Shared-Memory Parallelization of the Fast Marching Method Using an Overlapping Domain-Decomposition Approach

Josef Weinbub
Christian Doppler Laboratory
for High Performance TCAD
Institute for Microelectronics
Gußhausstraße 27-29/E360
TU Wien, 1040 Wien, Austria
weinbub@iue.tuwien.ac.at

Andreas Hössinger
Silvaco Europe Ltd.
Compass Point, St Ives
Cambridge, PE27 5JL
United Kingdom
andreas.hoessinger@silvaco.com

ABSTRACT

The fast marching method is used to compute a monotone front propagation of anisotropic nature by solving the eikonal equation. Due to the sequential nature of the original algorithm, parallel approaches presented so far were unconvincing. In this work, we introduce a shared-memory parallelization approach which is based on an overlapping domain decomposition technique. We introduce our parallel algorithm of the fast marching method tailored to shared-memory environments and discuss benchmark results based on a C++ implementation using OpenMP. We compare the sequential execution performance as well as the accuracy with reference implementations of the fast marching method and the fast iterative method; the latter is also used to evaluate the parallel scalability. Our shared-memory parallel fast marching method convinces both with regard to serial and parallel execution performance as well as with respect to accuracy.

Author Keywords

Fast marching method; parallel algorithm; domain decomposition; shared-memory; eikonal equation

ACM Classification Keywords

G.1.0 NUMERICAL ANALYSIS: General—Parallel algorithms

1. INTRODUCTION

Simulating an expanding front is a fundamental step in many computational science and engineering applications, such as image segmentation [5], brain connectivity mapping [12], medical tomography [11], seismic wave propagation [13], geological folds [7], semiconductor process simulation [18], or computational geometry [16].

In general, an expanding front originating from a start position Γ is described by its first time of arrival T to the points of a domain Ω. This problem can be described by solving the eikonal equation [15], which for n spatial dimensions reads:

$$\begin{aligned} \|\nabla T(\mathbf{x})\|_2 &= f(\mathbf{x}) \quad \mathbf{x} \in \Omega \subset \mathbb{R}^n, \\ T(\mathbf{x}) &= g(\mathbf{x}) \quad \mathbf{x} \in \Gamma \subset \Omega. \end{aligned}$$

$T(\mathbf{x})$ is the unknown solution (i.e. first time of arrival), $g(\mathbf{x})$ are boundary conditions for Γ, and $f(\mathbf{x})$ is an inverse velocity field, i.e., $f(\mathbf{x}) = 1/F(\mathbf{x})$: $F(\mathbf{x})$ is a positive speed function, with which the interface information propagates in the domain. Generally speaking, isosurfaces to the solution represent the position of the front at a given time, and can thus be regarded as the geodesic distance relative to Γ. If the velocity $F = 1$, then the solution $T(\mathbf{x})$ represents the minimal Euclidian distance from Γ to $\mathbf{x}$.

The most widely used method for solving the eikonal equation is the fast marching method (FMM) [15]. The FMM solves the eikonal equation in a single pass, allowing to track the evolution of an expanding front (cf. Section 2). This method is inherently sequential and attempts to parallelize it have so far been unsatisfactory [1][11]. However, recent investigations regarding a domain decomposition technique in a large-scale distributed-memory setting are promising [19], which is the basis for this work.

Other prominent approaches for solving the eikonal equation are available, the most prominent among them are the fast sweeping method (FSM) [20] and the fast iterative method (FIM) [10]. Both methods support parallel execution, where, contrary to the FSM, the FIM supports fine-grained parallelism: The FIM inherently offers greater potential for parallelism over the entire spectrum than the coarse-grained parallelism of FSM. Therefore, the FIM is used in this work as a frame of reference regarding parallel scalability. FIM was originally implemented for parallel execution on Cartesian meshes and later extended to triangular surface meshes [6]. FIM relies on a modification of a label correction scheme coupled with an iterative procedure for the mesh point update. The inherent high degree of parallelism is due to the ability of processing all nodes of an active list (i.e. narrow band) in parallel, thus efficiently supporting a single instruction, multiple data parallel execution model. Therefore, FIM is suitable for implementations on highly parallel accelerators, such as graphics adapters [10][11]. Although FIM has been primarily investigated regarding fine-grained parallelism on accelerators, investigations on shared-memory approaches have also been conducted [3][4][18]. Overall, based on those major methods, i.e., FMM, FSM, and FIM, several derivative techniques have been developed [8][9][17].

SpringSim-HPC 2016, April 3-6, Pasadena, CA, USA

Although methods are available which inherently favor parallelism, such as the FSM and the FIM, the fact that the FMM is a one-pass algorithm (i.e. non-iterative in nature) offers significant advantages with respect to accuracy. This fact makes the original serial FMM still one of the predominant methods to compute expanding fronts; even in today's parallel computing age. In particular, there are two major advantages of the FMM over other techniques [19] which represent the primary motivation for parallelizing the FMM: (1) the narrow band formulation and (2) the monotonic increasing order of the solution.

In this work, we focus on three-dimensional Cartesian grids and use an overlapping domain decomposition technique for parallelizing the FMM which fits the performance critical data locality aspects paramount to modern shared-memory environments. In Section 2 we give a short overview of the sequential FMM. In Section 3, we introduce our parallel, shared-memory FMM. In Section 4 we investigate the execution performance and accuracy of an OpenMP implementation of our parallel FMM by comparing it with two reference implementations: a pure serial version of the FMM and an OpenMP-parallelized implementation of the FIM.

2. SEQUENTIAL FAST MARCHING METHOD

In this section, we provide a short overview of the original, sequential FMM. A detailed description of the inner workings of the FMM is provided in [15] and [19]. To approximate the differential operator of the eikonal equation, we consider the widely applied Godunov-type finite difference scheme [14] in combination with first-order discretizations for the spatial derivatives. This approach offers a special upwind structure, i.e., a new solution only depends on the neighboring cells having a smaller solution value. The FMM makes use of this fact by solving the eikonal equation using only the upwind cells. For identifying the upwind directions three flags are used to indicate the status of a certain cell: (1) `KNOWN` indicates that a cell contains a final solution thus give the upwind direction, (2) `BAND` denote cells which contain solutions updated by its neighboring `KNOWN` cells but may be further updated by any new `KNOWN` neighbors, and (3) `FAR` identifies cells which are on the downwind side and do not have any `KNOWN` neighbors yet. The general algorithm is such that the cell with the smallest value in the `BAND` set is identified and re-tagged as `KNOWN`. It's neighboring `BAND` and `FAR` cells can be updated.

The identification of the cell with the smallest value is usually being implemented via a minimum heap data structure, which in a serial computing setting provides excellent performance. However, in this lies the reason why the FMM does not favor straightforward parallel approaches: The minimum heap data structure is hardly parallelizable as it would require a synchronized data access among the threads.

In a pre-processing step, all flags of the grid cells (i.e. the discretization of the domain) are set to `FAR` and the initial solution values for all cells are set to ∞, which ensures that the upwind scheme forces the front (i.e. the set of `BAND` cells) to reach all yet unprocessed cells of the grid. By initializing the values of the interface cells, i.e., cells relative to which the distances have to be computed by the FMM, and re-tagging those to `KNOWN` the FMM's front starts to propagate until all cells of the grid have been processed and thus the distances are computed. Figure 1 shows the fundamental principle of the FMM.

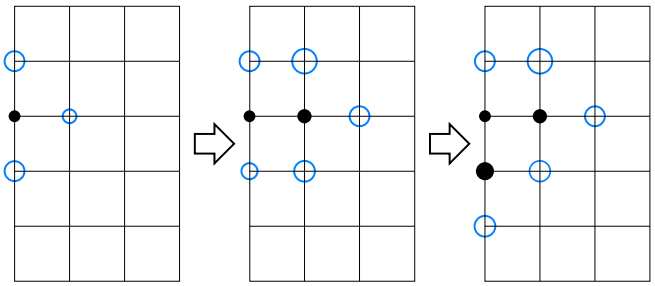

Figure 1: Schematic representation of the FMM. Full, black circles (`KNOWN`) denote computed travel times that are set. Blue, empty circles (`BAND`) refer to the front of the FMM. Grid points with no circles indicate yet unprocessed cells (`FAR`). The *travel time* (i.e. the solution value) is schematically depicted by the size of the circles; the larger the radius the greater the travel time. The FMM processes the cells starting with smallest travel time.

3. SHARED-MEMORY PARALLELIZATION

In this section we introduce our shared-memory approach for parallelizing the FMM using an overlapping domain decomposition technique. We partition the computational domain in, if possible, equal parts (i.e. subgrids) where each thread is responsible for a specific subgrid. A single ghost layer is used in all interior spatial decomposition directions to ensure that the parallel algorithm properly computes the entire domain; the updates at the *inner* boundary cells (i.e. boundary cells introduced by the partitioning scheme) are forwarded to the neighboring threads allowing the local solutions to influence the global solution process. By using a domain decomposition technique, which is due to the use of ghost zones to be considered an *overlapping* domain decomposition approach, each thread (and thus partition) has its own minimum heap data structure. Thereby the primary reason for the FMM to not favor parallelization is eliminated, as no synchronized access - avoiding race conditions - has to be implemented. The eikonal equation is solved by the threads on their individual subgrid, including the ghost cells, which gives rise to parallelism. The general idea for decomposing the domain is shown in Figure 2 for a two-dimensional case.

The key ingredient to our shared-memory domain decomposition approach is the parallel grid data structure. Each subgrid is represented by a dedicated grid data structure which is specialized regarding the corresponding thread identification and the total number of threads: Each subgrid derives from this thread information its neighbor communication configuration via index computations which is automatically generated upon initialization. We are using a straightforward uniform three-dimensional block decomposition method, i.e., each spatial dimension is split according to the number of threads.

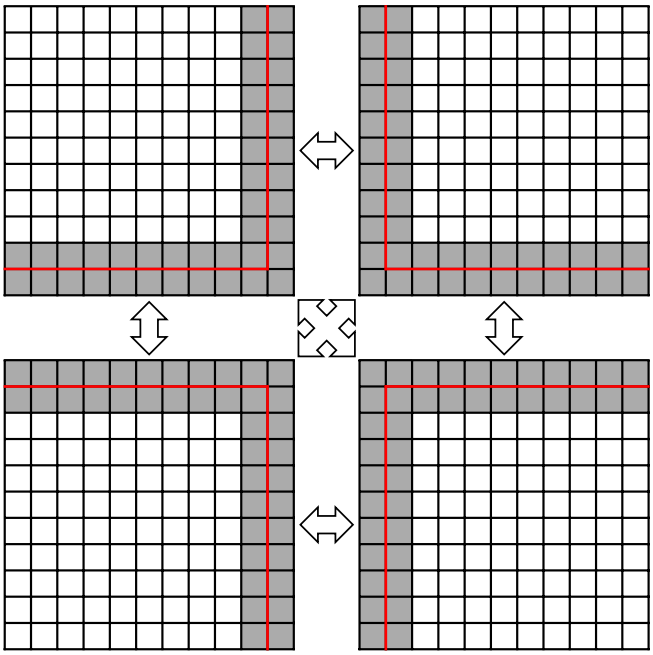

Figure 2: Two-dimensional schematic depicting the domain decomposition scheme for four threads used for our parallel FMM. The initial computational domain is split at the red lines which also indicate the overlap interfaces. A single ghost layer (grey cells) is used to ensure a proper solution coupling of the parallel algorithm.

For instance, when four threads are used, the x and y dimension is partitioned into two parts, whereas the z dimension is left untouched, thus decomposing the computational domain into four parts. In turn, for sixteen threads we are partitioning the x dimension into four parts, whereas the y and z dimension are partitioned in two parts and so forth.

To support non-uniform memory access (NUMA) environments, each thread instantiates and initializes its own subgrid data structure. The master thread holds an array of pointers to the individual subgrids, allowing shared access to neighboring subgrids for the thread team. This approach ensures that the threads' compute-intensive loops are executed on their respective locality domains, avoiding NUMA traffic and thus ultimately increasing parallel scalability. However, NUMA traffic cannot be entirely avoided as the ghost layer communication requires access to neighboring subgrids, triggering memory access to remote locality domains via the indicated shared global grid data structure. But as communication is - relative to the compute-intensive loops - much less required in the overall scheme, the performance hit of communication-related NUMA traffic is limited. Algorithm 1 shows a pseudocode depicting the NUMA-aware grid setup.

The overall algorithm for the parallel FMM is shown in Algorithm 2, which to a large degree follows the work in a distributed-memory setting [19]. However, the communication mechanisms used in our approach utilize the available shared-memory features. In particular, a reduction method

Algorithm 1 NUMA-aware setup of the parallel grid data structure suitable for domain decomposition

```
1: #Shared-array holds pointers to subgrids
2: grid ← subgrid[threads]
3: #omp parallel
4: {
5:    #Create and setup subgrid in proper locality domain
6:    grid[this_thread] ← new subgrid(cells)
7:    #Each thread executes the parallel FMM algorithm
8:    fmm_parallel(grid)
9: }
```

Algorithm 2 The shared-memory, parallel FMM algorithm (fmm_parallel(grid)) executed by all threads: T solution array, G flag array, NB the narrow band (i.e. heap), LC_{global} loop condition.

```
1: subgrid ← grid[this_thread]
2: T ← ∞
3: G ← FAR
4: NB ← ∅
5: initialize_interface (subgrid, T, G)
6: initialize_ heap (subgrid, T, G, NB)
7: while 1 do
8:    #Barrier: Ensure reduction obtains latest global data
9:    for subgrid ∈ grid do
10:      LC_global ← LC_local
11:   end for
12:   if not LC_global then
13:      Break loop
14:   end if
15:   march_narrow_band(subgrid, T, G, NB)
16:   exchange_overlapping_data(grid, T, G)
17:   #Barrier: Ensure remote write operations finished
18:   integrate_overlapping_data(subgrid, T, G, NB)
19:   march_narrow_band(subgrid, T, G, NB)
20: end while
```

(Lines 9-11) uses the available shared access to the global grid data structure to compute the loop condition variables, e.g., the global number of heap elements - used to terminate the overall algorithm - is calculated by accessing the remote heap sizes of each individual subgrid. A synchronization point is required prior to the shared-access to make sure that all threads can provide their latest information on their progress.

The march_narrow_band method performs a regular FMM step, i.e., the entire local heap is processed until it is empty. To ensure a proper coupling of the FMM's algorithm between the subgrids, a ghost layer exchange mechanism is utilized, which is schematically depicted in Figure 3: The exchange_overlapping_data method processes all ghost cells of the local partition and writes those to thread-exclusive receive buffers of the corresponding neighbor threads (i.e. partitions). Ghost cells are potentially transferred to more than one neighbor partition (Figure 2). Due to the use of thread-exclusive receive buffers, no write access guards are required.

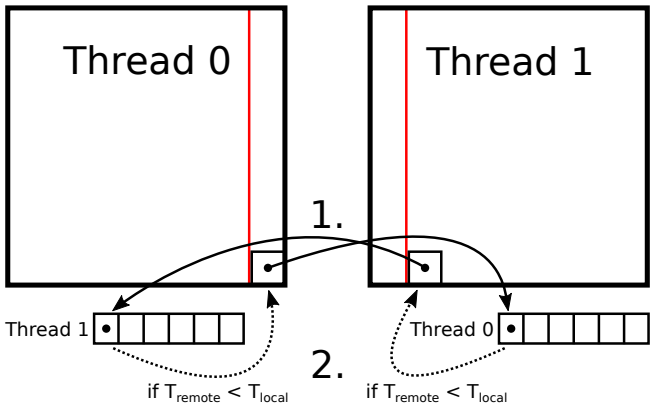

Figure 3: The ghost cell data exchange is implemented via (1.) the exchange_overlapping_data method and (2.) the integrate_overlapping_data method: (1.) The ghost cell indices and its local solution values are transferred to a thread-exclusive target buffer of the neighboring thread; (2.) If the remote solutions are smaller than their local counterparts (to uphold the upwind condition), the local solution values will be updated accordingly.

The integrate_overlapping_data method processes all of the local thread-exclusive receive buffers (one for each neighboring thread) and merges it with the local data sets, if the remote solutions are smaller than the local ones (to satisfy the fundamental upwind principle of the FMM). Before this step can be executed, all threads must have finished their write operations - requiring a synchronization point - conducted in exchange_overlapping_data as otherwise the merging step is corrupted due to potentially unfinished write operations. As the integration method potentially introduces new cells into the local heap, an additional march_narrow_band step is required to ensure that the remote information (via the overlapping ghost cells provided by the neighbor threads) is processed locally by the FMM's algorithm. Our entire shared-memory FMM algorithm requires two synchronization points; no additional guards for avoiding race-conditions are needed.

4. RESULTS

We investigate the performance and accuracy of our parallel FMM implementation relative to a reference FMM and FIM implementation. The FIM has been chosen as a parallel comparison target due to its excellent support for parallelism [10], which thus provides a proper frame of reference to judge upon the parallel execution performance of our parallel FMM. The FIM implementation follows a fine-grained shared-memory approach [3]. Our benchmarks use established synthetic problem cases [2][3][7] and cover different three-dimensional problems with varying problem sizes (100^3 and 200^3 Cartesian cube grids using $\mathbf{0.0}$ and $\mathbf{1.0}$ for lower and upper bounds, respectively), speed functions, and single/multiple-source configurations, i.e., a single center source node versus 100 source nodes randomly spread over the entire simulation domain. The entire simulation domains are computed.

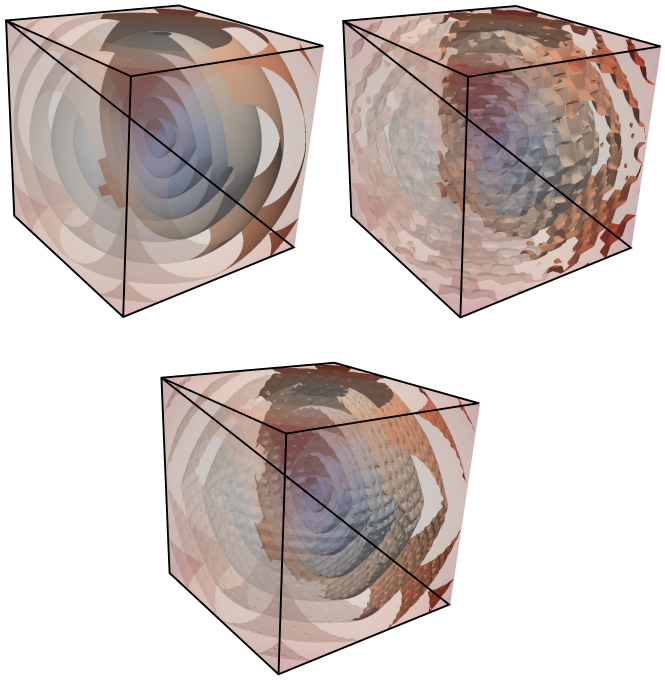

Figure 4: Isosurfaces of the F_{const} (top left), F_{check} (top right), and F_{osc} (bottom) solution on a 100^3 domain for a single center source

Regarding speed functions, we investigate three different configurations to evaluate the influence of different speed functions on the performance: (1) constant speed (F_{const}), where for the entire domain $F = 1$ is used; (2) checkerboard speed (F_{check}), where the computational domain is divided into eleven equally sized cubes in each direction and the velocity is alternated between $F = 1$ to $F = 2$ from cube to cube [2][7]; (3) oscillatory speed (F_{osc}), where the speed function is modeled by a highly oscillatory continuous speed function, being $F = 1 + \frac{1}{2}sin(20\pi x)sin(20\pi y)sin(20\pi z)$ [2]. The benchmark setups have been chosen to investigate the correlation between varying problem sizes (i.e. number of grid cells) and different speed functions as well as with various thread numbers.

Figure 4 depicts the isosurfaces of the solutions of the center test configurations for the 100^3 simulation grids, to provide a frame of reference for the benchmark setup and the solutions. The results for the 200^3 are similar, albeit offering an increased resolution.

The benchmarks have been carried out on a single compute node of the current installation (i.e. 3rd) of the Vienna Scientific Cluster[1]. The compute node provides two eight-core Intel Xeon E5-2650v2 (Ivy Bridge-EP) processors - offering a total of 16 physical and 32 logical cores using Hyperthreading - and 64 GB of DDR3 1866 ECC main memory. The Intel C++ compiler version 16.0.0 (-O3 flag) has been used as well as 64-bit floating point precision (i.e. `double`) and - for the FIM - an error threshold of $\varepsilon = 10^{-12}$ has been utilized. The fastest execution times out of five repetitions have been recorded and used for the investigation.

[1] http://vsc.ac.at/

Figure 5-7 compare the strong scaling results of our parallel FMM and the reference FIM for the 100^3 grid setup. The execution times are shown in logarithmic scale to increase the identifiability of the presented data sets, as the timings for the various setups would otherwise potentially be indistinguishable from each other. As can be seen from the results, the FIM struggles in general with complicated speed functions whereas the (parallel) FMM does not, which is to be expected [10]. However, parallel scalability for the FIM is better as compared to our parallel FMM approach. Nevertheless, for this problem size excellent speed up efficiency of around 90% for up to eight threads are achieved. A slight tendency for super-linear scaling can be identified which is due to the decomposition scheme: For one thread, the required additional decomposition logic results in a disproportionate overhead which is alleviated for higher thread numbers. However, for more than eight threads, the book keeping required for handling the larger thread numbers counters this effect again.

Figure 8-10 compare the strong scaling results of our parallel FMM and the reference FIM for a 200^3 grid. These results show that the parallel scalability improves for increased problem sizes. Our parallel FMM performs well (>70%) for up to sixteen threads, especially for multiple source configurations. For the constant speed function we can even beat the reference FIM implementation. Super-linear scaling effects can be identified similar to the 100^3 grid investigations. All in all, the results bode well for future real-world applications which offer large problem sizes (i.e. grids) as well as complicated interfaces providing a plethora of source cells.

To investigate the accuracy of the FMM, FIM, and the parallel FMM, the computed results of the single source problem with constant speed for a 100^3 grid have been compared to an analytic solution given by the Euclidian distance function. Table 1 shows the L_1, L_2, and L_∞ norms of the individual approaches, which remain constant for varying degrees of parallelism (i.e. using more than one thread does not affect the accuracy of the FIM and the parallel FMM). As can be seen from the results, the parallel FMM offers the same error norms as the reference FMM, meaning that both approaches compute exactly the same results. However, the FIM offers a larger error than the more accurate (parallel) FMM.

Table 2 and Table 3 compare the serial execution times (i.e. with one thread) of the sequential FMM, our parallel FMM, as well as the FIM for the single and multiple source problems. The results show that the parallel FMM introduces negligible overhead, which is in a single worst case around 5.5%.

Using a uniform domain decomposition approach has obvious limitations with respect to load balancing, if the sources are not (close to) equally distributed among the individual partitions. Future work will investigate dynamic partitioning approaches in the context of real-world applications to tune the partitioning to the input source configuration and thus ultimately reduce load balancing issues. Furthermore, the parallel FMM approach will be investigated with respect to mesh adaptivity by using nested mesh data structures.

	L_1	L_2	L_∞
FMM	$8 \cdot 10^{-3}$	$7.3 \cdot 10^{-5}$	$1 \cdot 10^{-3}$
parallel FMM	$8 \cdot 10^{-3}$	$7.3 \cdot 10^{-5}$	$1 \cdot 10^{-3}$
FIM	$17 \cdot 10^{-3}$	$31 \cdot 10^{-5}$	$15 \cdot 10^{-3}$

Table 1: The L_1, L_2, and L_∞ norms of the FMM, parallel FMM, and FIM are compared for the single source, 100^3 test case using constant speed.

100^3	F_{const}	F_{check}	F_{osc}
FMM	0.673322	1.02008	1.26559
parallel FMM	0.673001	1.04368	1.3274
FIM	0.44696	4.25954	20.0117
200^3	F_{const}	F_{check}	F_{osc}
FMM	8.19452	11.9365	14.7707
parallel FMM	8.06106	12.6134	15.5846
FIM	3.90304	47.9877	245.749

Table 2: Comparison of serial execution times between the FMM, parallel FMM, and FIM for the single source tests and different speed functions.

100^3	F_{const}	F_{check}	F_{osc}
FMM	0.717773	1.25512	1.44339
parallel FMM	0.715878	1.28597	1.51159
FIM	0.931384	2.76385	9.53287
200^3	F_{const}	F_{check}	F_{osc}
FMM	10.6511	17.6005	19.5338
parallel FMM	10.4973	17.9843	20.0329
FIM	10.1596	29.4321	112.024

Table 3: Comparison of serial execution times between the FMM, parallel FMM, and FIM for the multiple source tests and different speed functions.

5. CONCLUSION

An approach for parallelizing the FMM in a shared-memory setting has been introduced. Our method uses an overlapping uniform domain decomposition method and provides excellent accuracy as well as serial and parallel performance, especially when considering intricate speed setups and larger problem sizes. The overhead for implementing the shared-memory overlapping domain decomposition technique limits excellent parallel scalability for smaller problems to about eight threads. However, when increasing the problem size, excellent scalability for up to sixteen threads is achieved. This behavior bodes well for future real-world applications, offering large problem sizes and complicated source configurations which will be - in addition to load balancing techniques and mesh adaptivity - at the center of our upcoming research.

ACKNOWLEDGMENTS

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).

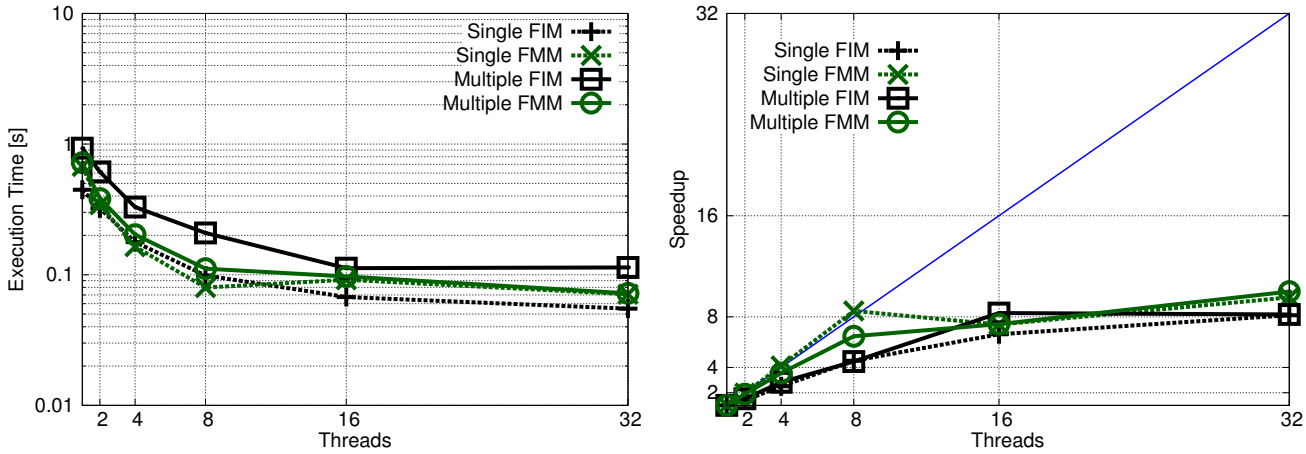

Figure 5: Execution times (left) and relative speedups (right) of the F_{const} problem on a 100^3 domain

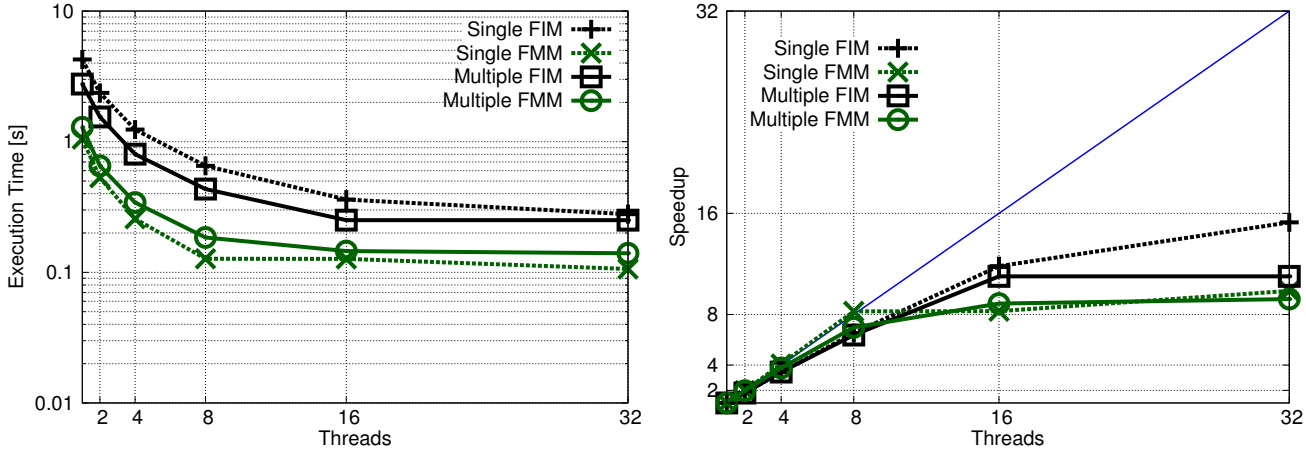

Figure 6: Execution times (left) and relative speedups (right) of the F_{check} problem on a 100^3 domain

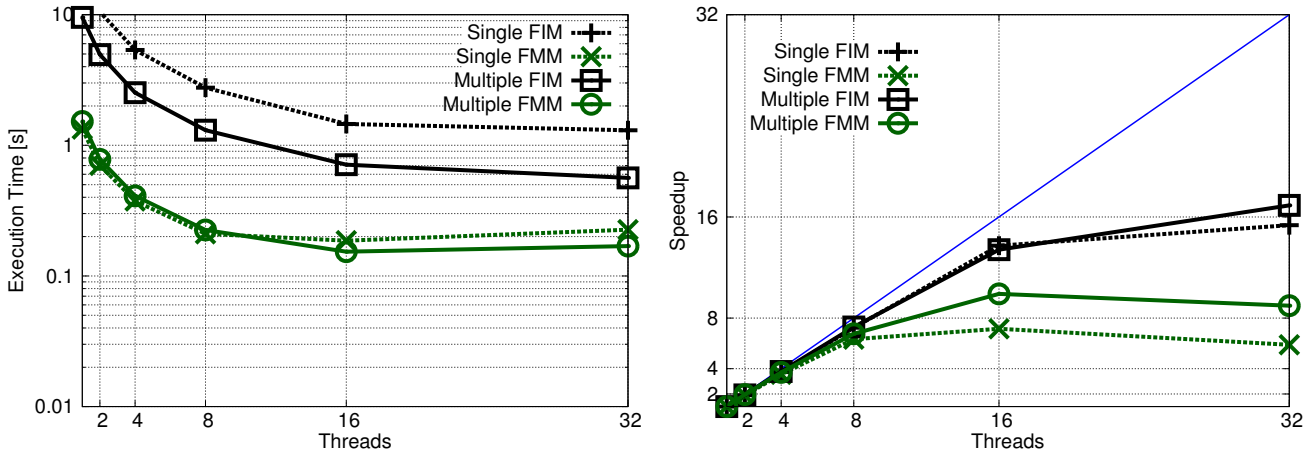

Figure 7: Execution times (left) and relative speedups (right) of the F_{osc} problem on a 100^3 domain

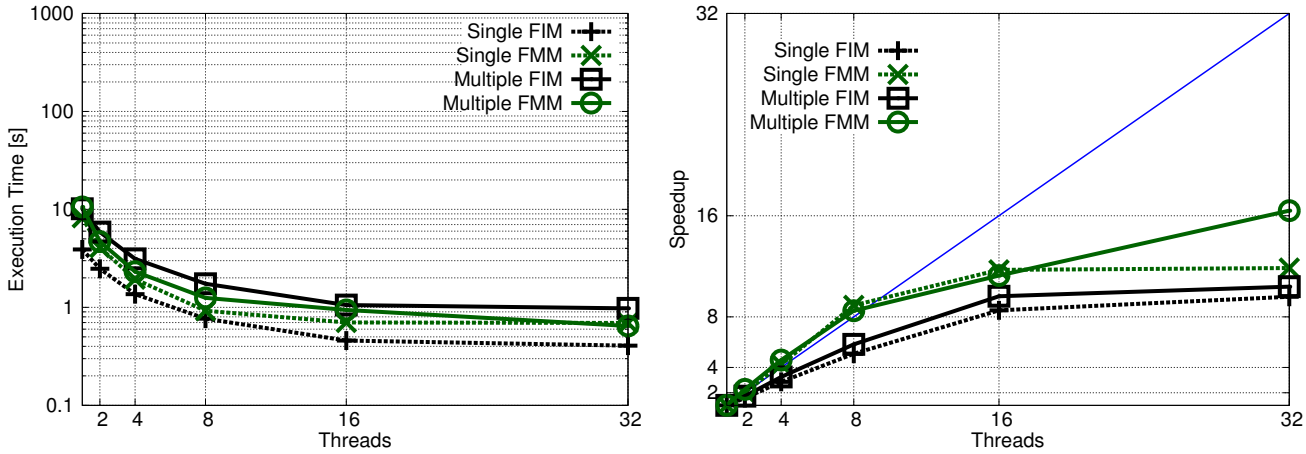

Figure 8: Execution times (left) and relative speedups (right) of the F_{const} problem on a 200^3 domain

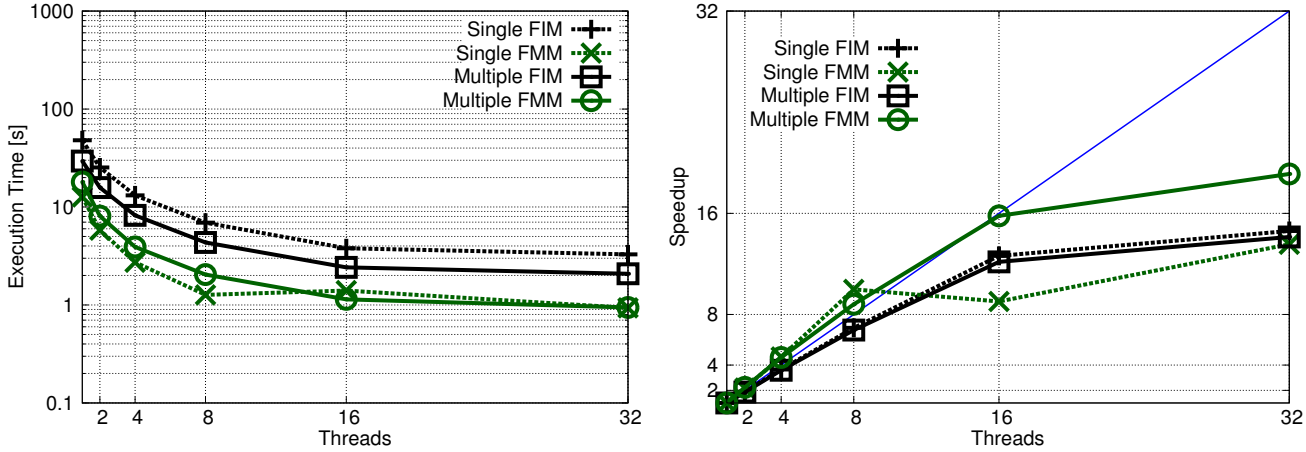

Figure 9: Execution times (left) and relative speedups (right) of the F_{check} problem on a 200^3 domain

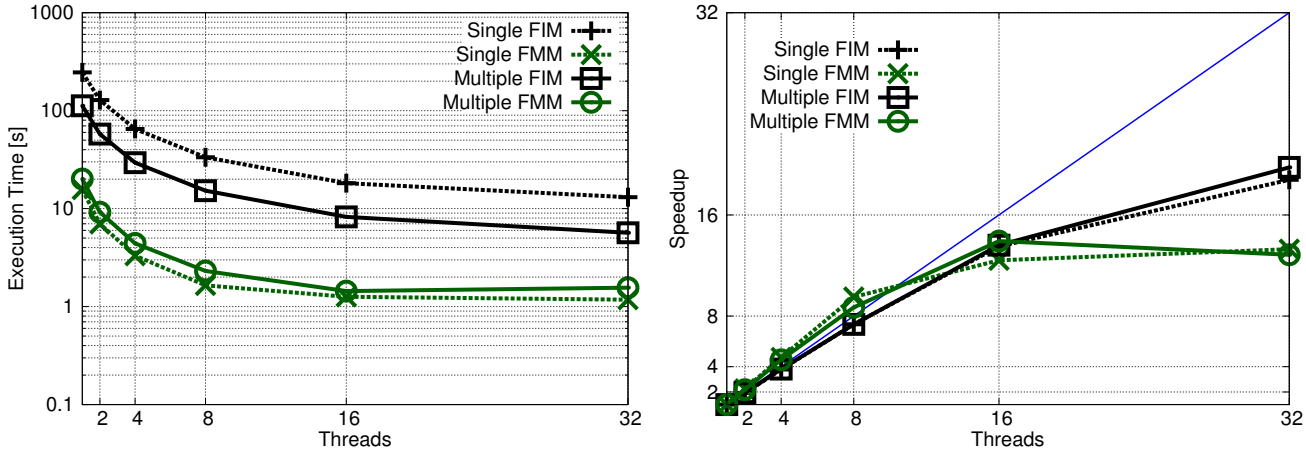

Figure 10: Execution times (left) and relative speedups (right) of the F_{osc} problem on a 200^3 domain

REFERENCES

1. Breuß, M., Cristiani, E., Gwosdek, P., and Vogel, O. An Adaptive Domain-Decomposition Technique for Parallelization of the Fast Marching Method. *Applied Mathematics and Computation 218*, 1 (2011), 32–44. DOI: 10.1016/j.amc.2011.05.041.
2. Chacon, A., and Vladimirsky, A. Fast Two-Scale Methods for Eikonal Equations. *SIAM Journal on Scientific Computing 34*, 2 (2012), A547–A578. DOI: 10.1137/10080909X.
3. Dang, F., and Emad, N. Fast Iterative Method in Solving Eikonal Equations: A Multi-level Parallel Approach. *Procedia Computer Science 29* (2014), 1859–1869. DOI: 10.1016/j.procs.2014.05.170.
4. Dang, F., Emad, N., and Fender, A. A Fine-Grained Parallel Model for the Fast Iterative Method in Solving Eikonal Equations. In *Proc. 3PGCIC* (2013), 152–157. DOI: 10.1109/3PGCIC.2013.29.
5. Forcadel, N., Le Guyader, C., and Gout, C. Generalized Fast Marching Method: Applications to Image Segmentation. *Numerical Algorithms 48*, 1-3 (2008), 189–211. DOI: 10.1007/s11075-008-9183-x.
6. Fu, Z., Jeong, W. K., Pan, Y., Kirby, R., and Whitaker, R. T. A Fast Iterative Method for Solving the Eikonal Equation on Triangulated Surfaces. *SIAM Journal on Scientific Computing 33*, 5 (2011), 2468–2488. DOI: 10.1137/100788951.
7. Gillberg, T., Bruaset, A. M., Hjelle, Ø., and Sourouri, M. Parallel Solutions of Static Hamilton-Jacobi Equations for Simulations of Geological Folds. *Journal of Mathematics in Industry 4*, 10 (2014), 1–31. DOI: 10.1186/2190-5983-4-10.
8. Gillberg, T., Sourouri, M., and Cai, X. A New Parallel 3D Front Propagation Algorithm for Fast Simulation of Geological Folds. *Procedia Computer Science 9* (2012), 947–955. DOI: 10.1016/j.procs.2012.04.101.
9. Gomez, J. V., Alvarez, D., Garrido, S., and Moreno, L. Fast Methods for Eikonal Equations: An Experimental Survey. *ACM Computing Surveys* (2015). Under review. Preprint available: `http://arxiv.org/abs/1506.03771`.
10. Jeong, W. K., and Whitaker, R. T. A Fast Iterative Method for Eikonal Equations. *SIAM Journal on Scientific Computing 30*, 5 (2008), 2512–2534. DOI: 10.1137/060670298.
11. Li, S., Mueller, K., Jackowski, M., Dione, D., and Staib, L. Physical-Space Refraction-Corrected Transmission Ultrasound Computed Tomography Made Computationally Practical. In *Lecture Notes in Computer Science*, vol. 5242 (2008), 280–288. DOI: 10.1007/978-3-540-85990-1_34.
12. Prados, E., Soatto, S., Lenglet, C., Pons, J.-P., Wotawa, N., Deriche, R., and Faugeras, O. Control Theory and Fast Marching Techniques for Brain Connectivity Mapping. In *Proc. IEEE CVPR*, vol. 1 (2006), 1076–1083. DOI: 10.1109/CVPR.2006.89.
13. Rawlinson, N., and Sambridge, M. Multiple Reflection and Transmission Phases in Complex Layered Media Using a Multistage Fast Marching Method. *Geophysics 69*, 5 (2004), 1338–1350. DOI: 10.1190/1.1801950.
14. Rouy, E., and Tourin, A. A Viscosity Solutions Approach to Shape-From-Shading. *SIAM Journal on Numerical Analysis 29*, 3 (1992), 867–884. DOI: 10.1137/0729053.
15. Sethian, J. A. A Fast Marching Level Set Method for Monotonically Advancing Fronts. *Proceedings of the National Academy of Sciences 93*, 4 (1996), 1591–1595.
16. Sethian, J. A. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1999. ISBN: 978-0521645577.
17. Weber, O., Devir, Y. S., Bronstein, A. M., Bronstein, M. M., and Kimmel, R. Parallel Algorithms for Approximation of Distance Maps on Parametric Surfaces. *ACM Transactions on Graphics 27*, 4 (2008), 104:1–104:16. DOI: 10.1145/1409625.1409626.
18. Weinbub, J., and Hössinger, A. Accelerated Redistancing for Level Set-Based Process Simulations with the Fast Iterative Method. *Journal of Computational Electronics 13*, 4 (2014), 877–884. DOI: 10.1007/s10825-014- 0604-x.
19. Yang, J., and Stern, F. A Highly Scalable Massively Parallel Fast Marching Method for the Eikonal Equation. *Journal of Computational Physics* (2015). Under revision. Preprint available: `http://arxiv.org/abs/1502.07303`.
20. Zhao, H. A Fast Sweeping Method for Eikonal Equations. *Mathematics of Computation 74*, 250 (2005), 603–627. DOI: 10.1090/S0025-5718-04-01678-3.