



# Shared-memory block-based fast marching method for hierarchical meshes

Michael Quell<sup>a,\*</sup>, Georgios Diamantopoulos<sup>a</sup>, Andreas Hössinger<sup>b</sup>, Josef Weinbub<sup>a</sup>

<sup>a</sup> Christian Doppler Laboratory for High Performance TCAD, Institute for Microelectronics, TU Wien, Gußhausstraße 27-29, 1040 Wien, Austria

<sup>b</sup> Silvaco Europe Ltd., Compass Point, St Ives, Cambridge PE27 5JL, United Kingdom

## ARTICLE INFO

### Article history:

Received 16 September 2020

Received in revised form 29 January 2021

### Keywords:

Fast marching method

Eikonal equation

Level-set method

Re-distancing

Hierarchical meshes

Shared-memory parallelization

## ABSTRACT

The fast marching method is commonly used in expanding front simulations in various fields, such as, fluid dynamics, computer graphics, and in microelectronics, to restore the signed-distance field property of the level-set function, also known as re-distancing. To improve the performance of the re-distancing step, parallel algorithms for the fast marching method as well as support for hierarchical meshes have been developed; the latter to locally support higher resolutions of the simulation domain whilst limiting the impact on the overall computational demand. In this work, the previously developed multi-mesh fast marching method is extended by a so-called block-based decomposition step to improve serial and parallel performance on hierarchical meshes. OpenMP tasks are used for the underlying coarse-grained parallelization on a per mesh basis. The developed approach offers improved load balancing as the algorithm employs a high mesh partitioning degree, enabling to balance mesh partitions with varying mesh sizes. Various benchmarks and parameter studies are performed on representative geometries with varying complexities. The serial performance is increased by up to 21% whereas parallel speedups ranging from 7.4 to 19.1 for various test cases on a 24-core Intel Skylake computing platform have been achieved, effectively doubling the parallel efficiency of the previous approach.

© 2021 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

The fast marching method (FMM) was developed to solve the Eikonal equation [15]

$$\|\nabla\phi(\vec{x})\| = \frac{1}{f(\vec{x})} \quad \vec{x} \in \Omega, \quad (1)$$

$$\phi(\vec{x}) = 0 \quad \vec{x} \in \Gamma \subset \Omega, \quad (2)$$

in the context of expanding front simulations: The Eikonal equation describes an expanding front emerging from an interface  $\Gamma$  and propagating with the strictly positive local speed  $f(\vec{x}) > 0$ . Applications of the Eikonal equation include seismic processing [16,19], path-finding [8], three-dimensional imaging [14], and the level-set method [6,12,15]; the latter

\* Corresponding author.

E-mail addresses: [michael.quell@tuwien.ac.at](mailto:michael.quell@tuwien.ac.at) (M. Quell), [georgios.diamantopoulos@tuwien.ac.at](mailto:georgios.diamantopoulos@tuwien.ac.at) (G. Diamantopoulos), [andreas.hoessinger@silvaco.com](mailto:andreas.hoessinger@silvaco.com) (A. Hössinger), [josef.weinbub@tuwien.ac.at](mailto:josef.weinbub@tuwien.ac.at) (J. Weinbub).

is a general framework to track moving interfaces [13]. In process technology computer-aided design (TCAD), which simulates various manufacturing processes of microelectronic devices and is the main context of this work, such as etching and deposition, the level-set method is used to track the position of the wafer surface (interface) using the level-set function  $\phi$ , which implicitly represents the wafer surface. In the level-set method the Eikonal equation using a speed function  $f = 1$  is solved to re-normalize the level-set function. This procedure is called re-distancing as the signed-distance to the interface  $\Gamma$  is calculated.

Due to the wide spread area of applications of the Eikonal equation several algorithms have been developed to efficiently solve the Eikonal equation [1,2,7,10,11,17]. In a recent review of serial Eikonal solvers [8], the authors showed that the FMM is sometimes outperformed in single-mesh<sup>1</sup> scenarios. The FMM has advantages regarding data exchanges between multiple meshes, i.e., when the computational problems consist of various meshes which require synchronization at their borders: The strict data ordering enforced by the FMM allows for efficient integration of exchanged data. Domains consisting of multiple meshes (in short: multi-meshes) natively arise in the context of hierarchical meshes. Hierarchical meshes consist of several levels of meshes with increasingly finer spatial resolutions. On all levels a global indexing scheme allows to uniquely identify all mesh points, especially the start index (i.e., mesh point with the lowest indices along all coordinate axis) of each mesh. A mesh is uniquely identified by its start index and mesh size (i.e., number of mesh points along each coordinate axis). This is essential to determine neighbor relationships between meshes on the same level. A single mesh covers the full computational domain on the coarsest spatial resolution (Level 0). On the next finer level (Level 1) only parts of the computational domain, which require a finer spatial resolution, are covered by non-overlapping meshes. Additional levels, further increasing the localized spatial resolution, may be employed if required by accuracy requirements of the simulation problem at hand. Specific *nesting* criteria ensure that the boundaries of a mesh on a level are either determined by a neighboring mesh or by a mesh on the next coarser level through interpolation.

In this work, the previously developed multi-mesh fast marching method [3] is extended by a so-called block-based decomposition step to improve serial and parallel performance on hierarchical meshes. OpenMP tasks are used for the underlying coarse-grained parallelization on a per mesh basis. The developed approach offers improved load balancing as the algorithm employs a high mesh partitioning degree, enabling to balance mesh partitions with varying mesh sizes. In Section 2, the basics of the FMM are discussed to set the stage. In Section 3, the newly developed block-based decomposition step is presented and evaluated in Section 4 based on three representative test cases. In particular, the influence of the performance-critical parameters block size and stride width is extensively studied on a per level basis. The last section, Section 5, summarizes the findings and provides an outlook for further research.

## 2. The fast marching method

In this work, we consider the Eikonal equation discretized on a Cartesian grid using finite differences and a first order upwind scheme, i.e., for three dimensions

$$\left[ (\max(D_{ijk}^{-x}, -D_{ijk}^{+x}, 0))^2 + (\max(D_{ijk}^{-y}, -D_{ijk}^{+y}, 0))^2 + (\max(D_{ijk}^{-z}, -D_{ijk}^{+z}, 0))^2 \right]^{\frac{1}{2}} = \frac{1}{f_{ijk}}. \tag{3}$$

The  $D_{ijk}^{\pm x,y,z}$  are the first order forward and backward finite difference operators, i.e., for the x-direction

$$D_{ijk}^{-x} = \frac{\phi_{ijk} - \phi_{i-1jk}}{\Delta x} \quad \text{and} \quad D_{ijk}^{+x} = \frac{\phi_{i+1jk} - \phi_{ijk}}{\Delta x}, \tag{4}$$

with  $\Delta x$  referring to the grid spacing in x-direction. Using the maximum of the forward finite difference operator, backward finite difference operator, and zero enforces the selection of the neighboring grid point with the lowest  $\phi$ -value. If the maximum is zero none of the neighboring grid points has a lower  $\phi$ -value.

The FMM processes the mesh points in ascending order (sorted by their  $\phi$  value), starting from mesh points next to  $\Gamma$ . This process is similar to Dijkstra's algorithm [5]; each mesh point is labeled by one of three states: Far, Band, and Known.

Initially, all mesh points are labeled Far and their  $\phi$  value is set to infinity, then the mesh points next to  $\Gamma$  are labeled Known, their  $\phi$  value is initialized, adjacent Far mesh points are changed to Band, and their  $\phi$  value is set by solving (3). The mesh point with the smallest value labeled Band is *accepted* (i.e. finished) and labeled Known, adjacent mesh points labeled Band or Far are updated ( $\phi$  value is recomputed) and labeled Band. The ordering of mesh points is typically achieved using a priority queue (implemented using a heap data structure), sorting the mesh points labeled Band by their  $\phi$  value. This step is repeated until all mesh points are labeled Known or the smallest point containing the label Band is above a threshold value, if the region of interest is only a narrow band around the interface. The ordered processing (sorting the mesh points by their  $\phi$  value) enforces that for the finite difference stencil all upwind neighbors are processed beforehand and that causality is preserved (only neighbors with a lower  $\phi$  value have an influence on the current mesh point). For  $N$  mesh points the FMM has a worst order complexity of  $\mathcal{O}(N \log N)$ , due to the sorting of the heap.

<sup>1</sup> In this work, the term *mesh* is used to describe a regularly axis-aligned discretized domain.

Regarding parallelizing the classical FMM, the central use of a priority queue is inherently detrimental to straight-forward parallelization of the algorithm. Some advancements were made based on a domain decomposition approach, originally on distributed-memory [9] and later on shared-memory platforms [4,18]. However, load balancing issues limited parallel efficiency. In other work [20], a parallel approach for distributed-memory systems using a restarted narrow band technique was presented, where synchronization between sub-domains is enforced when the front advances a distance of *stride width*.

Later, a shared-memory approach to load balancing was introduced in the context of multi-mesh re-distancing, by having more meshes than threads available and employing a *thread pool* technique [3]. However, the approach required a significantly large number of meshes to achieve high parallel efficiency. In [19] a similar approach as in [3] was used in combination with domain decomposition of a single mesh.

This paper combines the research presented in [3] and [19], by introducing an additional decomposition step to the multi-mesh FMM, thus enabling better parallel performance. The combination also allows for a unified approach to parallelizing the FMM on hierarchical meshes. The available approaches to parallelize the FMM on a single mesh (using so-called sub-meshes<sup>2</sup>) and in a multi-mesh setting, and their differences, are illustrated in Fig. 1.

### 3. Block-based FMM

In [3] it was shown that if the number of meshes is about 10 times bigger than the number of threads an acceptable parallel speedup is possible for the multi-mesh FMM. To improve the parallel performance in cases where the number of meshes is considerable smaller than that, e.g., a single mesh on Level 0, an additional decomposition step is employed.

A straightforward approach to increase the number of meshes would be to recursively split the largest mesh along the axis with the largest width into two sub-meshes until the desired number of meshes is reached. Drawbacks of this approach are that it is inherently serial and there is no lower bound on the size of the sub-meshes. This could lead to the creation of tiny (one mesh point wide) sub-meshes, deteriorating performance. In particular so as those meshes would still need to have a ghost layer. Therefore, we suggest a different approach: Only decompose meshes which are larger than a given block size (width) into sub-meshes which are smaller than or equal to the given block size.

In Fig. 2 an example of a multi-mesh domain consisting of two meshes is shown. The chosen block size of 10 leads to a decomposition of the larger mesh ( $14 \times 13$  mesh points), whilst the smaller mesh ( $7 \times 5$  mesh points) remains unchanged. The mesh points in the ghost layer are colored by their status, gray ones are set by the domain boundary, whereas the other colors are covered by a neighboring mesh. This approach inherently favors parallelism as the position where meshes are decomposed is independent from other present meshes. Additionally, setting the block size also allows to take cache sizes of the utilized computing platform into account to increase computational efficiency, as it is possible to tightly control the sizes of the sub-meshes and, therefore, their size in memory.

To enable the envisioned block-based FMM to be executed on hierarchical meshes, the neighbor relations between the (sub-)meshes have to be computed on each level and the ghost mesh points have to be set. Specifically, mesh points, which are covered by neighboring meshes and thus require a synchronized exchange, need to be identified. As part of this preparation step and as discussed before, the meshes are decomposed into sub-meshes according to our newly developed, optimized strategy, based on a given block size (see Section 3.1). After the preparation step, the multi-mesh FMM is applied without changes compared to [3] on each level of the hierarchical mesh. This allows for a straightforward comparison to the previous approach. In summary, the modified setup of the FMM consists of three sub-steps: Block decomposition, sub-mesh allocation, and ghost layer computation, see Fig. 3 and as discussed in the following sections.

#### 3.1. Block decomposition

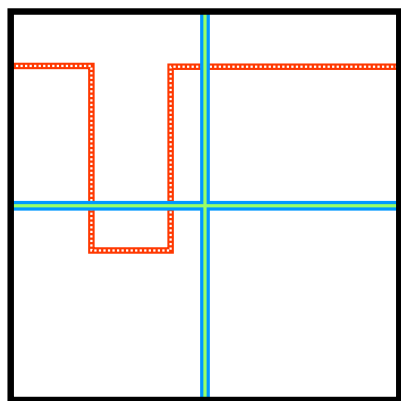
In general, decomposing a mesh is independent from decomposing other meshes, therefore, decomposing meshes is inherently parallel. In addition, the decomposition process is independently applicable to all spatial directions of a mesh, therefore, we will present it for simplicity only for the one-dimensional case (cf. Fig. 4). Let  $N$  be the number of mesh points (mesh size) of a given mesh and its start index given by  $S$ . For the maximal allowed block size  $B$ , the minimal number of necessary sub-meshes  $M$  is given by

$$M = \left\lceil \frac{N + B - 1}{B} \right\rceil. \quad (5)$$

Considering a regular mesh setting, based on the number of sub-meshes, the start indices  $S_i$  based on the global indexing scheme and number of mesh points  $N_i$  for each sub-mesh are calculated using  $N = M \cdot q + r$  ( $q$  and  $r$  are the unique quotient and remainder, respectively) as

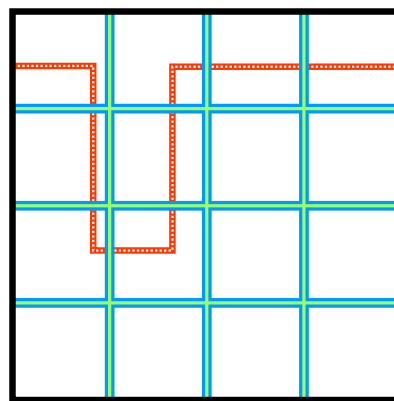
$$S_i = \begin{cases} S + i(q + 1) & \text{for } i < r \\ S + (i - r)q + r(q + 1) & \text{for } i \geq r \end{cases}, \quad (6)$$

<sup>2</sup> Sub-meshes are meshes created by decomposing the input mesh to create different data sets, enabling parallelization; sub-meshes have the same properties as a mesh.



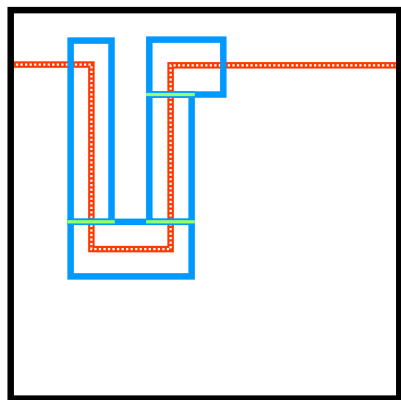
— Domain      — Decomposition  
 ..... Interface      — Synchronization

(a) Approach presented in [18]: Domain decomposition of a single mesh using one created sub-mesh per thread (four threads) is used, leading potentially to load balancing issues: The thread processing the lower right sub-mesh is idle as no interface is present.



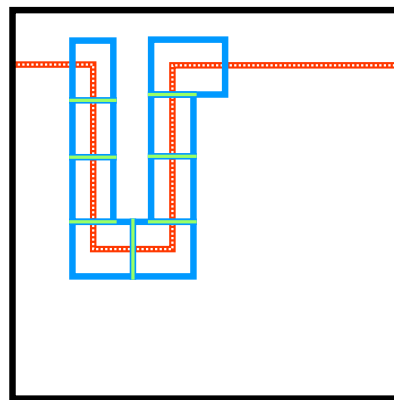
— Domain      — Decomposition  
 ..... Interface      — Synchronization

(b) Approach presented in [19]: Domain decomposition of a single mesh using multiple sub-meshes per thread and dynamically assigning the threads (i.e. four threads, but 16 meshes) is used, tackling the load balancing issue depicted in Figure 1(a).



— Domain      — Meshes  
 ..... Interface      — Synchronization

(c) Approach presented in [3]: Load balancing if multiple meshes per thread are available. The effectiveness of the parallelization depends on the number and size of the given meshes. The Eikonal equation is only solved in regions covered by a mesh.



— Domain      — Meshes  
 ..... Interface      — Synchronization

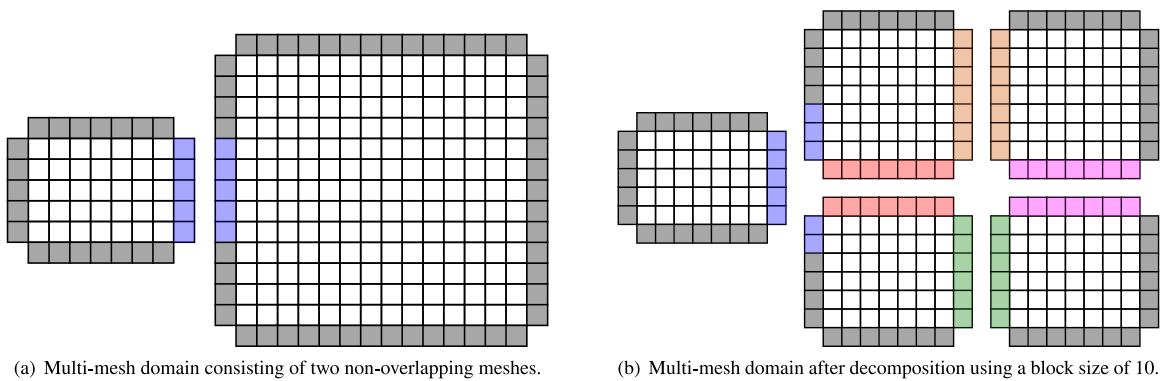
(d) Approach presented in this work: Improved parallel performance compared to approach shown in Figure 1(c) is achieved by automatically decomposing the given meshes into significantly more sub-meshes. This increases the mesh per thread ratio, which leads to more efficient load balancing and ultimately increases parallel performance.

**Fig. 1.** Approaches to parallelizing the FMM for a single mesh (top row) and multi-meshes (bottom row). The shape and position of the interface is inspired by a typical *trench* geometry in process TCAD simulation [3].

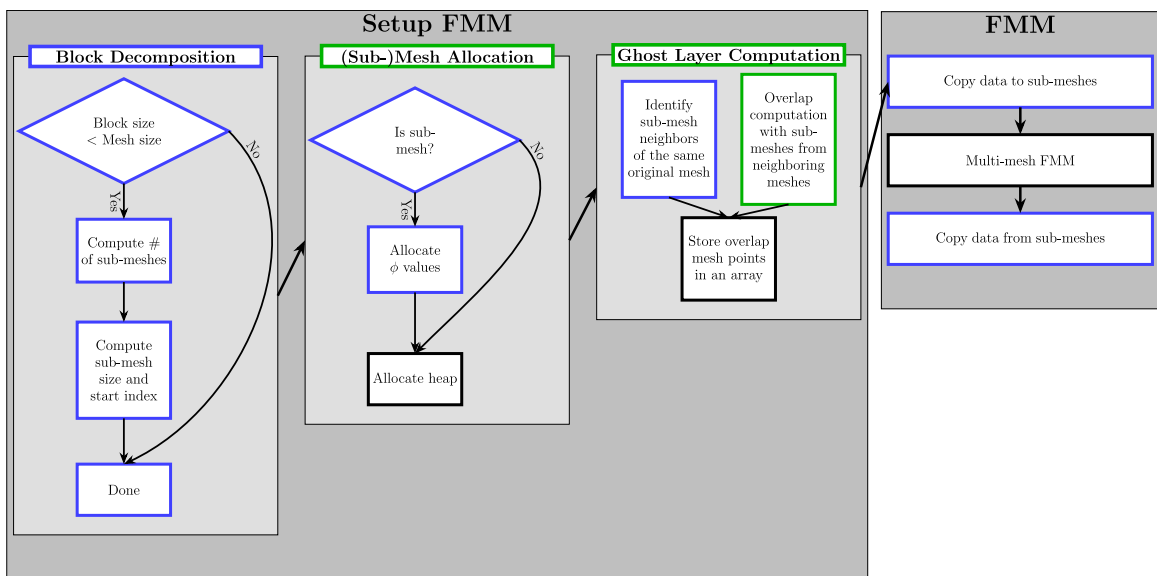
$$N_i = \begin{cases} q + 1 & \text{for } i < r \\ q & \text{for } i \geq r \end{cases} \quad (7)$$

The sub-meshes vary in size by at most one mesh point, this is the case if  $N$  is not divisible by  $M$ . This approach yields more equally sized sub-meshes compared to an approach where the original mesh is cut into  $B$ -sized sub-meshes but the last sub-mesh would have the size  $r$ : This may lead to cases with the last sub-mesh being only a single mesh point wide. Such meshes would require very frequent synchronization steps, drastically limiting parallel performance.

The block size is set to the same value in all spatial directions, as the mesh delta (distance between the mesh points in all spatial dimensions) is the same. In case of different mesh deltas, adapted block sizes for all dimensions are appropriate, but are out of scope of this paper.



**Fig. 2.** In (a) the computational domain consisting of two meshes and their appropriate ghost layer is shown: Gray mesh points in the ghost layer are given by the domain boundary, whereas the blue ones are covered by the neighboring mesh, which allows for seamless propagation of information between the meshes. In (b) the meshes after decomposition are shown. Only the bigger mesh is split into four sub-meshes, as the other one is smaller than the chosen block size of 10. The newly created mesh points in the ghost layers of the sub-meshes are colored differently. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



**Fig. 3.** Flow chart of the setup of the FMM and execution of the FMM itself. Steps colored in blue are new, their computational overhead vanishes if the chosen block size is larger than the mesh size. Green colored steps are modified compared to [3], whilst the black colored steps are unchanged. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

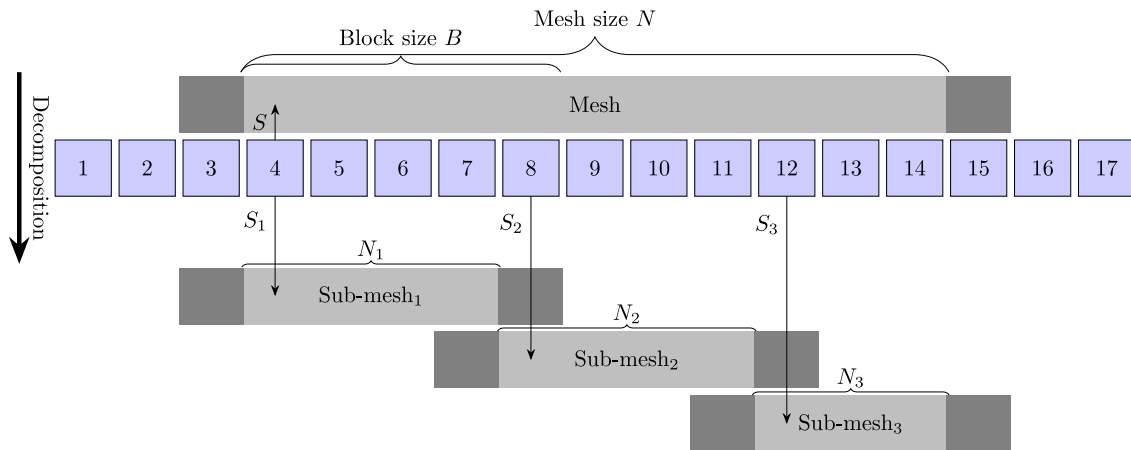
### 3.2. (Sub-)mesh allocation

The next step is the allocation of the (sub-)meshes. For newly created sub-meshes memory for the mesh points, including the ghost layer and the heap data structure, has to be allocated. Regarding the heap data structure, we utilize an indexed, pre-allocated binary heap. The pre-allocation avoids costly re-allocations should the heap outgrow the initial bounds during execution. In turn, the indexing allows for efficient updates of the priority (key) of entries when the value of a mesh point is recomputed by solving (3).

For meshes which have not been decomposed only the memory for the heap data structure is allocated. The mesh allocation step is parallelized by OpenMP tasks over the number of sub-meshes,<sup>3</sup> implicitly load balanced by a thread pool.<sup>4</sup> The mesh allocation step ends with a synchronization barrier to continue with the next step. During the mesh allocation step no data is copied to the sub-meshes. Data is only copied to and from the new sub-meshes, directly before

<sup>3</sup> For each sub-mesh an independent task is created.

<sup>4</sup> As soon as a thread finishes its current task, the thread has a new task assigned.



**Fig. 4.** The global indexing scheme is given by the blue squares ranging from 1 to 17. Above the given mesh (gray box) with its start index  $S = 4$  and mesh size  $N = 11$  covering all mesh points till 14 is shown. To the left and the right of the mesh the ghost layer is shown in dark gray. Below the global indexing scheme the created sub-meshes and their corresponding ghost layers are shown for the block size  $B = 5$ . The three sub-meshes, with their own start index  $S_i$  and mesh size  $N_i \leq B$  cover the same mesh points as the given mesh.

and after the FMM is executed. This allows to reuse the same decomposition for multiple time steps in a simulation until the underlying hierarchical mesh is changed.

### 3.3. Ghost layer computation

The neighbors (neighboring sub-meshes) of a sub-mesh are calculated in two steps: (1) Neighbors from the same original mesh are obtained via index calculation (the original mesh is regularly decomposed), because the neighboring sub-meshes of this kind either share an entire face<sup>5</sup> or no mesh point. (2) Neighbors from a different original mesh are determined by an overlap calculation of the sub-meshes with their individual ghost layers. The performance is improved by only considering sub-meshes of neighbors of the original mesh.

Each sub-mesh point in the ghost layer is either marked to belong to another mesh or as boundary mesh point. Boundary mesh points may receive their value either from the domain boundary condition or by interpolation from a coarser level of the hierarchical mesh. Mesh points marked to belong to a neighboring sub-mesh are collected on a per mesh basis to efficiently access them during the synchronized exchange step. The same parallelization strategy as for the sub-mesh allocation is employed.

## 4. Results and discussion

The benchmark results presented in the following were obtained on a single compute node of the Vienna Scientific Cluster 4 (VSC-4).<sup>6</sup> The compute node has two sockets, each equipped with an Intel Skylake Platinum 8174 processor running at 3.1 GHz with 24 cores. The processors have 32 KB, 1 MB, and 33 MB of L1-L3 cache, respectively as well as a total of 96 GB of main memory. No hyper-threading was used.

The benchmark implementation of the block-based multi-mesh FMM is based on the multi-mesh FMM presented in [3]. It uses C++11 features and OpenMP 4.5 for shared-memory parallelization. The gcc-9.1 compiler with optimization flags `-O3 -f1to` was used. To reduce non-uniform memory access (NUMA) effects, threads are pinned incrementally to individual cores.

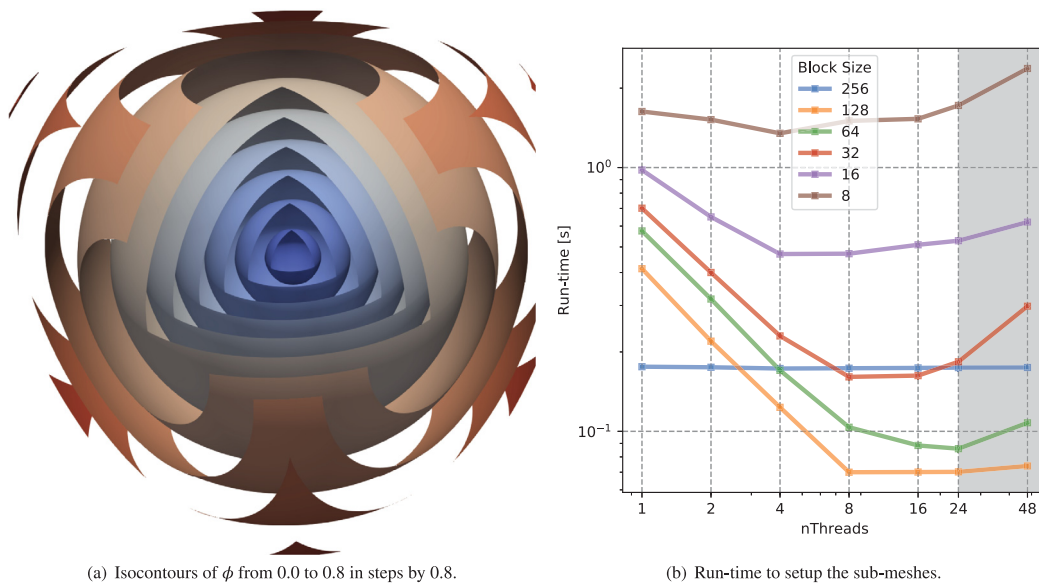
### 4.1. Point source

An established, fundamental test case for benchmarking re-distancing methods is the so-called *Point Source* test [19,20].<sup>7</sup> The equidistantly discretized domain covers the unit cube  $[-0.5, 0.5]^3$ , at its center  $([0,0,0])$  a single mesh point is set as source point (representing the interface, see Section 1). The boundary conditions are set symmetric in all spatial dimensions. The speed function is constant,  $f = 1$ . The computed isocontours (spheres centered around the point source) of this test case are shown in Fig. 5(a).

<sup>5</sup> All mesh points for one of the axis-aligned boundaries of the sub-mesh.

<sup>6</sup> <https://vsc.ac.at>.

<sup>7</sup> A direct comparison relative to the run-times reported in [19,20] is impossible, due to different computing architectures and incomplete implementation details on the chosen heap data structure.



**Fig. 5.** (a) The isocontours of  $\phi$  for the *Point Source* test case. (b) The run-time to setup the sub-meshes and to compute the neighbor relations for various block sizes and number of threads. The gray shaded area indicates the use of the second processor, thus influenced by NUMA effects.

First, the run-time of the setup of the FMM (setup sub-meshes and boundary conditions) is analyzed in Fig. 5(b). For a block size equal to the mesh size (256) only a single mesh covering the whole domain exists and, therefore, no parallelization is possible. As soon as the mesh is decomposed there is a significant serial run-time overhead as memory for the new sub-meshes has to be allocated and the number of ghost layer mesh points is increased. The usage of more threads decreases the run-time, leading to lower run-times than for the non-decomposed mesh. In case of block sizes 16 and 8 too many small meshes (4096 and 32768, respectively) are created and as a consequence the overhead cannot be compensated by the parallelization; the workload per parallel task is too small. The additional hardware resources from the second processor do not improve the run-time.

The run-time for the FMM itself and the corresponding parallel speedup is analyzed in Fig. 6 for block sizes ranging from 8 to 256 mesh points and stride widths (measured in multiples of the mesh delta). For the non-decomposed mesh the run-time is neither hardly affected by the stride width nor the number of threads. In the case where eight sub-meshes are created a serial speedup is measured for all stride widths smaller than 20. The serial speedup ranges from 1.01 (stride width of 0.5) to 1.14 (stride width of 3.5). This behavior is due to the lower number of mesh points in each heap and the better cache efficiency due to increased data locality. The parallel speedup saturates at eight threads, however, in this case a smaller stride width has an advantage. The source point located on exactly one sub-mesh requires a synchronization step to be exchanged to the neighboring sub-meshes, until then there is only one sub-mesh on which the FMM is executed serially.

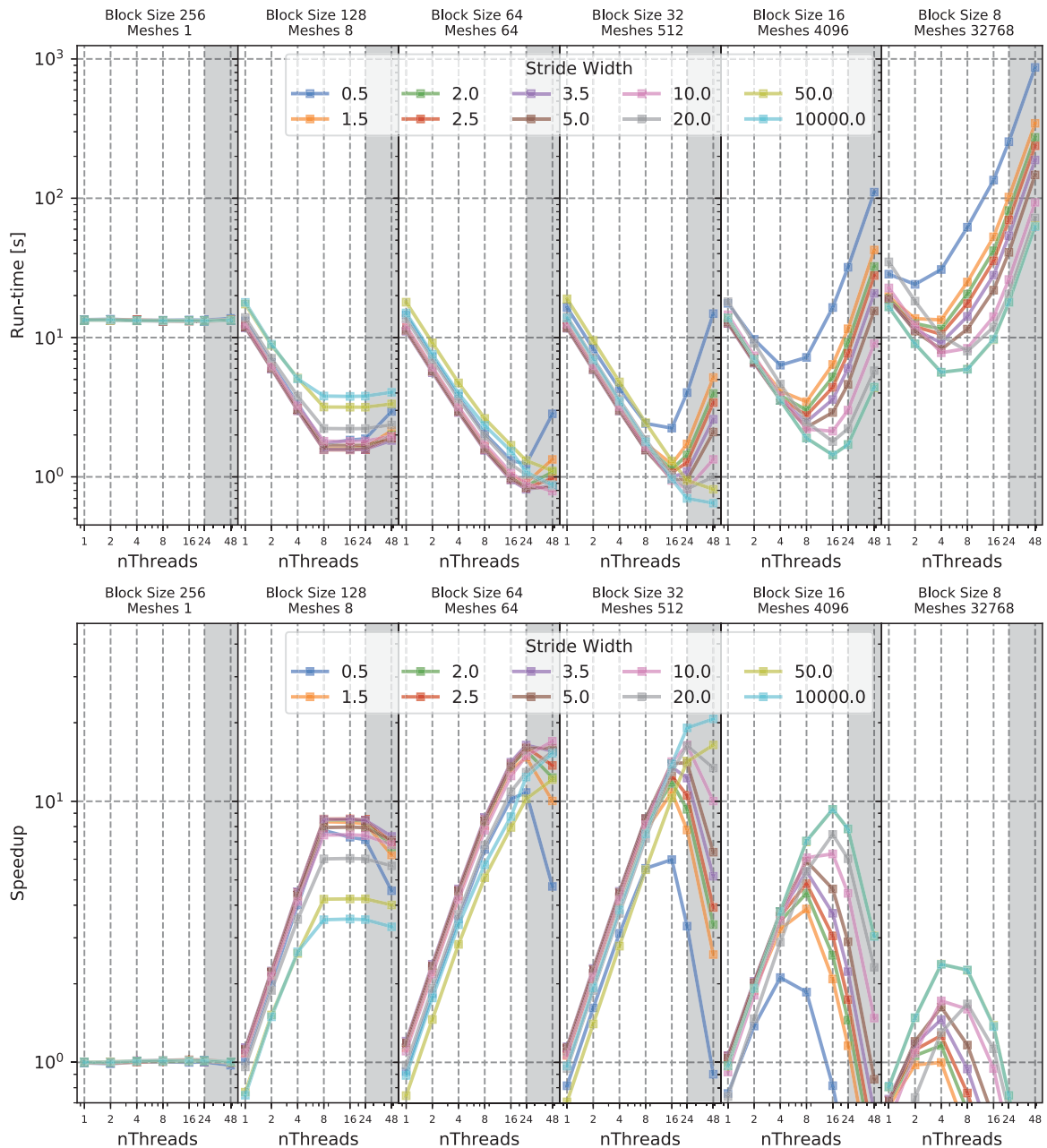
For higher sub-mesh counts a serial speedup is observed for stride widths smaller than 10. The best serial speedup of 1.21 is achieved for a block size of 64 and a stride width of 3.5. Using high thread counts the larger stride widths perform better, because the computational load per mesh decreases rapidly, but the synchronization overhead decreases slower in comparison (following a square-cube-law). A peak parallel speedup of 19.1 was achieved using 24 threads with a block size of 32 on a single processor.

A speedup using cores from the second processors socket is only observed for a block size of 64 and 32 and a large stride width (bigger than 10.0 and 50.0, respectively), because only then the computational load per task is big enough. The peak parallel speedup of 20.7 using all 48 threads is reached for the same parameters of block size and stride width as for the single processor case. The effects of NUMA severely limit parallel efficiency. The main issue for this approach used on NUMA systems is even more dominant for smaller stride widths, because the frequent synchronization steps cause task rescheduling. The OpenMP scheduling of the tasks to the cores does not consider where the memory of a mesh is located thus causing frequent indirect memory accesses.

#### 4.2. Mandrel

The *Mandrel* test geometry (cf. Fig. 7) is a representative scenario from the field of process TCAD: Two trenches are etched into a silicon wafer, one covering the full width of the simulation domain and the other covering half of the width of the simulation domain. Boundary conditions are set to symmetric. This example consists of two hierarchical levels, Level 0





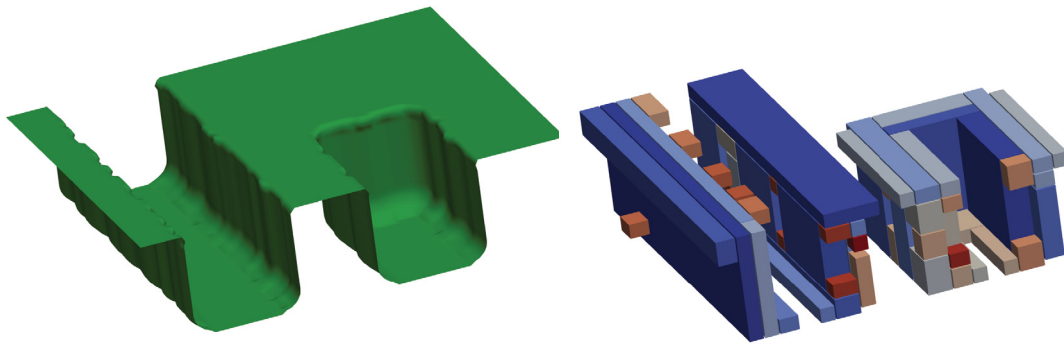
**Fig. 6.** Run-time of the FMM (top graph) and parallel speedup of the FMM (bottom graph) for the *Point Source* test case, using different values for the block size and stride width. The parallel speedup is compared to the run-time of the serial execution using a block size of 256.

being the coarse level and Level 1 being the finer level which has four times the spatial resolution in all directions. The single mesh on Level 0 has a size of  $84 \times 72 \times 312$ . The computed isocontours, for the constant speed function  $f = 1$ , are shown in Fig. 8.

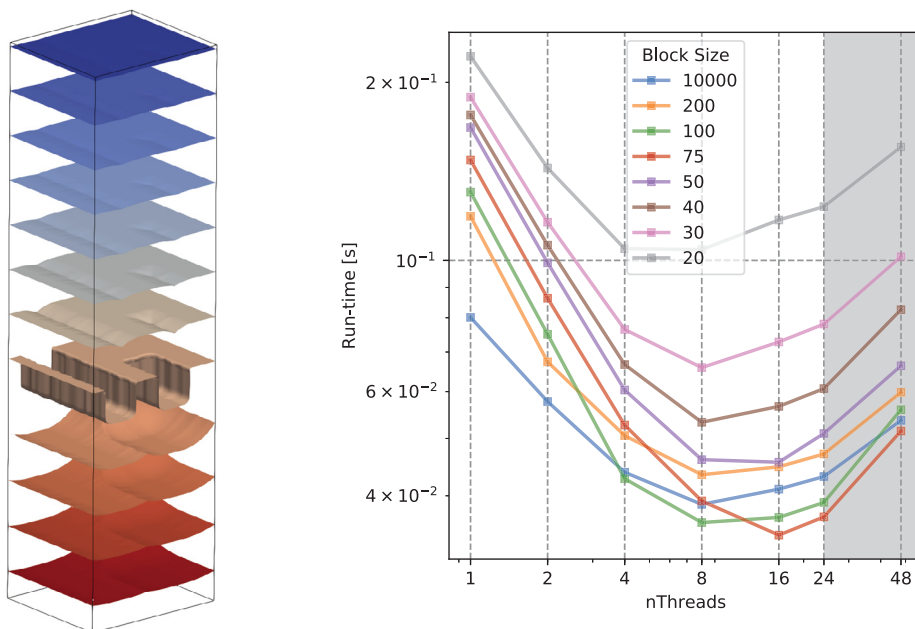
The meshes of the hierarchical mesh on Level 1 are placed around the trenches (cf. Fig. 7), especially around edges as they require a higher spatial resolution, because of their importance to the geometry. There are a total of 78 meshes with their sizes ranging from  $12 \times 288 \times 84$  to  $12 \times 12 \times 12$  mesh points.

For this test case the setup time is shortened by parallelization (cf. Fig. 8) even without any additional decomposition due to the 78 meshes existing on Level 1. The minimum run-time for the undecomposed case is reached using eight threads. The decomposition approach increases the scalability (at the cost of a slower serial execution) yielding the best performance for a block size of 75 and using 16 threads. The parallel speedup saturates between 8 and 16 threads for all





**Fig. 7.** The interface (0-level-set) of the *Mandrel* test case on the left and on the right the mesh placement on Level 1. The meshes are colored by their size from the biggest mesh (size  $12 \times 288 \times 84$ ) in blue to the smallest mesh (size  $12 \times 12 \times 12$ ) in red. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



(a) Isocontours of  $\phi$  from  $-0.7$  to  $0.5$  in steps by  $0.1$ .

(b) Run-time to setup the sub-meshes for various thread numbers.

**Fig. 8.** In (a) the computational domain and the isocontours of  $\phi$  are shown for the *Mandrel* test case. In (b) the time to setup the sub-meshes and to compute the neighbor relations for various block sizes and number of threads is shown. The gray shaded area indicates the use of the second processor, thus influenced by NUMA effects.

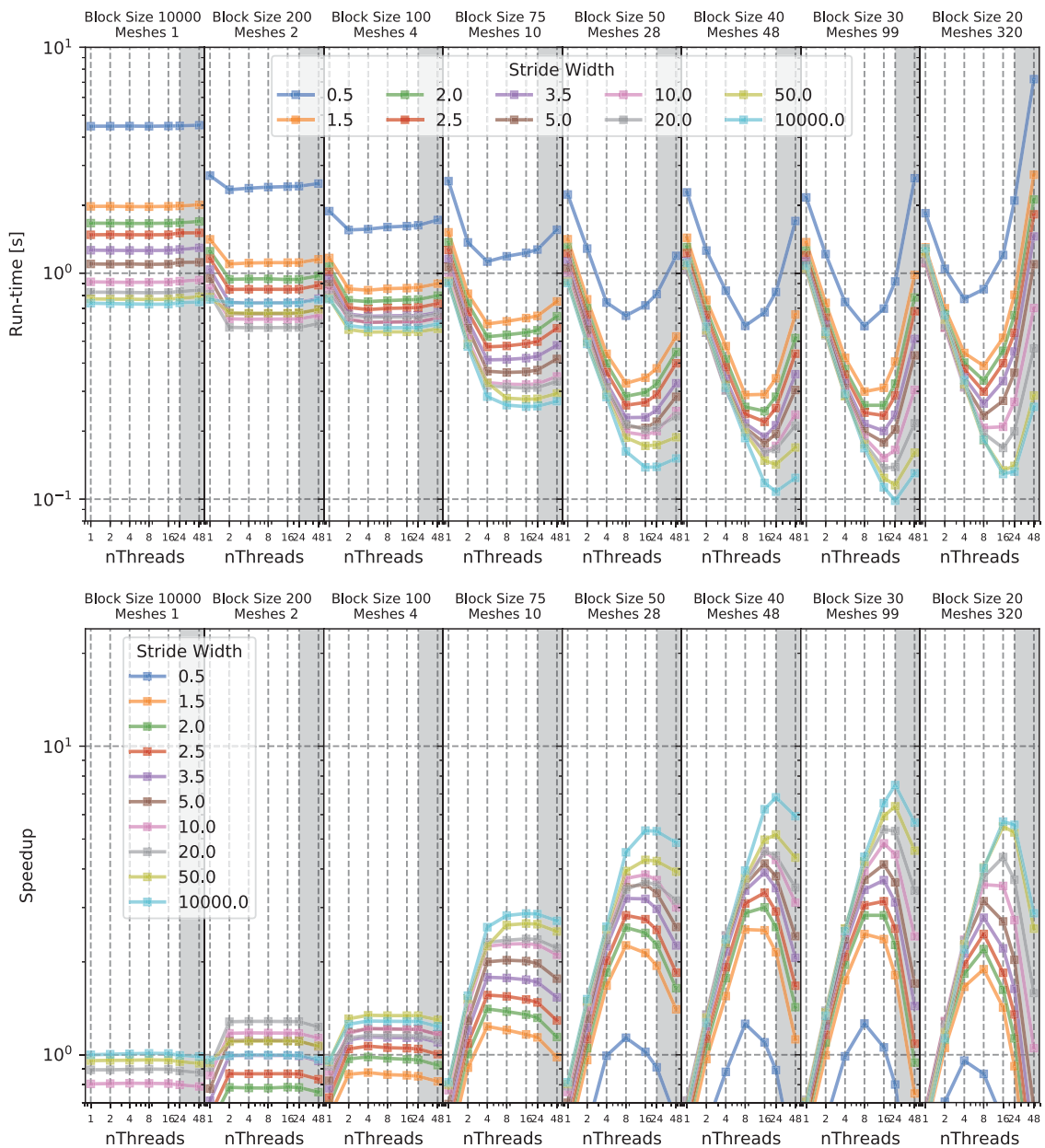
block sizes. Again, using the second processor deteriorates the performance, because of the previously discussed NUMA effects.

Considering the run-time of the FMM itself, a separate analysis on Level 0 (cf. Fig. 9) and Level 1 (cf. Fig. 10) is conducted in the following. On neither of those levels a performance increase was measured when the second processor (48 threads) was used.

#### 4.2.1. Level 0

Without decomposition a larger stride width performs better in all cases (cf. Fig. 9). Compared to the *Point Source* test case, where little influence of the stride width was measured, the (by an order of magnitude) shorter run-time and the significantly larger extent in one spatial dimension show that the impact of the stride width on the serial performance is not negligible, because every restart, even in case of no synchronization, introduces performance overhead.

A block size of 200 decomposes the mesh into two sub-meshes, enabling better performance for two threads for a stride width ranging from 10 to 50. Regarding the maximal stride width the sub-meshes are treated serially as the interface is completely located in one of the sub-meshes. For a stride width smaller than 10.0 the overhead caused by the additional



**Fig. 9.** Run-time and speedup compared to the serial execution using a block size of 10000 of the FMM for the *Mandrel* test case on Level 0 (coarse level) of the hierarchical mesh.

synchronization steps does not allow the serial run-time to break even. A serial speedup as observed in the *Point Source* test case 4.1 was not measured. This is because the decomposition creates only two strictly dependent sub-meshes compared to the eight almost independent sub-meshes for the *Point Source* test case.

Starting with a block size of 75 and less the mesh is additionally split in directions other than the z-axis, yielding a much better parallel performance. Those splits create sub-meshes with almost independent solutions, compared to the splits along the z-axis. The lowest run-time is always achieved with the maximal stride width, because all the sub-meshes carry little computational load. A maximal parallel speedup of 7.5 is achieved using block size 30, creating a total of 99 meshes. Smaller block sizes reduce the parallel speedup as the synchronization overhead increases.

#### 4.2.2. Level 1

The run-time on Level 1 is about three times longer than on Level 0, so, run-time on this level has a bigger impact (cf. Fig. 10). On Level 1 a serial speedup is observed if the stride width is between 1.5 and 10.0. The best serial speedup of

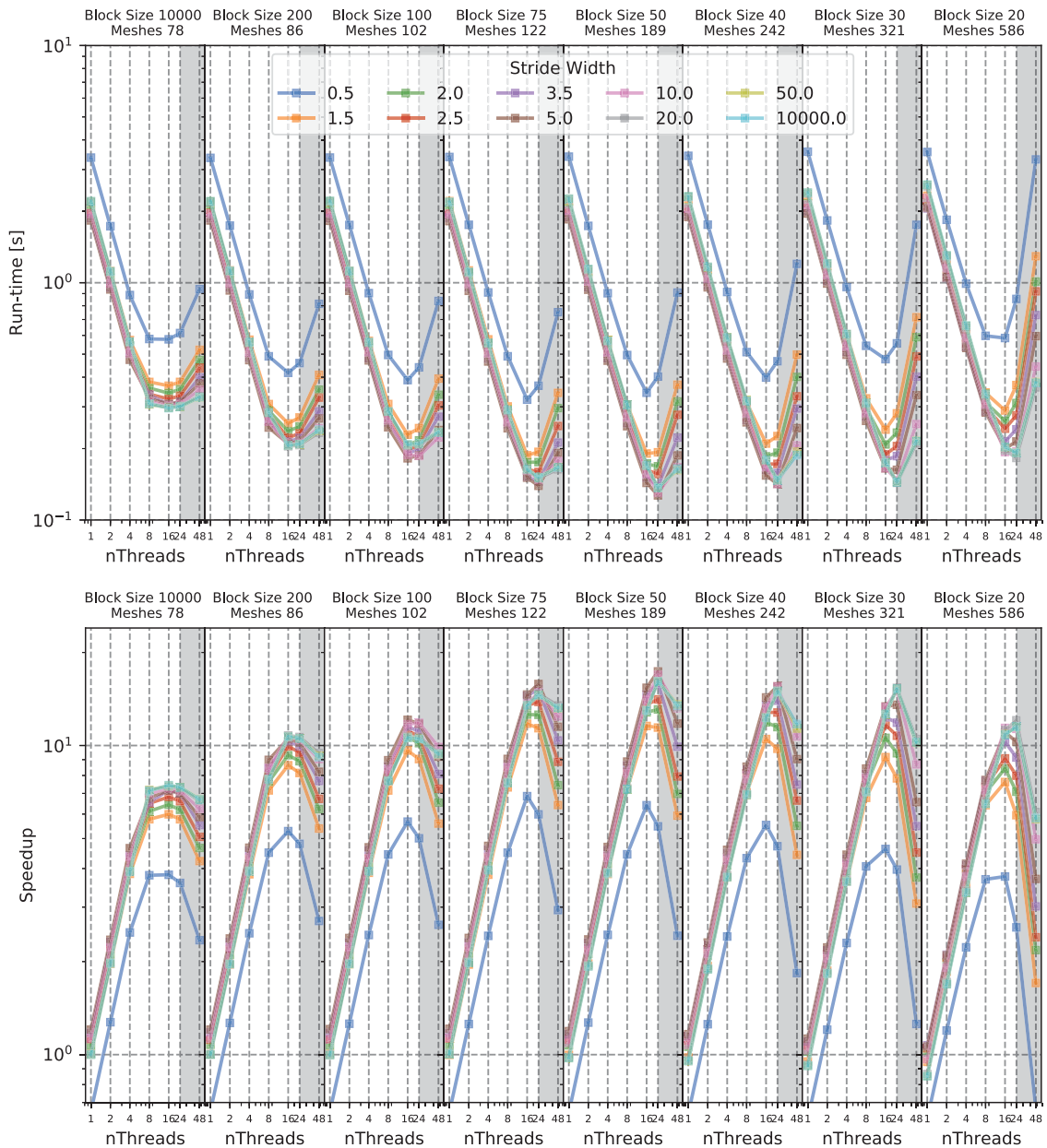
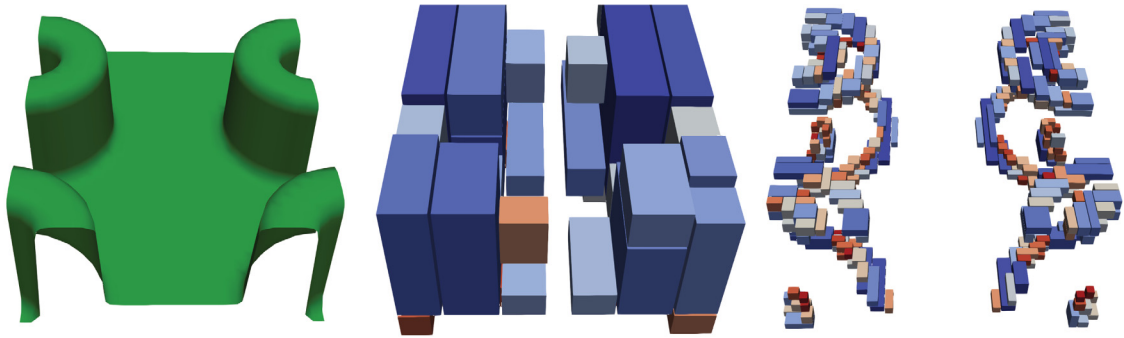


Fig. 10. Run-time and speedup compared to the serial execution using a block size of 10 000 of the FMM for the *Mandrel* test case on Level 1 (fine level) of the hierarchical mesh.

1.21 is measured for a stride width of 5.0 and a block size of 50. The reason is that with a lower stride width mesh points located in the ghost layer which are interpolated from a coarser mesh, are not immediately used in the FMM. This prevents unnecessary computations as most of the ghost layer mesh points have a higher  $\phi$  value than their non-ghost layer mesh point neighbors.<sup>8</sup> The block size itself does not influence the serial speedup in this case, as for all block sizes down to 40 the serial speedup for the stride width of 5.0 is around 1.2. For smaller block sizes the synchronization overhead slightly reduces the serial speedup.

Considering the parallel performance on Level 1, the base line code without any decomposition has a peak parallel speedup of 7.4 using 16 threads for the maximal stride width. The block based approach doubles the parallel speedup to 15.4 at 16 threads (block size 50 and stride width 5.0). Using all available cores on one processor yields the peak parallel

<sup>8</sup> Those mesh points typically do not represent sources, but they might be in case of an unfortunate mesh placement and, therefore, must be treated as sources.



**Fig. 11.** The interface of the *Quad-Hole* test geometry on the left, in the middle the mesh placement on Level 1, and on the right the mesh placement of Level 2. The meshes are colored by their size from the biggest (blue) to the smallest (red). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

speedup of 17.4 (same block size and stride width). Remarkably, the best performance even for small block sizes was not achieved using the maximal stride width but a smaller one (5.0), due to treating ghost layer mesh points as sources, which, as previously indicated, is necessary for algorithm correctness and robustness. A stride width of 0.5 yields the worst performance for all block sizes and thread numbers, because of the numerous synchronization steps.

#### 4.3. *Quad-Hole*

The *Quad-Hole* test case is another typical scenario from process TCAD [3]. This test case has four regions of interest each in the shape of a quarter hole (cf. Fig. 11). Around the edges of those holes the meshes on the higher levels of the hierarchical mesh are clustered. In particular the test case from [3] using 48 meshes corresponds to Level 1 of the hierarchical meshes and the 303 mesh test case corresponds to the Level 2 of the hierarchical meshes.

The mesh on Level 0 has a size of  $38 \times 28 \times 30$ , still allowing for parallelization, but due to the little serial run-time of only 0.013 s (compare Level 1 0.402 s [=  $30 \times$  Level 0] and Level 2 1.568 s [=  $120 \times$  Level 0 / =  $4 \times$  Level 1]), detailed results of the parallelization have been omitted for Level 0. On Level 1, the meshes are sized between  $68 \times 20 \times 52$  and  $12 \times 16 \times 12$ , and on Level 2 from  $164 \times 20 \times 12$  to  $12 \times 12 \times 12$ . In Fig. 12 and in Fig. 13 the run-time and the corresponding speedups for Level 1 and Level 2 are shown, respectively, and are discussed in the following.

##### 4.3.1. Level 1

A serial speedup of 1.14 is achieved for a stride width of 3.5 and block size 75. In this test case the serial speedup vanishes by using smaller block sizes. Considering parallelization, using a small stride width (2.0 to 10.0) reduces the run-time for up to eight threads, with the stride width 5.0 yielding typically the best performance. At 24 threads the maximal stride width outperforms all other stride widths. The smallest stride width (0.5) is always the worst except for a block size of 10 and less than four threads used. The peak performance is achieved for a maximal stride width and a block size of 50 and utilizing all 24 threads of a single processor. Comparing the speedup without any decomposition (speedup of 8.2 for stride width of 20), the decomposition approach using a block size of 50 and also a stride width of 50 achieved a speedup of 12.5.

The recurring optimal block size of 50 is related to the size of the L2-cache as sub-meshes occupy less than  $50^3 \times 8$  bytes<sup>9</sup> (1 MB) which is the size of the L2-cache (cf. Section 4). NUMA effects significantly deteriorate the performance in all cases.

##### 4.3.2. Level 2

The run-time on Level 2 is about four times longer than on Level 1 thus parallelization has the most impact on this level. For a stride width between 2.0 and 5.0 a serial speedup of up to 1.05 is measured. Again, a smaller block size reduces this speedup, until serial slowdown manifests for a block size of 30.

The highest parallel speedup reached for Level 2 is 16.6 for a block size of 75 and stride width of 20. A comparison to the achieved speedup without any decomposition (a speedup of 16.5 for stride width of 20) shows that for a high number of meshes the performance is barely affected by the decomposition. This shows that the devised decomposition approach solves the issue with load balancing for a small number of given meshes for the multi-mesh FMM. Only with a small block size ( $\leq 20$ ) the performance starts to deteriorate significantly, caused by the overhead of the additional created sub-meshes.

<sup>9</sup> The size of a double is eight bytes.

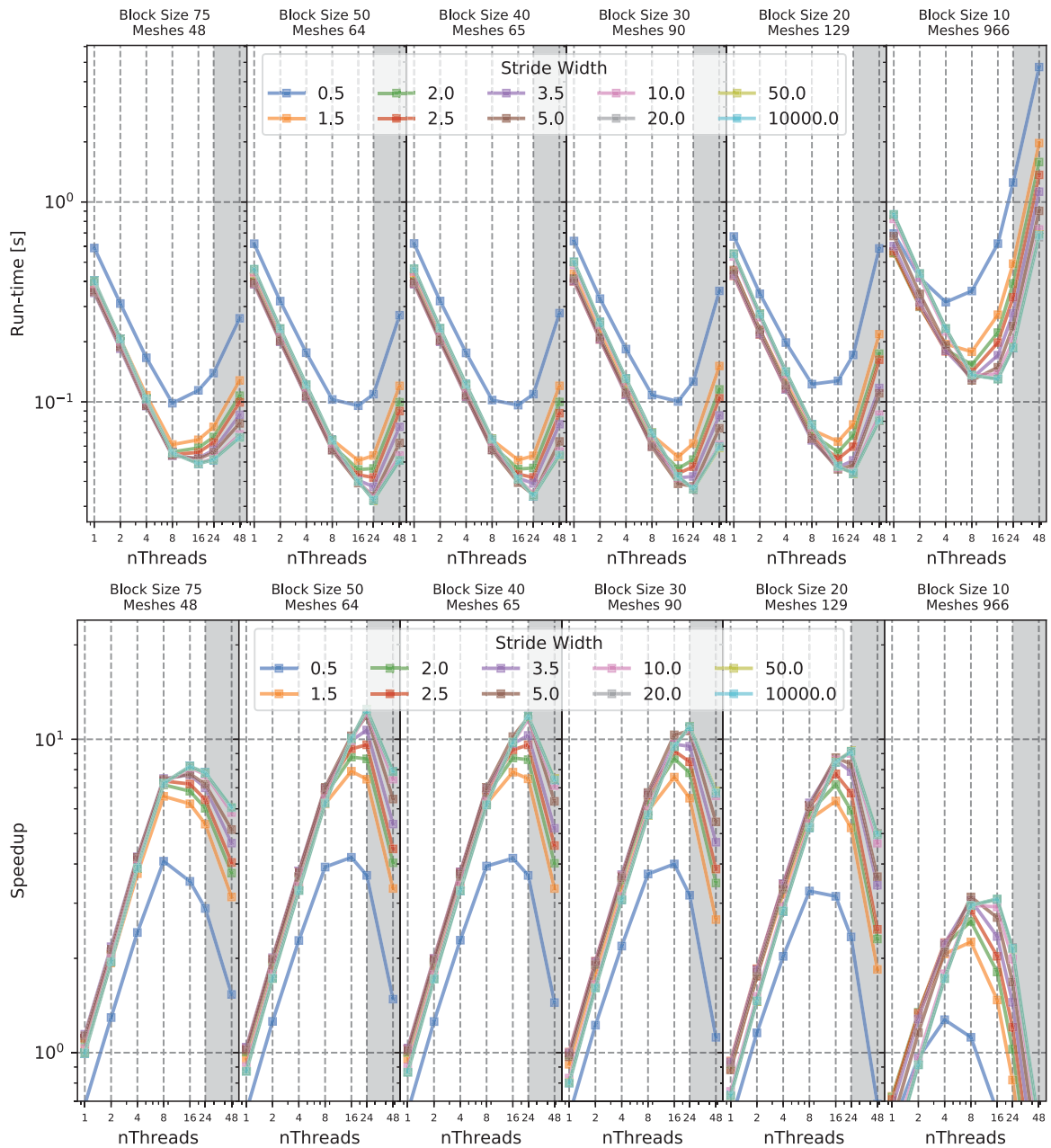
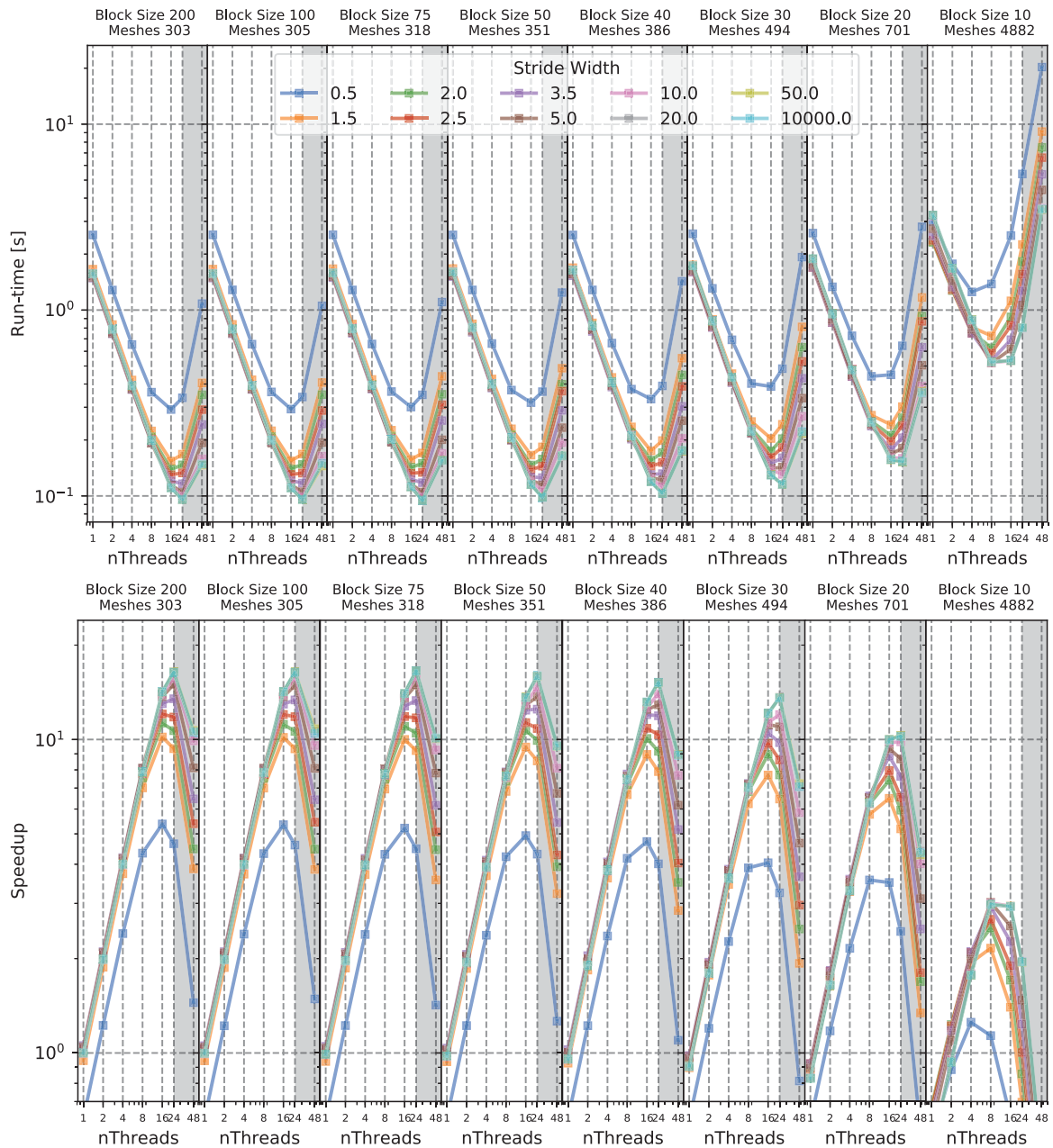


Fig. 12. Run-time and speedup of the FMM, using different values for the block size and stride width, compared to the serial execution using a block size of 10000 for the *Quad-Hole* test case on Level 1 of the hierarchical mesh.

### 5. Summary

The parallelization of the FMM has been studied in the context of hierarchical meshes and a unified approach for all levels of the hierarchical mesh is presented. Load balancing issues in the parallelization of previous attempts are overcome by a new decomposition strategy of the given meshes into smaller sub-meshes, thus automatically increasing the total mesh count and thereby increasing load balancing and parallel efficiency. For a generic point source test case speedups of up to 19.1 have been achieved for 24 threads. For two representative test cases from topography simulations in the field of microelectronics, speedups of up to 17.4 have been achieved for 24 threads. The extensive parameter study on the size of the sub-meshes has shown that limiting the sub-mesh size to fit into the level 2 cache of the processor performs best, allowing to balance the number of available meshes and synchronization overheads. Furthermore, the stride width analysis showed that for less than 8 threads (especially for serial execution) a small stride width (2.5 to 10.0) tends to





**Fig. 13.** Run-time and speedup of the FMM, using different values for the block size and stride width, compared to the serial execution using a block size of 10000 for the *Quad-Hole* test case on Level 2 of the hierarchical mesh.

improve performance whilst for higher thread counts the largest stride width (larger than the largest mesh size) performs best, due to the resulting reduced computational load per mesh.

**Acknowledgments**

The financial support by the *Austrian Federal Ministry for Digital and Economic Affairs* and the *National Foundation for Research, Technology and Development*, Austria is gratefully acknowledged. The authors acknowledge *TU Wien Bibliothek*, Austria for financial support through its Open Access Funding Programme. The computational results presented have been achieved using the Vienna Scientific Cluster (VSC).



## References

- [1] S. Bak, J. McLaughlin, D. Renzi, Some improvements for the Fast Sweeping Method, *SIAM J. Sci. Comput.* 32 (5) (2010) 2853–2874, <http://dx.doi.org/10.1137/090749645>.
- [2] K. Crane, C. Weischedel, M. Wardetzky, Geodesics in heat, *ACM Trans. Graph.* 32 (5) (2013) 1–11, <http://dx.doi.org/10.1145/2516971.2516977>.
- [3] G. Diamantopoulos, A. Hössinger, S. Selberherr, J. Weinbub, A shared memory parallel multi-mesh fast marching method for re-distancing, *Adv. Comput. Math.* 45 (4) (2019) 2029–2045, <http://dx.doi.org/10.1007/s10444-019-09683-z>.
- [4] G. Diamantopoulos, J. Weinbub, S. Selberherr, A. Hössinger, S. Selberherr, A. Hossinger, S. Selberherr, Evaluation of the shared-memory parallel fast marching method for re-distancing problems, in: *Proceedings of the International Conference on Computational Science and Its Applications (ICCSA)*, IEEE, 2017, pp. 1–8, <http://dx.doi.org/10.1109/ICCSA.2017.7999648>.
- [5] E.W. Dijkstra, A note on two problems in connexion with graphs, *Numer. Math.* 1 (1) (1959) 269–271, <http://dx.doi.org/10.1007/BF01386390>.
- [6] F. Gibou, R. Fedkiw, S. Osher, A review of level-set methods and some recent applications, *J. Comput. Phys.* 353 (2018) 82–109, <http://dx.doi.org/10.1016/j.jcp.2017.10.006>.
- [7] T. Gillberg, A semi-ordered fast iterative method (SOFI) for monotone front propagation in simulations of geological folding, in: *Proceedings of the International Congress on Modelling and Simulation (MODSIM)*, 2011, pp. 641–647.
- [8] J.V. Gomez, D. Alvarez, S. Garrido, L. Moreno, Fast methods for Eikonal Equations: An experimental survey, *IEEE Access* 7 (2019) 39005–39029, <http://dx.doi.org/10.1109/ACCESS.2019.2906782>, [arXiv:1506.03771](https://arxiv.org/abs/1506.03771).
- [9] M. Herrmann, *A Domain Decomposition Parallelization of the Fast Marching Method*, Center for Turbulence Research, 2003, pp. 213–225.
- [10] W.-K. Jeong, R.T. Whitaker, A fast iterative method for Eikonal Equations, *SIAM J. Sci. Comput.* 30 (5) (2008) 2512–2534, <http://dx.doi.org/10.1137/060670298>.
- [11] B. Lee, J. Darbon, S. Osher, M. Kang, Revisiting the redistancing problem using the Hopf–Lax Formula, *J. Comput. Phys.* 330 (2017) 268–281, <http://dx.doi.org/10.1016/j.jcp.2016.11.005>.
- [12] S. Osher, R. Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*, Springer, 2011, p. 153.
- [13] S. Osher, J.A. Sethian, Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton–Jacobi Formulations, *J. Comput. Phys.* 79 (1) (1988) 12–49, [http://dx.doi.org/10.1016/0021-9991\(88\)90002-2](http://dx.doi.org/10.1016/0021-9991(88)90002-2).
- [14] A.M. Popovici, J.A. Sethian, 3-D imaging using higher order fast Marching Traveltimes, *Geophysics* 67 (2) (2002) 604–609, <http://dx.doi.org/10.1190/1.1468621>.
- [15] J.A. Sethian, A fast marching level set method for Monotonically Advancing Fronts, *Proc. Natl. Acad. Sci.* 93 (1996) 1591–1595, <http://dx.doi.org/10.1073/pnas.93.4.1591>.
- [16] J.A. Sethian, A.M. Popovici, 3-D traveltime computation using the fast marching method, *Geophysics* 64 (2) (1999) 516–523, <http://dx.doi.org/10.1190/1.1444558>.
- [17] J.A. Sethian, A. Vladimirov, Fast methods for the eikonal and related Hamilton–Jacobi Equations on unstructured meshes, *Proc. Natl. Acad. Sci.* 97 (11) (2000) 5699–5703, <http://dx.doi.org/10.1073/pnas.090060097>.
- [18] J. Weinbub, A. Hössinger, Shared-memory parallelization of the fast marching method using an overlapping domain-decomposition approach, in: *Proceedings of the High Performance Computing Symposium (HPC)*, 2016, pp. 1–8, <http://dx.doi.org/10.22360/SpringSim.2016.HPC.052>.
- [19] J. Yang, An easily implemented, block-based fast marching method with superior sequential and parallel performance, *SIAM J. Sci. Comput.* 41 (5) (2019) C446–C478, <http://dx.doi.org/10.1137/18M1213464>, [arXiv:1811.00009](https://arxiv.org/abs/1811.00009).
- [20] J. Yang, F. Stern, A highly scalable massively parallel fast marching method for the eikonal equation, *J. Comput. Phys.* 332 (2017) 333–362, <http://dx.doi.org/10.1016/j.jcp.2016.12.012>, [arXiv:1502.07303](https://arxiv.org/abs/1502.07303).