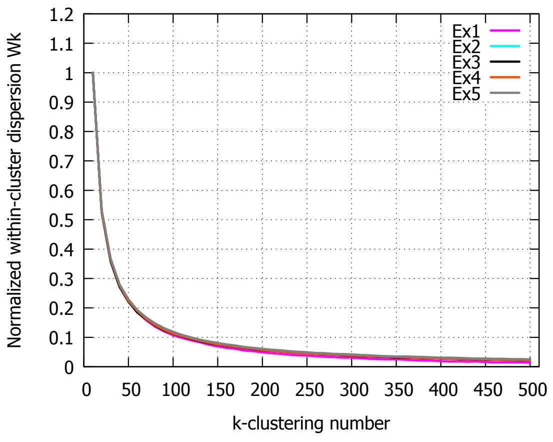


Fig. 5. Within-cluster dispersion for different clustering value K.

5 Results and Discussion

In this section 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 [10], Zhao's FSM [11] [15], Detrixhe *et al.*'s parallel FSM (DFSM) [16] and parallel Heap Cell Method [27]. In cases of GO-FIM and pHCM, we use two different block size configurations – GOFIM4 and pHCM4 for $4\times4\times4$ 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 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 [15] 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.

The speed maps used in our performance evaluation are 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 6 shows the color plot and iso-contour rendering of the solution (i.e., distance) of the Eikonal equation for each speed map.

5.1 Single-threaded result

Although we propose parallel algorithms in this paper, 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 2 (first multi-row), and their average update numbers are listed in Table 3. 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.

		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 2
Running time using different number of threads (1 to 32 threads) measured in second. The fastest time for each dataset is marked in boldface.

As expected, the performance of FMM did not vary over different examples because FMM is worst-case optimal

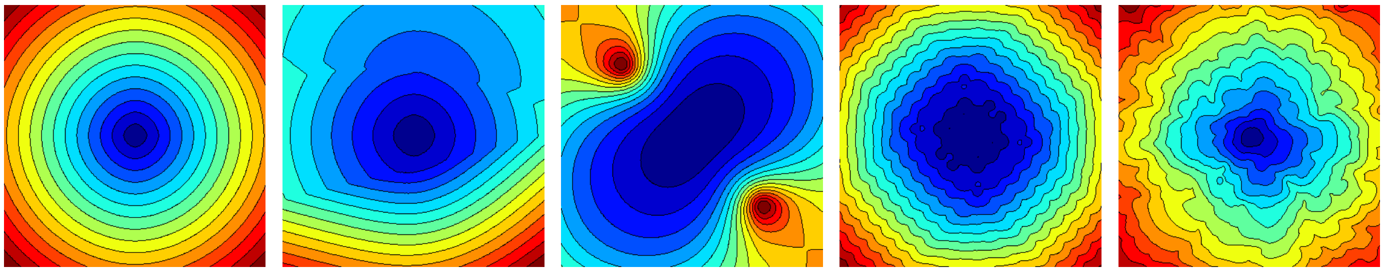

(a) Example 1 (b) Example 2 (c) Example 3 (d) Example 4 (e) Example 5

Fig. 6. 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.

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 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. More discussions about the relationship between the block size and performance will be given in Section 5.5.

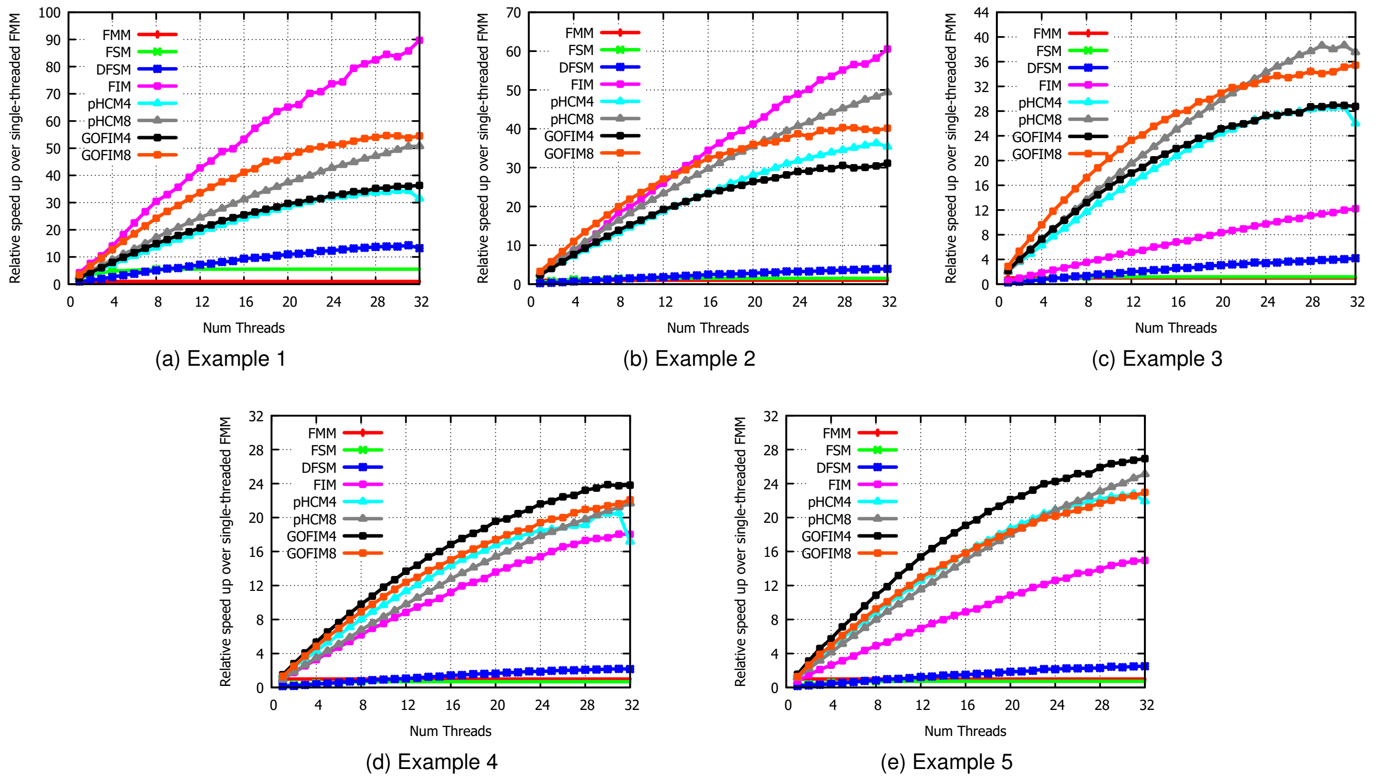
Even though GO-FIM and pHCM share a similar idea, there are also subtle but important differences that make two methods perform differently. pHCM restricts the computational domain to a single cell per thread, but GO-FIM expands the computational domain based on the order of block clusters (this is also 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 3, 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 case, [27]), but GO-FIM shows smaller overhead (e.g., preprocessing cost) around 4% at most (Table 4, thread 1). Therefore, even though pHCM8 requires fewer updates than GO-FIM8 for Example 5, GO-FIM8 is actually faster than pHCM8.

	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 3
Average number of update of Eikonal solvers on different examples.

5.2 Multi-threaded result

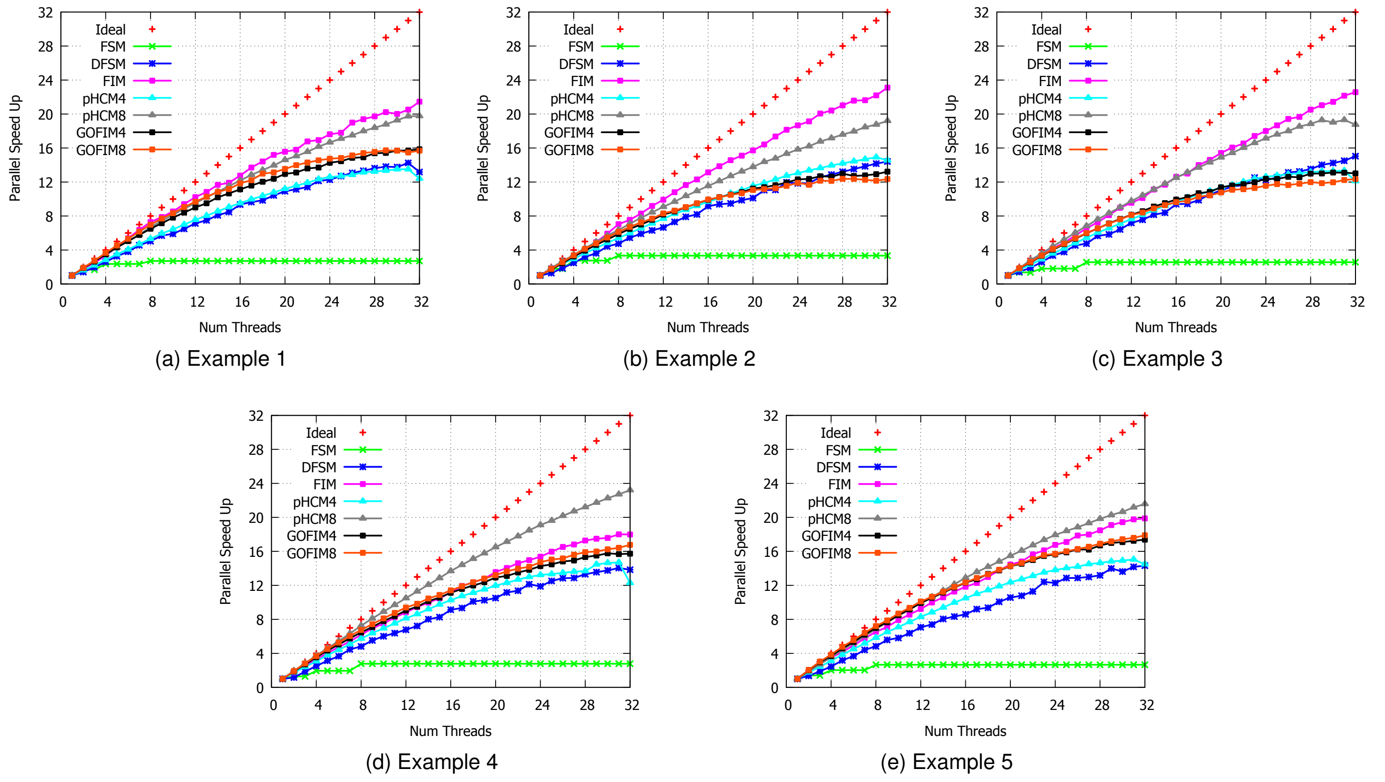
Multi-threaded results are demonstrated as raw running times (Table 2), relative performance over FMM (Fig 7), and parallel scalability of each solver (Fig 8). A popular parallel



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

(d) Example 4 (e) Example 5

Fig. 7. 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



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

(d) Example 4 (e) Example 5

Fig. 8. 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.

	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 4
The proportion of the preprocessing time of GO-FIM (measured in the percentage over the entire running time)

Eikonal solver is Zhao's parallel FSM [15]. 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 8, the observed maximum speed up for 32 threads over a single thread is only about a factor of three.

In DFSM [16], the authors proposed a diagonal sweeping approach in order to improve 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 7 and Table 2) 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 $24\times$ speed up for 32 threads, which results in about $80\times$ 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.* [27]), 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 outperforms GO-FIM8 on Example 2 for 13 threads and up (see Figure 7 (b) pink and orange curves cross near 13 threads), and pHCM8 outperforms GO-FIM8 for 22 or more threads on Example 3 (see Figure 7 (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.

5.3 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 9 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 counts.

5.4 GO-FIM on the GPU

The proposed GO-FIM can be thought of as an extension of BlockFIM [1] because GO-FIM uses rough (i.e., not exact) ordering of block update to reduce unnecessary computation of the original FIM. Table 5 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.

	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 5
Running Time comparison of BlockFIM and GO-FIM on an NVIDIA Tesla K40c GPU.

5.5 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-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

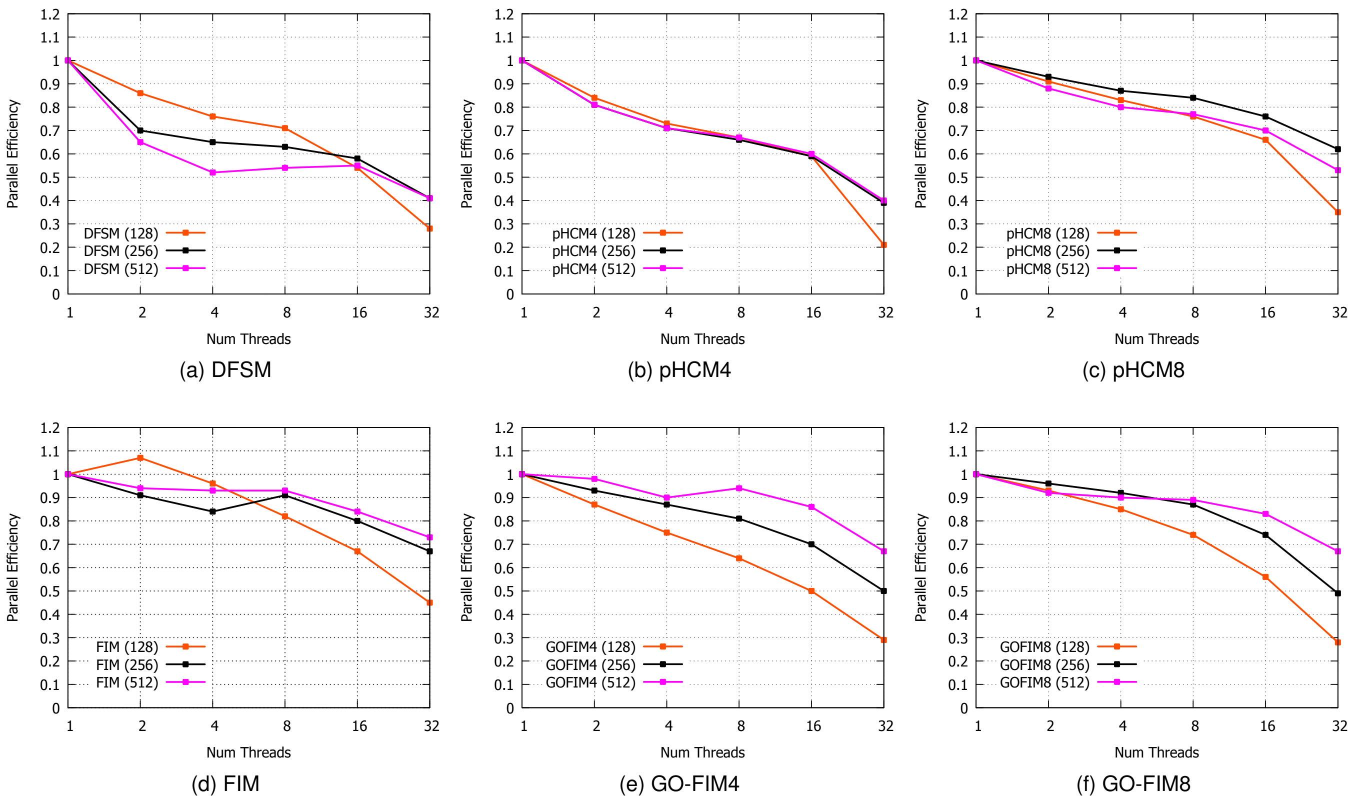


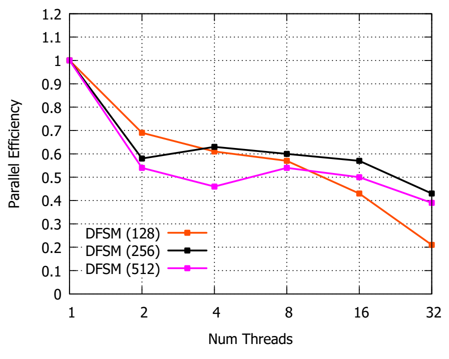
Fig. 9. Parallel scalability test result of Example 1 on different grid size (N^3). The horizontal axis shows the number of parallel threads by a log scale, and the vertical axis shows the parallel efficiency (parallel speed up / num threads).

preprocessing time as well. We found that the block size of 4 or 8 works best for 256^3 input data size.

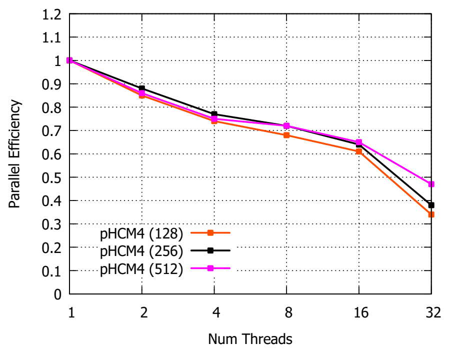
GO-FIM belongs to the class of two-scale hybrid algorithms, such as FMSM proposed by Chacon *et al.* [26], but there also exists important differences between FMSM and GO-FIM. In FMSM, as the authors pointed out, statically computed cell orders do not perfectly capture the correct causal relationship 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 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 11, 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 11 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.

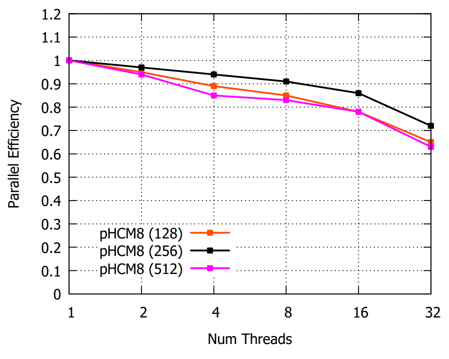
One limitation of GO-FIM is that the performance depends on the structure of the input speed map and the layout of coarse blocks. To emphasize this effect, we tested GO-FIM on the maze-like data having permeable barriers with a low speed value (see Figure 12). Dotted lines show the boundary of blocks. 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 12 (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 is the wave propagation direction of coarse blocks in GO-FIM, and the green arrow is the correct wave propagation direction). Figure 12 (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 12 (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



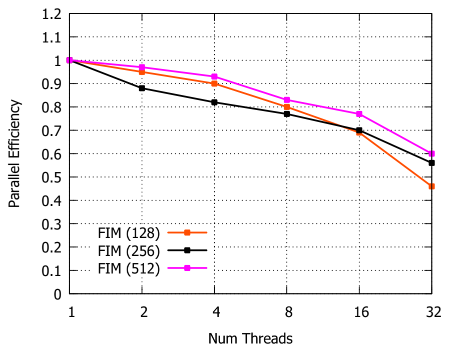
(a) DFSM



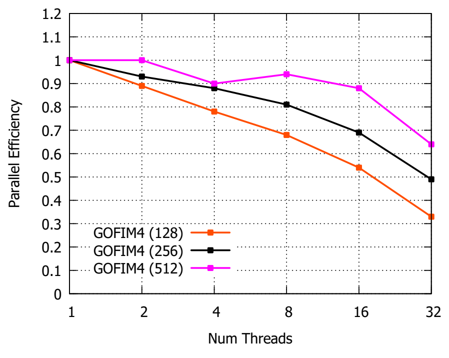
(b) pHCM4



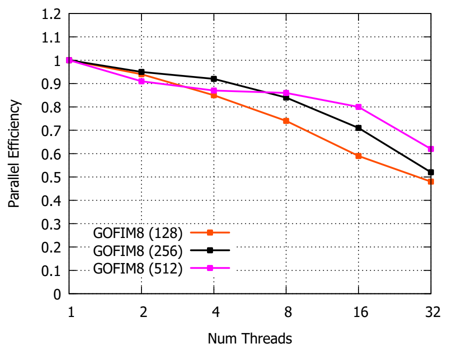
(c) pHCM8



(d) FIM



(e) GO-FIM4



(f) GO-FIM8

Fig. 10. 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).

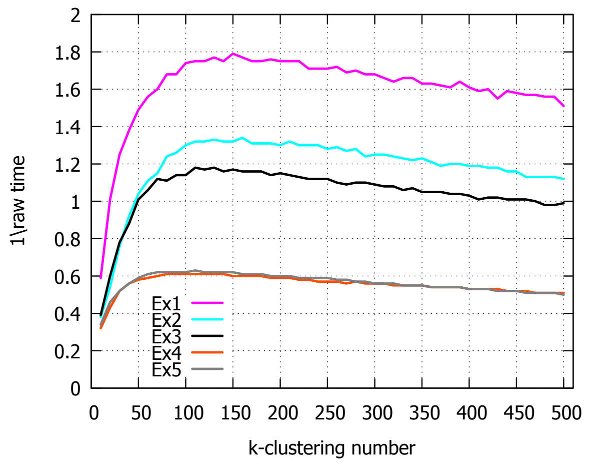


Fig. 11. Performance of GO-FIM8 with 16 threads measured using various cluster sizes.

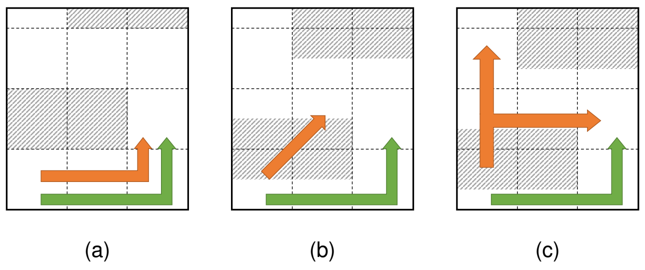


Fig. 12. Example of miss prediction on coarse grid. The speed value on white region is 1.0, and that of grey region (permeable barrier) is 0.01. The green arrow represents correct wave propagation direction, and the orange arrow represents wave propagation direction on the coarse grid of GO-FIM.

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.

6 Conclusion

In this paper, 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.

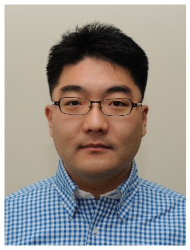
Acknowledgments

This work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No.R0190-15-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development), and Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education (NRF-2014R1A1A2058773).

References

[1] 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
[2] J. A. Sethian, *Level set methods and fast marching methods*. Cambridge University Press, 2002.
[3] J. A. Sethian and M. A. Popovici, "3-d traveltime computation using the fast marching method," *Geophysics*, vol. 64, pp. 516–523, 1999.
[4] 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
[5] E. Rouy and A. Tourin, "A viscosity solutions approach to shape-from-shading," *SIAM Journal on Numerical Analysis*, vol. 29, pp. 867–884, 1992.
[6] J. A. Sethian, "Fast marching methods," *SIAM Review*, vol. 41, no. 2, pp. 199–235, 1999.
[7] 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
[8] ——, "Finite-difference calculation of traveltimes in three dimensions," *Geophysics*, vol. 55, no. 5, pp. 521–526, 1990.
[9] 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.
[10] 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.
[11] H. Zhao, "A fast sweeping method for eikonal equations," *Mathematics of Computation*, vol. 74, pp. 603–627, 2004.
[12] 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.
[13] 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
[14] 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
[15] H. Zhao, "Parallel implementations of the fast sweeping method," *Journal of Computational Mathematics*, vol. 25, no. 4, pp. 421–429, 2007.
[16] 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
[17] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.
[18] 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
[19] 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.
[20] M. Falcone, "A numerical approach to the infinite horizon problem of deterministic control theory," *Applied Math. Optim.*, vol. 15, pp. 1–13, 1987.
[21] ——, "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.
[22] 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
[23] 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
[24] 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
[25] 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.
[26] 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.
[27] ——, "A parallel two-scale method for eikonal equations," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. A156–A180, 2015.
[28] J. A. Hartigan and M. A. Wong, "A k-means clustering algorithm," *JSTOR: Applied Statistics*, vol. 28, no. 1, pp. 100–108, 1979.
[29] 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.

Sumin Hong received a BS degree from Ulsan National Institute of Science and Technology (UNIST), Korea in 2013. He is currently working towards the PhD degree in the school of electrical and computer engineering, UNIST. His research interest includes parallel algorithms, GPU computing, distributed systems and cloud computing.



Won-Ki Jeong is currently an associate professor in the school of electrical and computer engineering at UNIST. Before joining UNIST, he was a research scientist in the Center for Brain Science at Harvard University from 2008 to 2011. His research interests include visualization, image processing, and parallel computing. He received a Ph.D. Degree in Computer Science from the University of Utah in 2008, and was a member of the Scientific Computing and Imaging (SCI) institute. He is a recipient of the NVIDIA Graduate Fellowship in 2007, and is currently the PI of the NVIDIA GPU Research Center at UNIST.