Parallelized Construction of Extension Velocities for the Level-Set Method

Michael Quell[1(✉)], Paul Manstetten[2], Andreas Hössinger[3], Siegfried Selberherr[2], and Josef Weinbub[1]

[1] Christian Doppler Laboratory for High Performance TCAD, Institute for Microelectronics, TU Wien, Vienna, Austria
{quell,weinbub}@iue.tuwien.ac.at
[2] Institute for Microelectronics, TU Wien, Vienna, Austria
{manstetten,selberherr}@iue.tuwien.ac.at
[3] Silvaco Europe Ltd., St Ives, UK
andreas.hoessinger@silvaco.com

Abstract. The level-set method is widely used to track the motion of interfaces driven by a velocity field. In many applications, the underlying physical model defines the velocity field only at the interface itself. For these applications, an extension of the velocity field to the simulation domain is required. This extension has to be performed in each time step of a simulation to account for the time-dependent velocity values at the interface. Therefore, the velocity extension is critical to the overall computational performance. We introduce an accelerated and parallelized approach to overcome the computational bottlenecks of the prevailing and serial-in-nature fast marching method, in which the level-set function is used to predetermine the computational order for the velocity extension. This allows to employ alternative data structures, which results in a straightforward parallelizable approach with reduced complexity for insertion and removal as well as improved cache efficiency. Compared to the prevailing fast marching method, our approach delivers a serial speedup of at least 1.6 and a shared-memory parallel efficiency of 66% for 8 threads and 37% for 16 threads.

Keywords: Velocity extension · Level-set method · Parallel computing · Fast marching method

1 Introduction

The level-set method [9] is widely used to track moving interfaces in different fields of science, such as in computer graphics [8], fluid dynamics [7], and microelectronics [14]. The level-set method represents an interface Γ implicitly as the zero-level-set of a higher-dimensional function, i.e., the level-set function $\phi(\boldsymbol{x}, t)$. The motion of the interface is given by the level-set equation

$$\frac{\partial \phi}{\partial t} = F_{ext} |\nabla \phi|, \tag{1}$$

R. Wyrzykowski et al. (Eds.): PPAM 2019, LNCS 12043, pp. 348–358, 2020.
https://doi.org/10.1007/978-3-030-43229-4_30

where $F_{ext}(\boldsymbol{x}, t)$ is the extended velocity field from the underlying model. The extended velocity field F_{ext} is not unique [1], as the only formal mathematical requirement is

$$\lim_{\boldsymbol{x} \to \boldsymbol{x_0}} F_{ext}(\boldsymbol{x}) = F(\boldsymbol{x_0}), \tag{2}$$

with $F(\boldsymbol{x}, t)$ being the given continuous velocity on the interface and $\boldsymbol{x_0}$ any point on the interface, i.e., $\phi(\boldsymbol{x_0}) = 0$. The extension should not introduce new artificial zero values. An extended velocity field F_{ext} fulfilling

$$\nabla F_{ext} \cdot \nabla \phi = 0 \text{ and } F_{ext}\big|_{\phi^{-1}(\mathbf{0})} = F \tag{3}$$

meets the requirement (2), does not introduce artificial zero values, and preserves the signed-distance property of the level-set function during the advection step, which is desirable as it leads to maximal numerical stability of the method [2]. The velocity has to be extended every time a new interface velocity is calculated (i.e., for a finite difference discretization in every time step).

A widely used approach to solve (3) is the fast marching method (FMM) [1]. Other methods such as the fast iterative method [6] and the fast sweeping method [11] are not considered here, because of their iterative nature, the computational costs to obtain an accuracy level comparable to FMM are too high. The FMM was originally developed to efficiently solve the Eikonal equation

$$\|\nabla \phi\| = 1 \quad \text{and} \quad \phi\big|_{\Gamma} = g(\boldsymbol{x}). \tag{4}$$

Within the level-set method, (4) and $g(x) = 0$ is solved to give the initial level-set function the signed-distance property [13]. Due to the utilization of a global heap (a single priority queue for the full domain) to track the order of the computations, the solution of (3) using the FMM has complexity $\mathcal{O}(n \log n)$, where n is the number of grid points (discrete points on the computational grid) and is inherently serial. Another approach achieves complexity $\mathcal{O}(n)$ by quantization of the keys of the heap at the cost of a different error bound [17].

There have been successful attempts to parallelize the FMM through domain decomposition for distributed-memory systems [16] and for shared-memory systems [15]. Therein FMM is executed on each sub-domain with its own heap, thus enabling parallelism. An explicit synchronized data exchange via a ghost layer is used to resolve inter-domain dependencies. The decomposition approach requires knowledge about the interface position to balance the load equally [4], on the other hand the proposed algorithm employs dynamic load balancing independent of the interface position, therefore, a fair comparison is not possible. In [10], a serial approach based on fast scanning is presented, but no information is given on how the computations are ordered, which is, however, essential for cache efficiency.

In Sect. 2, we provide the original FMM algorithm (Algorithm 1) for reference and details of our approach for an accelerated velocity extension algorithm, avoiding the aforementioned difficulties when utilizing the FMM. In Sect. 3, a new serial algorithm (Algorithm 2) and a parallel algorithm (Algorithm 3) are presented. In Sect. 4, the serial and parallel run-times of our approach are presented for an application example from the field of microelectronics.

2 Theory

The FMM and the two proposed algorithms assign each grid point an exclusive state: *Known* means the grid point has the final velocity assigned and no further updates are required, *Unknown* means the grid point does not yet have a velocity assigned, and *Band* means that the grid point has a velocity assigned, but it is not final. The FMM, orders the grid points in the *Band* by the distance to the interface, to determine which is processed next. The ordering is achieved using a minimum heap data structure, i.e., the top element (grid point) is always the closest to the interface. This ensures that a grid point's value is *final* (i.e., it conforms to (3)), when it is removed from the *Band*. A standard implementation for the FMM is given by Algorithm 1 [12].

The *compute()* sub-routine is used to update the value of a grid point, e.g., solving the Eikonal equation, or to compute the velocity, which is described in detail in [1]. The interface and its adjacent grid points (for which the velocity is known) divide the domain in two zones (*inside* and *outside*) and the algorithm has to be applied for each zone separately as both zones are independent of each other, (see Fig. 1).

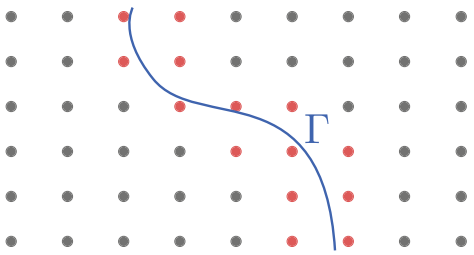


Fig. 1. The interface Γ (zero-level-set) is given in blue. For the red grid points next to the interface, the velocity values are calculated. From those points the velocity is extended to the remaining domain (black grid points). (Color figure online)

The run-time contributions of the computational sub-tasks of the velocity extension are show in Fig. 2, if the FMM is used to extend the velocity. Most of

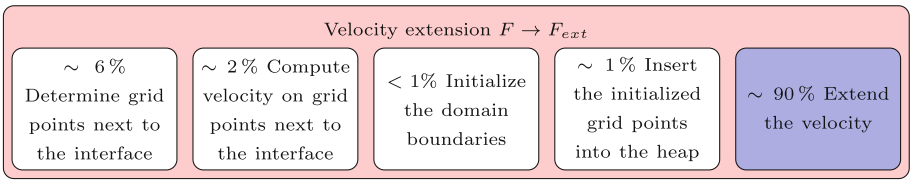


Fig. 2. Computational sub-tasks and their run-time contribution for the construction of the extended velocity field F_{ext}, utilizing the FMM. (Color figure online)

Algorithm 1: FMM

```
1  Known ← ∅
2  Band ← initialized grid points
3  Unknown ← all other grid points
4  while Band not empty do
5      q = first element of Band
6      add q to Known
7      forall p neighbors of q do
8          if p ∈ Unknown then
9              compute(p)
10             add p to Band
```

Algorithm 2: ExtendVelocity

```
1  Known ← ∅
2  Band ← initialized grid points
3  Unknown ← all other grid points
4  while Band not empty do
5      q = first element of Band
6      add q to Known
7      forall p neighbors of q do
8          if p ∈ Unknown and upwind
             neighbors ∉ Unknown then
9              compute(p)
10             add p to Band
```

the time is spent extending the velocity (last sub-task marked in blue), whilst the other steps require considerably less time.

For the heap data structure, the insertion or removal of a grid point triggers a sorting which results in $\mathcal{O}(\log n)$ operations. The overall complexity to populate the heap with n grid points is therefore $\mathcal{O}(n \log n)$. Formulating the problem in the context of graph theory, $\mathcal{O}(n)$ is achieved.

Assume an ordered graph $G(N, E)$, the nodes N are given by all the grid points and the edges E are given by the direct upwind neighbors of a grid point (i.e., neighboring grid points with a smaller distance to the interface). The order in which the nodes can be computed is a topological sort problem, which is solved in linear $\mathcal{O}(|N| + |E|)$ time [5]. This is also linear in $n = |N|$, which is the number of grid points in the domain, as the number of edges $|E|$ is limited by $6|N|$ in case of a 7-point three-dimensional stencil to compute the gradient.

The topological sort problem is solved by a depth-first or breadth-first traversal over the graph [3]. These traversals can be realized by adapting the ordering of the grid points which have the status *Band* in Algorithm 1. Using a queue or a stack as data structure corresponds to a breadth-first and depth-first traversal, respectively. Adding an element to the *Band* can be done for both data structures in $\mathcal{O}(1)$; this is an advantage compared to the heap. The parallelization of these algorithms is straight forward, by processing all nodes which do not have an unresolved dependence in parallel.

3 Parallel Velocity Extension

Based on the findings in the previous section, we investigate – as a first step – an adapted serial algorithm (Algorithm 2) which uses different data structures to implement the *Band*. This requires a check in the *neighbors loop*, whether the upwind neighbors are not in the *Unknown* state (Line 8), only then the velocity is computed. In Algorithm 1, this check is not necessary, as the heap guarantees no unknown dependencies of the top element.

Algorithm 3 is the parallel version of Algorithm 2. The parallelization is realized by treating every grid point next to the interface as an independent starting grid point for the traversal through the graph. The grid points in the *Band* (cf. Algorithm 3 Line 6), are exclusive (i.e., OpenMP private) to each thread, but the status of a grid point is shared with all threads. In principal, explicit synchronization would be necessary in the *neighbors loop* which calls the *compute()* sub-routine – not for correctness of the algorithm, but to ensure that no grid point will be treated by two different threads (avoiding *redundant* computations). However, our approach (Algorithm 3) deliberately recomputes the values by different threads as the computational overhead is negligible compared to explicit synchronization costs. Conflicting access of two threads to the data of a grid point is resolved by enforcing atomic read and write operations (cf. Sect. 4).

In order to keep the number of *redundant* computations small, the threads start on grid points evenly distributed over the full set of interface grid points. If the *Band* of a thread is empty, the thread is dynamically assigned a new starting grid point from the initialized grid points. In case two threads operate at the same location, the check for the state *Unknown* (Line 11) reduces the *redundant* computations. To further reduce the redundant computations, it is checked again if a different thread has already processed the grid point before adding the grid point to its exclusive *Band* (Line 13). In case of a serial execution the second check is redundant to the first one (Line 11). As we present in Sect. 4, the ratio of *redundant* computations to *necessary* computations is below 1%, which plainly favors redundancy over the explicit synchronization which would limit parallel scalability.

Algorithm 3: ExtendVelocityParallel

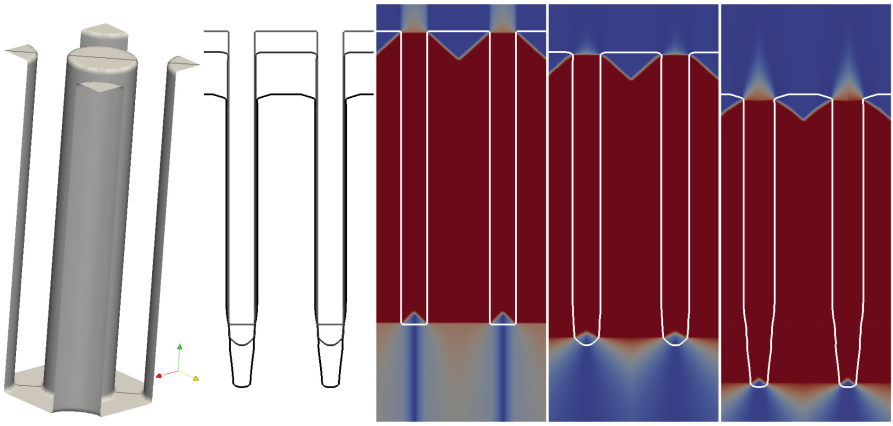
```
1  Known ← ∅
2  InitPoints ← initialized grid points
3  Unknown ← all other grid points
4  forall b in InitPoints in parallel do
5      Band ← ∅
6      Band add b
7      while Band not empty do
8          q = first element of Band
9          add q to Known
10         forall p neighbors of q do
11             if p ∈ Unknown and upwind neighbors ∉ Unknown then
12                 compute(p)
13                 if p ∈ Unknown then
14                     add p to Band
```

4 Computational Results

We evaluate the performance by benchmarking our velocity extension approach embedded in a simulation of a microelectronic fabrication process, specifically an etching simulation of a pillar-like structure (cf. Fig. 3a)[1]. This geometry provides a challenging and representative testbed as it includes flat, convex, and concave interface areas, which lead to shocks and rarefaction fans in the extended velocity field. The domain is discretized using a dense equidistant grid and the gradients are computed using first order finite differences. Symmetric boundaries are enforced by a ghost layer outside the domain.

Table 1. Properties of the discretization for different resolutions for the example geometry (cf. Fig. 3a).

	Resolution	# grid points	# initialized grid 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



(a) Isometric view (b) Interface (c) $t = 0$ (d) $t = 0.5$ (e) $t = 1.0$

Fig. 3. (a) Initial interface in isometric view. (b)–(e) Cross-section of the simulation domain through a plane with normal (1, 1, 0): (b) interface overlaid at different simulation times, (c)-(e) extended velocity (low velocity in red and high in blue) for times $t = 0$, $t = 0.5$, $t = 1.0$. (Color figure online)

[1] However, the presented algorithm and implementation details are not tailored or restricted to the field of microelectronics and can be applied to other fields as well.

A *low resolution case* and a *high resolution case* have been investigated. Table 1 summarizes the properties of the resulting discretization. In Fig. 3 a slice of the extended velocity is shown for three different times of the simulation.

The benchmark results are obtained on a single compute node of the Vienna Scientific Cluster[2] equipped with two Intel Xeon E5-26504 8 core processors and 64 GB of main memory. The algorithms are implemented in a GNU/Linux environment in C++11 (GCC-7.3 with optimization flag `-O3`). OpenMP is used for parallelization. The read and write operations from and to the state (*Unknown*, *Band*, and *Known*) of a grid point and the velocity of a grid point are implemented using the atomic directives of OpenMP. C++'s standard template library (STL) containers are used for the stack and the queue. The heap is based on the STL priority queue using the distance of the grid points to the interface as key. All following results report the run-time averaged over 10 executions for all sub-tasks depicted in Fig. 2. Both of the proposed algorithms calculate the exact same results as the reference FMM, also independent to the number of threads.

4.1 Serial Results

Table 2 compares the serial run-times of all three algorithms for both spatial resolutions (*Run-time*). The ratio of how often an upwind neighbor is in the state *Unknown* to the total number of updates (*Un.* up.) is used as metric for optimal traversal. An optimal traversal would have a rate of 0, though this metric neglects effects of different access times (and cache misses). This causes uncorrelated run-times to the ratio of *Unknown* upwind neighbors, because the heap and queue have similar ratios but drastically different run-times for Algorithm 2. The run-times of Algorithm 1 (i.e., FMM, using a heap) are at least 1.3 times slower compared to Algorithm 2, or Algorithm 3, when using a stack or a queue. The shortest run-times are obtained using Algorithm 2 combined with a queue data structure leading to a speedup of 1.6 and 2.0 for the *low* and the *high resolution case*, respectively. The stack has the highest rate of skipped velocity updates due to an *Unknown* upwind dependence (*Un.* up.), because the distance to the interface is not used to select the subsequently processed grid point.

Switching the algorithms Algorithms 2 and 3 for the stack solely reverts the initial order of the grid points in the *Band*. The heap profits from switching to Algorithm 3, as this reduces the size of the heap, which decreases the insert and removal time. The queue has an increased run-time with Algorithm 3, because the access pattern yields 4 times higher rates of *Unknown* upwind neighbors. In conclusion, for Algorithm 3 the data structure for the *Band* is less important, because the size of the *Band* (cf. Algorithm 3, Line 6) is small (starts with a single grid point) compared to the size of the *Band* in Algorithm 2 (starts with about half the number of the initialized grid points (cf. Table 1))[3].

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

[3] The other half of the initialized grid points resides in the second zone of the domain, which is processed independently.

4.2 Parallel Results

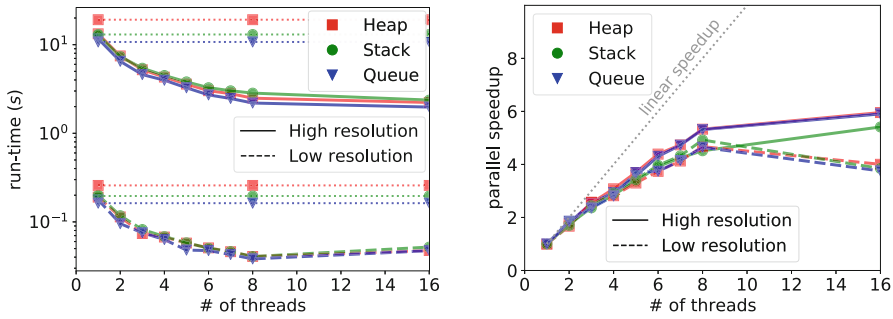
Figure 4a shows the run-time of Algorithm 3 and the achieved parallel speedup for up to 16 threads. The usage of hyper-threads has been investigated, but no further speedup was measured. Therefore, the analysis focuses on the available 16 physical cores on the compute node. Each thread is pinned to its own core, e.g Thread 0 on Core 0, Thread 1 on Core 1, and so forth.

Table 2. Serial run-time and the ratio how often *Unknown* upwind neighbors (*Un.* up.) were encountered compared to the total updates. Bold numbers indicate the fastest run-time for each resolution.

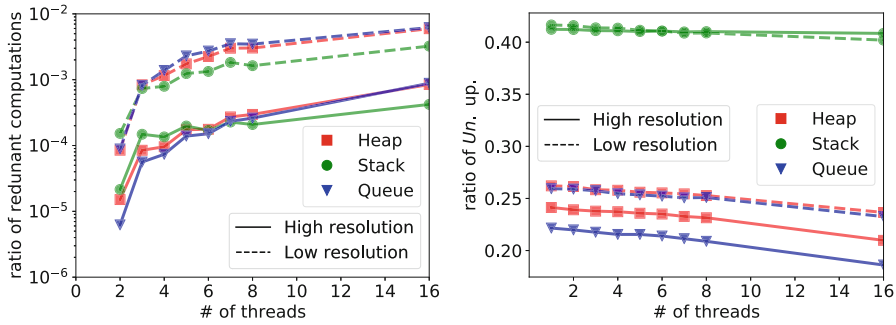
Data structure	Algorithm 1		Algorithm 2		Algorithm 3	
	Run-time	*Un.* up	Run-time	*Un.* up	Run-time	*Un.* up.
(a) *Low resolution case*						
Heap	0.265	0.0	0.258	0.034	0.190	0.262
Stack			0.196	0.418	0.200	0.416
Queue			**0.162**	0.077	0.177	0.259
(b) *High resolution case*						
Heap	19.99	0.0	19.14	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 serial results have already shown that for Algorithm 3 the data structure of the *Band* is less important (cf. Algorithm 3 in Table 2). The queue is also the fastest for the parallel case, because the ordering by first-in first-out avoids the sorting of the heap and reduces encountering of *Unknown* upwind neighbors compared to the stack. The shortest run-times are obtained for 8 and 16 threads for the *low resolution case* and *high resolution case*, respectively. The algorithm with the heap produces the best parallel speedup (not lowest run-time), because an increasing number of threads further reduces the size of the data structure of the *Band*. Small *Band* sizes are important for the heap, because the insertion of grid points scales with the number of grid points in the *Band* (stack and queue do not have this drawback).

The parallel efficiency for the *low resolution case* using 8 threads is 58% for the heap and the queue and 61% for the stack. For 8 threads, the *high resolution case* has a parallel efficiency of 56% for the stack, 66% for the queue, and 67% for the heap. For more than one thread, the parallel Algorithm 3 has a shorter run-time than the serial Algorithm 2. Above 8 threads, the utilization of cores on both processors induces non-uniform memory access leading to an increased run-time for the *low resolution case* (parallel efficiency of 25%) and only marginal speedup for the *high resolution case* (parallel efficiency of 37%) for all data structures. The memory is allocated by Thread 0 which resides on



(a) Parallel run-time (left) and speedup (right) for Alg. 3. For comparison, the serial run-time of Alg. 2 is shown by the dotted lines.



(b) On the left, the ratio of *redundant* to total calls to the *compute()* sub-routine due to implicit synchronization is shown. On the right, the ratio of how often *Unknown* upwind neighbors were encountered to the total amount of testing for *Unknown* upwind neigbors is shown.

Fig. 4. Parallel run-time results for Algorithm 3.

the socket of the first processor, therefore only the first 8 threads can directly access the memory. Also threads running on different sockets do not share the L3-cache, which forces communication via the main memory.

In Fig. 4b, the ratio of *redundant* computations and not performed computations due to an *Unknown* upwind neighbors are shown (cf. Sect. 3). The increase of the *redundant* computations ratio saturates with the number of threads. For 16 threads, less than 1% of the *compute()* calls are wasted (i.e., *redundant*) in the *low resolution case* and less than 0.1% in the *high resolution case*. The ratio of the *redundant* computations in the *high resolution case* is lower, because the threads process more grid points in relation to the grid points, at which threads can interfere. A similar situation is found for the ratio between the volume and the surface of a sphere. As already hinted in Sect. 3, enforcing explicit synchronization in the *neighbors loop* would lead to a significant decrease of parallel efficiency, because the explicit synchronization required to ensure that every grid point is only computed once has a higher computational cost compared to

the *redundant* computations introduced otherwise. The number of redundant computations is only related to the synchronization paradigm (independent to the shared-memory approach). The ratio how often the *compute()* sub-routine is skipped, because upwind neighbors were in the *Unknown* state (cf. Sect. 3), slightly decreases with the number of threads, as the possibility increases that another thread has computed an *Unknown* upwind neighbor just in time.

In comparison with the original FMM (cf. Algorithm 1), Algorithm 3 using the queue achieved a minimal run-time of 0.038 s which is due to a serial speedup of 1.5 and a parallel speedup of 5.6 for 8 threads in the *low resolution case*. In the *high resolution case* the minimal run-time of 1.975 s is also achieved with the queue data structure when utilizing all 16 threads (Serial speed up of 1.7 and parallel speedup of 10.1).

5 Conclusion

A new parallel approach to accelerate the velocity extension in the level-set method has been presented and compared to the prevailing FMM. The asymptotic complexity is $\mathcal{O}(n)$ by utilizing the level-set function to determine the order of computations. Furthermore, this approach opens an attractive path for parallelization. The serial speedup compared to the FMM is at least 1.6; a speedup of 2 is observed for a high resolution test case. The proposed parallel algorithm is tailored towards a shared-memory platform. The parallel efficiency is 58% for 8 threads; 66% are achieved for a high resolution test case. Overall, we provide a straight forward parallelizable algorithm (sparing any explicit synchronization) for velocity extension in the level-set method constituting an attractive drop-in replacement for the prevailing FMM.

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

References

1. Adalsteinsson, D., Sethian, J.A.: The fast construction of extension velocities in level set methods. J. Comput. Phys. **148**(1), 2–22 (1999). https://doi.org/10.1006/jcph.1998.6090
2. Cheng, L.T., Tsai, Y.H.: Redistancing by flow of time dependent eikonal equation. J. Comput. Phys. **227**(8), 4002–4017 (2008). https://doi.org/10.1016/j.jcp.2007.12.018
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2009). ISBN: 9780262033848
4. Diamantopoulos, G., Weinbub, J., Selberherr, S., Hössinger, A.: Evaluation of the shared-memory parallel fast marching method for re-distancing problems. In: Proceedings of the 17th International Conference on Computational Science and Its Applications, pp. 1–8 (2017). https://doi.org/10.1109/ICCSA.2017.7999648

5. Hagerup, T., Maas, M.: Generalized topological sorting in linear time. Fundam. Comput. Theory **710**, 279–288 (1993). https://doi.org/10.1007/3-540-57163-9_23
6. Jeong, W.K., Whitaker, R.T.: A fast iterative method for Eikonal equations. SIAM J. Sci. Comput. **30**(5), 2512–2534 (2008). https://doi.org/10.1137/060670298
7. Losasso, F., Gibou, F., Fedkiw, R.: Simulating water and smoke with an octree data structure. ACM Trans. Graph. **23**(3), 457–462 (2004). https://doi.org/10.1145/1015706.1015745
8. Museth, K.: VDB: high-resolution sparse volumes with dynamic topology. ACM Trans. Graph. **32**(3), 1–22 (2013). https://doi.org/10.1145/2487228.2487235
9. Osher, S., Sethian, J.A.: Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. J. Comput. Phys. **79**(1), 12–49 (1988). https://doi.org/10.1016/0021-9991(88)90002-2
10. Ouyang, G.F., Kuang, Y.C., Zhang, X.M.: A fast scanning algorithm for extension velocities in level set methods. Adv. Mater. Res. **328**(1), 677–680 (2011). https://doi.org/10.4028/www.scientific.net/AMR.328-330.677
11. Qian, J., Zhang, Y., Zhao, H.: Fast sweeping methods for Eikonal equations on triangular meshes. SIAM J. Numer. Anal. **45**(1), 83–107 (2007). https://doi.org/10.1137/050627083
12. Sethian, J.A.: A fast marching level set method for monotonically advancing fronts. Proc. Natl. Acad. Sci. **93**, 1591–1595 (1996). https://doi.org/10.1073/pnas.93.4.1591
13. Sethian, J.A., Vladimirsky, A.: Fast methods for the Eikonal and related Hamilton-Jacobi equations on unstructured meshes. Proc. Natl. Acad. Sci. **97**(11), 5699–5703 (2000). https://doi.org/10.1073/pnas.090060097
14. Suvorov, V., Hössinger, A., Djurić, Z., Ljepojevic, N.: A novel approach to three-dimensional semiconductor process simulation: application to thermal oxidation. J. Comput. Electron. **5**(4), 291–295 (2006). https://doi.org/10.1007/s10825-006-0003-z
15. Weinbub, J., Hössinger, A.: Shared-memory parallelization of the fast marching method using an overlapping domain-decomposition approach. In: Proceedings of the 24th High Performance Computing Symposium, pp. 1–8. Society for Modeling and Simulation International (2016). https://doi.org/10.22360/SpringSim.2016.HPC.052
16. Yang, J., Stern, F.: A highly scalable massively parallel fast marching method for the Eikonal equation. J. Comput. Phys. **332**, 333–362 (2017). https://doi.org/10.1016/j.jcp.2016.12.012
17. Yatziv, L., Bartesaghi, A., Sapiro, G.: O(N) implementation of the fast marching algorithm. J. Comput. Phys. **212**(2), 393–399 (2006). https://doi.org/10.1016/j.jcp.2005.08.005