DISSERTATION

Parallel Velocity Extension and Load-Balanced Re-Distancing on Hierarchical Grids for High Performance Process TCAD

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

unter der Betreuung von
Assistant Prof. Privatdoz. Dipl.-Ing. Dr.techn. Josef Weinbub, BSc
O.Univ.Prof. Dipl.-Ing. Dr.techn. Dr.h.c. Siegfried Selberherr

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik
von

Dipl.-Ing. Michael Quell, BSc

Matrikelnummer 01226394

Wien, im November 2021 _____

Abstract

The continuous developments and miniaturization of manufacturing processes for semiconductor devices require physical simulations to reduce the number of costly conventional experiments involved in the design and production processes. Most prominent are physical simulations which model individual physical processing steps like etching or deposition. These topography-changing simulations are commonly based on the level-set method, because of its capability to efficiently represent complex three-dimensional device structures. High accuracy demands of those simulations require the application of complex and, therefore, computationally expensive physical models.

In this work, three parallel algorithms belonging to two computational steps of the level-set method are introduced. The algorithms significantly reduce overall run-time and improve accuracy. The algorithms are tailored to simulations using adaptive discretizations with hierarchical grids to efficiently handle sharp features, e.g., corners and edges. The focus of the presented research is to efficiently utilize shared-memory parallel computing systems to stem the increasingly demanding level-set based physical simulations.

The first algorithm belongs to the computational step *Velocity Extension* which extends the velocity describing the deformation of an arbitrary structure from the structure's surface to the entire computational domain. The developed velocity extension algorithm is based on the fast marching method. The fast marching method allows to extend the velocity in a single pass through the computational domain by means of a strict ordering of the computations. The key advantage of the developed velocity extension algorithm is a relaxed ordering of the computations. This not only reduces the computational complexity but also enables parallelism. Different stages of the developed algorithm are evaluated by comparing run-times measured on representative computing systems. A run-time reduction by a factor of 18.5 using 10 threads has been achieved.

The second algorithm belongs to the computational step *Re-Distancing* which creates or restores a numerically stable representation of the structure by computing the signed-distance field relative to the surface of the structure. This algorithm is also based on the fast marching method, but because of self-referred data dependencies a different parallelization strategy was developed. A domain decomposition is introduced to increase the granularity of the parallel tasks. This enables a better implicit load-balancing compared to the native decomposition provided by the given hierarchical grid. A speedup of more than 17.4 has been achieved when using 24 threads.

Finally, a bottom-up correction algorithm was developed, also belonging to the computational step *Re-Distancing*, which increases the accuracy of the signed-distance field computed by the second algorithm. This correction algorithm utilizes the signed-distance field on higher resolved regions of hierarchical grids to also reduce the error in lower resolved regions. The developed algorithm adds negligible computational overhead to the second algorithm, yet reduces the error around corners by a factor of up to 2.7.

Combining all developed algorithms, it is shown that the run-time of a representative physical simulation is more than halved whilst the accuracy is further improved.

Kurzfassung

Die kontinuierlichen Entwicklungen und die Miniaturisierung der Herstellungsprozesse für Halbleiterbauelemente erfordern physikalische Simulationen, um die Zahl der kostspieligen konventionellen Experimente in den Entwurfs- und Produktionsprozessen zu verringern. Am bekanntesten sind physikalische Simulationen, die einzelne physikalische Prozessschritte wie Ätzen oder Abscheiden modellieren. Diese topografieverändernden Simulationen basieren gewöhnlich auf der Level-Set-Methode, da sie komplexe dreidimensionale Bauelementstrukturen effizient darstellen kann. Die hohen Genauigkeitsanforderungen dieser Simulationen erfordern die Anwendung komplexer und daher rechenintensiver physikalischer Modelle.

In dieser Arbeit werden drei parallele Algorithmen eingeführt, die zu zwei Rechenschritten der Level-Set-Methode gehören. Die Algorithmen verringern die Gesamtlaufzeit erheblich und verbessern die Genauigkeit. Die Algorithmen sind an Simulationen angepasst, die adaptive Diskretisierungen mit hierarchischen Gittern verwenden, um spitze Geometrien, z.B. Ecken und Kanten, effizient zu behandeln. Der Schwerpunkt der hier vorgestellten Forschung ist die effiziente Nutzung paralleler Rechensysteme mit gemeinsamem Speicher, um die immer anspruchsvolleren Level-Set-basierten physikalischen Simulationen zu bewältigen.

Der erste Algorithmus gehört zum Rechenschritt *Velocity Extension*, der die Geschwindigkeit, die die Verformung einer beliebigen Struktur beschreibt, von der Oberfläche der Struktur auf das gesamte Simulationsgebiet ausdehnt. Der entwickelte Algorithmus zur Geschwindigkeitserweiterung basiert auf der Fast-Marching-Methode. Die Fast-Marching-Methode ermöglicht es, die Geschwindigkeit in einem einzigen Durchgang durch das Simulationsgebiet zu berechnen, indem die Berechnungen in einer strengen Reihenfolge durchgeführt werden. Der Hauptvorteil des entwickelten Algorithmus ist eine relaxierte Reihenfolge der Berechnungen. Diese reduziert nicht nur die Komplexität der Berechnungen, sondern ermöglicht auch Parallelität. Verschiedenen Entwicklungsstufen des Algorithmus werden durch den Vergleich der auf repräsentativen Rechensystemen gemessenen Laufzeiten bewertet. Eine Laufzeitverkürzung um den Faktor 18.5 wurde bei der Verwendung von 10 Threads erreicht.

Der zweite Algorithmus gehört zum Rechenschritt *Re-Distancing*, der eine numerisch stabile Repräsentation der Struktur durch Berechnung des vorzeichenbehafteten Abstandsfeldes relativ zur Oberfläche der Struktur erzeugt oder wiederherstellt. Dieser Algorithmus basiert ebenfalls auf der Fast-Marching-Methode, aber wegen der selbstbezogenen Datenabhängigkeiten wurde eine andere Parallelisierungsstrategie entwickelt. Es wird eine Gebietszerlegung eingeführt, um die Granularität der parallelen Aufgaben zu erhöhen. Dies ermöglicht einen besseren impliziten Lastausgleich im Vergleich zur nativen Gebietszerlegung, die durch das gegebene hierarchische Gitter bereitgestellt wird. Eine Geschwindigkeitssteigerung von mehr als 17.4 wurde bei der Verwendung von 24 Threads erreicht.

Schließlich wurde ein Bottom-up-Korrekturalgorithmus entwickelt, der ebenfalls zum Rechenschritt *Re-Distancing* gehört und die Genauigkeit des vom zweiten Algorithmus berechneten vorzeichenbehafteten Abstandsfeldes erhöht.

Dieser Korrekturalgorithmus benutzt das vorzeichenbehaftete Abstandsfeld in höher aufgelösten Gebieten des hierarchischen Gitters, um auch den Fehler in niedriger aufgelösten Gebieten zu reduzieren. Der entwickelte Algorithmus fügt dem zweiten Algorithmus einen vernachlässigbaren Rechenaufwand hinzu, reduziert aber den Fehler bei Ecken um einen Faktor von bis zu 2.7.

Durch die Kombination aller entwickelten Algorithmen wird gezeigt, dass sich die Gesamtlaufzeit einer repräsentativen physikalischen Simulation mehr als halbiert, während die Genauigkeit weiter verbessert wird.

Acknowledgement

First, I want to thank the whole Institute for Microelectronics and all its members, for their welcoming support, when I started my journey in 2018.

Especially, I want to thank my supervisor Josef Weinbub, who is also the head of the Christian Doppler Laboratory for High Performance Technology Computer-Aided Design, for his continuous support and encouragement for my scientific path. He not only provided necessary guidance, but also excelled with personal wisdom.

I also want to thank my secondary supervisor Siegfried Selberherr, for providing excellent feedback content wise and grammatical. His eyes neither missed a single punctuation mark, nor a wrongly sized white space.

Additional thanks is directed to Andreas Hössinger from Silvaco Europe Ltd. who fueled the research with interesting questions and practical issues stemming from real world problems. I am grateful for his insights providing feedback and the valuable discussions.

From my colleagues, I would like to thank Alexander Toifl, for answering and explaining questions related to semiconductor devices to great detail, which lead to fruitful scientific collaborations. Also Paul Manstetten deserves my gratitude, because he provided me with high quality discussions and precise feedback especially related to high performance computations and presenting scientific results. I want to thank Luiz Felipe Aguinsky and Christoph Lenz, because our shared office lead to many insightful discussions on the chalkboard, leading to full blown research ideas and papers.

My thank is also directed towards the remaining present and former members of the Christian Doppler Laboratory for High Performance TCAD and the Institute for Microelectronics, Vito Simonka, Xaver Klemenschits, Georgios Diamantopoulos, Lukas Gnam, Alexander Scharinger, and Francio Rodrigues. The discussions with them during lunch widened my view on almost all topics concerning mankind.

Finally, I would like to thank my parents and siblings for their unconditional support during my journey, their nice words and encouragement on my goals. Only their altruistic deeds enabled me to pursue my career as a scientist.

Contents

List of Acronyms

2D	two-dimensional	4
3D	three-dimensional	1
AMR	adaptive mesh refinement	4
API	application programming interface	26
CFL	Courant-Friedrichs-Lewy	41
CMOS	complementary metal-oxide semiconductor	67
CPU	central processing unit	5
CSG	constructive solid geometry	7
EQ	exchange queue	66
FEM	finite element method	15
FIM	fast iterative method	80
FMM	fast marching method	11
FSM	fast sweeping method	80
FVM	finite volume method	15
HRLE	hierachical run length encoding	45
IBE	ion beam etching	47
LSM	locked sweeping method	82
MOSFET	metal-oxide semiconductor field effect transistor	7
MTJ	magnetic tunnel junction	67
NUMA	non-uniform memory access	24
PDE	partial differential equation	3
RAM	random access memory	67
RK	Runge-Kutta	40
STT	spin-transfer torque	67
STT-MRAM	spin-transfer torque magnetoresistive random access memory	47
TCAD	technology computer-aided design	1
TMR	tunneling magnetoresistance	67
TVD	total variation diminishing	40
WENO	weighted essentially non-oscillatory	16
WQ	work queue	59

Chapter 1

Introduction

The manufacturing of semiconductor devices takes place in reactor chambers where the environment (pressure, temperature, chemicals) is strictly controlled, allowing precise fabrication of device-relevant structures in the nanometer range. The devices are fabricated starting from a plane wafer which typically is a thin slice of monocrystalline silicon. Ongoing advances of semiconductor device design have lead to complex three-dimensional (3D) designs and expensive manufacturing steps. This also manifests in shrinking admissible tolerances and increased development cost.

The high development costs and expensive experiments in the development cycle iterating between design and verification of new semiconductor devices are pushing the usage of predictive simulations for the manufacturing and operation of semiconductor devices. Those simulations, known as technology computer-aided design (TCAD) [1, 2] support the fabrication development by reducing the need for time-consuming and expensive experiments (costly materials and equipment). This accelerates the development cycle and reduces development costs.

TCAD simulations are divided into three categories: *process* TCAD, *device* TCAD, and *circuit* TCAD.

Process TCAD models the manufacturing processes of a semiconductor device by simulating processes which change the structure and/or topography (material layout) of the wafer, thus forming the individual devices [3]. The manufacturing processes include:

- etching (material is removed)
- deposition (material is added)
- oxidation (materials are oxidized, turning them into an oxide, which often have properties of insulators)
- ion implantation (dopants/impurities are implanted to change electrical material properties)
- annealing and diffusion (repairing crystal lattice defects and relocating dopants)

Device TCAD on the other hand uses the final generated structure to simulate the operation of a semiconductor device and calculate the electrical properties [4].

Circuit TCAD uses the device characteristics (electrical properties) provided by device TCAD to simulate the interaction of several devices, i.e., a simulation of an integrated circuit.

The context of this work is process TCAD simulation, which is further subdivided into two subcategories: *Reactor scale* and *feature scale* simulations.

Reactor vs. Feature Scale Simulations

Reactor scale simulations investigate the manufacturing process on a macroscopic scale, i.e., how chemicals (liquids or gases) enter the reactor chamber and how to ensure an equal distribution onto a wafer [5, 6]. Typically, all to-be-built devices on a wafer are processed simultaneously together, e.g., a wafer is exposed to oxygen at high temperatures, which results in oxidizing the surface. Reactor scale simulations are useful to model and analyze processing variations in different regions of the wafer. Minimizing those variations is essential to ensure a high device yield[1]. They are also used to predict variations between devices located on different wafers, in case several wafers are put into the reactor chamber at the same time, or in case processing several wafers subsequently before the reactor chamber is reset (extensively cleaned) to its initial condition.

Feature scale simulations, the focus of this work, describe the actual structural (potentially topographical) changes of the wafer. Feature scale simulations operate on the scale of a single device, or even on a sub-part of a device. The size of a device ranges from nanometers for logic devices to millimeters for power devices. Figure 1.1 shows a schematic of a typical simulation domain for a feature scale simulation, which represents a small part of a wafer with appropriate boundary conditions for the structure and other parameters, e.g., temperature or particle flux. The boundary conditions are potentially set by a previous reactor scale simulation.

The core of a feature scale simulation is the process model. The process model is the (simplified) physical description of the modeled manufacturing process. For example a process model for an etching process determines the etch rates for each material: The process model of a silicon oxidation process determines the rate at which silicon is transformed into silicon dioxide. Thus it is of utmost importance for feature scale simulations to have a high accuracy representation of the material regions (volumes or structure) of the device. Especially, the boundaries of material regions are important, because those are the areas which are directly affected by the process model, e.g., exposed silicon surfaces which directly interact with the available oxygen in the reactor chamber.

The following paragraphs introduce the numerical methods and algorithms[2] used in the context of this thesis to track the topographical changes of these material regions.

[1] Percentage of devices fulfilling the admissible tolerances in relation to all devices on a wafer.
[2] An algorithm is a finite sequence of instructions used to solve a problem, i.e., complete a task.

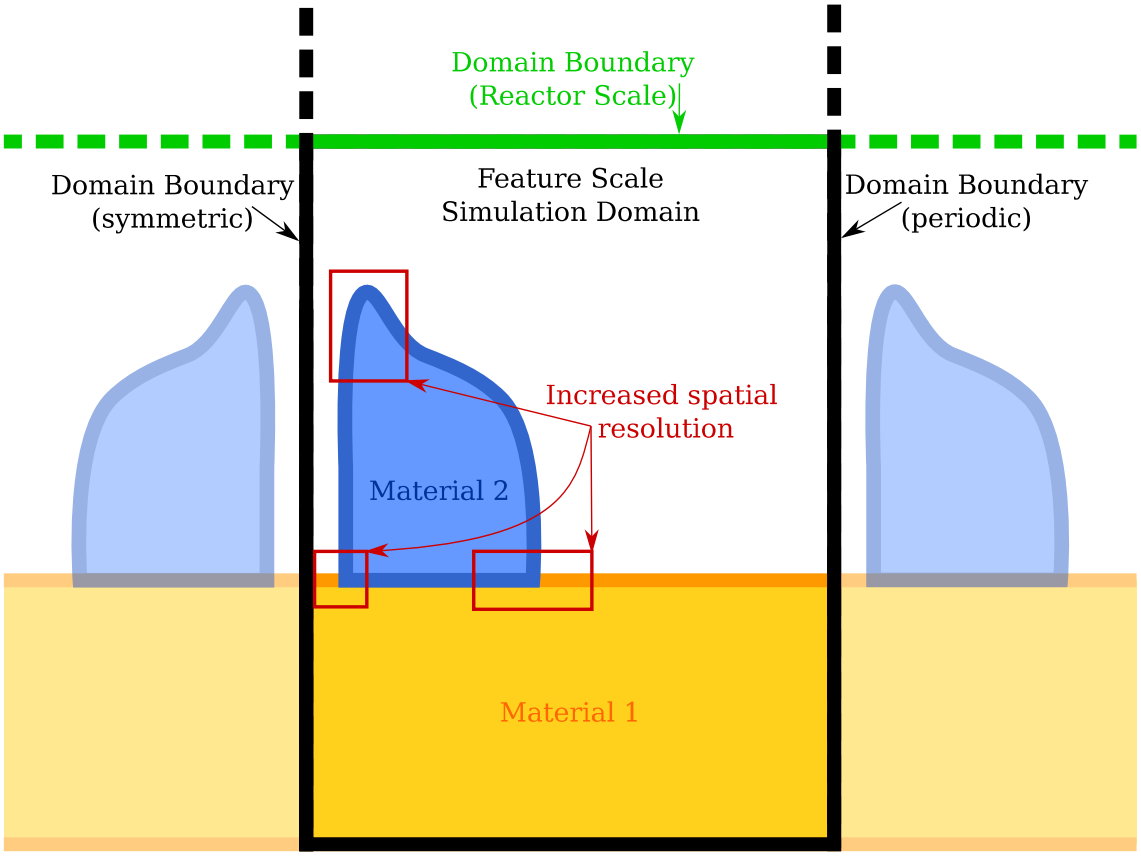


Figure 1.1: Simulation domain for a typical process TCAD simulation on the feature scale. The domain boundaries on the sides are often chosen as symmetric or periodic. The boundary on the top of the simulation domain (Reactor Scale) couples the simulation domain to the reactor scale. The level-set method tracks only the boundaries of material regions (darker outline of material regions). Only parts of the simulation domain (red rectangles) employ a fine spatial resolution.

Level-Set Method

The level-set method (cf. Chapter 4) tracks the material regions via the so-called level-set function [7]. The level-set function implicitly represents the boundary of a material region (interface between two materials regions, e.g., the outline of Material 1 and Material 2 in Figure 1.1) as the zero-level-set of the level-set function. By changing the zero-level-set of the level-set function the interface position is modified (material regions grow, shrink or deform). The implicit representation enables a robust handling of topological changes. The interface changes are prescribed by the process model which is typically defined via a partial differential equation (PDE), the level-set equation. The description of complex material regions using a level-set function is typically done on a discretized simulation domain. The considered rectilinear simulation domain is often discretized by a Cartesian grid, because it is convenient as derivatives used in process models as well as in the interface propagation can straight-forwardly be approximated by finite differences.

The level-set method is able to only track the boundary of material regions, which is computationally efficient compared to tracking the full volumes of material regions.

In this case only a narrow-band of grid points adjacent to the interface is used for interface tracking, giving the name *narrow-band* level-set method [8]. Those grid points have to be stored efficiently, which is achieved using a sparse volume data representation, e.g., [9, 10]. Additionally, the level-set method allows straightforward extension of a simulation to higher dimensions, i.e., switching from two-dimensional (2D) to 3D simulations. The same algorithms are employed but need to process an additional coordinate. The application of the level-set method in process TCAD simulations is a well-established method for feature scale simulations, e.g., [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21].

The above discussed benefits have led to a widespread use of the level-set method (spanning several research disciplines) for tracking moving interfaces. Application areas include computational fluid dynamics [22, 23, 24, 25, 26, 27, 28, 29, 30, 31], shape optimization [32, 33, 34, 35, 36, 37], computer graphics [38, 39, 40, 41, 42], image processing [43, 44, 45], and computational biophysics [46, 47, 48].

Explicit interface[3] tracking approaches in the context of process TCAD simulations are described in [49, 50, 51]. Currently, explicit approaches are not further pursued, because of issues with respect to rarefaction or accumulation of polygons and with self-intersection of polygons representing the interface, which is especially a challenge for 3D simulations.

The discussion continues with a more detailed view on the discretization scheme used in this work.

Hierarchical Grids

The goal to achieve high accuracy in level-set process TCAD simulations forced the usage of fine spatial discretizations. However, fine spatial resolutions, especially for engineering-relevant 3D simulations, significantly increase the memory requirements and the run-time. On a Cartesian grid the run-time scales with the third power of the spatial discretization for a 3D simulation, easily and thus gets impractical.

A strategy to reduce the run-time is to employ high spatial resolutions only in some regions of the simulation domain. This strategy is known as adaptive mesh refinement (AMR). There are various approaches to AMR, where in this work the focus is on hierarchical grids. A hierarchical grid consists of several nested rectangular domains (blocks) each using a Cartesian discretization with varying spatial resolutions. Their possible placement in also indicated in Figure 1.1. The details of the used hierarchical grid are presented in Chapter 2. The approach using hierarchical grids based on Cartesian grids is convenient, because the same numerical schemes to approximate derivatives as on a Cartesian grid can be employed.

Time Stepping

The time evolution of the material regions is typically described by a PDE.

[3] The interface is represented as a set of polygons, segments or triangles, depending on the spatial dimensions.

Typically, the PDE is discretized in time and advanced in time steps until the final simulation time is reached. Combined with the spatial discretization using hierarchical grids the computational steps[4] of the level-set method in every time step are:

- *Interface Velocity*: Coupling the process (physical accurate) model, describing the deformation of the interface, to the level-set representation of the material regions.
- *Velocity Extension*: Extending a velocity field from an interface (i.e., surface of a material region) to the entire computational domain. In particular, velocity refers to the physically determined velocity prescribing the movement of the interface.
- *Advection*: Using the previously extended velocity field to solve the advection equation of the interface, actually changing the interface position.
- *Re-Distancing*: Re-normalizing the signed-distance field to an interface. This step is essential for a robust interface representation and a geometric interpretation of the level-set function away from the interface.
- *Re-Gridding*: Adapting the hierarchical grid structure to fit the blocks of a fine spatial resolution to the deformed and displaced interfaces describing the material regions.

The computational steps (and algorithms used to solve the computational steps) of the level-set method are presented and discussed further in Chapter 4.

In addition to AMR this work also uses parallelization approaches to accelerate some key algorithms for process TCAD simulations. Therefore, the following section provides a motivation regarding the importance of parallelization.

Parallelization

While early research on parallel computations dates back to the 1960s, starting in the early 2000s the number of cores on a central processing unit (CPU) increased as the *frequency wall* was hit[5], limiting the exponential growth of serial performance (cf. Figure 1.2). This trend continues until today where top CPUs can offer over 100 logical cores, e.g., an AMD Ryzen Threadripper 3990X offers 128 logical cores [52], or an Intel Xeon Platinum 9282 has 112 logical cores [53]. Thus, to utilize the gain in available computing power the developed algorithms must be parallel: This work focuses on shared-memory parallel approaches to utilize the high degree of parallelism provided by the discussed high core counts of modern CPUs. Computer programs[6] implementing parallel algorithms on shared-memory systems use threads[7] to distribute their computations among the available CPU cores.

[4] A computational step is a task for which one ore more algorithms may be employed, e.g., *Sorting* is a computational step: Any sorting algorithm is a valid choice to complete the task.

[5] The frequency of CPUs could not be reliably increased further, due to power and heat limitations.

[6] A computer program is a set of instructions describing a specific implementation of an algorithm to a computer.

[7] A thread is the smallest set of instructions managed and independently scheduled by the operating system for execution on a CPU core. Each thread has access to the entire system memory.

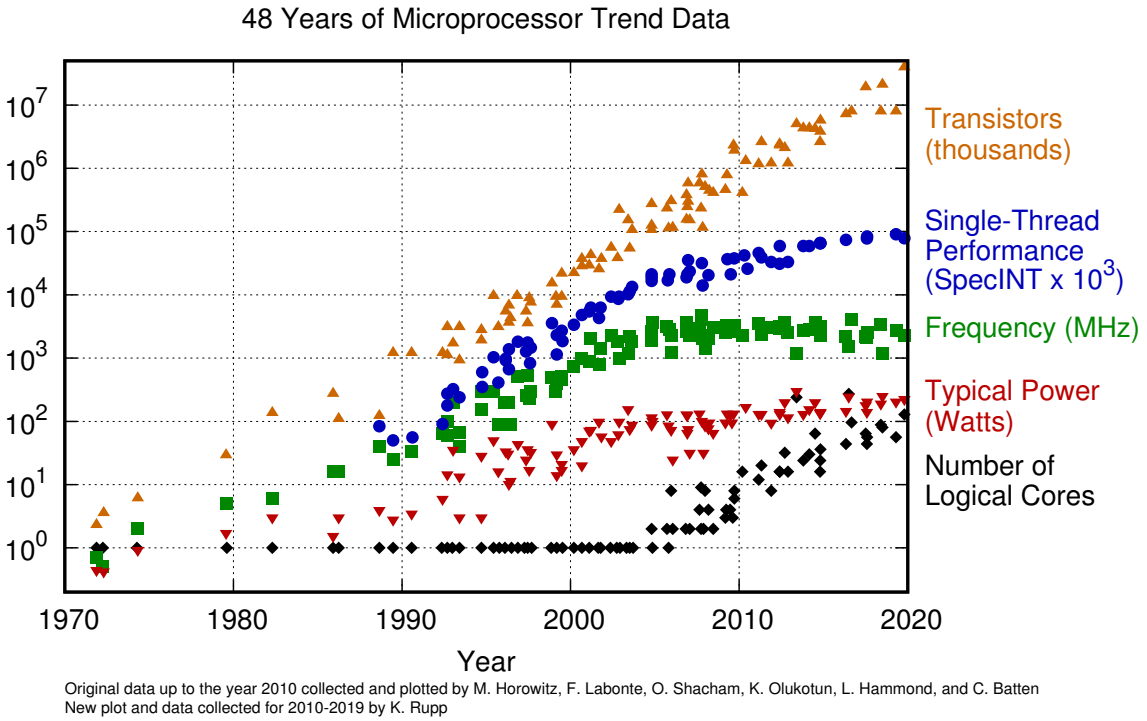


Figure 1.2: Key data for central processing units from the beginning until 2020. Reprinted from https://github.com/karlrupp/microprocessor-trend-data, © CC 4.0, http://creativecommons.org/licenses/by/4.0/.

Hierarchical grids provide an inherent potential for parallelism: Parallelization of algorithms on hierarchical grids is often implemented so that on each block the algorithm is performed independently in parallel. Obviously, this requires a synchronization step to align the parallel calculated results. However, this approach delivers only good parallel efficiency, if multiple blocks per thread are available, allowing for load-balancing to counter imbalances imposed by strongly varying block sizes and numbers. Load imbalances happen, if some threads have completed their task (share of computations), but have to wait for other threads to finish their task before they may proceed with their next task (synchronization barrier).

A concrete feature scale simulation example is presented in the next section to show the capability of current process TCAD simulations and to establish a baseline for the computational performance via benchmarking the simulation.

1.1 Motivational Example: Thermal Oxidation

Thermal oxidation is a fundamental processing step in the manufacturing of semiconductor devices [54]. It is used to create an insulating or protective layer of silicon dioxide (SiO_2), by exposing a silicon (Si) surface to oxygen gas (O_2) or water vapor (H_2O) at temperature ranges from 800 °C to 1400 °C in an oxidation reactor chamber (furnace). The oxidation furnace is usually operated using heating coils in combination with a temperature measurement and control system.

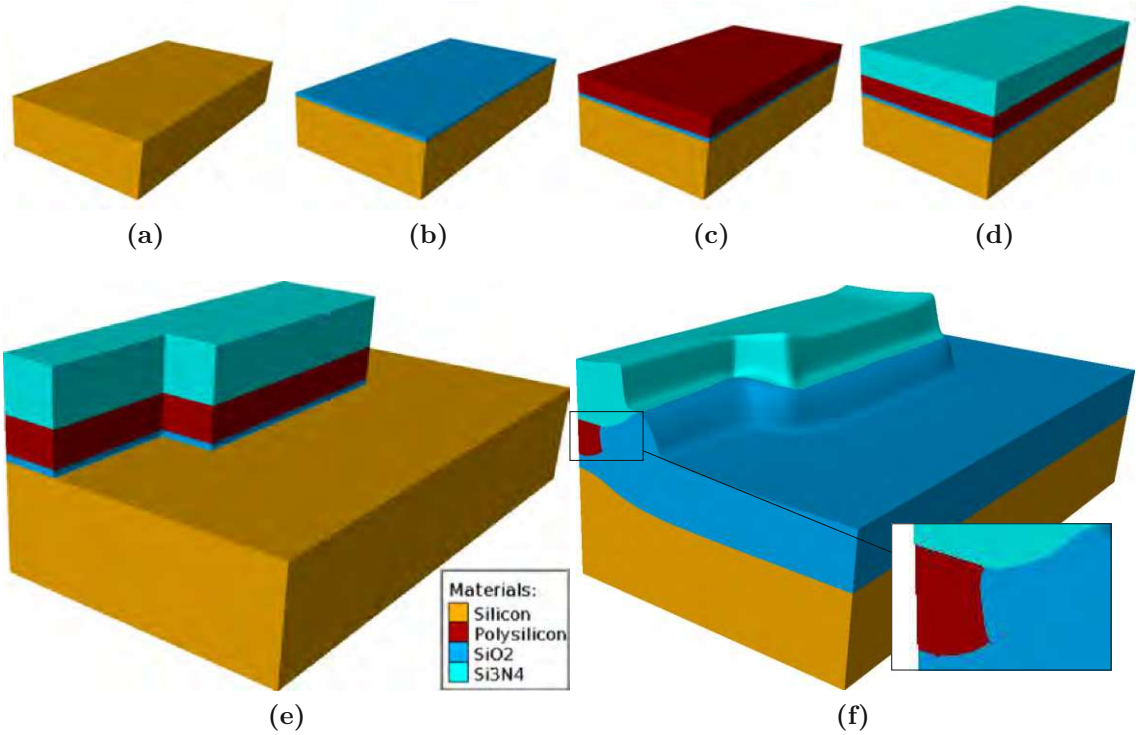


Figure 1.3: Process steps from bulk silicon (a) to the initial structure (e) for the thermal oxidation process presented in Section 1.1 using geometrical process models. The thermal oxidation process itself is simulated using a physically accurate process model leading to the final structure (f). The magnified inset shows the deformation of the materials on top of the oxide. Adapted with permission from Quell *et al.*, *IEEE Transactions on Electron Devices* 68.11 (2021), pp. 5430–5437 [55], © 2021 IEEE.

Thermal oxidation is also used to grow gate oxides insulating the gate from the source and drain of metal-oxide semiconductor field effect transistors (MOSFETs), which are among the basic building blocks for microelectronic circuits in the semiconductor industry.

For the thermal oxidation process a rectilinear simulation domain with symmetric (reflective) boundary conditions is considered. Figure 1.3 shows all process steps of a thermal oxidation simulation example. The initial material layout of the thermal oxidation simulation example is created using constructive solid geometry (CSG) operations. On the bulk silicon (cf. Figure 1.3a) several materials are deposited (in layers covering the whole simulation domain) (cf. Figure 1.3b – Figure 1.3d) and then parts of the deposited materials are geometrically etched (one could consider this as a very rudimentary process model based on simplified physics) using an L-shaped mask (cf. Figure 1.3e). In particular, the deposited material layers from bottom to top (in order of deposition) are silicon dioxide[8] (SiO_2), polysilicon (Polysilicon), and silicon nitride (Si_3N_4). This represents the starting point of the physically accurate simulated thermal oxidation process.

[8] In semiconductor manufacturing this is often called just *oxide*, because silicon is the predominantly oxidized material.

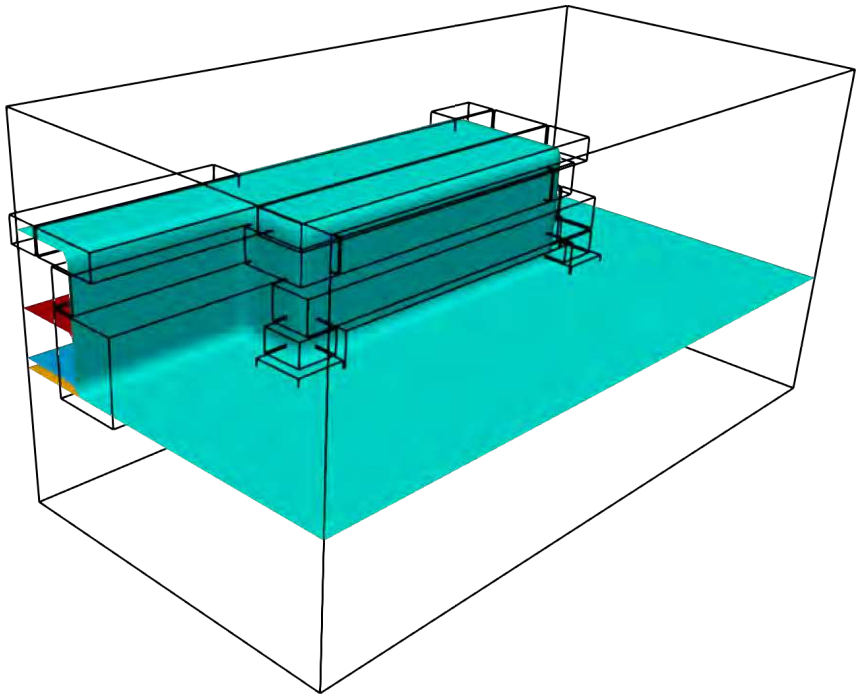


Figure 1.4: Representation of the initial material layout of the thermal oxidation example in the level-set method (for each material a dedicated level-set function is used). The computational domain is shown by the outermost black box. The placement of the refined grid regions (blocks) of the hierarchical grid are shown by the black boxes located around the edges and corners of the level-sets.

Figure 1.3e and Figure 1.3f show the material regions before and after the 15 min thermal oxidation process at 1000 °C. The silicon and polysilicon material regions shrink, because they are turned into oxide. The process expands the created oxide compared to the previously present material (silicon and polysilicon) and deforms the silicon nitride on top[9].

The investigation continues on the level-set method specifics of the example thermal oxidation simulation such as the material representation and the structure of hierarchical grids.

Level-Set Method on Hierarchical Grids

Figure 1.4 shows the interfaces represented by the four level-set functions (one for each material) used for the initial material layout of the thermal oxidation simulation shown in Figure 1.3e. The interfaces do not enclose a material region directly, but are selected so that they avoid material region overlaps and enable best computational performance. The level-set used to represent the silicon nitride material region is identical to the surface of the structure (surface visible form the outside). This is beneficial to the simulation, because the surface of the structure is the area of exchange of the structure with the reactants from the reactor chamber.

[9]The characteristic emergence of features resembling a *bird's beak* [51, 56] is observed between the silicon dioxide and the polysilicon.

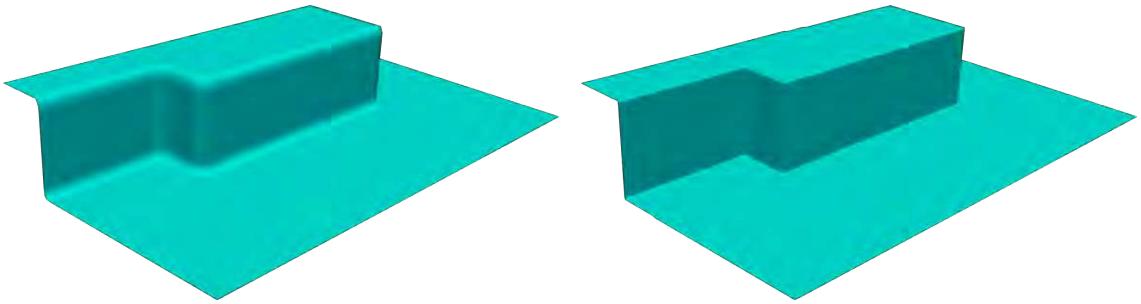


Figure 1.5: Comparison of the level-set representation of the silicon nitride of the thermal oxidation simulation using two different spatial resolutions, on the left using a coarse discretization and on the right using a fine one, which uses four times as many grid points in each spatial dimension. As expected, the corners are significantly better resolved by a higher spatial discretization, underlining the importance of feature resolution for simulation accuracy.

Additionally, Figure 1.4 also shows the layout (placement and size) of the blocks used on the hierarchical grid. Those blocks (rectangular domains) are shown by their outline colored in black. The outermost block covers the full simulation domain with a low spatial resolution, its boundaries are identical to the boundaries of the simulation domain. The remaining blocks which employ a higher spatial resolution are located around corners and edges of material regions (interfaces).

Figure 1.5 shows the zero-level-set of the level-set function used for representing the silicon nitride for a low (coarse) spatial resolution and a four times increased (finer) spatial resolution. The difference between those two spatial resolutions is striking at the edges and corners. The level-set inherent rounding of analytic sharp corners is directly related to the spatial resolution, affecting one to two grid points around the corner. Thus if only those corners are resolved by a grid with a higher finer spatial resolution the same accuracy for the interface is achieved, favoring an approach based on hierarchical grids (cf. Figure 1.6).

In the thermal oxidation example comparing the approach based on hierarchical grids (using 664 704 grid points) to a reference discretization using a single Cartesian grid with an uniform high spatial resolution (using 8 192 000 grid points) shows that the number of grid points is reduced by a factor of 12.

A key aspect of a hierarchical grid is to utilize the gained accuracy on the finer spatially resolved regions also on the coarse grid. Typically, this is done via an interpolation of the grid points of the coarse grid. The gained accuracy is especially beneficial for features for which the coarse grid is not able to resolve them correctly. Such features are in particular corners (cf. Figure 1.6) and thin trenches. The problem is that a straightforward interpolation does not affect grid points which are not covered by finer spatial resolutions. The coupling of the grids (the interpolation) is mainly performed in the *Re-Distancing* step, therefore, an improvement in this computational step has the greatest impact on the overall accuracy (cf. Section 4.2.6).

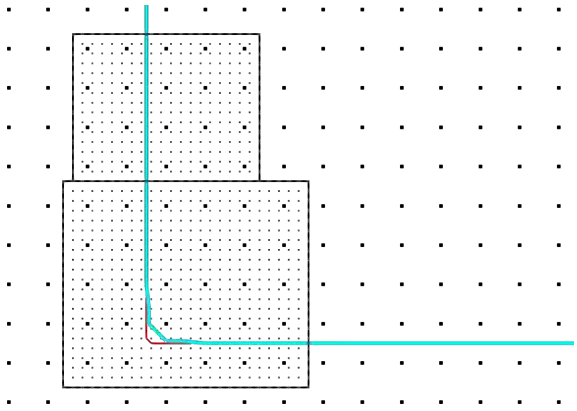


Figure 1.6: Slice of the interface shown in Figure 1.5 highlighting a corner (edge). There are two blocks where a four times increased spatial resolution is employed. The interface (turquoise line) extracted solely from the coarse grid (black points) shows a rounded corner. The interface (red line) extracted from the finer grid (gray points) shows that the rounding is directly proportional to the spatial resolution. In areas where there is no curvature the interfaces of both spatial resolutions match.

Benchmark Baseline

To analyze where the computational bottlenecks of the level-set method are, the thermal oxidation process is benchmarked using a reference simulator: Victory Process (cf. Section 4.3). The compute system used for the benchmark is a representative industrial compute server (ICS) (cf. Section 3.2).

The measured run-times for the computational steps of the level-set method (ordered chronologically) are shown in Figure 1.7 for every time step of the simulation separately. There are a total of 27 time steps involved in the considered oxidation simulation. The run-time for each time step increases during the simulation, because the regions where a high spatial discretization is required to sustain an accurate simulation increase in size, thus more grid points are present which in turn require more computations.

Each of the computational steps contributes to the total run-time. The three computational steps, which are the main contributors to the total run-time, are:

1. *Advection* (35.9 %),
2. *Re-Distancing* (30.7 %), and
3. *Velocity Extension* (22.1 %).

The biggest contributor – the *Advection* – is out of scope of the conducted research, because it has already been extensively studied [57, 58, 59]. Typical employed parallel algorithms allow for independent computations for each grid point enabling a straightforward parallelization.

This thesis, however, focuses on the computational steps *Velocity Extension* and *Re-Distancing*, because as can clearly be seen they are key contributors to the overall simulation run-time. Together they are responsible for more than half of the serial run-time, demanding efficient parallelization approaches to mitigate the serious performance hit.

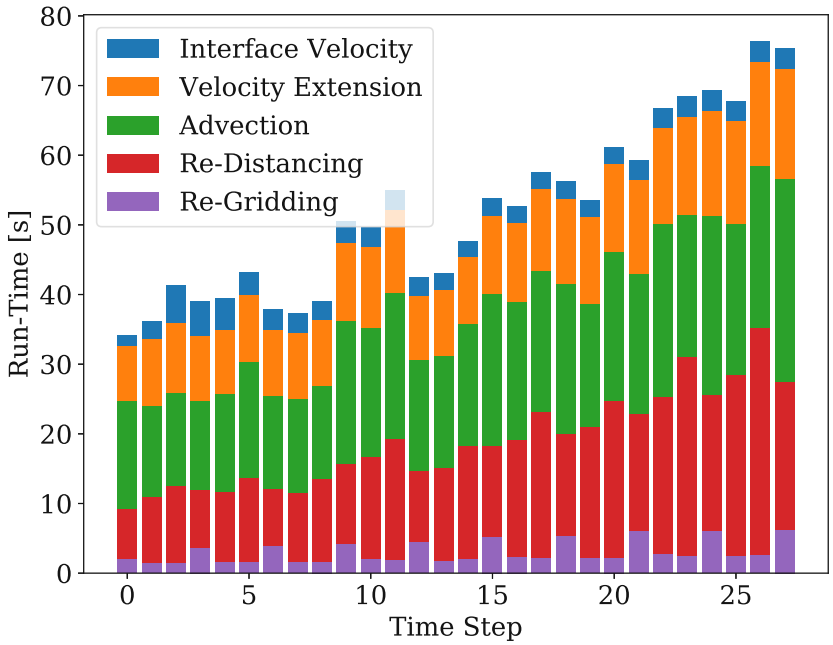


Figure 1.7: Serial run-time of the considered computational steps of the level-set based simulation for the thermal oxidation process shown for individual time steps, measured on the ICS.

The task for both computational steps is similar: Each has to extend a given field (either a velocity field or a signed-distance field) from the interface to the computational domain. However, the difference is whether the to-be-extended field influences the control flow of the extension, i.e., the order in which the values for the grid points are computed. In case of *Velocity Extension* the to-be-extended field (the interface velocity) does not influence the control flow, i.e., for different velocity fields at the interface, the order of the computations stays the same (the extended velocity values are obviously different). However, in case of *Re-Distancing* the to-be-extended field (the signed-distance[10]) does influence the control flow. The algorithmic solution for both computational steps is based on the fast marching method (FMM) [60].

The FMM finds wide spread use and parallel solution approaches are available (in cases where the control flow is influenced by the field) [61]. These approaches typically consider only Cartesian grids. A parallel algorithm considering hierarchical grids was developed albeit offering limited scalability due to load-balancing issues rooted in a non-optimal dependency on the number and dimension variations of hierarchical grid blocks [62]. For example using 10 threads for the previously shown thermal oxidation example leads to load-imbalances, because the hierarchical grid consists (depending on the time step) of 17 blocks only: In this case, three threads would only process a single block, preventing the compensation of varying run-times between blocks.

Hierarchical grids require data exchange between levels of different spatial resolution. Especially, for *Re-Distancing* high accuracy schemes are desired to fully utilize the advantages of hierarchical grids.

[10]The signed-distance field is typically zero at the interface.

For the *Velocity Extension* the same algorithm (the FMM) can theoretically be applied. The nature of an unchanged control flow of computations for the FMM should allow for higher optimizations of the computations. The dependencies of the computations could be determined beforehand, enabling shorter run-times and yielding opportunities for advanced parallelization.

1.2 Research Goals

The main goal of the conducted research is to reduce the turnaround time of 3D feature scale process TCAD simulations using the level-set method on hierarchical grids. The focus is to accelerate two of the key computational steps of the level-set method: 1) *Velocity Extension* and 2) *Re-Distancing*. These computational steps are selected because they significantly contribute to the overall run-time as previously discussed (cf. Figure 1.7).

The acceleration should be achieved by first parallelizing the *Velocity Extension* on a single Cartesian grid exploiting the computation order and then tailoring the developed parallel algorithm to hierarchical grids.

For the *Re-Distancing* an advanced block decomposition has to be developed for the FMM, which in principle allows for efficient parallelization on hierarchical grids by putting emphasis on load-balancing.

A further important goal is to increase the numerical accuracy of *Re-Distancing* for hierarchical grids, especially in cases where only the higher grid resolutions enable the representation of a feature like a thin trench. The computational overhead shall be minimized.

Research Setting

The research presented in this work was conducted within the scope of the Christian Doppler Laboratory for High Performance TCAD. The Christian Doppler Association funds cooperations between companies and research institutions pursuing application-orientated basic research. In this case, the research was lead by Josef Weinbub and involved the Institute for Microelectronics at the TU Wien and Silvaco Inc., a company developing and providing electronic device automation and TCAD software tools.

1.3 Outline

Chapter 2 presents an overview of spatial discretization methods, adaptive mesh refinement, and the implementation of the hierarchical grid used in the reference simulation software.
Chapter 3 presents an overview of the terms used in parallelization and gives context with respect to the compute systems used for evaluating the performance of the algorithmic developments.

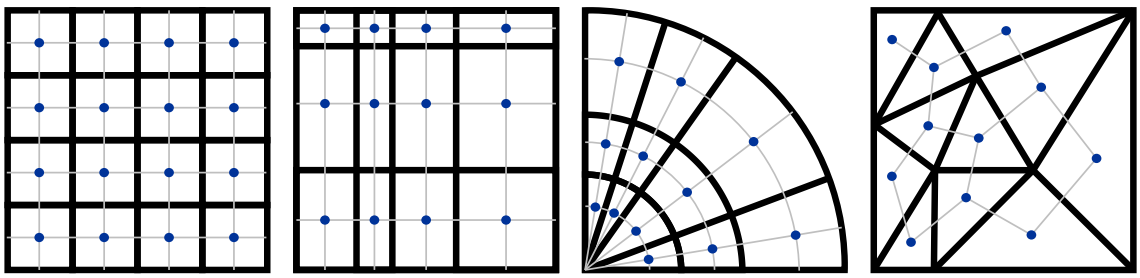
Chapter 4 portrays the level-set method with a special focus on process TCAD simulations. The mathematical background for the level-set method is given and all computational steps are discussed which have already been shown in Figure 1.7. The numerical implementation of the computational steps in the reference simulator, i.e., the reference simulation workflow, is discussed. Available simulation software for process TCAD simulations is listed.
Chapter 5 presents the newly developed parallelized velocity extension algorithm, which is first introduced for operating on a single Cartesian grid. Subsequently, an extension tailored towards the use on hierarchical grids is presented. The FMM, because it is the foundation of the improved computational steps, is presented. The parallelized velocity extension algorithm is analyzed and evaluated based on two representative process TCAD simulation examples for scalar and vector velocity fields, discussing run-time performance metrics.
Chapter 6 proposes an advanced parallelization algorithm for the FMM used in the *Re-Distancing* step. The key contribution is a novel domain decomposition approach to enable better load-balancing during the execution of the FMM, resulting in a shorter run-time for high core count CPUs. The granularity of the decomposition and the frequency of synchronizations is particularly focused on in the analysis. The parallel performance is evaluated based on representative interfaces (level-set functions) taken from typical process TCAD simulations.
Chapter 7 presents an algorithm to increase the accuracy of the signed-distance field on coarser levels of the hierarchical grid, by using a bottom-up correction algorithm. The algorithm is evaluated on generic test cases resembling geometries occurring in process TCAD simulations. This enables to compute the exact solution as a reference solution and, therefore, an accurate comparison of the corrected signed-distance field to the exact solution is possible.
In the last chapter, Chapter 8 the key findings of this thesis are summarized, the motivational example is revisited, and new ideas are proposed for future research.

Chapter 2

Hierarchical Grids

Numerical treatment of PDEs, e.g., the level-set equation, prescribing the time evolution of the interfaces, typically requires a discretization of the domain, the computational grid. The two approaches to discretization are structured (regular) grids and unstructured (irregular) grids.

Figure 2.1 shows some examples for structured and unstructured grids in two spatial dimensions. The representative for a structured grid is the Cartesian grid (cf. Figure 2.1a), where the grid points are located on a regular lattice and the cells (for a formal definition, see Section 2.1) are squares or cubes depending on the number of spatial dimensions. The connectivity of grid points is implicitly defined via their indexes on the regular lattice. The width along a spatial direction of such a cell is called grid resolution and is identical in all spatial dimensions. A rectilinear grid (cf. Figure 2.1b) allows different distances between grid points, thus the cells are rectangles or rectangular cuboids, in two dimensions and three dimensions, respectively. Rectilinear grids allow for a limited spatial adaptivity, by adapting the distance between grid points locally. The grid resolution is typically stored as a dedicated array of the distances between the grid points along each spatial dimension. A curvilinear grid (cf. Figure 2.1c) is often employed, if the PDE is formulated in curvilinear coordinates, i.e., in spherical or cylindrical coordinates, allowing for the spatial discretization to fit the PDE.



(a) Cartesian grid **(b)** Rectilinear grid **(c)** Curvilinear grid **(d)** Irregular grid

Figure 2.1: Four domains discretized by different grids, where the outline of the cells are colored in black, the corresponding grid point colored blue is located in the center of the cell, and the grid lines connecting the grid points are colored gray: **(a)**-**(c)** Examples of structured grids and **(d)** example of an unstructured grid.

Cartesian grids and rectilinear grids enable a straightforward computation of derivatives using finite difference methods [63, 64]. The main disadvantage of structured grids is that the discretized domain has to be regular, e.g., a rectangle or a cylinder. Nevertheless, for process TCAD simulations on the feature scale, where typically a rectangular domain of the wafer is considered, this is no disadvantage.

In contrast, unstructured grids (cf. Figure 2.1d) are not restricted to regular domains as their cells are typically polytopes, e.g., triangles or tetrahedra. The usage of polytopes allows for the discretization of arbitrarily shaped domains. The drawback is the irregular connectivity of the grid points which has to be stored explicitly. PDEs discretized on unstructured grids are typically solved using the finite element method (FEM) [65] or the finite volume method (FVM) [66], which involve explicitly stored large sparse matrices in the solving procedure.

This thesis does not investigate the level-set method applied on unstructured grids and, therefore, the reader is referred to [67, 68, 69, 70, 71, 72] for details on this matter.

The discussion continues with the discretization of a domain using a Cartesian grid.

2.1 Discretization

Let $\mathbb{R}^3$ be discretized by a Cartesian grid using the spatial resolution Δ_x along the x-axis, Δ_y along the y-axis, and Δ_z along the z-axis. The nodes $(i, j, k) \in \mathbb{Z}^3$ (triplets of indices) index all the grid points $(i\,\Delta_x, j\,\Delta_y, k\,\Delta_z)$. This global indexing enables a unique identification of every grid point. The volume surrounding a grid point belonging to the node (i, j, k)

$$[(i-0.5)\Delta_x, (i+0.5)\Delta_x] \times [(j-0.5)\Delta_y, (j+0.5)\Delta_y] \times [(k-0.5)\Delta_z, (k+0.5)\Delta_z]$$

is called cell, which is the smallest unit in the computational domain. For a given function $\Phi(x, y, z)$ defined on the domain, the function Φ (discretized on the grid) is denoted by $\Phi^{ijk} = \Phi\left(i\,\Delta_x, j\,\Delta_y, k\,\Delta_z\right)$ for all grid points. Information associated with a grid point is referred to as data, e.g., the discretized value of a function. Computations typically involve more than a single grid point; consider, for instance, the approximation of a derivative as discussed below. The collection of all grid points necessary for such a computation is called stencil which typically involves the neighboring grid points. The widely-known seven-point stencil in three dimensions for a node (i, j, k) includes the node itself and all direct neighbors, i.e., nodes which indices differs by at most one:

$$\begin{aligned} &(i-1, j, k), (i+1, j, k), \\ &(i, j-1, k), (i, j+1, k), \\ &(i, j, k-1), (i, j, k+1). \end{aligned} \tag{2.1}$$

The computation of derivatives on a computational grid is essential in the context of solving PDEs. For first-order approximations finite difference schemes are particularly convenient, as shown in the following. The first-order accurate forward difference along the x-axis,

$$\frac{\partial \Phi}{\partial x} = \lim_{\Delta_x \to 0} \frac{\Phi(x+\Delta_x) - \Phi(x)}{\Delta_x} \approx \frac{\Phi(x+\Delta_x) - \Phi(x)}{\Delta_x} = \frac{\Phi^{i+1jk} - \Phi^{ijk}}{\Delta_x}, \tag{2.2}$$

is referred to as $D^{+x}_{ijk}\Phi$. Similarly, the first-order accurate backward difference along the x-axis,

$$\frac{\partial \Phi}{\partial x} \approx \frac{\Phi^{ijk} - \Phi^{i-1jk}}{\Delta_x}, \tag{2.3}$$

is referred to as $D^{-x}_{ijk}\Phi$. Consequently, the second-order accurate central difference along the x-axis,

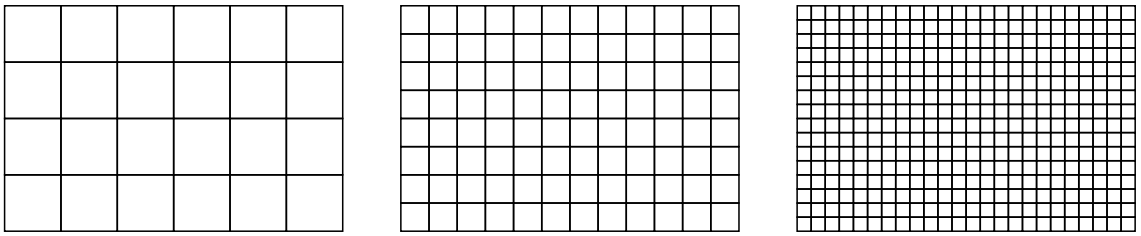
$$\frac{\partial \Phi}{\partial x} \approx \frac{\Phi^{i+1jk} - \Phi^{i-1jk}}{2\Delta_x}, \tag{2.4}$$

is referred to as $D^{x}_{ijk}\Phi$. The derivatives along other spatial dimensions such as y-axis and z-axis are defined analogously.

Higher order approximations are possible (if Φ is smooth enough) by using higher order finite difference schemes which, however, require a bigger stencil. In the here considered case of the level-set method so-called weighted essentially non-oscillatory (WENO) schemes are often employed [73, 74, 75, 76], if higher accuracies are desired. WENO schemes compute the derivative on several sub-stencils of a large stencil. Subsequently, the derivatives computed on the sub-stencils are combined through a convex combination to minimize spurious oscillations. Spurious oscillations occur, if the derived function lacks smoothness (differentiability) in the stencil considered for the finite difference computation. Level-set functions are not smooth around regions where the level-set function describes corners or edges of interfaces. This is an inherent feature, because corners are exactly those points where the level-set function is not differentiable. Edges and corners are typically present in interfaces used to describe structures considered in process TCAD simulations, thus higher order schemes are avoided, because they require more computational power and do not yield increased accuracy around corners, where it is most needed. Therefore, this thesis considers only first-order schemes.

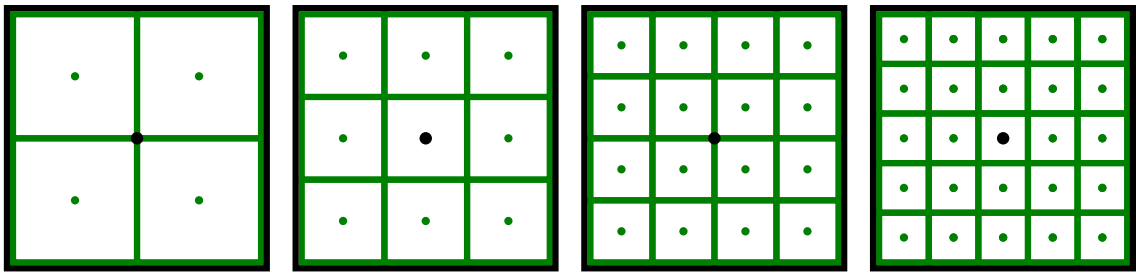
If a higher resolution is required to better resolve a corner in a structured grid, the entire grid has to be resolved with the desired higher spatial resolution. In case of 3D domains this quickly becomes unfeasible, because of high memory requirements to store all grid points and the wasted computation power in irrelevant regions. As hinted previously, local refinement, where only parts of the domain are resolved with a higher spatial resolution, are a viable solution, to enable high spatial resolution around corners, while keeping the overall computational effort feasible.

The discussion continues with approaches to local refinement of the spatial resolution of a Cartesian grid to enable efficient application of computation power to regions which have the highest potential for an overall increased solution accuracy.



(a) *Level 0* **(b)** *Level 1* **(c)** *Level 2*

Figure 2.2: A rectangular computational domain is discretized using different spatial resolutions on different levels of the hierarchical grid.



(a) Ref. ratio 2 **(b)** Ref. ratio 3 **(c)** Ref. ratio 4 **(d)** Ref. ratio 5

Figure 2.3: A black cell with its corresponding grid point in the center (black point) is refined using a refinement ratio of 2, 3, 4, and 5 (green cells; a-d). An uneven refinement ratio aligns the grid point (center of a cell) from the coarse cell to the grid point of the central refined cell.

2.2 Refinement

Techniques to locally increase the spatial discretization for structured grids date back to [77] and are referred to as adaptive mesh[1] refinement (AMR). AMR was first used to solve hyperbolic PDEs [77, 78, 79] in the context of compressible flows. In the context of process TCAD simulations, fundamental basics are provided in [57].

In AMR the computational domain is discretized by several layers (levels) of Cartesian grids. These grid layers cover the entire discretized domain, with different spatial resolutions, hence the name hierarchical grid. The coarsest grid is referred to as *Level 0*, whilst consecutively spatially finer resolving grids have a higher level, e.g., *Level 1* and *Level 2*. In Figure 2.2 three grids on different levels of a schematically depicted hierarchical grid are shown.

The ratio between the spatial resolutions of the levels is called refinement ratio. The refinement ratio is a positive integer greater than one. A visualization of the refinement of a single cell for some small integer refinement ratios is show in Figure 2.3 for the 2D case. For uneven refinement ratios grid points of the coarser grid are directly covered on a higher level. A high refinement ratio reduces the number of levels needed to reach a certain spatial resolution, at the cost of less gradual refinement. In [10] it is suggested to use different refinement ratios on different levels.

[1] The computational grid is often named mesh.

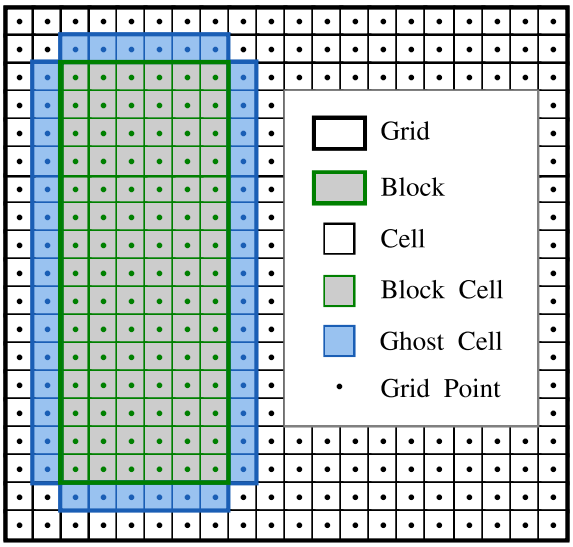


Figure 2.4: The grid (thick black) covers the entire discretized domain, whilst the block (green) covers only a sub-set. The cells directly neighboring the block cells are called ghost cells (blue). The shown ghost cells allow the usage of the seven-point stencil at block boundaries (2.1). They are used to exchange data with other blocks and to set boundary conditions.

However, all simulation examples presented in this thesis use a uniform refinement ratio on all levels, which is set to four and is considered an industry standard (cf. Figure 2.3c).

The goal of AMR is to increase the spatial resolution only locally, thus only parts of the grids are used for the computation. The parts of a grid that are used for computations (in process TCAD simulations those parts are regions around interfaces, especially around corners and edges, and material region boundaries) are defined by blocks.

A block is a rectilinear sub-domain of a grid consisting of a contiguous set of cells (cf. Figure 2.4). A block is uniquely identified by specifying the corner (node with the smallest indices in all spatial dimensions) and the size (number of grid points in each spatial dimension). The cells directly neighboring a block are the *ghost cells* of the block. Ghost cells enable the usage of the same stencil for computations on all grid points of the block (even for grid points on the border, where parts of the stencil in principle extend beyond the block boundaries). Depending on the targeted stencil size more than one direct neighbor may be necessary for the ghost cells. The usage of ghost cells has the cost of higher memory requirements, because, as a consequence, some cells have to be stored multiple times (once in a block and possibly several times in neighboring blocks as a ghost cell). The ghost cells are used to set boundary conditions on the block and exchange data with neighboring blocks.

A hierarchical grid is created in two steps:

- Flagging: The cells on a level are flagged (marked for refinement).
- Clustering: The flagged cells are clustered into blocks.

The flagging selects the cells on a level which have to be covered on the next higher (finer) level of the hierarchical grid. As example, on *Level 0* the cells which shall be covered on *Level 1* are flagged. In the context of process TCAD simulations the cells are typically selected by their level-set value (closeness to the interface), the local curvature (high curvature identifies corners), and distance to interfaces of other level-set functions (relevant for simulations containing multiple material regions). During a typical process TCAD simulation the regions where high spatial resolution is necessary change over time. This change is due to material regions (interfaces) being deformed, yielding different regions for high spatial resolution. Therefore, the hierarchical grid has to be adapted several times (cf. Section 4.2.7) during a process TCAD simulation.

There are three different approaches for clustering, depending on additional requirements on the size and placement of blocks:

- Cell-based AMR
- Tree-based AMR
- Block-based AMR

The three approaches are portrayed in the following.

Cell-based AMR

Cell-based AMR allows for individual cells to be refined independently [80, 81]. This enables a perfect refinement efficiency (number of flagged cells divided by number of refined cells), because only those cells which are flagged are refined. However, high memory requirements for the hierarchical structure and specialized stencil computations induce a non-negligible computational overhead. The typical data structure used are quadtrees [82, 83, 84] and octrees [23, 85, 86, 87, 42], in two dimensions and three dimensions, respectively. The refinement ratio is often two, but other refinement ratios are also considered [88, 10]. Issues arising on parallel systems, especially the even distribution of the data across the hardware resources for tree-like data structures is investigated in [89, 59]: A Z-curve (linear neighbor conserving traversal of space) is utilized to spread the data evenly across the available hardware resources.

Tree-based AMR

In contrast to cell-based AMR, tree-based AMR requires that only similar blocks (blocks of the same size) which are aligned to the grid are refined. The size of the blocks has to be bigger than one, else it is equivalent to the cell-based approach. So, if a single cell of a block is flagged all cells of the block are refined, thus the refinement efficiency is lower, but the computational overhead for storing the structure is smaller compared to the cell-based approach. Tree-based AMR is used for example in [10, 90, 91]. Parallelization of tree-based AMR is typically performed on a per block basis (each block is processed by a dedicated thread).

Block-based AMR

Block-based AMR (also known as patch-based AMR) allows differently sized blocks, thus allowing for a higher refinement efficiency. The cost of the higher refinement efficiency is a larger computational overhead for each block, because the size and the connectivity between blocks is not straightforward as in the tree-based approach [92, 93]. The connectivity (the neighboring blocks) is typically stored as a dedicated list for best performance. In this case the clustering is performed using a signature-inflection clustering method based on the approach presented in [94].

Starting from a single block covering the full domain, the block is iteratively split and trimmed to remove most of the not flagged cells. Typically a minimum size requirement is imposed on the blocks, which reduces the total number of blocks (high overhead of small blocks, due to the ghost cells). However, no maximum size for the blocks is set. Level-set simulations using this approach are presented by [95, 96]. Differently sized blocks (possible in this approach) have to be considered when parallelizing computations, because they may cause load-imbalances (computations on large blocks take usually significantly longer than on small ones).

In this thesis and in the presented reference simulation workflow (cf. Section 4.2) the block-based AMR approach is considered for all simulations examples. The reference simulation workflow always employs a single block on *Level 0* covering the full computational domain. On the higher levels of the hierarchical grid several blocks are present, depending on the simulation problem at hand.

There are other approaches to AMR considered in level-set simulations, which are portrayed shortly in the following for the sake of completeness.

Other Approaches

The approach to AMR considered in [97] employs, locally around the interface, cells that contain a higher order approximation of the level-set function. The approximation in those cells is based on Gauss-Lobatto quadrature nodes instead of a single grid point per cell. This approach is conceptually similar to a higher spatial discretization locally around the interface.

For level-set simulations, there are many approaches where only cells in a narrow-band (several grid cells wide) around the interface are stored. Those approaches typically consider only a single grid, which usually employs a high spatial resolution. For example, in [98] the usage of a hash table data structure to store the narrow-band is proposed, but the conducted performance comparison showed no advantage to a reference cell-based AMR implementation.

The approach to only store grid points in a narrow-band around the interface is taken to the extreme in [99]. This approach, named sparse field, stores only grid points which are directly next to the interface (any computations typically requires the extension of the band of stored grid points).

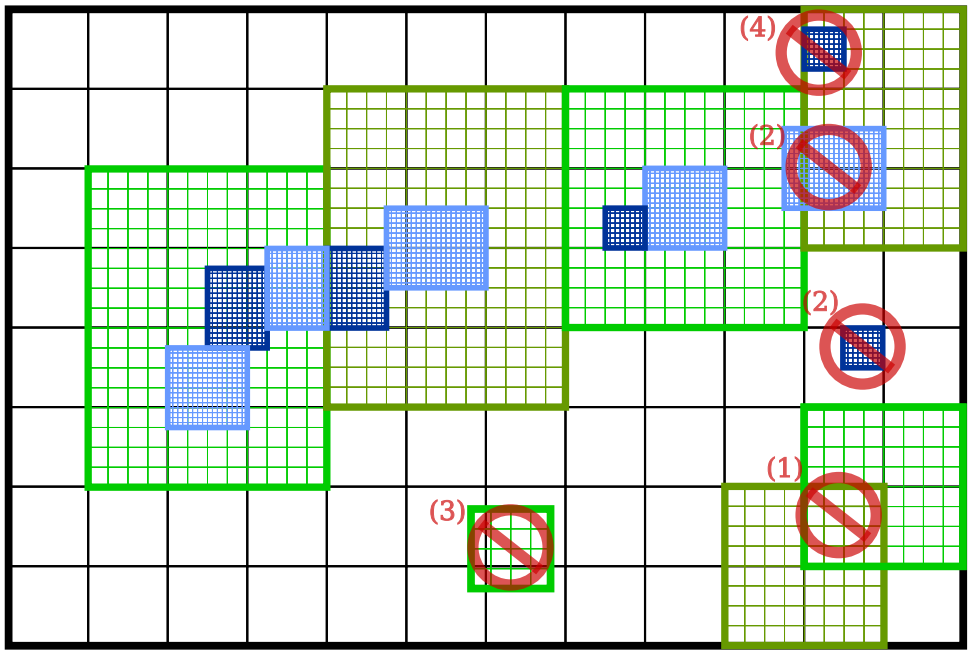


Figure 2.5: An exemplary hierarchical grid containing three levels with blocks. Blocks violating the four discussed nesting criteria are crossed out in red. A refinement ratio of four is used in both spatial dimensions. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Studies in Computational Intelligence* 902 (2021), pp. 438-451. [100], © 2021, under exclusive license to Springer Nature Switzerland AG.

2.3 Nesting Criteria

As mentioned previously, this thesis considers a block-based AMR approach. As such, the block placement and nesting criteria are essential and discussed in the following.

Data exchange between blocks of a hierarchical grid is necessary to couple the solution on different blocks. Data is exchanged between blocks on the same level, as well as between blocks on different levels. Data exchange procedures are costly, if an arbitrary relation between blocks has to be handled, because for each data exchange all blocks on all levels have to be considered. Therefore, enforcing placement rules on the blocks, i.e., restricting the possible relations between blocks, allows for optimized data exchange procedures which only require the consideration of a *small* sub-set of all blocks.

These placement rules are referred to as nesting criteria because they define how blocks are *nested* on hierarchical grids. The nesting criteria structure hierarchical grids and enable more efficient data exchange between blocks on different levels of a hierarchical grid. Data exchange from a coarse level to the next finer level is usually referred to as *interpolation* and conversely from a fine level to the next coarser level as *restriction*. If two blocks on different levels overlap the term *parent* for the block on the coarser level and the term *child* for the block on the finer level is used.

The four key nesting criteria considered in this work and representing an industry standard are:

1. Blocks shall not overlap other blocks on same level.
2. Each block has a unique parent block on the next lower level, except for *Level 0* where there is no parent block.
3. Blocks shall be aligned to the grid on the next lower level.
4. A block shall not border an area which is not refined on the next coarser level.

The first criterion excludes overlapping blocks which would otherwise cover cells multiple times, resulting in computational overhead. Thus only ghost cells are stored more than once on a level of a hierarchical grid.

The second criterion effects the interaction between blocks on different levels. The main advantage having a unique parent block compared to approaches where the parent block is not unique, is that for data exchange procedures between levels of a hierarchical grid only two blocks have to be considered: The parent and its child block. The drawback is that more blocks are created on higher levels of a hierarchical grid, because the blocks have a maximum size imposed by their parent blocks.

The third criterion is automatically fulfilled in case the refinement is based solely on the flags of the coarser level. The underlying grids with different spatial resolutions are inherently aligned. Thus a cell is always either fully covered by refined cells or not covered at all, allowing for simplified exchange procedures between the levels.

The fourth criterion avoids a *harsh* border in the spatial resolution: The so-created gradual change in the spatial resolution, enables a stable solution.

In Figure 2.5 an example hierarchical grid in two dimensions with a total of three levels is presented. The employed refinement factor is four. The single block on *Level 0* is outlined by a thick black line, whilst the block cells use a thin black outline. On *Level 1* the blocks (total of four valid blocks) are colored in shades of green, and on *Level 2* the blocks (total of seven valid blocks) are colored in shades of blue. Examples of blocks violating the four required nesting criteria a marked with respect to the violated nesting criterion list number (see list above) in red.

Collection of Terms

Analogously to [101] the terms used in this work to describe hierarchical grids are summarized (ordered alphabetically):

- **block**: Axis-aligned collection of continuous cells of the same size
- **block cells**: Cells belonging to a block
- **cell**: Smallest unit of the computational domain
- **ghost cell**: Halo of cells surrounding the block cells
- **ghost point**: Grid point belonging to a ghost cell
- **grid**: Generic description of the computational domain
- **grid line**: Axis-aligned line connecting two grid points
- **grid point**: Location of the center of a cell
- **level**: Union of blocks that have the same spatial resolution
- **node**: Index of a cell

Chapter 3

Parallelization and Hardware

This chapter presents the basic concepts of parallelization and multiprocessor programming [102, 103]. The subsequently established terminology allows for a precise description and analysis of the developed algorithms. At the end of the chapter the hardware resources used for the benchmarks presented in the remainder of this work are listed.

3.1 General Parallelization Strategies

Parallelization is the transformation of an algorithm to be able to execute it in parallel. The goal of parallelization is the reduction of the run-time required for execution of an algorithm, because (some of) the instructions of the algorithm are executed simultaneously. To be able to parallelize an algorithm there are two prerequisites: 1) The instructions of the algorithm have to allow reordering (concurrency) and 2) there must be hardware available to perform instructions simultaneously (parallel execution).

Concurrency

Concurrency is the ability to execute instructions, e.g., of an algorithm, out-of-order, without affecting the final outcome [102]. If a problem does not allow for concurrency, the instructions are not parallelizable. To illustrate the difference two situations are described in detail.

Consider two arrays storing a set of arbitrary numbers. The goal is to add both arrays element wise together. In this case a serial algorithm may add the first number of each array, then the second number and so forth. For the result (the element wise sum of the array numbers) the order in which each element pair of the arrays is added does not matter. Thus the example allows for concurrency and it is possible to parallelize the algorithm and compute the sum of the individual element pairs simultaneously.

Now consider the task to find the end of a linked list given its head (first element). The only way to reach the end of a linked list is to follow the pointer to the next element of the current element until there is no next element, thus the end is reached.

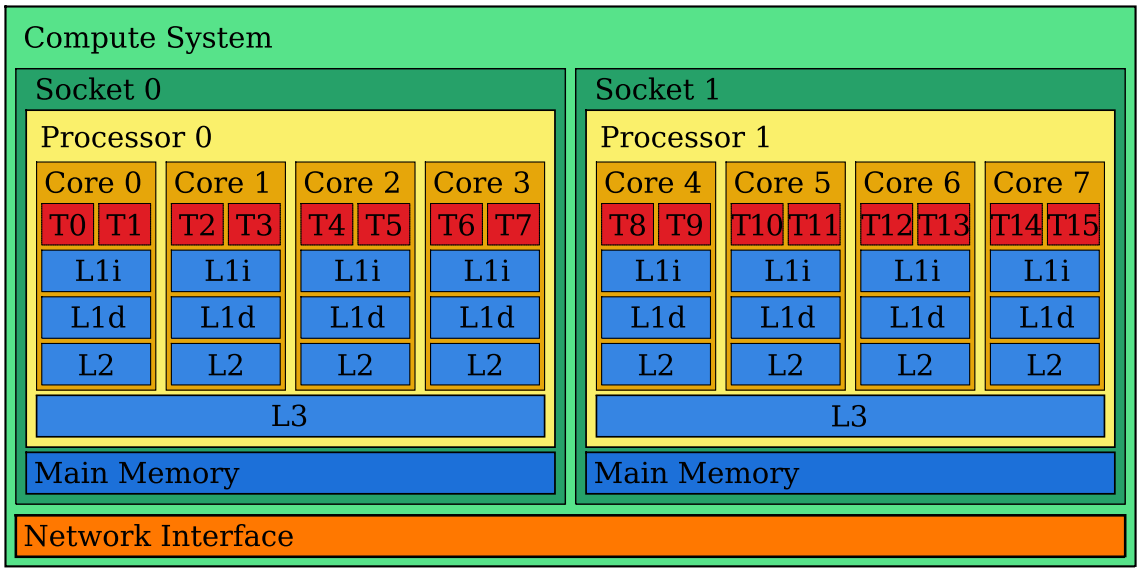


Figure 3.1: Schematic of an exemplary shared-memory parallel compute system which has two sockets each equipped with a four core processor. The cores of a processor share the level 3 cache (L3), but have dedicated level 2 (L2) and level 1 instruction and data (L1i and L1d) caches. Each core has the capability to execute two threads (e.g., T0 and T1).

Because there is only a single way to reach the end of the linked list and the steps require a definite ordering the task is unparallelizable.

Parallel Execution

To actually reduce the run-time of the concurrent instructions (the goal of parallelization), the independent sets of instructions resulting from concurrency considerations have to be executed simultaneously. As mentioned previously, in this work only shared-memory compute systems are considered. The parallel execution on shared-memory compute systems is typically achieved by scheduling threads to different cores on a processor.

Figure 3.1 shows a schematic of a typical shared-memory parallel compute system. On the depicted shared-memory parallel compute system there are two sockets each equipped with a processor consisting of four physical cores each supporting two-way simultaneous multithreading (i.e., support for executing two threads per core): a total of eight logical cores is thus provided by each processor.

A processor has a hierarchy of caches to reduce the data access time from the main memory. The size of a cache is indirectly proportional to the access time.

On a shared-memory parallel compute system, the main memory storing all data of a program is split into memory domains, one for each socket. Each core has still access to the entire main memory of the compute system. However, access to data residing in another memory domain comes with the cost of higher access latencies and lower bandwidths as data has to be transferred via an additional coherent link connecting the processor sockets and the associated memory domains. Such systems are labeled non-uniform memory access (NUMA) systems and are widely spread in professional workstations and large-scale compute clusters/supercomputers.

Data exchange among threads being executed within the same process is inherently possible due to shared access of the main memory. This data exchange mechanism is considered low overhead, considering alternatives like pipes and sockets offered by the operating system. However, the reading and writing to the same memory location has to be carefully implemented, potentially requiring the need for dedicated mechanisms (lock, or atomic operations) to get exclusive access to a memory location. This avoids data races (non-synchronized accesses to the same memory location), which may lead to unexpected results of the computation (undefined behavior).

In this work, parallelism is classified into two types:

- Coarse-grained
- Fine-grained

Coarse-grained parallelism is characterized by a relatively large amount of work per thread. Synchronization and data exchanges between threads are typically costly, thus they have to be *rare* so that the overall created overhead is low. Coarse-grained parallelization suits algorithms where the run-time of threads is predictable. This enables the creation of threads which will have the same run-time, so that scenarios where a single long-running thread blocks the continuation of execution for other threads are mitigated.

Considering the previous example where two arrays are added element wise together, the array could for instance be split into as many chunks of data as cores are available. Data exchange has to be performed in the beginning, i.e., distributing the chunks of the array to the threads, and in the end, i.e., synchronizing the threads, to ensure all threads are finished. The so created chunks of data are coarse (thus the name), because the number of instructions performed by a thread is high compared to the number of data exchanges between threads. However, the splitting of data creates additional computations (overhead), because determining the chunk size also requires computations. If the number of computations per thread is low (which of course is problem-specific), the overhead diminishes all gains from parallel execution.

Within the context of hierarchical grids computations on a single block typically correspond to a suitable chunk of data for a thread (balancing overhead and parallel performance).

Fine-grained parallelism, on the contrary, is characterized by a relatively small amount of work per thread before synchronization between threads is required. Fine-grained parallelization suits algorithms where the run-time of threads is unpredictable, but synchronization costs are low.

Again, considering the previous example where the two arrays are added element wise together, the array is split into significantly more chunks of data than cores are available. Thus a core is going to execute more than a single thread over the duration of the program. If a core is assigned a new thread synchronization is involved, i.e., identifying the not processed threads. Thus the overall run-time is typically larger compared to the coarse-grained parallelization approach for the considered example adding two arrays element wise together.

However, in case the run-time of the operation performed on each array element is unpredictable the fine-grained parallelization approach would be superior because run-times of different threads are balancing each other. In the end the cores will finish the execution of all threads almost at the same time (almost no idling).

Programming Model

The OpenMP application programming interface (API) specification [104] is widely used to develop software for shared-memory systems and is also used in this work OpenMP allows easy parallelization of existing programs, especially of for-loops, by using standard compiler directives, e.g., `#pragma omp parallel for`. By setting the number of threads the range of the for-loop is automatically divided into number of threads chunks which are executed in parallel.

To handle more complex parallelization scenarios where a straightforward parallelization of for-loops is not possible, OpenMP has the concept of `OpenMP tasks`. Conceptually, a task is the same as a thread, however, a task is managed by OpenMP, instead of the operating system. OpenMP internally utilizes a thread pool to enable fast and low overhead parallel execution of tasks. A thread pool has a fixed number of threads (managed by the operating system) which dynamically execute scheduled tasks (managed by OpenMP).

To understand the viable performance gains from parallelization the theoretical limits are explored in the next section.

Amdahl's Law

The theoretical expected maximum speedup (run-time reduction factor) achievable through parallelization is given by Amdahl's law. Most algorithms consist of parts which allow for concurrency and some parts which do not. Let the fraction (relative to the full algorithm) which allows for concurrency of an algorithm be denoted by c and the number of used threads (equal to the available cores) be t then the maximum parallel speedup S (ignoring any introduced overhead by parallelization) is given by

$$S = \frac{1}{1 - c + \frac{c}{t}}. \tag{3.1}$$

In the limit of an infinite number of available threads the parallel speedup is limited by

$$S = \frac{1}{1 - c}, \tag{3.2}$$

which is indirectly proportional to the fraction which does not allow for concurrency. Considering an algorithm with $c = 0.9$, i.e., 10 % of the algorithm is serial, the parallel speedup is limited by a factor of 10. Therefore, it is essential that even minor parts of an algorithm are parallelized (minimizing the serial part limiting the parallel speedup), if a highly parallel execution is targeted, i.e., multi-core processors.

In high performance computing the efficiency of parallel algorithms is evaluated by two common analyses quantifying the scalability.

Strong scaling analysis is defined as how the run-time varies with the number of threads for a fixed total problem size. The parallel speedup is defined as the single-threaded run-time compared to the multi-threaded run-time. Ideally the run-time is reduced linearly if more threads are used. The run-time reduction typically saturates for a high number of threads due to Amdahl's law.

Weak scaling analysis is defined as how the run-time varies with the number of threads for a fixed problem size per thread. For example, if the number of threads is doubled, the problem size is also doubled, but the run-time would be the same in case of optimal *weak scaling*. Amdahl's law is not applicable to the *weak scaling* analysis, because the total problem size is not fixed, i.e., the problem size grows with the number of used threads.

As will be shown in later chapters, this work primarily conducts *strong scaling* analyses as they allow for intuitive interpretation regarding parallel speedup when considering process TCAD simulation workflows.

The next section introduces the hardware used for benchmarking

3.2 Benchmark Systems

This section gives an overview of all the compute systems used for evaluating the performance henceforth denoted as *benchmark systems* of the implementations of the proposed algorithms. The benchmark systems are single compute nodes from two generations of Vienna Scientific Cluster[1] (VSC) supercomputers and an industrial compute system. In Table 3.1 the key properties of the three available benchmark systems are summarized.

Table 3.1: Summary and key properties of the used benchmark systems.

	VSC3	VSC4	ICS
Frequency (GHz)	2.6	3.1	2.8
Sockets	2	2	2
Cores per CPU	8	24	10
Logical cores per CPU	16	48	20
L1i cache	32 KByte	32 KByte	32 KByte
L1d cache	32 KByte	32 KByte	32 KByte
L2 cache	256 KByte	1024 KByte	256 KByte
L3 cache	20 MByte	33 MByte	26 MByte
Main memory	64 GByte	96 GByte	226 GByte

[1]The VSC is a collaboration of several Austrian universities that provides supercomputer resources and corresponding services [105].

Chapter 4

The Level-Set Method

This chapter starts with the theoretical (mathematical) background of the level-set method (cf. Section 4.1), clearly defining the used terms such as level-set function and signed-distance function. In Section 4.2, the reference implementation of the level-set method within the context of a simulation tool is presented, providing a detailed discussion of all the computational steps in dedicated subsections. The computational steps range from the creation of the level-set function representing the interface and the movement or deformation of said interfaces, and finally to the extraction of an explicit representation of the interfaces from the full simulation domain, which is then used for further process or device TCAD simulations. The chapter concludes with a short overview of the available process TCAD simulators which use the level-set method (cf. Section 4.3).

4.1 Theoretical Background

This section discusses the implicit representation of geometries using a function and their movement (deformation).

4.1.1 Level-Set Function

For a given $\Omega \subset \mathbb{R}^n$ (process TCAD simulations typically consider $n = 2$ or $n = 3$) and an interface $\Gamma \subset \Omega$ separating the *inside* $\Omega^- \subset \Omega$ from the *outside* $\Omega^+ = \Omega \setminus \Omega^-$. The level-set method describes the interface separating the *inside* from the *outside* implicitly by a function $\Phi(\vec{x})$. This function also known as the level-set function, has to be continuous and fulfill

$$\Phi(\vec{x}) \begin{cases} < 0 & \text{for } \vec{x} \in \Omega^-, \\ = 0 & \text{for } \vec{x} \in \Gamma, \\ > 0 & \text{for } \vec{x} \in \Omega^+. \end{cases} \tag{4.1}$$

Because this work only considers the 2D and 3D case, let $\vec{x} = (x, y, z)$, where the z-component is only considered in the 3D case. Figure 4.1 shows a 2D example of an interface. The choice on which side of the interface the sign is negative is arbitrary. The convention in this thesis is that the *inside* shall have the negative sign.

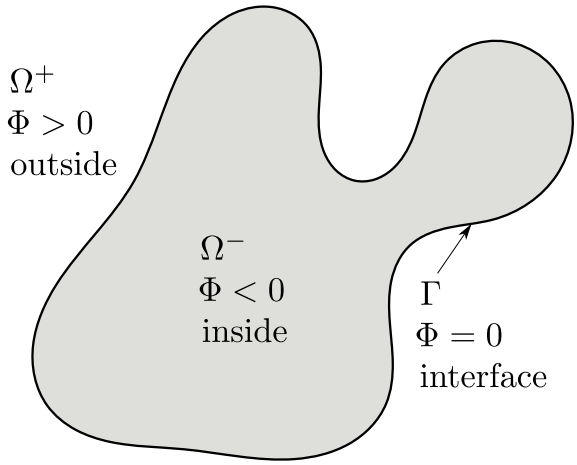


Figure 4.1: Representation of a 2D region Ω^- via the level-set function $\Phi(\vec{x})$.

Also, the choice to select the zero-level-set as the interface is arbitrary (any other value is possible as well), but by choosing zero the sign of the function is sufficient to discriminate between *inside* and *outside*.

The gradient of Φ, if it exists (on corners and edges of Γ the gradient does not exist), is given by

$$\nabla\Phi = \left(\frac{\partial\Phi}{\partial x}, \frac{\partial\Phi}{\partial y}, \frac{\partial\Phi}{\partial z}\right), \tag{4.2}$$

and is perpendicular to the iso-contours of Φ, including the special iso-contour for the value zero (zero-level-set) of Φ. Therefore, normalizing the gradient yields the normal vector[1] of the interface

$$\vec{n} = \frac{\nabla\Phi}{|\nabla\Phi|}. \tag{4.3}$$

The so defined normal vector points outwards because the gradient points in the direction of increasing Φ. Φ is by definition (4.1) smaller on the *inside* than on the *outside*. The definition of the normal vector via the normalized gradient allows for a straightforward embedding of the interface normal vector $\vec{n}$ in Ω. Therefore opening possibilities for a geometric interpretation of the level-set function off of the interface, e.g., curvature in the entire domain Ω. Unfortunately, the minimal requirements to the level-set function do not allow for a geometrical interpretation of the normal vector at points which are not on the interface Γ. Thus, practical level-set simulations typically require more properties of the level-set function, discussed in the next section.

[1] The term *normal vector* in this thesis always implies the unit normal vector, i.e., its norm is equal to one.

4.1.2 Signed-Distance Function

In practical applications the level-set function Φ is often chosen as a signed-distance function relative to Γ

$$\Phi(\vec{x}) = \begin{cases} -d(\vec{x}, \Gamma) & \text{for } \vec{x} \in \Omega^-, \\ 0 & \text{for } \vec{x} \in \Gamma, \\ +d(\vec{x}, \Gamma) & \text{for } \vec{x} \in \Omega^+, \end{cases} \tag{4.4}$$

with d the Euclidean distance to the interface Γ. The Euclidean distance between two points is given by

$$d(\vec{x}, \vec{y}) = \sqrt{\sum_{i \in \{x,y,z\}} (\vec{x}_i - \vec{y}_i)^2}, \tag{4.5}$$

with $\vec{x}_i$ the component of the vector $\vec{x}$ in the corresponding spatial direction. The distance between a point and a set (i.e., the interface Γ) is given by the infimum of the Euclidean distance over all points of the set

$$d(\vec{x}, \Gamma) = \inf_{\vec{y} \in \Gamma} d(\vec{x}, \vec{y}). \tag{4.6}$$

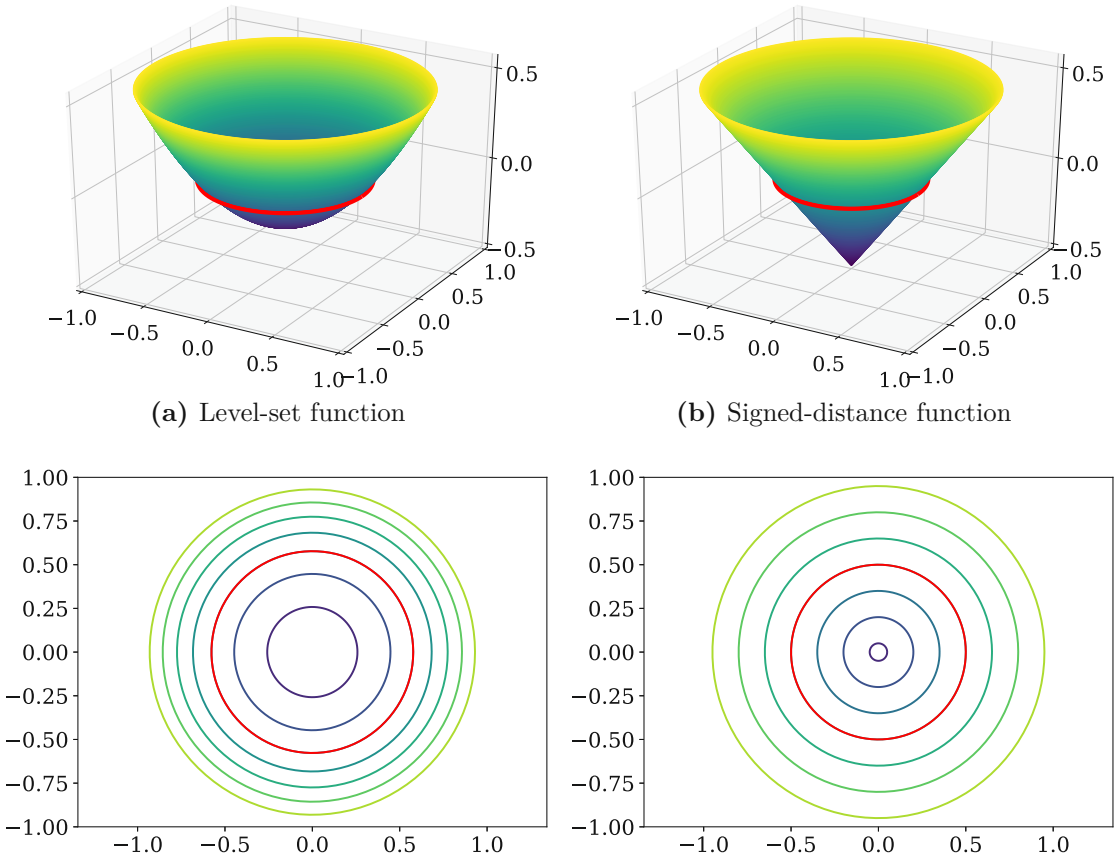
The choice, to use a signed-distance function, allows for geometrical interpretation of the normal even for points not on the interface. The normal points away from the closest point on the interface on the *outside*, and to the closest point on the interface on the *inside*. If Φ is a singed-distance function $|\nabla\Phi| = 1$ holds.

This fact is easily reasoned by considering a point $\vec{x}$ and the corresponding closest point on the interface $\vec{x_\Gamma}$. All points $\vec{y}$ on the shortest path connecting $\vec{x}$ and $\vec{x_\Gamma}$ have the same point $\vec{x_\Gamma}$ as the closest point on the interface. This is due to the triangle inequality. Thus the path connecting $\vec{x}$ and $\vec{x_\Gamma}$ is the path of steepest descent for the level-set function Φ, evaluating to $-\nabla\Phi$. Furthermore, because Φ is scaled according to the Euclidean distance it follows that $|\nabla\Phi| = 1$.

The representation using a signed-distance function allows for a faster computation of the normal vector of the interface, because the normalization step is dispensable

$$\vec{n} = \frac{\nabla\Phi}{\underbrace{|\nabla\Phi|}_{=1}} = \nabla\Phi. \tag{4.7}$$

In Figure 4.2 two different level-set functions for the same interface (a circle) are shown. Figure 4.2a shows an arbitrary level-set function whereas Figure 4.2b shows the use of a signed-distance function. Figure 4.2c and Figure 4.2d show the corresponding iso-contours. In case of the signed-distance function the iso-contours are evenly spaced, whereas in the other case they tend to cramp up or spread out. If the level-set method is solved analytically, this is not an issue, but in case of a discretization and numerical procedures issues arise.



(a) Level-set function

(b) Signed-distance function

(c) Iso-contours of the level-set function

(d) Iso-contours of the signed-distance function

Figure 4.2: A circle with radius $r = 0.5$ is represented **(a)** using the level-set function $\Phi(x, y) = x^2 + y^2 - r^2$ and **(b)** using the signed-distance function $\Phi(x, y) = \sqrt{x^2 + y^2} - r$. The zero-level-set is drawn in red. Iso-contours (level-sets for other values) are irregular spaced in **(c)**, but in the case of a signed-distance function they are equidistantly spaced **(d)**.

Numerical issues arising from such steep or flat gradients of the level-set function are:

- Steep gradients may exceed the numerical representation of floating-point numbers.
- Flat gradients are prone to distortions of the interface position (small perturbations of the level-set function lead to enormous perturbations of the interface position).

One point in the domain for the signed-distance function in Figure 4.2 is special, i.e., the center of the circle (the apex of the cone), because it is the only point for which $|\nabla\Phi|$ is undefined. Generally, for a signed-distance function the gradient for all points on the skeleton of Γ is undefined. The skeleton of an interface Γ consists of all points $\vec{p} \in \Omega$ which have more than one closest point on Γ, e.g., for a circle the center or for a square the diagonals [106].

This seems to be a critical drawback when using a signed-distance function, but as the equations under consideration in this work are generally true, e.g., $|\nabla\Phi| = 1$ holds almost everywhere (except for a negligible subset of Ω).

The advantages of the geometric interpretation of the level-set function and its numerical robust gradient outweigh the drawback not being able to define the gradient everywhere. An almost everywhere true equation, e.g., $|\nabla\Phi| = 1$, may still be approximated numerically, if the approximation *'Fails in a graceful way'* [18], meaning that the failure does not cause a deterioration of the underlying numerical method.

4.1.3 Interface Movement

Assume the velocity $\vec{V}$ is given for each point of the interface Γ. The movement of the interface is given by the movement of all interface points, whereas the movement for a single interface point $\vec{x}$ is described by

$$\frac{d\vec{x}}{dt} = \vec{V}\left(\vec{x}, t\right). \tag{4.8}$$

Such a description of the interface movement would require an explicit representation of the interface, because the movement of individual points of the interface is described. The level-set method is an implicit approach, therefore, the approach has to be adapted. The level-set function Φ is used in the description of the interface movement. For the interface movement a time dependency is introduced to the level-set function $\Phi\left(\vec{x}, t\right)$. The movement of the interface is described by the advection (convection) equation

$$\frac{\partial\Phi\left(\vec{x}, t\right)}{\partial t} = \vec{V}\left(\vec{x}, t\right) \cdot \nabla\Phi\left(\vec{x}, t\right), \tag{4.9}$$

where Φ is the well-known signed-distance function and $\vec{V}$ is a velocity field describing the interface movement. Equation (4.9) is also known as the level-set equation. The interface is moved because the zero-level-set of Φ changes over time.

In contrast to the formulation in (4.8), which presents the movement from a Lagrangian specification (the observer follows points), (4.9) uses the Eulerian specification (observer watches which points pass by). Both specifications are related via the equation

$$\Phi\left(\vec{X}\left(\vec{x}_0, t\right), t\right) = \frac{\partial\vec{X}}{\partial t}\left(\vec{x}_0, t\right), \tag{4.10}$$

where $\vec{X}\left(\vec{x}, t\right)$ is the position of the point $\vec{x}$ according to (4.8) at time t and $\vec{x}_0$ is a generic point on the interface.

In cases where strictly only the interface is of interest a scalar velocity field is sufficient to prescribe the interface movement. The interface position is only affected by the to the interface orthogonal component of the velocity field. Let $\vec{V} = v\vec{n} + \vec{P}$, where $\nabla\Phi \perp \vec{P}$ holds and v is the scalar velocity in normal direction of the interface.

Then (4.9) simplifies to

$$\frac{\partial \Phi}{\partial t} = \vec{V} \cdot \nabla \Phi, \tag{4.11}$$

$$= \left(v\vec{n} + \vec{P}\right) \cdot \nabla \Phi, \tag{4.12}$$

$$= v \underbrace{\vec{n} \cdot \nabla \Phi}_{=|\nabla \Phi|} + \underbrace{\vec{P} \cdot \nabla \Phi}_{=0}. \tag{4.13}$$

Additionally, in case of a signed-distance function, (4.9) simplifies further to

$$\frac{\partial \Phi}{\partial t} = v. \tag{4.14}$$

This final formulation allows for a short and elegant way to describe the interface movement for a given velocity field.

In the next sections the computational steps of the used reference simulator are presented.

4.2 Reference Simulation Workflow

This section provides details of the considered reference process TCAD simulation workflow which is based on the level-set method. In essence, the simulation workflow consists of three main parts (cf. Figure 4.3):

- The initialization, in which the level-set functions are set up (Section 4.2.1).
- The main time loop (Section 4.2.2 – Section 4.2.7) where the device structure (topography) is advanced in small time steps (solving the level-set equation (4.9) in time steps of size Δ_t).
- The finalization in which an explicit representation of the material regions is extracted (Section 4.2.8).

The next sections provide a more in-depth overview of the individual steps shown in Figure 4.3.

4.2.1 Initial Interfaces

The first step of a level-set simulation is to create the level-set functions to represent the material regions. The spatial discretization approach considered here (hierarchical grids) allows to discuss the implementation as if it would use as single Cartesian grid, except for *Re-Gridding* which adapts the blocks on the hierarchical grid in position and size to the changing material regions.

Figure 4.4 shows a discretized level-set function on a Cartesian grid. On each grid point an approximation of the signed-distance to the interface is stored. Each grid point is located in the center of a cell (black square). Such a level-set function may be used to describe a single material region.

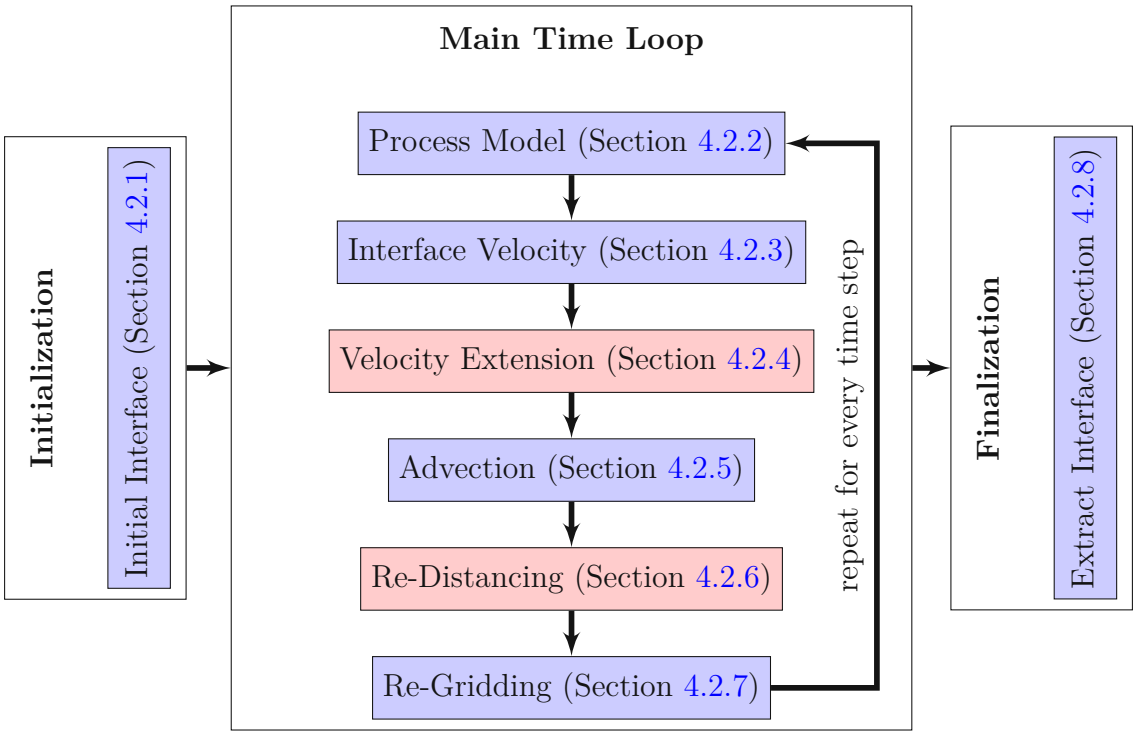


Figure 4.3: Simulation workflow of a level-set simulation, clustered into the three main parts. The focus of this thesis is on the computational steps marked in red.

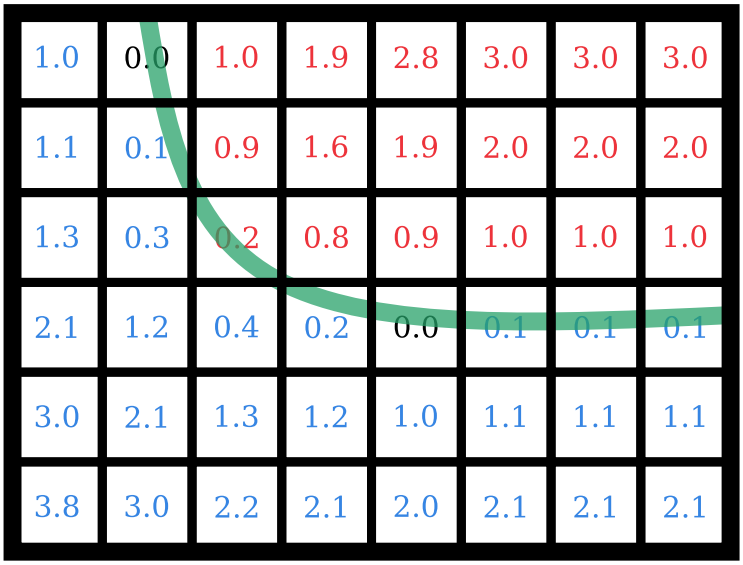
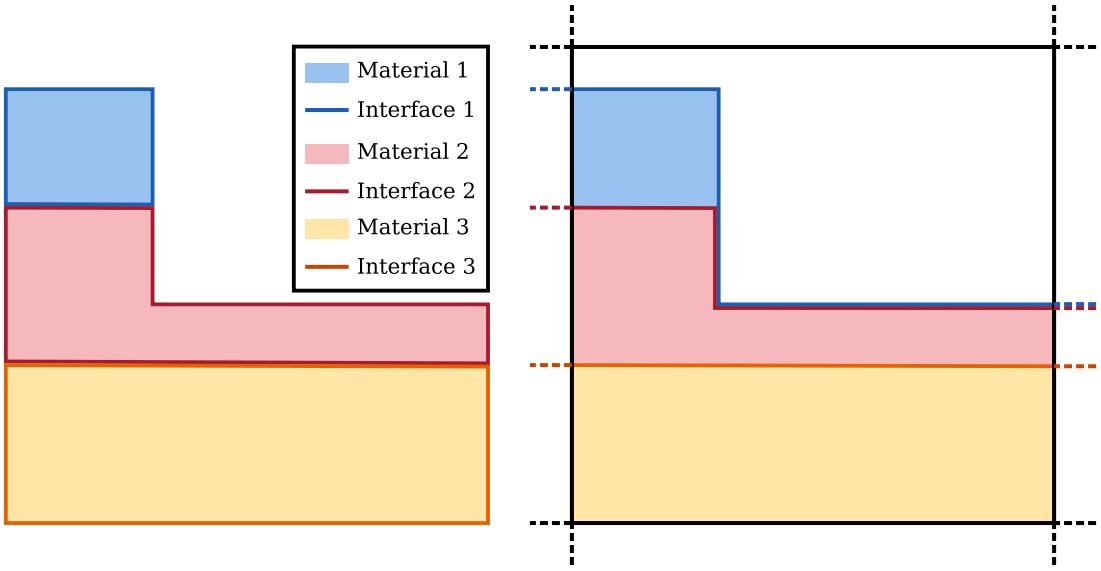


Figure 4.4: A level-set function for the interface (green curve) discretized on a Cartesian grid with a grid resolution of one on all axis. The color of a level-set value (valid at the center of a box) defines the sign of the level-set value, blue *inside*, red *outside*, and black directly on the interface.



(a) Material regions **(b)** Additive level-sets

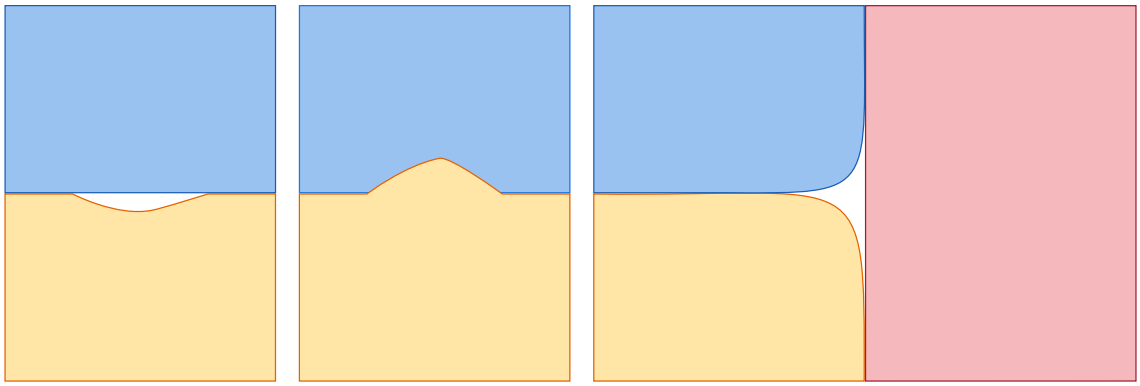
Figure 4.5: In **(a)** three material regions are enclosed by three separate interfaces. In **(b)** the same material regions are represented by additive level-sets and the domain boundary conditions are taken into account. The interfaces Interface 1 and Interface 2 are overlapping (identical) on the exposed area of Material 2. Adapted with permission from Quell *et al., IEEE Transactions on Electron Devices* 68.11 (2021), pp. 5430–5437 [55], © 2021 IEEE.

Technically, a level-set function describes two regions: 1) A material region and 2) the complement to the material region, e.g., the device structure and the void above the device structure in the simulation domain.

The discuss continues with a detailed description of material representation approaches.

Material Representation

As shown by the thermal oxidation example (cf. Section 1.1) device topographies consist of more than a single material region. A level-set function may only model a single material interface, thus several level-set functions are necessary to represent multiple material regions (cf. Figure 4.5). There are different approaches how the level-set functions are configured to achieve this goal. The straightforward approach using a dedicated level-set function encapsulating the material region for each material as shown in Figure 4.5a, has some drawbacks which are further illustrated in Figure 4.6. At the interface between two material regions non-physical voids may form (due to numerical inaccuracies of the level-set functions), because two level-set functions represent the same material interface (cf. Figure 4.6a). Sometimes no voids materialize, but the material regions defined by the level-set functions overlap. However, typical process TCAD simulations do not consider (allow) alloys, i.e., the material has to be unique at each point (cf. Figure 4.6b).



(a) Void **(b)** Overlap **(c)** Triple-junction

Figure 4.6: Drawback of the straightforward approach using a level-set function individually encapsulating each material region.

Also *triple-junctions* (points where three material regions meet) are destined to form non-physical voids, because a numerical implementation causes slight unavoidable rounding of corners (cf. Figure 4.6c). The rounding is related to the grid resolution, if the grid resolution approaches zero the rounding vanishes completely.

Several strategies to avoid the aforementioned voids were developed, primarily driven by research on multiphase flows [107, 108, 109, 110]. The material-specific level-set functions are either held together, e.g., deliberately changing the velocity field before the *Advection* to avoid the forming of voids or forced together, e.g., using Boolean operations after the *Advection* to remove overlap of the level-set functions.

In general, the focus of the various research strategies is on *triple-junctions* in 2D fluid dynamic simulations. The previously discussed approaches all require $M-1$ level-set functions for M different materials. A level-set function separates two material regions, thus regions belonging to no level-set function are not explicitly represented. In context of process TCAD simulations the void (*vacuum* or *gas*) above the to-be-simulated structure is typically the material that is not explicitly represented.

In [111] a conceptually different approach is presented, which uses for each material pair interface a dedicated level-set function leading to a maximum number of level-set functions of $M(M-1)/2$ for M materials and all materials having a pairwise interface. A sophisticated voting mechanism decides which of all these level-set functions is used for the actual computations. This enables to represent *triple-junctions* without voids and overlaps. However, the high number of level-set functions required in this approach represents a computational burden. The most widely-used approach in process TCAD simulations is to use additive level-sets [3, 20] (cf. Figure 4.5b). Instead of storing individual level-set functions representing a dedicated material, the level-set functions store a union of materials. The approach is considered in this work and further presented in the following paragraphs.

In an additive level-set approach the first material (used in a simulation) is represented with a single level-set function, as described in Section 4.1.1. Every time a new material is added (can happen several times during a practical simulation workflow) a new level-set function describing the union of all previously present material regions and the new material is added. The defined level-set function always represents the interface between the structure (device) and the *vacuum* or *gas* region.

The additive level-set approach prevents the formation of non-physical voids, because at a material boundary only a single level-set function defines the interface, in contrast to the straightforward approach (cf. Figure 4.5a), where two level-sets are present (one for each material).

Additionally, the additive level-set approach allows representing material regions with thicknesses less than the grid resolution. The union with the other materials creates a thicker material region which can be represented by a level-set function. The straightforward approach representing material regions individually is not able to reliably store such thin material regions: Storing such thin material regions is only possible, if the material region is aligned to the grid and has a symmetric offset to the grid points it encloses. Thus, only planar structures are possible in the straightforward approach, if thin layers are considered.

The additive level-set approach allows simulating physically accurate etching processes where thin *etch stop layers* (a thin material region which is hardly affected by the etching process) are employed, without the necessity for unfeasible high spatial discretization. The material regions of the *etch stop layers* may then be reconstructed via Boolean operations after an explicit interface representation has been extracted (explicit interfaces are not bound to a grid and therefore have no restriction originating from the grid resolution on their thickness).

A modification to the additive level-set approach to simulate deposition processes, where the process model is highly dependent on the interface normals (e.g., epitaxial crystal growth), is derived in [112]. There, a different strategy to unionize the material regions is used, with the goal that the outer most level-set, which is in this case not the wafer surface, has the correct local curvature (convexity and concavity) enabling the formation of crystal facets. However, Boolean operations allow a straightforward conversion to the additive level-set approach.

The additive level-set approach heavily relies on Boolean operations, which are straightforward for level-set functions and discussed in the following.

Boolean Operations

Boolean operations on level-set functions allow for efficient and high performing implementations [106], because their computation is based on a point-wise evaluation of minimum, maximum or multiplication by -1.

For two level-set functions Φ_1 and Φ_2 the operations are defined by

$$\Phi_1 \cup \Phi_2 = \min\left(\Phi_1, \Phi_2\right), \tag{4.15}$$
$$\Phi_1 \cap \Phi_2 = \max\left(\Phi_1, \Phi_2\right), \tag{4.16}$$
$$\Phi_1 \setminus \Phi_2 = \max\left(\Phi_1, -\Phi_2\right), \tag{4.17}$$
$$\Phi_1^c = -\Phi_1. \tag{4.18}$$

The union of those two level-set functions is given by (4.15), the intersection by (4.16), the difference by (4.17), and the complement of a single level-set function by (4.18). Boolean operations do not preserve the signed-distance property of the level-set function.

4.2.2 Process Model

The process model is fully responsible to define the changes to the topography by determining how the interfaces shall be transformed. In other words, the process model captures all the physics behind the simulation.

The complexity of the chosen process model varies heavily depending on the required accuracy and simulated process. The least complex process models are Boolean operations (see Section 4.2.1) and uniform deposition or etching models (equivalent to the computation of a constant offset of the zero-level-set). More complex process models are interface normal dependent models [113, 114, 115, 116] modeling anisotropic etching processes or epitaxial growth processes. Material flow processes, such as thermal oxidation [15], require the solution of a Navier-Stokes equation. There are also process models which use visibility calculations (ray tracing) to simulate direct particle transport [117, 118, 20, 13]. More sophisticated models extend their particle transport process models further to also account for an additional external flow [26, 119, 120]. The particle transport is essential for so-called reactive ion etch processes, in which the wafer surface is bombarded with ions, kinetically removing material and consequently realizing the creation of high aspect ratio devices.

The usage of the level-set method strictly decouples the process model from the interface advection. This decoupling enables a straightforward switching of the process model, while still using the same material representation and interface evolution. Therefore, comparing different process models of varying physical complexity and accuracy for the same process step is viable, which is important for practical process TCAD simulations to fine-tune predictions of fabrication processes.

In the scope of the reference simulation workflow the interaction of the process model and the level-set method uses a standardized API, which is specified in the next section.

4.2.3 Interface Velocity

The API between the process model and the level-set method is realized via *Cross Points*. *Cross Points* are intersections of the interface with grid lines (cf. Figure 4.7).

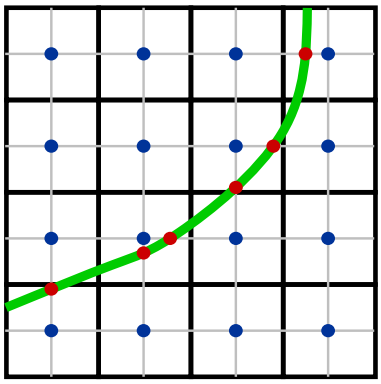


Figure 4.7: *Cross Points* (red points) are points on the interface where the interface (green curve) crosses the grid lines (gray lines). The grid lines connect the grid points (blue points) located in the center of the corresponding cell (black square).

Cross Points are chosen because they are the minimal requirement to a process model: A process model must be able to describe the movement of points on the interface. Some process models (e.g., isotropic deposition) directly yield velocities for all grid points, whilst others (e.g., models using visibility calculation) are not able to provide meaningful velocities at grid points off the interface.

A velocity given only at *Cross Points* would enable solely a Lagrangian movement of the interface (cf. Section 4.1.3). Thus, the velocity of the interface has to be extended to enable the Eulerian movement necessary for the implicit interface representation employed by the level-set method. This extension step is called *Velocity Extension* and is discussed in the following.

4.2.4 Velocity Extension

The computational step *Velocity Extension* is used to extend the velocity from the *Cross Points* to the grid points of the computational domain. This is necessary because the solution of (4.9) requires the velocity to be defined not only directly at the interface but also on the entire computational domain. The requirement for this extended velocity is to describe the interface movement, therefore, at the interface the extended velocity field and the interface velocity should match. If the extended velocity and the interface velocity do not match at the interface, they describe different interface movements.

To that end, *Velocity Extension* assigns each grid point the velocity of the closest interface point [121]. A direct computation of the closest interface point and the corresponding *Cross Point* on an implicit interface is prone to numerical errors. The closest point on the interface $\vec{x}_\Gamma$ of $\vec{x}$ is computed by $\vec{x}_\Gamma = \vec{x} - \Phi(\vec{x})\nabla\Phi(\vec{x})$ (this holds only for a signed-distance function). Even minor deviations of Φ from a signed-distance function are able to break the approach. Especially, near corners of the interface (on different sides of a corner the velocities may differ by a large margin, e.g., caused by a process model using visibility calculations and one side of the corner is shadowed) and for points further away form the interface this approach becomes very inaccurate due to numerical issues (the uncertainty of the computed point scales with the distance to the interface).

An alternative to direct computation of the closest interface point is an extension from the interface constant along the normal direction of the interface, in an outwards marching manner. The approach is formulated by a PDE (a boundary value problem) known as velocity extension equation [122]

$$\nabla\Phi\left(\vec{x},t\right)\cdot\nabla V\left(\vec{x},t\right)=0,\quad \vec{x}\in\Omega, \tag{4.19}$$

$$V\left(\vec{x},t\right)=V_I\left(\vec{x},t\right),\quad \vec{x}\in\Gamma. \tag{4.20}$$

The given velocity on the *Cross Points* (interface) is denoted by V_I whilst the extended velocity, which is used for the advection is denoted by V.

The solution to this PDE is constant along the normal direction ($\nabla\Phi$ approximates the normal direction) of the interface. Because the change of the velocity (expressed by ∇V) is orthogonal to the normal direction (4.19) each point gets the velocity of the closest point of the interface assigned by solving above equations. Extending according to (4.19) has additional numerical benefits, such as a reduced distortion of the signed-distance property, which is particularly relevant for *Advection*, as discussed in the following. The details and proposed algorithms for *Velocity Extension* (solving the PDE (4.19)) are presented in Chapter 5.

4.2.5 Advection

After *Velocity Extension* the level-set equation,

$$\frac{\partial\Phi\left(\vec{x},t\right)}{\partial t}=\underbrace{\vec{V}\cdot\nabla\Phi\left(\vec{x},t\right)}_{H(\vec{x},\Phi,\nabla\Phi,t)}, \tag{4.9 revisited}$$

is well-defined in the computational domain. The right-hand side H is called Hamiltonian in the context of Hamilton-Jacobi equations [73, 18]. The level-set equation is discretized in time[2]. For a time t^n, let $\Phi^n=\Phi\left(t^n\right)$ be the description of the interface at t^n and let $\Phi^{n+1}=\Phi\left(t^{n+1}\right)$ be the description of the interface after a *small* time step $\Delta_t=t^{n+1}-t^n$. A first-order accurate solution of (4.9) is given by the forward Euler method

$$\frac{\Phi^{n+1}-\Phi^n}{\Delta_t}=H(\vec{x},\Phi^n,\nabla\Phi^n,t^n), \tag{4.21}$$

with H the Hamilton evaluated at time t^n.

Higher order schemes in time are the total variation diminishing (TVD) Runge-Kutta (RK) schemes, introduced in [123] and further developed in [124]. The schemes combine several sequential forward Euler steps effectively canceling low order error terms, thus yielding a higher order in time. Those TVD RK schemes avoid spurious oscillations as long as the underlying forward Euler scheme does not introduce them. However, the numerical benefit of schemes of order four or higher does not contribute significantly to the accuracy of practical simulation problems [18].

[2] This constitutes the main time loop.

For the discretization of the Hamiltonian several schemes have been developed: Lax-Friedrich [125], Godunov [126], and Roe-Fix with entropy correction [123]. They all have in common that they make use of a numerical Hamiltonian H' which includes artificial dissipation to damp spurious oscillations in the solution. This allows the basic forward Euler step to be stable.

The typically used Lax-Friedrich scheme computes the artificial dissipation globally giving high dissipation, but often lead to smoothed solutions. High dissipation is not desirable, because the level-set method should not affect the process model, e.g., smoothed structures. To counter the high dissipation, schemes which compute the dissipation locally, like the Local-Lax-Friedrich scheme [127], were developed. A recently developed scheme for anisotropic etching process TCAD simulation considers more grid points than the the Local-Lax-Friedrich scheme to determine the artificial dissipation [115, 112].

The numerical solution of the forward Euler method (4.9) is explicit, thus the limit on the size of the time step is only given by the Courant-Friedrichs-Lewy (CFL) condition

$$\Delta_t < \frac{\Delta_{\vec{x}}}{\max \left| \vec{V} \right|}, \tag{4.22}$$

where the maximum over the computational domain is chosen and with $\Delta_{\vec{x}}$ a measure for the grid resolution, e.g., the maximum of the grid resolution in all spatial dimensions. Often the signed-distance property of the level-set function is lost during the advection [128]. This is often the result of a *bad* velocity function (i.e., a velocity not fulfilling (4.19)) or the result of accumulated numerical errors.

4.2.6 Re-Distancing

Re-Distancing restores the signed-distance property of a level-set function. This is necessary because the signed-distance property is distorted by the previous computational step, the *Advection*. *Re-Distancing* avoids numerical issues arising from steep and flat gradients of Φ (cf. Section 4.1.2).

The approaches considered here to compute the signed-distance stem from an generalized mathematical problem which is known as the Eikonal equation [129]

$$\left| \nabla \Phi \left(\vec{x} \right) \right| = F \left(\vec{x} \right) \quad \vec{x} \in \Omega, \tag{4.23}$$

$$\Phi \left(\vec{x} \right) = G \left(\vec{x} \right) \quad \vec{x} \in \Gamma. \tag{4.24}$$

The Eikonal equation describes a wave front emerging from Γ and marching through Ω. The wave speed is given by $\frac{1}{F(\vec{x})}$, which has to be positive to be well-defined (negative or zero wave speed would not allow the wave front to reach the entire Ω).

The solution to Φ in this context describes the shortest travel time for the wave emerging from Γ. The initial values given by G on Γ determine the departure time from Γ. Early departures are given by $G < 0$ and late departures by $G > 0$.

If the wave speed is uniformly equal to one and the departure time is zero, the travel time is equal to the traveled distance. Thus, for *Re-Distancing* the PDE

$$|\nabla \Phi(\vec{x})| = 1 \quad \vec{x} \in \Omega, \tag{4.25}$$

$$\Phi(\vec{x}) = 0 \quad \vec{x} \in \Gamma, \tag{4.26}$$

is solved. Note, this would just compute the distance field (not signed-distance as both sides of the interface are positive), thus the computed values on Ω^- have to be multiplied by -1 to get the signed-distance function. The algorithmic details, the implementation details, and the developed advanced parallelization strategies are presented in Chapter 6.

4.2.7 Re-Gridding

Re-Gridding adapts the blocks on the hierarchical grid to fit the new interface position (regions requiring fine spatial resolution). *Re-Gridding* is split into two sub-steps:

- Flagging of the regions which need a higher spatial resolution.
- Clustering those regions (covering with blocks) in order to create the hierarchical grid.

Figure 4.8 shows the interface displacement by the advection and the two sub-steps of *Re-Gridding*.

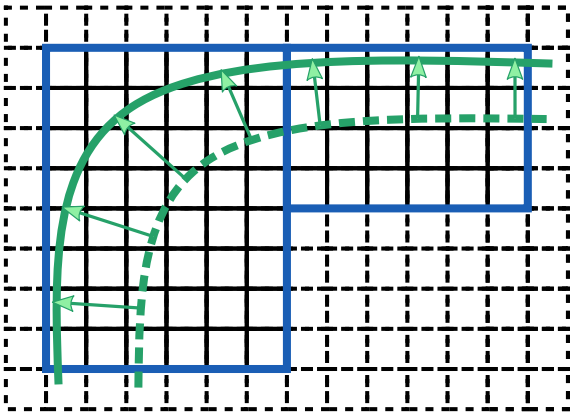
Flagging

The flagging sub-step selects grid points based on the distance to the interface (approximated by their level-set value Φ), the difference of normals on grid points to normals on neighboring grid points (detecting regions with curvature, i.e., corners and edges), and distance to other level-set functions present in the simulation domain (detection of material borders and *triple-junctions*). Flagging of regions with high curvature is particularly important, because the maximum representable curvature is indirectly proportional to the spatial resolution [18]. This is essential because the often present sharp corners in process TCAD simulations would otherwise become artificially rounded.

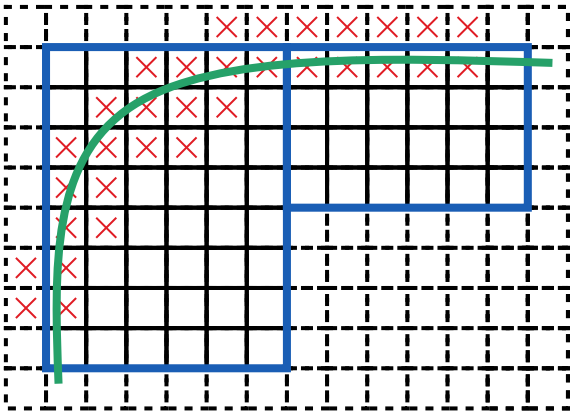
Clustering

Clustering is important to efficiently transform the flagged points to actual blocks of the hierarchical grid. The main challenges are to create as few blocks as possible containing the least amount of grid points, whilst still covering all flagged grid points. The run-time of a step in the main time loop almost linearly depends on the number of discretized points, thus minimizing them is important for the overall run-time efficiency of the overall simulation.

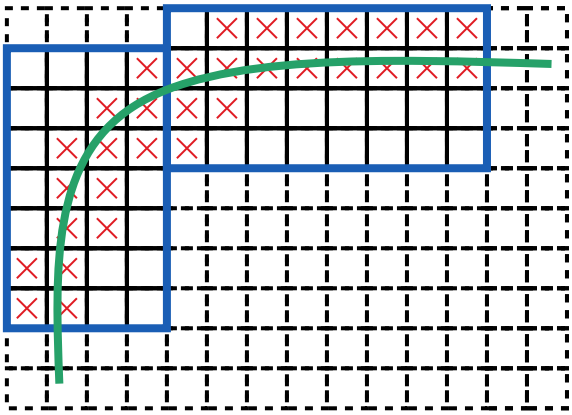
This concludes the computational steps of the main time loop, which are repeated until the desired end time of the simulation is reached. After all time steps are done the simulation results are ready to be transferred to the next process step of the manufacturing of the device.



(a) Advection



(b) Flagging



(c) Clustering

Figure 4.8: The *Re-Gridding* step is schematically shown for two blocks on a single resolution grid (blue rectangles) placed near a corner of the interface (green curve). The interface is moved by the advection step from the dashed to the solid green curve (cf. Figure 4.8a). The blocks of the hierarchical grid have to be adapted to fit the new interface position. Next, grid points are flagged (red crosses), which shall be available for the next simulation step in the desired spatial resolution (cf. Figure 4.8b). Finally, the flagged points are clustered (grouped together) to be covered efficiently by blocks (cf. Figure 4.8c).

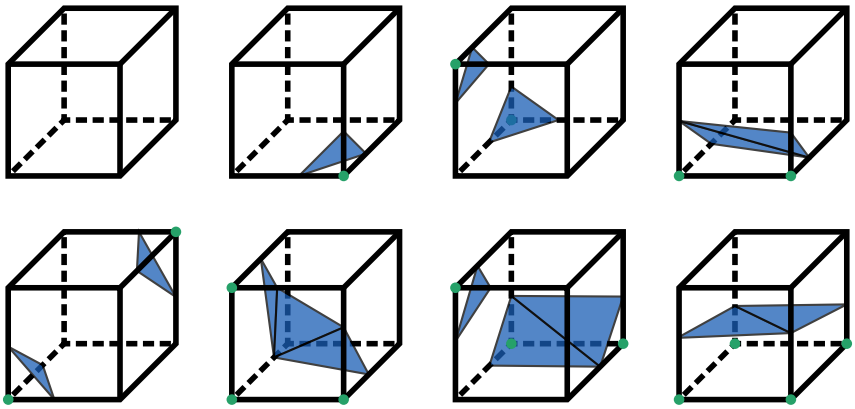


Figure 4.9: Selected templates for the marching cubes algorithm. The corners with opposing signs are marked by a green dot and the corresponding polygon template by the blue triangles.

This could be starting a new time loop with a different process model or in case the device manufacturing is completed the topography is extracted for the subsequent device TCAD simulation. This latter step requires the interface to be extracted, as is discussed in the following.

4.2.8 Interface Extraction

Device TCAD simulations require an explicit description of the materials regions, thus the implicit interfaces are extracted to an explicit description of the material regions. The preferred explicit representation of the material regions for device TCAD simulations is a polygonal surface representation (e.g., triangle mesh). The polygonal representation is reached by computing an explicit interface from the implicit representation of the level-set function and then use Boolean operations to restore the material regions from the additive level-sets. The dominant algorithm for explicit interface extraction is the marching cubes algorithm [130] in three dimensions, the analog algorithm in two dimensions is called marching squares. Other approaches are for example the cut cell method [131], which is conceptually similar to the marching cubes algorithm, but is tailored towards fluid dynamic simulations. Because this thesis is not focused on this computational step the reader is referred to [132] for an comprehensive overview of advancements to the base marching cube algorithm presented in the following.

Starting from a Cartesian grid, chunks of eight neighbor points (corners of a cube) are used to determine the polygonal representation of the interface passing through the cube. The sign of the points determines which of the 2^8 possible polygon templates is chosen. In Figure 4.9 eight different sign combinations with their corresponding polygon templates are shown. The number of templates is reduced through symmetry exploitations (reflection, rotation, and mirror) to a minimum of 14 cases. If not all symmetries are exploited the number of cases is higher. The explicit vertices of the polygons are given by the zero-crossings along the edges of the cube (they are identical to the *Cross Points* introduced earlier).

In a last step, the polygons of all chunks are merged together forming the polygonal representation of the full interface. The so-created polygonal surface representations are often of *poor quality* (e.g., triangles with high aspect ratios, neighboring triangles with significant different diameters).

Therefore, the initially marching-cubes generated explicit polygonal surface representations are typically optimized and consequently volume-meshed to enable the full-range of subsequent device TCAD simulations (e.g., simulations of charge carrier densities under bias conditions inside the device and at device contacts) [133].

In the next and final section of this chapter, an overview of software tools is given, which implement level-set based process TCAD simulations.

4.3 Software

The advantages of the level-set method in tracking topography evolution lead to the development of several commercial and open source software tools, referred to as process TCAD simulators. They utilize the level-set method for 3D process TCAD simulations on the feature scale. In the following, the two simulation tools are shortly introduced. Note that other tools exist as well, e.g., Sentaurus Process [134], but are not further discussed as they are out of scope.

Victory Process is a commercial process TCAD simulator developed by Silvaco [135]. The simulator allows to create a *digital twin* of electronic device manufacturing processes, such as etching, deposition, and oxidation. Victory Process uses a level-set engine based on hierarchical grids, allowing to represent different parts of material regions with varying spatial resolution. Details on the hierarchical grid data structure are presented in Chapter 2. Victory Process is written in C++ and pThreads and OpenMP are used for parallelization. The underlying simulation framework of Victory Process is the basis for the developed algorithms presented in Chapter 5, Chapter 6 and Chapter 7.

ViennaTS is a process TCAD simulator developed by the Institute for Microelectronics at the TU Wien, focusing on processing challenges for micro- and nanoelectronics [136]. It uses the level-set method on a hierachical run length encoding (HRLE) [99] data structure in combination with a sparse field approach. The open source topography simulator is written in C++ and uses OpenMP for parallelization.

Chapter 5

Parallel Velocity Extension

This chapter presents the general details of the computational step *Velocity Extension* and, in particular, the developed algorithmic advances enabling efficient parallel execution on hierarchical grids.

Velocity Extension extends the given interface velocity (originating from a process model) from the *Cross Points* (cf. Section 4.2.3, crossings of the grid lines with the interface) to all grid points of the computational domain. Depending on the process model the interface velocity may be a scalar or vector field. The extension is necessary for the level-set equation to be well defined, because a velocity field is required (due to the Eulerian formulation) in the entire computational domain.

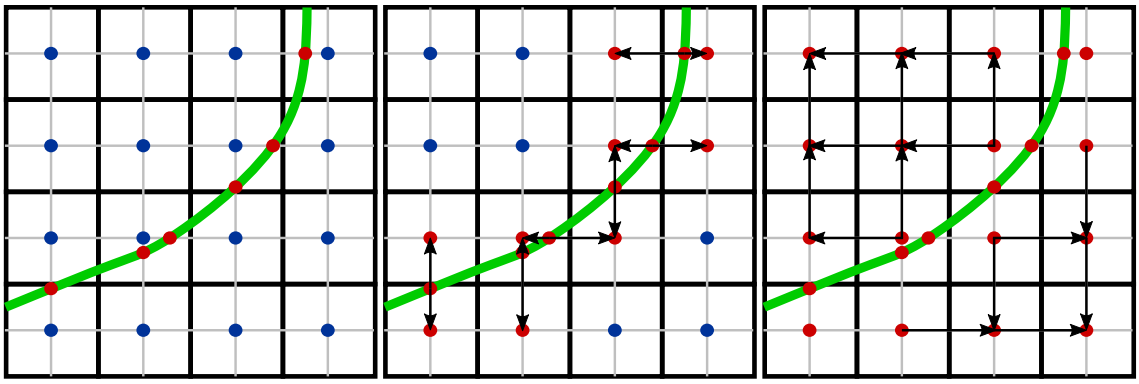
First, the analytic requirements to the extended velocity field are precisely defined, which is followed by a literature review of the previous approaches (Section 5.1). The *Velocity Extension* typically consist of two phases:

The first phase is the extension from the *Cross Points* to the *Close Points* (Section 5.2). *Cross Points* are located directly at the interface and do not belong to the grid. Figure 5.1a schematically shows the *Cross Points* on which the velocity is given. *Close Points* are grid points which have a neighboring grid point with a different sign, thus the interface is next to them and as well as at least one *Cross Point*. Figure 5.1b shows the extension of the velocity from the *Cross Points* (along the arrows) to the *Close Points*. This phase is computationally negligible and straightforward parallelizable. The second phase is the extension to the remaining grid points (cf. Figure 5.1c), which is the computationally most intense part.

The extension in the second phase is based on the FMM; the fundamentals of the FMM are presented in detail in Section 5.3. The bottleneck of the FMM is its usage of a heap data structure to sort grid points by their distance to the interface. The sorting determines the order in which the velocity is extended to the grid points. The usage of a heap data structure is inherently serial, because only a single point (the one with the smallest distance) is available for computation.

The core contribution presented in this chapter is to overcome this bottleneck and parallelize the FMM for *Velocity Extension*. To that end, three advancements were developed and evaluated [55, 137, 138]:

Advancement 1: Through an interpretation of the order of computations in the context of graph theory, it is possible to relax the strict sorting employed by the FMM.



(a) Given velocity (red dots on interface, i.e., green line)

(b) Extension to *Close Points*

(c) Extension to remaining grid points

Figure 5.1: The velocity is given on the *Cross Points* **(a)**. In the first phase the velocity is extended to the *Close Points* (i.e., red grid points next to cross points) **(b)** and finally the velocity is extended to the remaining grid points **(c)**.

This enables the usage of alternative data structures compared to the inherently serial heap employed by the FMM (Section 5.4).

Advancement 2: The change of the data structure allows for a straightforward parallelization of the computations on a Cartesian grid (Section 5.5). The proposed parallelization minimizes the number of explicit synchronization constructs, which favors parallelism. This approach introduces limited redundant computations, i.e., velocity computed on a grid point multiple times. However, the redundancy has negligible negative impact on the parallel performance.

Advancement 3: Finally, the algorithm is tailored to hierarchical grids by a load-balancing approach (Section 5.6). The goal is to reduce global synchronization barriers necessary in the exchange of data between blocks and considerations due to data locality allowing better cache reuse. The different stages of the development of the new algorithm are shown to specifically highlight the advances made and discuss their impact. The performance is evaluated on two representative process TCAD simulations (Section 5.7).

First, an ion beam etching (IBE) simulation, which is part of the fabrication process of a novel device used in memory technology[1] is considered (Section 5.7.1). In this example the scalar velocity extension on a Cartesian grid is analyzed by comparing the run-time of the velocity extension for three different data structures. The parallelization is evaluated by measuring the parallel speedup and the ratio of redundant computations (due to less explicit synchronization constructs).

Second, the thermal oxidation simulation from the motivational example in Section 1.1 is used for analysis, because the process models require the extension of a scalar and a vector velocity (Section 5.7.2).

[1]The spin-transfer torque magnetoresistive random access memory (STT-MRAM) uses magnetism for permanent storage (without requiring constant supply of electricity, prevalent in current random access memory devices) of information.

This simulation fully utilizes a hierarchical grid and, therefore, is suited to evaluate the proposed load-balancing and reduced number of global synchronization barriers. The evaluation is performed by measuring the run-time and parallel speedup of the velocity extension step.

5.1 General Ideas

In a semiconductor process TCAD simulation the process model typically provides the interface propagation velocity only for points on the interface. As previously discussed, these points are computationally captured with *Cross Points*. However, in order to advect the interface the velocities are necessary in the entire computation domain (on the actual grid points). The formal mathematical requirement to an extended velocity field is that it is continuous in regions close to the interface, i.e.:

$$\lim_{\vec{x}\to\Gamma} V = V_I. \tag{5.1}$$

The continuity is achieved in practice by assigning each point the velocity of the closest point on the interface. The velocity extension equation

$$\nabla\Phi\left(\vec{x},t\right)\cdot\nabla V\left(\vec{x},t\right)=0, \tag{4.19 revisited}$$

is used to describe this. In case of a signed-distance field Φ it is equivalent to constantly extend the velocity along interface normal vectors. Additionally, an extension according to (4.19) avoids distortions of the signed-distance field (still distortions occur due to numeric errors). This fact was first proven in [122] and follows:

$$\begin{aligned}\frac{d|\nabla\Phi\left(\vec{x},t\right)|^2}{dt} &= \frac{d}{dt}\left(\nabla\Phi\left(\vec{x},t\right)\cdot\nabla\Phi\left(\vec{x},t\right)\right)\\ &= 2\nabla\Phi\left(\vec{x},t\right)\cdot\frac{d}{dt}\nabla\Phi\left(\vec{x},t\right)\\ &= -2\nabla\Phi\left(\vec{x},t\right)\cdot\nabla V|\nabla\Phi\left(\vec{x},t\right)| - 2\nabla\Phi\left(\vec{x},t\right)\cdot\nabla|\nabla\Phi\left(\vec{x},t\right)|V.\end{aligned}$$

For a signed-distance function ($|\nabla\Phi\left(\vec{x},t=0\right)| = 1$), Φ stays a signed-distance function. The changes to the last line result from swapping the time and spatial derivative and using the level-set equation (4.9). The first term is zero because of the choice for the extended velocity (4.19). The second term is zero due to starting with a signed-distance function (gradient of a constant function is zero). In case of a vector-valued velocity field (4.19) is solved for each component of the vector velocity as in the scalar case.

The investigation starts with a review of approaches to solve the velocity extension problem.

Literature Review

There are several strategies to solve the velocity extension problem, considering only the basic case of a Cartesian grid.

The first attempt was made in [122] using the FMM which is based on Dijkstra's algorithm [139]. The FMM traverses the grid points of the computational domain using a priority queue in ascending order visiting every grid point exactly once. The computational complexity of this approach is $\mathcal{O}(n \log(n))$ with n the number of grid points in the computational domain, due to the necessary sorting of the priority queue (cf. Section 5.3).

In [140] the approach is further developed to yield higher accuracy for characteristic curves[2] at the cost of higher computational load. The higher accuracy is especially important, if the velocity field shall be used for several time steps of a simulation (reducing the number of evaluations of the process model), i.e., evaluating the process model only every third time step. This is not used in the presented level-set simulations, because the underlying velocities from the process model can change significantly between two consecutive time steps.

In [141] an approach based on the fast scanning method is introduced. It iteratively computes the velocity on all grid points of the computational domain using several predefined stencil configurations. The run-time complexity is $\mathcal{O}(n)$ with n the number of grid points in the computational domain. A drawback is that the approach visits every grid point of the computational domain 2^d times, with d the number of spatial dimensions.

The velocity extension presented in [142] allows for faster convergence for shape optimization simulations based on the level-set method, i.e., minimizing the volume of a cantilever but maintaining a certain resistance to deformation. This approach is not applicable to process TCAD simulations, because it requires an already established velocity field for one side of the interface: In process TCAD simulations, the velocity field is only available at the *Cross Points* directly at the interface.

An approach based on the biharmonic expansion is presented in [143], but is not applicable because an already established velocity field for one side of the interface is required. The same issue applies to the extrapolation approach based on fast sweeping methods presented in [144].

In [145] an extension based on solving local Riemann problems is proposed. This approach achieves a higher simulation accuracy compared to an approach based on the FMM. For small time steps the asymptotic limit of the extension based on local Riemann problems and the FMM are the same. Thus the simulation results are also the same.

However, approaches for efficient parallelization of the velocity extension are missing, in particular when considering hierarchical grids.

5.2 Extension from *Cross Points* to *Close Points*

The first step for all reviewed velocity extension algorithms (which are applicable to the process TCAD simulation setting) is to extend the velocity from *Cross Points* to the *Close Points*.

[2] A curve describing the movement of a point driven by a PDE.

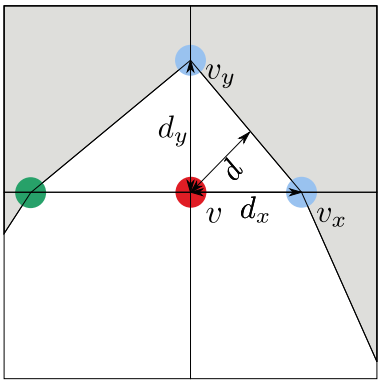


Figure 5.2: The velocity v of a *Close Point* (red) is computed using the velocities from the *Cross Points* (blue). The *Cross Point* (green) is ignored as a closer *Cross Point* in the x-dimension is available.

Let d_x, d_y, and d_z be the distances from a *Close Point* to the *Cross Points* in x-dimension, y-dimension, and z-dimension, respectively. The gradient of Φ computed at the *Close Point* via first-order finite differences to the *Cross Points* is given by

$$\left(\frac{d}{d_x}, \frac{d}{d_y}, \frac{d}{d_z}\right), \tag{5.2}$$

with d the distance to the interface of the *Close Point* (Φ value).

The velocity at the *Cross Points* is v_x, v_y, and v_z. Inserting the finite difference approximations to the gradients into the velocity extension equation (4.19) yields

$$\begin{aligned} 0 &= \left(\frac{v - v_x}{d_x}, \frac{v - v_y}{d_y}, \frac{v - v_z}{d_z}\right) \cdot \left(\frac{d}{d_x}, \frac{d}{d_y}, \frac{d}{d_z}\right) \\ &= d\left(\frac{v - v_x}{d_x^2} + \frac{v - v_y}{d_y^2} + \frac{v - v_z}{d_z^2}\right). \end{aligned}$$

Solving for v gives

$$v = \frac{d_y^2 d_z^2 v_x + d_x^2 d_z^2 v_y + d_x^2 d_y^2 v_z}{d_x^2 d_y^2 + d_x^2 d_z^2 + d_y^2 d_z^2}. \tag{5.3}$$

Thus the velocity on the *Close Points* is given as the weighted average of the velocity of the closest *Cross Point* in each spatial dimension.

In case no *Cross Point* is available for a specific spatial dimension (5.3) a lower dimensional computation is performed, e.g., if in z-dimension no *Cross Point* is available, the formula is simplified to

$$v = \frac{d_y^2 v_x + d_x^2 v_y}{d_x^2 + d_y^2}. \tag{5.4}$$

Figure 5.2 shows the variables denoted above for the 2D case.

If a *Cross Point* is only available in a single spatial dimension, e.g., x-dimension, no computation is necessary as the velocity from the *Cross Point* is directly assigned

$$v = v_x. \tag{5.5}$$

The distance d of the *Close Point* is not used in the computation of v and only the velocities from *Cross Points* are used. The velocity is computed independently for all *Close Points*, thus allowing for a straightforward parallelization of the extension to the *Close Points*.

Now that the velocity is available for all *Close Points*, the FMM is used to extend the velocity to the remaining grid points (computational domain). Due to the data dependencies between the grid points this is more complicated than the above discussed extension from *Cross Points* to *Close Point* and an ordering scheme is required.

5.3 Fast Marching Method

The FMM assigns each grid point of the computational domain one of three exclusive flags (states):

- *Known*: The grid point has its final velocity (value) assigned and no further updates are necessary.
- *Band*: The grid point has a temporary velocity (value) assigned (the velocity might be changed by a subsequent update).
- *Unknown*: The grid point has no velocity (value) assigned.

Based on these flags the FMM for the velocity extension is described as it is presented in [122]. First, a set of initial grid points is chosen around the interface (*Close Points*), and afterwards the velocity is extended to neighboring grid points by solving the discretized version of (4.19) using an upwind scheme. This allows to compute the solution in a from the interface outwards *marching* manner.

The computation of the velocity on a grid point is described in Algorithm 1. It is similar to the computation used for the extension from the *Cross Points* to the *Close Points*, but now all involved points are part of the grid. First, the upwind neighbors are determined in each spatial dimension (neighboring grid points with a smaller distance to the interface). The Φ values of the neighboring grid points are compared based on considering three cases: In case the *lower* neighboring grid point (neighboring grid point with the smaller index) is selected (cf. Algorithm 1 Line: 5), in case the *higher* neighboring grid point (neighboring grid point with a larger index) is selected (cf. Algorithm 1 Line: 9), and in case none is selected the contribution from this spatial dimension is zero, subsequently ignored. The selected neighbors are then used in the approximation to the gradient by computing the forward/backward differences. Finally, the weighted average of the velocities is computed (Algorithm 1 Line: 14) and the grid point is flagged *Band* (Algorithm 1 Line: 15).

The computation of the velocity on a grid point is illustrated by a small example.

Example

In order to illustrate the update of a grid point P with its index ijk, the following specific configuration is considered.

Algorithm 1: Procedure to update (compute) the velocity on the given grid point P. First, the upwind neighbors of P are determined and then the weighted average of their velocities is computed and the flag is set to *Band*.

procedure `Update(`P`):`
  `/* Determine upwind neighbors of` P `*/`
  for $i \in \{x, y, z\}$ **do**
    $D_i \leftarrow 0$
    $V_i \leftarrow 0$
    if $P^{-i}.\Phi < P^{+i}.\Phi$ *and* $P^{-i}.\Phi < P.\Phi$ **then**
      $D_i \leftarrow \frac{P.\Phi - P^{-i}.\Phi}{\Delta_i^2}$
      $V_i \leftarrow P^{-i}.V$
    end if
    if $P^{-i}.\Phi > P^{+i}.\Phi$ *and* $P^{+i}.\Phi < P.\Phi$ **then**
      $D_i \leftarrow \frac{P.\Phi - P^{+i}.\Phi}{\Delta_i^2}$
      $V_i \leftarrow P^{+i}.V$
    end if
  end for
  `/* Compute velocity for` P `according to the upwind scheme */`
  $P.V \leftarrow \frac{\sum_{i \in \{x,y,z\}} V_i D_i}{\sum_{i \in \{x,y,z\}} D_i}$
  $P.\text{flag} \leftarrow \mathit{Band}$
end procedure

In x-dimension the lower neighboring grid point (P^{-x} with index $i-1jk$) is closer to the interface, as well as in the y-dimension (P^{-y} with index $ij-1k$), but in the z-dimension none of the neighboring grid points is closer to the interface, thus the used first-order upwind scheme is given by

$$0 = \left(\frac{V_{i-1jk} - V_{ijk}}{\Delta_x}, \frac{V_{ij-1k} - V_{ijk}}{\Delta_y}, 0 \right) \cdot \left(\frac{\Phi_{i-1jk} - \Phi_{ijk}}{\Delta_x}, \frac{\Phi_{ij-1k} - \Phi_{ijk}}{\Delta_y}, 0 \right). \tag{5.6}$$

Solving for the velocity in this case yields

$$V_{ijk} = \frac{V_{i-1jk} \frac{\Phi_{i-1jk} - \Phi_{ijk}}{\Delta_x^2} + V_{ij-1k} \frac{\Phi_{ij-1k} - \Phi_{ijk}}{\Delta_y^2}}{\frac{\Phi_{i-1jk} - \Phi_{ijk}}{\Delta_x^2} + \frac{\Phi_{ij-1k} - \Phi_{ijk}}{\Delta_y^2}}. \tag{5.7}$$

In case the grid resolution is the same for all spatial dimensions $\Delta_x = \Delta_y = \Delta_z$, the dependency of the solution on the grid resolution vanishes yielding

$$V_{ijk} = \frac{V_{i-1jk}(\Phi_{i-1jk} - \Phi_{ijk}) + V_{ij-1k}(\Phi_{ij-1k} - \Phi_{ijk})}{\Phi_{i-1jk} - \Phi_{ijk} + \Phi_{ij-1k} - \Phi_{ijk}}. \tag{5.8}$$

The detailed discussion of the FMM as shown in Algorithm 2 follows.

Algorithm 2: The FMM used on a Cartesian grid for the velocity extension.

```
1  procedure FastMarchingMethod():
       /* Initialization - Grid points and Band  */
2      foreach grid point [P] do                    // Initialize all grid points
3          P.flag ← Unknown
4      end foreach
5      foreach Close Points [CP] do
6          CP.V ← Velocity computed based on the Cross Points
7          CP.flag ← Known
8      end foreach
9      foreach Close Points [CP] do                 // Setup the priority queue
10         foreach CP.Neighbors [N] do
11             if N.flag = Unknown then
12                 Update(N)
13             end if
14         end foreach
15     end foreach
       /* Marching - Extend to the computational domain */
16     while Band ≠ ∅ do
17         P ← min Band                             // Main challenge for parallelization
18         P.flag ← Known
19         foreach P.Neighbors [N] do
20             if N.flag = Unknown then
21                 Update(N)
22             end if
23         end foreach
24     end while
25 end procedure
```

FMM Algorithm

The FMM is initialized by first flagging all grid points *Unknown* (Algorithm 2 Line: 3) and then the velocity on all the *Close Points* is computed and they are flagged *Known* (Algorithm 2 Line: 7). The next step is to advance to all the neighboring grid points of the *Close Points* and compute their velocity and set their flag to *Band*. The computation of the velocity on a neighboring grid point is performed using Algorithm 1.

The key idea of the FMM is that from all the grid points flagged *Band* the one with the smallest distance to the interface is chosen and flagged *Known* (Algorithm 2 Line: 17). This guarantees that for the immediately following update of all neighboring grid points (Algorithm 2 Line: 21) the velocities on grid points used in Algorithm 1 are already computed beforehand.

The updated neighboring grid points are flagged *Band*, thus they are also considered when the next grid point is chosen to be flagged *Known*.

The step selecting the grid point flagged *Band* with the smallest distance to the interface is repeated until no more grid point is flagged *Band*, thus all grid points are flagged *Known*. The first part of Algorithm 2, in particular Lines: 2-15, is typically referred to as *initialization* of the FMM, because it sets the stage for the second part, the *marching*, in particular Algorithm 2 Lines: 16-24. The term *marching* originates from the Eikonal equation (Section 6.1), where the solution process corresponds to a front *marching* away from an interface.

The efficient selection of the minimum in Algorithm 2 Line: 17 is key to the performance of the FMM, which is discussed in the next section.

5.4 Data Structures

For an efficient selection of the grid point (also flagged *Band*) with the smallest distance to the interface all *Band* grid points are ordered by their distance to the interface, which is implemented by a priority queue. To efficiently sort the grid points in a priority queue a specialized data structure is employed. Typically a heap data structure (*Heap*) is used [122, 67, 146]. From the available heap data structures the binary heap (in form of a binary tree) is the most used implementation. The data structure has to provide the following functionality:

- `getFirst` shall return the *first* point (for the FMM the first point is the one with the smallest distance to the interface) and remove it from the data structure.
- `insert` shall insert a grid point into the data structure. In case of a *Heap* `insert` additionally verifies that no grid point is inserted twice, because, if a grid point is already present in the *Heap*, only the position within the heap is updated.

Figure 5.3 depicts the `insert` and `getFirst` operations. The `insert` operation appends the new element to the binary tree (last position). Afterwards, the element *bubbles up* (is swapped with its parent) until its current parent is smaller.

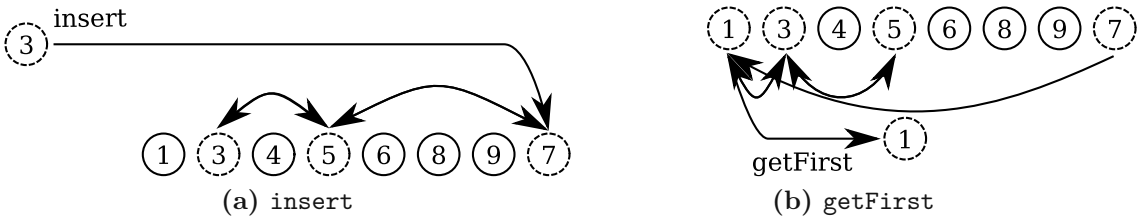


Figure 5.3: `insert` and `getFirst` operations shown for a *Heap* data structure. The number in the circles is the *key* (data responsible for the ordering, i.e., the distance to the interface) of an element. The operations require several swaps (arrows) to ensure the heap property, if an element is inserted or removed.

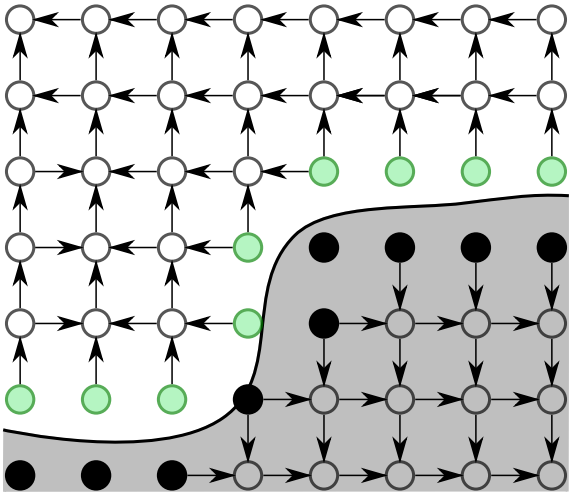


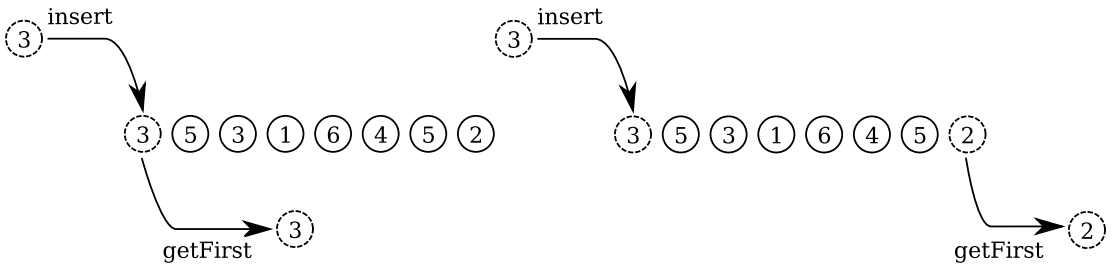
Figure 5.4: Graph interpretation of the velocity extension problem, circles are the nodes of the graph (corresponding to the grid points) and the arrows the directed edges. The interface (black curve) divides the graph into two disjunct partitions. Considering only one partition, the velocity values are *Known* on the *Close Points* (green). The remaining grid points require the velocity on all neighbors which have an arrow pointed to them to be computed.

The `getFirst` operation removes the smallest element and puts the last element in the binary tree on its position. Afterwards, the last element (now on the first place) *bubbles down* (is swapped with its children) until its current parent is smaller. Thus, both operations require the swapping of several other elements of the heap to ensure the correct ordering of the elements in the heap.

Inserting a grid point to the binary heap and removing the first grid point has the computational complexity of $\mathcal{O}(\log(n))$. Removing the first grid point also has the computational complexity of $\mathcal{O}(\log(n))$. In both cases several swaps are necessary to keep the heap in order. Consequently, the FMM has a computational complexity of $\mathcal{O}(n\log(n))$, because all n grid points of the computational domain have to be inserted into and removed from the binary heap. The usage of a priority queue in form of a *Heap* is counterproductive for parallelization, because only the single element with the smallest key (distance to the interface) is eligible for an update. The goal of the next paragraphs is to lay out the developed approach of relaxing the strict ordering enforced by the binary heap to (i) enable a computational complexity of $\mathcal{O}(n)$ and (ii) to enable parallelization of the algorithm.

Graph Theory

To relax the ordering enforced by the FMM, the problem is interpreted in the context of graph theory. Let $G(N, E)$ be an ordered graph, where the nodes N are given by all the grid points and the edges E are given by the upwind neighbor relationship. Figure 5.4 shows an exemplary graphical representation of such a graph. The graph is divided into two independent partitions through the interface (*inside* and *outside*). For simplicity's sake only one of those partitions is considered here, because the exact same algorithm is applied to the other partition.



(a) *Stack* **(b)** *Queue*

Figure 5.5: Simplicity of the `insert` and `getFirst` operations shown for a *Stack* and a *Queue*.

This enables a straightforward parallelization for up to two threads processing both sides of the interface simultaneously.

The order in which the nodes are able to be computed is equivalent to the topological sort problem, which is possible to be solved in linear $\mathcal{O}(|N| + |E|)$ time as shown in [147]. Topological sort of a directed graph is a linear ordering of its nodes such that every edge is directed the same, i.e., consider all nodes to be placed on a straight line and all edges point to the right. The topological order of a graph is not unique, if there is more than one source which is a node with no incoming edge, i.e., in the context of the velocity extension all the *Close Points* are sources.

The computational complexity is also linear in the number of grid points in the computational domain, because $n = |N|$ and the number of edges $|E|$ in the graph is limited by the used stencil to approximate the gradients. The typically used stencil contains the direct neighbors only, thus $|E| \leq 2d|N|$ with d the number of spatial dimensions. For example, in three spatial dimensions the seven-point stencil yields $|E| \leq 6|N|$. Comparing the newly developed approach which requires in the worst case only $2d$ visits of each grid point, to the fast scanning method approach which requires always 2^d visits, shows the advantage.

The topological sort problem is typically solved by a depth-first or breadth-first traversal of the graph [148]. A depth-first traversal moves along an edge to the next node before it returns to explore the other edges of a node. A breadth-first traversal explores first all edges of a node before it moves to the next node.

Those algorithms can be adapted to the FMM based approach, as they influence the ordering of the grid points in the *Band*. Using a *Stack* (first-in last-out queue, Figure 5.5a) corresponds to a depth-first and using a *Queue* (first-in first-out queue, Figure 5.5b) corresponds to a breadth-first traversal, respectively. The classic FMM uses a *Heap* (binary heap) data structure. Figure 5.5 shows graphically that `insert` and `getFirst` are less complex for the *Stack* and *Queue* compared to the *Heap* (cf. Figure 5.3). The binary heap data structure performs `insert` and `getFirst` operations in $\mathcal{O}(\log(n))$, whilst the *Stack* and *Queue* data structure perform those operations in $\mathcal{O}(1)$.

The necessary changes to the algorithms are presented and discussed in the next paragraphs.

Algorithm 3: First modification of the FMM (now just called `Velocity Extension`), which enables the usage of different data structures for the *Band*. The `Update` algorithm is also changed to use the modified algorithm `Update2`. The blue colored lines are added/modified compared to Algorithm 2.

```
1  procedure Velocity Extension():
       /* Initialization - Grid points and Band  */
2      foreach Points [P] do                    // Initialize all grid points
3          P.flag ← Unknown
4      end foreach
5      foreach Close Points [CP] do
6          CP.velocity ← Velocity computed based on the Cross Points
7          CP.flag ← Known
8      end foreach
9      foreach Close Points [CP] do                  // Setup the priority
        queue/queue/stack
10         foreach CP.Neighbors [N] do
11             if N.flag = Unknown then
12                 Update2(N)
13             end if
14         end foreach
15     end foreach
       /* Marching - Extend to the computational domain */
16     while Band ≠ ∅ do
17         P ← getFirst Band // first (instead of smallest) grid point
            is chosen
18         P.flag ← Known
19         foreach P.Neighbors [N] do
20             if N.flag = Unknown then
21                 Update2(N)
22             end if
23         end foreach
24     end while
25 end procedure
```

Algorithmic Changes

The adaptions necessary to the FMM are shown in Algorithm 3 and the changes to the `Update` algorithm are shown in Algorithm 4, which is extended compared to Algorithm 1. A check whether the grid point is already computed Algorithm 4 Line: 2 is added, because for a *Stack* and *Queue* a grid point might be inserted multiple times into the data structure of the *Band*. Also, a check whether all upwind neighbors have already a velocity assigned is necessary (Algorithm 4 Line: 9 and Algorithm 4 Line: 16).

Algorithm 4: Procedure to update (compute) the velocity on the given grid point P, with additional checks compared to Algorithm 1 (blue colored lines). The checks ensure that the velocity is computed on the upwind neighbors.

```
1  procedure Update2(P):
2      if P.flag != Unknown then                    // Skip if already computed
3          return
4      end if
       /* Determine upwind neighbors of P and their flag */
5      for i ∈ {x, y, z} do
6          D_i ← 0
7          V_i ← 0
8          if P^{-i}.Φ < P^{+i}.Φ and P^{-i}.Φ < P.Φ then
9              if P^{-i}.flag = Unknown then
10                 return                           // No update takes place
11             end if
12             D_i ← (P.Φ − P^{-i}.Φ) / Δ_i^2
13             V_i ← P^{-i}.V
14         end if
15         if P^{-i}.Φ > P^{+i}.Φ and P^{+i}.Φ < P.Φ then
16             if P^{+i}.flag = Unknown then
17                 return                           // No update takes place
18             end if
19             D_i ← (P.Φ − P^{+i}.Φ) / Δ_i^2
20             V_i ← P^{+i}.V
21         end if
22     end for
       /* Compute velocity for P according to the upwind scheme */
23     P.V ← (Σ_{i∈{x,y,z}} V_i D_i) / (Σ_{i∈{x,y,z}} D_i)
24     P.flag ← Band
25 end procedure
```

Only the ordering of the computations by the *Heap* guarantees that all upwind neighbors are computed beforehand. If not all upwind neighbors have already a velocity assigned (one of them is flagged *Unknown*) then the velocity is not computed. The grid point is again considered when the previously *Unknown* upwind neighbor is computed and, therefore, it is guaranteed that all grid points are computed when the algorithm terminates.

The impact of the different data structures on the performance is evaluated in Section 5.7.1, however, first the parallelization of the `Velocity Extension` is discussed.

5.5 Parallelization

The relaxation of the strict ordering of the FMM allows for a parallelization of the velocity extension algorithm, which is shown in Algorithm 5. The for-loops in Algorithm 5 Line: 2 and Line: 5 are straightforward parallelizable as all the iterations are independent.

A single data structure storing the grid points flagged *Band* is not viable due to the synchronization overhead, if all threads would operate on the same data structure. Thus, for every *Close Point* a dedicated data structure called work queue (WQ) is created (Algorithm 5 Line: 10). The WQ is a data structure that supports the `insert` and `getFirst` operations. As the WQs track all the grid points flagged *Band* the explicit label *Band* is not required anymore and only *Known* and *Unknown* are used. Therefore, the initialization of the *Band* grid points is modified and the *Close Points* are directly used for the WQ (Algorithm 5 Line: 11). The WQ data structure is exclusive (in OpenMP terms: *private*) to the executing thread. The block where all grid points (with their velocity and flag) are stored is shared between all the threads.

In principle, explicit synchronization between the threads would be necessary every time a grid point from the shared block is accessed. This is not required for correctness of the velocity extension, but to avoid redundant computations, i.e., two threads compute the velocity simultaneously for the same grid point. That being said, Algorithm 5 deliberately re-computes the velocities, as the computational overhead is negligible compared to an otherwise introduced synchronization overhead. Both threads compute the identical velocity as the upwind grid points and their velocity value used for the computations are the same. Access of a thread to a grid point is required to be an *atomic* operation for read and write operations. Atomicity is needed to ensure that values are read/written in a consistent manner. Otherwise, writing identical values by two threads to the same grid point might corrupt the data. The changes to Algorithm 5 also require changes to the update algorithm shown in the next paragraph.

Final Update Algorithm

Also the `Update2` algorithm is modified yielding the final version of the `Update3` algorithm Algorithm 6. The algorithm returns a Boolean indicating whether the update has been successful. The update is not successful, if the grid point is already computed Algorithm 6 Line: 2, or any upwind neighbor is not computed beforehand (Algorithm 6 Line: 9 and Algorithm 6 Line: 16). After the computation of the velocity an additional check is introduced whether the grid point has not been computed in the meanwhile by a different thread (Algorithm 6 Line: 25). This check is not explicitly synchronized with other threads, but reduces redundant computations, especially the otherwise following redundant insertion into the WQ is avoided. To completely avoid the redundant computations an explicit locking mechanism could be considered in principle, however, such an approach would seriously deteriorate the performance.

Algorithm 5: Parallelized velocity extension, by creating an independent WQ for each *Close Point*, allowing for parallel computations.

```
1  procedure Velocity Extension Parallel():
       /* Initialization - Grid points */
2      foreach Points [P] do                                   // In parallel
3          P.flag ← Unknown                          // Initialize all grid points
4      end foreach
5      foreach Close Points [CP] do                            // In parallel
6          CP.velocity ← Velocity computed based on the Cross Points
7          CP.flag ← Known
8      end foreach
       /* Marching - Extend to the computational domain */
9      foreach Close Points [CP] do                            // In parallel
10         create WQ
11         WQ.insert(CP)
12         while WQ ≠ ∅ do                  // Extend velocity to entire domain
13             P ← WQ.getFirst
14             foreach Neighbors [N] of P do
15                 if N.flag = Unknown then
16                     if Update3(N) then
17                         WQ.insert(N)
18                     end if
19                 end if
20             end foreach
21         end while
22     end foreach
23 end procedure
```

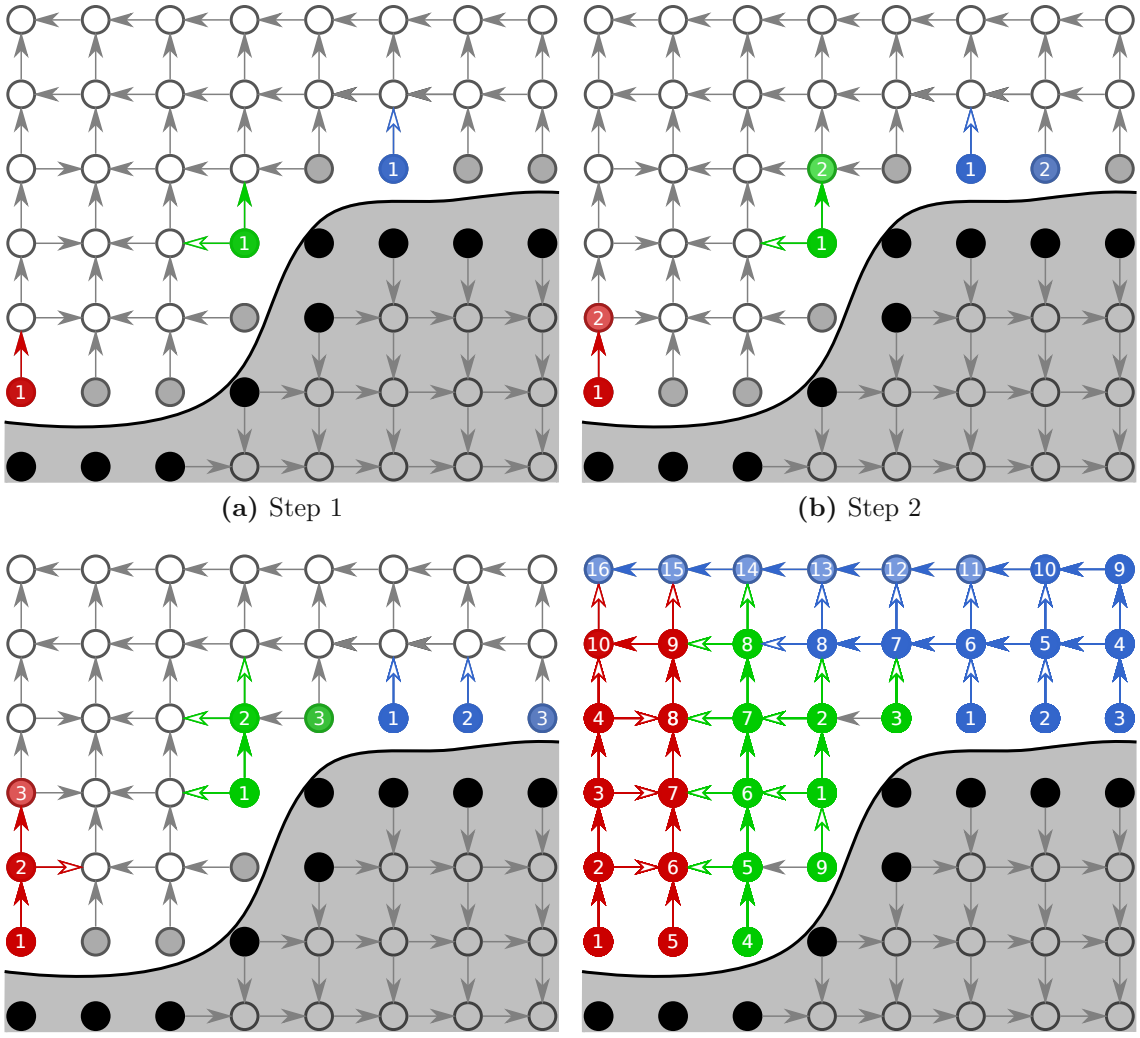
Only if the update is successful (Algorithm 6 returns true), the grid point is inserted into the WQ (Algorithm 5 Line: 17). The algorithm terminates if all WQs are empty, meaning all grid points have a velocity assigned.

Explanatory Example

To better illustrate the process of the newly developed algorithm an explanatory example is discussed in the following. In Figure 5.6 an example for a parallel execution of Algorithm 5 is shown using three threads. Three threads work in parallel on the grid. The work completed by a thread is shown by the corresponding color (red, green, blue). For this example it is assumed that each thread processes one node (grid point) per step and is able to handle all neighbors. The edges (arrows) to the neighbor are also colored using the thread color. The arrows are filled, if the neighbor is successfully computed and inserted into a thread's own WQ. In the first step, each thread has a single WQ containing a single *Close Point* assigned (marked with a white one in Figure 5.6a).

Algorithm 6: Final procedure to update (compute) the velocity on the given grid point P, with additional checks and a Boolean return value compared to Algorithm 4, to reduce the redundant computations introduced by the parallelization.

```
1  procedure Update3(P):
2      if P.flag != Unknown then                    // Computed by another thread
3          return false
4      end if
       /* Determine upwind neighbors of P and their flag */
5      for i ∈ {x, y, z} do
6          D_i ← 0
7          V_i ← 0
8          if P^{-i}.Φ < P^{+i}.Φ and P^{-i}.Φ < P.Φ then
9              if P^{-i}.flag = Unknown then
10                 return false                     // No update takes place
11             end if
12             D_i ← (P.Φ − P^{-i}.Φ) / Δ_i^2
13             V_i ← P^{-i}.V
14         end if
15         if P^{-i}.Φ > P^{+i}.Φ and P^{+i}.Φ < P.Φ then
16             if P^{+i}.flag = Unknown then
17                 return false                     // No update takes place
18             end if
19             D_i ← (P.Φ − P^{+i}.Φ) / Δ_i^2
20             V_i ← P^{+i}.V
21         end if
22     end for
       /* Compute velocity for P according upwind scheme */
23     P.V ← (Σ_{i∈{x,y,z}} V_i D_i) / (Σ_{i∈{x,y,z}} D_i)
24     P.flag ← Band
25     if P.flag != Unknown then                    Redundant computation
26         return false
27     else
28         P.V ← V
29         P.flag ← Known
30         return true
31     end if
32 end procedure
```



(a) Step 1 **(b)** Step 2

(c) Step 3 **(d)** Step 16

Figure 5.6: Exemplary computation order for the velocity extension using three threads in parallel using a heap. The nodes (grid points) are colored (red, green, and blue) by the thread which processed them. Edges (arrows) are filled, if an *Unknown* upwind neighbor prevents the computation. Adapted with permission from Quell *et al., Proc. Int. Conf. Simulation Semiconductor Processes Devices (SISPAD)* (2019), pp. 1-4 [137], © 2019 IEEE.

The number on a node indicates the step in which a node has been processed. If the WQ is empty a new WQ is assigned, by selecting a new *Close Point*, e.g., for the blue thread this happens at step two (Figure 5.6b) and for the green one at step three (Figure 5.6c).

Threads require different numbers of steps to finish, i.e., the green thread requires nine steps whilst the blue thread requires 16 (Figure 5.6d). The green thread is finished (out of work), because no more *Close Points* (WQs) are available to be processed. This is a typical load-imbalance problem. The cause of this load-imbalance is found in the structure of the given dependencies of the grid points. The longest dependency chain is 10 grid points long.

Algorithm 7: The velocity extension algorithm tailored towards a hierarchical grid, the changes colored in red are lines which are removed in comparison to the multi-block FMM.

```
1  procedure Extension Hierarchical():
2      setBoundaryConditionsOnLevel 0()
3      foreach Levels do                                  // From coarsest to finest
4          foreach Blocks on Level do                          // Parallel region
5              WQ ← InitialPoints                                  // Create task
6              Velocity Extension(Blocks,WQ)
7          end foreach
8          Wait                                           // Synchronization barrier
9          WQ ← Exchanged ghost points
10         while WQ ≠ ∅ do
11             foreach Blocks on Level do                      // Parallel region
12                 Velocity Extension(Blocks,WQ)                   // Create task
13             end foreach
14             Wait                                       // Synchronization barrier
15             WQ ← Exchanged ghost points
16         end while
17         setBoundaryConditionsOnNextLevel()
18     end foreach
19 end procedure
```

In practical applications, however, the load-imbalance is negligible due to the several orders of magnitude higher number of *Close Points* compared to the number of used threads.

The developed parallelization approach is evaluated in Section 5.7.1, on a Cartesian grid[3]. The next section discusses adaptions of the parallel velocity extension step to support hierarchical grids.

5.6 Hierarchical Grids

In the previous sections the velocity extension has been discussed within the context of a single resolution Cartesian grid. This section extends the algorithms to be used on hierarchical grids.

The hierarchical algorithm is given in Algorithm 7. The developed advancements relative to the previously developed multi-block FMM [62] are highlighted. Algorithm 7 operates in a top-down manner, starting the extension on *Level 0* (the coarsest level) and successively extending the velocity on finer (higher) levels. The algorithm also explicitly sets the boundary conditions (cf. Algorithm 7 Line: 2 and Algorithm 7 Line: 17) by involving ghost points.

[3] In the context of hierarchical grids, this is equivalent to a hierarchical grid containing only a single level, hence, also only a single block.

Previous presented algorithms operating on a Cartesian grid ignored the boundary conditions for reasons of a more accessible presentation, but in case of a hierarchical grid setting this it not possible. Ghost points are either set via linear interpolation of the velocity from a coarser level or, if they are covered by a block on the same level, by a non-ghost point of the respective block.

After the boundary conditions are set, for each block on a level a parallel OpenMP `task` (cf. Section 3.1) is created, allowing the computation of blocks on the same level in parallel (Algorithm 7 Line: 4). Each task creates a dedicated WQ using the *Queue* as underlying data structure. The union of the *Close Points* and ghost points for which the velocity is known (*InitialPoints*) is inserted into the WQ. Subsequently, Algorithm 8 is executed, which extends the velocity based on the given WQ to the corresponding block. After that a global synchronization barrier is enforced, waiting for all tasks to be finished before proceeding as otherwise some ghost points are possibly exchanged before their velocity is computed. Global synchronization barriers are detrimental to parallel performance, because they require all threads to synchronize. This causes threads to idle until the last thread reaches the synchronization barrier. Thus it is desirable to avoid global barriers as proposed in the next paragraphs.

Reducing Global Synchronization

The former multi-block FMM [62] algorithm uses a synchronized exchange step. The synchronized exchange step internally uses two global synchronization barriers. In between the global synchronization barriers the ghost points of all blocks are updated. The here proposed advanced algorithm does not need those synchronized exchange steps anymore because the functionality was transferred into Algorithm 8. This reduces the number of global synchronization barriers, except for one at the end of each processed level. If the velocity for a ghost point is changed by the synchronized exchange step, the ghost point is inserted into the WQ of the corresponding block (Algorithm 7 Line: 9). As long as there is a non-empty WQ, Algorithm 8 is executed again with the same synchronization barrier and synchronized exchange step.

A level of the hierarchical grid is finished after the ghost points on the next level are set (Algorithm 7 Line: 17), in which case the algorithm proceeds to the next level. When all levels are finished Algorithm 7 terminates.

Algorithm 8 is an adapted version from Algorithm 5 introducing two main enhancements which are discussed below:

- WQ splitting, enabling a better load-balancing.
- Localized exchange, reducing global synchronization barriers.

The initialization of the grid points (Algorithm 8 Lines: 2-8) is removed, because the grid points (WQ) from which the velocity shall be extended are provided as input to the algorithm.

Algorithm 8: Velocity extension on a block of hierarchical grid, with the added capability of load-balancing through splitting the WQ and to extend the velocity without synchronization barriers to the neighboring blocks. Red colored lines are removed and blue colored lines are added in comparison to Algorithm 5.

```
1  procedure Velocity Extension(Block, WQ):
2      foreach Points [P] do
3          P.flag ← Unknown                          // Initialize all grid points
4      end foreach
5      foreach Close Points [CP] do
6          CP.velocity ← Velocity computed based on the Cross Points
7          CP.flag ← Known
8      end foreach
9      while WQ ≠ ∅ do
10         if WQ.length > limit then
11             WQ1,WQ2 ← Split WQ
12             Velocity Extension(Block, WQ1)                    // Create Task
13             WQ ← WQ2
14         end if
15         P ← WQ.getFirst()
16         foreach Neighbors [N] of P do
17             if N.flag = Unknown then
18                 if Update3(N) then
19                     WQ.insert(N)
20                     if Overlap(N) then
                           /* EQ gathers overlapping grid points in one local
                              queue per neighboring block */
21                         EQ.insert(neighboring block, N)
22                     end if
23                 end if
24             end if
25         end foreach
26     end while
27     foreach neighboring block [NB] of Block do
28         Velocity Extension(NB, EQ(NB))                    // Create Task
29     end foreach
30 end procedure
```

WQ Splitting

The splitting of a WQ (Algorithm 8 Lines: 10-14) takes place, if a WQ exceeds a certain size. The size is chosen in order that the WQ and the required grid points fits into the cache of the used CPU, e.g., for the compute system ICS it is set to 512.

The computation requires for each grid point the velocity V and the signed-distance Φ (each a double requiring eight bytes) and the same values for the remaining six grid points in the seven-point stencil. However, half of the grid points in the stencil are shared by the grid points in the WQ, due to the WQs locality. Thus, the estimated working set is about

$$\underbrace{512}_{\text{WQ size}} \times \underbrace{8}_{\text{double size}} \times \underbrace{2}_{V \text{ and } \Phi} \times \underbrace{7}_{\text{stencil size}} \times \underbrace{\frac{1}{2}}_{\text{WQ locality}} = 28\,672 \text{ Byte} \tag{5.9}$$

which is less than 32KByte and thus fits into the L1d cache of the CPU used in the ICS. Additionally, the splitting reduces the load-imbalances between the threads as WQs may vastly differ in size. The split inserts the first halve into a new WQ, which is processed in a recursive execution of Algorithm 8 in parallel. The second halve remains in place and is processed by the current execution of Algorithm 8. Splitting the WQ in the proposed manner gives the WQs a better spatial locality (grid points in the queue share their neighboring grid points), compared to an approach where grid points are alternatingly inserted into the new WQs.

Localized Exchange

The second optimization is the localized data exchange from one block to another. This reduces the global synchronization barriers to a single one, which is necessary before continuing on the next level of a hierarchical grid). For the localized data exchange an additional check is made after a successful update of the velocity for a grid point. The check identifies whether the just updated grid point is a ghost point for one of the neighboring blocks (Algorithm 8 Line: 20). If so, the grid point is additionally inserted into a WQ called exchange queue (EQ). For each neighboring block such an EQ collects all the grid points which shall be exchanged (Algorithm 8 Line: 21). Once the WQ of the current block is empty the EQs are used in a recursive execution of Algorithm 8. The algorithm is applied on the neighboring block using the corresponding EQ (Algorithm 8 Line: 28). This allows to extend the velocity to the neighboring blocks without a global synchronization barrier.

Those recursive executions may also be performed in parallel. The goal behind collecting all the grid points belonging to a neighboring block is to reduce the overhead introduced by creating WQs which contain only a single grid point and forcing data locality, because grid points exchanged between blocks neighbor each other.

The performance impact of the developed algorithms (Algorithm 7 and Algorithm 8) is evaluated by considering a thermal oxidation example in Section 5.7.2. The following section starts with the benchmark examples to evaluate the presented algorithms used for *Velocity Extension*.

5.7 Benchmark Examples and Analyses

The performance of the individual variants of the developed velocity extension algorithm, in particular concerning their parallel efficiency, is evaluated based on a 3D example simulation of an IBE process for a STT-MRAM device and based on the thermal oxidation simulation presented in the introduction Section 1.1. The presented simulations were conducted using Silvaco's Victory Process simulator which was augmented with the new velocity extension algorithms for evaluation purposes.

In Table 5.1 the presented velocity extension algorithms are summarized, identifying the velocity extension algorithm, the utilized algorithm to compute the update on a grid point, which levels of a hierarchical grid are applicable to the algorithm, and a short comment of the algorithm.

Table 5.1: Summary of the presented algorithms used to extend the velocity.

Velocity Extension	Utilized Update	Level	Comment
Algorithm 2	Algorithm 1	0	standard FMM
Algorithm 3	Algorithm 4	0	serially optimized FMM
Algorithm 5	Algorithm 6	0	parallelized FMM
Algorithm 7	Algorithm 6	All	optimized multi-block FMM

5.7.1 STT-MRAM

The results presented in this section were published in [137, 138]. Recent devices proposed in the field of emerging memory technologies [149] particularly demand optimized nano-patterning to enable small feature sizes and high density memory cells in order to replace conventional CMOS-based random access memory (RAM) [150, 151, 152]. One of the key advantages of STT-MRAM memory cells is non-volatility [153]. STT-MRAM memory cells consist of a magnetic tunnel junction (MTJ) which in turn consist of a ferromagnetic layer with a reference magnetization, a thin insulating barrier layer, and the ferromagnetic storage layer where the magnetization is variable [154]. The functionality of the device is driven by two physical phenomena: 1) the tunneling magnetoresistance (TMR) effect for reading and 2) the spin-transfer torque (STT) effect for writing [155]. The latter phenomena is responsible for the name of the device.

The most critical step in fabricating STT-MRAM devices is the creation of an array of MTJ pillars. Those pillars are created by an IBE process which transfers the pattern of the mask onto the underlying layers. In the IBE process ions are accelerated to high velocities, hitting the interface and mechanically sputtering material of the exposed surface. The corresponding process model uses a visibility calculation of the interface to the ion source plane which is positioned on the top of the simulation domain (above the structure). The IBE process is very anisotropic, resulting in significantly larger vertical etch rates than lateral etch rates. The velocities computed by the process model is a scalar field and is only well-defined at the surface of the structure (interface).

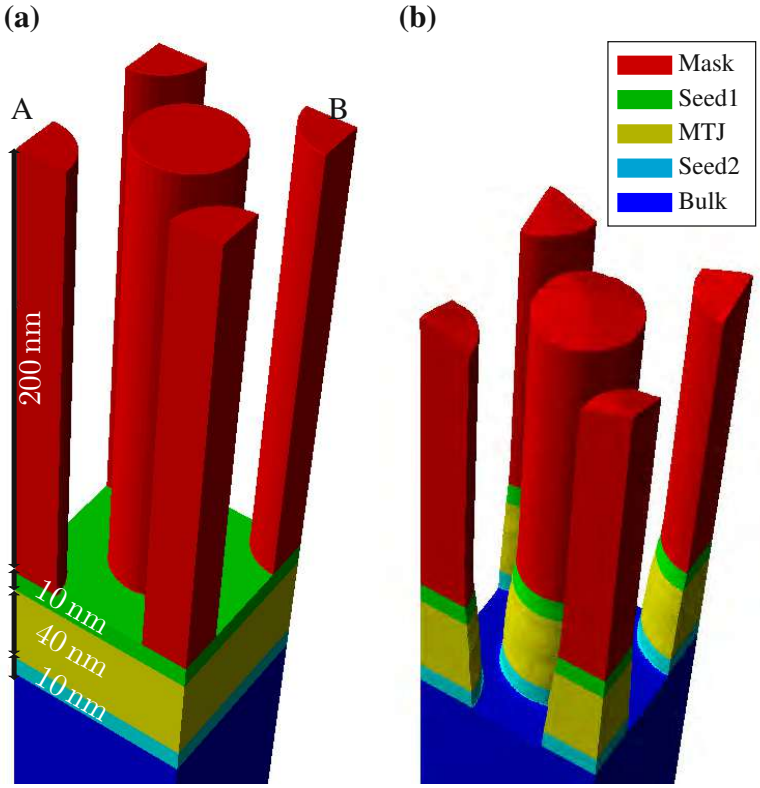


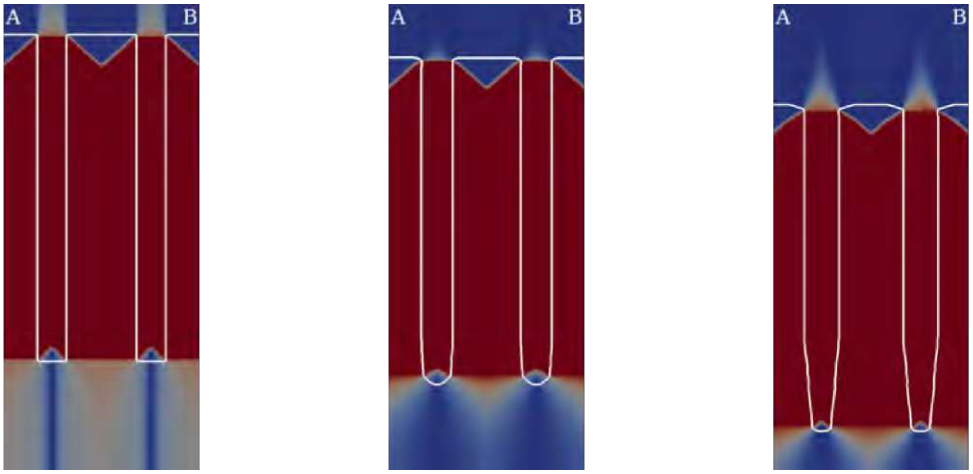
Figure 5.7: On the left **(a)** the initial structure topography of the STT-MRAM device is shown, consisting of the MTJ layer and the two corresponding seed layers, all on top of the bulk silicon. The right **(b)** shows the final pillars after the IBE process. Adapted with permission from Quell *et al., Proc. Int. Conf. Simulation Semiconductor Processes Devices (SISPAD)* (2019), pp. 1-4 [137], © 2019 IEEE.

The proposed velocity extension algorithms Algorithm 3 and Algorithm 5 are evaluated by simulating the discussed IBE process [156, 157].

Simulation Parameters

The considered simulation domain is 80 nm × 80 nm. The initial structure topography consists (from bottom to top) of bulk silicon, followed by a seed layer (combination of materials, e.g., hafnium and ruthenium, to protect the MTJ from degradation in following process steps [158]) with a thickness of 10 nm, the MTJ layer with a thickness of 20 nm, and a different seed layer with a thickness of 10 nm. At the top is the patterned mask layer with a thickness of 200 nm. The structure topography before and after the 6 min IBE process is shown in Figure 5.7.

The here discussed STT-MRAM example is used to evaluate the developed velocity extension algorithms on a Cartesian grid to show the fundamental capabilities without considering hierarchical grids. In the setting of hierarchical grids this setup is equivalent to the hierarchical grid containing only *Level 0*. The simulation is carried out in two different spatial resolutions: The *low resolution case* using a grid resolution of 2 nm and the *high resolution case* using a grid resolution of 0.5 nm, allowing to investigate performance behavior for varying loads.



(a) $t = 0\,\text{min}$ **(b)** $t = 3\,\text{min}$ **(c)** $t = 6\,\text{min}$

Figure 5.8: Extended velocity shown on a vertical slice (through A and B, cf. Figure 5.7) for different times of the simulation. High etch rates are given by blue colors, whereas red colors indicate a low etch rate. The interface representing the surface of the structure is drawn with a white curve. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Lecture Notes in Computer Science* 12043 (2020), pp. 348–358. [138], © 2020, under exclusive license to Springer Nature Switzerland AG.

Table 5.2 summarizes the properties of the resulting discretization. The interface geometry with its combination of flat, convex, and concave interface regions, leading to shocks and rarefaction fans in the extended velocity field is a challenging and representative test case for the developed velocity extension algorithms.

Table 5.2: Properties of the discretization for different resolutions for the example STT-MRAM device geometry (cf. Fig 5.7).

	Resolution	# grid points	# *Close Points*
Low Resolution Case	$40 \times 40 \times 700$	1 235 200	26 168
High Resolution Case	$160 \times 160 \times 2800$	73 523 200	411 896

In Figure 5.8 a slice of the extended velocity field is shown for three different representative time steps during the simulation. The interface (surface of the structure) is shown by the white curve. The extended velocity is constant along the normals on the interface. The blue colors indicate a high etch rate, whereas red colors indicate a low etch rate (nearly zero). The high etch rates are mostly present in interface regions of the structure which face the *top* of the simulation domain which is the source of the ions. The IBE is highly anisotropic, thus at the corners of the MTJ pillars, the interface velocity abruptly changes, but are still well resolved. The rarefaction fans (continuous extension of the velocity field in concave areas of the interface) and the shocks (discontinuous extension of the velocity field in convex areas of the interface) are clearly visible.

As a first step, the serial performance of the proposed algorithms and data structures is evaluated, followed by the evaluation of the parallel performance.

Serial Performance Evaluation

The compute system VSC3 is used for the evaluation of this simulation example. The first benchmark compares (cf. Table 5.3) the serial run-time of the algorithms for both resolutions. The other metric shown in the table is the ratio of how often at least a single upwind neighbor is in the state *Unknown* compared to the total number of executions to the update algorithm (*Un.* up). This metric is a measure for optimal traversal. A traversal of all grid points is optimal, if each grid point is visited exactly once, yielding a ratio of zero. A ratio of 0.5 corresponds to visiting every grid point twice.

Algorithm 2 achieves the optimum (the dependencies to compute the velocity are always fulfilled). However, a disadvantage of the metric is that it neglects effects on the performance caused by access times and cache misses, thus the run-times do not correlate with the optimal traversal metric. Comparing the run-time for Algorithm 3 using a *Heap* and *Queue* data structure shows that although they have a similar ratio of *Unknown* upwind neighbors their run-times are vastly different. This shows the superiority of the less complex `insert` and `getFirst` operations of the *Queue* compared to the *Heap*. The *Stack* has the highest ratio of *Unknown* upwind neighbors, because the *Stack* implements a depth-first traversal, which often selects grid points further away from the interface first.

Table 5.3: Serial run-time (Run-Time) in seconds and the ratio of how often *Unknown* upwind neighbors (*Un.* Up.) were encountered compared to the total updates. Bold numbers indicate the fastest run-time for each resolution. The reference Algorithm 2 strictly requires a *Heap* to be correct (compute consistently all grid points), thus no results are obtained for the other data structures.

(a) *Low Resolution Case*

	Algorithm 2		Algorithm 3		Algorithm 5	
Data Structure	Run-Time	*Un.* up.	Run-Time	*Un.* up.	Run-Time	*Un.* up.
Heap	0.265	0.0	0.258	0.034	0.190	0.262
Stack			0.196	0.418	0.200	0.416
Queue			**0.162**	0.077	0.177	0.259

(b) *High Resolution Case*

	Algorithm 2		Algorithm 3		Algorithm 5	
Data Structure	Run-Time	*Un.* Up.	Run-Time	*Un.* Up.	Run-Time	*Un.* Up.
Heap	19.99	0.0	19.14	0.076	13.27	0.241
Stack			13.27	0.414	12.83	0.412
Queue			**10.27**	0.052	11.67	0.221

The run-times of Algorithm 2 are at least 1.3 times slower compared to Algorithm 3 and Algorithm 5, if considering a *Stack* or *Queue*. The shortest run-time is achieved using Algorithm 3 using a *Queue* data structure, yielding a serial speedup of 1.6 and 2.0 for the *low* and *high resolution case*, respectively. The *high resolution case* takes about 64 times longer than the *low resolution case*, which is approximately the same factor as for the number of grid points those cases differ.

Thus a linear dependence of the run-time relative to the number of grid points is shown.

The difference between Algorithm 3 and Algorithm 5, using the *Stack*, is a reversed order of the grid points in the *Band* due to the stacks depth first traversal. The run-time is barely affected, because the ratio of *Unknown* upwind neighbors is about the same. For the *Heap* switching to Algorithm 5 is beneficial for the run-time, as it reduces the heap size manifesting in `insert` and `getFirst` operations to require a smaller share in run-time. The run-time of the *Queue* suffers from switching to Algorithm 5 as the ratio of *Unknown* upwind neighbors increases by a factor of four. The data structure used for the WQ (*Band*) in Algorithm 5 is less important compared to Algorithm 3, as the size of the *Band* is small. In Algorithm 5 the *Band* starts with a single grid point, compared to Algorithm 3 the *Band* contains about halve the *Close Points* (the other halve is used on the other side of the interface).

Parallel Performance Evaluation

The parallelization is evaluated for Algorithm 5 on one node of VSC3 for up to 16 threads utilizing all available physical cores[4]. Thread-pinning was used to avoid thread migration. The data is averaged over 10 iterations.

Figure 5.9 shows the run-time and parallel speedup of Algorithm 5. The shortest run-times are obtained for eight and 16 threads for the *low resolution case* and *high resolution case*, respectively. In both cases the *Queue* performed best, due to its serial superiority. Starting from two threads the serial-superior algorithm Algorithm 3 using a *Queue* is outperformed by the parallel algorithm regardless of the used data structure.

The highest parallel speedup (not lowest run-time) is achieved using the *Heap*, because using more threads further decreases the WQ size. Small WQ sizes are essential for the *Heap* as the `insertion` scales with the WQ size (*Stack* and *Queue* do not have this drawback).

The parallel speedup for the *low resolution case* using eight threads is 4.6 for the heap and the queue and 4.9 for the stack. For eight threads, the *high resolution case* has a parallel speedup of 4.5 for the stack, 5.3 for the queue, and 5.4 for the heap.

Using more than eight threads, requires the threads to be distributed over two memory domains (NUMA effects), leading to an increased run-time for the *low resolution case* (parallel speedup of 4.0) and only marginal speedup for the *high resolution case* (parallel speedup of 5.9) for all data structures. The memory is solely allocated by the first thread which resides on core 0 (part of the first processor), thus every thread running on the second processor has to indirectly access the memory. Additionally, threads running on different processors do not have shared caches further limiting speedup.

In Figure 5.10 the ratio of *redundant* computations and the ratio of *Unknown* upwind neighbors are shown.

[4]Evaluations considering simultaneous multithreading showed no noticeable speedup and have thus been excluded from presentation.

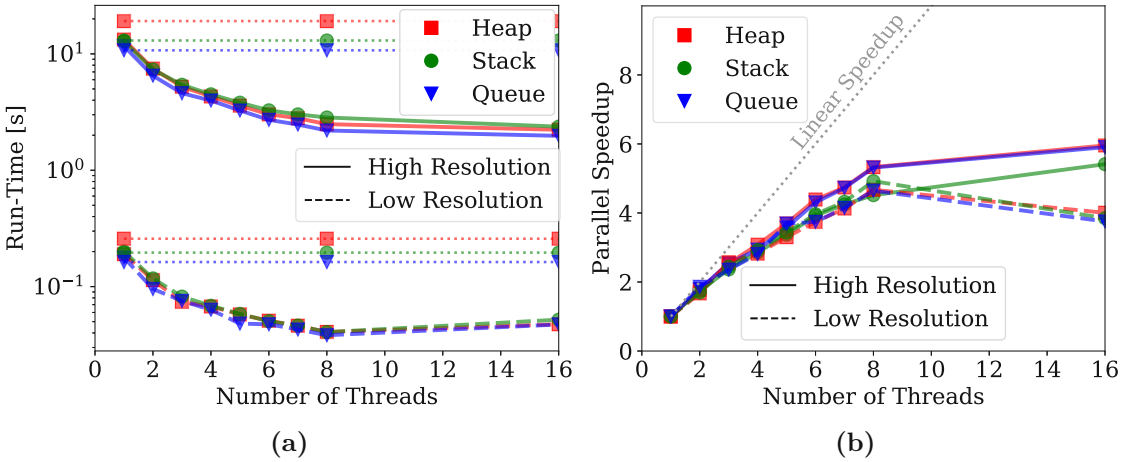


Figure 5.9: Run-time **(a)** and parallel speedup **(b)** of Algorithm 5. For reference, the serial run-time for Algorithm 3 is shown by dotted lines. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Lecture Notes in Computer Science* 12043 (2020), pp. 348–358. [138], © 2020, under exclusive license to Springer Nature Switzerland AG.

The ratio of *redundant* computations is investigated because Algorithm 5 does not use explicit synchronization between threads. Consequently, there might be cases where a grid point is computed more than once. As already mentioned in Section 5.5, the involved threads compute the same values. Considering two threads the ratio of redundant computations is below 0.01 %. For an increasing number of threads the ratio of redundant computations saturates, i.e., in the *low resolution case* below 1 % and *high resolution case* below 0.1 %. The ratio in the *low resolution case* is higher than in the *high resolution case*, as the number of grid points computed by a thread compared to the grid points where threads might interfere grow with different rates. A similar situation is found for the ratio between the volume and the surface of a sphere (square-cube law).

The ratio of *Unknown* upwind neighbors shown in Figure 5.10b declines slowly for increasing number of threads. That is because, if more threads are available, one of them might compute the *Unknown* upwind neighbors beforehand. The *Queue* has in the *high resolution case* a lower rate than the *Heap*, because the WQ has a better spatial locality.

Directly comparing the run-time of the serial execution of the FMM (Algorithm 2) to the best parallel execution of Algorithm 5 using the *Queue* shows that the run-time is reduced from 0.265 s to 0.038 s. The run-time reduction is attributed to a serial speedup of 1.5 and a parallel speedup of 5.6 for eight threads in the *low resolution case*. The *high resolution case* shows a reduction of the run-time from 19.99 s to 1.975 s utilizing all 16 threads. The serial speedup's contribution is a factor of 1.7 and the parallel speedup's contribution is a factor of 10.1.

In conclusion, for serial execution Algorithm 3 is the best choice and for parallel execution Algorithm 5 is the best choice. In both cases the *Queue* is superior to the other evaluated data structures, thus for the next benchmark example only the *Queue* is considered.

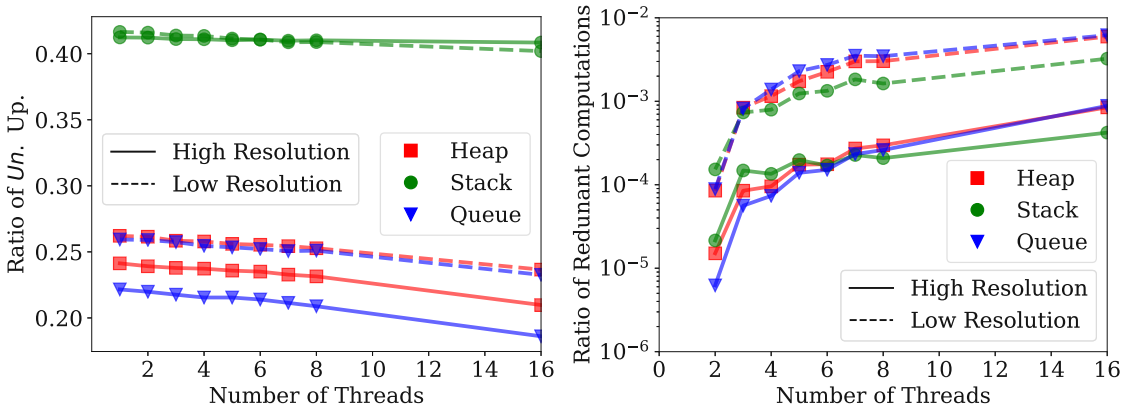


Figure 5.10: Ratio of *Unknown* upwind neighbors **(a)** and ratio of redundant computations **(b)** for different thread numbers. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Lecture Notes in Computer Science* 12043 (2020), pp. 348–358. [138], © 2020, under exclusive license to Springer Nature Switzerland AG.

5.7.2 Thermal Oxidation

The results presented in this section were published in [55]. The developed velocity extension algorithm optimized for the full hierarchical grid (cf. Algorithm 7) is evaluated using the thermal oxidation example discussed in Section 1.1. The process model for the oxidation step consists of two physical problems [15]: (1) The transport and reaction of the oxygen (diffusion); and (2) the volume expansion due to the chemical conversion, silicon to silicon dioxide, which is accompanied by the material flow (displacement) of all materials above the reactive material (reaction). Each of the physical problems yields a velocity field at the interface. A particular challenge in this case is that one scalar and one vector velocity field has to be considered. This scenario requires two separate velocity extensions and specialized advection schemes for each extended velocity field.

The first physical problem (oxidant diffusing through the oxide), is mathematically described by the Poisson equation

$$\frac{\partial}{\partial x_i}\left(D\frac{\partial C}{\partial x_i}\right) = 0, \tag{5.10}$$

$$-D\frac{\partial C}{\partial \vec{n}}\bigg|_{\mathrm{Si/SiO_2}} = kC, -D\frac{\partial C}{\partial \vec{n}}\bigg|_{\mathrm{SiO_2/Si}} = h(C_0 - C), \tag{5.11}$$

with C the oxidant concentration, D the diffusion coefficient, k the reaction rate, h the gas-phase mass-transfer coefficient, C_0 the equilibrium concentration in the oxide, and $\vec{n}$ the normal to the corresponding material interface. This gives a reaction rate at the silicon interface which is ultimately transformed to a scalar velocity field v at the *Cross Points* of the gas interface (surface of the structure).

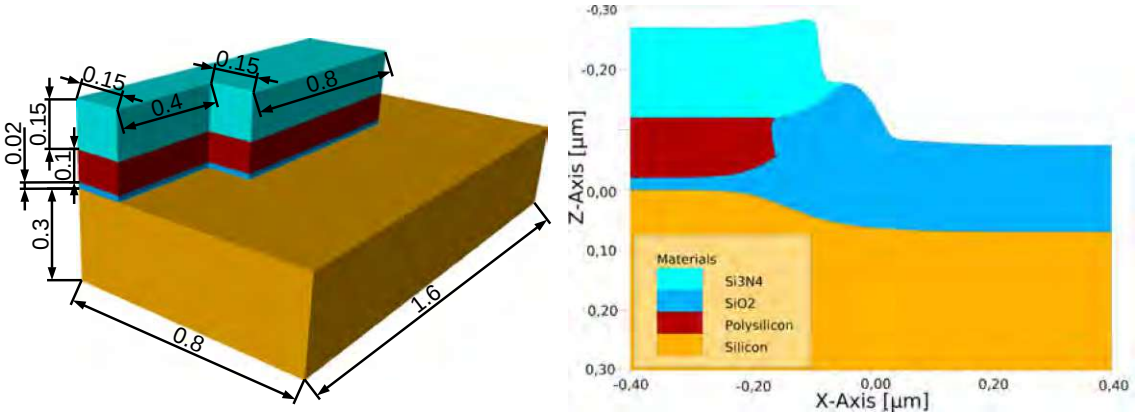


Figure 5.11: Material regions representing the initial structure topography of the thermal oxidation example on the left. On the right, a slice of the final material regions is shown including the manifested bird's beak. All lengths are given in µm.

The second physical problem (volume expansion from the chemical reaction and displacement of materials) is mathematically described by a creeping flow

$$\frac{\partial S_{ij}}{\partial x_i} = 0, \tag{5.12}$$

with $S_{ij} = -p \cdot \delta_{ij} + \sigma_{ij}$ denoting the Cauchy stress tensor, p the pressure, and δ_{ij} the Kronecker delta. The shear tensor σ_{ij} uses the Maxwell visco-elastic fluid model which, combined with further simplifications, yields the system of Stokes equations

$$\mu \Delta \vec{v} = \nabla p, \tag{5.13}$$

$$\nabla \cdot \vec{v} = 0, \tag{5.14}$$

with $\vec{v}$ the vector velocity field and μ the dynamic viscosity. The vector velocity field $\vec{v}$ is the second velocity field which needs an extension for this simulation.

Simulation Parameters

The simulation is performed on a rectilinear simulation domain with symmetric boundary conditions, representing a unit domain of the full wafer. The simulation domain covers a volume of 1.6 µm × 0.8 µm× 1.0 µm. The material stack consists from bottom to top of: 0.3 µm bulk silicon, 0.02 µm padding silicon dioxide, 0.1 µm buffer polysilicon, and 0.15 µm hard mask silicon nitride. Figure 5.11 shows the detailed topography before the thermal oxidation step and a slice of the aftermath of the 15 min thermal oxidation step at a temperature of 1000 °C. The bird's beak formed by the silicon oxide between the silicon and the polysilicon is well resolved.

The single block covering *Level 0* contains $40 \times 80 \times 40 \mathrel{\hat{=}} 128\,000$ grid points. The data of the hierarchical grid on *Level 1* is shown in Table 5.4. During the simulation which requires a total of 27 time steps, every third time step the hierarchical grid is re-gridded to fit the current topography of the structure. The number of blocks varies between 17 and 34, with the general trend that the number goes up due to an increased complexity of the material regions.

Table 5.4: Evolution of the number of blocks and total number of grid points on *Level 1* of the hierarchical grid shown for all time steps.

Time Step	Number of Blocks	Number of Grid Points
0	18	536 704
3	22	492 544
6	18	503 808
9	20	620 544
12	17	566 592
15	28	683 840
18	18	774 016
21	30	764 992
24	33	868 928
27	34	969 536

The number of blocks is important for the parallelization, because the unmodified algorithm Algorithm 7 (using red parts) and the corresponding Algorithm 8 (not using advances highlighted in blue) are directly limited by the number of blocks.

The number of grid points on *Level 1* varies (between 492 544 at the beginning and 969 536 grid points at the end of the simulation). There is no direct correlation between the number of blocks and number of grid points. The general trend for more grid points during the simulation, caused by the increased complexity of the material regions, also holds for the number of grid points.

Performance Evaluation

In Figure 5.12 the run-time and parallel speedup are shown for the velocity extension. The run-times were obtained using the ICS compute system utilizing up to 10 cores. The run-times labeled *new* are based on using the new Algorithm 7 with all its changes once for the scalar velocity case and once for the vector velocity case.

The run-times labeled *Original* are obtained from previous versions of the velocity extension. For scalar and vector velocity cases different different algorithms are used as base line. In particular *Original Vector Velocity Extension* uses the original FMM (Algorithm 2). It employs a single global priority queue (implemented via a heap) for all blocks on a level of the hierarchical grid. This does not allow for any parallelization. The minor variations in the run-time for different number of threads is created by noise, e.g., originating from the operating system.

The *Original ScalarVelocity Extension* uses Algorithm 7 without the proposed changes and optimization for the hierarchical grid presented in Section 5.6. Thus, the global *Heap* is replaced by the *Queue* on a per block basis. The proposed algorithm has 4 % slower serial run-time, because the performance is impacted by the modified exchange step and for a single-threaded execution global synchronization barriers are irrelevant.

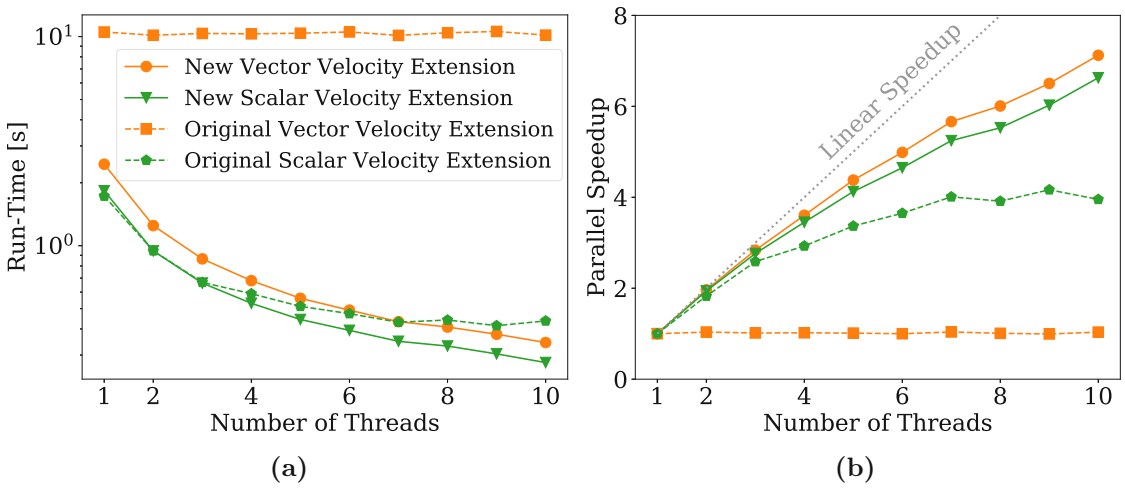


Figure 5.12: Averaged (over all 27 time steps) run-time **(a)** and parallel speedup **(b)** of Algorithm 7 for the extension of the vector and scalar velocity field. Adapted with permission from Quell *et al., IEEE Transactions on Electron Devices* 68.11 (2021), pp. 5430–5437 [55], © 2021 IEEE.

The parallel speedup for the scalar velocity extension maxes out at 6.6 for 10 threads. Comparing this to the original implementation which only reached a parallel speedup of 4.1, shows an improvement of 60 %. The parallel speedup of the original implementation starts to saturate at four threads, because the implicit load-balancing from the utilized thread pool deteriorates. The deterioration is caused by the limited number of tasks which is directly related to the number of blocks on a level, e.g., 17 blocks and 10 threads mean that three threads will get only a single block to process thus balancing the different workloads per block is not possible.

The new vector velocity extension reaches a parallel speedup of 7.1 for 10 threads. This is an improvement compared to the scalar case, which is caused by the three times higher computational load at each grid point, because the vector velocity has three components in a 3D simulation. The serial run-times for the scalar and vector velocity extension differ only by 43 % instead of the 200 % (each component of the vector adds 100 % run-time) expected from a three times as high workload per grid point. This indicates that most of the run-time is spent on checks and ordering of the grid points (memory intensive) rather than velocity computations. A further extension of the proposed algorithms is to extend the scalar and vector velocity together, if the process model allows it.

5.8 Summary

In this chapter the computational step *Velocity Extension* has been discussed in detail. First the extension from the *Cross Points* to the *Close Points*, which is trivially parallelizable, was presented. Then the original FMM used for the velocity extension was presented, because it is the basis and reference for the developed algorithmic advancements.

The key challenge in parallelizing the FMM for the velocity extension was to overcome the use of a heap data structure to determine the order in which the velocity is extended to the grid points of a block.

Based on an interpretation of the FMM through graph theory a new approach was developed where different data structures were employable to determine the order of the extension. The algorithmic changes were evaluated on a Cartesian grid using a single thread, where a serial speedup ranging from 1.6 to 2.0 for the velocity extension was measured. In particular, three different data structures *Heap*, *Stack*, and *Queue* were evaluated: The *Queue* performed best.

The changes to the data structure enabled a parallelization of the algorithm on a Cartesian grid. The parallelization was performed without any explicit synchronization constructs, thus redundant computations (points are computed multiple times) manifested. However, redundant computations count less than 1 % for 16 threads. Depending on the spatial resolution parallel speedups ranging from 4.9 to 5.3 using eight threads were achieved. Overall, a parallel speedup of 5.9 for 16 threads was achieved, further improvements being limited by NUMA effects.

Finally the proposed algorithm was adapted to a hierarchical grid, by reducing global synchronization barriers present in the original algorithm. The global synchronization barriers were reduced by developing a specialized exchange mechanism between blocks, which does not require additional synchronization barriers. This allowed for a parallel speedup of 7.1 for 10 threads, which is 60 % higher compared to the original algorithm. A direct comparison between the extension of a scalar and a vector velocity field showed that most of the run-time is spent on the checks and ordering and not on the computation of velocities.

Chapter 6

Load-Balanced Parallel Re-Distancing

This chapter presents the details of the computational step *Re-Distancing* and the developed load-balanced parallel FMM approach. *Re-Distancing* restores the signed-distance property of the level-set function without altering the interface position. Aside from discussing other approaches, the focus is on introducing the new block decomposition for the FMM which allows for superior parallel efficiency compared to previous approaches.

In principle, there are three strategies to compute the signed-distance function relative to a given level-set function:

- Re-initialization, uses the level-set equation with a velocity field, which converges to a signed-distance function.
- Direct computation of the signed-distance, which is based on finding the closest point on the interface.
- Eikonal equation, considers the computation of the signed-distance as a special case of the more general Eikonal equation.

The first two approaches are briefly discussed below, however, the focus is on solving the Eikonal equation (see Section 6.1), because the FMM for which the block decomposition was developed belongs to the class of Eikonal equation solvers.

Previous shared-memory parallelized approaches using the FMM suffer from load-imbalances, if the ratio of blocks (on a level of a hierarchical grid) per thread is low, e.g., below 10 [62]. Therefore, the core contribution presented in this chapter and published in [159], is a block decomposition approach to enable load-balancing. The developed block decomposition approach temporarily increases the number of blocks on all levels of a hierarchical grid (Section 6.2).

In the last section, the developed block decomposition approach is evaluated via a parameter study on the granularity of the decomposition and frequency of the data exchange steps between blocks (Section 6.3). For the evaluation a generic test case (a point source) for Eikonal solvers (Section 6.3.1) is considered, as well as, two representative example interfaces stemming from process TCAD simulations (Section 6.3.2 and Section 6.3.3).

The discussion continues with an overview of the strategies to compute the signed-distance.

Re-initialization

Re-initialization strategies use the level-set equation itself, employing a specific velocity function which ultimately leads to

$$\frac{\partial \Phi}{\partial t} = \operatorname{sgn}\left(\Phi^0\right)\left(1 - |\nabla \Phi|\right), \tag{6.1}$$

$$\Phi\left(\vec{x}, 0\right) = \Phi^0\left(\vec{x}\right), \tag{6.2}$$

where Φ^0 is the given distorted level-set function and Φ is the desired signed-distance function. The sgn term forces that the sign of the level-set function does not change during the re-initialization, i.e., *inside* stays *inside* and *outside* stays *outside*.

Equation (6.1) is solved numerically to a steady state (advanced for some time steps until the difference between two consecutive solutions Φ is below a threshold), yielding the signed-distance function Φ [160]. The signed-distance property of Φ follows directly from (6.1) by using $\frac{\partial \Phi}{\partial t} = 0$ (due to the steady sate) and moving $|\nabla \Phi|$ to the left hand side (compare (4.25)).

The drawback of this approach is that the interface is typically moved during the process; this leads to smoothening of sharp features, e.g., corners, and is obviously counterproductive in a process TCAD simulation setting as critical device features would deteriorate. The movement of the interface is dependent on the number of iterations necessary to reach the steady state, where a higher number of iterations results in a bigger deviation of the interface position. The number of required iterations increase the further the initial Φ^0 deviates from a signed-distance function.

Improvements to this method were made by modifying the used velocity [161, 162, 163]. These advancements yield the same steady state solution, but allow for a faster convergence to the steady state, as well as a smaller disturbance of the interface position.

Direct Computation of the Signed-Distance

The methods belonging to this strategy compute the signed-distance by calculating the closest point on the interface, i.e., minimizing the distance to the interface. The methods are similar to a gradient descent algorithm [164]. The closest interface point is computed by first using the gradient and the level-set value for an educated guess of the interface location. The quality of the educated guess is high, if the given level-set function is close to a signed-distance function.

The educated guess is then refined by directional optimization [165]. Conceptually similar methods based on the Hopf-Lax formula are presented in [166, 167].

Direct computation of the distance methods are easy parallelizable, because the computations on different grid points are independent [168]. The main disadvantage, however, is that the level-set function has to be close to a signed-distance function for optimal efficiency. Achieving high accuracy at corners is challenging, because the interface normals are ill-defined at those locations. The accuracy of methods belonging to this class is bound by the interpolation scheme used by the directional optimization to determine the exact interface position.

6.1 Eikonal Equation

The Eikonal equation is developed to describe a wave front emerging from Γ and marching through Ω

$$|\nabla \Phi(\vec{x})| = F(\vec{x}), \quad \vec{x} \in \Omega, \qquad \text{(4.23 revisited)}$$
$$\Phi(\vec{x}) = G(\vec{x}), \quad \vec{x} \in \Gamma, \qquad \text{(4.24 revisited)}$$

with the given wave speed $\frac{1}{F(\vec{x})}$ and departure time G. Using a wave speed of one and a zero departure time the distance to the interface is computed.

The Eikonal equation has applications in various areas of science and engineering, such as seismic processing [169, 170, 171], path-finding [172, 173, 146], and 3D imaging [106]. Therefore, there are several available computational approaches to solve the Eikonal equation. A short overview of the approaches to solve the Eikonal equation on Cartesian grids is given in [146][1]. The most widely-used solution approaches are (further details provided in the following):

- Fast sweeping method (FSM)
- Fast iterative method (FIM)
- FMM

The methods use a set of seed points (representing the discretized interface Γ on the grid). For *Re-Distancing* those seed points are the same as the *Close Points* (cf. Section 5.2). It is possible to also re-compute the signed-distance on the *Close Points* increasing the accuracy of the solution, but doing so alters the interface position [162]. The change of the interface position is strictly unwanted (see previous reasoning concerning the preservation of the process TCAD critical geometrical features of the to-be-simulated devices), thus *Close Points* are never modified.

All methods to solve the Eikonal equation have an `Update` algorithm (cf. Algorithm 9) in common, which solves the discretized Eikonal equation

$$\begin{bmatrix} \max\left(D_{ijk}^{-x}\Phi, D_{ijk}^{+x}\Phi, 0\right)^2 + \\ \max\left(D_{ijk}^{-y}\Phi, D_{ijk}^{+y}\Phi, 0\right)^2 + \\ \max\left(D_{ijk}^{-z}\Phi, D_{ijk}^{+z}\Phi, 0\right)^2 \end{bmatrix}^{\frac{1}{2}} = F_{ijk}, \qquad (6.3)$$

with $D_{ijk}^{+x}\Phi$ the forward difference approximation and $D_{ijk}^{-x}\Phi$ the backward difference approximation to the spatial derivative $\frac{\partial \Phi}{\partial x}$. The discretization by (6.3) is establishing an upwind scheme, because it uses in each spatial dimension only grid points with lower Φ values (Algorithm 9 Line: 5). The forward and backward differences are typically computed using a first-order scheme (cf. Section 2.1), which requires a stencil containing direct neighboring grid points (grid points for which the sum over all spatial dimensions of the absolute index differences is equal to one).

[1] The methods to solve the Eikonal equation on unstructured grids are conceptually similar to the ones discussed here, but due to the restriction of this thesis to structured grids the reader is referred to [174, 175, 176] for implementation details.

Algorithm 9: The algorithm solves the discretized Eikonal equation on a grid point. Its parameters are the P grid point on which it shall be solved, N the neighboring grid points, which shall be considered, and F the discretized speed.

```
1  procedure Update(P, N, F):
       /* Collect upwind neighbors */
2      for i ∈ {x, y, z} do
3          T_i ← 0
4          h_i ← 0
5          if P.Φ > N_i.Φ then
6              T_x ← N_x.Φ
7              h_x ← Δ_x
8          end if
9      end for
       /* Solve quadratic */
10     a ← Σ_{i∈{x,y,z}} h_i^2
11     b ← −2 Σ_{i∈{x,y,z}} h_i^2 T_i^2
12     c ← Σ_{i∈{x,y,z}} h_i^2 T_i^2 − F
13     if b^2 − 4ac ≥ 0 then                    // Real solution exists
14         P.Φ ← (−b+√(b^2−4ac)) / 2a
15         return
16     else                                     // Lower dimensional update
17         i ← max_{i∈{x,y,z}} N_i.Φ
18         T_i ← 0
19         h_i ← 0
20         goto Line: 10
21     end if
22 end procedure
```

Higher order schemes require a wider stencil (containing more grid points), a second-order scheme for example is presented in [177]. The higher order schemes are out of scope for process TCAD simulations: In case the interface is not smooth, e.g., at interface corners, the higher order of those schemes is reduced. In order to solve (6.3) using the first-order scheme for Φ_{ijk}, rearranging the terms of (6.3) reveals the structure of a quadratic equation in Φ_{ijk}

$$\underbrace{\sum_{i\in\{x,y,z\}} \Delta_i^2}_{=a}\,\Phi_{ijk}^2 + \underbrace{-2 \sum_{i\in\{x,y,z\}} \Delta_i^2 T_i^2}_{=b}\,\Phi_{ijk} + \underbrace{\sum_{i\in\{x,y,z\}} \Delta_i^2 T_i^2 - F_{ijk}}_{=c} = 0, \tag{6.4}$$

with T_i the Φ value of the chosen upwind neighbor in the corresponding spatial dimension. This equation is solved in Algorithm 9 Line: 14.

In case no real solution exists (the quadratic has two complex roots) the largest upwind neighbor is removed (its contributions are set to zero cf. Algorithm 9 Lines: 17-19). Then the procedure to solve the quadratic equation is restarted. If only a single dimension has nonzero contributions, a real solution to the quadratic is guaranteed to exist.

FSM

The FSM [178], computes the solution by repetitively *sweeping* the computational domain. A *sweep* computes for every grid point an `Update` by considering only neighbors in, e.g., negative x/y/z-direction. For the next *sweep* the directions are changed considering the previous example: positive x-direction and negative y/z-direction. A full *sweep* consist of all the 2^d (d is the number of dimensions) possible combinations of sweep directions. Improvements to the FSM that *lock* some of the grid points, i.e., locked sweeping method (LSM) such that not all grid points have to be computed in a sweep, are presented in [179]. The FSM terminates, if the difference of the solution between two consecutive full *sweeps* is below a threshold. The threshold is selected based on the desired accuracy of the solution. In case of a constant speed function F the solution is already reached after a single full *sweep*, thus 2^d *sweeps* are necessary for *Re-Distancing*.

The Eikonal equation is a minimization problem, therefore, the solution is given by the smallest computed value at each grid point over all *sweeps* of a full *sweep*. Therefore, the individual *sweeps* may be computed in parallel allowing up to 2^d threads simultaneously. If more threads are available, a *sweep* is additionally parallelizable because all grid points with the same sum of their indices are independently computable (a strict order applies to the order of the sum of the indices) [180, 181].

FIM

The FIM [182, 183] (as its name suggest) iteratively uses the `Update` function on all grid points. The FIM terminates when the difference of the solution between two iterations falls below a threshold value. The threshold corresponds to the desired accuracy of the solution. In comparison with the FSM the FIM considers always all neighbors. Improvements to the FIM are that for an iteration only selected grid points are computed again [184]. For a constant speed function F the iteration terminates in a maximum number of iterations, which is equivalent to the diameter of the domain in grid points. Parallelization for the FIM is straightforward, because the update for all grid points is performed independently in each iteration [185, 186, 187].

FMM

The FMM as portrayed previously in Section 5.3 is unique in the sense that it is a single pass algorithm (every grid point is computed only once).

The modifications presented in Chapter 5 for parallelizing the FMM are not applicable to the FMM used in *Re-Distancing*, because the dependencies (upwind neighbors) of the grid points are not known beforehand.

Thus the approaches for parallelizing the FMM for *Re-Distancing* (or more generally the Eikonal equation) are based on domain decomposition. The domain is decomposed and an instance of the FMM is performed independently on each sub-domain. Consequently, this requires a merging of the solution from the different sub-domains. The merging inevitably leads to iterative rollback mechanisms for any domain decomposition approach. The role back mechanism invalidates the solution on parts of the domain and initiates a re-computation of the solution, thus the single pass property of the FMM is lost [188]. There are several approaches trying to limit the number grid points affected by rollbacks [189, 190]. The approaches differ by whether the sub-domains are overlapping, considered neighbors of the sub-domains, and exchange frequency between the sub-domains.

The exchange frequency between sub-domains is controlled by a parameter called *stride width* which limits how far the solution may advance before a mandatory data exchange takes place [61, 171]. So far the effect on the performance of the *stride width* has not been studied for hierarchical grids. Choosing a small *stride width* reduces the by the rollbacks affected points, whilst a large value reduces the number of synchronization barriers due to the reduced data exchanges.

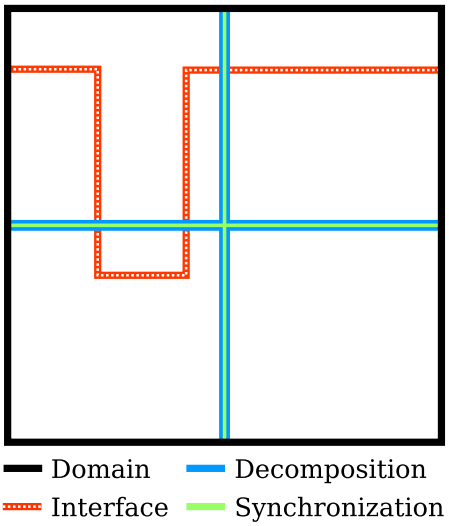
An overview of available FMM approaches in a single block and multi-block context is summarized in Figure 6.1.

In what follows, we consider the block decomposition to only apply to the *Re-Distancing* and not interfere with the given hierarchical grid itself, which is tailored to solution requirements of the physical simulation steps, i.e., optimal grid resolution in areas of interest, such as corners, to optimize robustness, accuracy, and computational complexity.

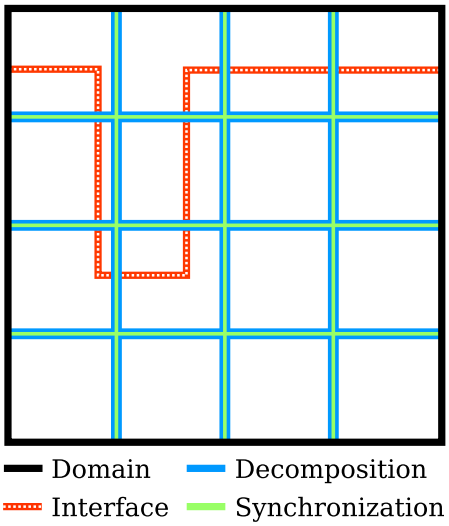
6.2 Block Decomposition

Previous analyses show that, if the number of blocks is about 10 times bigger than the number of threads, the load-balancing is possible [62], increasing parallel efficiency. To artificially increase the number of blocks on a level of a given hierarchical grid, a decomposition of the available blocks into sub-blocks is necessary. Sub-blocks are exactly like blocks, but to differentiate from the original blocks the distinguishable descriptor, sub-blocks, is used.

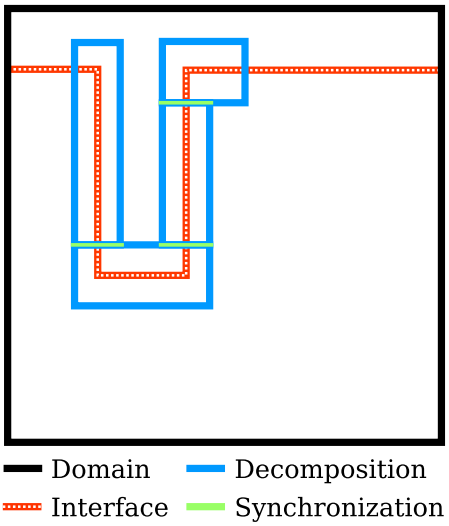
The naive decomposition approach splits the largest block into two sub-blocks along its longest side until the desired number of blocks is reached. This is not a viable option because this approach is inherently serial due to the selection of the largest block. Additionally, there is no lower bound on the block size, so excess creation of tiny sub-blocks, e.g., one grid point wide blocks, will deteriorate performance (as those blocks would still need a ghost layer, which would be bigger than the actual block).



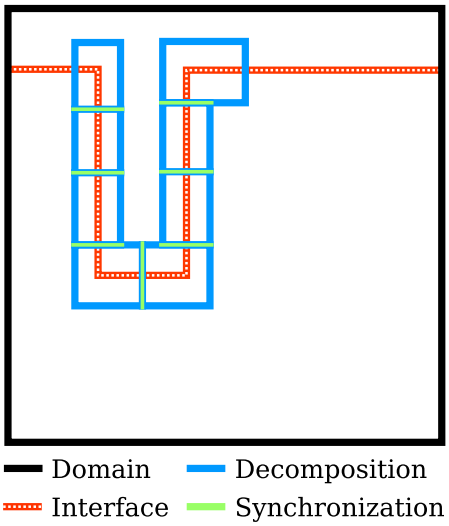
(a) Approach presented in [189]: Domain decomposition of a single block using one created sub-block per thread (four threads) is used, leading potentially to load-balancing issues: The thread processing the lower right sub-block is initially idle as no interface is present.



(b) Approach presented in [171]: Domain decomposition of a single block using multiple sub-blocks per thread and dynamically assigning the threads (i.e., four threads, but 16 blocks) is used, tackling the load-balancing issue depicted in (a).

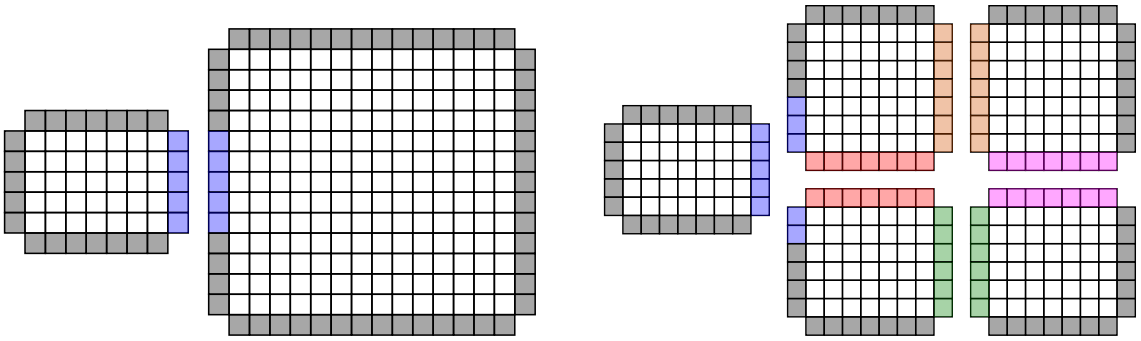


(c) Approach presented in [62]: Load-balancing if multiple blocks per thread are available. The effectiveness of the parallelization depends on the number and relative size of the given blocks. The Eikonal equation is only solved in regions covered by a block.



(d) The approach presented in this work: Improved parallel performance compared to approach shown in (c) is achieved by automatically decomposing the given blocks into significantly more sub-blocks. This increases the block per thread ratio, which leads to more efficient load-balancing and ultimately increases parallel performance, also presented in [159].

Figure 6.1: Approaches to parallelizing the FMM for a single block, e.g., *Level 0* of a hierarchical grid (top row) and multi-blocks, e.g., *Level 1* or *Level 2* of a hierarchical grid (bottom row). The shape and position of the interface is inspired by a typical *trench* geometry in process TCAD simulations [62]. Adapted with permission from Quell *et al., Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.



(a) Level of a hierarchal grid consisting of two non-overlapping blocks.

(b) Decomposition of the level shown in (a) using a *block size* of 10.

Figure 6.2: In **(a)** the level of a hierarchical grid consisting of two blocks and their appropriate ghost layer is shown: Grey cells in the ghost layer are given by interpolation or the domain boundary, whereas the blue ones are covered by the neighboring block, which allows for seamless propagation of information between the blocks. In **(b)** the blocks after the decomposition are shown. Only the bigger block is split into four sub-blocks, as the other one is smaller than the chosen *block size* of 10. The newly created grid points in the ghost layers of the sub-blocks are colored differently. Adapted with permission from Quell *et al., Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.

In contrast, a superior approach is developed and presented in the following. The approach decomposes blocks only, if they are larger than a given *block size* into sub-blocks smaller or equal in size of the *block size* (Figure 6.2). For a chosen *block size* of 10, only the larger block (14×13) is split, whilst the smaller block (7×5) remains unchanged. The larger block is simultaneously split into four sub-blocks, two with a size of (7×7) and two with a size of (7×6). This approach favors parallelism, because every block is split independently. Additionally, the *block size* parameter allows to take cache sizes of the underlying hardware into account, because the *block size* parameter gives tight control over the created sub-block sizes. The sub-block size has direct influence on the required memory.

To deploy the decomposition onto a hierarchical grid the neighbor relations between the (sub-)blocks have to be computed and the ghost layers have to be checked for overlaps, identifying grid points which require a synchronized exchange. During the preparation the blocks are split into sub-blocks, based on the proposed *block size*. Finally, the multi-block FMM is applied [62]. In summary, the developed advancements of the parallel FMM consist of three sub-steps: Block decomposition, sub-block allocation, and ghost layer computation. Figure 6.3 illustrates the algorithm via a flow chart and each step is discussed in the following.

Block Decomposition

The decomposition of a block is independent from the decomposition of other blocks, thus decomposing the blocks is inherently parallel. Additionally, the decomposition is also independently applicable to all spatial dimensions, therefore, it is sufficient to present the one-dimensional case.

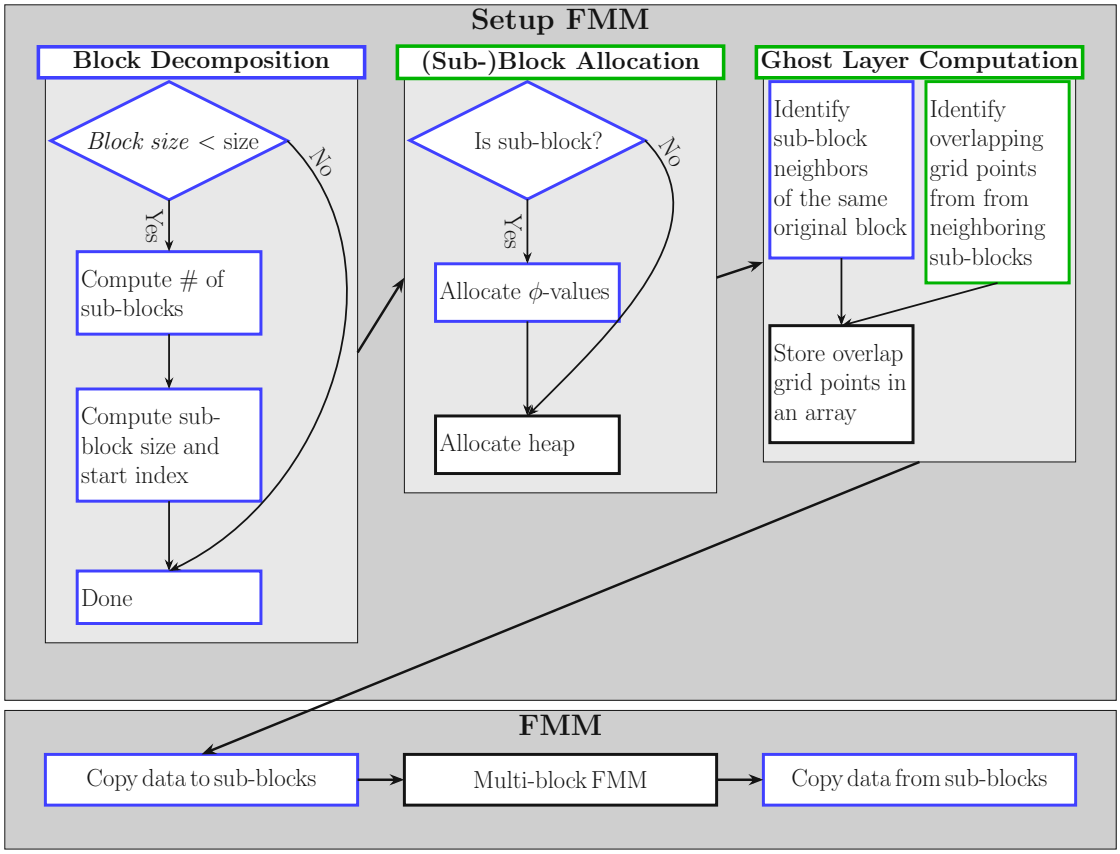


Figure 6.3: Flow chart of the setup of the FMM and execution of the FMM itself. Steps colored in blue are new, their computational overhead *vanishes*, if the chosen *block size* is larger than the block size. Green colored steps are modified compared to [62], whilst the black colored steps are unchanged. Adapted with permission from Quell *et al., Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.

For a given block by its start index S[2], its original size N in grid points, and the chosen *block size* B, the number of sub-blocks M is computed by

$$M = \left\lfloor \frac{N + B - 1}{B} \right\rfloor . \tag{6.5}$$

Thus there are M sub-blocks necessary so that none of them has to be bigger than B.

[2]The start index is later needed to compute the start indices of the sub-blocs

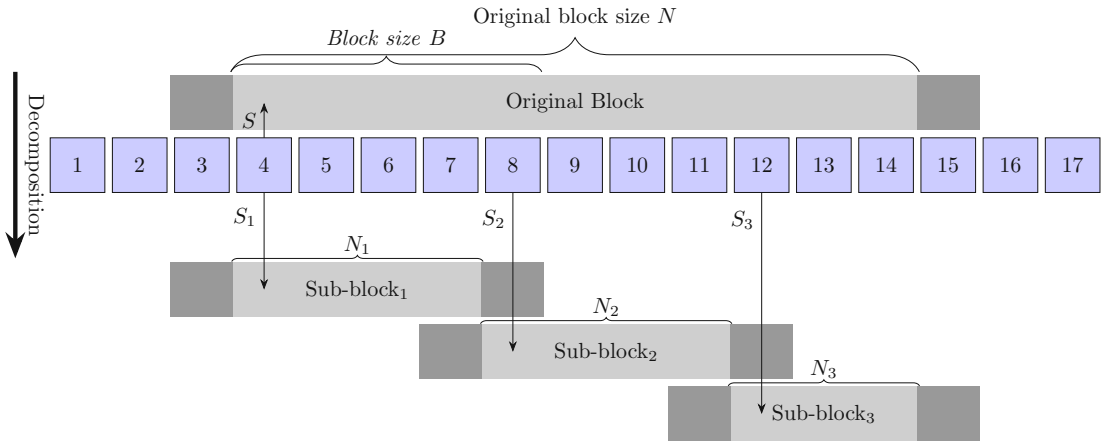


Figure 6.4: The global indexing scheme is given by the blue squares with their indices ranging from 1 to 17. Above the original block (gray box) with its start index $S = 4$ and original block size $N = 11$ spanning all grid points up to index 14 is shown. To the left and the right of the block the one grid point wide ghost layer is shown in dark gray. Below the global indexing scheme the created sub-blocks and their corresponding ghost layers are shown for the chosen *block size* $B = 5$. The three sub-blocks, with their own start index S_i and sub-block size $N_i \leq B$ (four or three) cover the same grid points as the given block. None of the sub-blocks has a sub-block size of five, because the decomposition strategy creates sub-blocks which may differ only by one grid point in size. Adapted with permission from Quell *et al., Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.

The individual starting index S_i of the sub-blocks and the size N_i for the sub-block i are computed using

$$N = M \cdot q + r, \tag{6.6}$$

with q the unique quotient and r the remainder. So, S_i and N_i are given by

$$S_i = \begin{cases} S + i(q+1) & \text{for } i < r \\ S + (i-r)q + r(q+1) & \text{for } i \geq r \end{cases}, \tag{6.7}$$

$$N_i = \begin{cases} q+1 & \text{for } i < r \\ q & \text{for } i \geq r \end{cases}. \tag{6.8}$$

The created sub-blocks vary in size by at most a single grid point, because of their definition. The case occurs, if N is not divisible by M. Figure 6.4 shows an exemplary block decomposition, with all variables graphically shown.

Consider an alternative approach where the original block is cut into B sized sub-blocks except for the last sub-block which is only r grid points wide. This alternative approach is inferior to the proposed approach because in case the last sub-block with size r is only one grid point wide requires frequent data exchange steps which deteriorate the parallel performance or in case of infrequent data exchanges the rollbacks affect many points.

In the higher dimensional case, the *block size* is equal in all spatial dimensions for the standard case where the grid resolution (distance between two grid points) is the same.

If the grid resolution is different along spatial dimensions, different *block sizes* along the different spatial dimensions are appropriate. If the sub-blocks are most similar to a cube, the spatial locality of the blocks is increased. This is not further investigated in this thesis, but might offer an interesting path for future research, especially for high aspect ratios of the grid resolution along different spatial dimensions.

(Sub-)Block Allocation

After the sub-blocks are defined, memory for the grid points, ghost layer and, the FMM's binary heap has to be allocated (cf. Figure 6.3). The heap is preallocated to avoid costly re-allocations during the execution of the FMM in case the heap outgrows the initial chosen size. The preallocation size is chosen such that all grid points of a block are able to fit, thus no re-allocations are necessary. The preallocation allows to use an indexed lookup into the heap, if grid points are already present, shortening the time required to update the priority (key) for a grid point.

Blocks which have not been decomposed require only the allocation of the heap data structure. The whole process is parallelized over the sub-blocks, enabling the parallel execution of up to the number of sub-blocks threads. If less threads are available load-balancing takes place, because sub-blocks which do not require a decomposition take significantly less time. A synchronization barrier is needed to proceed to the next step.

Importantly, no data is copied during the block allocation to the sub-blocks. Data is only copied directly before and after the FMM is executed. This enables an efficient reuse of the sub-blocks over several time steps of a full process TCAD simulation, as long as the underlying hierarchical grid does not change.

Ghost Layer Computation

The neighboring sub-blocks (in the following, referred to as neighbors) are computed in two steps: 1) Neighbors from the same original block and 2) neighbors from a different original block (cf. Figure 6.3). The neighbors from the same original block are computed by index calculation, because the original block is regularly decomposed. Those neighbors either share a full face (i.e., all grid points of one of the axis-aligned sides) or no grid point at all. The neighbors from a different original block, are computed with a pairwise overlap computation of their ghost layer, for the sub-blocks. In the overlap computation only sub-blocks which originate from a neighboring blocks of their original block are considered. Thus the performance is increased, because not all sub-blocks have to be considered. The grid points in the ghost layer are marked to belong to the externally set grid points[3] or to the to-be-synchronized grid points, i.e., they are covered by a neighboring sub-block. In the latter case, the grid points are collected on a per block basis to allow for an efficient data exchange with the neighboring sub-blocks. The parallelization strategy is the same as the one employed for the sub-block allocation, allowing high parallelization.

[3]Grid points for which the signed-distance value is given by domain boundary conditions or by interpolation from a coarser level of the hierarchical grid. Their signed-distance value is not changed by the FMM.

6.3 Benchmark Examples and Analyses

The proposed block-based FMM is evaluated based on three benchmark examples. The first example is a *Point Source* example. The other two (*Mandrel* and *Quad-Hole*) examples are inspired by process TCAD simulations.

The results are obtained from the benchmark system VSC4 (cf. Section 3.2). Different values for the parameter *block size* are compared as well as different values for the parameter *stride width*.

6.3.1 Point Source

The *Point Source* example is a fundamental test case for benchmarking Eikonal solvers [61, 146, 171]. The computational domain covers the cube $[-0.5, 0.5]^3$ using a Cartesian grid (*Level 0* of a hierarchical grid). The spatial discretization has 256 grid points along each spatial dimension yielding a total of 16 777 216 grid points. The domain boundary conditions are chosen to be symmetric in all spatial dimensions. At the center ($[0, 0, 0]$) a single grid point is set to be the source point (interface). The speed function is constant, $F = 1$. Thus the iso-contours of Φ are spheres centered on the source point (cf. Figure 6.5a).

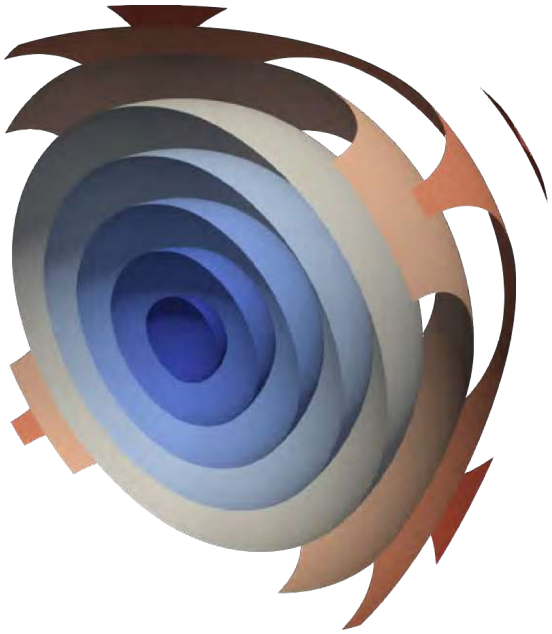
First, the modified step *Setup FMM* (cf. Figure 6.3) is analyzed and then the performance of the FMM itself is analyzed.

Setup FMM

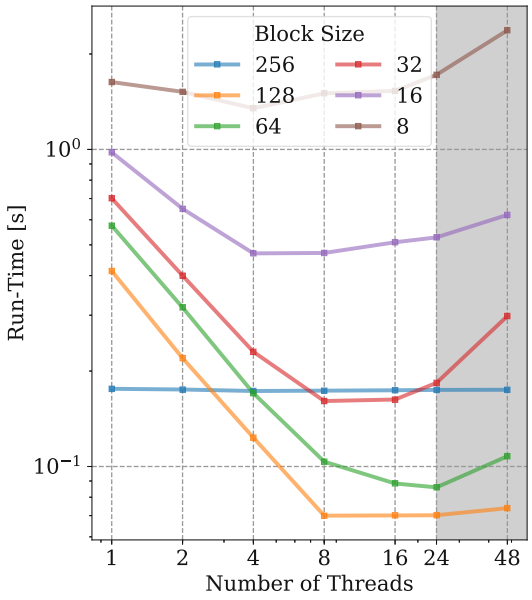
The measured run-times are shown in Figure 6.5b for *block sizes* ranging from 256 (a single block) down to eight (32 768 blocks). In case of *block size* 256 no decomposition is performed, giving no parallelization possibilities. In the other cases the block is decomposed, creating a significant serial overhead, due to memory allocation and additional ghost point computations. Parallel execution on the other had is now possible, yielding (depending on the chosen *block size* and used number of threads) a shorter run-time than the base case without decomposition. For *block sizes* of eight or 16 the break even point is never reached because the enormous number of blocks (4 096 and 32 768, respectively), each with only a little computational, load suffer from synchronization overhead which materializes with more than four threads. Usage of computational resources from the second processor (more than 24 threads) did in no case increase the performance as NUMA effects add to the already memory-bound problem.

FMM

To analyze the run-time of the advanced FMM itself and the parallel speedup, Figure 6.6 shows the run-time and speedup for *block sizes* from eight to 256 and for different values of *stride width* measured in multiples of the grid resolution. In case no decomposition takes place (*block size* 256), the run-time is hardly affected by the used number of threads as well as from the *stride width*. The next finer *block size* 128, creates eight blocks.



(a) Isocontours of Φ from 0 to 0.8 in steps by 0.01. The domain is cut in half to provide an inside view.



(b) Run-time to setup the sub-blocks.

Figure 6.5: **(a)** The iso-contours of Φ for the *Point Source* example. **(b)** The run-time to setup the sub-blocks 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, indicating NUMA effects. Adapted with permission from Quell *et al., Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.

A serial speedup is measured for *stride widths* smaller than 20. The speedup ranges from 1.01 (*stride width* of 0.5) to 1.14 (*stride width* of 3.5), the cause of the speedup is the reduced number of grid points per block and smaller heap sizes and better cache efficiency due to data locality. The parallel speedup saturates for eight threads, because there are only eight blocks available. Small *stride widths* perform better, because the source point is located on a single sub-block, allowing computations on other sub-blocks only after the first exchange step, which is caused earlier by a smaller *stride width*.

For smaller *block sizes* a serial speedup is observed for *stride widths* of less than 10, reaching the highest serial speedup of 1.21 for a *block size* of 64 and a *stride width* of 3.5. If the used number of threads is high, large *stride widths* perform better, because the computational load per sub-block is decreasing rapidly, but the overhead caused by the synchronization decreases slower in comparison (following a square-cube-law). The peak parallel speedup of 19.1 using all 24 threads of a single processor is achieved with a *block size* of 32 and a *stride width* being equivalent to infinity (i.e., 10 000).

Utilizing the second processor only gives a speedup for *block sizes* of 64 and 32 and large *stride widths*, bigger than (depending on the block size) 10 and 50, respectively. The reason is that the typical computational load per task is to small too compensate the synchronization overhead.

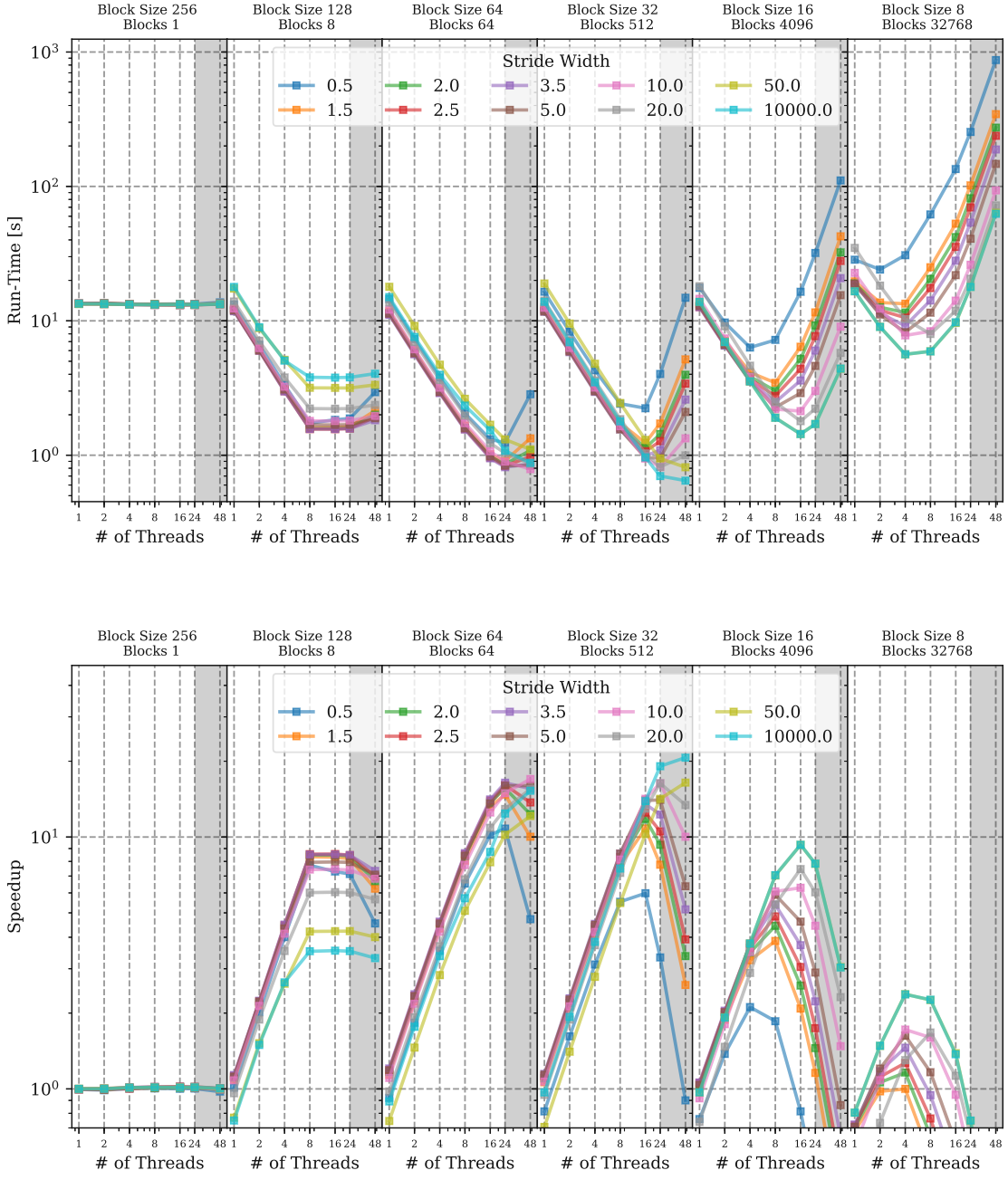
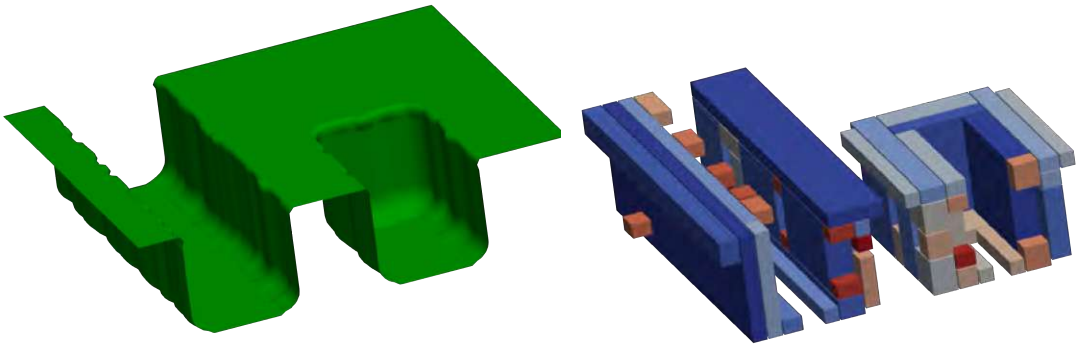


Figure 6.6: Run-time of the advanced FMM (top graph) and parallel speedup (bottom graph) for the *Point Source* example, using different values for the *block size* and *stride width*. The parallel speedup is compared to the run-time of the serial execution using a *block size* of 256, which is equal to the domain size. Adapted with permission from Quell *et al.*, *Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.



(a) Interface extracted from *Level 0*. **(b)** Block placement on *Level 1*.

Figure 6.7: In **(a)** the interface of the *Mandrel* example is shown and in **(b)** the block placement on *Level 1* is shown. The blocks are colored from red to blue by their size from the smallest block (size $12 \times 12 \times 12$) to the biggest block (size $12 \times 288 \times 84$). Adapted with permission from Quell *et al., Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.

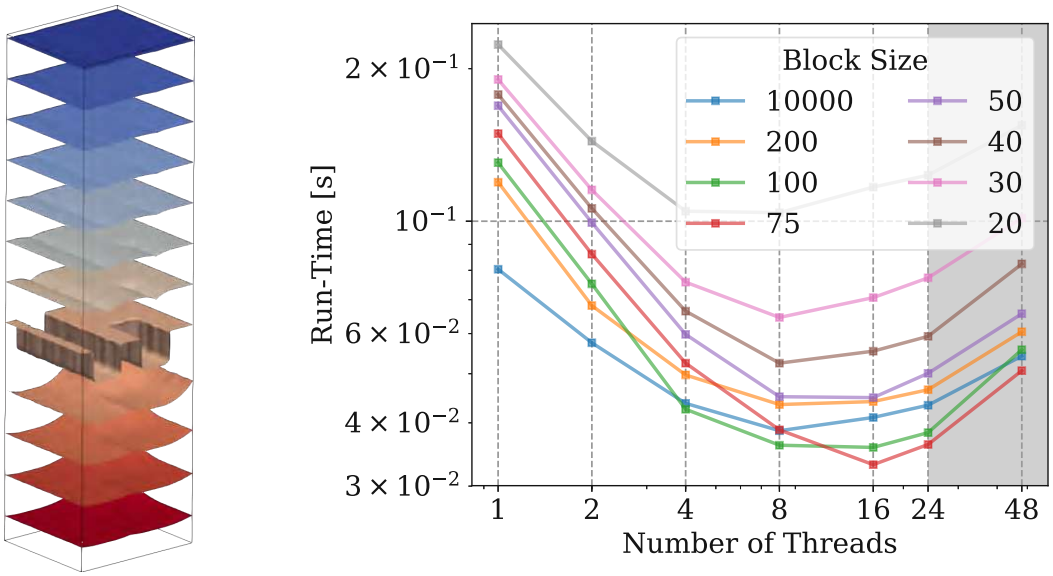
The same parameters of *block size* and *stride width* as for the usage of a single processor yield the peak parallel speedup of 20.7 for 48 threads: NUMA effects limit parallel performance, because frequent (especially for small *stride width*) synchronization steps cause a task rescheduling. The task rescheduling in OpenMP does not consider memory layout, resulting in many indirect memory accesses, because a block may have been processed earlier by a thread located on a different processor.

The next evaluation example is based on a process TCAD simulation and a hierarchical grid.

6.3.2 Mandrel

The *Mandrel* benchmark example is taken from a representative process TCAD simulation, where two trenches are etched into a silicon waver. One trench spans the full width of the simulation domain, and the other only half. Again, symmetric boundary conditions are used. Figure 6.7a shows the 0-level-set for which the signed-distance function is computed (constant speed function $F = 1$). The hierarchical grid consists of two levels. The block on *Level 0* has a size of $84 \times 72 \times 312$, totaling about 1.8 million grid points. On *Level 1* there are 78 blocks with their sizes ranging from $12 \times 12 \times 12$ to $12 \times 288 \times 84$, totaling about 2.5 million grid points. In Figure 6.7b the blocks on *Level 1* are visualized: They are placed around the trenches. The signed-distance field (computed relative to the interface) is visualized via iso-contours, as shown in Figure 6.8a.

First, the *Setup FMM* is investigated and then the performance of the FMM itself is analyzed on both levels of the hierarchical grid separately.



(a) Isocontours of Φ from -0.7 to 0.5 in steps by 0.1.

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

Figure 6.8: In **(a)** the computational domain and the iso-contours of Φ are shown for the *Mandrel* example. In **(b)** the time to setup the sub-blocks and to compute the neighbor relations for various *block sizes* and number of threads is shown. Adapted with permission from Quell *et al., Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.

Setup FMM

Figure 6.8b shows the measured run-time on VSC4. The setup time for the *Mandrel* example profits from using more threads, even without the block decomposition, because on *Level 1* there are natively 78 blocks which may be processed in parallel. The shortest run-time is achieved using eight threads. Considering also the run-time with the block decomposition, a serial overhead materializes (due to memory allocation) with the added benefit of a better parallel scalability. The shortest run-time is achieved for a *block size* of 75 using 16 threads. Increasing the number of threads beyond 16 increases the run-time, especially for 48, threads when both processors are utilized. This is attributed to NUMA effects and the total lower computational load (maximum 2.5 million grid points) compared to the *Point Source* example (16 million grid points).

FMM Level 0

In Figure 6.9 the run-time and parallel speedup for the *Level 0* of the *Mandrel* example are shown. A larger *stride width* performs better in the case where the block is not decomposed. Compared to the *Point Source* example where little influence of the *stride width* has been found: The run-time is an order of magnitude shorter and reveals that the introduced overhead by the restarts of a small *stride width* is not negligible.

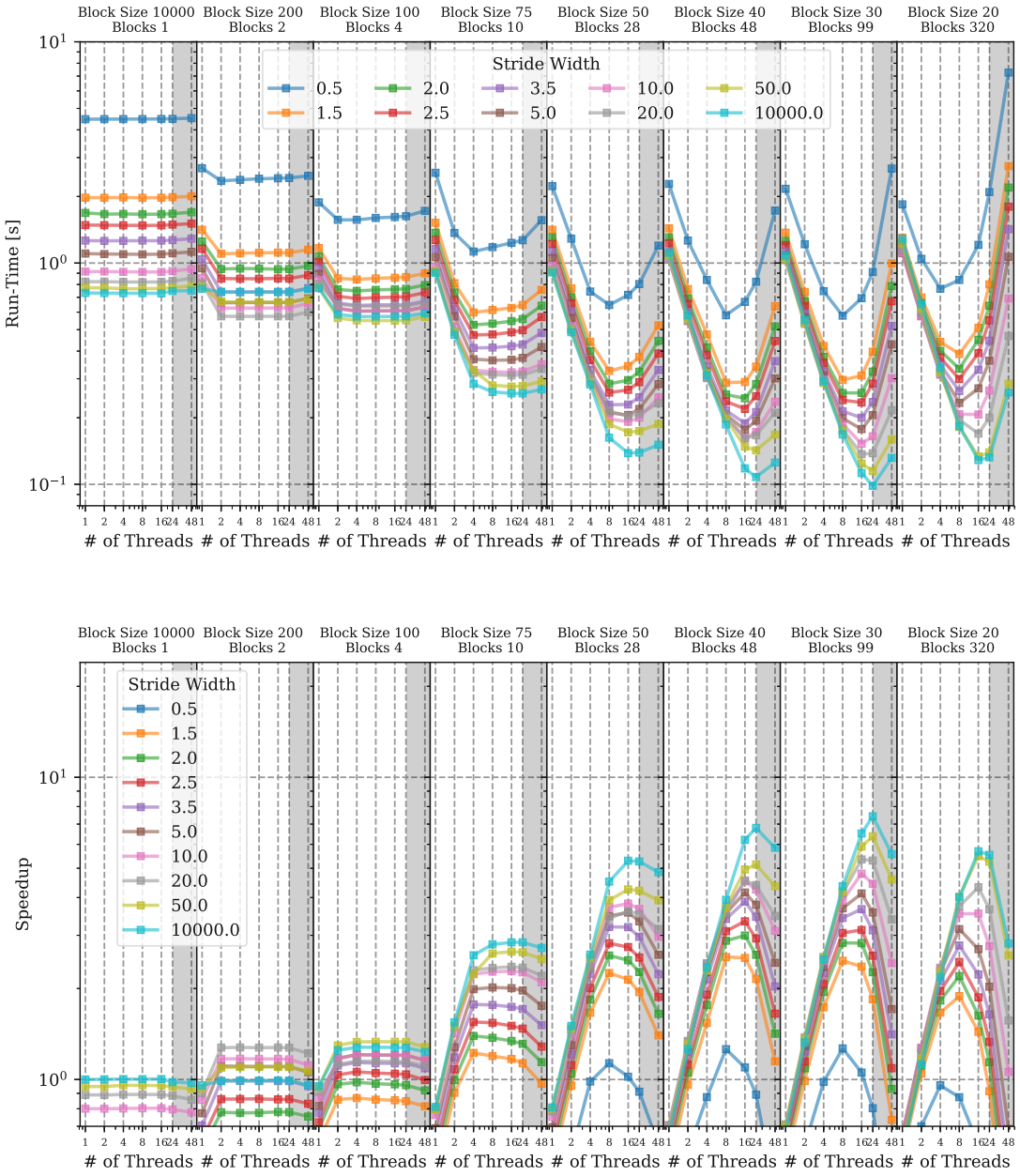


Figure 6.9: Run-time and speedup compared to the serial execution using a *block size* of 10 000 of the FMM for the *Mandrel* example on *Level 0* (coarse level) of the hierarchical grid. Adapted with permission from Quell *et al.*, *Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.

Choosing a *block size* of 200 creates two sub-blocks (the domain is split along the z-axis). A serial speedup is not observed, because the split almost aligns with the interface, which already partitions the domain in two independent sets. The run-time is decreased for parallel execution only for a *stride width* from 10 to 50. For the maximum *stride width* of 10 000 no parallel speedup is observed, because the interface and thus all initial points are located in a single sub-block, which forces a sequential computation of the sub-blocks.

For *stride widths* smaller than 10 the restart overhead is too large to reach the performance of the not decomposed case.

A *block size* of 100 still splits the block only along the z-axis, giving no run-time reduction, because the solutions of the sub-blocks strictly depend on each other. The only noticeable improvement is for a *stride width* 10 000, because now the interface is present in two sub-blocks allowing for parallelization.

If the *block size* is 75 or smaller, the block is not only split along the z-axis, but also along the x-axis and y-axis, allowing for a better parallel performance. The sub-blocks created from splits along x-axis and y-axis are compared to the previous splits along the z-axis are almost independent. Independently of the *block size*, the shortest run-time is achieved with the largest *stride width*. The best speedup (7.5) is achieved using a *block size* of 30 creating 99 blocks and using all 24 threads of a single processor.

FMM Level 1

On *Level 1* the run-time is about three times higher than on *Level 0*, thus the performance impact on the overall run-time of this level is bigger. The measured run-time and speedup are shown in Figure 6.10.

A serial speedup is observed for *stride widths* from 1.5 to 10. The highest serial speedup is of 1.21 is achieved for a *stride width* of five and a *block size* of 50. The main reason for the serial speedup on this level is that the initial grid points in the ghost layer, which are interpolated from the coarser blocks, are not immediately used for the FMM (because their Φ value is beyond the current *stride width*). Ignoring those ghost points generally is not a viable option, because some of them might be essential for the correct solution. Ghost points are usually not source points for the finally computed signed-distance field, except in cases of an unfortunate block placement with respect to the interface.

Figure 6.11 shows such an unfortunate block placement (with respect to the interface). The interface crosses the block but on one side it is outside the block but still close. The yellow marked ghost cells are sources for the signed-distance field (they are closer to the interface than their neighboring block cells), ignoring them would create wrong results. The gray marked ghost cells are not sources, their interpolated value does not influence the computation of the final signed-distance field, they could be safely ignored. Differentiating between those two ghost cell types before the computation of the signed-distance field is infeasible because the to-be-computed signed-distance field has to be known on neighboring grid points.

The *block size* itself does not influence the serial speedup, because the serial speedup is about 1.2 for a *stride width* of five, until a *block size* of 40. For smaller *block sizes* the serial speedup is slightly less, because the synchronization overhead impacts the performance.

The peak parallel performance of the FMM on *Level 1* without any decomposition is 7.4 using 16 threads and the maximum *stride width*. The proposed block decomposition doubles the peak parallel speedup to 15.4 at 16 threads for a *block size* of 50 and a *stride width* of five.

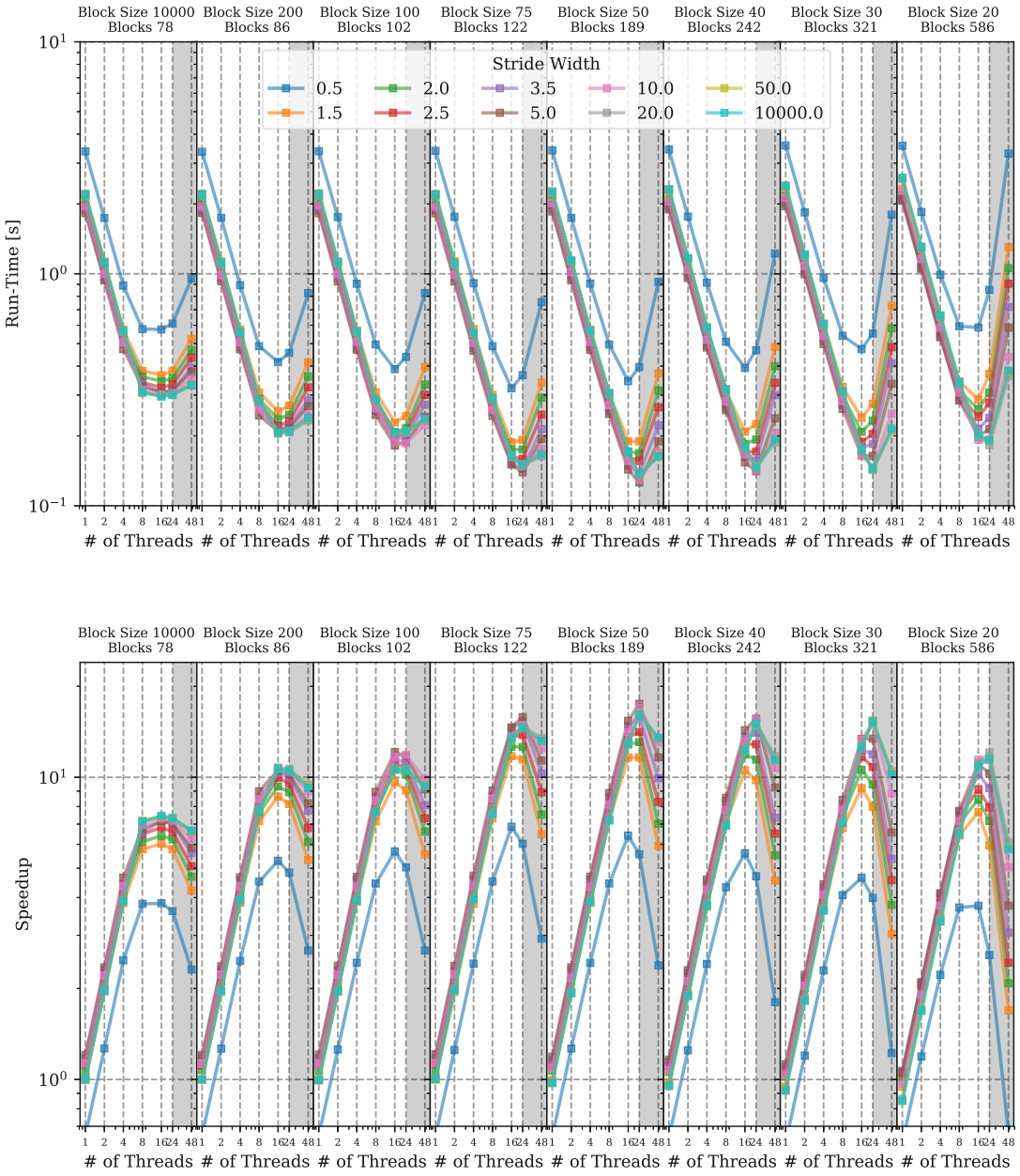


Figure 6.10: Run-time and speedup compared to the serial execution using a *block size* of 10 000 of the FMM for the *Mandrel* example on *Level 1* (fine level) of the hierarchical grid. Adapted with permission from Quell *et al., Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.

If all cores of a single processor are used, the parallel speedup reaches 17.4 for the same parameters of *block size* and *stride width*. The best performance with respect to the *stride width* is achieved for a *stride width* of five, because the issue arising from treating the ghost points as potential sources (necessary for algorithm correctness and robustness) is mitigated.

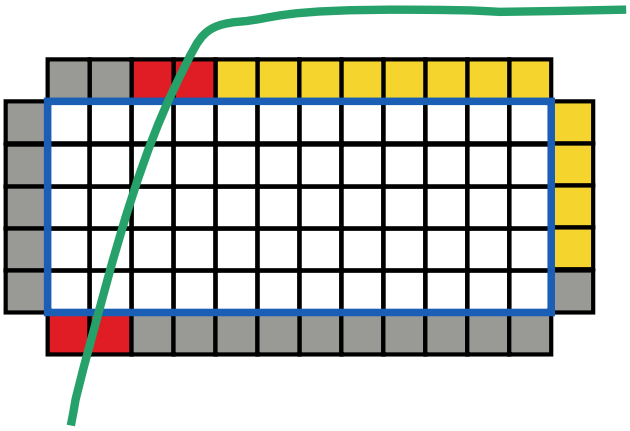


Figure 6.11: Interpolated ghost cells (yellow) are sources for the signed-distance field for the interface (green curve). The red colored ghost cells are next to the interface. The gray colored ghost cells are no sources, they do not influence the signed-distance field.

The investigation is concluded with the second example based on an interface from a process TCAD simulation, demonstrating the applicability of the proposed block decomposition on several interfaces.

6.3.3 Quad-Hole

The *Quad-Hole* example is also based on a process TCAD simulation [62]. The interface domain has four regions of interest, two half holes and two quarter holes (cf. Figure 6.12a). This example is also analyzed in [62], where the example with 48 blocks corresponds to *Level 1* and the example with 303 blocks to *Level 2*.

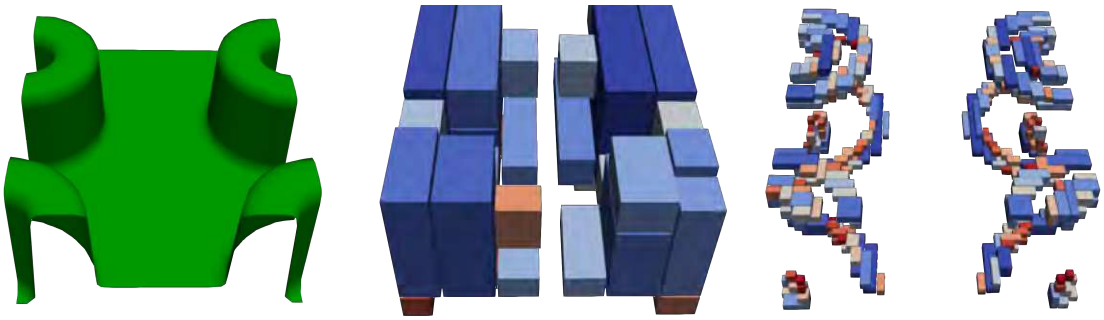
The only block on *Level 0* has a size of $38 \times 28 \times 30$. There are 48 blocks on *Level 1*, with their sizes ranging from $12 \times 16 \times 12$ to $68 \times 20 \times 52$. Their placement is shown in Figure 6.12b. They cover the regions around the quad holes completely. On *Level 2* there are 303 blocks, with their sizes ranging from $12 \times 12 \times 12$ to $164 \times 20 \times 12$ and their placement is shown in Figure 6.12c. They cover only the regions at the top and bottom of the quad holes where there are sharp edges.

The presentation of the results for the *Setup FMM* is omitted, because the results are qualitatively the same to the ones obtained in the *Mandrel* example. No new insights are provided: The decomposition introduces a run-time overhead, if only a single thread is used, but by using a higher number of threads the increased parallel efficiency outperform the approach without decomposition.

The single-threaded run-time on *Level 0* with a *block size* and *stride width* of 10 000 is 0.0013 s. For comparison, the single-threaded run-time with the same parameters on *Level 1* is 0.402 s (30 times as long as *Level 0*) and on *Level 2* is 1.568 s (120 times as long as *Level 0*). Thus, the discussion of *Level 0* is skipped.

FMM Level 1

Figure 6.13 shows the gathered run-time data for *Level 1*. The peak serial speedup of 1.14 is achieved for a *stride width* of 3.5 and a *block size* of 75.



(a) Interface extracted from *Level 0*. **(b)** Block placement on *Level 1*. **(c)** Block placement on *Level 2*.

Figure 6.12: The interface of the *Quad-Hole* example is shown in **(a)**, in **(b)** the block placement on *Level 1*, and **(c)** the block placement of *Level 2* is visualized. The blocks are colored by their size, from biggest (blue) to smallest (red). Adapted with permission from Quell *et al.*, *Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.

The serial speedup decreases with smaller block sizes, because of growing overhead introduced by the data exchange between blocks.

Considering parallelization, a *stride width* between two and 10 is beneficial for the run-time for less than eight threads. The best performance is typically achieved with a *stride width* of five. The worst performance is achieved with the *stride width* of 0.5 in almost all cases (except *block size* 10 and using less than four threads). For a high number of threads, e.g., 24 threads, the maximum *stride width* performs best. The overall peak performance is achieved with a *block size* of 50 using 24 threads.

The impact of the block decomposition on the parallel speedup is clear when the best parallel speedup without the developed decomposition (speedup of 8.2) is compared to the best parallel speedup when the developed decomposition is used (speedup of 12.5). Utilizing the second processor does not give any additional performance.

FMM Level 2

On *Level 2* the total run-time (cf. Figure 6.14) is four times as high as on *Level 1*, thus a speedup here has a bigger impact on the overall level-set simulation. Again, a serial speedup of up to 1.05 is observed, for a *stride width* between two and five, for smaller *block sizes* the serial speedup declines. Beginning with *block size* 30 a serial slowdown is measured, again due to the growing overhead introduced by the data exchange between blocks. The peak parallel speedup of 16.6 is achieved for a *block size* of 75 and a *stride width* of 20 on *Level 2*. Comparing the parallel speedup of the block decomposed FMM to the non-decomposed parallel speedup (which reaches 16.5) shows that for an already high number of blocks the decomposition barely effects the performance. The performance only deteriorates in cases with a very small *block size* (less than 20) where many sub-blocks are created. This is due to the additional overhead created by the synchronization of the sub-blocks.

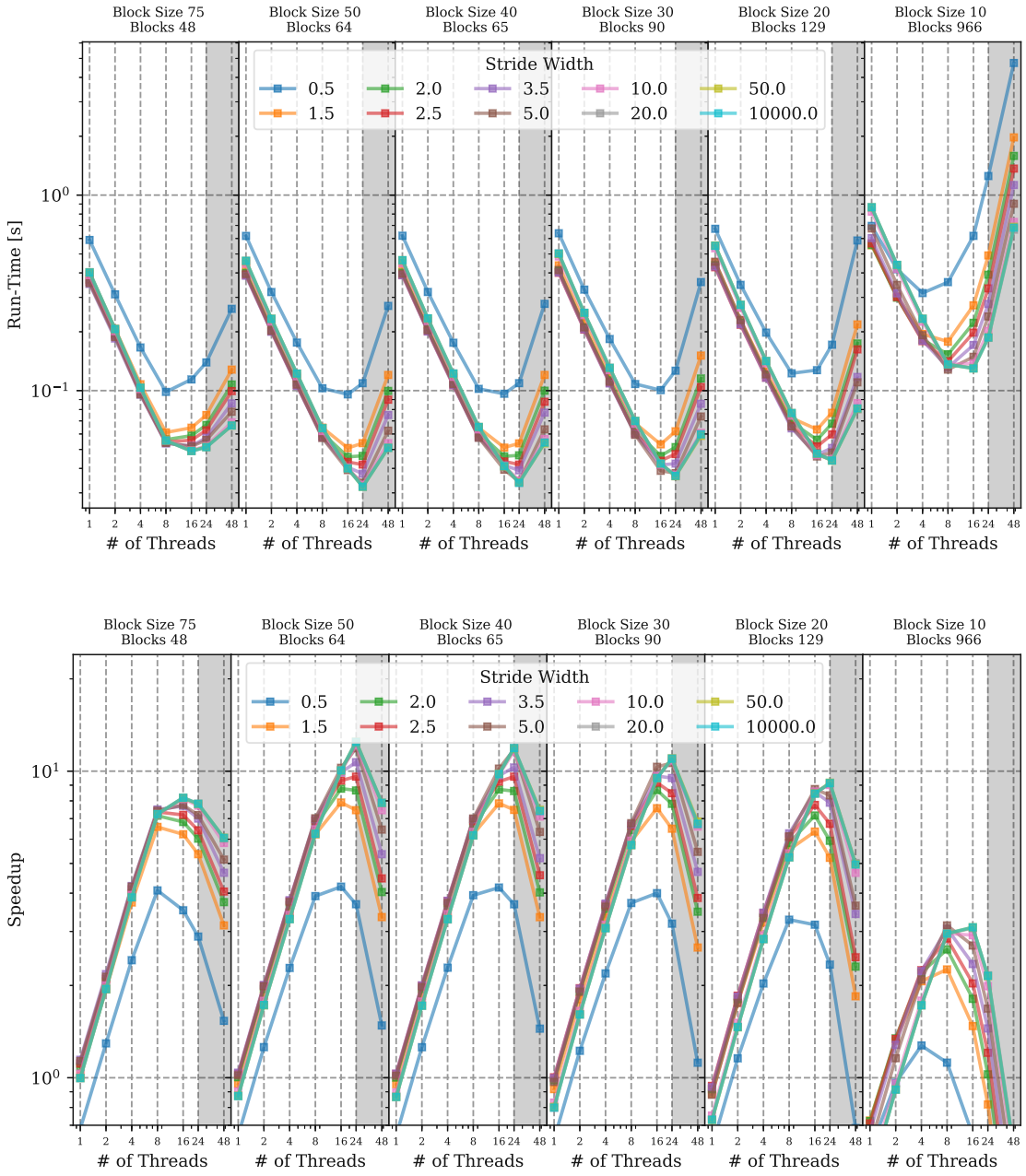


Figure 6.13: Run-time and speedup of the FMM, using different values for the *block size* and *stride width*, compared to the serial execution using a *block size* of 10 000 for the *Quad-Hole* example on Level 1 of the hierarchical grid. Adapted with permission from Quell *et al.*, *Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.

6.4 Summary

This chapter presented a block decomposition to increase the parallel performance of the FMM on a hierarchical grid. The block decomposition is applicable to all levels of a hierarchical grid yielding a unified parallelization approach. The limited parallel speedup (caused by load-imbalances) of the multi-block FMM on the given blocks of a hierarchical grid are overcome by a novel decomposition strategy.

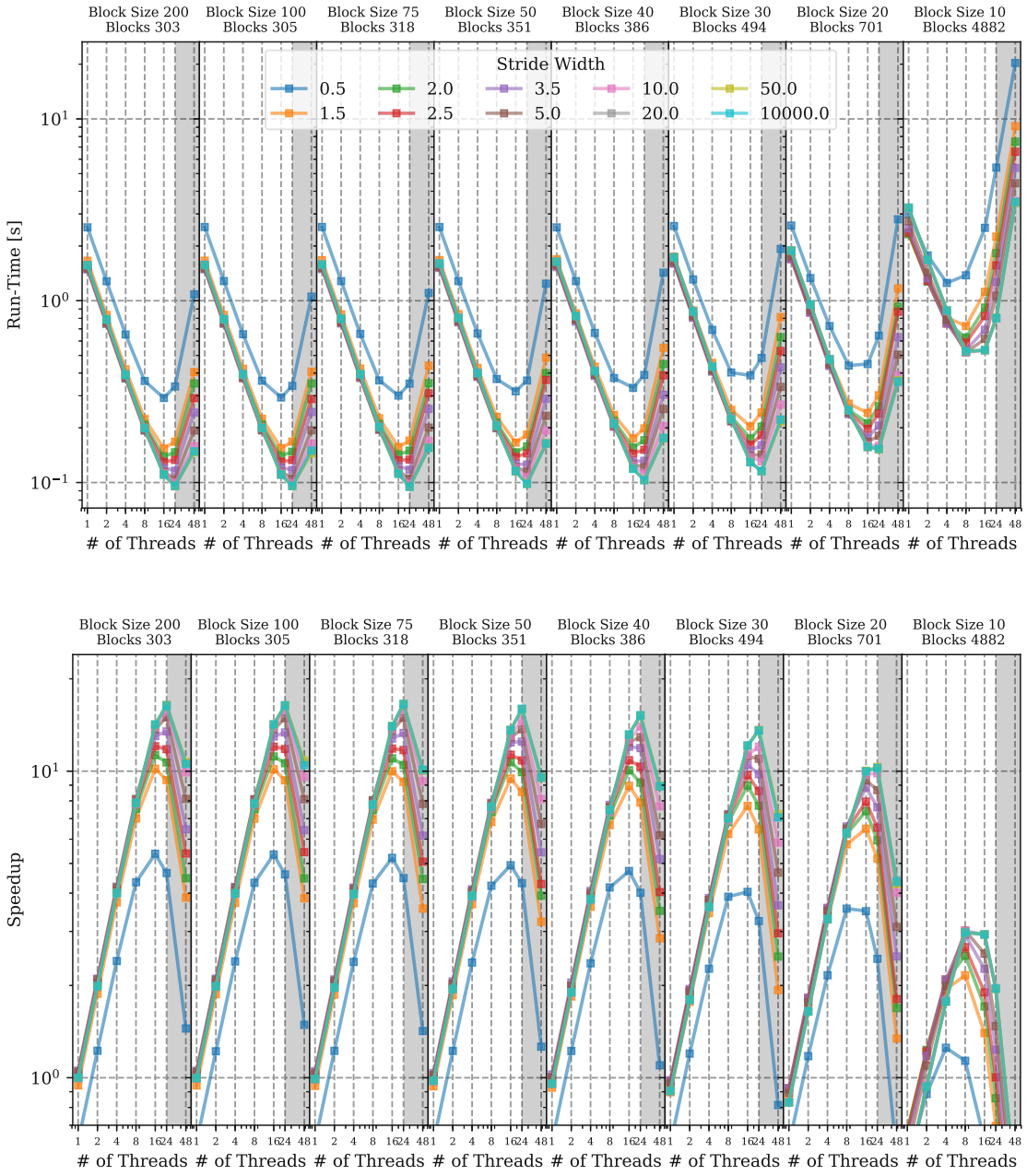


Figure 6.14: Run-time and speedup of the FMM, using different values for the *block size* and *stride width*, compared to the serial execution using a *block size* of 10 000 for the *Quad-Hole* example on Level 2 of the hierarchical grid. Adapted with permission from Quell *et al.*, *Journal of Computational and Applied Mathematics* 392, (2021) p. 113488. [159], © CC 4.0, http://creativecommons.org/licenses/by/4.0/.

The decomposition strategy splits the given blocks based on a threshold value (*block size*) into smaller sub-blocks. Thus, the total block count is increased, enabling the control of the number of blocks used in the multi-block FMM. The number of enforced data exchange steps between the blocks based on another parameter (*stride width*) is evaluated on a hierarchical grid.

The performance of the proposed block decomposition and the parameter values (*block size* and *stride width*) is evaluated using three examples. For the generic point source example (a typical test case for benchmarking Eikonal equation solvers) a parallel speedup of 19.1 is achieved using 24 threads. For interface geometries based on process TCAD simulations, speedups of 17.4 for 24 threads are achieved. The original approach without block decomposition achieved a parallel speedup of only 7.4, which is not even half the parallel speedup obtained with the block decomposition. As was shown, the *block size* shall not be chosen smaller than 30, because for smaller *block sizes* the overhead usually deteriorate the performance.

The *stride width* should be chosen between 2.5 and 10 for less than eight threads for best performance, because on a hierarchical grid this reduces the unwanted computations from ghost cell sources. For more than eight threads the maximum *stride width* (not introducing any additional data exchanges) performs best (shortest run-time), because the computational load between two data exchanges is insufficient otherwise little (the additional synchronization overhead is bigger than the gained performance due to parallelism).

Chapter 7

Bottom-Up Correction for Re-Distancing

The previous chapter presented an algorithm used in the *Re-Distancing* step in a top-down manner (processing the levels of a hierarchical grid from coarsest to finest). In contrast, this chapter proposes a bottom-up correction algorithm to increase the accuracy of the signed-distance field on coarser levels of the hierarchical grid. This is important to couple the levels of the hierarchical grid together, because *Re-Gridding* may remove higher levels from the hierarchical grid. An additional goal is to keep the computational overhead low with respect to the always necessary top-down re-distancing algorithm. The developed algorithm does not only increase the accuracy in regions covered by blocks on a higher level, as is possible through straightforward interpolation, but also in regions not covered by blocks on higher levels.

First, the idea and the algorithmic implementation of the proposed bottom-up correction algorithm, which is the core contribution of this chapter [100], is discussed (Section 7.1). Then the proposed bottom-up correction algorithm is evaluated, by two criteria (Section 7.2):

- Accuracy: Computing the errors (difference to an exact solution) of the signed-distance field.
- Performance: Measuring the run-time compared to *Re-Distancing* without the correction algorithm.

The evaluation examples consider interfaces representing typical challenges in process TCAD simulations, i.e., corners and trenches. The examples are prepared and chosen so as to allow for the exact computation of a signed-distance field as explicit interface representations (e.g., triangles) are available.

7.1 Algorithmic Implementation

The core idea of the developed bottom-up correction algorithm is to process the levels of a hierarchical grid in reversed order. The reversed order allows to incorporate the solution from spatially fine resolved regions into coarse resolved regions.

Algorithm 10: The bottom-up correction algorithm processes the hierarchical grid in a bottom-up manner. First, the signed-distance field on a level is interpolated based on the solution of the next higher level. Then the FMM is initialized keeping the flags from the previous (top-down) redistancing. Finally, an unmodified marching (from the FMM) is used to correct the solution also in unrefined regions.

```
1  procedure Correction():
2      l ← highest level -1
3      while l ≥ 0 do
           /* Initialization Phase */
4          foreach Block on Level l do                          // Parallel region
5              foreach ChildBlock on Level l + 1 do   // Nested parallel region
6                  InterpolatedLists ← interpolateGridPoints()       // Create task
7              end foreach
8              mergeInterpolatedLists()
9              initializeFMMCorrection()
10         end foreach
11         Wait                                          // Synchronization barrier
           /* Marching Phase */
12         foreach Block on Level l do                      // Unmodified marching
13             Marching()                                           // Create task
14         end foreach
15         Wait                                          // Synchronization barrier
16         l ← l − 1                                        // Move to next level
17     end while
18 end procedure
```

The algorithm (cf. Algorithm 10) operates in a bottom-up manner, starting from the second highest level: The highest level may not be corrected, because there is no higher level from which to derive a correction. First, the signed-distance field is interpolated based on the solution on the higher level of the hierarchical grid and then the FMM is used to improve the accuracy of the signed-distance field. The usage of the FMM allows to improve the accuracy in regions which are not covered by blocks on higher levels.

In a practical setting, the overhead of the bottom-up correction algorithm is reduced by reusing the flags (adapting the initialization of the FMM) used by the presented (top-down) algorithm in Section 5.3.

On every level the developed algorithm is split into two phases (same phases as the FMM):

- Initialization: Interpolation of the signed-distance values based on the values on higher levels and collect the corresponding grid points in block-specific lists. Those lists are then used to initialize the *Band* data structure (*Heap*).
- Marching: Execute the core multi-block FMM. For efficiency purposes grid points keep their signed-distance value and flag from the previous top-down re-distancing algorithm.

Initialization

The initialization phase starts in Algorithm 10 Line: 4. All blocks on the current level (*Level l*) are processed in parallel. For each block on the current level a nested parallel region for all its child blocks is created. With the nested parallel region the interpolation is able to utilize the usually higher number of blocks on higher levels of a hierarchical grid (*Level (l + 1)*) for a better load-balancing and, therefore, a better parallelization.

The Φ values of grid points covered by a block on the next higher level are interpolated using a linear interpolation along all spatial directions. Thus, bilinear interpolation in 2D simulations and trilinear interpolation in 3D simulations. In Figure 7.1 a schematic representation of the interpolation cases is shown: The interface is shown by the green curves on both levels (the higher spatial resolution on the higher level allows for a more accurate, i.e., sharper, representation of corners). The red dots mark grid points on the *outside* and blue ones on the *inside*. The grid points on the lower level are labeled with a letter from A to L. Some of the cells on the higher level are not marked with a colored point, because the signed-distance field has been computed only in a narrow-band[1]. However, on a hierarchical grid the blocks on levels higher than zero, typically already form a narrow-band like structure. Thus, setting the narrow-band based on an explicit threshold value is rarely encountered on block based AMR, nevertheless still possible, therefore the case is also considered.

The grid points A, E, and I are not covered by a block on a higher level and, therefore, not interpolated. The grid points G, H, J, K, and L have a neighboring grid point with opposing sign. Thus, they are *Close Points* and must not be modified. The grid point B is also not interpolated as it is not fully covered by the computed values of the narrow-band on the higher level. The remaining grid points C, D, and F are interpolated.

In case of parallelization, there are no restrictions with respect to synchronizations (race conditions) because the nesting criteria of the hierarchical grid enforce that grid points are uniquely refined, i.e., each grid point on the lower level is interpolated by a dedicated set of grid points on the higher level. The interpolated points are first collected in a separate list: One list per block on the higher level.

[1]For the level-set method it is sufficient to solve the level-set equation only in a narrow-band around the interface (on grid points where the absolute distance to the interface is less than a given threshold); this is done for efficiency purposes

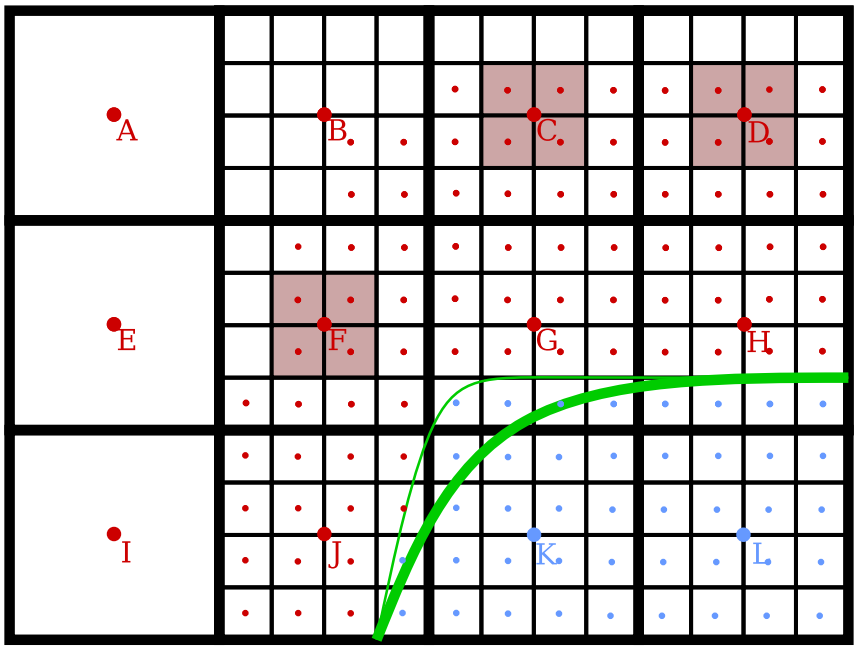


Figure 7.1: The interface (thick green curve) extracted on the lower level differs from the interface on the higher level (thin green curve), because the higher spatial resolution allows for a more accurate representation. Cells are drawn as squares and the corresponding grid points flagged *Known* are in their center (not all grid points on the higher level are *Known*, because only a narrow-band is computed). The points on the lower level are labeled by a letter from A to L. The color of grid points indicates the sign. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Studies in Computational Intelligence* 902 (2021), pp. 438-451. [100], © 2021, under exclusive license to Springer Nature Switzerland AG.

The lists are then merged through a reduction operation, so that for each block on the current level a single list is created containing all interpolated points on the block (Algorithm 10 Line: 8).

Afterwards the merged list of interpolated grid points is used to initialize the multi-block FMM [62]. The initialization is different compared to the multi-block FMM, because all grid points keep their current Φ value and their current flag. Only the grid points from the merged list are set to *Known* and inserted into the *Heap*.

Marching

Subsequently, the unmodified marching of the FMM is performed until all the heaps are empty (cf. Algorithm 10 Line: 12). The specialized initialization avoids the re-computation of all grid points, because a re-computation of all grid points would result in an approximately doubled run-time per level compared to the *Re-Distancing* step without correction algorithm. The main disadvantage of this approach is that grid points for which the distance to the interface has been under-estimated are not corrected. This is inherent to the FMM because grid points are only processed, if their newly computed distance is lower than their current one. However, this is only a minor issue because the FMM tends to over-estimates the distance [169].

After a global synchronization barrier (ensuring all heaps are empty), the algorithm eventually moves to the next level by decreasing the level counter l by one Algorithm 10 Line: 16. The algorithm terminates when *Level 0* is reached and corrected.

7.2 Benchmark Examples and Analyses

The effect of the bottom-up correction algorithm on the accuracy is first analyzed on 2D examples. Additionally, a 3D example is used to study the performance impact.

The accuracy is measured by computing the error (difference of the signed-distance field to an exact solution) in three discrete norms

$$L_1\text{-norm}: \quad \sum_{i\in I} |\Phi_i - \Phi|, \tag{7.1}$$

$$L_2\text{-norm}: \quad \sqrt{\sum_{i\in I} (\Phi_i - \Phi)^2}, \tag{7.2}$$

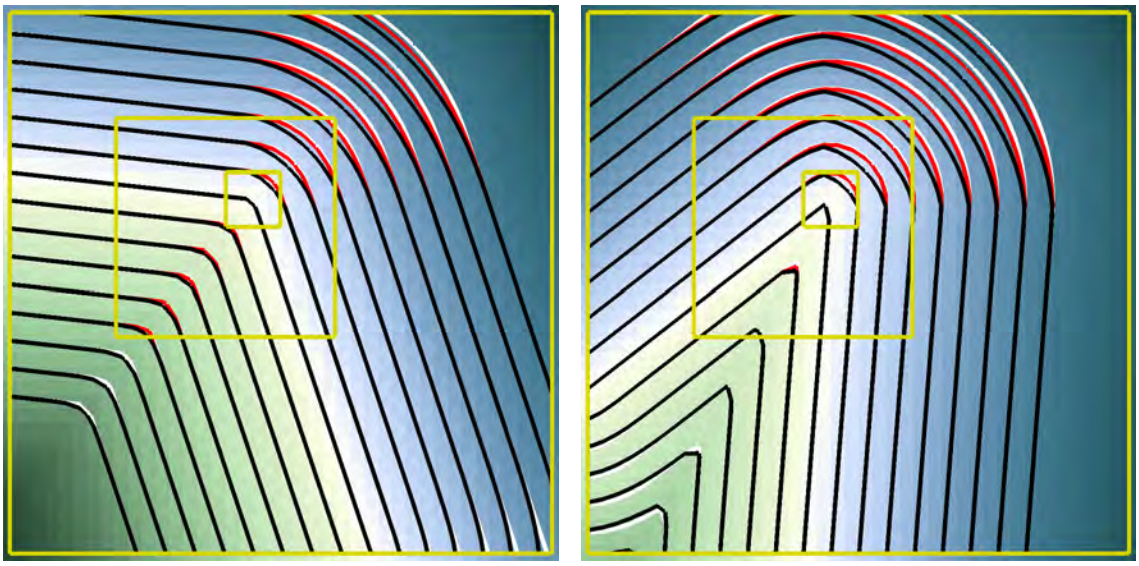
$$L_\infty\text{-norm}: \quad \inf_{i\in I} |\Phi_i - \Phi|. \tag{7.3}$$

Φ is the exact solution, while Φ_i is the discrete approximation and the index set I is given by the considered grid points, i.e., all nodes on a level of the hierarchical grid. The 2D examples consist of two corner examples (*Corner* and *Sharp corner*) providing basic insights on the bottom-up correction algorithm and the *2D Trench* example (a typical geometry in process TCAD) highlights issues arising from thin trenches. The 3D example *3D Trench* is a combination of the trench and corner geometries. Due to its higher computational load it is also used to evaluate the performance impact of the bottom-up correction algorithm.

The domain for all benchmark examples is chosen as $[-1, 1]^d$ with $d \in \{2, 3\}$ with symmetric boundary conditions. For the 2D examples the domain is discretized using 40 grid points in all spatial directions on *Level 0* and a refinement factor of four is used for both refinement level (*Level 1* and *Level 2*). The 3D example has higher spatial discretization (50 grid points) on *Level 0*. The other grid settings are the same as in the 2D examples. The accuracy is compared to the exact solution computed using the explicit representations, i.e., triangles and lines for two and three dimensions, respectively. The exact solution is also used as the initial data of the *Close Points* utilized in the *Re-Distancing* step.

7.2.1 Corner

The *Corner* and *Sharp corner* examples have a corner located near the center of the domain. The corner as well as the sides of the angle are purposely not aligned to the computational grid to account for the generic case of a level-set simulation. The angle of the *Corner* example is 110° and for the *Sharp corner* example 50°. Figure 7.2 shows the level-set values together with several iso-lines extracted form *Level 0* and the block placement (yellow rectangles) for both corner examples.



(a) *Corner* example **(b)** *Sharp Corner* example

Figure 7.2: Iso-lines on *Level 0* (coarsest grid): The Black lines show the solutions based on the top-down approach without the bottom-up correction algorithm, red lines with the bottom-up correction algorithm, and white lines for the exact solution, showing an improvement to the geometry representation. In the background, the green and blue background colors give the distance to the interface. The yellow boxes show the outline of the blocks, there is only a single block on each level. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Studies in Computational Intelligence* 902 (2021), pp. 438-451. [100], © 2021, under exclusive license to Springer Nature Switzerland AG.

On each level of the hierarchical grid only a single block is present, which is located around the corner near the center of the domain. In regions with low curvature only the black iso-line is visible, because all three solutions match and, therefore, their iso-lines overlap.

The symmetric boundary conditions which are applied to the lateral domain dimensions create additional corners at the domain boundary. Around these corners purposely no refinement is made (no block placed), therefore, the signed-distance field is not corrected around those corners: This allows for evaluating the effect precisely for a single corner. Generally speaking, the first-order approximation computed by the FMM over-estimates the distance to the interface for rarefaction fans (reflex angle side) and under-estimates the distance to the interface for shock waves (obtuse and acute angle side). The bottom-up correction algorithm increases the accuracy of the signed-distance field on rarefaction fans even outside the refined regions, due to the marching.

In Table 7.1 and Table 7.2 the error norms and the reduction by the proposed bottom-up correction algorithm are shown. For the *Corner* example the L_1-norm and L_2-norm errors are reduced by a factor of at least 2.1 on *Level 0* and by a factor of 1.8 on *Level 1*. The *Sharp Corner* example shows an even higher reduction, 2.7 on *Level 0* and 2.1 on *Level 1*, because sharper corners benefit more from the bottom-up correction algorithm. On *Level 2* no correction is possible because it is the highest level.

Table 7.1: Error norms for the *Corner* example, with and without the bottom-up correction algorithm applied, and the corresponding factor by which the error norm is reduced. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Studies in Computational Intelligence* 902 (2021), pp. 438-451. [100], © 2021, under exclusive license to Springer Nature Switzerland AG.

Level	L_1-norm	L1-reduc.	L_2-norm	L2-reduc.	L_∞-norm	inf-reduc.
0	5.437e-3		3.260e-4		4.785e-2	
0 corrected	2.491e-3	2.2	1.550e-4	2.1	3.079e-2	1.6
1	1.122e-3		5.101e-5		1.393e-2	
1 corrected	6.035e-4	1.9	2.792e-5	1.8	8.541e-3	1.6
2	5.126e-4		1.819e-5		3.757e-3	

Table 7.2: Error norms for the *Sharp Corner* example, with and without the bottom-up correction algorithm applied, and the corresponding factor by which the error norm is reduced. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Studies in Computational Intelligence* 902 (2021), pp. 438-451. [100], © 2021, under exclusive license to Springer Nature Switzerland AG.

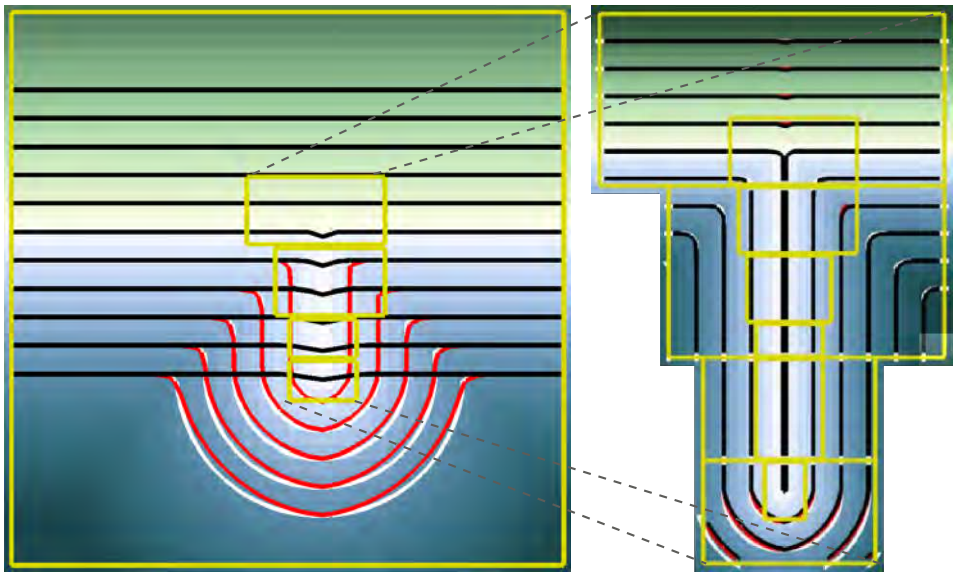
Level	L_1-norm	L1-reduc.	L_2-norm	L2-reduc.	L_∞-norm	inf-reduc.
0	9.110e-3		4.823e-4		6.212e-2	
0 corrected	3.388e-3	2.7	1.812e-4	2.7	2.707e-2	2.3
1	1.894e-3		7.264e-5		1.753e-2	
1 corrected	8.957e-4	2.1	3.484e-5	2.1	9.546e-3	1.8
2	8.866e-4		2.569e-5		4.661e-3	

7.2.2 Two-Dimensional Trench

The *2D Trench* is an axis-aligned thin trench on an otherwise flat surface. The interesting fact about this trench is the small width of only 0.001, because the *Level 0* grid has only a spatial resolution of 0.05, thus is not able to resolve the trench, because no grid points with opposing sign exist along the trench. There are four blocks on *Level 1* and six on *Level 2*, covering the trench completely and, therefore, enable the interface representation.

Such high aspect ratio trenches, are common in semiconductor manufacturing [191, 120] and thus by extension also in process TCAD simulations.

As shown in Figure 7.3 the *Re-distancing* step without the bottom-up correction algorithm shows only a small dent of the level-set on *Level 0*, while with the bottom-up correction algorithm the trench is well-resolved and present. This yields a significant reduction in the measured error norms (cf. Table 7.3): A reduction of 15.3 and 14.4, respectively for the L_1-norm and L_2-norm on *Level 0*. The impact on *Level 1* is less (reduction of the error norms of 1.6) compared to the effect on *Level 0*, because the trench is able to be resolved natively on this level. The accuracy on this level is mainly increased at the rarefaction fans created by the two corners forming the bottom of the trench. The reduction of the error norm is lower compared to the *Corner* examples, because the rarefaction fans cover relatively (to the total number of grid points on a level) a smaller number of grid points.



(a) *Level 0* and *Level 1* **(b)** *Level 1* and *Level 2*

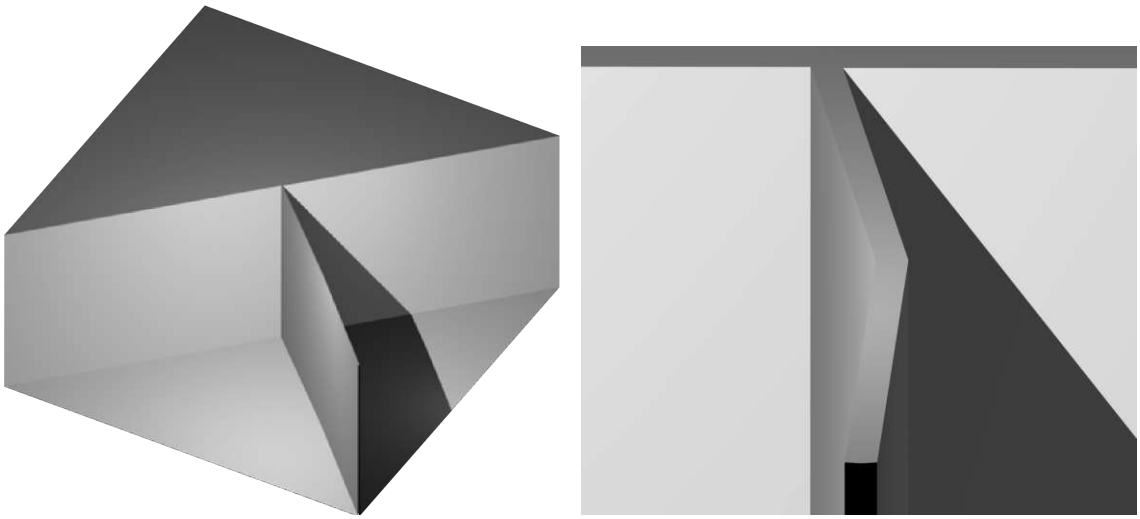
Figure 7.3: Iso-lines for *Level 0* and *Level 1* for the *2D Trench* example. The Black lines show the solutions based on the top-down approach without the bottom-up correction algorithm, red lines with the bottom-up correction algorithm, and white lines for the exact solution. In the background, the green and blue background colors give the distance to the interface. The yellow rectangles show the outline of the blocks on *Level 0* and *Level 1* in **(a)** and *Level 1* and *Level 2* in **(b)**. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Studies in Computational Intelligence* 902 (2021), pp. 438-451. [100], © 2021, under exclusive license to Springer Nature Switzerland AG.

Table 7.3: Error norms for the *2D Trench* example, with and without the bottom-up correction algorithm applied, and the corresponding factor by which the error norm is reduced. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Studies in Computational Intelligence* 902 (2021), pp. 438-451. [100], © 2021, under exclusive license to Springer Nature Switzerland AG.

Level	L_1-norm	L1-reduc.	L_2-norm	L2-reduc.	L_∞-norm	inf-reduc.
0	9.101e-2		4.539e-3		4.839e-1	
0 corrected	5.941e-3	15.3	3.148e-4	14.4	4.005e-2	12.1
1	6.732e-4		5.222e-5		1.262e-2	
1 corrected	4.129e-4	1.6	3.339e-5	1.6	8.398e-3	1.5
2	6.709e-5		4.236e-6		3.400e-3	

7.2.3 Three-Dimensional Trench

The 3D example consists of (i) a step diagonal through the simulation domain and (ii) a thin trench from the center of the domain to one corner. In Figure 7.4 a rendering of the interface representing the trench is shown from the bottom (viewing it from the top would only show the step and a thin line unable to visually grasp the trench). Note that from the bottom perspective the trench looks like a thin fin instead of a thin trench because the difference between a fin and trench is just the viewpoint.



(a) Full view **(b)** Closeup of the dent

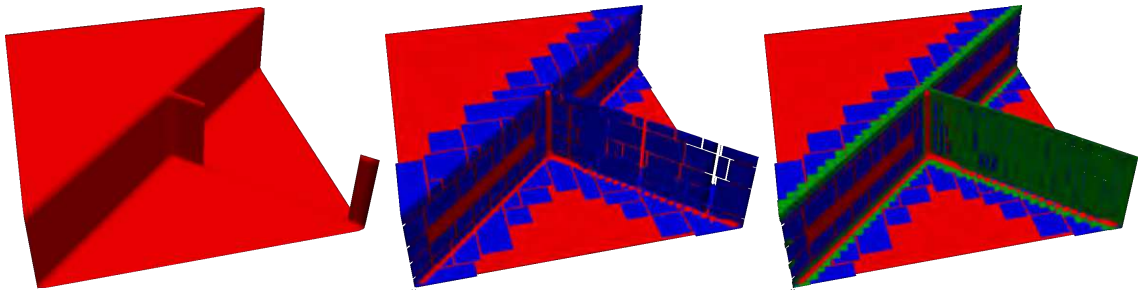
Figure 7.4: **(a)** Rendering of the *3D Trench* example (view from the bottom). **(b)** The dent is only visible from a particular angle. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Studies in Computational Intelligence* 902 (2021), pp. 438-451. [100], © 2021, under exclusive license to Springer Nature Switzerland AG.

The trench, as in the *2D Trench* example, has a width smaller than the spatial resolution of *Level 0*. Additionally, the trench has a slight dent to avoid an alignment with the computational grid, to provide a more challenging test case. The dent is shown in Figure 7.4b, by a close up on the trench (also shown from the bottom). The example is also inspired by the common high aspect ratio geometries in microelectronic devices [192, 193].

Compared to the 2D examples there are significantly more blocks on *Level 1* and *Level 2*, 107 and 591, respectively. Their placement is visualized in Figure 7.5, by showing their individual contributions to the interface. All blocks on *Level 1* have combined 674 496 grid points and on *Level 2* 6 146 688 grid points.

Accuracy Evaluation

The measured error norms are shown in Table 7.4. On *Level 0* the reduction for any error norm is at least a factor of 4.2, because the trench is not resolved on this level (cf. Figure 7.5a, where the extracted interface is based on *Level 0* only). The reductions in the error norm on *Level 1* are similar to the *2D Trench* examples for the L_1-norm and L_2-norm, due to the same reasons (increased accuracy at rarefaction fans at the bottom of the trench). In contrast, the L_∞-norm for *Level 1* is not decreased by applying the bottom-up correction algorithm. The reason for this is that the grid point causing the high L_∞-norm is neither covered directly by a block on the higher level nor in a region where the marching increases the accuracy. Instead, the point is located on a shock wave. The proposed bottom-up correction algorithm is not able to improve the accuracy there. The cause of such errors is that the stencil used for calculating this grid point crosses the skeleton [194, 195].



(a) *Level 0* **(b)** *Level 1* **(c)** *Level 2*

Figure 7.5: Interface representation on the different grid levels (view from the bottom). On *Level 0* (red) the trench is too thin to be resolved by the spatial discretization. *Level 1* (blue) and *Level 2* (green) show the placement of the blocks (small gaps in between). Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Studies in Computational Intelligence* 902 (2021), pp. 438-451. [100], © 2021, under exclusive license to Springer Nature Switzerland AG.

Table 7.4: Error norms for the *3D Trench* example, with and without the bottom-up correction algorithm applied, and the corresponding factor by which the error norm is reduced. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Studies in Computational Intelligence* 902 (2021), pp. 438-451. [100], © 2021, under exclusive license to Springer Nature Switzerland AG.

Level	L_1-norm	L1-reduc.	L_2-norm	L2-reduc.	L_∞-norm	inf-reduc.
0	1.853e-2		1.356e-4		4.593e-1	
0 corrected	4.422e-3	4.2	2.644e-5	5.1	3.478e-2	13.2
1	2.426e-4		1.259e-6		1.418e-2	
1 corrected	1.588e-4	1.5	8.429e-7	1.5	1.418e-2	1.0
2	3.165e-5		7.340e-8		3.017e-3	

The skeleton is the union of all shock waves or, equivalently the skeleton is the union of all points which do not have a unique closest point on the interface. In [100] a skeleton aware approach is suggested (but not further investigated here), which might overcome this issue, by adapting the FMM to only consider grid points which are not separated by the skeleton in addition to the upwind direction.

Performance Evaluation

The *3D Trench* example has a high computational load compared to the previously considered 2D examples. This allows to evaluate the performance impact of the bottom-up correction algorithm by measuring the run-time of the top-down re-distancing algorithm and bottom-up correction algorithm on the compute system VSC3. The block-decomposition proposed in Chapter 6 is not employed, because the focus is on the effects of the proposed bottom-up correction algorithm. Figure 7.6 shows the run-time and parallel speedup for the top-down re-distancing algorithm (Re-Distancing), the bottom-up correction algorithm (Correction), and both together (Total) for all three levels of the hierarchical grid combined. For a single thread the run-time introduced by the additional correction algorithm is increased by 4 % and for 16 threads (full utilization of the compute system) by 10 %.

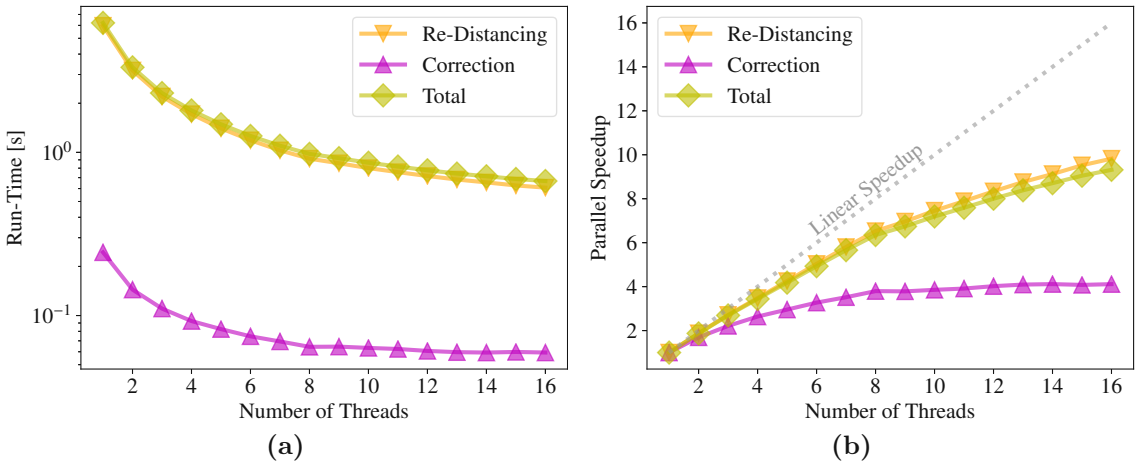


Figure 7.6: **(a)** Run-time and **(b)** parallel speedup (all levels combined) for the top-down re-distancing algorithm and the bottom-up correction algorithm for up to 16 threads using a single compute node of VSC3. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Studies in Computational Intelligence* 902 (2021), pp. 438-451. [100], © 2021, under exclusive license to Springer Nature Switzerland AG.

The overhead introduced by the by the additional correction algorithm is acceptable, because it allows a proper signed-distance field for thin trenches.

A parallel speedup of 9.3 is achieved with 16 threads for the *Re-Distancing* with the bottom-up correction algorithm, which is slightly below the parallel speedup for the *Re-Distancing* without the bottom-up correction algorithm (parallel speedup 9.8). The inferior parallel speedup of 4.1 of the bottom-up correction algorithm alone is caused by the non-parallelized marching on *Level 0* (it contains only a single block). Therefore, a level-by-level comparison is necessary to properly evaluate the performance of the bottom-up correction algorithm.

Level-by-Level Comparison

Figure 7.7 shows the level-by-level comparison, except for *Level 2*, because this is the last level and, as previously mentioned, no correction is performed. The run-time and parallel speedup depend strongly on the level, because the number of blocks and the number of grid points differ significantly.

Starting the investigation on *Level 1*, the bottom-up correction algorithm has a parallel speedup of 7.5, which is similar to the top-down re-distancing algorithm with a parallel speedup of 7.8. The reason for the slightly reduced parallel speedup is the reduced computational load (not all grid points are re-computed). This leads to a larger overhead of the synchronization tasks. Also, for more than eight threads (fully utilizing one of the two processors of the single utilized node of VSC3) the initialization shows no additional parallel speedup because of NUMA limitations (interpolation between blocks occurs across memory domains).

Finalizing the investigation on *Level 0* shows that practically no parallel speedup is observed, because this level only contains a single block.

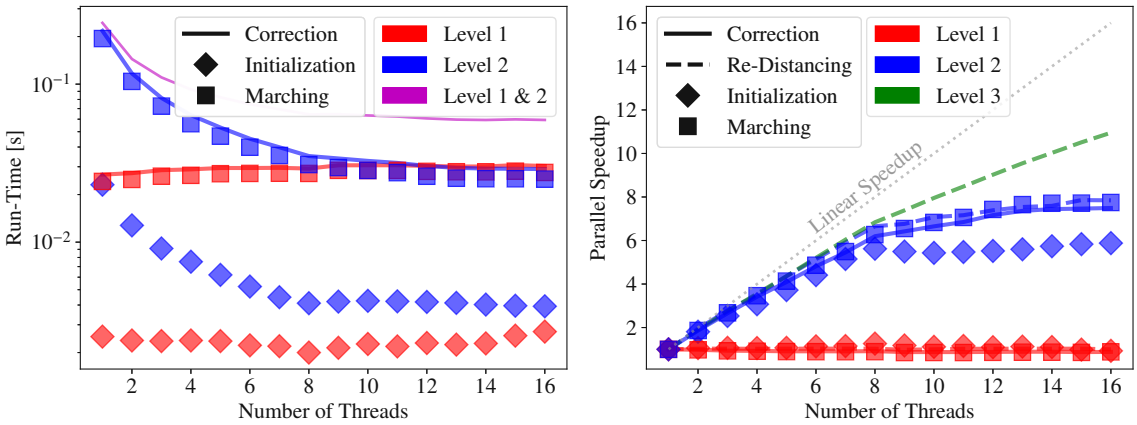


Figure 7.7: **(a)** Run-time and **(b)** parallel speedup per level up to 16 threads using a single compute node of VSC3. The parallel speedup on the same level is similar between the top-down re-distancing algorithm and bottom-up correction algorithm. The initialization and marching times are given for the bottom-up correction algorithm. The marching time dominates the total run-time. Adapted with permission from Springer Nature: Springer Cham, Quell *et al., Studies in Computational Intelligence* 902 (2021), pp. 438-451. [100], © 2021, under exclusive license to Springer Nature Switzerland AG.

However, the block-decomposition proposed in Chapter 6 solves this issue but is not further considered here. Only the initialization benefits slightly from the additional threads, because the interpolation is parallelized on the blocks on *Level 1*. The minimum run-time is reached using eight threads. For more than eight threads NUMA effects increase the run-time again.

7.3 Summary

In this chapter a bottom-up correction algorithm for the computational step *Re-distancing* has been presented. This correction algorithm is tailored towards the FMM and hierarchical grids. The signed-distance field is corrected by interpolation from higher levels of the hierarchical grid and a specialized restarted FMM allows for the bottom-up correction of the signed-distance field even in regions not covered by blocks on higher levels.

The advantageous effect on the accuracy has been evaluated on 2D examples as well on a 3D example. The accuracy of the signed-distance field is significantly increased for a corner by a factor of up to 2.7 (depending on the angel enclosing the corner). In a case where a feature is too small to be represented on a coarse level, i.e., a trench thinner than the grid resolution, the correction algorithm ensures a proper representation. The feature is now represented in the signed-distance field on the coarser level.

For the 3D example the impact of the bottom-up correction algorithm on the computational performance has been evaluated, yielding a run-time overhead compared to the top-down re-distancing algorithm presented in Chapter 6 between 4 % and 10 %, which is an acceptable trade off for the proper representation for thin trenches. The evaluation of the parallelization showed a parallel speedup of 9.3 for 16 threads for all levels combined and a similar parallel speedup as for *Re-Distancing* without the bottom-up correction algorithm for a level-by-level comparison. The block decomposition presented in Chapter 6, may also be employed for the bottom-up correction algorithm to enable a better performance.

Chapter 8

Conclusion and Outlook

This work presents major contributions to accelerate key computational steps of process TCAD simulations based on the level-set method. In particular, the focus is on the computational steps *Velocity Extension* and *Re-Distancing* for which three parallel algorithms, based on the FMM , were developed. Typically, process TCAD simulations require an adaptive spatial discretization to be computationally and practically feasible. The developed algorithms are, therefore, tailored towards hierarchical grids, are able to utilize parallel computational resources, and are thus able to reduce the turnaround time for a wide range of process TCAD simulations. In the following, the key contributions of this work are summarized:

For the computational step *Velocity Extension* an algorithm employing a relaxed computation order for the grid points reducing the computational complexity is derived, resulting in a reduction of the serial run-time (Chapter 5). Three different orders of computation based on different data structures are compared: The *Queue* data structure performs best on a representative set of test cases.

In addition, the changes that enable the reordering of the computation also provide the basis for parallelization of the algorithm on a Cartesian grid enabling a parallel speedup of 5.8 for eight threads. Further optimizations, i.e., tailoring the developed algorithm to hierarchical grids, increase the parallel speedup compared to a previous strategy centered around the so-called multi-block FMM. The increased parallel speedup of up to 7.1 for 10 threads is achieved by reducing global synchronization barriers in the developed algorithm and thus enabling a better load-balancing.

For the computational step *Re-Distancing* an algorithm utilizing block decomposition, which only splits large blocks of a given hierarchical grid, is developed to increase parallel scalability (Chapter 6). The decomposition enables a better implicit load-balancing, by creating a relatively high number of blocks compared to the used number of threads. Additionally, by optimizing the frequency of synchronization steps between the sub-blocks, the overall performance is increased. The so achieved parallel speedup is 17.4 for 24 threads. The performance increase is caused by reducing the influence of ghost points which are not sources for the final signed-distance field.

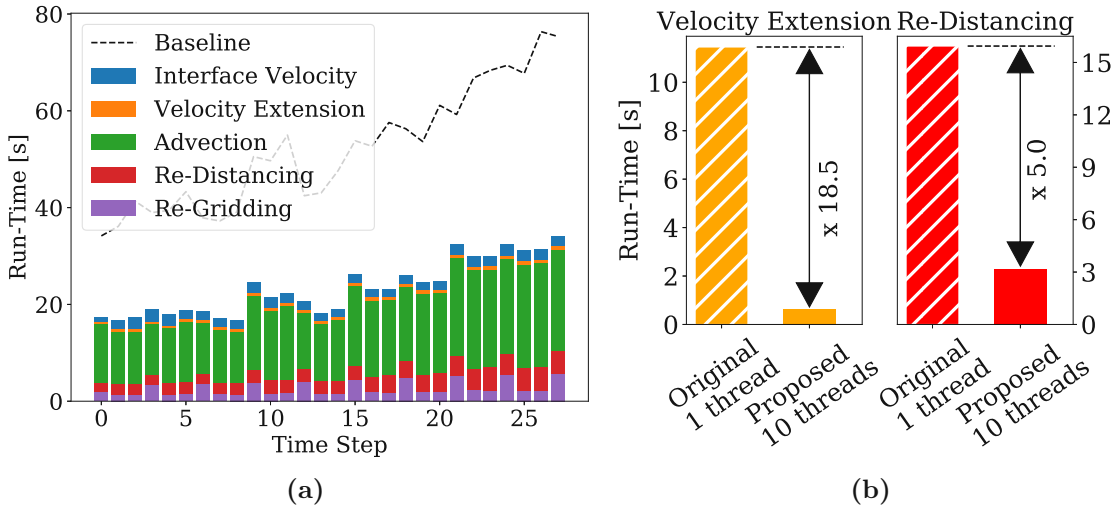


Figure 8.1: **(a)** Run-time of the level-set method utilizing the developed and presented algorithms for the thermal oxidation process shown for individual time steps, measured on the compute system ICS using 10 threads. The dashed line is the baseline (reference) run-time (combining the run-time of all computational steps for each time step as shown in Figure 1.7). **(b)** Highlighting the achievements for *Velocity Extension* and *Re-Distancing*.

The application of the decomposition algorithm prior to the setup of the FMM enables a straightforward extension of this approach to other methods for computing the signed-distance field, e.g., FIM and FSM. Another extension of the research could be considering cases in which the grid resolution along different axes differs strongly. In this case a dedicated block size along each axis might improve the spatial locality of the created blocks, thus potentially further increasing parallel performance.

For the computational step *Re-Distancing* an algorithm, which increases the accuracy of the computed signed-distance field via a bottom-up approach, is proposed (Chapter 7). The algorithm efficiently uses a previous top-down re-distancing algorithm to only re-compute the distance at grid points where an improvement of the accuracy is possible. The proposed bottom-up correction algorithm has a low run-time overhead (between 4 % and 10 %). The accuracy of the signed-distance field is significantly increased, e.g., around corners by a factor of up to 2.7. In cases where features smaller than the grid resolution are present on lower grid levels, e.g., a trench thinner than the grid resolution on *Level 0*, the proposed bottom-up correction algorithm enables correct representation of those features.

Finally, all developed algorithms are applied in combination to the simulation presented as the motivational example (cf. Chapter 1). Figure 8.1 shows a comparison of the original run-time for each time step (cf. Figure 1.7) to the run-time of the reference simulator using the new developed and parallelized algorithms. The overall run-time of the simulation in each time step is more than halved.

Re-Distancing which has previously been the second biggest contributor to the run-time in a time step is reduced to second or third biggest contributor, depending on whether a *Re-Gridding* takes place in the specific time step.

The overall speedup of 5.0 (cf. Figure 8.1b) for *Re-Distancing* is in the expected range. The blocks of the hierarchical grid are relatively small (about 45 grid points wide). For such small blocks the gained parallel performance is only slightly larger than the overhead from decomposition.

The leading contributor to the run-time in a time step is *Advection*, because significant parts of this computational step are not parallelized (and out of scope of this thesis) in the considered reference simulator.

It is especially important to point out that the contribution from *Velocity Extension*, which originally has been the third highest contributor to the run-time, is now the least contributor (using 10 threads). The serial and parallel speedup combined reduced the run-time by a factor of 18.5 (cf. Figure 8.1b). A further reduction of the run-time is possible, if the process model allows a combined extension of the scalar and vector velocity.

Bibliography

[1] A. H. Gencer, A. Lebedev, and P. Pfäffli. "Efficient Full-Flow Process Simulation for 3D Structures Including Stress Modeling". In: *Journal of Computational Electronics* 5.4 (2006), pp. 353–356. DOI: 10.1007/s10825-006-0024-7.

[2] C. K. Maiti. *Introducing Technology Computer-Aided Design (TCAD): Fundamentals, Simulations, and Applications*. Boca Raton: Jenny Stanford Publishing, 2017. DOI: 10.1201/9781315364506.

[3] O. Ertl. "Numerical Methods for Topography Simulation". Doctoral dissertation. TU Wien, 2010. DOI: 10.34726/hss.2010.001.

[4] S. Berrada, H. Carrillo-Nunez, J. Lee, C. Medina-Bailon, T. Dutta, O. Badami, F. Adamu-Lema, V. Thirunavukkarasu, V. Georgiev, and A. Asenov. "Nano-Electronic Simulation Software (NESS): A Flexible Nano-Device Simulation Platform". In: *Journal of Computational Electronics* 19.3 (2020), pp. 1031–1046. DOI: 10.1007/s10825-020-01519-0.

[5] K. Nishi. "Design with Fluctuations of Device Characteristics - TCAD Can Be of Any Help?" In: *Proceedings of the International Conference on ASIC (ASICON)*. Shanghai: IEEE, 2005, pp. 750–755. DOI: 10.1109/ICASIC.2005.1611436.

[6] M. R. Shaeri, T.-C. C. Jen, C. Y. Yuan, and M. Behnia. "Investigating Atomic Layer Deposition Characteristics in Multi-Outlet Viscous Flow Reactors Through Reactor Scale Simulations". In: *International Journal of Heat and Mass Transfer* 89 (2015), pp. 468–481. DOI: 10.1016/j.ijheatmasstransfer.2015.05.079.

[7] S. Osher and J. A. Sethian. "Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations". In: *Journal of Computational Physics* 79.1 (1988), pp. 12–49. DOI: 10.1016/0021-9991(88)90002-2.

[8] D. Adalsteinsson and J. A. Sethian. "A Fast Level Set Method for Propagating Interfaces". In: *Journal of Computational Physics* 118.2 (1995), pp. 269–277. DOI: 10.1006/jcph.1995.1098.

[9] M. Labschutz, S. Bruckner, M. E. Gröller, M. Hadwiger, and P. Rautek. "JiTTree: A Just-in-Time Compiled Sparse GPU Volume Data Structure". In: *IEEE Transactions on Visualization and Computer Graphics* 22.1 (2016), pp. 1025–1034. DOI: 10.1109/TVCG.2015.2467331.

[10] K. Museth. "VDB: High-Resolution Sparse Volumes with Dynamic Topology". In: *ACM Transactions on Graphics* 32.3 (2013), pp. 1–22. DOI: 10.1145/2487228.2487235.

[11] D. Adalsteinsson and J. A. Sethian. "A Level Set Approach to a Unified Model for Etching, Deposition, and Lithography II: Three-Dimensional Simulations". In: *Journal of Computational Physics* 122.2 (1995), pp. 348–366. DOI: 10.1006/jcph.1995.1221.

[12] D. Adalsteinsson and J. A. Sethian. "A Level Set Approach to a Unified Model for Etching, Deposition, and Lithography III: Redeposition, Reemission, Surface Diffusion, and Complex Simulations". In: *Journal of Computational Physics* 138.1 (1997), pp. 193–223. DOI: 10.1006/jcph.1997.5817.

[13] J. A. Sethian and D. Adalsteinsson. "An Overview of Level Set Methods for Etching, Deposition, and Lithography Development". In: *IEEE Transactions on Semiconductor Manufacturing* 10.1 (1997), pp. 167–184. DOI: 10.1109/66.554505.

[14] J. A. Sethian. "Evolution, Implementation, and Application of Level Set and Fast Marching Methods for Advancing Fronts". In: *Journal of Computational Physics* 169.2 (2001), pp. 503–555. DOI: 10.1006/jcph.2000.6657.

[15] V. Suvorov, A. Hössinger, Z. Djurić, and N. Ljepojevic. "A Novel Approach to Three-Dimensional Semiconductor Process Simulation: Application to Thermal Oxidation". In: *Journal of Computational Electronics* 5.4 (2006), pp. 291–295. DOI: 10.1007/s10825-006-0003-z.

[16] B. Radjenovic, M. Radmilovic-Radjenovic, and M. Mitric. "Application of the Level Set Method on the Non-Convex Hamiltonians". In: *Facta Universitatis - Series: Physics, Chemistry and Technology* 7.1 (2009), pp. 33–44. DOI: 10.2298/FUPCT0901033R.

[17] B. Radjenović, M. Radmilović-Radjenović, and M. Mitrić. "Level Set Approach to Anisotropic Wet Etching of Silicon". In: *Sensors* 10.5 (2010), pp. 4950–4967. DOI: 10.3390/s100504950.

[18] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Vol. 153. New York: Springer, 2003. DOI: 10.1007/b98879.

[19] A. S. Bahm. "Predictive Modelling of Gas Assisted Electron and Ion Beam Induced Etching and Deposition". PhD thesis. University of Technology Sydney, 2016.

[20] P. Manstetten. "Efficient Flux Calculations for Topography Simulation". Doctoral dissertation. TU Wien, 2018. DOI: 10.34726/hss.2018.57263.

[21] R. I. Saye and J. A. Sethian. "A Review of Level Set Methods to Model Interfaces Moving under Complex Physics: Recent Challenges and Advances". In: *Handbook of Numerical Analysis*. 1st ed. Oxford: Elsevier, 2020, pp. 509–554. DOI: 10.1016/bs.hna.2019.07.003.

[22] H.-K. Zhao, B. Merriman, S. Osher, and L. Wang. "Capturing the Behavior of Bubbles and Drops Using the Variational Level Set Approach". In: *Journal of Computational Physics* 143.2 (1998), pp. 495–518. DOI: 10.1006/jcph.1997.5810.

[23] F. Losasso, F. Gibou, and R. Fedkiw. "Simulating Water and Smoke with an Octree Data Structure". In: *ACM Transactions on Graphics* 23.3 (2004), pp. 457–462. DOI: 10.1145/1015706.1015745.

[24] F. Losasso, R. Fedkiw, and S. Osher. "Spatially Adaptive Techniques for Level Set Methods and Incompressible Flow". In: *Computers & Fluids* 35.10 (2006), pp. 995–1010. DOI: 10.1016/j.compfluid.2005.01.006.

[25] M. Jemison, E. Loch, M. Sussman, M. Shashkov, M. Arienti, M. Ohta, and Y. Wang. "A Coupled Level Set-Moment of Fluid Method for Incompressible Two-Phase Flows". In: *Journal of Scientific Computing* 54.2-3 (2013), pp. 454–491. DOI: 10.1007/s10915-012-9614-7.

[26] Y. F. Yap, F. M. Vargas, and J. Chai. "A Level-Set Method for Convective-Diffusive Particle Deposition". In: *Applied Mathematical Modelling* 37.7 (2013), pp. 5245–5259. DOI: 10.1016/j.apm.2012.10.039.

[27] A. Sharma. "Level Set Method for Computational Multi-Fluid Dynamics: A Review on Developments, Applications and Analysis". In: *Sadhana* 40.3 (2015), pp. 627–652. DOI: 10.1007/s12046-014-0329-3.

[28] V. T. Nguyen, V. D. Thang, and W. G. Park. "A Novel Sharp Interface Capturing Method for Two- and Three-Phase Incompressible Flows". In: *Computers & Fluids* 172 (2018), pp. 147–161. DOI: 10.1016/j.compfluid.2018.06.020.

[29] K. Luo, C. Shao, M. Chai, and J. Fan. "Level Set Method for Atomization and Evaporation Simulations". In: *Progress in Energy and Combustion Science* 73 (2019), pp. 65–94. DOI: 10.1016/j.pecs.2019.03.001.

[30] T. Du, K. Wu, A. Spielberg, W. Matusik, B. Zhu, and E. Sifakis. "Functional Optimization of Fluidic Devices with Differentiable Stokes Flow". In: *ACM Transactions on Graphics* 39.6 (2020), pp. 1–15. DOI: 10.1145/3414685.3417795.

[31] M. L. Garzon and J. A. Sethian. "Droplet Pairs Electrical Computations Using a Level Set Based Algorithm". In: *Journal of Electrostatics* 106 (2020), p. 103458. DOI: 10.1016/j.elstat.2020.103458.

[32] M. Gao, A. P. Tampubolon, C. Jiang, and E. Sifakis. "An Adaptive Generalized Interpolation Material Point Method for Simulating Elastoplastic Materials". In: *ACM Transactions on Graphics* 36.6 (2017). DOI: 10.1145/3130800.3130879.

[33] J. Liu, Q. Chen, Y. Zheng, R. Ahmad, J. Tang, and Y. Ma. "Level Set-Based Heterogeneous Object Modeling and Optimization". In: *Computer-Aided Design* 110 (2019), pp. 50–68. DOI: 10.1016/j.cad.2019.01.002.

[34] H. Liu, Y. Hu, B. Zhu, W. Matusik, and E. Sifakis. "Narrow-band Topology Optimization on a Sparsely Populated Grid". In: *ACM Transactions on Graphics* 37.6 (2019), pp. 1–14. DOI: 10.1145/3272127.3275012.

[35] Y. Wang, Z. Kang, and P. Liu. "Velocity Field Level-Set Method for Topological Shape Optimization Using Freely Distributed Design Variables". In: *International Journal for Numerical Methods in Engineering* 120.13 (2019), pp. 1411–1427. DOI: 10.1002/nme.6185.

[36] S. Kambampati, C. Jauregui, K. Museth, and H. A. Kim. "Large-Scale Level Set Topology Optimization for Elasticity and Heat Conduction". In: *Structural and Multidisciplinary Optimization* 61.1 (2020), pp. 19–38. DOI: 10.1007/s00158-019-02440-2.

[37] M. Doškář, J. Zeman, D. Rypl, and J. Novák. "Level-Set Based Design of Wang Tiles for Modelling Complex Microstructures". In: *Computer-Aided Design* 123 (2020), p. 102827. DOI: 10.1016/j.cad.2020.102827.

[38] B. Wyvill, A. Guy, and E. Galin. "Extending the CSG Tree. Warping, Blending and Boolean Operations in an Implicit Surface Modeling System". In: *Computer Graphics Forum* 18.2 (1999), pp. 149–158. DOI: 10.1111/1467-8659.00365.

[39] K. Museth, D. E. Breen, R. T. Whitaker, and A. H. Barr. "Level Set Surface Editing Operators". In: *Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. New York: ACM Press, 2002, p. 330. DOI: 10.1145/566570.566585.

[40] K. Museth. "DB+Grid: A Novel Dynamic Blocked Grid For Sparse High-Resolution Volumes and Level Sets". In: *Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. New York: ACM Press, 2011, p. 1. DOI: 10.1145/2037826.2037894.

[41] R. K. Hoetzlein. "GVDB: Raytracing Sparse Voxel Database Structures on the GPU". In: *High Performance Graphics* (2016). DOI: 10.2312/hpg.20161197.

[42] F. Gibou, R. Fedkiw, and S. Osher. "A Review of Level-Set Methods and Some Recent Applications". In: *Journal of Computational Physics* 353 (2018), pp. 82–109. DOI: 10.1016/j.jcp.2017.10.006.

[43] E. Sifakis, C. Garcia, and G. Tziritas. "Bayesian Level Sets for Image Segmentation". In: *Journal of Visual Communication and Image Representation* 13.1-2 (2002), pp. 44–64. DOI: 10.1006/jvci.2001.0474.

[44] L. A Vese and T. F Chan. "A Multiphase Level Set Framework for Image Segmentation Using the Mumford and Shah Model". In: *International Journal of Computer Vision* 50.3 (2002), pp. 271–293. DOI: 10.1023/A:1020874308076.

[45] H. Yang, M. Fuchs, B. Jüttler, O. Scherzer, Huaiping Yang, M. Fuchs, B. Juttler, O. Scherzer, H. Yang, M. Fuchs, B. Jüttler, and O. Scherzer. "Evolution of T-Spline Level Sets with Distance Field Constraints for Geometry Reconstruction and Image Segmentation". In: *Proceedings of IEEE International Conference on Shape Modeling and Applications (SMI)*. Matsushima: IEEE, 2006, pp. 37–37. DOI: 10.1109/SMI.2006.12.

[46] L.-T. Cheng, J. Dzubiella, J. A. McCammon, and B. Li. "Application of the Level-Set Method to the Implicit Solvation of Nonpolar Molecules". In: *The Journal of Chemical Physics* 127.8 (2007), p. 084503. DOI: 10.1063/1.2757169.

[47] S. Zhou, L.-T. Cheng, H. Sun, J. Che, J. Dzubiella, B. Li, and J. A. McCammon. "LS-VISM: A software Package for Analysis of Biomolecular Solvation". In: *Journal of Computational Chemistry* 36.14 (2015), pp. 1047–1059. DOI: https://doi.org/10.1002/jcc.23890.

[48] Z. Zhang, C. G. Ricci, C. Fan, L.-T. T. Cheng, B. Li, and J. A. McCammon. "Coupling Monte Carlo, Variational Implicit Solvation, and Binary Level-Set for Simulations of Biomolecular Binding". In: *Journal of Chemical Theory and Computation* 17.4 (2021), acs.jctc.0c01109. DOI: 10.1021/acs.jctc.0c01109.

[49] T. Thurgate. "Segment-Based Etch Algorithm and Modeling". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10.9 (1991), pp. 1101–1109. DOI: 10.1109/43.85756.

[50] M. E. Law. "Grid Adaption near Moving Boundaries in two Dimensions for IC Process Simulation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 14.10 (1995), pp. 1223–1230. DOI: 10.1109/43.466338.

[51] M. E. Law and S. M. Cea. "Continuum Based Modeling of Silicon Integrated Circuit Processing: An Object Oriented Approach". In: *Computational Materials Science* 12.4 (1998), pp. 289–308. DOI: 10.1016/S0927-0256(98)00020-2.

[52] *AMD Ryzen Threadripper 3990X*. https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3990x. (accessed November 2, 2021).

[53] *Intel Xeon Platinum 9282*. https://www.intel.com/content/www/us/en/products/sku/194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz/specifications.html. (accessed November 2, 2021).

[54] B. El-Kareh. *Fundamentals of Semiconductor Processing Technology*. Boston: Springer, 1995. DOI: 10.1007/978-1-4615-2209-6.

[55] M. Quell, V. Suvorov, A. Hössinger, and J. Weinbub. "Parallel Velocity Extension for Level-Set-Based Material Flow on Hierarchical Meshes in Process TCAD". In: *IEEE Transactions on Electron Devices* 68.11 (2021), pp. 5430–5437. DOI: 10.1109/TED.2021.3087451.

[56] D. Guoy, A. H. Gencer, Z. Tan, S. Chalasani, M. Johnson, L. Villablanca, and S. Simeonov. "3-D Simulation of Silicon Oxidation: Challenges, Progress and Results". In: *Proceedings of the International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*. Glasgow: IEEE, 2013, pp. 196–199. DOI: 10.1109/SISPAD.2013.6650608.

[57] W. Joppich and S. Mijalković. *Multigrid Methods for Process Simulation*. Vienna: Springer, 1993. DOI: 10.1007/978-3-7091-9253-5.

[58] O. Ertl and S. Selberherr. "A Fast Level Set Framework for Large Three-Dimensional Topography Simulations". In: *Computer Physics Communications* 180.8 (2009), pp. 1242–1250. DOI: 10.1016/j.cpc.2009.02.002.

[59] M. Mirzadeh, A. Guittet, C. Burstedde, and F. Gibou. "Parallel Level-Set Methods on Adaptive Tree-Based Grids". In: *Journal of Computational Physics* 322 (2016), pp. 345–364. DOI: 10.1016/j.jcp.2016.06.017.

[60] J. A. Sethian. "A Fast Marching Level Set Method for Monotonically Advancing Fronts". In: *Proceedings of the National Academy of Sciences* 93.4 (1996), pp. 1591–1595. DOI: 10.1073/pnas.93.4.1591.

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

[62] G. Diamantopoulos, A. Hössinger, S. Selberherr, and J. Weinbub. "A Shared Memory Parallel Multi-Mesh Fast Marching Method for Re-Distancing". In: *Advances in Computational Mathematics* 45.4 (2019), pp. 2029–2045. DOI: 10.1007/s10444-019-09683-z.

[63] J. F. Thompson, B. K. Soni, and N. P. Weatherill. *Handbook of Grid Generation*. Boca Raton: CRC Press, 1998. DOI: 10.1201/9781420050349.

[64] J. C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. Madison: Society for Industrial and Applied Mathematics, 2004. DOI: 10.1137/1.9780898717938.

[65] R. E. White. *An Introduction to the Finite Element Method with Applications to Nonlinear Problems*. New York: Wiley, 1985.

[66] R. Eymard, T. Gallouët, and R. Herbin. "Finite Volume Methods". In: *Handbook of Numerical Analysis*. Oxford: Elsevier, 2000, pp. 713–1018. DOI: https://doi.org/10.1016/S1570-8659(00)07005-8.

[67] J. A. Sethian. "Fast Marching Methods". In: *SIAM Review* 41.2 (1999), pp. 199–235. DOI: 10.1137/S0036144598347059.

[68] X. Yang, A. J. James, J. Lowengrub, X. Zheng, and V. Cristini. "An Adaptive Coupled Level-Set/Volume-of-Fluid Interface Capturing Method for Unstructured Triangular Grids". In: *Journal of Computational Physics* 217.2 (2006), pp. 364–394. DOI: 10.1016/j.jcp.2006.01.007.

[69] M. A. Herrmann. “A Balanced Force Refined Level Set Grid Method for Two-Phase Flows on Unstructured Flow Solver Grids”. In: *Journal of Computational Physics* 227.4 (2008), pp. 2674–2706. DOI: 10.1016/j.jcp.2007.11.002.

[70] R. Abgrall, H. Beaugendre, and C. Dobrzynski. “An Immersed Boundary Method Using Unstructured Anisotropic Mesh Adaptation Combined with Level-Sets and Penalization Techniques”. In: *Journal of Computational Physics* 257.PA (2014), pp. 83–101. DOI: 10.1016/j.jcp.2013.08.052.

[71] N. R. Morgan and J. I. Waltz. “3D Level Set Methods for Evolving Fronts on Tetrahedral Meshes with Adaptive Mesh Refinement”. In: *Journal of Computational Physics* 336 (2017), pp. 492–512. DOI: 10.1016/j.jcp.2017.02.030.

[72] M. Quezada de Luna, D. Kuzmin, and C. E. Kees. “A Monolithic Conservative Level Set Method with Built-In Redistancing”. In: *Journal of Computational Physics* 379 (2019), pp. 262–278. DOI: 10.1016/j.jcp.2018.11.044.

[73] G.-S. Jiang and D. Peng. “Weighted ENO Schemes for Hamilton–Jacobi Equations”. In: *SIAM Journal on Scientific Computing* 21.6 (2000), pp. 2126–2143. DOI: 10.1137/S106482759732455X.

[74] S. Serna and J. Qian. “Fifth-Order Weighted Power-ENO Schemes for Hamilton-Jacobi Equations”. In: *Journal of Scientific Computing* 29.1 (2006), pp. 57–81. DOI: 10.1007/s10915-005-9015-2.

[75] X.-D. Liu, S. Osher, and T. Chan. “Weighted Essentially Non-Oscillatory Schemes”. In: *Journal of Computational Physics* 115.1 (1994), pp. 200–212. DOI: 10.1006/jcph.1994.1187.

[76] C.-W. Shu. “High Order Numerical Methods for Time Dependent Hamilton-Jacobi Equations”. In: *Mathematics and Computation in Imaging Science and Information Processing*. Singapore: National University of Singapore, 2010, pp. 47–91. DOI: 10.1142/9789812709066_0002.

[77] M. J. Berger and J. Oliger. “Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations”. In: *Journal of Computational Physics* 53.3 (1984), pp. 484–512. DOI: 10.1016/0021-9991(84)90073-1.

[78] M. J. Berger and P. Colella. “Local Adaptive Mesh Refinement for Shock Hydrodynamics”. In: *Journal of Computational Physics* 82.1 (1989), pp. 64–84. DOI: 10.1016/0021-9991(89)90035-1.

[79] J. Bell, M. J. Berger, J. Saltzman, and M. Welcome. “Three-Dimensional Adaptive Mesh Refinement for Hyperbolic Conservation Laws”. In: *SIAM Journal on Scientific Computing* 15.1 (1994), pp. 127–138. DOI: 10.1137/0915008.

[80] K. G. Powell, P. L. Roe, and J. Quirk. “Adaptive-Mesh Algorithms for Computational Fluid Dynamics”. In: *Algorithmic Trends in Computational Fluid Dynamics*. New York: Springer, 1993, pp. 303–337. DOI: 10.1007/978-1-4612-2708-3.

[81] W. J. Coirier and K. G. Powell. “Solution-Adaptive Cartesian Cell Approach for Viscous and Inviscid Flows”. In: *AIAA Journal* 34.5 (1996), pp. 938–945. DOI: 10.2514/3.13171.

[82] J. Strain. “Tree Methods for Moving Interfaces”. In: *Journal of Computational Physics* 151.2 (1999), pp. 616–648. DOI: 10.1006/jcph.1999.6205.

[83] V. Sochnikov and S. Efrima. “Level Set Calculations of the Evolution of Boundaries on a Dynamically Adaptive Grid”. In: *International Journal for Numerical Methods in Engineering* 56.13 (2003), pp. 1913–1929. DOI: 10.1002/nme.641.

[84] N. Shervani-Tabar and O. V. Vasilyev. “Stabilized Conservative Level Set Method”. In: *Journal of Computational Physics* 375 (2018), pp. 1033–1044. DOI: 10.1016/j.jcp.2018.09.020.

[85] C. Min and F. Gibou. “A Second Order Accurate Level Set Method on Non-Graded Adaptive Cartesian Grids”. In: *Journal of Computational Physics* 225.1 (2007), pp. 300–321. DOI: 10.1016/j.jcp.2006.11.034.

[86] H. Kim and M.-S. S. Liou. “Accurate Adaptive Level Set Method and Sharpening Technique for Three Dimensional Deforming Interfaces”. In: *Computers & Fluids* 44.1 (2011), pp. 111–129. DOI: 10.1016/j.compfluid.2010.12.020.

[87] S. Péron and C. Benoit. “Automatic Off-Body Overset Adaptive Cartesian Mesh Method Based on an Octree Approach”. In: *Journal of Computational Physics* 232.1 (2013), pp. 153–173. DOI: 10.1016/j.jcp.2012.07.029.

[88] Q. F. Stout, D. L. De Zeeuw, T. I. Gombosi, C. P. T. Groth, H. G. Marshall, and K. G. Powell. “Adaptive Blocks: A High Performance Data Structure”. In: *Proceedings of ACM/IEEE Conference on Supercomputing (SC)*. New York: ACM Press, 1997, pp. 1–10. DOI: 10.1145/509593.509650.

[89] M. Parashar and J. C. Browne. “On Partitioning Dynamic Adaptive Grid Hierarchies”. In: *Proceedings of the International Conference on System Sciences*. Wailea: IEEE, 1996, pp. 604–613. DOI: 10.1109/HICSS.1996.495511.

[90] F. Golay, M. Ersoy, L. Yushchenko, and D. Sous. “Block-Based Adaptive Mesh Refinement Scheme Using Numerical Density of Entropy Production for Three-Dimensional Two-Fluid Flows”. In: *International Journal of Computational Fluid Dynamics* 29.1 (2015), pp. 67–81. DOI: 10.1080/10618562.2015.1012161.

[91] K. Wu, N. Truong, C. Yuksel, and R. Hoetzlein. “Fast Fluid Simulations with Sparse Volumes on the GPU”. In: *Computer Graphics Forum* 37.2 (2018), pp. 157–167. DOI: 10.1111/cgf.13350.

[92] M. Adams, P. Colella, D. Graves, J. Johnson, N. Keen, T. Ligocki, D. Martin, P. McCorquodale, D. Modiano, T. Schwartz, P.O. Sternberg, and B. van Straalen. "Chombo Software Package for AMR Applications - Design Document". In: *Lawrence Berkeley Natl. Lab. Tech. Rep. LBNL-6616E* (2015).

[93] B. T. Gunney and R. W. Anderson. "Advances in Patch-Based Adaptive Mesh Refinement Scalability". In: *Journal of Parallel and Distributed Computing* 89 (2016), pp. 65–84. DOI: 10.1016/j.jpdc.2015.11.005.

[94] M. J. Berger and I. Rigoutsos. "An Algorithm for Point Clustering and Grid Generation". In: *IEEE Transactions on Systems, Man, and Cybernetics* 21.5 (1991), pp. 1278–1286. DOI: 10.1109/21.120081.

[95] M. Sussman, A. S. Almgren, J. B. Bell, P. Colella, L. H. Howell, and M. L. Welcome. "An Adaptive Level Set Approach for Incompressible Two-Phase Flows". In: *Journal of Computational Physics* 148.1 (1999), pp. 81–124. DOI: 10.1006/jcph.1998.6106.

[96] R. R. Nourgaliev, S. Wiri, N. T. Dinh, and T. G. Theofanous. "On Improving Mass Conservation of Level Set by Reducing Spatial Discretization Errors". In: *International Journal of Multiphase Flow* 31.12 (2005), pp. 1329–1336. DOI: 10.1016/j.ijmultiphaseflow.2005.08.003.

[97] O. Desjardins and H. Pitsch. "A Spectrally Refined Interface Approach for Simulating Multiphase Flows". In: *Journal of Computational Physics* 228.5 (2009), pp. 1658–1677. DOI: 10.1016/j.jcp.2008.11.005.

[98] E. Brun, A. Guittet, and F. Gibou. "A Local Level-Set Method Using a Hash Table Data Structure". In: *Journal of Computational Physics* 231.6 (2012), pp. 2528–2536. DOI: 10.1016/j.jcp.2011.12.001.

[99] B. Houston, M. B. Nielsen, C. Batty, O. Nilsson, and K. Museth. "Hierarchical RLE Level Set: A Compact and Versatile Deformable Surface Representation". In: *ACM Transactions on Graphics* 25.1 (2006), pp. 151–175. DOI: 10.1145/1122501.1122508.

[100] M. Quell, G. Diamantopoulos, A. Hössinger, S. Selberherr, and J. Weinbub. "Parallel Correction for Hierarchical Re-Distancing Using the Fast Marching Method". In: *Advances in High Performance Computing*. Cham: Springer, 2021, pp. 438–451. DOI: 10.1007/978-3-030-55347-0_37.

[101] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, B. O'Shea, E. Schnetter, B. Van Straalen, and K. Weide. "A Survey of High Level Frameworks in Block-Structured Adaptive Mesh Refinement Packages". In: *Journal of Parallel and Distributed Computing* 74.12 (2014), pp. 3217–3227. DOI: 10.1016/j.jpdc.2014.07.001.

[102] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th. San Francisco: Morgan Kaufmann Publishers Inc., 2017.

[103] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. 2nd. Burlington: Elsevier, 2021. DOI: 10.1016/C2011-0-06993-4.

[104] L. Dagum and R. Menon. "OpenMP: An Industry Standard API for Shared-Memory Programming". In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[105] *Vienna Scientific Cluster*. https://vsc.ac.at/. (accessed November 2, 2021).

[106] M. W. Jones, J. A. Baerentzen, and M. Sramek. "3D Distance Fields: A Survey of Techniques and Applications". In: *IEEE Transactions on Visualization and Computer Graphics* 12.4 (2006), pp. 581–599. DOI: 10.1109/TVCG.2006.56.

[107] H.-K. K. Zhao, T. Chan, B. Merriman, and S. J. Osher. "A Variational Level Set Approach to Multiphase Motion". In: *Journal of Computational Physics* 127.1 (1996), pp. 179–195. DOI: 10.1006/jcph.1996.0167.

[108] S. J. Ruuth. "A Diffusion-Generated Approach to Multiphase Motion". In: *Journal of Computational Physics* 145.1 (1998), pp. 166–192. DOI: 10.1006/jcph.1998.6028.

[109] K. Smith, F. Solis, and D. Chopp. "A Projection Method for Motion of Triple Junctions by Level Sets". In: *Interfaces and Free Boundaries* 4.3 (2002), pp. 263–276. DOI: 10.4171/IFB/61.

[110] H. Li, Y. F. Yap, J. Lou, and Z. Shang. "Numerical Modelling of Three-Fluid Flow Using the Level-Set Method". In: *Chemical Engineering Science* 126 (2015), pp. 224–236. DOI: 10.1016/j.ces.2014.11.062.

[111] D. P. Starinshak, S. Karni, and P. L. Roe. "A New Level Set Model for Multimaterial Flows". In: *Journal of Computational Physics* 262 (2014), pp. 1–16. DOI: 10.1016/j.jcp.2013.12.036.

[112] A. Toifl, M. Quell, X. Klemenschits, P. Manstetten, A. Hössinger, S. Selberherr, and J. Weinbub. "The Level-Set Method for Multi-Material Wet Etching and Non-Planar Selective Epitaxy". In: *IEEE Access* 8 (2020), pp. 115406–115422. DOI: 10.1109/ACCESS.2020.3004136.

[113] Á. Montoliu, N. Ferrando, M. A. Gosálvez, J. Cerdá, and R. J. Colom. "Implementation and Evaluation of the Level Set Method: Towards Efficient and Accurate Simulation of Wet Etching for Microengineering Applications". In: *Computer Physics Communications* 184.10 (2013), pp. 2299–2309. DOI: 10.1016/j.cpc.2013.05.016.

[114] Á. Montoliu, N. Ferrando, M. A. Gosálvez, J. Cerdá, R. J. Colom, C. Montoliu, N. Ferrando, M. A. Gosálvez, J. Cerdá, and R. J. Colom. "Level Set Implementation for the Simulation of Anisotropic Etching: Application to Complex MEMS Micromachining". In: *Journal of Micromechanics and Microengineering* 23.7 (2013), p. 075017. DOI: 10.1088/0960-1317/23/7/075017.

[115] A. Toifl, M. Quell, A. Hössinger, A. Babayan, S. Selberherr, and J. Weinbub. "Novel Numerical Dissipation Scheme for Level-Set Based Anisotropic Etching Simulations". In: *Proceedings of the International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*. Udine: IEEE, 2019, pp. 1–4. DOI: 10.1109/SISPAD.2019.8870443.

[116] M. M. Smiljanić, Ž. Lazić, B. Radjenović, M. Radmilović-Radjenović, and V. Jović. "Evolution of Si Crystallographic Planes-Etching of Square and Circle Patterns in 25 wt % TMAH". In: *Micromachines* 10.2 (2019), pp. 26–32. DOI: 10.3390/mi10020102.

[117] H. Liao and T. S. Cale. "Three-Dimensional Simulation of an Isolation Trench Refill Process". In: *Thin Solid Films* 236.1-2 (1993), pp. 352–358. DOI: 10.1016/0040-6090(93)90695-L.

[118] X. Klemenschits, S. Selberherr, and L. Filipovic. "Modeling of Gate Stack Patterning for Advanced Technology Nodes: A Review". In: *Micromachines* 9.12 (2018), p. 631. DOI: 10.3390/mi9120631.

[119] J.-C. Yu, Z.-F. Zhou, J.-L. Su, C.-F. Xia, X.-W. Zhang, Z.-Z. Wu, and Q.-A. Huang. "Three-Dimensional Simulation of DRIE Process Based on the Narrow Band Level Set and Monte Carlo Method". In: *Micromachines* 9.2 (2018), p. 74. DOI: 10.3390/mi9020074.

[120] A. Yanguas-Gil. *Growth and Transport in Nanostructured Materials*. Cham: Springer, 2017. DOI: 10.1007/978-3-319-24672-7.

[121] R. Malladi, J. A. Sethian, and B. C. B. Vemuri. "Shape Modeling with Front Propagation: A Level Set Approach". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17.2 (1995), pp. 158–175. DOI: 10.1109/34.368173.

[122] D. Adalsteinsson and J. A. Sethian. "The Fast Construction of Extension Velocities in Level Set Methods". In: *Journal of Computational Physics* 148.1 (1999), pp. 2–22. DOI: 10.1006/jcph.1998.6090.

[123] C.-W. Shu and S. Osher. "Efficient Implementation of Essentially Non-Oscillatory Shock-Capturing Schemes". In: *Journal of Computational Physics* 77.2 (1988), pp. 439–471. DOI: 10.1016/0021-9991(88)90177-5.

[124] R. J. Spiteri and S. J. Ruuth. "A New Class of Optimal High-Order Strong-Stability-Preserving Time Discretization Methods". In: *SIAM Journal on Numerical Analysis* 40.2 (2002), pp. 469–491. DOI: 10.1137/S0036142901389025.

[125] M. G. Crandall and P.-L. Lions. "Two Approximations of Solutions of Hamilton-Jacobi Equations". In: *Mathematics of Computation* 43.167 (1984), pp. 1–1. DOI: 10.1090/S0025-5718-1984-0744921-8.

[126] S. K. Godunov. "A Finite Difference Method for the Computation of Discontinuous Solutions of the Equations of Fluid Dynamics." In: *Sbornik: Mathematics* 47.8-9 (1959), pp. 357–393.

[127] S. Osher and C.-W. Shu. "High-Order Essentially Nonoscillatory Schemes for Hamilton–Jacobi Equations". In: *SIAM Journal on Numerical Analysis* 28.4 (1991), pp. 907–922. DOI: 10.1137/0728049.

[128] M. F. Trujillo, L. Anumolu, and D. Ryddner. "The Distortion of the Level Set Gradient Under Advection". In: *Journal of Computational Physics* 334 (2017), pp. 81–101. DOI: 10.1016/j.jcp.2016.11.050.

[129] L. C. Evans. *Partial Differential Equations*. Berkeley: Graduate Studies in Mathematics, 1998. DOI: 10.1090/gsm/019.

[130] W. E. Lorensen and H. E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. New York: ACM Press, 1987, pp. 163–169. DOI: 10.1145/37401.37422.

[131] Y. Shen, Y. Ren, and H. Ding. "A 3D Conservative Sharp Interface Method for Simulation of Compressible Two-Phase Flows". In: *Journal of Computational Physics* 403 (2020), p. 109107. DOI: 10.1016/j.jcp.2019.109107.

[132] T. S. Newman and H. Yi. "A Survey of the Marching Cubes Algorithm". In: *Computers & Graphics* 30.5 (2006), pp. 854–879. DOI: 10.1016/j.cag.2006.07.021.

[133] L. Gnam. "High Performance Mesh Adaptation for Technology Computer-Aided Design". Doctoral dissertation. TU Wien, 2020. DOI: 10.34726/hss.2020.76784.

[134] *TCAD - Sentaurus Process*. https://www.synopsys.com/silicon/tcad/process-simulation/sentaurus-process.html. (accessed November 2, 2021).

[135] *Silvaco Victory Process*. https://silvaco.com/tcad/victory-process-3d/. (accessed November 2, 2021).

[136] O. Ertl, L. Filipovic, P. Manstetten, X. Klemenschits, and J. Weinbub. *ViennaTS - The Vienna Topography Simulator*. (accessed November 2, 2021). URL: https://github.com/viennats/viennats-dev.

[137] M. Quell, A. Toifl, A. Hössinger, S. Selberherr, and J. Weinbub. "Parallelized Level-Set Velocity Extension Algorithm for Nanopatterning Applications". In: *Proceedings of the International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*. Udine: IEEE, 2019, pp. 1–4. DOI: 10.1109/SISPAD.2019.8870482.

[138] M. Quell, P. Manstetten, A. Hössinger, S. Selberherr, and J. Weinbub. "Parallelized Construction of Extension Velocities for the Level-Set Method". In: *Lecture Notes in Computer Science*. Cham: Springer, 2020, pp. 348–358. DOI: 10.1007/978-3-030-43229-4_30.

[139] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1.1 (1959), pp. 269–271. DOI: 10.1007/BF01386390.

[140] D. L. Chopp. “Another Look at Velocity Extensions in the Level Set Method”. In: *SIAM Journal on Scientific Computing* 31.5 (2009), pp. 3255–3273. DOI: 10.1137/070686329.

[141] G. F. Ouyang, Y. C. Kuang, and X. M. Zhang. “A Fast Scanning Algorithm for Extension Velocities in Level Set Methods”. In: *Advanced Materials Research* 328.1 (2011), pp. 677–680. DOI: 10.4028/www.scientific.net/AMR.328-330.677.

[142] F. de Gournay and D. E. Gournay. “Velocity Extension for the Level-Set Method and Multiple Eigenvalues in Shape Optimization”. In: *SIAM Journal on Control and Optimization* 45.1 (2006), pp. 343–367. DOI: 10.1137/050624108.

[143] T. J. Moroney, D. R. Lusmore, S. W. McCue, and D. L. McElwain. “Extending Fields in a Level-Set Method by Solving a Biharmonic Equation”. In: *Journal of Computational Physics* 343 (2017), pp. 170–185. DOI: 10.1016/j.jcp.2017.04.049.

[144] T. Aslam, S. Luo, and H. Zhao. “A Static PDE Approach for MultiDimensional Extrapolation Using Fast Sweeping Methods”. In: *SIAM Journal on Scientific Computing* 36.6 (2014), A2907–A2928. DOI: 10.1137/140956919.

[145] D. F. Richards, M. O. Bloomfield, S. Sen, and T. S. Cale. “Extension Velocities for Level Set Based Surface Profile Evolution”. In: *Journal of Vacuum Science & Technology A: Vacuum, Surfaces, and Films* 19.4 (2001), pp. 1630–1635. DOI: 10.1116/1.1380230.

[146] J. V. Gomez, D. Alvarez, S. Garrido, and L. Moreno. “Fast Methods for Eikonal Equations: An Experimental Survey”. In: *IEEE Access* 7 (2019), pp. 39005–39029. DOI: 10.1109/ACCESS.2019.2906782.

[147] T. Hagerup and M. Maas. “Generalized Topological Sorting in Linear Time”. In: *Nordic Journal of Computing* 1.710 (1993), pp. 279–288. DOI: 10.1007/3-540-57163-9_23.

[148] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. Cambridge: MIT Press, 2009.

[149] S. Bhatti, R. Sbiaa, A. Hirohata, H. Ohno, S. Fukami, and S. N. Piramanayagam. “Spintronics Based Random Access Memory: A Review”. In: *Materials Today* 20.9 (2017), pp. 530–548. DOI: 10.1016/j.mattod.2017.07.007.

[150] D. Apalkov, B. Dieny, and J. M. Slaughter. “Magnetoresistive Random Access Memory”. In: *Proceedings of the IEEE* 104.10 (2016), pp. 1796–1830. DOI: 10.1109/JPROC.2016.2590142.

[151] V. T. Nguyen, P. Sabon, J. Chatterjee, L. Tille, P. V. Coelho, S. Auffret, R. Sousa, L. Prejbeanu, E. Gautier, L. Vila, and B. Dieny. "Novel Approach for Nano-Patterning Magnetic Tunnel Junctions Stacks at Narrow Pitch: A Route Towards high Density STT-MRAM Applications". In: *Proceedings of IEEE International Electron Devices Meeting (IEDM)*. San Francisco: IEEE, 2017, pp. 38.5.1–38.5.4. DOI: 10.1109/IEDM.2017.8268517.

[152] T. Endoh and H. Honjo. "A Recent Progress of Spintronics Devices for Integrated Circuit Applications". In: *Journal of Low Power Electronics and Applications* 8.4 (2018), p. 44. DOI: 10.3390/jlpea8040044.

[153] T. Hanyu, T. Endoh, D. Suzuki, H. Koike, Y. Ma, N. Onizawa, M. Natsui, S. Ikeda, and H. Ohno. "Standby-Power-Free Integrated Circuits Using MTJ-Based VLSI Computing". In: *Proceedings of the IEEE* 104.10 (2016), pp. 1844–1863. DOI: 10.1109/JPROC.2016.2574939.

[154] M. Pak, W. Zanders, P. Wong, and S. Halder. "Comparison of Different Lithography Approaches for STT-MRAM Orthogonal Array MTJ Pillars". In: *Micro and Nano Engineering* 10 (2021), p. 100082. DOI: 10.1016/j.mne.2021.100082.

[155] A. Khvalkovskiy, D. Apalkov, S. Watts, R. Chepulskii, R. S. Beach, A. Ong, X. Tang, A. Driskill-Smith, W. H. Butler, P. B. Visscher, D. Lottis, E. Chen, V. Nikitin, and M. Krounbi. "Basic Principles of STT-MRAM Cell Operation in Memory Arrays". In: *Journal of Physics D: Applied Physics* 46.7 (2013), p. 074001. DOI: 10.1088/0022-3727/46/7/074001.

[156] M. Gajek, J. J. Nowak, J. Z. Sun, P. L. Trouilloud, E. J. O'Sullivan, D. W. Abraham, M. C. Gaidis, G. Hu, S. Brown, Y. Zhu, R. P. Robertazzi, W. J. Gallagher, and D. C. Worledge. "Spin Torque Switching of 20 nm Magnetic Tunnel Junctions with Perpendicular Anisotropy". In: *Applied Physics Letters* 100.13 (2012), p. 132408. DOI: 10.1063/1.3694270.

[157] V. Ip, S. Huang, S. D. Carnevale, I. L. Berry, K. Rook, T. B. Lill, A. P. Paranjpe, and F. Cerio. "Ion Beam Patterning of High-Density STT-RAM Devices". In: *IEEE Transactions on Magnetics* 53.2 (2017), pp. 1–4. DOI: 10.1109/TMAG.2016.2603921.

[158] J. Chatterjee, T. Tahmasebi, J. Swerts, G. S. Kar, and J. De Boeck. "Impact of Seed Layer on Post-Annealing Behavior of Transport and Magnetic Properties of Co/Pt Multilayer-Based Bottom-Pinned Perpendicular Magnetic Tunnel Junctions". In: *Applied Physics Express* 8.6 (2015), p. 063002. DOI: 10.7567/APEX.8.063002.

[159] M. Quell, G. Diamantopoulos, A. Hössinger, and J. Weinbub. "Shared-Memory Block-Based Fast Marching Method for Hierarchical Meshes". In: *Journal of Computational and Applied Mathematics* 392 (2021), p. 113488. DOI: 10.1016/j.cam.2021.113488.

[160] M. Sussman, P. Smereka, and S. Osher. “A Level Set Approach for Computing Solutions to Incompressible Two-Phase Flow”. In: *Journal of Computational Physics* 114.1 (1994), pp. 146–159. DOI: 10.1006/jcph.1994.1155.

[161] L.-T. Cheng and Y.-H. Tsai. “Redistancing by Flow of Time Dependent Eikonal Equation”. In: *Journal of Computational Physics* 227.8 (2008), pp. 4002–4017. DOI: 10.1016/j.jcp.2007.12.018.

[162] G. Russo and P. Smereka. “A Remark on Computing Distance Functions”. In: *Journal of Computational Physics* 163.1 (2000), pp. 51–67. DOI: 10.1006/jcph.2000.6553.

[163] T. Wacławczyk. “A Consistent Solution of the Re-Initialization Equation in the Conservative Level-Set Method”. In: *Journal of Computational Physics* 299 (2015), pp. 487–525. DOI: 10.1016/j.jcp.2015.06.029.

[164] H. B. Curry. “The Method of Steepest Descent for Non-Linear Minimization Problems”. In: *Quarterly of Applied Mathematics* 2.3 (1944), pp. 258–261. DOI: 10.1090/qam/10667.

[165] M. Elsey and S. Esedoḡlu. “Fast and Accurate Redistancing by Directional Optimization”. In: *SIAM Journal on Scientific Computing* 36.1 (2014), A219–A231. DOI: 10.1137/120889447.

[166] M. W. Royston. “A Hopf-Lax Formulation of the Eikonal Equation for Parallel Redistancing and Oblique Projection”. PhD thesis. University of California, 2017.

[167] B. Lee, J. Darbon, S. Osher, and M. Kang. “Revisiting the Redistancing Problem Using the Hopf–Lax Formula”. In: *Journal of Computational Physics* 330 (2017), pp. 268–281. DOI: 10.1016/j.jcp.2016.11.005.

[168] M. Royston, A. Pradhana, B. Lee, Y. T. Chow, W. Yin, J. Teran, and S. Osher. “Parallel Redistancing Using the Hopf–Lax Formula”. In: *Journal of Computational Physics* 365 (2018), pp. 7–17. DOI: 10.1016/j.jcp.2018.01.035.

[169] J. A. Sethian and A. M. Popovici. “3-D Traveltime Computation Using the Fast Marching Method”. In: *Geophysics* 64.2 (1999), pp. 516–523. DOI: 10.1190/1.1444558.

[170] A. M. Popovici and J. A. Sethian. “3-D Imaging Using Higher Order Fast Marching Traveltimes”. In: *Geophysics* 67.2 (2002), pp. 604–609. DOI: 10.1190/1.1468621.

[171] J. Yang. “An Easily Implemented, Block-Based Fast Marching Method with Superior Sequential and Parallel Performance”. In: *SIAM Journal on Scientific Computing* 41.5 (2019), pp. C446–C478. DOI: 10.1137/18M1213464.

[172] J. N. Tsitsiklis. “Efficient Algorithms for Globally Optimal Trajectories”. In: *IEEE Transactions on Automatic Control* 40.9 (1995), pp. 1528–1538. DOI: 10.1109/9.412624.

[173] J. Gomez Gonzalez and S. Engineering. “Fast Marching Methods in Path and Motion Planning: Improvements and High-Level Applications”. PhD thesis. Universidad Carlos III Madrid, 2015.

[174] F. Mut, G. C. Buscaglia, and E. A. Dari. “New Mass-Conserving Algorithm for Level Set Redistancing on Unstructured Meshes”. In: *Journal of Applied Mechanics* 73.6 (2006), pp. 1011–1016. DOI: 10.1115/1.2198244.

[175] J. Qian, Y.-T. Zhang, and H.-K. Zhao. “Fast Sweeping Methods for Eikonal Equations on Triangular Meshes”. In: *SIAM Journal on Numerical Analysis* 45.1 (2007), pp. 83–107. DOI: 10.1137/050627083.

[176] Y. Wu, J. Man, and Z. Xie. “A Double Layer Method for Constructing Signed Distance Fields from Triangle Meshes”. In: *Graphical Models* 76.4 (2014), pp. 214–223. DOI: 10.1016/j.gmod.2014.04.011.

[177] V. Ramanuj and R. Sankaran. “High Order Anchoring and Reinitialization of Level Set Function for Simulating Interface Motion”. In: *Journal of Scientific Computing* (2019). DOI: 10.1007/s10915-019-01076-0.

[178] H. Zhao. “A Fast Sweeping Method for Eikonal Equations”. In: *Mathematics of Computation* 74.250 (2004), pp. 603–628. DOI: 10.1090/S0025-5718-04-01678-3.

[179] S. Bak, J. McLaughlin, and D. Renzi. “Some Improvements for the Fast Sweeping Method”. In: *SIAM Journal on Scientific Computing* 32.5 (2010), pp. 2853–2874. DOI: 10.1137/090749645.

[180] M. Detrixhe, F. Gibou, and C. Min. “A Parallel Fast Sweeping Method for the Eikonal Equation”. In: *Journal of Computational Physics* 237 (2013), pp. 46–55. DOI: 10.1016/j.jcp.2012.11.042.

[181] A. A. Nikitin, A. S. Serdyukov, and A. A. Duchkov. “Cache-Efficient Parallel Eikonal Solver for Multicore CPUs”. In: *Computational Geosciences* 22.3 (2018), pp. 775–787. DOI: 10.1007/s10596-018-9725-9.

[182] W.-K. Jeong and R. T. Whitaker. “A Fast Iterative Method for Eikonal Equations”. In: *SIAM Journal on Scientific Computing* 30.5 (2008), pp. 2512–2534. DOI: 10.1137/060670298.

[183] J. Weinbub and A. Hössinger. “Accelerated Redistancing for Level Set-Based Process Simulations with the Fast Iterative Method”. In: *Journal of Computational Electronics* 13.4 (2014), pp. 877–884. DOI: 10.1007/s10825-014-0604-x.

[184] 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 (MSSANZ)*. Perth: Modelling, Simulation Society of Australia, and New Zealand, Inc., 2011, pp. 641–647. DOI: 10.36334/modsim.2011.A9.gillberg.

[185] J. Weinbub, F. Dang, T. Gillberg, and S. Selberherr. “Shared-Memory Parallelization of the Semi-Ordered Fast Iterative Method”. In: *Proceedings of the Symposium on High Performance Computing (HPDC)*. Alexandria: ACM Press, 2015, pp. 217–224.

[186] J. Weinbub and A. Hössinger. “Comparison of the Parallel Fast Marching Method, the Fast Iterative Method, and the Parallel Semi-Ordered Fast Iterative Method”. In: *Procedia Computer Science* 80 (2016), pp. 2271–2275. DOI: 10.1016/j.procs.2016.05.408.

[187] S. Hong and W.-K. Jeong. “A Group-Ordered Fast Iterative Method for Eikonal Equations”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.2 (2017), pp. 318–331. DOI: 10.1109/TPDS.2016.2567397.

[188] M. A. Herrmann. “A Domain Decomposition Parallelization of the Fast Marching Method”. In: *Center for Turbulence Research* (2003), pp. 213–225.

[189] J. Weinbub and 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)*. San Diego: Society for Computer Simulation International, 2016, pp. 1–8. DOI: 10.22360/SpringSim.2016.HPC.052.

[190] G. Diamantopoulos, J. Weinbub, A. Hössinger, and 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)*. Trieste: IEEE, 2017, pp. 1–8. DOI: 10.1109/ICCSA.2017.7999648.

[191] E. Becker, W. Ehrfeld, P. Hagmann, A. Maner, and D. Münchmeyer. “Fabrication of Microstructures with High Aspect Ratios and Great Structural Heights by Synchrotron Radiation Lithography, Galvanoforming, and Plastic Moulding (LIGA Process)”. In: *Microelectronic Engineering* 4.1 (1986), pp. 35–56. DOI: 10.1016/0167-9317(86)90004-3.

[192] B. Radjenović, J. K. Lee, and M. Radmilović-Radjenović. “Sparse Field Level Set Method for Non-Convex Hamiltonians in 3D Plasma Etching Profile Simulations”. In: *Computer Physics Communications* 174.2 (2006), pp. 127–132. DOI: 10.1016/j.cpc.2005.09.010.

[193] P. Liu, D. Zhang, J. Guo, W. Wang, and F. Yang. “Optimization of Photoresist Development and DRIE Processes to Fabricate High Aspect Ratio Si Structure in 5 nm Scale”. In: *Journal of Micromechanics and Microengineering* 29.3 (2019), p. 035006. DOI: 10.1088/1361-6439/aaf940.

[194] A. Belyaev and P.-A. Fayolle. “An ADMM-Based Scheme for Distance Function Approximation”. In: *Numerical Algorithms* 84.3 (2020), pp. 983–996. DOI: 10.1007/s11075-019-00789-5.

[195] N. Cornea, D. Silver, and P. Min. “Curve-Skeleton Applications”. In: *Proceedings of IEEE Visualization (VIS)*. Minneapolis: IEEE, 2005, pp. 95–102. DOI: 10.1109/VISUAL.2005.1532783.

Own Publications

Journal Articles

[1] **M. Quell**, V. Suvorov, A. Hössinger, and J. Weinbub. "Parallel Velocity Extension for Level-Set-Based Material Flow on Hierarchical Meshes in Process TCAD". In: *IEEE Transactions on Electron Devices* 68.11 (2021), pp. 5430–5437. DOI: 10.1109/TED.2021.3087451.

[2] **M. Quell**, G. Diamantopoulos, A. Hössinger, and J. Weinbub. "Shared-Memory Block-Based Fast Marching Method for Hierarchical Meshes". In: *Journal of Computational and Applied Mathematics* 392 (2021), pp. 113488-1–113488-15. DOI: 10.1016/j.cam.2021.113488.

[3] W. Auzinger, H. Hofstätter, O. Koch, and **M. Quell**. "Adaptive Time Propagation for Time-Dependent Schrödinger Equations". In: *International Journal of Applied and Computational Mathematics* 7.1 (2021), pp. 6-1–6-14. DOI: 10.1007/s40819-020-00937-9.

[4] A. Toifl, **M. Quell**, X. Klemenschits, P. Manstetten, A. Hössinger, S. Selberherr, and J. Weinbub. "The Level-Set Method for Multi-Material Wet Etching and Non-Planar Selective Epitaxy". In: *IEEE Access* 8 (2020), pp. 115406–115422. DOI: 10.1109/ACCESS.2020.3004136.

[5] W. Auzinger, I. Brezinova, H. Hofstätter, O. Koch, and **M. Quell**. "Practical Splitting Methods for the Adaptive Integration of Nonlinear Evolution Equations. Part II: Comparison of Local Error Estimation and Step-Selection Strategies for Nonlinear Schrödinger and Wave Equations". In: *Computer Physics Communications* 234 (2019), pp. 55–71. DOI: 10.1016/j.cpc.2018.08.003.

[6] W. Auzinger, H. Hofstätter, O. Koch, **M. Quell**, and M. Thalhammer. "A Posteriori Error Estimation for Magnus-Type Integrators". In: *ESAIM: Mathematical Modelling and Numerical Analysis* 53.1 (2019), pp. 197–218. DOI: 10.1051/m2an/2018050.

[7] W. Auzinger, O. Koch, and **M. Quell**. "Adaptive High-Order Splitting Methods for Systems of Nonlinear Evolution Equations with Periodic Boundary Conditions". In: *Numerical Algorithms* 75.1 (2017), pp. 261–283. DOI: 10.1007/s11075-016-0206-8.

Book Contributions

[8] **M. Quell**, G. Diamantopoulos, A. Hössinger, S. Selberherr, and J. Weinbub. "Parallel Correction for Hierarchical Re-Distancing Using the Fast Marching Method". In: *Advances in High Performance Computing, Studies in Computational Intelligence*. Cham: Springer International Publishing, 2020, pp. 438–451. DOI: 10.1007/978-3-030-55347-0_37.

[9] **M. Quell**, P. Manstetten, A. Hössinger, S. Selberherr, and J. Weinbub. "Parallelized Construction of Extension Velocities for the Level-Set Method". In: *Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science*. Cham: Springer International Publishing, 2020, pp. 348–358. DOI: 10.1007/978-3-030-43229-4_30.

[10] W. Auzinger, H. Hofstätter, O. Koch, **M. Quell**, and M. Thalhammer. "A Posteriori Error Estimation for Magnus-Type Integrators". In: *ASC Report 1/2018*. Wien: Vienna University of Technology, 2018, pp. 1–19.

[11] W. Auzinger, I. Brezinova, H. Hofstätter, O. Koch, and **M. Quell**. "Practical Splitting Methods for the Adaptive Integration of Nonlinear Evolution Equations. Part II: Comparison of Local Error Estimation and Step-Selection Strategies for Nonlinear Schrödinger and Wave Equations". In: *ASC Report 14/2017*. Wien: Vienna University of Technology, 2017, pp. 1–40.

[12] W. Auzinger, O. Koch, and **M. Quell**. "Adaptive High-Order Splitting Methods for Systems of Nonlinear Evolution Equations with Periodic Boundary Conditions". In: *ASC Report 41/2015*. Wien: Vienna University of Technology, 2015, pp. 1–29.

[13] W. Auzinger, O. Koch, and **M. Quell**. "Splittingverfahren für die Gray-Scott-Gleichung". In: *ASC Report 07/2015*. Wien: Vienna University of Technology, 2015, pp. 1–11.

Conference Contributions

[14] C. Lenz, A. Scharinger, **M. Quell**, P. Manstetten, A. Hössinger, and J. Weinbub. "Evaluating Parallel Feature Detection Methods for Implicit Surfaces". In: *Proceedings of the Austrian-Slovenian HPC Meeting (ASHPC)*. Maribor: University of Ljubljana, 2021, p. 31. DOI: 10.3359/2021hpc.

[15] **M. Quell**, G. Diamantopoulos, A. Hössinger, and J. Weinbub. "Shared-Memory Block-Based Fast Marching Method for Hierarchical Meshes". In: *Proceedings of the European Seminar on Computing (ESCO)*. Pilsen: University of West Bohemia, 2020.

[16] **M. Quell**, G. Diamantopoulos, A. Hössinger, S. Selberherr, and J. Weinbub. "Parallelized Bottom-Up Correction in Hierarchical Re-Distancing for Topography Simulation". In: *Procedings of the High Performance Computing Conference (HPC)*. Borovets: Bulgarian Academy of Sciences, 2019, p. 45.

[17] **M. Quell**, P. Manstetten, A. Hössinger, S. Selberherr, and J. Weinbub. "Parallelized Construction of Extension Velocities for the Level-Set Method". In: *Proceedings of the International Conference on Parallel Processing and Applied Mathematics (PPAM)*. Bialystok: Czestochowa University of Technology, 2019, p. 42.

[18] **M. Quell**, A. Toifl, A. Hössinger, S. Selberherr, and J. Weinbub. "Parallelized Level-Set Velocity Extension Algorithm for Nanopatterning Applications". In: *Proceedings of the International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*. Udine: IEEE, 2019, pp. 335–338. DOI: 10.1109/SISPAD.2019.8870482.

[19] A. Toifl, **M. Quell**, A. Hössinger, A. Babayan, S. Selberherr, and J. Weinbub. "Novel Numerical Dissipation Scheme for Level-Set Based Anisotropic Etching Simulations". In: *Proceedings of the International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*. Udine: IEEE, 2019, pp. 327–330. DOI: 10.1109/SISPAD.2019.8870443.

[20] G. Diamantopoulos, P. Manstetten, L. Gnam, V. Simonka, L. F. Aguinsky, **M. Quell**, A. Toifl, A. Hössinger, and J. Weinbub. "Recent Advances in High Performance Process TCAD". In: *Proceedings of the SIAM Conference on Computational Science and Engineering (CSE)*. Spokane: Society for Industrial and Applied Mathematics, 2019, p. 335.

[21] L. Gnam, P. Manstetten, **M. Quell**, K. Rupp, S. Selberherr, and J. Weinbub. "A Flexible Shared-Memory Parallel Mesh Adaptation Framework". In: *Proceedings of the International Conference on Computational Science and Its Applications (ICCSA)*. Saint Petersburg: IEEE, 2019, pp. 158–165. DOI: 10.1109/ICCSA.2019.00016.

[22] A. Hössinger, P. Manstetten, G. Diamantopoulos, **M. Quell**, and J. Weinbub. "High Performance Computing Aspects in Semiconductor Process Simulation". In: *Proceedings of the Workshop on High Performance TCAD (WHPTCAD)*. Chicago: Institute for Microelectronics, TU Wien, 2019, pp. 3–4.

[23] P. Manstetten, G. Diamantopoulos, L. Gnam, L. F. Aguinsky, **M. Quell**, A. Toifl, A. Scharinger, A. Hössinger, M. Ballicchia, M. Nedjalkov, and J. Weinbub. "High Performance TCAD: From Simulating Fabrication Processes to Wigner Quantum Transport". In: *Proceedings of the Workshop on High Performance TCAD (WHPTCAD)*. Chicago: Institute for Microelectronics, TU Wien, 2019, p. 13.

Curriculum Vitae

Personal Information

Name	Michael Julian Augustus Quell
Nationality	Austrian
Place of Birth	Vienna, Austria

Education

06/2018	-	present	Doctoral Program, *Electrical Engineering*, *Institute for Microelectronics*, *Technische Universität (TU) Wien*
04/2016	-	04/2018	Graduate Studies (MSc), *Technical Mathematics*, *TU Wien*, *Faculty of Mathematics and Geoinformation*
10/2012	-	04/2016	Graduate Studies (BSc), *Technical Mathematics*, *TU Wien*, *Faculty of Mathematics and Geoinformation*
09/2011	-	09/2012	Active Reserve Officer Training, *Signaling*, *Fernmeldetruppenschule, Wien*
09/2003	-	06/2011	Matura, *Majors: Mathematics, German, Applied Computer Aided Geometry, English(FL)* *BRG 8 Albertgasse, Wien*

Research Positions

06/2019	-	present	University Assistant, *Institute for Microelectronics*, *TU Wien*
06/2018	-	present	Project Assistant, *Christian Doppler Laboratory for High Performance TCAD*, *Institute for Microelectronics*, *TU Wien*
10/2014	-	10/2016	Study Assistant, *Institute for Analysis and Scientific Computing*, *TU Wien*