J Sci Comput (2011) 46: 243–264
DOI 10.1007/s10915-010-9399-5

A Memory and Computation Efficient Sparse Level-Set Method

Wladimir J. van der Laan · Andrei C. Jalba · Jos B.T.M. Roerdink

Received: 8 September 2009 / Revised: 24 June 2010 / Accepted: 24 June 2010 /
Published online: 11 July 2010

Abstract Since its introduction, the level set method has become the favorite technique for capturing and tracking moving interfaces, and found applications in a wide variety of scientific fields. In this paper we present efficient data structures and algorithms for tracking dynamic interfaces through the level set method. Several approaches which address both computational and memory requirements have been very recently introduced. We show that our method is up to 8.5 times faster than these recent approaches. More importantly, our algorithm can greatly benefit from both fine- and coarse-grain parallelization by leveraging SIMD and/or multi-core parallel architectures.

Keywords Level sets · Sparse-grid method · Tile management

1 Introduction

Since its introduction by Osher and Sethian [17], the level set method has become the method of choice for capturing and tracking moving interfaces. It has found applications in a wide variety of scientific fields, ranging from chemistry and physics to computer vision and graphics. For example, in computer vision, most state-of-the-art segmentation techniques are based on level sets to steer the evolving contour or surface towards the objects of interest [24].

The main idea of the level set method is to represent the dynamic interface (e.g., contour, surface, etc.) implicitly and embed it as the zero level set of a time-dependent, higher-dimensional function. Then, evolving the interface with a given velocity in the normal direction becomes equivalent to solving a time-dependent partial differential equation (PDE)

W.J. van der Laan · J.B.T.M. Roerdink (✉)
Johann Bernoulli Institute for Mathematics and Computer Science, University of Groningen,
P.O. Box 407, 9700 AK Groningen, The Netherlands
e-mail: j.b.t.m.roerdink@rug.nl

A.C. Jalba
Department of Mathematics and Computer Science, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

for the embedding level set function. The main advantage of the level set method is that it allows the interface to undergo arbitrary topological changes, which is much more difficult to handle using explicit representations. The cost which has to be paid for the flexibility offered by the level set method is twofold. First, *computationally*, one has to solve the time-dependent level-set PDE in a higher-dimensional space than that of the embedded interface, and secondly, the *memory requirements* are higher than the size of the interface, as one needs to explicitly store a uniform Cartesian grid for solving the level set PDE. To address the computational issue, a number of techniques have been proposed, such as the so-called narrow-band schemes, see Sect. 2. Such methods rely on the fact that it suffices to solve the PDE only in the vicinity of the interface in order to preserve the embedding. Thus, the computational requirements scale with the size of the interface. However, most narrow-band methods require the (uniform) computational grid to be explicitly stored. As shown by Nielsen and Museth [13], hierarchical structures (e.g., based on octrees) only need to store points of the finest grid at the interface, while courser grids are used in the remaining part of the narrow band [4, 11, 12, 23]. However, these methods still require large amounts of memory, use complicated data structures, and additionally, increase computational requirements compared to narrow-band methods, as access to grid points is relatively slow.

In this paper we present efficient data structures and algorithms for tracking dynamic interfaces through the level set method.

Our method, which we call Sorted Tile List, uses *tiles*, i.e., small, fixed-size blocks, to represent the narrow band around the interface. Instead of using a coarse grid of pointers to tiles that intersect the interface, our method implicitly locates the neighbours of each tile by maintaining an active list of tiles lexicographically sorted by coordinates. While several methods which address both computational and memory requirements have been very recently introduced [2, 8, 13], we show that our *sequential algorithm* is faster than these recent approaches and more importantly, that our algorithm can greatly benefit from both fine- and coarse-grain parallelization by leveraging SIMD and/or multi-core configurations.

The main contributions of this paper are:

- A highly efficient, tile-based, sparse-grid method for the level-set representation, which works in any number of dimensions.
- A fast, parallelizable and memory-efficient data structure that can be used to manage sparse grids of unbounded size.
- Fast, scalable algorithms for iterating and maintaining the proposed data structure, enabling efficient simulations based on level sets. More specifically, the temporal and spatial complexities of our data structure with the associated sequential algorithms are as follows:
  - *Storage requirements* are proportional to the number N of tiles intersecting the interface, it follows that the memory footprint of our method is *optimal* and scales with $O(N^{d-1})$, where $d > 1$ is the number of spatial dimensions.
  - *Sequential access* to grid points has time complexity $O(N)$.
  - *Access to neighbouring grid points* within the computational finite-difference stencil has time complexity $O(1)$.
  - *Random access* to grid points has complexity $O(\log N)$.
  - *Maintaining* the sparse data structure is linear, i.e., $O(N)$.

2 Previous and Related Work

As the level-set method has proven to be a very useful tool in any application requiring the tracking of moving interfaces, there has been continuous interest in developing efficient algorithms to address the large computational and memory requirements involved, see Sect. 1.

The computational issue was first addressed with the introduction of the so-called "narrow-band schemes" [1, 3]. The basic idea is to restrict the computations to a small vicinity around the zero level set used to represent the deforming interface. Whitaker [25] further improved the efficiency of this scheme, by performing calculations only at grid locations corresponding to the interface, resulting in a stencil only as wide as necessary for the finite-difference calculations at these locations. Peng *et al.* [18] also solved the level set PDE only in a narrow band around the interface, but used for its storage simple arrays as opposed to the more complex linked lists used in [25].

Whereas narrow-band methods effectively address the computational issue, they still need to explicitly store a full, regular Cartesian grid and additional data structures (e.g., arrays or lists) to pinpoint grid points belonging to the narrow band. Thus, such methods still have storage complexities scaling with the size of the grid. The first attempt to overcome this limitation was due to Strain [23], who used quadtree meshes to reduce memory requirements to the size of the interface, as opposed to the size of the grid. Improvements to the original quadtree method were later developed [4, 12], and an extension to surfaces by means of octree grids was recently introduced [11]. As pointed out by Nielsen and Museth [13], while tree-based methods achieve smaller memory footprints, they also have a number of drawbacks. Most notably, the non-uniform discretization of tree-based meshes makes it non-trivial to use high-order, finite difference schemes. Therefore, such methods use semi-Lagrangian schemes [23], which limit the class of problems which can be tackled to hyperbolic ones; see [13] for further details.

An interesting approach to reduce the memory requirements of the level set method was presented by Bridson in [2], dubbed the "Sparse Block Grid" (SBG) method. In 3D, this method divides the volume of size n^3 into small blocks of size m^3 voxels each. A coarse grid of size $(n/m)^3$ stores pointers to blocks that intersect the interface. Although this method has non-optimal storage complexity, it maintains constant access time similar to the full-grid method.

Recently, Nielsen and Museth introduced the *Dynamic Tubular Grid* (DT-Grid) method [13], a recursive, compressed level-set representation inspired by the compressed-row-storage technique used to represent sparse matrices. The authors showed that the memory requirement of DT-Grid is optimal, i.e., it is proportional to the size of the interface. Moreover, their experiments showed that the 3D DT-Grid is faster and more memory efficient than state-of-the-art octree-based approaches. Huston *et al.* [8] used hierarchical run-length encoding (RLE) in a dimensional-recursive fashion to encode the domain in a series of runs, each associated with a specific run code. Regions away from the narrow band are encoded to just their sign representation, while the narrow band is stored in full precision. Although this method is more flexible than DT-Grid [8], the price paid is a slight increase in computation time and memory usage.

Our method is similar to the SBG method [2], in that it uses tiles, i.e., small, fixed-size blocks, to represent the narrow band around the interface, see Fig. 2. Unlike SBG, our method does not use a coarse grid of pointers to tiles that intersect the interface. Instead, we propose an approach that can implicitly locate the neighbours of each tile, by maintaining a list of active tiles lexicographically sorted by coordinates. Thus, a requirement in every step of our method is to preserve the ordering of the active tiles, such that re-sorting them is

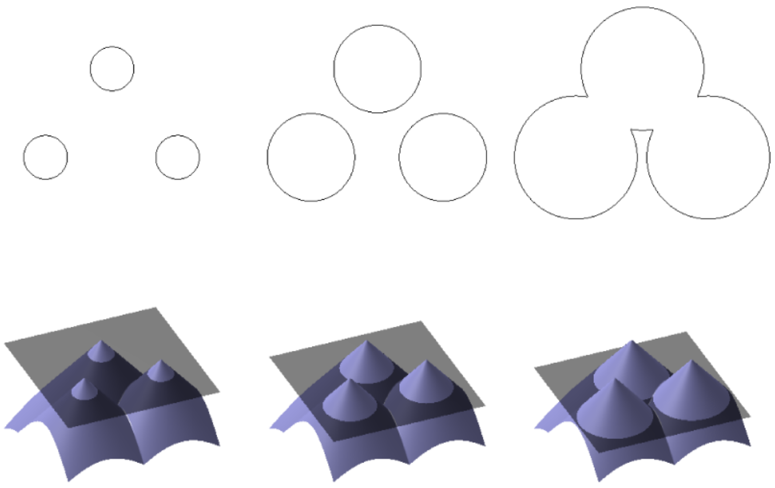

Fig. 1 Level-set example in 2D. *Top row*: evolving interface $\Gamma(t)$. *Bottom row*: corresponding graphs of $\phi(\mathbf{x},t)$ at three different time steps; *left-to-right*: three initial contours expand at constant speed and eventually merge. The interface is given by the intersection of the plane $z=0$ (indicated in *dark grey*) with the graph of ϕ

not necessary. The proposed method also bears some similarities to the method of Lefohn *et al.* [9]. Similar to this approach, our method is tile-based and also uses gradient information from all elements of each active page to determine whether the page has still to be active during the next iteration. However, in contrast to this method, we do not need to store a map of the complete domain or to maintain a list of neighbours for each tile. Further, the complex paging mechanism of Lefohn's method is avoided altogether in our method, and updating the active list involves a simple, sequential traversal of the list. Moreover, our method is not bound to a fixed domain. Instead, it allocates and de-allocates new tiles as the interface propagates to accommodate the deformations. To conclude, the main advantages of our tile-based approach are that the resulting method is highly efficient and straightforward.

3 Overview of the Level Set Method

In the level set method, a closed $(d-1)$-dimensional hyper-surface $\Gamma(t=0)$ is implicitly defined as the zero set of an d-dimensional Lipschitz continuous function $\phi(\mathbf{x},t=0):\mathbb{R}^d\to\mathbb{R}$, e.g., the distance to $\Gamma(t=0)$, with $\mathbf{x}\in\mathbb{R}^d$. Propagating $\Gamma(t)$ along its normal direction with speed v can be done by evolving the function ϕ defined as follows. Let $\phi(\mathbf{x},t=0)=\pm\delta$, with δ the (signed) distance from $\mathbf{x}$ to $\Gamma(t=0)$. Thus, the set $S=\{\mathbf{x}\in\mathbb{R}^d\,|\,\phi(\mathbf{x},t=0)=0\}$ corresponds to the location of the embedded hyper-surface $\Gamma(t=0)$, see Fig. 1. Throughout the paper we assume that the function ϕ has positive values outside the contour, and negative values inside.

With this notation, the equation for the function $\phi(\mathbf{x},t)$ that represents the evolution of $\Gamma(t)$ is then [17]:

$$\frac{\partial\phi}{\partial t}=-v\,|\nabla\phi|\,. \tag{1}$$

If the speed function v is a positive constant, ϕ will shift up along the z-axis, so the contour will expand, see Fig. 1; if v is negative, the contour will shrink.

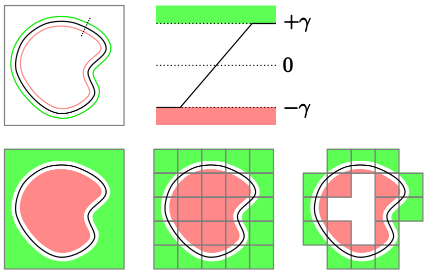

Fig. 2 *Top row*: *Left image*—current interface (black), and the $-\gamma$ and γ iso-contours inside/outside the *black curve*. *Right image*—profile of function ϕ along the dotted line in the first image. *Bottom row*: *Left image*—the domain of function ϕ, the $\geq \gamma$ area outside the interface and the $\leq -\gamma$ area inside the interface, with the curve in the middle. *Middle image*—the domain divided into tiles of equal size. *Right image*—the sparse domain, with inactive tiles (completely inside or outside) removed

Intrinsic geometric properties of the evolving hyper-surface are easily determined from the level set function ϕ. In 3D for example, the mean curvature κ of each level set of ϕ is

$$\kappa = \frac{1}{2}\,\nabla \cdot \frac{\nabla\phi}{|\nabla\phi|}. \tag{2}$$

For more details and applications of the level set method we refer to [14, 15, 17, 20, 24] and the references therein.

3.1 Sparse-Grid Level Set Representations

Considering that the goal is to track a moving interface represented by the zero level set of the embedding function ϕ, solving the PDE in (1) for ϕ on the entire domain would be inefficient, both in terms of memory and computation. Therefore, efficient algorithms for solving level-set equations perform the required computations only in a narrow band along the zero level set, see [1, 3, 18, 25]. Further, to minimize memory requirements sparse methods [2, 8, 13] have been introduced, see Sect. 2.

When a signed distance transform is used as the underlying function ϕ, the calculation can be restricted to a certain distance from the interface, by setting a range of values that we are interested in. The result can be clamped to the range $(-\gamma, \gamma)$, so that the function ϕ at a certain distance away from the interface has a constant value and a zero gradient magnitude, see [18]. This is illustrated in Fig. 2.

Similar to [2, 9], our method divides the domain into fixed-size tiles. Each tile represents a part of the domain of the function ϕ. Tiles that only contain locations with values outside the range $(-\gamma, \gamma)$ are considered irrelevant and thus are not computed or stored. Hence, the remaining tiles will necessarily contain the zero level set (or be very close to it); we call the collection of such tiles the *active set*.

The active set has to be constantly maintained while the simulation is running, so that the moving interface remains inside the active tiles. It follows that new tiles need to be dynamically created when the interface approaches a boundary of a tile, and that tiles can be deleted when they become irrelevant to the simulation.

3.2 Reshaping the Level Set Function

Due to small numerical inaccuracies that build up over time when integrating the PDE in (1), the density of the level sets might not remain constant over the domain, i.e., ϕ is no longer the signed distance transform to the interface. If the density becomes too low, a sparse representation becomes inefficient, as the range $(-\gamma, \gamma)$ spreads over a wider band. On the other hand, if the density becomes too high, instabilities can result. To maintain a consistent level-set density, we make use of the *rescaling speed term* introduced in [9].

If ϕ is a signed distance transform, the gradient magnitude of the function will be unity over the entire active domain, i.e., $|\nabla\phi(\mathbf{x})| = 1$. We can set up a PDE of the form $\frac{\partial\phi}{\partial t} = \text{sgn}(\phi)(1 - |\nabla\phi|)$, which constrains the level sets to have the desired density or gradient magnitude. When discretizing this PDE using a central difference approximation, instabilities arise. This happens because, numerically, information propagates outward from the boundary. Therefore, as suggested in [21], the correct way of discretizing the spatial derivatives is by using an upwind scheme.

Further, using a smoother function to replace the step function in $\text{sgn}(\cdot)$, stability can be improved. If the distance transform is bounded in a very narrow band, the function ϕ itself can be used instead of its sign, see [9].

4 The Proposed Method

In this section we present our memory and computationally-efficient method for evolving the level-set function. As explained in Sect. 3.1, our algorithm makes use of a sparse representation of the level-set function ϕ, and uses a tile-based storage format that only keeps the function values in the vicinity of the zero level. As in other narrow-band methods, all other tiles are considered to be either inside or outside, but too far from the zero level set to have any influence on its evolution.

Our method keeps a list of active tiles, lexicographically sorted by coordinates. Moreover, all our processing steps maintain the list of active tiles sorted in lexicographical order, such that an expensive sorting step during each iteration is avoided. In the following section, the data structure will be explained in further detail.

4.1 The Data Structure

In this section we present the data structure that is central to our approach. The operations that can be applied are the same as those on a full grid, but differ in complexity. The theoretical complexities for various operations as derived in later sections are shown in Table 1.

Basically, the idea behind our structure is to have a list of tiles which are ordered lexicographically by coordinate. Additionally, some basic tile management functionality is needed to maintain the structure over time in an efficient way.

For each tile the following attributes are stored: the coordinates of the tile, a small cube of floating point data, and a set of flags that we will call the *border flags*. The border flags store the sign of the data outside the tile in every direction, in case the tile has no direct neighbour there.

Table 1 Major operations acting on the data structure in the Sorted Tile List method. N is the number of tiles intersecting the interface

Algorithm	Time complexity
Append	$O(1)$
Sequential access with stencil	$O(N)$
Random access	$O(\log N)$
Tile management	$O(N)$

In what follows, we shall call this list of tiles the *active list*. In the list an index or pointer to the actual data is stored. The alternative would be to store the data in the active list itself, but this would incur some extra copying in the management step. Also, in this way, the entire active list might fit into the cache, and it is easier to align the tile data to a larger power of two without wasting memory (given that the tile dimensions are powers of two).

In the following subsections, the initialization and other basic operations (e.g., append, sequential access within stencil, random access, tile management) associated with the data structure are presented.

4.2 Initialization

To build our structure from existing data, we set up a sorted list of active tiles containing the clamped distance transform to the interface $\Gamma(t=0)$, see also Sect. 3. Depending on the input data it might be efficient to compute the distance transform on the whole domain, then delete inactive tiles, and finally, sort the resulting structure. Other methods, similar to that in [7] for converting a triangle mesh to a distance transform, already calculate a clamped distance transform close to the surface of the objects; such methods can be used directly.

The steps performed during the initialization stage are:

- Compute the signed distance transform to the input surface. Note that it is only necessary to compute this up to distance $\pm\gamma$. This results in a sparse volume, defined only near the interface.
- Divide the volume resulting from the first step into tiles, see Fig. 2.
- Add those tiles to the active list that contain parts of the active level set, i.e., locations where $-\gamma < \phi_{ijk} < \gamma$, with ϕ_{ijk} the level set function at position (i, j, k).
- For each tile that does not contain the interface, but shares at least one border with the tile set defined by the active list, update the appropriate border flag of the neighbouring tiles to signify that this tile is inside (filled with value $-\gamma$) or outside (filled with value γ).
- Sort the resulting *active list* in lexicographical order, if the list was not generated in this order.

4.3 Append Operation

The append operation adds a tile to the end of the list. This is analogous to the ‘push’ operation in [13]. As the structure is lexicographically ordered by coordinates, the new tile must have a coordinate higher than that of the existing last tile. To add a tile, the new tile attributes are written to the end of the active list. Hence, adding one tile has complexity of order $O(1)$.

4.4 Sequential Access With Stencil

Sequential access can be simply achieved by moving a pointer over the list. However, as many operations also need access to neighbouring tiles adjacent to the tile that is being iterated, and since this requires some extra bookkeeping, we illustrate such a traversal of the active list in Algorithm 1. In this algorithm, `size` corresponds to the number N of tiles, `coord` contains the coordinates for each tile, `B` denotes the number of neighbours and `neighbourhood` denotes the relative coordinates of the neighbourhood tiles. Coordinates of tiles are compared lexicographically, and are added and subtracted component-wise like vectors. The user-defined function *iter* is called for each active tile, passing two parameters: `match`, a bit field signifying which neighbours are present and which are not,

Algorithm 1 Iterating over the sorted tile list. The function *iter* is called for each tile

```
Input: size, coord[size], B, neighbourhood[B]
1: offset ← 0 {beginning of tile-set}
2: for i = 0 to B − 1 do
3:   ptr[i] ← 0
4: end for
5: while offset < size do
6:   cur ← coord[offset] {take coord of current tile}
7:   match ← 0 {bit field of neighbours that exist}
8:   for i = 0 to B − 1 do
9:     c ← cur + neighbourhood[i] {calculate coord of neighbour i}
10:    while ptr[i] < size and coord[ptr[i]] < c do
11:      ptr[i] ← ptr[i] + 1 {track neighbour}
12:    end while
13:    if ptr[i] < size and coord[ptr[i]] = c then
14:      match ← match | $2^i$ {if coord matches, neighbour i exists}
15:    end if
16:  end for
17:  iter (match, ptr) {call iterator}
18:  offset ← offset + 1 {advance to next tile}
19: end while
```

and `ptr`, the offset into the active list for each neighbour (only valid if the according bit in `match` is set). In 2D, an example of a 3^2 `neighbourhood` definition would be $((-1,-1), (-1,0), \ldots, (1,0), (1,1))$, and thus `B` would be 8. This extends trivially to three or more dimensions.

The complexity of this algorithm is linear in the number of tiles. This can be derived in the following way. There are `B` + 1 pointers, `offset` and `ptr[0..B-1]`, that are initialized to the start of the active list, at the beginning of the algorithm. With each top-level iteration of the **while** loop, `offset` is incremented, and usually several of `ptr[0..B-1]` will also be incremented at least once, as these track a certain neighbourhood around the current tile with index `offset`. All pointers are incremented at most `size` times, after which they have reached the end of the active list. At the end of the algorithm, `offset` will always be equal to `size`. The other pointers will not necessarily have reached the end, so that the total number of pointer increments is smaller or equal to (`B` + 1)`size`. As `B` is constant, the time complexity is thus linear in the number of tiles.

The last tile iteration in a time step should write its resulting data blocks in active list order, consecutively, and thus remove deleted blocks without the need for an extra iteration over the data. This ensures that the tile data is always at consecutive memory locations, and ordered in the same (lexicographical) order as the active list. Moreover, this also means that during the next tile management step, free tiles can easily be allocated with a simple counter starting from the end of the data, without the need for more complex memory management strategies.

4.5 Random Access

As the *active list* is an ordered list of tile coordinates, an efficient method to look for a random tile is to do a binary search, comparing at each step the desired coordinate to the coordinate at the current position. Thus, the complexity of a random access operation is $O(\log N)$.

4.6 Tile Management

In the tile management step, the list of active tiles is updated as follows. Tiles that are no longer close to the zero level set are removed, and tiles bordering the zero level set that are needed in the next time-step will be added. For each currently-active tile, it is first determined which of the neighbouring tiles are needed in the next time step. If the interface approaches a tile border, the tile at the other side of that border has to be present in the next time-step to continue the computation. The borders of a tile that are being approached by the evolving contour, are signalled through a set of activity flags by the time-stepping computation (as explained in Sect. 4.7). If the activity flag for a certain border is set, a tile has to be created if it is not yet present in the direction of that flag. If the interface has just left a certain tile, all the activity flags for that tile will be zero. If none of the neighbouring tiles requests for the tile to be retained, it can be safely removed to free memory.

The basic idea then is to iterate over the list of tiles, and for each tile to expand the tile set by creating tiles in the directions whose activity flags are set. Along the way, one has to remember which tiles are inactive and not requested by any other neighbouring tile. Such tiles will be discarded.

A straightforward implementation of this idea would create tiles multiple times when they are requested by different tiles, resulting in tile duplicates. It helps to look at this in another way: as new tiles will always be direct neighbours of the current tile-set, the process can be seen as a morphological dilation of the set of active tiles by a 3^3 structuring element [19]. For each element in the dilated version of the set, it should then be determined whether to create, remove or keep the tile at that position. This assures that new tiles will only be created at most one time.

4.6.1 Dilation of a Sorted List

Algorithm 2 shows the main steps needed for iterating over a sorted tile list, while dilating it on the fly. Here `size` is the number of tiles, and `coord` is a sorted list of coordinate vectors for each tile. The variables `B` and `neighbourhood` define the size and coordinate tuples of the structuring element. Here, in contrast to the algorithm presented in Sect. 4.4, `neighbourhood` must include the middle tile $(0, 0)$. The variable `maxcoord` tells where to stop iterating: a value of (∞, ∞) indicates that iteration should proceed until the entire tile-set is dilated. The user-defined function *iter* is called for each tile in the dilated version of the list, passing three parameters: `cur`, the current coordinate, `match`, a bit field signifying which neighbours are present, and `ptr`, the offset into the tile list for each neighbour (only valid if the according bit in `match` is set).

The algorithm works by moving the structuring element over the tile-set in lexicographical order, continuously selecting the first tile that is intersected by it, see Fig. 3. In the beginning, all pointers are initialized to 0 (the beginning of the active list). Then, the relative position of the first tile that intersects the structuring element is chosen, and its position is set as the current one (lines 5–11). Next, the algorithm checks if it has reached the end of the data (line 12). If not, it builds a bit field of all the neighbours that are present (lines 15–20) and calls the iterator (line 21). At the end of the loop, all the pointers that point to matched neighbours are increased (line 22–26). Clearly, the complexity of this algorithm is linear in the number of tiles, provided that the complexity of the iteration function *iter* is constant.

4.6.2 Tile Update

Now that the algorithm for iterating over the sorted and dilated list of tiles is established, we use it to do the actual tile management (function *iter*) shown in Algorithm 3.

Algorithm 2 Iterating over a dilated version of a lexicographically-sorted list of tile coordinates, maintaining the sorted order. Function *iter* is called for each tile

```
Input: size, coord[size], B, neighbourhood[B], maxcoord
 1: for i = 0 to B − 1 do
 2:     ptr[i] ← 0 {beginning of tile-set}
 3: end for
 4: loop
 5:     cur ← maxcoord {init to max. coord.}
 6:     for i = 0 to B − 1 do {loop over structuring element}
 7:         if ptr[i] < size then
 8:             {remember first tile in lexicographic order}
 9:             cur ← min(cur, coord[ptr[i]] − neighbourhood[i])
10:         end if
11:     end for
12:     if cur = maxcoord then
13:         break {end of tile-set reached}
14:     end if
15:     match ← 0 {bit field of neighbours that exist}
16:     for i = 0 to B − 1 do
17:         if ptr[i] < size and coord[ptr[i]] = (cur + neighbourhood[i]) then
18:             match ← match | $2^{i}$ {if coord matches, neighbour i exists}
19:         end if
20:     end for
21:     iter (cur, match, ptr) {call the iterator}
22:     for i = 0 to B − 1 do
23:         if match & $2^{i}$ then
24:             ptr[i] ← ptr[i] + 1 {advance structuring element}
25:         end if
26:     end for
27: end loop
```

The goal of this iterator is to determine which tiles to keep, which to add and which to remove. For this purpose, the active flags of the neighbours are examined (lines 1–6). Here `neighbour_trigger` is a list of bit fields that signify when a tile needs to be retained or created, given the activity flags of the neighbours and the tile itself. We want to "trigger" a tile when there is activity in the tile itself, or if the border facing that tile or one of the surrounding tiles is active; this is illustrated in Fig. 4. There are two different cases (line 7): either this is a currently active tile (Sect. 4.6.3), or this is a possibly new tile (Sect. 4.6.4). This is determined by checking if the middle (current) tile exists. Here `MID` is the offset of the middle tile $(0, 0)$ in the *neighbours* list.

4.6.3 Existing Tile

If the tile is a current tile, we determine whether we need to keep it, by looking at the value of `trigger` (line 9). If it is set, we keep the tile and just append it to the new active list for the next time-step, using `activelist_append(coord, tileid)` (line 10). Otherwise, it is not added to this list, and will not be active during the next time step. Another step that needs to be taken when a tile is removed, is to notify the neighbours whether the removed tile was inside or outside by appropriately setting their border flags. First, we fetch the

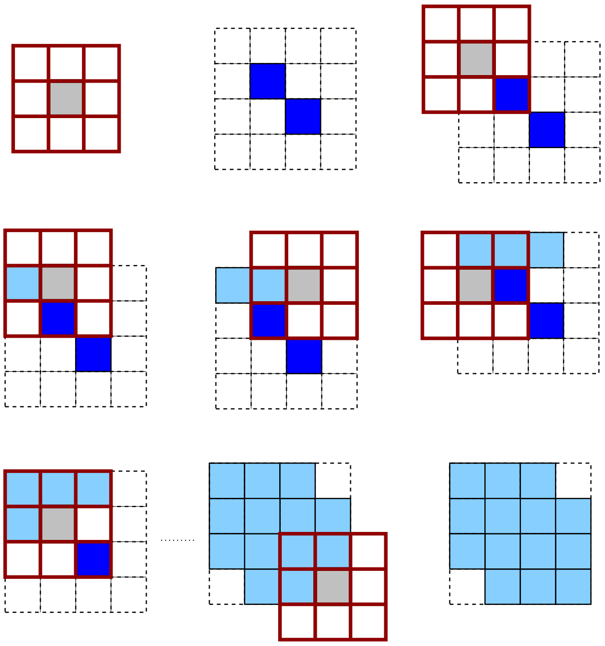
Fig. 3 A 3 × 3 structuring element is moved over the tile-set in lexicographical order, dilating the set in a tile-by-tile fashion. *Top-left*: structuring element, *center*: original set consisting of two active tiles, *left-to-right*, *top-to-bottom*: the structuring element starts at the first possible position, and shifts over the set, until the end result is obtained (*bottom right*)

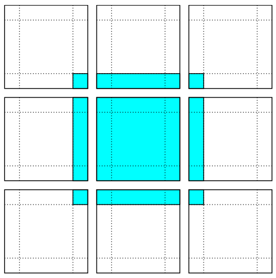
Fig. 4 The set *neighbour_trigger*, in the 2D case. The current tile and its 8 neighbours are shown, each divided into 9 areas. The marked areas are included in the set

classification (inside or outside) using `classify(x)`, which extracts the bit in activity flags `x` that classifies the tile as either inside or outside (line 12), see Sect. 4.7. We then set the border flag using `setborderflag(p,q,v)` which sets the value of border flag `q` of tile `p` to `v`, see Sect. 4.1. Here `reverse(p)` is a function that reverses a border direction. This is needed to set the flag of the border facing the current tile. In 3D, this is simply defined as 27 − `p` (line 16).

4.6.4 New Tile

For a tile that we are considering to create, we first look at the value of `trigger` (line 21). If this tile was triggered, then it must be created, otherwise nothing is done. Next, we check the border flags of the surrounding neighbours using `borderflag(p,q)`, which computes the value of border flag `q` of tile `p`, to determine how to initialize the new tile (lines 22–27). Then we allocate a new tile using `allocate_tile(flag)`, which gives us the next free tile offset, and fills it with values $-\gamma$ or γ depending on `flag`. The border flags of the new

Algorithm 3 *iter* function for tile management

```
Input: cur, match, ptr, B, neighbourhood[B], neighbour_trigger[B],
    activeflags[], index[]
 1: trigger ← 0 {default to 0, unless triggered}
 2: for i = 0 to B − 1 do
 3:   if match & $2^{i}$ and (activeflags[ptr[i]] & neighbour_trigger[i])
      then
 4:     trigger ← 1 {neighbour i has contour approaching this tile}
 5:   end if
 6: end for
 7: if match & $2^{MID}$ then {current tile already exists}
 8:   tileid ← index[ptr[MID]]
 9:   if trigger then {if triggered, keep tile}
10:     activelist_append(cur, tileid)
11:   else {otherwise, remove it}
12:     flag ← classify(ptr[i]) {classify as inside or outside}
13:     for i = 0 to B − 1 do
14:       if match & $2^{i}$ then
15:         {notify neighbours of removal}
16:         setborderflag(index[ptr[i]], reverse(i), flag)
17:       end if
18:     end for
19:   end if
20: else {current tile doesn't exist yet}
21:   if trigger then {if triggered, create tile}
22:     flag ← 0 {determine if new tile should be inside or outside}
23:     for i = 0 to B − 1 do
24:       if match & $2^{i}$ then
25:         flag ← flag | borderflag(index[ptr[p]], reverse(p))
26:       end if
27:     end for
28:     tileid ← allocate_tile(flag) {create and init. new tile}
29:     for i = 0 to B − 1 do {init. border flags}
30:       setborderflag(tileid, i, flag)
31:     end for
32:     activelist_append(cur, tileid)
33:   end if
34: end if
```

tile are initialized (lines 29–31), and finally, the tile is appended to the active list for the next time-step (line 32).

Note that, as the complexity of Algorithm 2 is linear in the number of tiles, and Algorithm 3 executes a constant number of steps, the overall complexity of the tile management stage of the proposed method is linear in the number of tiles.

4.7 Updating the level-set

The time-integration step evolves the level set function ϕ according to the generic level-set PDE in (1). To do this, we make use of the *sequential access with stencil* algorithm described in Sect. 4.4.

For the sake of clarity, we explain the time-integration step using our method for a specific level set PDE (see (3)), using forward first order differences in time, and first and second order differences in space with a 3^3 stencil. However, we also implemented fifth-order accurate HJ-WENO discretization for spatial hyperbolic terms (see Sect. 5) and third-order accurate TVD Runge-Kutta for time integration, the details of which can be found in, e.g., [10] and [22]. Note that larger stencils can easily be accommodated by our method if the application at hand requires it, by increasing the tile size.

The level-set PDE we are here concerned with, consists of three terms: a data-dependent speed term, a surface-area minimizing term (based on the mean curvature of the surface, see (2)), and a rescaling-speed term (see Sect. 3.2) to keep the level set as closely as possible to a signed distance transform, i.e.,

$$\frac{\partial \phi}{\partial t} = -F(\mathbf{x}, t)\,|\nabla \phi| + \epsilon\kappa\,|\nabla\phi| + \operatorname{sgn}(\phi)\,(1 - |\nabla\phi|)\,. \tag{3}$$

The data-dependent speed term F can vary in both position and time. The strength of the curvature-based term (the second term) is determined by a constant ϵ, which is usually fixed. The rescaling-speed term completely depends on the $\operatorname{sgn}(\cdot)$ function used, which otherwise is free of parameters, see Sect. 3.2.

When discretizing this equation, the first and third terms must be approximated using an upwind scheme, which satisfies the entropy condition [21]. The curvature-based term can simply be approximated using central differences. Initially, the first order approximations to the spatial derivatives of the level set function ϕ at each position (i, j, k) in the tiles are calculated: central differences D^{0x}_{ijk}, D^{0y}_{ijk}, D^{0z}_{ijk}, backward differences D^{-x}_{ijk},D^{-y}_{ijk}, D^{-z}_{ijk} and forward differences D^{+x}_{ijk},D^{+y}_{ijk}, D^{+z}_{ijk}. Central differences are necessary for the computation of the curvature term, whereas backward and forward ones are needed for the upwind scheme. For evaluating the curvature we also need to compute central difference approximations to all second order spatial derivatives, D^{0xx}_{ijk}, D^{0xy}_{ijk}, ..., D^{0zz}_{ijk}.

To compute the curvature we start from (2). The gradient of ϕ is approximated by central differences, i.e., $\nabla\phi \approx (D^{0x}_{ijk}, D^{0y}_{ijk}, D^{0z}_{ijk})$. The central difference approximation of the gradient magnitude is denoted by ∇^0, i.e.,

$$\nabla^0 = \left({D^{0x}_{ijk}}^2 + {D^{0y}_{ijk}}^2 + {D^{0z}_{ijk}}^2\right)^{\frac{1}{2}}$$

By approximating all first and second order derivatives in (2) one finds the following approximation for the curvature κ:

$$\kappa \approx \frac{1}{2}\sum_{\alpha=x,y,z}\left(\frac{D^{0\alpha\alpha}_{ijk}}{\nabla^0} - \frac{D^{0\alpha}_{ijk}}{(\nabla^0)^3}\sum_{\beta=x,y,z} D^{0\beta}_{ijk} D^{0\beta\alpha}_{ijk}\right). \tag{4}$$

Three approximations of the gradient magnitude need to be computed: the central difference approximation ∇^0 as defined above, and two other ones for the upwind scheme [21], i.e.,

$$\begin{aligned}
\nabla^+ = \Big(&\max{}^2\left(D^{-x}_{ijk}, 0\right) + \min{}^2\left(D^{+x}_{ijk}, 0\right) + \max{}^2\left(D^{-y}_{ijk}, 0\right) \\
&+ \min{}^2\left(D^{+y}_{ijk}, 0\right) + \max{}^2\left(D^{-z}_{ijk}, 0\right) + \min{}^2\left(D^{+z}_{ijk}, 0\right)\Big)^{\frac{1}{2}}, \\
\nabla^- = \Big(&\max{}^2\left(D^{+x}_{ijk}, 0\right) + \min{}^2\left(D^{-x}_{ijk}, 0\right) + \max{}^2\left(D^{+y}_{ijk}, 0\right) \\
&+ \min{}^2\left(D^{-y}_{ijk}, 0\right) + \max{}^2\left(D^{+z}_{ijk}, 0\right) + \min{}^2\left(D^{-z}_{ijk}, 0\right)\Big)^{\frac{1}{2}}.
\end{aligned}$$

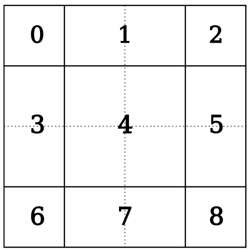

Fig. 5 *Guard bands* in the time-step iteration guarding the borders of a 4×4 tile, used to set activity flags for corners $0, 2, 6, 8$; borders $1, 3, 5, 7$; and center 4

All calculations above need a 3^3 stencil centered around the current element. At the borders of a tile, values from neighbouring tiles are required. These values can be fetched using the neighbour pointers provided by the algorithm in Sect. 4.4. If a neighbour tile is not active at the moment, a dummy tile filled with either $-\gamma$ (inside) or γ (outside) is used, depending on the border flag in the direction of the required tile.

The value at the current position for the next time-step is then calculated using Euler integration in time, i.e.,

$$\phi_{ijk}^{n+1} = \phi_{ijk}^{n} + \Delta t\, S,$$

in which Δt is the time-step and S is defined as the following sum of speed terms:

$$\begin{aligned} S = &-\big(\max(F_{ijk}, 0)\nabla^{+} + \min(F_{ijk}, 0)\nabla^{-}\big) \\ &+ \big(\epsilon\, \kappa_{ijk}^{n} \nabla^{0}\big) \\ &+ \max\big(\operatorname{sgn}(\phi_{ijk}^{n}), 0\big)\big(1 - \nabla^{+}\big) \\ &+ \min\big(\operatorname{sgn}(\phi_{ijk}^{n}), 0\big)\big(1 - \nabla^{-}\big), \end{aligned}$$

where the sign of F_{ijk} determines whether the motion is inward or outward w.r.t. the interface, and κ_{ijk}^{n} is the central difference approximation to the mean curvature at the current position.

During these computations, the old set of tiles from the previous time-step needs to be read, while the values for the new time-step have to be written. To accomplish this, we temporarily need to double the memory requirements for each tile. After the values for the next time-step have been computed for all tiles, the old values are discarded, and the whole process is repeated.

Activity Flags

While calculating the level set function for each element in the tile, an activity flag for each border, corner and central part is maintained using guard bands (see Fig. 5). This flag is set if the result in that band is between the two cut-off values, that is $-\gamma < \phi_{ijk} < \gamma$. The stored flags are used in the tile management step (Sect. 4.6), which determines which tiles need to be activated and which can be deactivated.

In case that all the resulting activity flags are zero, meaning that there is no activity in a tile at all, the tile can be classified as entirely inside ($\phi_{ijk} = -\gamma$) or entirely outside ($\phi_{ijk} = \gamma$). This classification is stored with the activity flags, and used to update the border flags of neighbouring tiles when the tile is removed.

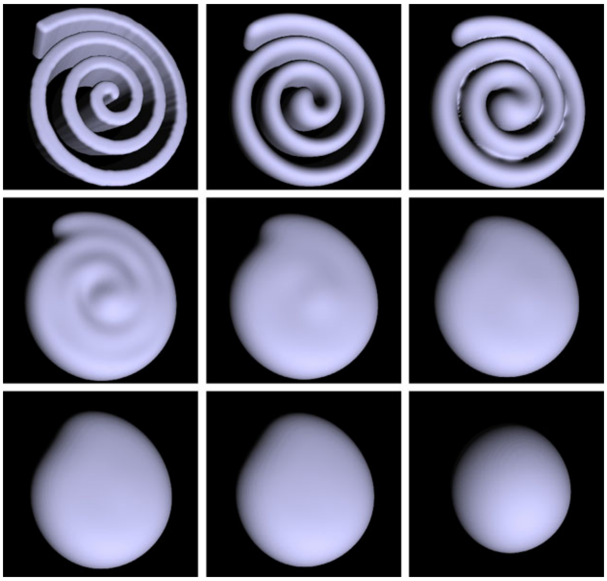
Fig. 6 *Left-to-right, top-to-bottom*: snapshots of the volume-conserving mean curvature flow simulation by the Sorted Tile List method, taken every 100 time steps. The final image shows the result after 1700 time steps

5 Results

In this section we present both numerical and performance-related results that show the strengths and weaknesses of our new approach, compared to previous methods.

In [13], the DT-Grid method was already compared to other state-of-the-art level-set data structures, and it was shown that this method achieves the highest performance. Therefore, to measure the relative performance of our new method, we implemented the DT-Grid method [13] and used it as a reference. In all comparisons in this section, the same level-set simulation code was used for the computational routines which are common to both our method and DT-Grid, so as to fairly compare the performance of both representations. Only in the indicated cases, our method was additionally optimized using Intel SIMD instructions (SSE2) which compute four floating point values at once, in a single operation. The machine used for benchmarking was an Intel Core 2 Quad 2.4 GHz, running a recent 32-bit Linux kernel; the compiler used was GNU g++ 4.2.4.

5.1 Mean Curvature Flow

In the first experiment, we simulated a sphere collapsing under mean curvature flow. The simulation was performed on a 256^3 grid. The simple discretizations from Sect. 4.7 were used. The number of time-steps it took for the sphere to collapse to a point and then disappear was equal for both the DT-Grid and our method, confirming that the evolution of the sphere was the same. The complete simulation took 2674 seconds using the DT-Grid and 429 seconds using our method, thus resulting in a speed-up factor of about 6.2.

5.2 Volume-Conserving Mean Curvature Flow

Next we benchmarked our algorithm by performing the simulation of an extruded spiral evolving under volume-conserving mean curvature flow [18], see Fig. 6. The PDE describing

Table 2 Performance comparison of level-set data structures

Method	Time (s)
DT-Grid	680.0
Sorted Tile List method (tile size 4^3)	240.0
Sorted Tile List method (tile size 3^3)	230.0
Sorted Tile List method (SSE2, tile size 4^3)	80.1

the evolution is

$$\frac{\partial \phi}{\partial t} = (\kappa - \overline{\kappa})|\nabla \phi|, \tag{5}$$

where κ is the mean curvature of the level sets in $(-\gamma, \gamma)$ and $\overline{\kappa}$ is the average curvature of the interface. To approximate (5), second-order central differences (see Sect. 4.7 and (4)) were used for the parabolic term ($\kappa|\nabla\phi|$) and the fifth-order accurate HJ-WENO scheme [16] for the spatial hyperbolic term ($\overline{\kappa}|\nabla\phi|$). The average curvature of the interface $\overline{\kappa}$ is computed by

$$\overline{\kappa} = \frac{\int_\Gamma \kappa\, \delta(\phi)|\nabla\phi|\, dx}{\int_\Gamma \delta(\phi)|\nabla\phi|\, dx}, \tag{6}$$

where the delta function is approximated by [18]

$$\delta(\phi) = \begin{cases} \frac{1}{2\epsilon}(1 + \cos(\frac{\pi\phi}{\epsilon})) & \text{if } \phi < \epsilon, \\ 0 & \text{otherwise,} \end{cases} \tag{7}$$

where ϵ is a parameter proportional to the grid spacing. Finally, a third-order accurate TVD Runge-Kutta time-stepping method [22] was employed. The simulation was performed on a 128^3 grid, and the spiral initially had a radius of 50 and a depth of 40.

In the SSE-optimized version of our method, we used aligned loads to fetch stencil values from the tile set and perform all stencil computations on four values at once. The tile-based memory storage format of our method allows for this vectorization to be done easily: the only change in the computation code was to replace the occurrences of *float* with *vec4*, an overloaded type that represents four floats at once. These optimizations are not feasible for DT-Grid, as the extra work spent in collecting stencil values from the voxel-centric storage and storing them into aligned memory areas would outweigh the gain of computing four floating point values at once.

Timings for the first 1000 iterations are shown in Table 2. It can be seen that our method is about three times faster than the DT-Grid method, and the SSE-optimized version of our method is about 8.5 times faster. Figure 7 shows the computation time per time step, for DT-Grid, our method, and our SSE-optimized method. As can be seen, the time usage of our method is almost constant, as voxels are grouped in tiles and only full tiles are switched on and off for computation. Thus, the amount of computations differs less from time step to time step, and is consistently lower than that of DT-Grid. The graph of the SSE-optimized method follows the graph for the non-optimized method.

5.3 Memory Usage

We also compared the memory usage of our method (using tiles of size 4^3 voxels) and the DT-Grid method, for the same simulation as in Sect. 5.1, see Fig. 8. During the simulation,

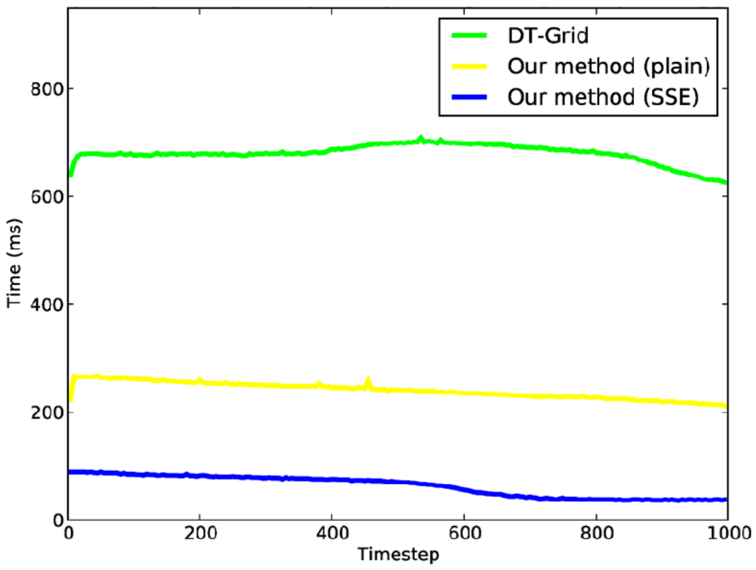

Fig. 7 Computation time per time step, for DT-Grid, our method, and our SSE-optimized method

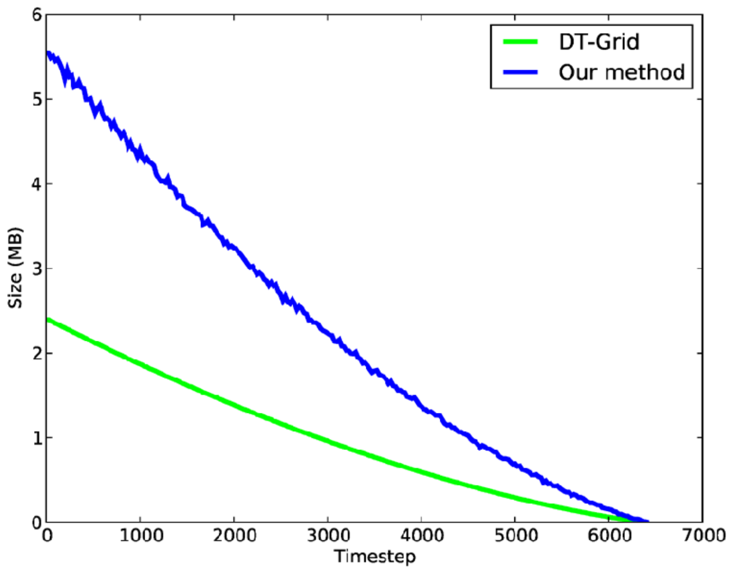

Fig. 8 Memory usage of our method and the DT-Grid method for the same level set computation (for comparison, a full 256^3 grid would take 67.1 Mb of memory)

the size of the interface shrinks to a point, and then around time step 6500 it disappears. From Fig. 8 it can be concluded that the memory usage of our method is a factor of about 2.5 larger than that of DT-Grid, seemingly independent of the size of the interface.

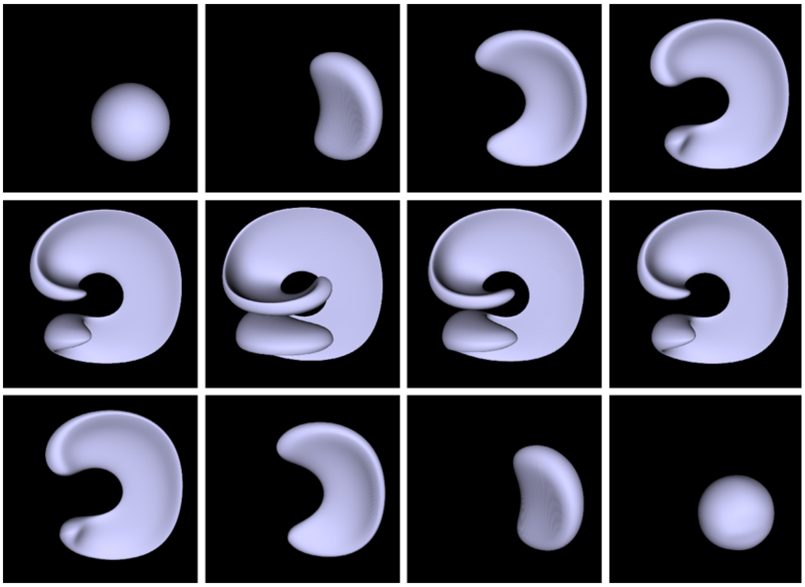

Fig. 9 Simulation of periodic velocity field advection, as proposed by Enright *et al.* [5]. Our method allows high resolution level set representations (a 1024^3 grid was used for this experiment). *Left-to-right*, *top-to-bottom*: snapshots taken during the simulation

5.4 Periodic Velocity Field Advection

Similar to DT-Grid, our method has a low memory footprint, allowing large computational grids, or conversely, large resolution level-set representations. Enright *et al.* [5] proposed an experiment to demonstrate the excellent volume-conservation property of their particle level set (PLS) method. Whereas they used a 100^3 grid and were able to resolve the interface, we ran the experiment on a 1024^3 grid, similar to Nielsen and Museth [13] for their DT-Grid method. Figure 9 shows some snapshots taken during the simulation. As can be seen, our method can fully resolve the interface on a 1024^3 grid, similar to the DT-Grid method. The peak memory usage of our method was 149 MB, which is about twice as much as for DT-Grid, whereas a full-grid method would need more than 4 GB of memory, see [5, 13] for further details.

In cases where the velocity field cannot be evaluated analytically (as is the case for Enright's test), or at arbitrary grid locations, standard interpolation methods such as trilinear interpolation have to be used. Further, if the velocity field is defined on a grid with a different resolution than the level set grid, resampling followed by interpolation can also be employed.

To further test the accuracy of our method, we successively increased the grid resolution and observed the maximum time t_d that can be achieved without deteriorating the developing thin sheets. The grid resolution G was set to $G = 128^3, 256^3, 512^3, 1024^3$ and the period of the field was set to $T = 5$, so that the maximum deformation takes place when the simulation time t becomes $t_d = T/2$. The results of this experiment are shown in Fig. 10. As also found in [6] and shown in Fig. 10, a grid resolution of 512^3 is sufficient to accommodate the deformation, provided that the period of the field is at most $T = 3.0$. Note that, periods as large as $T = 4$ are also possible if the grid resolution satisfies $G \geq 1024^3$.

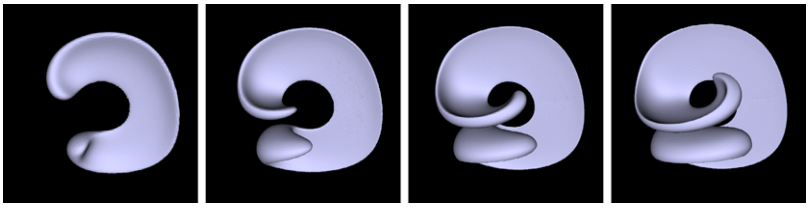

Fig. 10 Maximum deformation versus increasing grid resolution for Enright's test. *Left-to-right*: snapshots at maximum deformation time $t_d = 0.77, 1.1, 1.55, 2.05$ obtained at increasing grid resolutions $G = 128^3, 256^3, 512^3, 1024^3$

5.5 Tile Size Considerations

The drawback of our approach is that it has a larger memory overhead than DT-Grid, because of the granularity of the tiles. This overhead depends on the tile size—up to a point, the tile size is a compromise between computational efficiency and memory overhead. Choosing a tile size of 3^3 voxels hampers performance as it becomes inconvenient to use SSE SIMD instructions. The use of even smaller tiles would make the tile management overhead larger and similar to that of DT-Grid (see below), as more smaller active tiles need to be managed. Choosing a larger tile size results in both memory and computation overhead, as more values are computed unnecessarily.

We have also performed the experiment of Sect. 5.1 with 3^3 tiles and without SIMD optimizations, and did not observe any significant difference with regard to speed, compared to the case when 4^3 tiles were used. Thus, in this case, one can conclude that the gain due to less numerical computation was offset by the added tile management overhead.

5.6 Tile Management Overhead

In practice the DT-Grid method has the advantage that the interface can be represented more sparsely. This implies less memory usage and less time spent on actual computation, e.g., no need to compute 4^3 voxels for those tiles in which only one voxel of the interface resides. Yet, DT-Grid adds significantly more overhead for maintaining the stencils. The following are some of the requirements of DT-Grid:

– an iterator for each stencil voxel is needed, which means 39 instead of just the 27 that we have for the surrounding blocks (HJ-WENO case);
– the iterators have to be advanced after each computed voxel; in our case, time can be spent entirely on computing the values of a tile before advancing them;
– DT-Grid uses 9 arrays (three for each dimension), while our method uses only a flat sorted list, making it easier to append ("push") without having to manage various connected components.

Quantifying the exact impact of the tile management overhead is quite complex in practice. This overhead is always lower for our method than for DT-Grid, as the number of tiles is always smaller (assuming the tile size is not 1^3) than the number of voxels. Furthermore, assuming a tile size of 4^3, the overhead can be 64 times less. This would happen only in the specific situation that DT-Grid would fill up every 4^3 tile entirely. In this case, the memory usage of DT-Grid and our method would also be (approximately) the same. In realistic cases, the interface will only occupy part of the tiles, and DT-Grid would only store voxels close to the interface. The average number of voxels per tile that are part of the interface will also be substantially larger than 1, as the level set embeds a 2-D surface in 3-D space. Simplistically spoken, a tile embedding an axis-aligned plane would use 4^2 of the 4^3 voxels, implying 4 times overhead. Experimental measurements using profiling tools revealed this factor to be consistently around 2.8 (bigger is worse), which approximately coincides with the memory overhead that we measured in Sect. 5.3. This implies that the effective savings in tile management overhead will be a factor of about $\frac{64}{2.8} \approx 23$.

5.7 Discussion of Our Method

As our method is a hybrid between sparse methods with data structures for efficient level set computation such as DT-Grid on the one hand, and dense representation such as the full grid-based method on the other, it has favorable properties of both.

First, the speed and optimization potential of our method is crucial. As the main data structure is a sorted list processed in order, cache coherence can be maximized. Also, the tile size can be set so that tiles start at memory page boundaries, thus minimizing memory transfer overhead. Furthermore, as the values for a tile are stored together, and there is little data dependency (conditional logic) in the computation of the values for a tile, fine-grain parallelism in the form of SIMD instructions can be utilized to compute up to a whole tile of values at the same time.

Hierarchical structures such as octrees and DT-Grid can be quite intricate, while our method is much more straightforward to implement. As tiles are computed in a fashion similar to the full-grid approach, this constitutes an advantage when starting from a grid-based or narrow-band implementation. The only major difference is that an active list has to be maintained around the level set surface. There are no special structures with pointers that need to be maintained, which also facilitates hardware implementations.

The memory locality within tiles can also increase the performance of methods used to *visualize* interfaces, such as the Marching Cubes algorithm, as finding neighbour values is very fast. Here too, fine-grain parallelism can be used, for example to find the zero crossings for the entire tile at once.

5.8 Parallelization Over Multiple CPUs

Our method is well suited for both fine and coarse grain parallelism. Within a tile, all values can be computed at once, which we demonstrated above by using SIMD instructions. Secondly, all the tiles could in principle be computed at the same time, by parallelizing this process over multiple CPUs.

Moving the stencil of neighbouring tiles over the data structure (Sect. 4.4) is inherently a serial operation. This has to be taken into account when dividing the computational work over processors. For example, we could index into the structure using random access (Sect. 4.5), and use this to initialize the neighbour pointers for each processor once at the start of the iteration process.

The other challenge lies in parallelizing the tile management step, as the active list has to be maintained lexicographically sorted by coordinate. This could be achieved by allocating a part of the active list to each processor, having it process the dilation for that part of the list, then collecting the results in the correct order into a new active list.

6 Conclusions and Future Work

In this paper we have presented an efficient data structure with associated operations for the level set representation, and compared the resulting method with the current method of choice, the DT-Grid method. With regard to performance, given the same numerical simulation code, our method turned out to be faster by a significant factor. After fine-grain parallelization using SIMD instructions the performance gain was further increased to a factor of 8.5.

As our method is also well suited for coarse-grain parallelization, we are in the process of devoting further work to various possibilities for leveraging highly parallel architectures such as GPUs. It would also be interesting to see whether it is possible to reduce the larger memory requirements of our method, without sacrificing one of its prominent advantages, namely, low data structure management overhead.

References

1. Adalsteinsson, D., Sethian, J.A.: A fast level set method for propagating interfaces. J. Comput. Phys. **118**(2), 269–277 (1995). http://dx.doi.org/10.1006/jcph.1995.1098
2. Bridson, R.E.: Computational aspects of dynamic surfaces. Ph.D. thesis, Stanford University, Stanford, CA, USA (2003). Adviser-Ronald Fedkiw
3. Chopp, D.L.: Computing minimal surfaces via level set curvature flow. J. Comput. Phys. **106**(1), 77–91 (1993). http://dx.doi.org/10.1006/jcph.1993.1092
4. Droske, M., Meyer, B., Rumpf, M., Schaller, C.: An adaptive level set method for medical image segmentation. In: IPMI '01: Proceedings of the 17th International Conference on Information Processing in Medical Imaging, pp. 416–422. Springer, London (2001)
5. Enright, D., Fedkiw, R., Ferziger, J., Mitchell, I.: A hybrid particle level set method for improved interface capturing. J. Comput. Phys. **183**, 83–116 (2002)
6. Enright, D., Losasso, F., Fedkiw, R.: A fast and accurate semi-Lagrangian particle level set method. Comput. Struct. **83**(6–7), 479–490 (2005)
7. Fournier, M., Dischler, J.M., Bechmann, D.: 3D distance transform adaptive filtering for smoothing and denoising triangle meshes. In: GRAPHITE '06: Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia, pp. 407–416. ACM, New York (2006). http://doi.acm.org/10.1145/1174429.1174497
8. Houston, B., Nielsen, M.B., Batty, C., Nilsson, O., Museth, K.: Hierarchical RLE level set: a compact and versatile deformable surface representation. ACM Trans. Graph. **25**(1), 151–175 (2006)
9. Lefohn, A.E., Kniss, J.M., Hansen, C.D., Whitaker, R.T.: A streaming narrow-band algorithm: interactive computation and visualization of level sets. IEEE Trans. Vis. Comput. Graph. **10**(4), 422–433 (2004)
10. Liu, X.D., Osher, S., Chan, T.: Weighted essentially non-oscillatory schemes. J. Comput. Phys. **115**(1), 200–212 (1994). http://dx.doi.org/10.1006/jcph.1994.1187
11. Losasso, F., Gibou, F., Fedkiw, R.: Simulating water and smoke with an octree data structure. In: SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, pp. 457–462. ACM, New York (2004). http://doi.acm.org/10.1145/1186562.1015745
12. Min, C.: Local level set method in high dimension and codimension. J. Comput. Phys. **200**(1), 368–382 (2004). http://dx.doi.org/10.1016/j.jcp.2004.04.019
13. Nielsen, M.B., Museth, K.: Dynamic Tubular Grid: An efficient data structure and algorithms for high resolution level sets. J. Sci. Comput. **26**(3), 261–299 (2006). http://dx.doi.org/10.1007/s10915-005-9062-8
14. Osher, S., Fedkiw, R.: Level set methods: an overview and some recent results. J. Comput. Phys. **169**(2), 463–502 (2001). http://dx.doi.org/10.1006/jcph.2000.6636
15. Osher, S., Fedkiw, R.: Level-Set Methods and Dynamic Implicit Surfaces. Springer, New York (2002)
16. Osher, S., Chan, T., dong Liu, X., dong Liu, X.: Weighted essentially non-oscillatory schemes. J. Comput. Phys. **115**, 200–212 (1994)
17. 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)
18. Peng, D., Merriman, B., Osher, S., Zhao, H., Kang, M.: A PDE-based fast local level set method. J. Comput. Phys. **155**(2), 410–438 (1999). http://dx.doi.org/10.1006/jcph.1999.6345
19. Serra, J.: Image Analysis and Mathematical Morphology. Academic Press, New York (1982)

20. Sethian, J.A.: Level Set Methods: Evolving Interfaces in Geometry, Fluid Mechanics, Computer Vision, and Materials Science. Cambridge University Press, Cambridge (1996)
21. Sethian, J.A.: Level Set Methods and Fast Marching Methods. Cambridge University Press, Cambridge (1999)
22. Shu, C.W., Osher, S.: Efficient implementation of essentially non-oscillatory shock-capturing schemes. J. Comput. Phys. **77**(2), 439–471 (1988). http://dx.doi.org/10.1016/0021-9991(88)90177-5
23. Strain, J.: Tree methods for moving interfaces. J. Comput. Phys. **151**(2), 616–648 (1999). http://dx.doi.org/10.1006/jcph.1999.6205
24. Tsai, R., Osher, S.: Level set methods and their applications in image science. Commun. Math. Sci. **1**(4), 623–656 (2003)
25. Whitaker, R.T.: A level-set approach to 3D reconstruction from range data. Int. J. Comput. Vis. **29**(3), 203–231 (1998). http://dx.doi.org/10.1023/A:1008036829907