Evaluation of the Shared-Memory Parallel Fast Marching Method for Re-Distancing Problems

Georgios Diamantopoulos
Christian Doppler Laboratory for High Performance TCAD, Institute for Microelectronics, TU Wien, Austria
diamantopoulos@iue.tuwien.ac.at

Josef Weinbub
Christian Doppler Laboratory for High Performance TCAD, Institute for Microelectronics, TU Wien, Austria
weinbub@iue.tuwien.ac.at

Andreas Hössinger
Silvaco Europe Ltd., United Kingdom
andreas.hoessinger@silvaco.com

Siegfried Selberherr
Institute for Microelectronics, TU Wien, Austria
selberherr@iue.tuwien.ac.at

***Abstract*—The Fast Marching Method is widely used for the solution of the Eikonal equation in problems arising in science and engineering applications. A common application is the calculation of the distance to an interface resulting in a signed distance field. The Fast Marching Method is a non-iterative high accuracy method. However, the sequential nature of the algorithm does not favor a straightforward parallelization. Many parallelization approaches have been presented so far, but they do not provide a reasonable parallel efficiency. Recently, a promising new approach based on an overlapping domain decomposition technique has been introduced, providing a scalable, widely applicable parallel algorithm. However, investigations so far were limited to synthetic point-based problem cases which do not cover the much more challenging cases of interface re-distancing, where instead of simplistic point sets, challenging interface geometries are processed. In this work we fill this gap by analyzing a shared-memory implementation of the parallel Fast Marching Method for three-dimensional interface problems, covering the key challenges and various domain decomposition strategies. The parallel efficiency and run-time performance is examined on a dual-socket Ivy Bridge-EP compute node. Our analyses clearly outline the feasibility limits of a shared-memory parallel Fast Marching Method.**

I. Introduction

The simulation of an expanding front is a problem arising in several science an engineering applications, such as computational fluid dynamics, micro- and nanoelectronics, computational geometry, and computer vision [1]. The problem can be described by the Eikonal equation which in $\mathbb{R}^n$ reads:

$$\begin{aligned} |\nabla\phi(\mathbf{x})|F(\mathbf{x}) &= 1, \qquad \mathbf{x} \in \Omega \setminus \Gamma, \\ \phi(\mathbf{x}) &= 0, \qquad \mathbf{x} \in \Gamma \subset \Omega. \end{aligned} \tag{1}$$

Ω is a domain in $\mathbb{R}^n$, Γ is the initial interface (boundary), $\phi(\mathbf{x})$ is the unknown, and $F(\mathbf{x})$ is a positive speed function. The interface information propagates in the domain with the speed F. For $F = 1$, the solution $\phi(\mathbf{x})$ represents the minimum Euclidean distance of the point $\mathbf{x}$ to the interface Γ.

This work is focused on cases where $F = 1$, with the objective being the re-calculation of the distance of every point to the interface. In general, this task is referred to as re-distancing.

For the solution of this problem the Fast Marching Method (FMM), as presented [1], is a widely used method. The FMM is a one-pass (i.e., non-iterative) algorithm based on an upwind difference scheme. It uses a priority queue in order to determine the sequence of the nodes to be processed and works in a similar manner than Dijkstra's algorithm for finding the shortest path between nodes in a graph [2]. Because of the non-iterative nature of this method and its reliance on a priority queue, parallelization is not straightforward. There have been many attempts to provide a scalable parallel version of the FMM [3] [4] [5]. However, only recently a promising approach, based on overlapping domain decomposition, has been developed [6]. This approach provides a widely applicable, scalable algorithm for distributed-memory systems. A shared-memory version of this algorithm has also been developed [7] [8]. Although the parallel FMM has been investigated in detail, the evaluations are solely based on simplistic point sources as input and a single domain decomposition approach. This is insufficient to evaluate the general applicability to science and engineering problems as typically much more challenging interfaces have to be processed as input. In this work we evaluate an OpenMP-parallelized C++ implementation of the FMM for a set of challenging interfaces inspired by real-world problems, with various domain decomposition scenarios in one and two directions.

The paper is organized as follows: Section II discusses the related work. The serial and parallel FMM is briefly presented in Section III. In Section IV an analysis of the accuracy and the parallel performance of the implementation is provided. Finally, conclusions and future work are presented in Section V.

II. Related Work

The underlying algorithm of the FMM is presented in a detailed and complete form by Sethian in [1]. Since the FMM does not offer a straightforward way for parallelization, alternative methods for the solution of the Eikonal equation have been developed, which are easier to parallelize. The most predominant are the Fast Sweeping Method (FSM) and the Fast Iterative Method (FIM). The FSM is presented in [9] and its parallel variants in [10]. It uses upwind differences for the discretization, like the FMM, and Gauss-Seidel iterations with alternating sweeping order for the solution of the discretized

978-1-5386-3893-4/17/$31.00

system. The FIM, presented in [11], manages a list of active nodes that it iteratively updates until convergence is reached. It uses a block-based domain decomposition and, unlike the FMM, it is not relying on a priority queue, thus it is suitable for Single Instruction Multiple Data based parallelization. The drawback of these two methods is that because of their iterative nature they are not as accurate as the FMM.

Another set of methods is the family of Two-Scale methods [12]. The Heap-Cell method, for which the parallel version is presented in [13], has a two-scale technique which combines the FMM with FSM on different scales. However, these methods are out of scope for this work where only one scale is considered.

There have been many attempts to parallelize the FMM. In [3] a domain decomposition parallelization of the FMM is introduced and it forms the basis for other domain-decomposition techniques; however, the rollback operations for the communication among the sub-domains introduce computation and communication overheads that limit the parallel performance. Another domain decomposition parallelization is presented in [4]; however, the method is not very scalable. The method proposed in [5] provides a large-scale parallel FMM; however, it relies on the exact calculation of the distances in the ghost layer of every sub-domain, which is not possible for arbitrary interfaces. The method proposed in [6] is an overlapping domain decomposition approach, originally developed for a distributed-memory environment. The priority heap is local to each sub-domain, thus computation takes place independently in each sub-domain; and communication is performed with synchronized data exchanges. This is the most promising approach and a shared-memory adaptation [7] is used as the basis for our investigations.

III. The Fast Marching Method

In this section, a brief overview of the serial and parallel version of the FMM is given. The discussion of the FMM is the basis for the analyses presented in Section IV.

A. The Serial Fast Marching Method

The here discussed serial FMM corresponds to the original method as presented in [1]. The first order Godunov-type finite difference scheme [14] is used to approximate the gradient term of the Eikonal equation. The discretized version is given by:

$$\left[\begin{array}{l} max\left(D^{-x}_{i,j,k}\phi, -D^{+x}_{i,j,k}\phi, 0\right)^2 + \\ max\left(D^{-y}_{i,j,k}\phi, -D^{+y}_{i,j,k}\phi, 0\right)^2 + \\ max\left(D^{-z}_{i,j,k}\phi, -D^{+z}_{i,j,k}\phi, 0\right)^2 \end{array}\right]^{1/2} = \frac{1}{F_{i,j,k}} \quad (2)$$

where $D^{-}_{i,j,k}$ and $D^{+}_{i,j,k}$ define the first order backward and forward difference operators, respectively, calculated in the center of each cell. The method has an $O(h)$ accuracy [1].

Note that Eq. 2 shows a special upwind structure, namely the solution at $\phi_{i,j,k}$ only depends on the neighboring cells with a smaller value. The upwind structure is the key element of the FMM algorithm, as it starts from the interface Γ and is moving away from it, processing the cells with the smallest value in each step. In order to identify the upwind directions and specify the update order, every cell is marked with one of the three status tags: (1) `KNOWN` marks the cells where the final solution is already known, (2) `BAND` marks the cells that have been already updated by a `KNOWN` neighbor, but the solution may be further updated by a new `KNOWN` neighbor, and (3) `FAR` marks the cells where the solution has not been updated yet, namely the cells that do not have `KNOWN` neighbors.

The FMM update loop works as follows. Initially all cells are marked as `FAR` and the values are set to infinity. The cells on the interface Γ are first marked as `KNOWN`. Then the neighbors of the interface cells are updated and marked as `BAND`. In order to specify the order with which the `BAND` cells are processed, they are placed in a priority queue. The cell with the minimum value is selected from the priority queue, it is assigned a `KNOWN` status and all its neighbors are updated. The newly updated cells are marked as `BAND` and added to the priority queue. This loop continues until the queue is empty, i.e., until the solution in all cells is calculated. There is also the option to restrict the calculation to a narrow band. In this case the loop continues until the minimum value in the queue exceeds the width of the narrow band.

Because of its non-iterative nature, this algorithm provides a higher accuracy compared to alternative methods. However, it relies on a global priority queue. This impedes a straightforward parallelization, as it would require a locked access mechanism to the priority queue, which would ruin parallel efficiency.

B. The Parallel Fast Marching Method

The original parallel FMM implementation was developed as a distributed-memory approach [6], but was then adapted to a shared-memory approach [7]; however, both follow the same strategy.

The first step is the domain decomposition into sub-domains. Every sub-domain has a ghost layer which is overlapping with its neighbors. Fig. 1 shows an example of a decomposition into four sub-domains. The green lines indicate where the initial domain is split; the light grey cells overlap. In our OpenMP implementation, each thread is responsible for one sub-domain.

Each sub-domain has its own priority queue, thus fast marching is performed simultaneously in each sub-domain independently of each other, which gives rise to parallelization. It is important to mention that the solution is also calculated on the ghost layer, because the information on a sub-domain containing the interface is more accurate than the one on its neighbor where the interface is not contained. After applying the FMM in each sub-domain, there is a synchronized data exchange of the overlapping data. Updated values in the overlapping regions are *communicated* via shared-memory access to the respective neighbors and placed in a buffer. The communicated values are integrated under the condition that they are smaller (in absolute value) than the local ones as

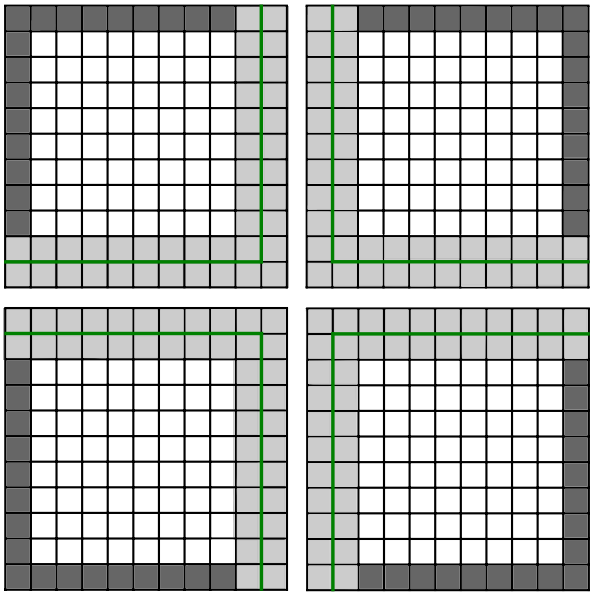


Fig. 1: Partition of the computational domain in four sub-domains. The green lines show where the partition is split, the light grey zones represent the cells in the overlapping regions, and the dark grey zones represent the ghost cells that do not overlap.

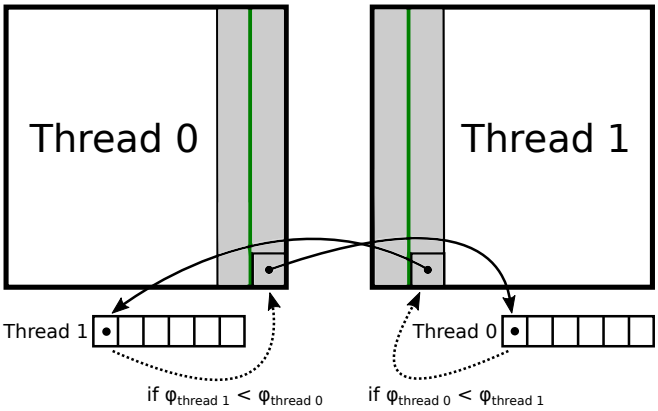


Fig. 2: Data exchange: The updated values of the overlap cells of each thread are copied to a thread-exclusive buffer of the neighbor. A communicated value is integrated into the neighbor only if it is smaller than the local value, upholding the FMM's fundamental upwind principle.

shown in Fig. 2. The newly integrated values are added to the priority queue and the fast marching loop is restarted. This procedure is repeated until no data integrations occur after the data exchange. In case that a narrow band width is specified the loop continues until the minimum value among all priority queues is smaller than the width of the narrow band. To avoid unnecessary restarts, which would degrade performance, neighboring values, which differ from the local values by less than a small value (close to the order of the rounding error, e.g., 10^{-12}), are automatically dismissed from the local integration mechanism. Neither the performance nor the accuracy of the overall method are degraded by this additional step.

The Parallel FMM algorithm has a parameter that puts an upper limit to how far (from the interface) the computation will proceed before data is exchanged. This is done in order to regulate the workload among threads. This parameter, called `stride`, is a freely chosen parameter. The fast marching loop is terminated, when the solution has moved further than a length of `stride` away from the beginning of the calculation. When there are no newly integrated data, the fast marching loop continues until the queues are empty or the narrow band threshold is reached. With the `stride` set to zero there will be data exchanges every time a cell is updated. Setting the `stride` to infinity means that the computation will proceed until the local priority queues are empty, effectively meaning that the entire domain is processed.

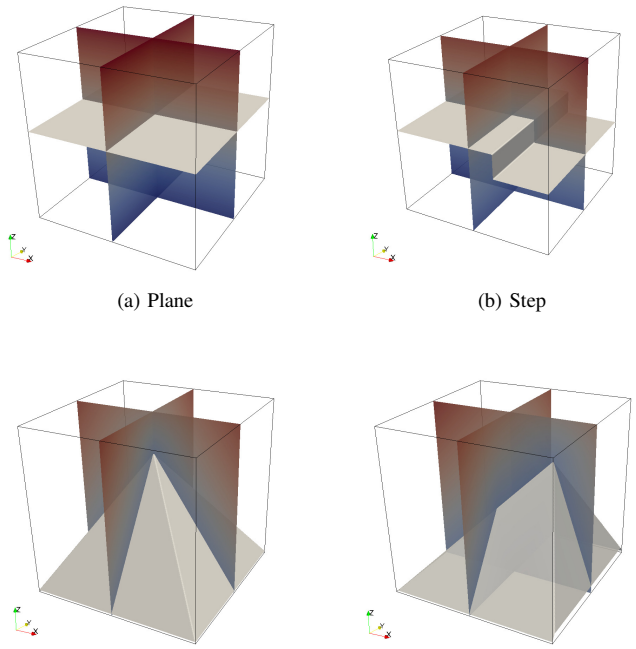
Despite the promising potential of the Parallel FMM and its potential to be used in different parallelization scales (i.e. small scale via a shared-memory approach or large scale via a distributed-memory approach) there are three challenges for the parallel performance. The first one is the relative volume of the overlapping regions where the solution is calculated by more than one thread. This introduces additional work, especially when the number of threads is large. The second is the number of restarts of the fast marching loop after data exchange, which could lead to recalculation in some cells. The third challenge comes from the explicit barriers required in order to prevent race conditions. In the current implementation there is one explicit barrier before the data integration step and another one before the global termination criterion is checked. Aside from that, no further barriers or locks are required.

IV. Analysis

A. Test cases

In order to rigorously evaluate the accuracy and the performance of the shared-memory parallel FMM algorithm, the test cases shown in Fig. 3 are introduced. The *plane* in Fig. 3a is the simplest test case. In this case, the numerical results are expected to have no numerical errors, which is reflected in the results in Table I. For the plane test case any partition in x- and/or y-direction results in an even distribution of the interface among the sub-domains, therefore promising an optimal load balance. For the *step* interface in Fig. 3b, numerical errors are expected around the edges of the step and the domain decomposition does not result in an even distribution unless a y-direction partitioning scheme is used. Fig. 3c and 3d show a *straight* and a *shifted pyramid*, respectively. The edges and the sharp corner at the peak are expected to introduce numerical errors. The straight pyramid can be evenly distributed only in the case of a partition in the x- or y-direction with two sub-domains or an xy-partition with four sub-domains (two in each direction). On the other hand, the shifted pyramid has this property only for the case that the decomposition splits the

domain in two in the y-direction. As it is displayed in Table I, for the last three test cases (Fig. 3b, 3c, 3d) the error is in the order of the resolution and in the worst cases roughly two and a half times the resolution. Thus, the error is in the expected $O(h)$ order. It is also important to mention that comparing the results of the parallel FMM with the serial version, the maximum difference is in the order of 10^{-13}, verifying the correctness of the parallel algorithm. These four test cases can have load-balanced and imbalanced domain decompositions, which is important in order to have a detailed analysis of the parallel performance of the FMM, which will be investigated in the following section.



(a) Plane (b) Step

(c) Straight Pyramid (d) Shifted Pyramid

Fig. 3: Test cases in the domain $\Omega = [-0.96, 0.96]^3$: (a) plane in XY, (b) planar surface in XY with a step, (c) pyramid with the peak at $(0, 0, 0.5)$, (d) pyramid with the peak at $(0.5, 0, 0.5)$. The respective interface is in grey. The numerical solution is displayed for all test cases, high values are represented with red hues and low values with blue hues.

	Plane	Step	Straight Pyramid	Shifted Pyramid
$h = 0.02$	10^{-13}	0.0277	0.0262	0.0505
$h = 0.01$	10^{-13}	0.0163	0.0199	0.0278
$h = 0.005$	10^{-13}	0.0094	0.0094	0.0109

TABLE I: Maximum numerical error for various resolutions h.

B. Benchmarking Platform

Benchmarking results have been produced on a single node of the Vienna Scientific Cluster 3 (VSC-3). A VSC-3 node has two sockets, each offering an 8-core Intel Xeon E5-2650v2 Ivy Bridge-EP processor, running at 2.6 GHz with 20MB of L3 cache. Therefore, the node offers 16 physical and 32 logical cores which are accompanied by 64 GB of DDR3 memory.

C. Parallel Performance Results

The parallel performance of the four test cases is tested for partitioning in x-direction, y-direction, and xy-direction. For the straight pyramid a z-direction partitioning is also tested in order to cover the influence of different decomposition strategies for a more intricately shaped interface. However, this z-decomposition analysis is only presented for the straight pyramid, as the behavior is similar for the shifted pyramid. For the xy-partition it is not always possible to have an equal number of threads in each direction, thus a representative selection has been made: 1x1, 1x2, 2x2, 2x4, 4x4. In all test cases the partitions are equidistant in each direction.

The runtime and parallel speedup for the plane problem are shown in Fig. 4. A good parallel scalability is observed for all domain decompositions. The reason is that the part of the interface which each thread owns has the same shape, and thus represents a load-balanced situation. However, the speedup is not ideal. As it is mentioned in Section III-B there are overlapping regions among the threads and the solution is also calculated in the ghost layer. This is leading to an increased overall amount of work, since some parts of the domain are calculated by more than one thread. As the number of threads is increasing, the number of gridpoints in those regions is also increasing, leading to a higher overall workload. Therefore, the parallel performance is affected.

For the step problem, Fig. 5, the parallel performance is good for the decomposition in y-direction but poor for the xy-decomposition and even poorer for the x-direction. Decomposing the domain in y-direction results in a load-balanced domain decomposition, similarly to the plane problem. Here, the speedup is even higher than for the step, because the overall workload is higher (comparing the runtime for one thread), thus parallelization has a better effect. On the other hand, a domain decomposition in x-direction results in an inhomogeneous decomposition. The step itself belongs to one thread which has to communicate the calculated values to its neighbors. Then, the fast marching loop is repeated. The number of repetitions (i.e., restarts) is displayed in Table II. Restarting the calculation means, in this case, that additional work is required in order to compute the solution, which obviously degrades the performance. The performance for the xy-decomposition is affected by that and the x-direction decomposition shows a very poor performance, because a large number of restarts is performed.

For the straight pyramid problem, as discussed in Section IV-A, an even distribution of the interface is possible in the case that the domain is split in two in x-and/or y-direction. Therefore, as it is shown in Fig. 6, a linear speedup is observed for the x-and y-decomposition for two threads and in xy-decomposition for four threads. The worst case for this

problem is the z-direction decomposition. As it is shown in Table III, the number of restarts increases dramatically, when the number of threads is increasing. In fact, for 8 and 16 threads, some threads do not own any part of the interface and they remain idle until the first communication step. Then, after the first restart, the workload of these threads is higher than the others leading to a completely imbalanced load distribution.

On the other hand, for the shifted pyramid, as shown in Fig. 7, the speedup is acceptable only in the case of the y-decomposition with two threads. In all other cases the interface is unevenly distributed. Especially the peak of the pyramid is not owned by more than two threads, but it influences the computation in the biggest portion of the domain. Therefore, as shown in Table IV, the large number of restarts decimates the performance.

Threads	**2**	**4**	**8**	**16**
X	1	1	3	6
Y	0	0	0	0
XY	0	1	1	1

TABLE II: Number of restarts for the step case.

Threads	**2**	**4**	**8**	**16**
X	1	2	4	7
Y	1	2	4	7
XY	1	2	3	3
Z	1	2	5	10

TABLE III: Number of restarts for the straight pyramid case.

Threads	**2**	**4**	**8**	**16**
X	1	3	5	10
Y	1	2	3	7
XY	1	2	3	5

TABLE IV: Number of restarts for the shifted pyramid case.

The effect of the `stride` parameter (cf. Section III-B) is studied for the step problem with x-direction partitioning. In Fig. 8, different values of `stride` are tested. As it can be seen, using a `stride` smaller than infinity reduces the runtime slightly but it does not bring any significant improvement to the performance. Different `stride` parameters are also tested for other test cases but only minor improvements have been noticed.

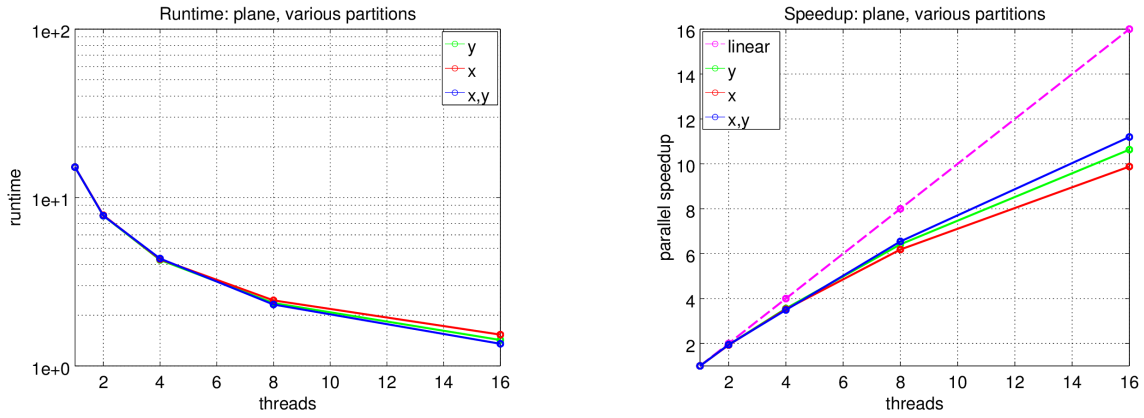
In order to investigate the influence of different grid sizes on the parallel efficiency the step problem with y-direction decomposition, which shows a good scalability, is chosen. In Fig. 9, the performance for three different grid sizes is presented. Here, the grid performance for the 384^3 grid (i.e. the green lines) is the same as presented in Fig. 5. As it is shown in Fig. 9 the performance for up to 8 threads is scalable for all grid sizes. However, for 16 threads the performance with the smaller grids is suboptimal: A decomposition of the domain in 16 parts, leaves the grid which has initially 96 points only with 8 grid points per sub-domain in y-direction (6 after decomposition plus 2 of the ghost layer), and the grid with initially 192 points only with 18 points per sub-domain in y-direction. The computational time in this cases is very small such that the cost of OpenMP parallelization and the explicit barriers (cf. Section III-B) overcomes the benefits of parallelization. Moreover, for 16 threads the effect of NUMA is present that further increases the parallelization overhead.

V. Conclusion

The parallel FMM, has been evaluated for interfaces inspired by practical problems and for different domain decomposition scenarios. The parallel FMM is widely applicable and has a straight-forward implementation. In terms of parallel performance, the method shows good scalability for cases where the interface is (close to) evenly partitioned among the sub-domains. However, the performance is not ideal due to the overlapping nature of the domain decomposition. On the other hand, poor performance has been noticed in case of an uneven domain decomposition. In addition to this, the choice of the stride parameter does not contribute any significant improvement to the performance. Our analyses show the parallel efficiency limits of the parallel FMM's static domain decomposition approach. Therefore, future work will investigate dynamic domain decomposition approaches to further increase the speedup.

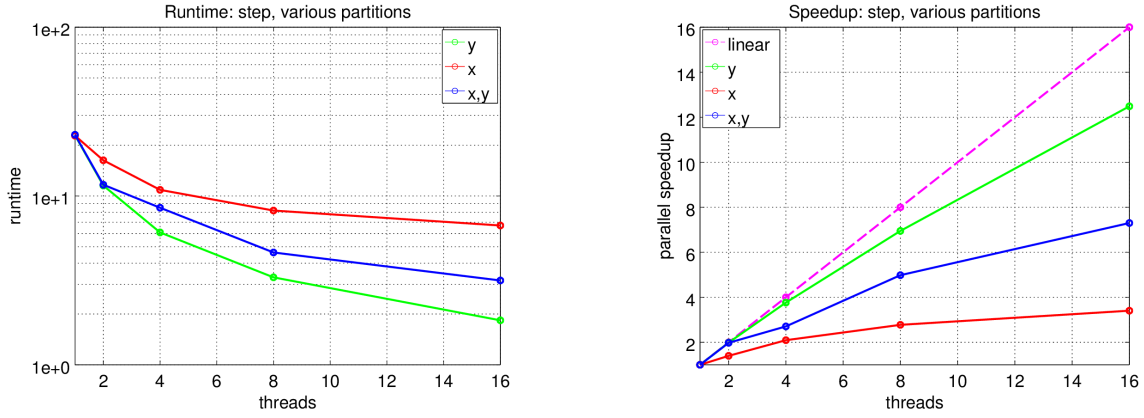
Acknowledgment

The financial support by the Austrian Federal Ministry of Science, Research and Economy and the National Foundation for Research, Technology and Development is gratefully acknowledged. The presented computational results have been achieved using the Vienna Scientific Cluster (VSC).



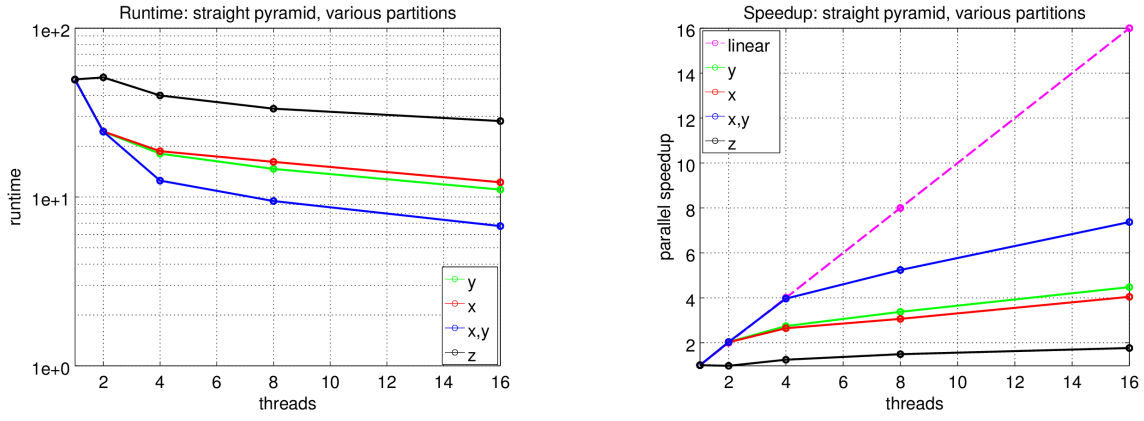
(a) (b)

Fig. 4: Runtime (a) and parallel speedup (b) for the plane test case with various partitions. Resolution $h = 0.005$, 384 gridpoints in each direction, `stride` is set to infinity.



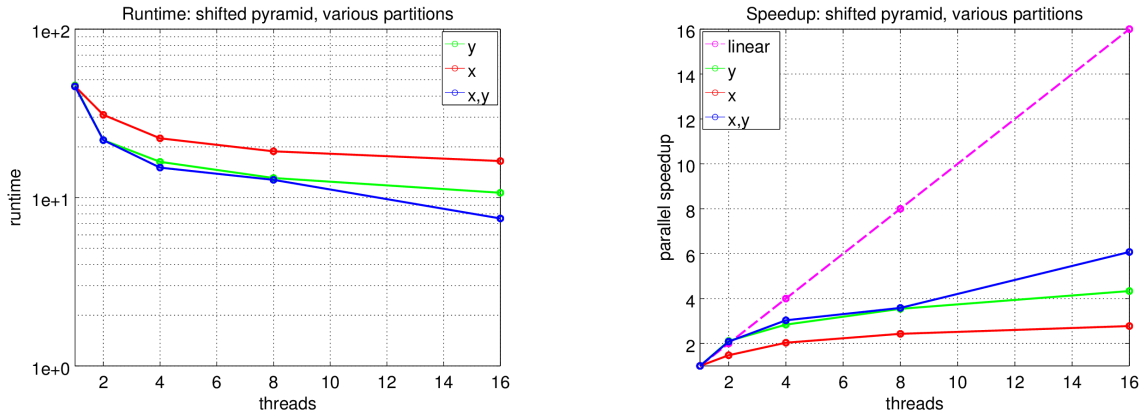
(a) (b)

Fig. 5: Runtime (a) and parallel speedup (b) for the step test case with various partitions. Resolution $h = 0.005$, 384 gridpoints in each direction, `stride` is set to infinity.



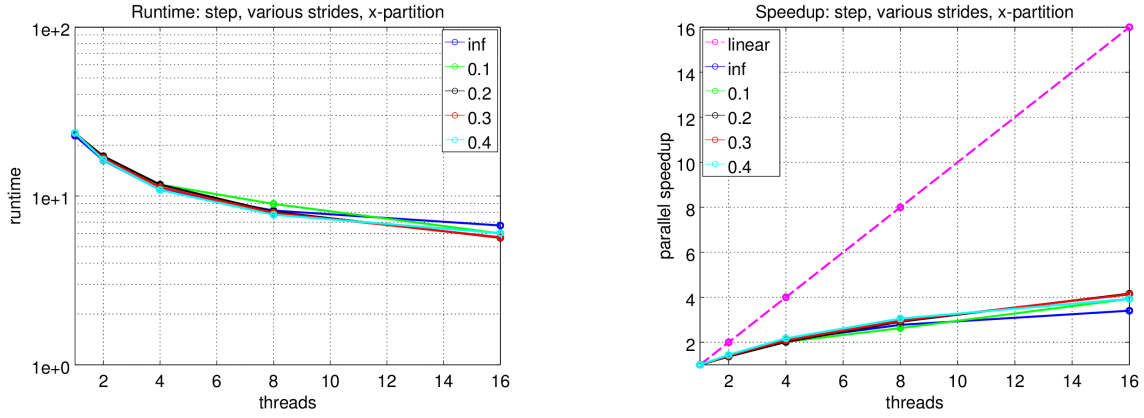
(a) (b)

Fig. 6: Runtime (a) and parallel speedup (b) for the straight pyramid test case with various partitions. Resolution $h = 0.005$, 384 gridpoints in each direction, `stride` is set to infinity.



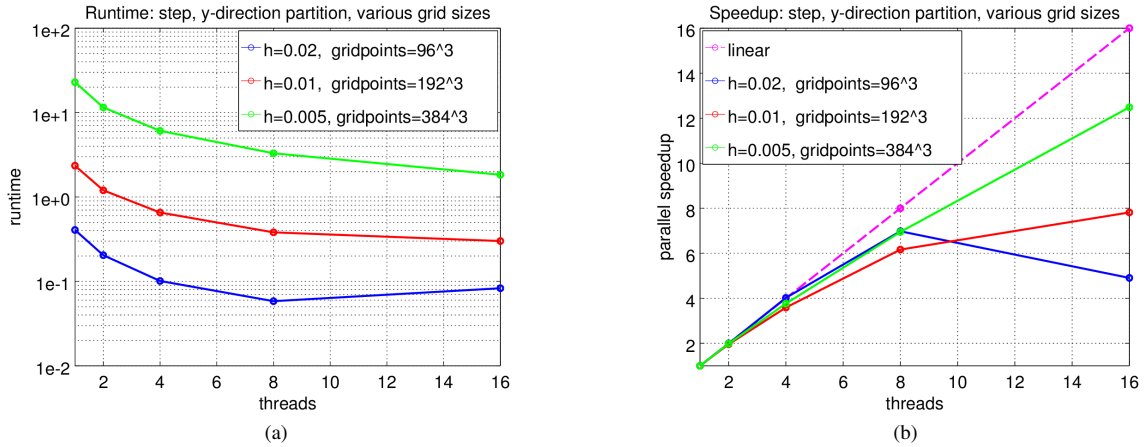
(a) (b)

Fig. 7: Runtime (a) and parallel speedup (b) for the shifted pyramid test case with various partitions. Resolution $h = 0.005$, 384 gridpoints in each direction, `stride` is set to infinity.



(a) (b)

Fig. 8: Runtime (a) and parallel speedup (b) for the step test case, with partition in x-direction and various `stride` values. Resolution $h = 0.005$, 384 gridpoints in each direction.



(a) (b)

Fig. 9: Runtime (a) and parallel speedup (b) for the step test case, with partition in y-direction and various grid sizes. `Stride` is set to infinity.

REFERENCES

[1] J. A. Sethian, *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*, 2nd ed. Cambridge University Press, 1999.

[2] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959, doi:10.1007/BF01386390.

[3] M. Herrmann, "A Domain Decomposition Parallelization of the Fast Marching Method," in *Anual Research Briefs*. Center for Turbulence Research, Stanford University, 2003, pp. 213–225.

[4] M. Breuß, E. Cristiani, P. Gwosdek, and O. Vogel, "An Adaptive Domain-Decomposition Technique for Parallelization of the Fast Marching Method," *Applied Mathematics and Computation*, vol. 218, no. 1, pp. 32–44, 2011, doi:10.1016/j.amc.2011.05.041.

[5] P. Kotas, R. Croce, V. Poletti, V. Vondrak, and R. Krause, "A Massive Parallel Fast Marching Method," in *Domain Decomposition Methods in Science and Engineering XXII*. Springer, 2016, pp. 311–318, doi:10.1007/978-3-319-18827-0_30.

[6] J. Yang and F. Stern, "A Highly Scalable Massively Parallel Fast Marching Method for the Eikonal Equation," *Journal of Computational Physics*, vol. 332, pp. 333–362, 2017, doi:10.1016/j.jcp.2016.12.012.

[7] J. Weinbub and A. Hössinger, "Shared-Memory Parallelization of the Fast Marching Method Using an Overlapping Domain-Decomposition Approach," in *Proceedings of the 24th High Performance Computing Symposium*, 2016, pp. 18:1–18:8, doi:10.22360/SpringSim.2016.HPC.052.

[8] ——, "Comparison of the Parallel Fast Marching Method, the Fast Iterative Method, and the Parallel Semi-Ordered Fast Iterative Method," *Procedia Computer Science*, vol. 80, pp. 2271–2275, 2016, doi:10.1016/j.procs.2016.05.408.

[9] H. Zhao, "A Fast Sweeping Method for Eikonal Equations," *Mathematics of Computation*, vol. 74, no. 250, pp. 603–627, 2005, doi:10.1090/S0025-5718-04-01678-3.

[10] ——, "Parallel Implementations of the Fast Sweeping Method," *Journal of Computational Mathematics*, vol. 25, no. 4, pp. 421–429, 2007.

[11] W.-K. Jeong and R. T. Whitaker, "A Fast Iterative Method for Eikonal Equations," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2512–2534, 2008, doi:10.1137/060670298.

[12] 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, doi:10.1137/10080909X.

[13] ——, "A Parallel Two-Scale Method for Eikonal Equations," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. A156–A180, 2015, doi:10.1137/12088197X.

[14] E. Rouy and A. Tourin, "A Viscosity Solutions Approach to Shape-From-Shading," *SIAM Journal on Numerical Analysis*, vol. 29, no. 3, pp. 867–884, 1992, doi:10.1137/0729053.