Advances in Computational Mathematics (2019) 45:2029–2045
https://doi.org/10.1007/s10444-019-09683-z

A shared memory parallel multi-mesh fast marching method for re-distancing

Georgios Diamantopoulos[1] · Andreas Hössinger[2] · Siegfried Selberherr[3] · Josef Weinbub[1]

Received: 29 September 2018 / Accepted: 8 March 2019 /
Published online: 11 April 2019

Abstract

A common problem arising in expanding front simulations is to restore the signed distance field property of a discretized domain (i.e., a mesh), by calculating the minimum distance of mesh points to an interface. This problem is referred to as re-distancing and a widely used method for its solution is the fast marching method (FMM). In many cases, a particular high accuracy in specific regions around the interface is required. There, meshes with a finer resolution are defined in the regions of interest, enabling the problem to be solved locally with a higher accuracy. Additionally, this gives rise to coarse-grained parallelization, as such meshes can be re-distanced in parallel. An efficient parallelization approach, however, has to deal with interface-sharing meshes, load-balancing issues, and must offer reasonable parallel efficiency for narrow band and full band re-distancing. We present a parallel multi-mesh FMM to tackle these challenges: Interface-sharing meshes are resolved using a synchronized data exchanges strategy. Parallelization is introduced by applying a pool of tasks concept, implemented using OpenMP tasks. Meshes are processed by OpenMP tasks as soon as threads become available, efficiently balancing out the computational load of unequally sized meshes over the entire computation. Our investigations cover parallel performance of full and narrow band re-distancing. The resulting algorithm shows a good parallel efficiency, if the problem consists of significantly more meshes than the available processor cores.

Keywords Fast marching method · Shared memory parallelism · Eikonal equation · Re-distancing

Mathematics Subject Classification (2010) 68W10 · 65Y05 · 65Y10 · 65Y20

Communicated by: Pavel Solin

✉ Georgios Diamantopoulos
diamantopoulos@iue.tuwien.ac.at

Extended author information available on the last page of the article.

1 Introduction

In order to avoid computationally costly methods of analytically computing the minimum Euclidean distance of every mesh point to an interface, alternative, accelerated methods are sought to reduce the computational effort and thus the computing time. To that end, the problem of re-calculating the distance to an interface in order to restore the signed distance field property in a domain can be treated as the simulation of an expanding front with unit speed. This problem is arising in several applications of science and engineering, for example, in computational geometry, computer vision, micro- and nanoelectronics, and computational fluid dynamics [10].

The expanding front problem can be mathematically formulated 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 the initial interface (boundary), $\phi(\mathbf{x})$ the unknown function, and $F(\mathbf{x})$ a positive function which describes the speed with which the interface information propagates in the domain. In case that $F = 1$, the solution $\phi(\mathbf{x})$ represents the minimum Euclidean distance between the point $\mathbf{x}$ and the interface Γ. This case is generally referred to as re-distancing and is the main focus of this work.

A popular and widely used method for the solution of the eikonal equation is the fast marching method (FMM) [10]. The FMM is a non-iterative method based on upwind differences, offering $O(h)$ accuracy. A priority queue is used in order to specify the order in which the nodes are visited, similar to Dijkstra's algorithm for the shortest path between nodes in a graph [5]. Due to the fact that the method is non-iterative and relies on a single priority queue, parallelization is not straightforward, as, for instance, parallel write access to the queue would have to be guarded from race conditions basically nullifying any parallelization attempt.

Because of the non-straightforward parallelization of the FMM, alternative parallel methods have been developed. The most predominant of these methods is the fast sweeping method (FSM) [14, 15] and the fast iterative method (FIM) [7]. Both methods support parallel processing. In particular, the FIM uses a block-based domain decomposition and is not relying on a priority queue; hence, it supports single instruction multiple data (SIMD), which is attractive for fine-grained parallelization platforms, such as accelerators. The drawback of both methods, however, is that their accuracy is lower than the FMMs due to their iterative nature. Another set of methods is the family of two-scale methods [2]. The Heap-Cell method, for which the parallel version is presented in [3], has a two-scale technique which combines the FMM with FSM on different scales/resolutions. However, these methods are out of scope for this work, as here we focus on applying the FMM on several meshes on the same scale (i.e., offering the same resolution) without interpolating/restricting the solution to the finer/coarser level.

Several attempts to parallelize the FMM have been made. The parallel algorithms provided in [1, 6] do not provide a scalable and generally applicable algorithm. However, the authors introduce the concept of domain decomposition into the FMM. Recently, a promising parallel version of the FMM based on overlapping domain decomposition has been introduced [13]. This approach provides a scalable algorithm for distributed memory applications. In our previous contributions, a shared memory variant of the algorithm has been developed [11, 12] and an extended evaluation of the developed method based on various problem cases has been presented [4].

This work is related to a mesh refinement workflow where the eikonal equation is solved on a Cartesian hierarchical mesh data structure, such as in feature-scale topography simulations based on solving the level set equation for etching or deposition processes in microelectronics [8]. The Cartesian mesh hierarchy allows to use an efficient finite difference scheme for solving the level set problem while also being able to cover geometrically challenging problems due to local mesh refinement: The mesh hierarchy is composed of a base mesh covering the entire simulation domain as well as additional *refinement levels*: Each refinement level contains a set of locally defined meshes (with the same higher resolution compared to the base mesh) around certain areas with increased accuracy demands (e.g., corners). The meshes do not overlap but neighboring meshes can potentially share an interface which is creating dependencies during re-distancing.

We present a parallel multi-mesh FMM which is capable of re-distancing an arbitrary number of meshes with the same resolution in parallel (referring to the locally defined meshes of a refinement level, as previously discussed). A synchronized data exchange is introduced in order to resolve dependencies among neighboring meshes. Despite the fact that there are re-distancing methods available, where different resolutions can be processed by the same algorithm in parallel [2, 3], there is no available re-distancing method which can process several neighboring meshes of the same resolution simultaneously. However, as previously indicated, such an algorithm is particularly important to adaptive mesh refinement approaches, where several meshes with a certain resolution are defined on specific regions of interest, giving rise to a coarse-grained parallelization approach. We focus on introducing the re-distancing algorithm, as such, we do not cover mesh generation aspects, which are out of scope of this work. Moreover, the aspect of correcting the solution in the coarser meshes (after computing the solution in the finer ones) is not covered in this work.

The proposed algorithm is implemented in C++ using *OpenMP Tasks* as the core mechanism for parallelization. We evaluate the method with test cases based on problems arising from the field of micro- and nanoelectronics fabrication simulations and for various mesh configurations. The results include full-band re-distancing, where the solution is calculated in the whole domain, as well as narrow-band re-distancing scenarios, where the solution is calculated only for a certain number of cells away from the interface.

The paper is organized as follows: In Section 2, the proposed parallel algorithm is presented based on a short overview of the original, serial algorithm to set the stage. The parallel performance of the implementation is analyzed in Section 3.

2 A multi-mesh fast marching method

In this section, the serial FMM and the proposed algorithm for multiple meshes are explained.

2.1 The serial fast marching method

First, the originally presented FMM [10] (an extended description is available [13]) is summarized. The approximation of the gradient term of the eikonal equation (see Eq. 1) is based on a first-order Godunov-type finite difference scheme [9]. The discretized version is given by:

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

$D^{-}_{i,j,k}$ and $D^{+}_{i,j,k}$ are the first order backward and forward difference operators, respectively, calculated in the cell centers. We consider a cell to be an element of the structured mesh, described by the spatial indices (i, j, k), which refer to the mesh coordinates of the cell center. The method has $O(h)$ accuracy [10].

The discretized equation (Eq. 2) shows an upwind structure which is the key element of the FMM algorithm, because the solution at a certain mesh point $\phi_{i,j,k}$ depends only on the neighboring cells with a smaller value. In the FMM algorithm, the upwind structure is preserved, because the algorithm starts from the interface Γ and progresses outwards, visiting the cell with the smallest value in each step. The cell with the next minimum value is provided by a priority queue. In order to specify the upwind order with which the cells are processed, three status flags are used. Initially all cells which are not on the interface Γ are marked with the flag `FAR` and their value is set to infinity. Interface cells are marked as `KNOWN` and are initialized to a known value. In the initialization of the FMM, the cells neighboring to the interface cells are marked as `BAND`, their solution is calculated by solving Eq. 2, and they are added to the priority queue. The algorithm's inherent loop proceeds as follows: In each loop iteration, the cell with the minimum value is removed from the heap, its value is fixed, and it is marked as `KNOWN`. The neighbors of this cell, which at this point are not marked as `KNOWN`, are evaluated by solving Eq. 2, they are marked as `BAND`, and they are added to the priority queue. In case that a cell has already been marked as `BAND` its value and position in the queue are updated. The loop continues until all cells are marked as `KNOWN`, namely until the priority queue is empty. In case that the solution is to be calculated solely within a narrow band around the interface Γ, the loop terminates when the absolute value of the minimum element in the priority queue is larger than the narrow band width.

2.2 The multi-mesh fast marching method

The original FMM, as presented in Section 2.1, relies on a global priority queue; hence, a parallelization of the algorithm is not straightforward. However, a very promising approach for a parallel FMM has been presented for distributed memory systems in [13] and for shared memory systems in [12]. This approach serves as the basis for the multi-mesh FMM described in this section. The concept behind parallelizing the FMM on a single mesh (which is based on domain decomposition) is here re-applied, only that no domain decomposition step is required as the neighboring meshes represent the decomposed domains.

The difference between the problem which is solved by the method presented in [12] and [13] and the problem which is solved by the method presented here is illustrated in Fig. 1. In Fig. 1a, there is one mesh defined in the whole domain; in order to parallelize this problem, a domain decomposition is introduced and the solution

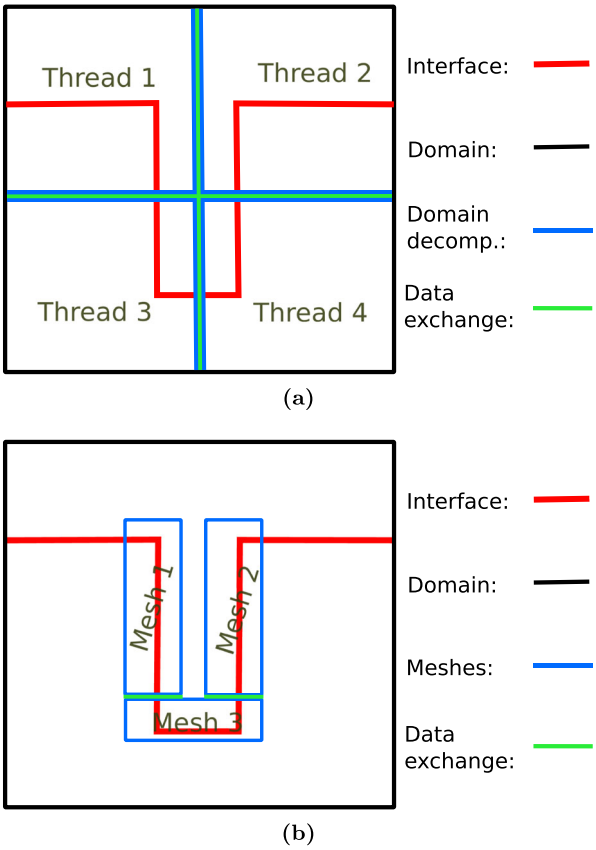

Fig. 1 Exemplary single trench problem with **a** one mesh and domain decomposition into four partitions, each processed by on thread, and **b** three meshes around the trench each undertaken one `OpenMP` task

in each partition is computed by a separate parallel unit, e.g., a thread. Figure 1b shows an exemplary adaptive mesh refinement scenario: Three finer meshes, offering the same resolution, are defined around the interface on specific regions of interest. The solution in each mesh is computed by, in this case, a separate task. What is common in both cases is that the dependences among the partitions in the first case and the interface-sharing dependences among neighboring meshes in the second case are resolved using the synchronized data exchange approach introduced in [13].

Before starting the multi-mesh FMM algorithm, two steps are necessary. The first is to introduce a ghost layer for each mesh. In this way, overlapping areas between meshes are potentially introduced, if the meshes are close to each other. Later on, during the execution of the algorithm, the data in the overlapping areas is exchanged in order to resolve dependencies among meshes. The second step is the detection and initialization of the interface cells. This step is required in order to provide the initial condition of Eq. 1 on the interface Γ.

In the trivial case where meshes are independent of each other (i.e., they do not share an interface), the FMM can be applied independently with a local priority queue in each mesh. However, in the case of neighboring meshes, there are dependencies among neighbors, as the solutions of the neighboring meshes influence each other, requiring an inter-mesh communication. In order to resolve dependencies and at the same time achieve parallel processing of the meshes, the proposed solution is the following (cf. Algorithm 1): The FMM algorithm is independently applied to each mesh, then the solutions within overlapping halo regions towards the neighboring meshes are communicated in a synchronized data exchange step. The communicated solutions are evaluated and those that preserve the upwind principle of the FMM are integrated into the solution. In cases where neighboring solutions are accepted for integration into the local solution of a particular mesh, this mesh has to repeat the FMM algorithm to fully integrate the neighbor solution into the local one. The loop continues until no new solutions are integrated after data exchange.

The multi-mesh FMM algorithm, as described in Algorithm 1, works as follows: In the beginning, all meshes are initialized, namely the neighbors of the interface cells are identified, evaluated and added to the heap (lines 1–5). This process can be independently performed in each mesh; hence, it can be executed by independent tasks. However, a task-wait (i.e., synchronization) clause is necessary before starting the fast marching loop (line 6) in order to make sure that all meshes are properly initialized. The next step is the fast marching loop (lines 6–26). Before discussing the loop details, the concept of *active meshes* has to be introduced. A mesh is considered *active*, when its priority queue is not empty and the minimum element is within the narrow band. For every active mesh, a separate task is spawned. The task consists of three functions: In the first function, `march_narrow_band`, the FMM loop as explained in Section 2.1 is performed. In the second function, `collect_overlapping_data`, the updated values in the regions which overlap with neighboring meshes are collected (i.e., detected and added to an output buffer) by each mesh independently. Finally, in the third function, `exchange_overlapping_data`, the collected solutions are communicated to the corresponding neighboring meshes. The communicated solutions are not integrated yet into the data of the neighboring meshes, but they are placed in a dedicated,

mesh-exclusive buffer (which thus does not require guarding from race conditions) and they are evaluated at a later stage of the algorithm. In order to ensure that all communication is completed before moving to the next step, a task-wait clause is used again for synchronization (line 17). The next step is the conditional integration of the overlapping data (lines 21–25). This is performed by new tasks (where a separate task is spawned for every mesh): In the function `integrate_overlapping_data`, all received solutions are evaluated and the cell value is updated, if the received (by the neighbor) value is smaller than the current value. It is important that all meshes (and not only the active ones) perform this step, because even inactive meshes might receive relevant update data from their active neighbors. All cells with updated values are placed in the priority queue of their meshes and the loop is restarted. The loop terminates in case that all meshes are inactive.

Algorithm 1 Multi-Mesh FMM.

```
 1: for every mesh do                                   ▷ Parallel region
 2:     Create Task:
 3:         initialize_FMM
 4:     End Task
 5: end for                                             ▷ Task-wait
 6: while 1 do
 7:     ActiveMeshes ← 0                                ▷ Shared variable
 8:     for every mesh do                               ▷ Parallel region
 9:         if Mesh is Active then
10:             ActiveMeshes ← ActiveMeshes + 1
11:             Create Task:
12:                 march_narrow_band
13:                 collect_overlapping_data
14:                 exchange_overlapping_data
15:             End Task
16:         end if
17:     end for                                         ▷ Task-wait
18:     if ActiveMeshes = 0 then                        ▷ Algorithm finished
19:         Break Loop
20:     end if
21:     for every mesh do                               ▷ Parallel region
22:         Create Task:
23:             integrate_overlapping_data
24:         End Task
25:     end for                                         ▷ Task-wait
26: end while
```

The entire algorithm uses an implicit *pool of tasks* to enable load balancing, as meshes potentially have different sizes and thus represent different computational loads. For each mesh, a separate task is created and is executed as soon as a thread becomes available. In this way, threads do not stay idle. Hence, the computational

load of unequally sized meshes can be balanced out (to some degree), creating an efficient strategy for the solution of problems with varying mesh sizes.

3 Parallel performance results and analyses

This section evaluates the performance of the introduced parallel multi-mesh FMM. It is important to note that regarding the knowledge of the authors there is no other algorithm available allowing for re-distancing of several meshes (which potentially share an interface) with the same resolution in parallel. Therefore, a comparison is not possible. However, a detailed analysis of the algorithm's performance is provided, allowing to judge the algorithm's principal capabilities.

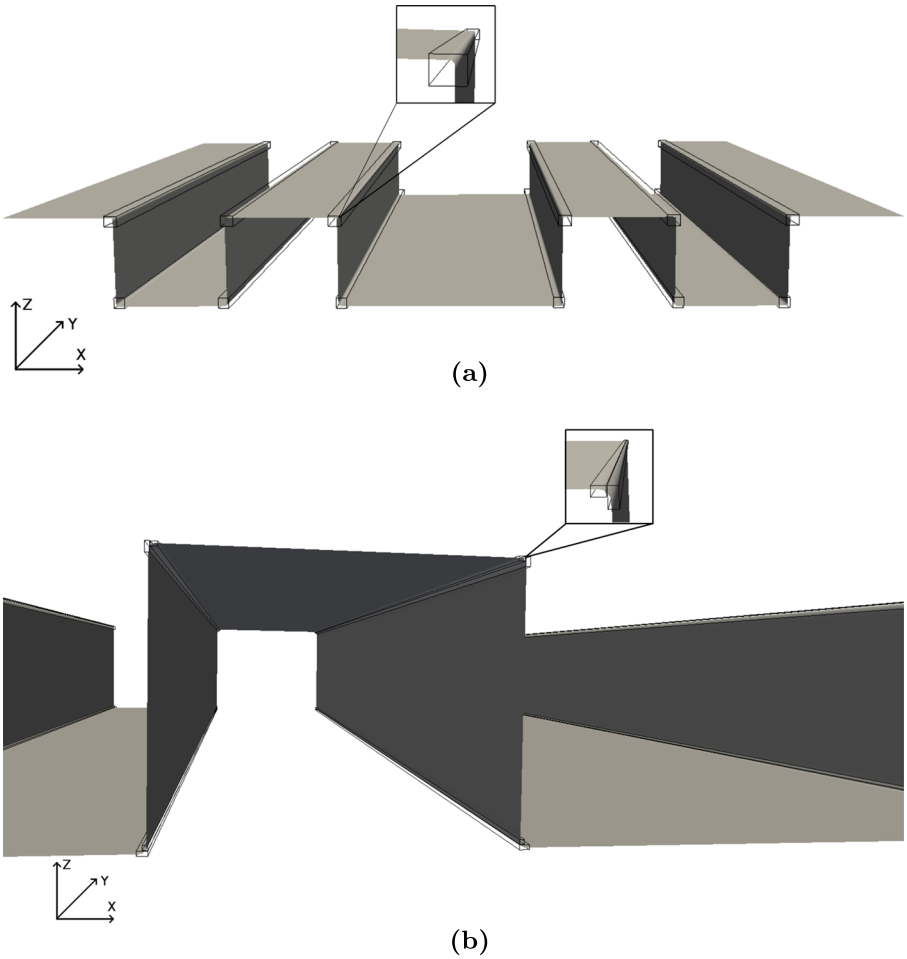

Fig. 2 *Three-Trenches* test case. **a** 12 meshes—one in each corner. **b** 24 meshes—two in each corner

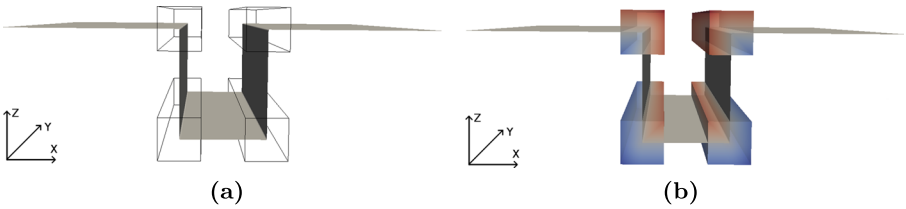

Fig. 3 *Single-Trench* test case with four meshes: **a** outlines, **b** solutions; red and blue areas with increasing intensity denote two sides of increasing minimum distance towards the interface

3.1 Test cases

For the evaluation of the parallel performance of the parallel multi-mesh FMM, test cases inspired by problems from microelectronics engineering are chosen, but of course the algorithm can be applied to arbitrary re-distancing problems. The first test case in Fig. 2 is an interface with three trenches, offering 12 meshes (one in each corner) with a grid resolution of 0.00125 in each direction (see Fig. 2a) and 24 meshes (two in each corner) with a resolution of 0.0003125 (see Fig. 2b) in the domain $\Omega = [-1, 1] \times [-1, 1] \times [-1, 0.5]$ (referred to as *Three-Trenches*). For the 12 meshes case, the meshes residing on the upper corners of the interface are by 16, 7% bigger than those on the lower corner. For the 24 meshes case, the mesh pair is differently sized, as can be seen in Fig. 2b: on the upper corners, they differ by 33.3% and on the lower corners by 50%. The second test case, shown in Fig. 3, is an interface with one trench and four meshes (one in each corner) with resolution 0.00125 in each direction in the domain $\Omega = [-1, 1] \times [-1, 1] \times [-1, 0.5]$ (referred to as *Single-Trench*). Here, all meshes have exactly the same size. The third test case is a complex interface visualized in Fig. 4, (referred to as *Quad-Holes*). In this test case, multiple meshes are defined and they are clustered around the four structures of interest on the interface. In Fig. 4a, 48 meshes of various sizes are defined with resolution 0.005 and in Fig. 4b, 303 meshes with resolution 0.00125. Moreover, this test case is additionally evaluated for 489 meshes with resolution 0.00125 and 2389 meshes with resolution 0.0003125 in each direction, in the domain $\Omega = [-1, 1] \times [-1, 1] \times [-1, 0.5]$. In all test cases, the performance is benchmarked for full band re-distancing (i.e., computation for the whole domain) as well as for narrow-band re-distancing (i.e., computation for a certain number of cells away from the interface). These test cases offer a combination of different problem sizes, enabling an evaluation of the performance for load-balanced and load-imbalanced scenarios.

3.2 Benchmarking platform

Benchmarking results have been produced on a single node of the Vienna Scientific Cluster 3 (VSC-3).[1] The node has two sockets, each equipped with an 8-core Intel Xeon E5-2650v2 Ivy Bridge-EP processor, running at 2.6 GHz with 20 MB of L3

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

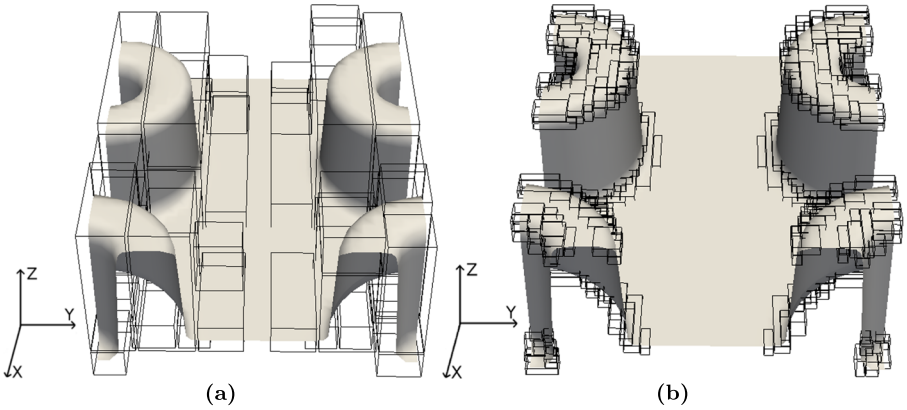

Fig. 4 *Quad-Holes* test case: **a** 48 meshes, **b** 303 meshes

cache. The node offers 16 physical and 32 logical cores which are accompanied by 64 GB of DDR3 memory. The benchmark implementation of the multi-mesh FMM is based on GNU/Linux and C++ (Intel C++ compiler version 18.0).

3.3 Results and analysis

3.3.1 Full-band re-distancing

The runtime and parallel speedup for full-band re-distancing for the *Three-Trenches* case (cf. Fig. 2) is presented in Fig. 5. For up to six threads, the parallel efficiency in both cases is around 50%. Beyond that, saturation effects start to severely limit the efficiency, more so for the case with 12 meshes. This is in particular visible for more than 12 threads: Increasing thread under-utilization (i.e., more threads available than meshes to be processed) leads to degradation of the speedup.

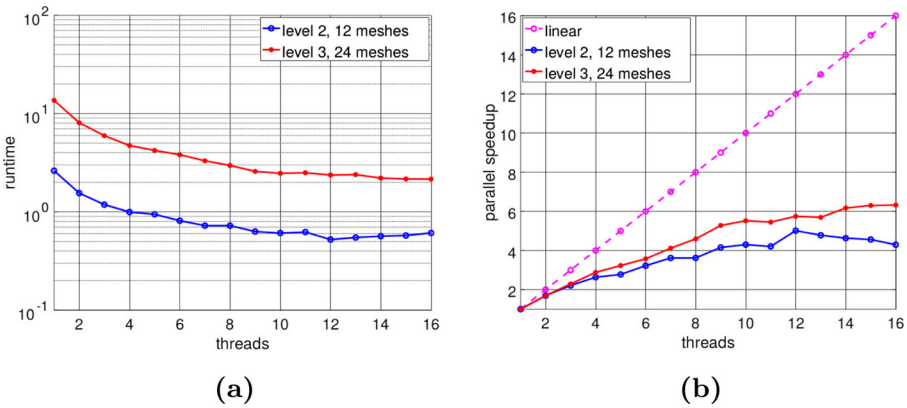

Fig. 5 Runtime (**a**) and parallel speedup (**b**) for the *Three-Trenches* test case with 12 and 24 meshes (cf. Fig. 2)

In order to explain the suboptimal parallel performance of the *Three-Trenches* test case, a simple test case offering four equally sized meshes, i.e., *Single-Trench* (cf. Fig. 3), is investigated using four threads only, thereby providing a simplified picture which is more amenable for analysis. The performance results are illustrated in Fig. 6. In this case, the expected performance should offer a nearly linear speedup, because the workload is equally distributed among threads. However, as can be seen in Fig. 6a and b, the performance scaling is sub-linear, already for two threads. This can be explained by the average runtime per mesh, i.e., average time per task (excluding synchronization), which is illustrated with the red (bottom) line in Fig. 6a. Here the average runtime is increasing as the number of threads is increasing. This shows the impact of shared computing resources, in particular the memory subsystem, on the overall scalability.

The performance for the *Quad-Holes* case is visualized in Fig. 7 for 48 and 303 meshes, and in Fig. 8 for 489 and 2389 meshes. As can be seen for all cases, the performance for up to 8 threads is quite good (efficiency around 88%). For the 48 meshes case (Fig. 7), the speedup drops beyond 8 threads. The reason for this performance drop is that there are 8 meshes for which the FMM requires more than 10 milliseconds (for each mesh). For all other meshes, the FMM runtime is around 1 millisecond or less, which causes a massive load imbalance. In this case, 8 threads will be busy with executing the FMM for the big meshes and the other threads will become idle as soon as the small meshes are processed. In all other cases, the performance is scalable (efficiency around 75% or more) even for up to 16 threads. Beyond 8 threads (where the second socket of the dual-socket compute platform is utilized), scaling continues to be very good (up to 79% efficiency for 16 threads and the 2389 meshes case), although non-unified memory access (NUMA) effects limit further speedup improvements. For all cases where the parallel efficiency is high (i.e., around 75% or more) the problem consists of many more meshes—at least 10 times more—than the compute cores/threads available (see Fig. 4b) with various sizes. Due to the fact that the number of meshes (hence the number of tasks) is much larger than the number of

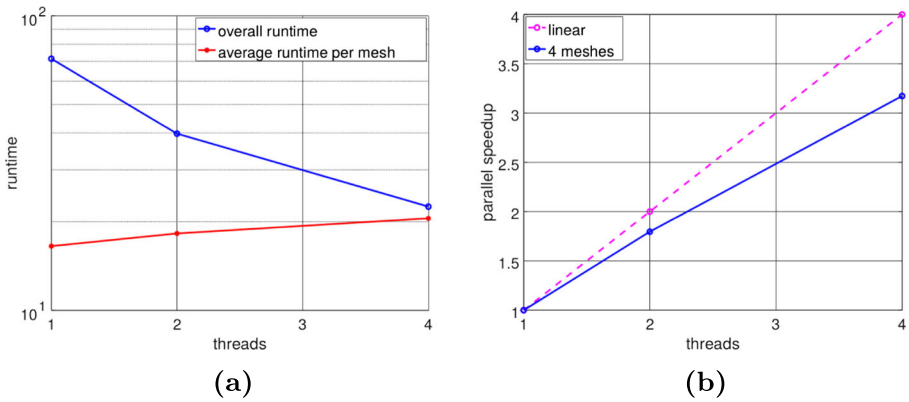

Fig. 6 Overall runtime and average runtime per mesh (**a**), parallel speedup (**b**) for the *Single-Trench* test case with 4 meshes (cf. Fig. 3)

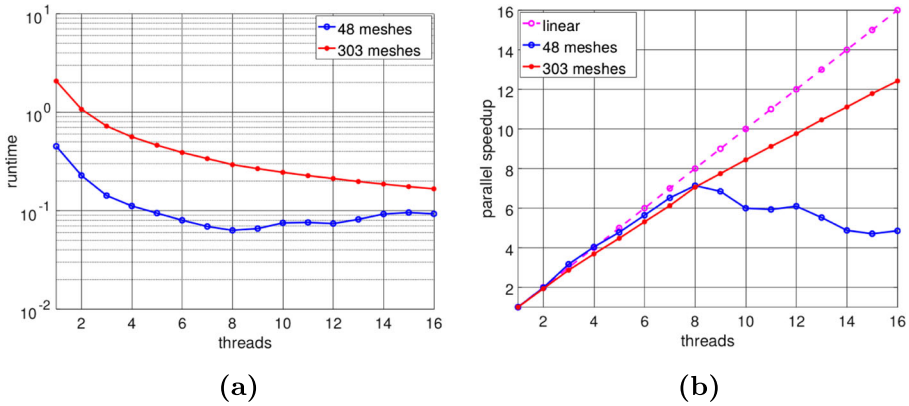

Fig. 7 Runtime (**a**) and parallel speedup (**b**) for the *Quad-Holes* test case with 48 and 303 meshes (cf. Fig. 4)

threads, the threads do not stay idle (except only at synchronization points), because a new task is assigned to them as soon as they become available.

The here applied pool of tasks concept has an inherent advantage over a single mesh decomposition approach [4]: It inherently offers load balancing. Using a pool of tasks (and the dynamic thread scheduling it provides) allows the parallel multi-mesh FMM to reach a parallel efficiency of up to 79% for 16 threads and even higher for 8 threads, regardless of the shape and location of the interface that is used as initial condition. By comparison, the domain decomposition approach used in [4], uses a static workload distribution where each thread is responsible for one part of the domain without the option of re-distributing the work in case that one or more threads become idle. The parallel performance of a static domain decomposition approach relies heavily on the shape and location of the interface and whether this can be evenly distributed among the available threads. In case of an uneven thread distribution of the interface and domain (which is to be expected in realistic application cases), the result is an unbalanced workload which ultimately limits parallel efficiency.

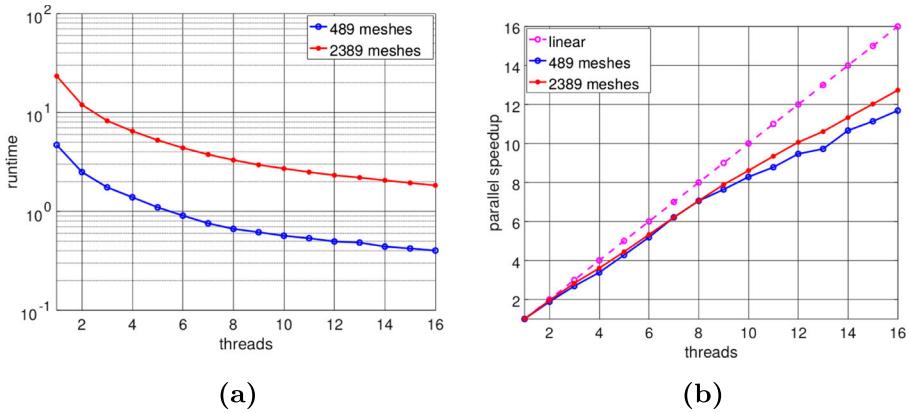

Fig. 8 Runtime (**a**) and parallel speedup (**b**) for the *Quad-Holes* test case with 489 and 2389 meshes

3.3.2 Narrow-band re-distancing

The runtime and parallel speedup for narrow-band re-distancing is shown in Fig. 9 for the *Three-Trenches* test case and in Figs. 10 and 11 for the *Quad-Holes* test cases. In these figures, the performance benchmarks for a narrow band of 5 and 10 cells away from the interface as well as for full-band re-distancing is illustrated.

In case of narrow-band re-distancing, the solution is calculated for a part of the domain only (within a specific distance towards the interface); therefore, the workload is smaller compared to the full-band re-distancing where the solution is calculated for the whole domain. For this reason, in all cases, the runtime for a narrow band of 5 cells is the lowest since the workload here is the minimum compared to the other two cases. For the 10 cells narrow band, in Figs. 10 and 11, the runtime lies between the runtime for the 5 cells narrow band and the runtime for the full band because the workload in this case is larger than for the 5 cells case but smaller than the full band. In Fig. 9, on the other hand, the runtime for the 10 cells narrow

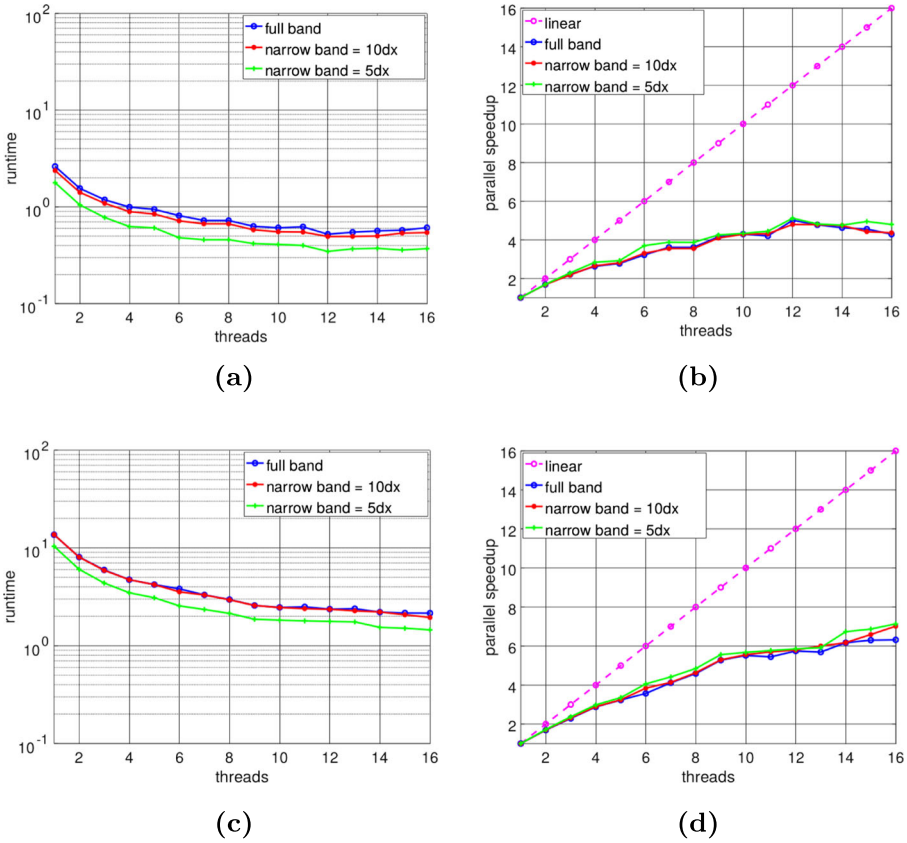

Fig. 9 Parallel performance for the *Three-Trenches* test case for full-band re-distancing (cf. Fig. 5) as well as 10 cells and 5 cells narrow band re-distancing. Runtime (**a**) and speedup (**b**) with 12 meshes, runtime (**c**) and speedup (**d**) with 24 meshes (cf. Fig. 2)

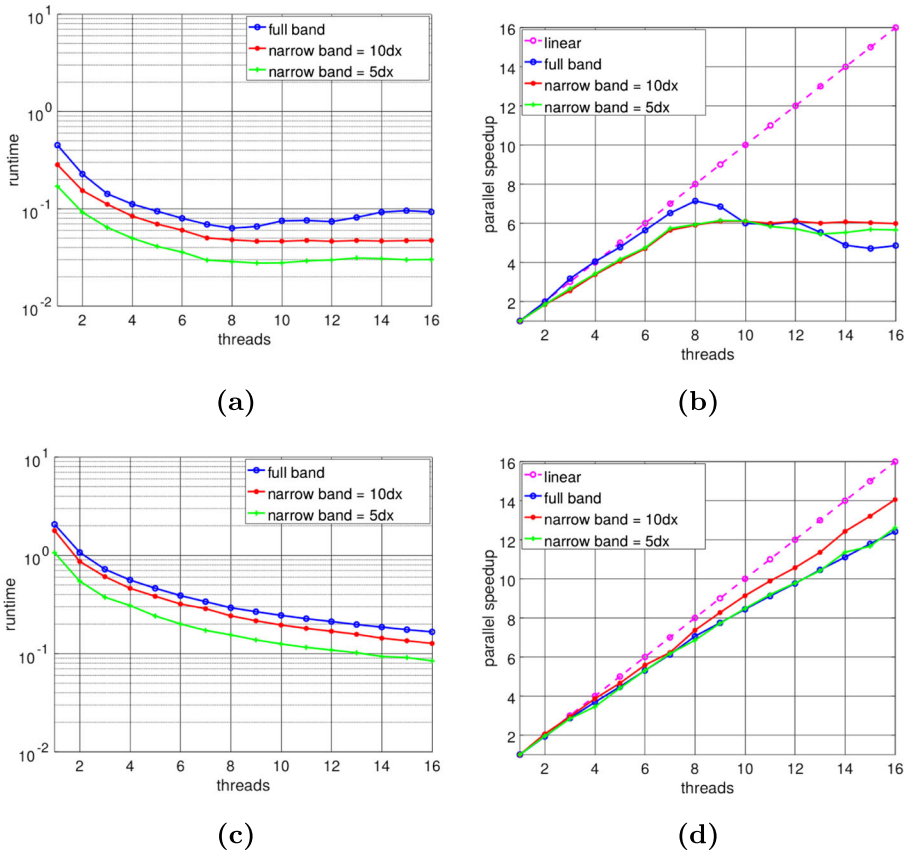

Fig. 10 Parallel performance for the *Quad-Holes* test case for full-band re-distancing (cf. Fig. 7) as well as 10 cells and 5 cells narrow band re-distancing. Runtime (**a**) and speedup (**b**) with 48 meshes, runtime (**c**) and speedup (**d**) with 303 meshes (cf. Fig. 4)

band and the full band are almost the same. The reason for that is that the size of the meshes is approximately the same as the narrow band width. For the 12 meshes case (cf. Fig. 2a), the number of cells in the x-direction is 24 for the meshes on the upper corners and 20 for the meshes on the lower corners and in the z-direction there are 20 cells for every mesh. For the 24 meshes case (cf. Fig. 2b), there are 12–24 cells in the x-direction and 12–24 cells in z-direction. Therefore, in case that a 10 cells narrow band is used (10 cells away from the interface in each direction, hence 20 cells in total), the solution is computed almost in the whole domain. For this reason, the runtime for full band re-distancing and a 10 cells narrow-band re-distancing is almost the same.

In terms of parallel efficiency, the results for the narrow-band re-distancing are, in most cases, not significantly different to that of the full-band computation. Despite the fact that the runtimes are lower, the serial version has a smaller runtime too; hence, the parallel speedup is not higher for the narrow-band re-distancing. A case

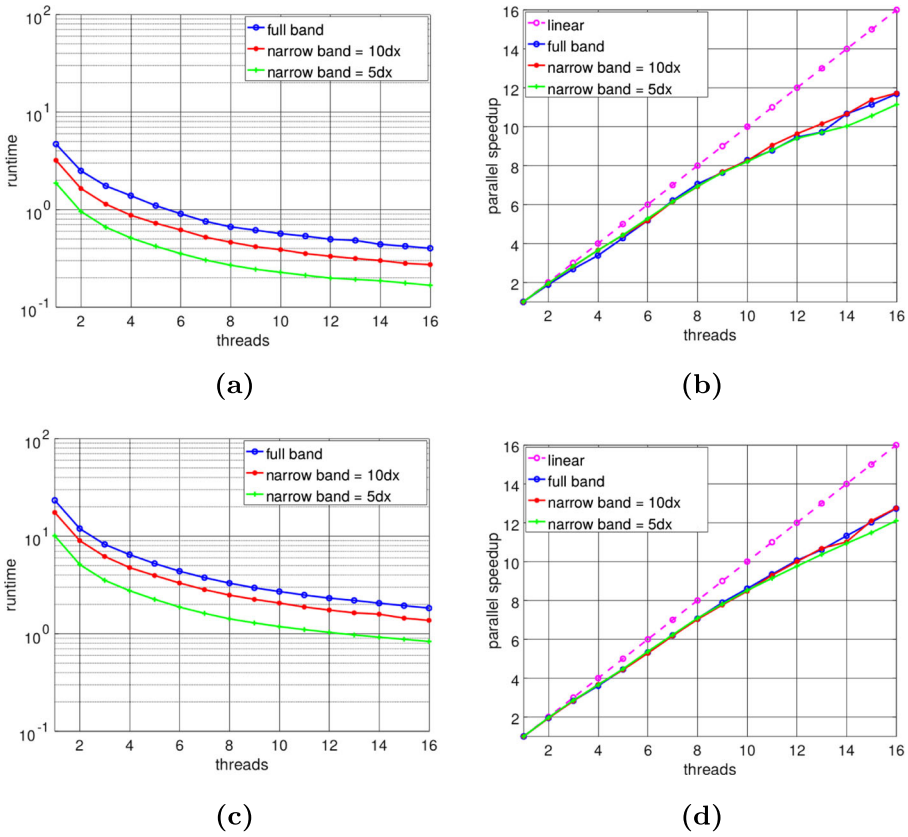

Fig. 11 Parallel performance for the *Quad-Holes* test case for full-band re-distancing (cf. Fig. 8) as well as 10 cells and 5 cells narrow band re-distancing. Runtime (**a**) and speedup (**b**) with 489 meshes, runtime (**c**) and speedup (**d**) with 2389 meshes

where the situation is different is the *Quad-Holes* test case with 48 meshes, which is illustrated in Fig. 10a and b, the parallel efficiency for narrow-band re-distancing is lower compared to the full-band computation for up to 8 threads but is better—although constant—for larger thread numbers: As mentioned above, in this case, there are 8 meshes for which the full-band FMM has a runtime approximately ten times higher than the other meshes. In case of narrow-band re-distancing though, the runtime for those meshes is smaller (due to the fact that the solution is calculated only for a part of the domain). Therefore, the load-imbalance observed in the full-band computation is moderate for the narrow-band computation.

4 Summary

A parallel multi-mesh FMM algorithm which uses tasking in a shared memory environment for parallelization has been introduced. The proposed method shows a

highly scalable performance for cases where there are many more (at least 10 times more) meshes available than compute cores/threads, favoring our approach's inherent load-balancing capabilities via the applied pool of tasks concept. On the other hand, the performance for a smaller number of meshes with larger size is limited and becomes suboptimal for cases where the workload is not evenly distributed among the meshes.

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

References

1. Breuß, M., Cristiani, E., Gwosdek, P., Vogel, O.: An adaptive domain-decomposition technique for parallelization of the fast marching method. Appl. Math. Comput. **218**(1), 32–44 (2011). https://doi.org/10.1016/j.amc.2011.05.041
2. Chacon, A., Vladimirsky, A.: Fast two-scale methods for Eikonal equations. SIAM J. Sci. Comput. **34**(2), A547–A578 (2012). https://doi.org/10.1137/10080909X
3. Chacon, A., Vladimirsky, A.: A parallel two-scale method for Eikonal equations. SIAM J. Sci. Comput. **37**(1), A156–A180 (2015). https://doi.org/10.1137/12088197X
4. Diamantopoulos, G., Weinbub, J., Hössinger, A., Selberherr, S.: Evaluation of the shared-memory parallel fast marching method for re-distancing problems. In: Proceedings of the 17th International Conference on Computational Science and Its Applications (ICCSA), pp. 1–8. https://doi.org/10.1109/ICCSA.2017.7999648 (2017)
5. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959). https://doi.org/10.1007/BF01386390
6. Herrmann, M.: A domain decomposition parallelization of the fast marching method. In: Anual Research Briefs, pp. 213–225. Center for Turbulence Research, Stanford University (2003)
7. Jeong, W.K., Whitaker, R.T.: A fast iterative method for Eikonal equations. SIAM J. Sci. Comput. **30**(5), 2512–2534 (2008). https://doi.org/10.1137/060670298
8. Manstetten, P.: Efficient Flux Calculations for Topography Simulation. Ph.D. thesis, TU Wien (2018). http://www.iue.tuwien.ac.at/phd/manstetten/
9. Rouy, E., Tourin, A.: A viscosity solutions approach to shape-from-shading. SIAM J. Numer. Anal. **29**(3), 867–884 (1992). https://doi.org/10.1137/0729053
10. Sethian, J.A.: Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science, 2nd edn. Cambridge University Press (1999)
11. Weinbub, J., Hössinger, A.: Comparison of the parallel fast marching method, the fast iterative method, and the parallel semi-ordered fast iterative method. Procedia Comput. Sci. **80**, 2271–2275 (2016). https://doi.org/10.1016/j.procs.2016.05.408
12. Weinbub, J., Hössinger, A.: Shared-memory parallelization of the fast marching method using an overlapping domain-decomposition approach. In: Proceedings of the 24th High Performance Computing Symposium, pp. 18:1–18:8. https://doi.org/10.22360/SpringSim.2016.HPC.052 (2016)
13. Yang, J., Stern, F.: A highly scalable massively parallel fast marching method for the Eikonal equation. J. Comput. Phys. **332**, 333–362 (2017). https://doi.org/10.1016/j.jcp.2016.12.012
14. Zhao, H.: A fast sweeping method for Eikonal equations. Math. Comput. **74**(250), 603–627 (2005). https://doi.org/10.1090/S0025-5718-04-01678-3
15. Zhao, H.: Parallel implementations of the fast sweeping method. J. Comput. Math. **25**(4), 421–429 (2007). https://www.jstor.org/stable/43693378

Affiliations

Georgios Diamantopoulos[1] · Andreas Hössinger[2] · Siegfried Selberherr[3] · Josef Weinbub[1]

Andreas Hössinger
andreas.hoessinger@silvaco.com

Siegfried Selberherr
selberherr@iue.tuwien.ac.at

Josef Weinbub
weinbub@iue.tuwien.ac.at

1 Christian Doppler Laboratory for High Performance TCAD, Institute for Microelectronics, TU Wien, Vienna, Austria

2 Silvaco Europe Ltd., St Ives, Cambridgeshire, UK

3 Institute for Microelectronics, TU Wien, Vienna, Austria