Procedia Computer Science
Volume 80, 2016, Pages 190–200
ICCS 2016. The International Conference on Computational Science

A Multi-GPU Fast Iterative Method for Eikonal Equations using On-the-fly Adaptive Domain Decomposition

Sumin Hong and Won-Ki Jeong

Ulsan National Institute of Science and Technology
Ulsan, Republic of Korea
{sumin246,wkjeong}@unist.ac.kr

Abstract

The recent research trend of Eikonal solver focuses on employing state-of-the-art parallel computing technology, such as GPUs. Even though there exists previous work on GPU-based parallel Eikonal solvers, only little research literature exists on the multi-GPU Eikonal solver due to its complication in data and work management. In this paper, we propose a novel on-the-fly, adaptive domain decomposition method for efficient implementation of the Block-based Fast Iterative Method on a multi-GPU system. The proposed method is based on dynamic domain decomposition so that the region to be processed by each GPU is determined on-the-fly when the solver is running. In addition, we propose an efficient domain assignment algorithm that minimizes communication overhead while maximizing load balancing between GPUs. The proposed method scales well, up to 6.17× for eight GPUs, and can handle large computing problems that do not fit to limited GPU memory. We assess the parallel efficiency and runtime performance of the proposed method on various distance computation examples using up to eight GPUs.

Keywords: Eikonal Equation, GPU, parallel computing, domain decomposition

1 Introduction

The eikonal equation has a wide range of applications including geoscience [1], computer vision [8], image processing [7], path planning [5] and computer graphics [10], and is defined 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 a computational domain defined on an n-dimensional rectilinear grid, Γ is the collection of seed points (i.e., boundary condition), $\phi(\mathbf{x})$ is the first arrival time from the seed

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.309

region to the location $\mathbf{x}$, and $f(\mathbf{x})$ is a scalar speed function defined on $\mathbf{x}$. We refer *node* as a grid point on Ω defined by an n-tuple of numbers (i, j, k). On a discrete grid, we can use the first order Godunov upwind discretization $g(\mathbf{x})$ of the Hamiltonian $\mathrm{H}(\mathbf{x}, \nabla\phi)$ to numerically solve Equation (1) as shown in [8, 4]. In this paper, we focus on the three dimensional case only ($n = 3$). The solution of the eikonal equation represents wave propagation from the seed region where the motion is governed by the speed function.

The most popular Eikonal solvers include Fast Marching Method(FMM) [9] and Fast Sweeping method(FSM) [11]. FMM is based on the label-setting algorithm similar to Dijkstra's shortest path algorithm. Even though FMM is worst-case optimal, its parallel implementation is challenging due to the serial nature of the algorithm. FSM belongs to the label-correcting algorithm, i.e., the solution is iteratively updated until the entire domain converges to a steady state. FSM uses a special computing sequence, i.e., Gauss-Seidel update, in order to increase the convergence rate. Zhao *et al.* [12] parallelized FSM by executing each Gauss-Seidel update concurrently using parallel processors, but the algorithm restricts the maximum concurrency to eight for 3D grids. Later, Weber *et al.* [10] introduced an alternative discretization and update sequence to overcome the limitation of Zhao's parallel FSM, but this method only deals with Eikonal equations defined on parametric surfaces. More recently, a similar idea has been implemented on a single [2] and multiple GPUs [6], respectively, to solve Eikonal equations on rectilinear grids. Even though Krishnasamy *et al.* [6] attempted to address the same problem as ours, i.e., multi-GPU Eikonal solver on a shared memory system, their approach is based on a static domain decomposition and is not efficiently applicable to adaptive algorithms like our method. Fast Iterative Method(FIM) [4] is a variant of the label correcting algorithm specifically designed to exploit fine-grain parallelism of recent many-core processors, such as GPUs. The algorithm manages *active list*, a collection of grid nodes being updated by the solver. Unlike FMM, FIM does not manage expensive ordered data structures, and the entire list is updated concurrently. Activation of node is determined by convergence, i.e., the solution on the node does not change over consecutive iterations, and nodes are allowed to be re-activated even after removed from the active list. Jeong *et al.* showed that FIM can be efficiently implemented on a single GPU by employing a block-based update scheme, called *BlockFIM* [4].

Even though BlockFIM showed good performance on a single GPU, it is not straightforward to leverage multiple GPUs due to the adaptive nature of the algorithm – meaning that a naive static domain decomposition, which works well on other iterative solvers like FSM, may result in load imbalance and excessive communication overhead. To address this issue, we propose an on-the-fly adaptive domain decomposition method for multi-GPU BlockFIM. The core idea is that each GPU owns its disjoint local sub-domain that progressively grows as the active list expands. Our proposed algorithm estimates the regions to be computed in the following iteration and assigns under-utilized GPUs with a higher priority while promoting the same GPUs to be assigned adjacent to each other. By doing so, we can maximize load balancing and minimize communication between GPUs. The proposed method showed up to around 2$\times$ speed up compared to naive static domain decomposition methods.

2 Multi-GPU Extension of BlockFIM

BlockFIM is a variant of FIM specifically designed for Single Instruction Multiple Data (SIMD) architecture, such as GPUs. BlockFIM splits the computational domain into disjoint *blocks* where each block consists of multiple nodes (in our experiments, we use an $8 \times 8 \times 8$ block), and treats the block as a basic compute primitive – meaning that the nodes in the same block are updated concurrently by the parallel computing cores in the GPU. Therefore, BlockFIM manages the active list that contains blocks instead of nodes, and the algorithm iteratively updates solutions of the active blocks. However, the previous BlockFIM targets only a single GPU, so we propose a multi-GPU extension of BlockFIM using a domain decomposition method, which splits the input computational domain into sub-domains and assigns them to GPUs. Each sub-domain is a collection of blocks, and partially overlaps with its adjacent sub-domains around its boundary, called *halo* (Figure 1), which allows each sub-domain to be processed independently on different GPUs. When a boundary block is updated, then its adjacent halo region must be synchronized accordingly via communication between GPUs. Since the amount of communication depends on the halo size, domain decomposition strategies significantly affect the performance of the solver.

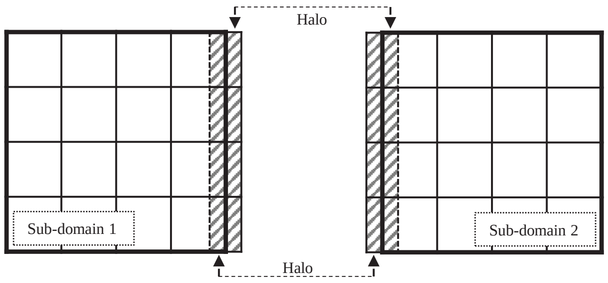


Figure 1: Halo at the boundary of adjacent sub-domains

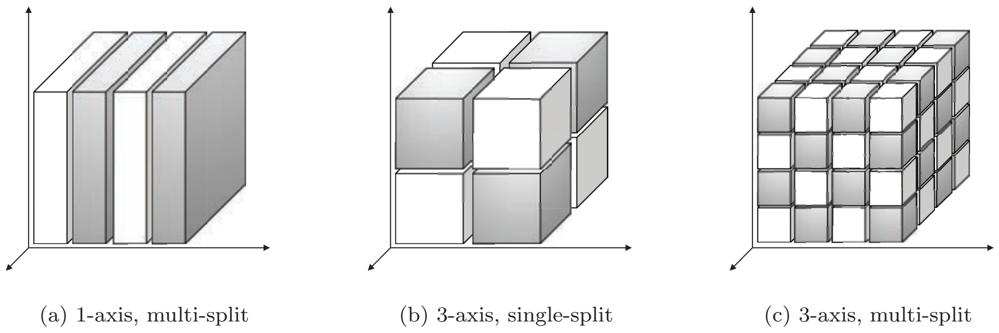


Figure 2: Examples of regular domain decomposition on a 3D rectilinear grid. (a) is multiple splits along one axis, (b) is single split along each axis, and (c) is multiple splits along each axis.

2.1 Regular Domain Decomposition

A commonly used domain decomposition strategy is a simple axis-aligned grid splitting as shown in Figure 2. Most of existing parallel Eikonal solvers also employ a regular domain decomposition method – Herrmann *et al.* [3] parallelized FMM [9] on regular sub-domains using multiple computing processes. The proposed algorithm maintains an independent local heap list to select the computing node on each sub-domain, which significantly impairs the scalability on massively parallel architecture. Krishnasamy *et al.* [6] extended a 3D parallel sweeping method [2] on multiple GPUs. The solver uses plane sweeping, i.e., a 2D plane sweeps the 3D volume along axis-aligned sweep directions while the entire 2D plane is updated in parallel. In order to parallelize sweeps over multiple GPUs, domain decomposition is restricted to 1D axis-aligned splitting along the sweep direction (as in Figure 2 (a)).

Any domain decomposition strategy can be applied to multi-GPU BlockFIM because blocks in active list can be independently updated. However, the shape of active list can be arbitrary depending on the input speed function, so evenly distributing blocks across regular sub-domains is infeasible in most cases. For example, Figure 3 shows 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, and each sub-domain consists of nine blocks). Assume each sub-domain is assigned to a GPU. In Figure 3 (a), the initial active list is evenly distributed across four sub-domains so that two blocks are located in each sub-domain. After one BlockFIM update, only a half of the active list converged (Figure 3 (b), marked in green). In the next iteration, the green blocks will be removed from the active list and some of its adjacent neighbor blocks are activated (Figure 3 (c)). Because the active list expands towards the bottom-left direction, more active blocks are located in the bottom-left region (four blue circles) than other sub-domains (three blue circles in top-left and bottom right regions, and two blue circles in top-right region), which introduces load imbalance. This was not the case for other parallel Eikonal solvers, especially sweeping algorithms, because they are not relying on active list and update the entire grid per each sweeping iteration. Therefore, we need a more flexible domain decomposition method to address this issue.

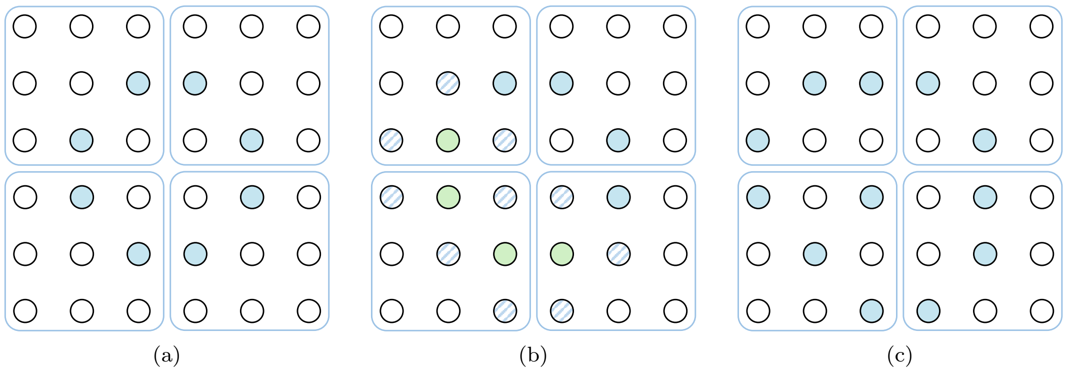


Figure 3: Example of load imbalance in regular domain decomposition. Blue circles are active blocks, green circles are converged blocks, and stripe circles are candidate blocks to be active in the following iteration. Due to partial convergence of active list, blocks are not evenly distributed in (c).

2.2 On-the-fly Adaptive Domain Decomposition

The problem of regular domain decomposition discussed above is mainly due to the fact that dynamics of the active list cannot be determined in advance. In FIM, the active list dynamically grows or shrinks depending on the speed value defined on each grid node in a nonlinear fashion. Therefore, the strategy we propose is to dynamically decompose the domain as the active list propagates – meaning that as the active list expands, we incrementally expand sub-domains so that roughly the same number of active blocks are assigned to each sub-domain to maximize *load balance*. Another important performance metric to consider is *communication cost* for halo synchronization – meaning that we try to minimize the halo size between different sub-domains. For this, blocks assigned to the same sub-domain should be spatially clustered as much as possible.

For efficient implementation of the proposed dynamic domain decomposition, we use a simple block-to-subdomain mapping table, i.e., Domain Mapping Table (DMT). When the solver starts, the DMT entries for initial active blocks are filled with their corresponding sub-domain index (can be assigned randomly at the beginning). The DMT entries for other non-active blocks are marked as *un-assigned*. As the active list expands, some of active blocks are converged and removed from the active list, and some non-active blocks are activated and inserted into the active list. For newly activated blocks, we check their DMT entries and only *un-assigned* block's sub-domain index is assigned using the domain assignment algorithms discussed below.

Domain Assignment Algorithm In order to consider both load balancing and communication cost, we propose Algorithm 1 based on the following strategies. First, we assign adjacent blocks to the same sub-domain as much as possible, i.e., if two blocks are adjacent each other then they belong to the same sub-domain, to increase clustering and reduce halo communication. This can be done by assigning a new block to a sub-domain that one of its adjacent neighbor blocks belong to. To further reduce the communication cost between sub-domains, we choose the largest adjacent sub-domain when assigning a new block. For example, Figure 4 (a), the orange circle (center) is the new block to assign a sub-domain. It has three adjacent neighbor blocks whose sub-domains are already assigned (left, right, and bottom). Among them, the left and bottom blocks are assigned to the blue sub-domain, while the right block is assigned to the green sub-domain. Since the block sub-domain is the largest (i.e., assigned to two blocks) among the adjacent blocks, the orange block is assigned to the blue sub-domain. This is mainly for reducing the size of boundary between adjacent sub-domains as shown in the Figure. However, this strategy does not consider fair assignment of sub-domains because it does not take into account the global information (i.e., the size of active lists). Therefore, the second strategy we propose is to enforce load balance between newly assigned blocks. We repeatedly move blocks from the largest to the smallest active list until its difference becomes less than the user given threshold (ε_1). Algorithm 1 shows the entire procedure as the combination of these two strategies.

Load balancing in Algorithm 1 is mainly focusing on matching task loads among sub-domains for the next iteration. However, frequent load balancing will impair clustering of sub-domains and increase communication overhead. Thus, we need to perform load balancing as sparsely as possible. One strategy is performing load balancing once per every pre-determined number of iterations, or using a metric to detect load imbalance. In this paper, we use the standard deviation of the size of active lists as the load imbalance metric.

Improved Clustering Algorithm Algorithm 1 shown above tries to address two problems, load balance and clustering, which are intrinsically orthogonal. The solution we provide is performing load balancing periodically but not too frequently, but there exists a room for improvement especially in clustering. In order to increase the performance further, we propose an improved clustering algorithm using a local load balancing strategy. The idea is that when we insert an unassigned block v to a temporally list Q, we examine not only the count of adjacent sub-domains but also the number of *expected* task loads for each sub-domain. Simply

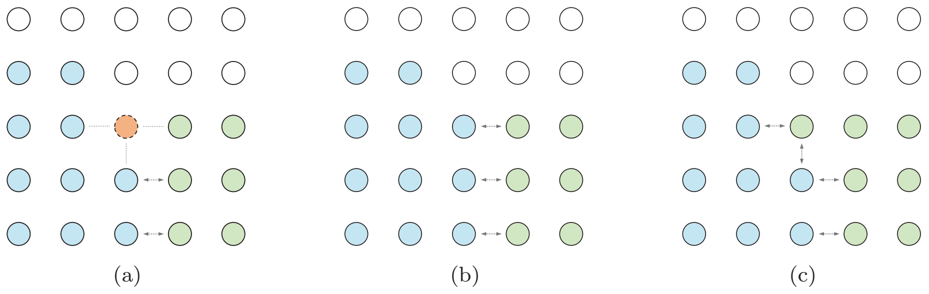


Figure 4: Domain assignment for clustering. (a) Orange circle (center) is an un-assigned block that has two blue and one green adjacent blocks, (b) blue is assigned to the center, and (c) green is assigned to the center. Arrows represent communication across sub-domains.

speaking, this approach tries to maximize clustering as well as load balancing between adjacent sub-domains using estimated active list size c_i where α and β affect how fast the list grows relatively. If c_j and c_k are similar, i.e., the size of largest and smallest active lists are similar, then there is no problem in load balancing so the current block should be assigned to Q_j to promote clustering. Otherwise, the block is assigned to Q_k to improve load balance because k is the sub-domain that has the smallest expected active list at this moment. Algorithm 2 describes the proposed improved clustering algorithm, and this code block can simply replace the clustering code in Algorithm 1 from line 1 to 7 (assume P is collected from L in advance).

3 Result

We evaluate the performance of proposed method on a computing server equipped with eight NVIDIA Tesla M2080 GPUs (each has 512 CUDA cores and 6GB device memory). We implement the experiment code in C++ and CUDA 6.0 with OpenMP with the -O3 level optimization. The grid dimension is 640^3, and 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 seed location is intentionally placed near the corner to show the worst-case performance of the algorithm for unbalanced task loads. All computations are conducted in double precision, and the block width is 8. The speed maps used in our experiments are defined as follows (and their distance maps are shown in Figure 5):

We tested three regular decomposition methods and our on-the-fly adaptive decomposition methods. Each decomposition generates same number of sub-domains as the number of GPUs (except 3-axis multi-split) so that each sub-domain is assigned to a single GPU. 1-axis multi-split method splits the data uniformly only along z-direction. 3-axis single-split method splits the data along the different axis whenever the number of GPUs is doubled. 3-axis multi-split method splits the data multiple times along each axis to generate more sub-domains than the number of GPUs (we use 16^3 sub-domain size) and assign GPUs to sub-domains multiple times in a checkerboard fashion. For our methods, we used the empirically-obtained best parameters values α, β, ε_1, ε_2, and δ as 1, 6, 50, 1.05, and 0.1, respectively. Table 1 lists the running time of each domain decomposition method on different speed maps using various number of GPUs. Among regular decomposition methods, 3-axis multi-split shows the best performance due to its ability to distribute tasks evenly over many small sub-domains. Our on-the-fly adaptive

Algorithm 1: DOMAIN ASSIGNMENT ALGORITHM

```
Input: Active List $L$, Domain Mapping Table $T$
/* $N$: total number of sub-domains */
/* $L_i$: sub active list for sub-domain $i$ ($L = L_1 \cup L_2 \cup ... \cup L_N$) */
/* 1.  Block clustering for reducing halo communication */
/* $P_i$ is the list of already assigned blocks in $L_i$ */
/* $Q_i$ is the list of unassigned blocks in $L_i$ */
1  forall the $i = 1$ to $N$ do
2      forall the $v \in L_i$ do
3          if $T(v)$ = unassigned then
4              $j \leftarrow$ index of sub-domain adjacent to $v$ that appears most
5              Insert $v$ to $Q_j$
6          else
7              Insert $v$ to $P_i$
/* 2.  Load balancing between $Q_1, ..., Q_N$ */
8  if stddev$(L_1, ..., L_N) > \delta$ then
9      repeat
10         forall the $i = 1$ to $N$ do
11             $n_i \leftarrow$ size of $P_i$ + size of $Q_i$
12             $i_{max} \leftarrow i$ if $n_i \geq n_j$ for all $j = 1, .., i-1$
13             $i_{min} \leftarrow i$ if $n_i \leq n_j$ for all $j = 1, .., i-1$
14         $\Delta = n_{i_{max}} - n_{i_{min}}$
15         Load balancing by moving blocks from $Q_{i_{max}}$ to $Q_{i_{min}}$
16     until $\Delta > \varepsilon_1$
/* Update Domain Mapping Table and Collect Active List */
17 forall the $i = 1$ to $N$ do
18     forall the block $v \in Q_i$ do
19         $T(v) \leftarrow i$
20     $L_i \leftarrow P_i \cup Q_i$
```

decomposition (Algorithm 2) outperformed regular decomposition in most cases, up to 3.3× and 6.1× speed up on four and eight GPUs, respectively.

Figure 6 shows the amount of task (upper curves) and data communication (lower curves) for Map 1 and 3 running on four GPUs. We select these two maps as representative results because they are the fastest and slowest results, as shown in Table 1. As shown in Figure 6's left column, the amount of task per GPU diverges as iteration goes but periodically adjusted by load balancing algorithm. The middle column is the case that load balancing is performed more frequently, which makes task distribution more evenly but communication cost increases (because global load balancing in Algorithm 1 does not take into account clustering). However, due to the improved clustering method in Algorithm 2, the right column shows better load balancing with much low communication overhead.

Figure 7 shows weak scaling efficiency on Map 1 and Map 3 to elaborate how each decomposition method can handle large data that does not fit to a single GPU. We assign 2 GB of data per GPU, so up to 16 GB of data is processed by eight GPUs. Similar to strong scalability

Algorithm 2: Improved Clustering Algorithm using Local Load Balancing

```
/* $N$: total number of sub-domains */
Input: Active List $L_1, ..., L_N$, Assigned Block List $P_1, .., P_N$
Output: Unassigned block list $Q_1, ..., Q_N$
/* $c_i$: expected size of active list $L_i$ after clustering */
1  forall the $i = 1$ to $N$ do
2      $c_i \leftarrow \alpha \times$ size of $P_i$
3  forall the $i = 1$ to $N$ do
4      forall the $v \in L_i$ do
5          if $T(v) =$ unassigned then
6              $j \leftarrow$ index of sub-domain adjacent to $v$ that appears most
7              $k \leftarrow$ index of sub-domain adjacent to $v$ whose $c$ is the smallest
8              if $c_j/c_k < \varepsilon_2$ then
9                  Insert $v$ to $Q_j$          // Default policy, increase clustering
10                 $c_j = c_j + \beta$
11             else
12                 Insert $v$ to $Q_k$           // Expected task loads are unbalanced
13                 $c_k = c_k + \beta$
```

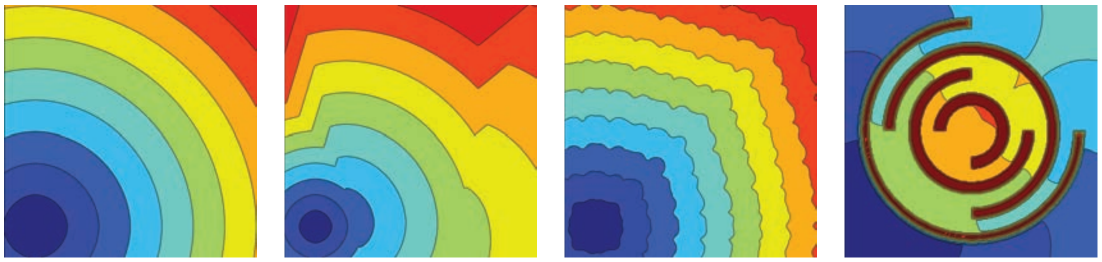
Map 1: $f = 1$. Constant speed map.
Map 2: $f = 1/4$, $1/2$, 1. Speed map with three layers of different speed values.
Map 3: $f = 1 + 0.5\sin(20\pi x) * \sin(20\pi y) * \sin(20\pi z)$. $(x, y, z \in \Omega = [0, 1])$
Map 4: Circular maze map with permeable barriers. ($f = 0.01$ on barriers, otherwise 1)

result, Algorithm 2 shows the best scaling performance in all cases.

Lastly, we wrap up our discussion by briefly comparing our result with the state-of-the-art method. To the best of our knowledge, the multi-GPU 3D parallel sweeping by Krishnasamy *et al.* [6] is the only similar work to ours as of today. In this work, they reported up to 2.8× speed up on four GPU, which is lower than the scaling performance of our method (up to 3.3×). We believe this is mainly due to the fact that their sweeping approach requires data shuffling between each sweep iteration (for example, after sweeping along x-axis, the entire data should be shuffled in order to proceed sweeping along y-axis in the next iteration). In addition, our load balancing and clustering methods can effectively handle the adaptive nature of BlockFIM algorithm and improve the performance on multiple GPUs.

4 Conclusion

In this paper, we present a novel on-the-fly adaptive domain decomposition algorithm for parallel BlockFIM on multi-GPU systems. In order to handle the adaptive nature of FIM, the proposed method progressively expands the computation domain as the active list of FIM grows. We proposed a simple table-based mapping scheme and domain assignment algorithm that takes into account spatial clustering and load balancing in order to minimize communication between GPUs while maximizing computing resource utilization. We showed that the proposed method outperforms other regular and naive dynamic domain decomposition strate-



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

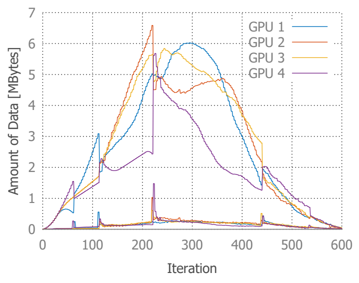
Figure 5: Color-coded distance map with iso-contours of our test datasets visualized in 2D. Blue to red color : from nearest to farthest distances to the seed point.

		Map 1	**Map 2**	**Map 3**	**Map 4**
	1 GPU	4.88	20.29	29.17	21.80
1-axis, multi-split	2 GPU	3.62 (1.35x)	16.05 (1.26x)	19.43 (1.50x)	14.97 (1.46x)
	4 GPU	2.18 (2.23x)	11.04 (1.84x)	13.02 (2.24x)	9.66 (2.26x)
	8 GPU	1.25 (3.90x)	6.54 (3.10x)	6.99 (4.17x)	5.48 (3.98x)
3-axis, single-split	2 GPU	3.62 (1.35x)	16.05 (1.26x)	19.43 (1.50x)	14.97 (1.46x)
	4 GPU	2.82 (1.73x)	10.37 (1.96x)	12.54 (2.33x)	9.99 (2.18x)
	8 GPU	2.04 (2.39x)	7.54 (2.69x)	9.57 (3.05x)	6.65 (3.28x)
3-axis, multi-split	2 GPU	3.25 (1.50x)	**11.03 (1.84x)**	16.24 (1.80x)	13.09 (1.67x)
	4 GPU	2.13 (2.29x)	6.71 (3.02x)	10.88 (2.68x)	**7.07 (3.08x)**
	8 GPU	1.64 (2.97x)	5.68 (3.57x)	8.13 (3.59x)	5.34 (4.08x)
Algorithm 1	2 GPU	2.99 (1.63x)	11.89 (1.71x)	17.95 (1.62x)	12.81 (1.70x)
	4 GPU	2.04 (2.39x)	6.71 (3.02x)	9.26 (3.15x)	7.42 (2.94x)
	8 GPU	1.16 (4.20x)	3.72 (5.45x)	4.92 (5.93x)	4.42 (4.93x)
Algorithm 2	2 GPU	**2.96 (1.64x)**	11.29 (1.80x)	**16.01 (1.82x)**	**12.63 (1.73x)**
	4 GPU	**1.70 (2.86x)**	**6.69 (3.03x)**	**8.72 (3.34x)**	7.11 (3.07x)
	8 GPU	**1.05 (4.64x)**	**3.62 (5.60x)**	**4.73 (6.17x)**	**4.41 (4.94x)**

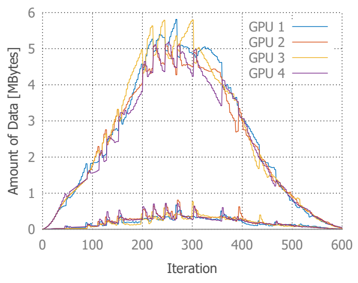
Table 1: Running time using different number of GPU device (1 to 8) measured in second. The fastest time for each dataset is marked in boldface.

gies by a large margin, and observed that the proposed method allows BlockFIM to scale up to 6.1× on eight GPUs. We also showed that our method scales better than state-of-the-art multi-GPU Eikonal solvers.

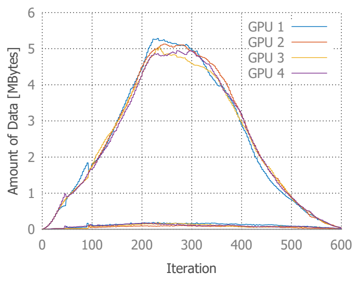
In the future, we plan to extend our dynamic domain decomposition algorithm to distributed systems using message passing interface (MPI) and streaming computing strategy to solve the Eikonal equation on extremely large computational domains. Exploring real-world application of the Eikonal equation on large computing problems, such as seismic image analysis and simulations, will be another interesting future research direction.



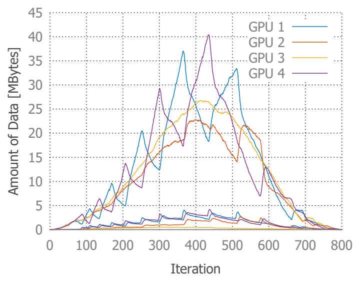
(a) Map 1, Algorithm 1, δ=0.3



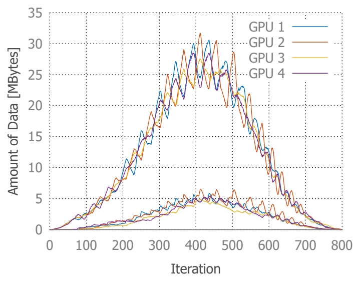
(b) Map 1, Algorithm 1, δ=0.1



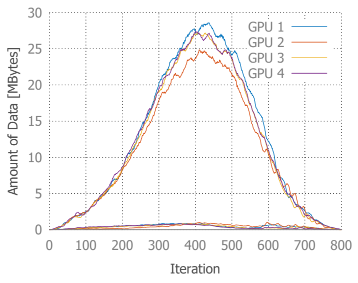
(c) Map 1, Algorithm 2, δ=0.1



(d) Map 3, Algorithm 1, δ=0.3

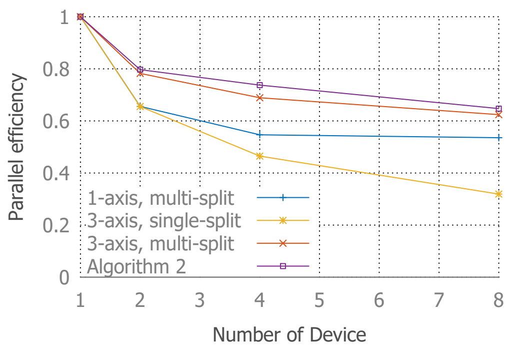


(e) Map 3, Algorithm 1, δ=0.1

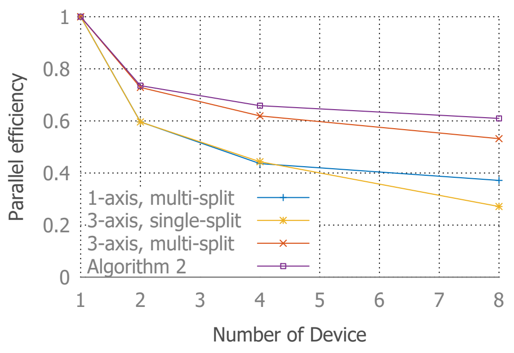


(f) Map 3, Algorithm 2, δ=0.1

Figure 6: Amount of task loads and data communication on Map 1 and 3 using four GPUs. The left column ((a) and (d)) is Algorithm 1 performed with infrequent load balancing ($\delta = 0.3$). The middle column ((b) and (e))is Algorithm 1 performed with more frequent load balancing ($\delta = 0.1$). The right column ((c) and (f)) is Algorithms 2. In each graph, upper four curves are task loads and lower four curves are communication loads per GPU. Algorithm 2 performed well in both of task loads and communication loads.



(a) Map 1



(b) Map 3

Figure 7: Weak scaling efficiency test on Map 1 and 3. The data size is set to around 2 GB per GPU (640^3, double precision)

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), Technology Innovation Program (No.10054548, Development of Suspended Heterogeneous Nanostructure-based Hazardous Gas Microsensor System) funded by the Ministry of Trade, industry & Energy(MI, Korea), and Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education (NRF-2014R1A1A2058773).

References

[1] Tor 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.

[2] Tor Gillberg, Mohammed Sourouri, and Xing Cai. A new parallel 3d front propagation algorithm for fast simulation of geological folds. In ICCS, volume 9 of Procedia Computer Science, pages 947–955. Elsevier, 2012.

[3] M Herrmann. A domain decomposition parallelization of the fast marching method. Technical report, DTIC Document, 2003.

[4] Won-Ki Jeong and Ross T. Whitaker. A fast iterative method for eikonal equations. SIAM J. Sci. Comput., 30(5):2512–2534, July 2008.

[5] Ron Kimmel and James A Sethian. Optimal algorithm for shape from shading and path planning. Journal of Mathematical Imaging and Vision, 14(3):237–244, 2001.

[6] Ezhilmathi Krishnasamy, Mohammed Sourouri, and Xing Cai. Multi-gpu implementations of parallel 3d sweeping algorithms with application to geological folding. Procedia Computer Science, 51:1494–1503, 2015.

[7] R. Malladi and J. Sethian. A unified approach to noise removal, image enhancement, and shape recovery. IEEE Trans. on Image Processing, 5(11):1554–1568, 1996.

[8] E. Rouy and A. Tourin. A viscosity solutions approach to shape-from-shading. SIAM Journal on Numerical Analysis, 29:867–884, 1992.

[9] James A. Sethian. A fast marching level set method for monotonically advancing fronts. In Proc. Natl. Acad. Sci., volume 93, pages 1591–1595, February 1996.

[10] Ofir Weber, Yohai S. Devir, Alexander M. Bronstein, Michael M. Bronstein, and Ron Kimmel. Parallel algorithms for approximation of distance maps on parametric surfaces. ACM Trans. Graph., 27(4):104:1–104:16, November 2008.

[11] H. Zhao. A fast sweeping method for eikonal equations. Mathematics of Computation, 74:603–627, 2004.

[12] H. Zhao. Parallel implementations of the fast sweeping method. Journal of Computational Mathematics, 25(4):421–429, 2007.