J Sci Comput
DOI 10.1007/s10915-011-9488-0

Out-of-Core Computations of High-Resolution Level Sets by Means of Code Transformation

Brian B. Christensen · Michael B. Nielsen · Ken Museth

Received: 5 March 2010 / Revised: 28 March 2011 / Accepted: 8 April 2011

Abstract We propose a storage efficient, fast and parallelizable out-of-core framework for streaming computations of high resolution level sets. The fundamental techniques are *skewing* and *tiling* transformations of *streamed* level set computations which allow for the combination of interface propagation, re-normalization and narrow-band rebuild into a single pass over the data stored on disk. When combined with a new data layout on disk, this improves the overall performance when compared to previous streaming level set frameworks that require multiple passes over the data for each time-step. As a result, streaming level set computations are now CPU bound and consequently the overall performance is unaffected by disk latency and bandwidth limitations. We demonstrate this with several benchmark tests that show sustained out-of-core throughputs close to that of in-core level set simulations.

Keywords Level set methods · Out-of-core · Streaming · Code transformation · Loop skewing · Loop tiling · Parallelism

B.B. Christensen (✉) · M.B. Nielsen · K. Museth
Aarhus University, Aarhus, Denmark
e-mail: bbc@cs.au.dk

M.B. Nielsen
e-mail: nielsenmb@gmail.com

K. Museth
e-mail: museth@acm.org

B.B. Christensen
Alexandra Institute, Aarhus, Denmark

M.B. Nielsen
Weta Digital, Miramar, Wellington, New Zealand

K. Museth
DreamWorks Animation, Glendale, CA, USA

Published online: 30 April 2011

1 Introduction

While the idea of using implicit functions for interface capturing can be dated back as far as [9, 10], the level set method and the underlying numerical schemes were first proposed in [46]. Since then, it has been applied to a wide range of interface capturing problems in scientific computing and related fields. Examples hereof include the simulation of multi-phase flows [53] such as bubbles and drops, solidification [14], Willmore flow [8], partial differential equations and variational problems on manifolds [3], geometric optics [45] as well as fluid animation [12] and geometric modeling in computer graphics [35].

The propagation of a time-dependent level set interface is given by partial differential equations e.g. of the Hamilton-Jacobi type. In most cases the scalar function is sampled on a regular Eulerian grid, although recent work has also employed fully Lagrangian representations [16]. In order to adequately capture interface details and obtain sufficient numerical accuracy, a combination of high order discretization schemes and/or high resolution Eulerian grids is often required.

In cases where only a single level set is of interest (e.g. the zero-crossing of the interface) computations can be restricted to a narrow band of grid points surrounding the interface [2, 6, 47, 61]. More recent work combines the idea of restricting the computations to a narrow band with sparse data structures in order to reduce storage requirements and enable interfaces to be sampled on higher resolution grids. These narrow-band data structures include blocked grids [5, 31, 37, 40], dynamic tubular grids (DT-Grid) [42, 43] and hierarchically run-length encoded grids [17]. Several authors have developed adaptive methods that do not restrict computations to a narrow band but instead refine the computational grid, typically closer to the interface [30, 38, 39, 49, 55].

1.1 Problem Statement

Although these level set data structures enable computations on high resolution grids, they are all limited by the available main memory. Despite the fact that modern 64-bit operating systems allow for a virtual address space of 16 exabytes, RAM modules remain a relatively expensive commodity. In comparison, disk storage is two to three orders of magnitude cheaper per byte, consumes about two orders of magnitude less power per byte and offers a capacity that is typically several orders of magnitude larger [29]. Hence algorithms capable of utilizing disk space, often referred to as *out-of-core* or *external memory* algorithms, have the potential of higher resolution simulations at lower costs. However, disk storage has much higher latency (referencing a single data item is four to five orders of magnitude slower than main memory access), and the development and study of efficient external memory algorithms has emerged into a field of its own [57, 60]. Note that the ongoing development of solid-state drive technology offers promising speed improvements for random access to external memory in the future.

Nielsen et al. [44] proposed an out-of-core framework for narrow band level set simulations based on the DT-Grid data structure, and this paper improves on that work. The main contributions of the out-of-core framework presented in [44] are prefetching and page-replacement algorithms designed for stencil based level set computations. Whilst allowing for grid resolutions only limited by the available disk storage, that method remains IO limited, and the throughput (measured in computed grid points per second) drops to 42% of in-core simulation throughput for some numerical schemes. One of the main reasons for the IO limitation in [44] is the fact that each step in the level set simulation requires the data to be streamed to and from disk multiple times. A typical time-step actually requires between

5 to 10 passes over the data. In contrast, the method proposed in this paper requires data to be streamed only once per time-step. In fact, for simulations with certain properties, data is only required to be streamed once for a number of subsequent time-steps, hence reducing bandwidth usage further. As a result, our new out-of-core algorithms are CPU limited as opposed to IO limited and exhibit a sustained, i.e. resolution independent, throughput of 77–92% (depending on the numerical scheme) of the throughput obtainable by internal memory simulations.[1]

Our general approach is to leverage on established theory from the area of compiler algorithms which performs *code transformations*, such as skewing (i.e. shearing the iteration space by a linear transform) and tiling (i.e. partitioning). These code transformations try to optimize cache *locality*, i.e. minimize the number of times a given data element is loaded into the cache from main memory. In particular, we employ the mathematical model of reuse and locality developed by Wolf and Lam [62]. Applying code transformations to out-of-core as opposed to in-core level set simulations poses unique challenges since our goal is to minimize the number of times a given data element is transferred from disk to main memory. The ratio of disk to main memory latency is much higher than the corresponding ratio of main memory to cache latency. Consequently a data layout that works well in-core may need to be redesigned for an out-of-core application, although they may have the same asymptotical IO complexity.

1.2 Contributions

In this paper we prove and demonstrate by implementation that the finite difference (FD) schemes used for level set simulations, HJ ENO [15], HJ WENO [19, 21, 33], BFECC [7] and TVD RK [52], have data reuse both temporally and spatially. However, locality is not directly implied. To improve locality we derive code transformations based on well-established skewing and tiling transformations and prove that these transformations maximize locality both spatially and temporally in the model of Wolf and Lam [62]. In particular, tiling alone is not sufficient to optimize locality for the FD stencil based level set simulations. Code transformations are applied to all steps in the narrow band level set computation, including propagation/advection, re-distancing and narrow band rebuild. In this way only a single pass over the data is required for each time-step or sequence of time-steps. Additionally we propose a tiled version of the Fast Iterative Method [22] which enables fast out-of-core solution of the Eikonal re-distancing equation $|\nabla\phi| = 1$ for narrow band level sets. To reduce memory requirements during simulation, we propose an in-core storage mapping for the intermediate values associated with the skewing transformation that is linear in the number of intermediate values. Furthermore, we also propose a linear out-of-core storage mapping associated with the tiling transformations that partitions the narrow band into tiles and boundary grids and facilitates computation on each tile independently and in parallel. Our framework is optimal with respect to streaming to and from disk in both the IO model [1] and the Cache Oblivious model [13], as only sequential stencil access is required. The sequential access enables utilization of the DT-Grid in combination with the page-replacement and prefetching algorithms from [44]. Finally, the resolution of the computational grids utilized in our method is limited only by the amount of disk space available.

[1] Our out-of-core technique introduces a computational overhead due to an increase in implementation complexity compared to regular in-core simulations. This explains why the efficiency is still below 100% even though the method is not IO limited.

To illustrate the feasibility of our new out-of-core framework we document its performance with benchmarks of several different types of level set simulations and numerical schemes. Additionally, we have used our framework for several applications of the level set method on high resolution computational grids. This includes advection in a divergence free velocity field and surface smoothing by mean-curvature flow of a model with an effective grid resolution of $17149 \times 9987 \times 5734$. In order to further improve simulation times, we also demonstrate a multi-core implementation of our out-of-core framework.

The remainder of this article is organized as follows: Sect. 2 first summarizes related work. In Sect. 3 we provide an overview of our out-of-core framework and its individual components, and introduce the terminology. Next we describe our proposed skewing and tiling transformations and the associated storage mapping schemes. Detailed descriptions of the transformations along with proofs of their locality properties are provided in the Appendix. Section 4 then proceeds to present benchmark results. Finally, Sect. 5 demonstrates some applications of our framework and Sect. 6 concludes and outlines directions for future work.

2 Related Work

In this section we review related work in the areas of out-of-core algorithms, simulation and loop transformation theory and algorithms. Out-of-core, or streaming, algorithms are applicable in all areas of computational science and scientific computing that involve massive data sets not feasible for storage in main memory due to hardware and cost limitations. These application domains include image repositories, digital libraries, relational and spatial databases, computational geometry, simulation, linear algebra and computer graphics. For a general survey of external memory algorithms see [60], and for a specific survey in the area of linear algebra and simulation we refer to [57].

2.1 Out-of-Core Simulations

Despite its potential for large scale simulation, we are only aware of a surprisingly small body of previous work directly related to out-of-core simulations. Pioneering work was done by Salmon and Warren [56] for N-body simulation in astrophysics. Their work was based on tree data structures and applied reordered traversals and a Least-Recently-Used page-replacement policy for efficiency. Bibireata et al. [4] use loop fusion and tiling (see below) to perform out-of-core tensor contractions for simulations of electron structures. Trac and Pen [18] proposed an out-of-core algorithm for Eulerian grid based cosmological simulation. In their method, global information is computed on a low resolution grid that fits entirely in memory, whereas local information is computed on an out-of-core high resolution grid tiled into individual blocks that fit into memory. The individual blocks are loaded and simulated in parallel for a number of time-steps and then written back to disk. More recently Nielsen et al. [44] proposed a combined framework for compression and out-of-core simulation of Eulerian grid based level sets and fluids based on the DT-Grid data structure [42] (to be discussed in more detail in Sect. 2.3). A fundamental property of their work is that existing level set and fluid simulation code does not have to be re-written. More specifically they introduce additional software layers that contain modules for storing and retrieving data on disk as well as compressing and decompressing data. Hence the focus of the article is on developing prediction schemes for statistically based compression as well as prefetching and replacement strategies for stencil based access. However, a consequence of this property is that the method remains IO limited, as the Eulerian grids need to be streamed through memory several times for each step of the simulation.

2.2 Loop Transformation

In the field of compiler algorithms and cache locality a lot of attention has been devoted to the so-called "loop transformation theory"—see [26] for a comprehensive overview. In particular Wolf and Lam [62] propose a theory of reuse and locality as well as an automatic algorithm for improving the data locality of loop nests by applying a sequence of loop transformations. The loop nest is analyzed for reuse and the resulting loop transformation is found as the maximum of an objective function measuring data locality of equivalence classes of localized iteration spaces. This theory formed the basis for the optimized loop transformations suggested by [36].

Wonnacott [63] introduced a particular type of transformation denoted time skewing and an associated storage scheme that takes advantage of the available cache memory. The transformation results in locality both in the spatial and temporal dimensions of the iteration space and is proposed for in-core time-step stencil computations. Computations are active only on a wavefront of grid points—the *wavefront of execution*—that can be fitted into the cache via the proposed storage scheme. Since this wavefront is skewed with respect to time in all spatial dimensions it is not well suited for implementation on a sparse narrow band data structure such as the DT-Grid. This is due to the fact that the skewed wavefront iteration order does not correspond to sequential access into the underlying data structure. In contrast the loop transformations we propose in this article can be implemented as sequential access which is faster than random access for this type of data structure. Time skewing along with various other optimization approaches for stencil computations, both cache aware and cache oblivious, were also studied by Kamil et al. [25].

Kandemir et al. [24] present a unified optimization framework that targets perfectly nested loops of computations running out-of-core and in parallel. In particular their framework is intended for integration in a compiler and it optimizes for locality of data references, array file layout, parallelism and reduction of communication overhead simultaneously. Their method considers only tiling for improving data locality and constructing file-layouts, which is too restrictive in order to obtain temporal locality for stencil-based level set computations. More recently, Kandemir et al. proposed an I/O-Conscious Tiling Strategy for Disk-Resident Data Sets [23]. Their method focuses on adapting traditional tiling algorithms for scientific computation loop nests to out-of-core computations in order to obtain higher IO performance. In particular, they show that both loop and data transformations are often required to achieve this goal. Again only tiling transformations are considered, and the class of algorithms investigated does not include stencil-based computations.

Song and Li [51] present a scheme which optimizes cache locality for a certain class of nontrivial imperfectly-nested loops. They propose a number of loop transformations to enable tiling. Specifically, they provide a compiler algorithm for optimizing skewing of the spatial dimensions subject to dependencies, automatically selecting the optimal tile size, and deducing an efficient storage scheme through array duplication. Their method is, however, limited to computations which only depend on references to values from the same or the previous iteration of the time-step loop. When using higher order temporal schemes such as TVD RK, our computations do not adhere to this limitation.

2.3 Dynamic Turbular Grid

Similar to the out-of-core simulation framework in [44], the improved framework proposed in this paper is based on the compact data structure, DT-Grid, introduced in [42]. For the sake of completion we shall briefly summarize its most important characteristics, but refer

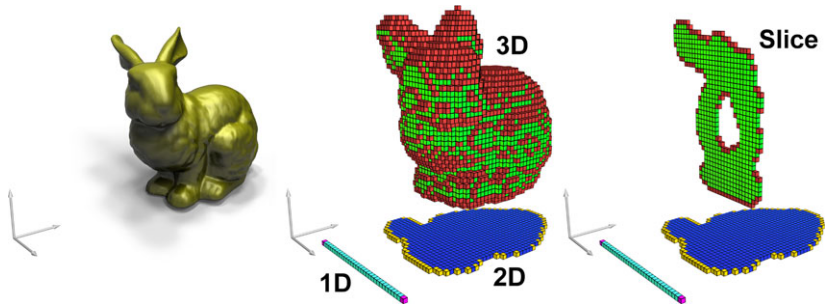


Fig. 1 Illustration of a narrow-band level set of a Stanford Bunny in a DT-Grid, see text

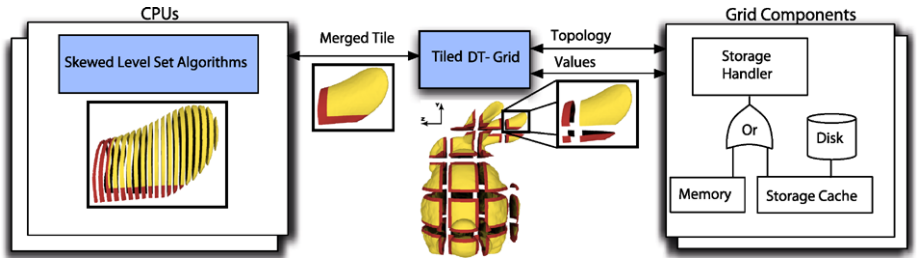


Fig. 2 (Color online) This figure gives an overview of our out-of-core level set framework that applies skewing and tiling transformations. *Rightmost*: Shows the components handling streaming to and from memory and/or disk, including prefetching and page-replacement [44]. *Middle*: Illustrates the central data structure, the *Tiled DT-Grid*, that implements our tiled storage mapping. The level set inside each tile (*yellow*) and the level sets at each tile boundary (*red*) are stored *separately* as narrow bands in DT-Grid data structures (*middle-right*) and continuously merged into a *single* narrow band during simulation (*middle-left*). *Leftmost*: The skewed level set algorithms process one slice of a narrow band tile at a time

the reader to the original paper for full details. Figure 1 illustrates a narrow-band level set of the "Stanford bunny" represented in a DT-Grid. It essentially works by separately encoding the signed distance values of all the narrow-band voxels (green and red) stored in lexicographic order, as well as the topology of all the boundary voxels (red, yellow and cyan) obtained by progressive projection onto the axes. This effectively means the z-coordinate is only stored for the red voxels in Fig. 1, the y-coordinates for the yellow voxels and the x-coordinate for the (two) magenta voxels. The cyan voxels between the two magenta voxels in the 1D projection encode pointers into the y-columns in the 2D projection. In turn the blue voxels enclosed by the yellow voxels in the 2D projection encode pointers to the z-columns bounded by the red voxels (see slice of z-columns to the right). Overall this leads to a compact volumetric data structure that offers constant-time sequential (i.e. lexicographic) data access and logarithmic random-access.

3 Skewing and Tiling Level Set Computations and Data Structures

Figure 2 provides an overview of our out-of-core level set framework, where the components corresponding to our contributions are highlighted in blue. Central to this framework is a tiled DT-Grid data structure shown in the middle of Fig. 2 and in Fig. 3. This data structure partitions a narrow band level set into a number of axis aligned tiles, storing only grid

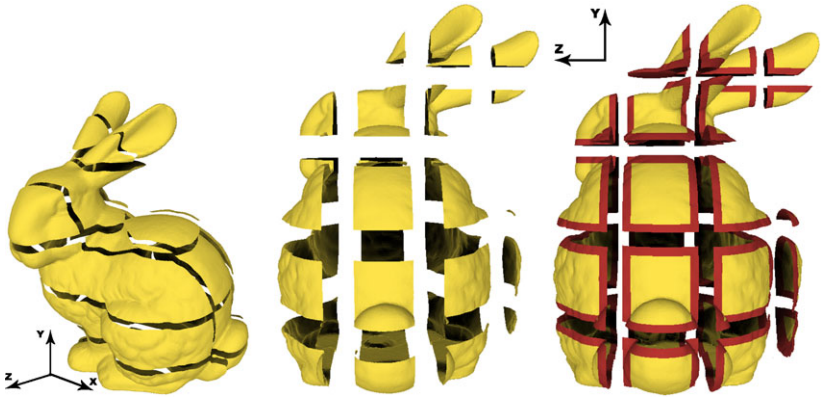


Fig. 3 (Color online) This figure illustrates how a level set surface of the Stanford Bunny is divided into axis aligned tiles. To maximize locality it is only necessary to tile along the y- and z-axes. The boundary grids of each tile are illustrated in *red* in the *rightmost* image

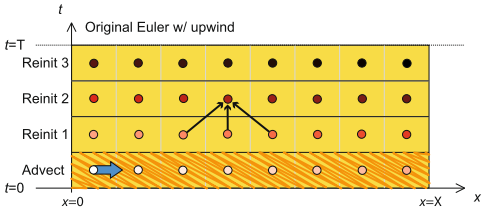
points that are part of the narrow band inside each tile (shown in yellow). Boundary grids (shown in red) are extracted from the narrow band on the boundary of each tile and used by the skewed level set computations on adjacent tiles. This effectively reduces bandwidth requirements since level set computations on a single tile only access the tile itself plus a number of small boundary grids, as opposed to streaming all the adjacent tiles. The size of the boundary grids is a function of the size of the stencil used in the FD computations as well as the number of level set steps performed on each tile. Generally the width of a boundary grid is small compared to the size of a tile. Hence, the overhead associated with the boundary grids is a fraction of the total storage requirements and computation time. As we show in the Appendix, given an N-dimensional grid, it is only necessary to tile in $N-1$ directions in order to maximize locality in both the temporal and spatial dimensions. Hence we always leave the x-direction untiled as shown in Fig. 3. Note that the choice of untiled direction is arbitrary, see the discussion in Sect. 3.2.2. Generally we tile in as few spatial dimensions as possible to minimize the computational overhead and at most $N-1$ dimensions as noted above. For example, if the required $(N-1)$-dimensional slices of the N-dimensional grid fit in memory, we do not tile the grid. The level set surfaces in each tile and boundary are stored separately as narrow bands in DT-Grid data structures using the out-of-core framework introduced by Nielsen et al. [44]. This is illustrated by the separated tile and boundaries beneath the rightmost arrows in the center of Fig. 2. The topology and values of each DT-Grid are stored independently as indicated by the layered boxes in the rightmost part of Fig. 2. Storing a particular component (values or topology) is managed by a *Storage Handler* which streams data either to memory or disk. In the case of streaming to disk, pre-fetching and page-replacement algorithms designed for stencil-based access are implemented by a *Storage Cache* [44]. Note that the level sets stored in tiles and boundaries may not represent closed surfaces. As long as the level set surface is intersected by convex tiles (e.g. axis aligned boxes), the DT-Grid data structure supports this [41].

Skewed level set computations can be performed on each tile independently and hence multiple computational threads can process separate tiles in parallel as indicated by the layered boxes in the leftmost part of Fig. 2. All computations take place on a single and partially in-core DT-Grid data structure storing only the active $(N-1)$-dimensional slices. This ensures that the in-core level set computations are cache efficient [41]. The in-core

DT-Grid data structure is generated on the fly by merging the grid points from the tile and the boundary grids generated from adjacent tiles. This is facilitated by the lexicographic storage order of the grid points utilized by the DT-Grid. The merging process is illustrated by the concatenated tile and boundary grids beneath the leftmost arrow in the center of Fig. 2. The level set computations are skewed in the spatial dimensions with respect to both the level of computation i.e. propagation, re-distancing and narrow band rebuild, as well as with respect to time. Hence all levels of computation, possibly at several time-steps, are performed simultaneously on a tile, but in such a way that data is streamed from disk exactly once, and such that data dependencies are not violated. As indicated in the leftmost part of Fig. 2, computations are performed on $(N-1)$-dimensional slices of an N-dimensional tile.

3.1 Skewing

A time-step of the overall level set iteration—a *level set step*—typically consists of an advection or propagation, a re-initialization and a narrow band rebuild *step*. In each of these level set steps a stencil is iterated over the spatial domain of the level set function to perform computations. An advection for example solves the hyperbolic level set equation $\frac{\partial \phi}{\partial t} - \vec{V} \cdot \nabla \phi = 0$, where ϕ is the level set function and $\vec{V}$ is a velocity field. The advection step is followed by a re-initialization of the level set function to a signed distance function which involves solving the PDE $|\nabla \phi| = 1$. The signed distance function representation has several advantages for numerical computations [47]. Finally the narrow band rebuild involves including grid points into the narrow band that move into the vicinity of the interface, as well as discarding grid points from the narrow band that move out of the vicinity of the interface. Each of these steps consists of one or several *sub-steps*, which each require one pass over the data if skewing is not applied. For example, advection with third order TVD Runge-Kutta time-integration consists of five sub-steps. In the context of code transformation theory, this situation corresponds to an imperfect loop nest since there are several loops (one for each sub-step) at the innermost nesting depth inside an outer time loop. We can convert this to a perfect loop nest by introducing a fictitious time variable and use it to distinguish which sub-step to perform in the loop body. Thus, one sub-step counts as one fictitious time-step. In the leftmost side of Fig. 4 the front of active computations, or *wavefront of execution*, progressing through a level set step is illustrated by the sketched orange row. The front is moving upwards, and the thick blue arrow indicates the progress of computations or iterations inside the front itself. To determine where the intermediate results of this execution must be stored, one employs a *storage mapping* which maps each computation on the wavefront to the address where it should store the value it produces [63]. This mapping results in a *wavefront of temporaries* which determines how much memory is




```
1: for t ← 0, T do
2:     for x ← 0, X do
3:         A[t+1,x] ← step(A[t,x],A[t,x−1],A[t,x+1])
4:     end for
5: end for
```

Fig. 4 (Color online) A number of steps on a 1D level set using the traversal outlined in the pseudocode. The sketched *orange row* indicates the wavefront of execution travelling parallel to the t-axis. The *thin black arrows* indicate the dependencies of one computation

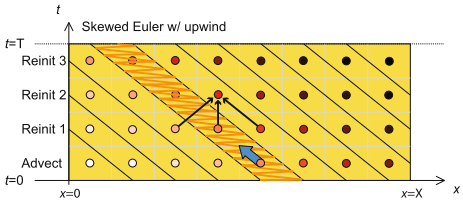
required. Recall, the goal is to reduce the number of passes over the level set data (thereby minimizing data traffic) to one for a sequence of M steps. The motivation is that we want to eliminate the IO limitation of the previous out-of-core framework [44]. Hence, we want to perform as many steps using in-core temporaries as made possible by the amount of main memory. Therefore we need transformations of the code to make the wavefront of execution independent of the grid dimensions, such that the wavefront of temporaries fits in memory.

We perform the following analyses and transformations on full grids in one dimension for simplicity, and then generalize to N-dimensional grids in the end. In the Appendix, we provide proofs of the validity of these transformations. In practice, the algorithms are implemented on sparse DT-Grids which provide constant time sequential access to neighboring grid points within a stencil.

3.1.1 Transforming the Iteration Space

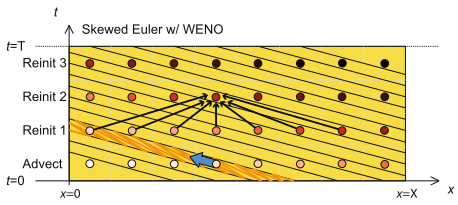
Figure 4 shows the pseudocode of a number of sub-steps on a one-dimensional level set using simple first order upwinding in space and first order Forward Euler integration in time. The yellow box illustrates the iteration space of the nested loop. To simplify the following explanation, we assume that we have expanded the one-dimensional array which represents ϕ with a dimension containing the time axis, thus obtaining an array A of size $(T+1) \times X$, where T is the number of fictional time-steps and X is the size of the spatial domain. The traversal order of the iteration space is indicated by the coloring of the individual iterations, starting from white over red to black. In Fig. 4, the entire spatial domain is traversed in each time-step before moving on to the next. Not all traversals of the iteration space are valid, since a given computation $[t, x]$ has dependencies which limits the traversal possibilities. In this example, we employ a computational stencil with a width of three grid points, needed to implement the first order upwind scheme for advection and reinitialisation. As illustrated by the thin black arrows in Fig. 4 this means for example, that the result at iteration $[2, 3]$ cannot be computed before the results for iterations $[1, 2]$, $[1, 3]$ and $[1, 4]$ are known. See the Appendix for a rigorous definition and analysis of dependencies. One consequence of the dependencies is that we cannot immediately interchange the t and x loops in the shown algorithm.

The algorithm cannot just iterate over the entire spatial domain for each fictional time-step since the references to A in previous steps will be evicted from memory before they can be reused due to the large number of intermediate computations. We transform the code to improve this by skewing the spatial dimension of the iteration space just enough to be able to interchange the loops. Specifically, we transform the loop bounds using the transformation $T_1 : [t, x] \rightarrow [t, x + t]$ and then apply T_1^{-1} to the array references. As explained in the Appendix, this skewing allows us to interchange the loops without violating the dependencies of the algorithm. Figure 5 shows the resulting traversal of the iteration space along




```
1: for x ← 0, X + T do
2:     for t ← 0, T do
3:         if max(t,x) < min(X + t, x + 1) then
4:             w ← x − t
5:             A[t + 1, w] ← step(A[t, w], A[t, w − 1], A[t, w + 1])
6:         end if
7:     end for
8: end for
```

Fig. 5 A number of steps on a 1D level set using the transformed traversal outlined in the pseudocode. Note, that the skewed traversal order is depicted in the original iteration space




```
1: for x ← 0, X + 3T do
2:     for t ← 0, T do
3:         if max(t,x) < min(X + 3t, x + 1) then
4:             w ← x − 3t
5:             A[t + 1, w] ← step(A[t,w], A[t,w − 3], A[t,w − 2], A[t,w − 1], A[t,w + 1], A[t,w + 2], A[t,w + 3])
6:         end if
7:     end for
8: end for
```

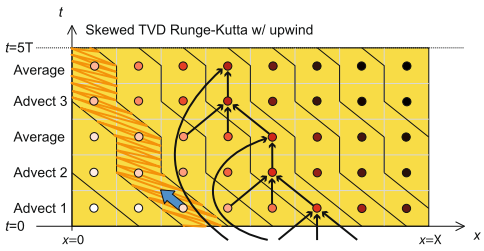
Fig. 6 A number of steps on a 1D level set using the transformed traversal outlined in the pseudocode

with the transformed pseudocode. Iterations with references to the same entries of A are much closer together using this traversal, thus increasing the locality. If T is of the same magnitude as X, we have of course not achieved locality, and in general, loop skewing, loop interchange, etc. must be combined with tiling transformations as explained in Sect. 3.2 [62, 63].

For computations with higher order spatial schemes such as the variable third to fifth order accurate HJ WENO scheme, we can perform similar transformations to improve locality. The HJ WENO scheme employs a stencil of seven grid points, and thus each iteration depends on as many as seven previous results. However, this merely means that we have to skew the spatial dimension even more. In particular, the transformation $T_2 : [t, x] \rightarrow [t, x + 3t]$ ensures that the loops can be interchanged. The resulting traversal order is shown in Fig. 6. Note how the "slope" of the skewed loops has changed to reflect the wider area of dependence of the stencil. Also note that the transformation used in this example would be perfectly "legal" in the previous example. The key observation is, that we seek the "legal" transformation that optimizes the slope, i.e. minimizes the *skew factor* of each loop as this provably minimizes memory reference costs [34]. In other words, we want to minimize the width of the wavefront of execution projected onto the x-axis.

The third order accurate TVD Runge-Kutta scheme for temporal discretization consists of five sub-steps in order to advance the solution one step forward in time [52]. More specifically it consists of two convex combinations of three Forward Euler time-steps. The separate sub-steps in the method have different dependencies, e.g. the Euler steps have dependencies corresponding to the stencil used in the first example, while the convex combination steps only depend on earlier results on the same spatial position. These differences could be ignored, and we could apply a legal skewing transformation like T_1, but that would result in a suboptimal slope and width of the wavefront of execution, since the averaging steps do not require skewing in x. An optimal transformation must take this into account, and for a first order upwind scheme, we propose $T_3 : [t, x] \rightarrow [t, x + 3\lfloor \frac{t}{5} \rfloor + \min(t \bmod 5, 1) + \lfloor \frac{t \bmod 5}{3} \rfloor)]$, where $\lfloor \rfloor$ denotes the floor function. Figure 7 shows the resulting traversal of the iteration space, when the skewing has been combined with a loop interchange. Note that t is now used as a fictitious time variable such that $\lfloor \frac{t}{5} \rfloor$ denotes the time-step and t mod 5 uniquely identifies one of the five assignment statements or sub-steps in the loop body. The described transformation goes beyond the framework of Wolf and Lam [62], and a careful analysis of the validity of the proposed transformation is provided in the Appendix.[2] Transformations and code for TVD Runge-Kutta and BFECC combined with HJ WENO as well as generalizations to more spatial dimensions are also presented in the Appendix.

[2]The analysis is performed on the algorithmically similar *Back and Forth Error Compensation and Correction* (BFECC) scheme, and the validity of the transformation for the TVD Runge-Kutta scheme is derived from that.




```
1: for x ← 0, X + 3⌊(5T−1)/5⌋ + 2 do
2:     for t ← 0, 5T do
3:         x_start ← 3⌊t/5⌋ + min(t mod 5, 1) + ⌊(t mod 5)/3⌋
4:         if max(x, x_start) < min(x + 1, X + x_start) then
5:             w ← max(x, x_start) − x_start
6:             if t mod 5 = 0 then
7:                 A[t + 1, w] ← step(A[t, w − 1], A[t, w], A[t, w + 1])
8:             else if t mod 5 = 1 then
9:                 A[t + 1, w] ← step(A[t, w − 1], A[t, w], A[t, w + 1])
10:            else if t mod 5 = 2 then
11:                A[t + 1, w] ← average(A[t, w], A[t − 2, w])
12:            else if t mod 5 = 3 then
13:                A[t + 1, w] ← step(A[t, w − 1], A[t, w], A[t, w + 1])
14:            else
15:                A[t + 1, w] ← average(A[t, w], A[t − 4, w])
16:            end if
17:        end if
18:    end for
19: end for
```

Fig. 7 An advection step on a 1D level set using the transformed traversal outlined in the pseudocode. Dependencies are shown for all the sub-steps of the advection step

3.1.2 Storage Mapping

Using the above transformations (combined with tiling) we achieve a wavefront of execution which permits locality. However, if each iteration writes to a separate entry in the expanded array A, the memory usage scales with the size of the level set grid. Therefore, we do not in practice store the entire temporal and spatial dimensions of A. Instead, each level set step streams an out-of-core grid as input and another as output while everything in between is stored in-core using a suitable storage mapping which maps each iteration to the address where it should store the value it produces [63]. The goal is to minimize memory usage, and at the very least ensure that it does not scale with the size of the level set grid. The storage mapping applied in the first Euler example above (Fig. 5) is the trivial $\{[t,x] \to A[t+1, x-t]\}$, which is not independent of the size of the level set grid. Observing that the computations only depend on the results of the previous step, one improvement would be to only store the latest two rows of A and use the mapping $\{[t,x] \to A[(t+1) \bmod 2, x-t]\}$. This storage mapping still writes to a number of memory locations which scales with X, and thus the required memory does not fit in-core even though the wavefront of temporaries does. To improve this, we propose a storage mapping which is skewed in the same manner as the iteration space has been skewed. This idea leads to a small *buffer of temporaries* which only holds the wavefront of temporaries [63].

While the storage mappings of the previous paragraph lend themselves to be described by a simple formula, the skewed storage mappings which we propose are not as easily expressed in this formalism. Therefore we shall use diagrams to illustrate the storage mappings. This also provides the necessary intuition for code implementation.

In the following we present the mappings for some of the most popular level set discretization schemes. Figure 8 shows a "snapshot" of the execution of the Forward Euler scheme with upwinding, and in particular all the computations on the wavefront of execution. Orange areas indicate results which are read in from and written to out-of-core grids (labeled OOC Grid), while the yellow areas represent results stored in the in-core buffer of temporaries (labeled Buffer). The greyed out area indicates results from iterations that are not needed anymore, i.e. that are no longer part of the wavefront and buffer of temporaries. Finally, the yellow areas with dotted outlines indicate that the result stored there can

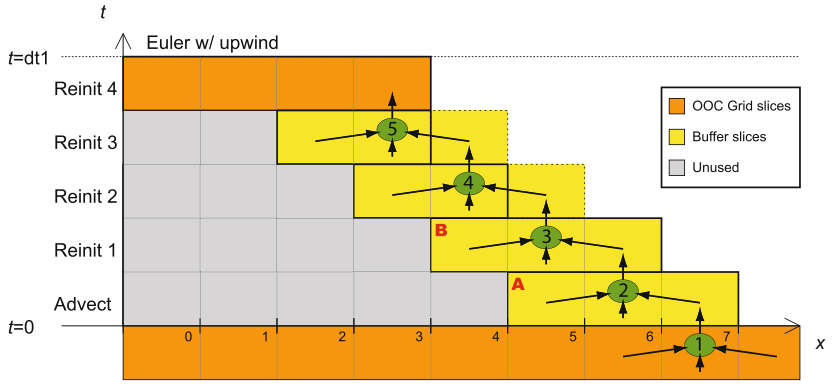


Fig. 8 (Color online) The storage requirements of a full level set step with advection and re-initialization steps using the Forward Euler scheme and upwinding. *Yellow areas* with *dotted outlines* indicate that a computation result replaces an entry in the buffer of temporaries corresponding to the same spatial position. The entry being replaced, which is not needed anymore by the computations on the wavefront of execution, is marked by a *bold red letter*

replace one of the other entries in the buffer of temporaries corresponding to the same spatial position. These replaceable entries are marked by bold red letters. The numbered green circles each correspond to a computation and the black arrows indicate the dependencies. Two buffer entries above each other are needed to effectively propagate the wavefront of computations through the iteration space. Suppose the first computation (i.e. the circle with the number one) is about to advect entry 7 in the input out-of-core grid. It needs to store the result in the buffer of temporaries and because it cannot overwrite entries needed by the second through fifth computations, it stores the result as indicated. The second computation can similarly not overwrite needed values and therefore its result is stored as indicated. It should be noted that in one dimension it is possible that the second computation could use the entry in the buffer of temporaries directly to the left of it to store its result. However, it is important to stress that this does not generalize to N-dimensional level sets on DT-Grids. When we perform this generalization, the entries in the buffer of temporaries are in fact $(N-1)$-dimensional slices of varying sizes, so the result from the second computation would not necessarily fit.

Returning to Fig. 8, we note that the third computation can write its result to the entry in the buffer of temporaries marked by the letter **A**, thus overwriting the entry which the second computation on the wavefront does not need anymore. Since this result is of the same size as the one overwritten, this does not pose a problem in N dimensions. Similarly, the fourth computation can write its result in the buffer entry marked **B**. This behavior generalizes to n advection/propagation and re-initialization steps, and the storage scheme works for computations of any stencil width w. The required size of the buffer of temporaries is given by $2r(n-1)+2$, where $r = (\lfloor \frac{w}{2} \rfloor)$ is the *effective radius* of the stencil.

Figure 9 shows a similar "snapshot" of the TVD Runge-Kutta scheme, again with first order upwinding. As can be seen from the grey and yellow areas with dotted outlines, we can immediately overwrite the result of the second computation (stored in the entry marked **A**) with the result of the first averaging computation (computation number 3), thus saving memory. Also, the fourth computation is able to write its result in the buffer entry marked **B**. Note that the storage pattern is slightly different for the second step, because the values written by the fifth computation cannot be overwritten as they are needed by the final averaging

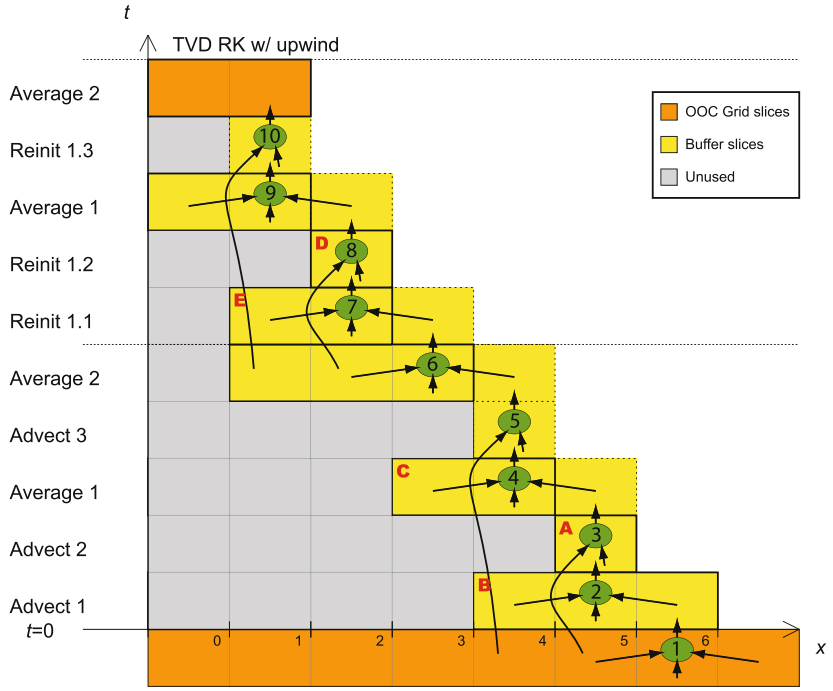


Fig. 9 The storage requirements of an advection and a re-initialization step with the TVD RK scheme and upwinding. Note that the storage requirements are greater for subsequent steps than for the first step due to the averaging computations

Table 1 The proposed memory requirements. r is the radius (or half the width rounded down) of the computational stencil employed, while M is the total number of steps in the level set step. Each entry in the buffer of temporaries is an $(N-1)$-dimensional slice

Temporal scheme	Buffer size
Forward Euler	$2r(M-1)+2$
TVD Runge-Kutta	$(7r+1)(M-1)+4r+2$
BFECC	$(6r+1)(M-1)+4r+2$

computation. The generalized formula for the size of the buffer of temporaries is given in Table 1 along with the one for BFECC, which can be derived in a similar way. The proposed mappings result in memory requirements which scale with the size of the wavefront of execution rather than the size of the grid.

As mentioned, each entry in the out-of-core grids and the buffer of temporaries is actually an entire $(N-1)$-dimensional slice of the narrow band. In the one-dimensional examples above, the values are therefore just scalars, which the computations can operate on directly. In higher dimensions, we perform computations using sequential access on entire slices at a time. Furthermore, when dealing with more spatial dimensions, we tile the space and utilize the boundary grids mentioned in the overview. The specific storage mappings proposed for doing this will be explained in Sect. 3.2.

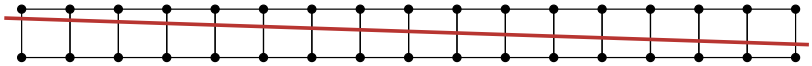


Fig. 10 (Color online) An illustration of a pathological case for our modified FIM. A level set intersecting a number of grid cells is shown in *red*. Even though the maximal distance computed is limited by the width of the narrow band, during computations, distance information may travel far on a micro-scale due to the discrete star-stencil. In this particular example the computation of the distance value at the node furthest from the level set (lower left node) is dependent on the computation of the distance values at all other nodes. Thus it is not theoretically possible simply to divide the domain into blocks, and compute distance information in each block separately

3.1.3 The Fast Iterative Method

In Sect. 3.1.1 we described how to solve the (pseudo time-dependent) re-initialization equation $\partial\phi/\partial t + S(\phi_0)(|\nabla\phi| - 1) = 0$ to steady state in our out-of-core framework. In this section we describe how to apply a tiling scheme to the recently proposed *Fast Iterative Method* (FIM) [22] for solving the eikonal equation $|\nabla\phi| = 1$. This enables an out-of-core implementation that requires data to be streamed to and from memory only once. Furthermore, it can be combined with the skewing framework in Sect. 3.1, hence requiring streaming to and from memory only once for all steps making up a level set iteration. The FIM has a number of properties which makes it well-suited for an out-of-core parallel implementation. First of all, it does not require a separate, heterogeneous data structure such as the heap required by the Fast Marching Method (FMM) [50, 58, 59]. Secondly, it can simultaneously update multiple grid points.

The FIM manages a list of active grid points which are iteratively updated until convergence. This *active list* is updated by adding and removing grid points based on a convergence measure. The main difference from the FMM is that grid points are updated independently and that the active list can move over grid points previously removed from the list and re-activate them as new information is propagated across the narrow band. This is in contrast to the FMM which maintains a heap of grid points sorted after their current distance to the zero-crossing and only removes them once they have the correct value. The consequence of the FIM's approach is that the grid points do not need to be updated in a strict order based on their distance to the zero-crossing as they can be reactivated.

To enable a streaming implementation, we modify the original algorithm by partitioning the level set slices into groups which are treated separately. In order to allow the correct propagation of distance information between the groups, a band of slices is shared between neighboring groups. The width of this band must be at least the same as the width of the narrow band. The motivation behind this is that, in the continuous case, the distance value at a certain point originates from points no further than 'distance' away from it. In our case the maximum distance computed equals the width of the narrow band. However, due to the seven-point star-stencils used for numerical computations, distance can, in the discretized case, travel infinitely on a sub-scale. This means that the distance value at a certain grid point may rely on the distance first being computed correctly at grid points further away than the width of the narrow band. A pathological case is shown in Fig. 10. However, as is evident from our numerical experiments in Table 2, the error introduced by limiting the width of the shared band of slices appears to be below the truncation error in practice. Figure 11 shows three slice groups where the raised, middle group is currently being processed by our algorithm. The orange slices indicate the region in which the active list can compute and propagate distance information. We will call these the *active slices*. Notice that the last few slices of the previous group remain active such that new information can correctly propagate

Table 2 Error norms of the various FIM implementations on two different surfaces. Our sliced FIM implementations are as exact as the original algorithm

Model	Reinitialization equation		FIM		Sliced FIM w/List		Sliced FIM w/Mask	
	$\lvert\cdot\rvert_\infty$	$\lvert\cdot\rvert_2$	$\lvert\cdot\rvert_\infty$	$\lvert\cdot\rvert_2$	$\lvert\cdot\rvert_\infty$	$\lvert\cdot\rvert_2$	$\lvert\cdot\rvert_\infty$	$\lvert\cdot\rvert_2$
	0.00452	0.000148	0.00697	0.000200	0.00697	0.000196	0.00697	0.000197
	0.0164	0.00251	0.0186	0.00203	0.0186	0.00200	0.0186	0.00200

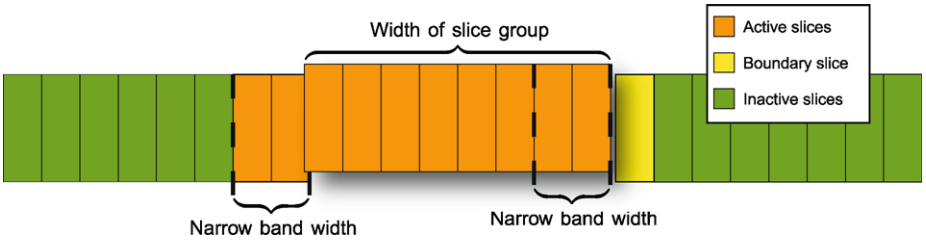


Fig. 11 (Color online) Our modified FIM processes the slices one group at a time. The active list can only propagate distance information within the *orange slices*. A band of slices from the previous group remain active to ensure correct backward propagation of information

back to them. The yellow slice of the next slice group acts as a *boundary slice* and stores grid points which should have been included in the current active list. They are then used as the initial active list when treating the next group of slices.

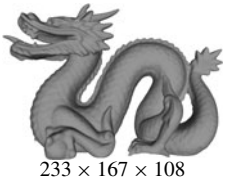
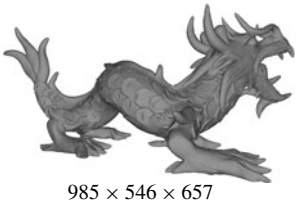
We have experimented with two implementations of the active list. The main difference lies in the access pattern of the grid points in the list. The first implementation simply stores the points in a list and when processing them sequentially, it uses random access into the DT-Grid which is logarithmic in the number of connected components of the DT-Grid [42]. The second implementation utilizes a bit mask to determine which grid points are in the list. It then sequentially scans through all grid points in the current group of the DT-Grid and updates the ones which are in the mask. This is done iteratively until convergence. We refer to these implementations as *Sliced FIM with List* and *Sliced FIM with Mask*, respectively.

Table 2 shows that the accuracy of both Sliced FIM implementations is as good as for regular FIM. For both these examples, we used a slice group width of $2\times$ the narrow band width. For the algorithm from Sect. 3.1, the initial ϕ was reset to $\pm\gamma$ away from the zero-crossing and the algorithm was run for 40 iterations using Forward Euler and first order upwind differencing for the temporal and spatial derivatives, respectively. Table 3 shows the running times for a few bigger examples. It is evident that the Sliced FIM with List algorithm outperforms Sliced FIM with Mask despite the slower access method.

3.1.4 Rebuild

To enable one or several complete level set steps to be implemented out-of-core in a single streaming pass, we must also consider how to adapt the narrow band rebuild algorithm of

Table 3 CPU times in seconds of the various FIM implementations. The examples were run with the same settings as in Table 2

Algorithm	 233 × 167 × 108	 985 × 546 × 657
Reinit. eq.	5.9	61.8
Sliced FIM w/List	4.4	49.4
Sliced FIM w/Mask	6.8	108.6

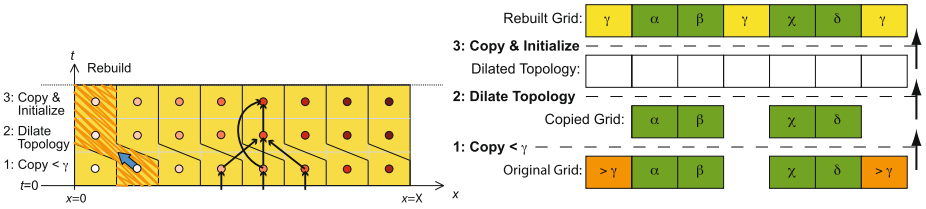


Fig. 12 (Color online) *Leftmost*: Illustrates the skewed iteration space of the rebuild process with the wavefront of execution sketched in *orange*. The *thin black arrows* indicate the dependencies. *Rightmost*: Illustrates the steps involved in the rebuild process of a 1D DT-Grid. To emphasize the relationship with the skewed iteration space (*left*), the figure should be read from the bottom and up. In this example the original grid consists of two separate connected components. Grid points with numerical value less than γ are shown in *green* and labeled α, β, χ and δ. Values larger than γ are shown in *orange* in the *bottom row* and new grid points are shown in *yellow* in the *top row*

the DT-Grid. Similar to the substeps of computation in the TVD RK algorithm in Fig. 7, the substeps of the rebuild algorithm can be skewed to maximize locality. We refer to the original DT-Grid paper for full details on the rebuild algorithm [42], and present here only a simplified version. In particular the rebuild algorithm consists of the following steps, where we assume ϕ to be a signed distance function, γ to be the Euclidean width of the narrow band, N to be the dimension, and H to be the width of the dilation measured in grid cells (the rebuild process is illustrated in the rightmost part of Fig. 12 with $N = 1$ and $H = 1$):

1. Copy grid points with $|\phi| < \gamma$ to an intermediate grid.
2. Dilate the topology of the intermediate grid with a stencil shaped as a hypercube of dimensions $(2H + 1)^N$. This may change the topology of the grid. Then allocate uninitialized storage for the values of the grid.
3. Copy values of grid points that exist in the intermediate grid to the final grid, and initialize new grid points to a numerical value of γ.

In the original paper each of these steps are completed before the next commences. However, as illustrated in the leftmost part of Fig. 12 the wavefront of execution can be made independent of the grid dimensions. In particular steps 1 and 3 require no skewing since a computation at these substeps depends only on the computation immediately below in the original iteration space. Since step 2, the dilation step, utilizes a hypercube-shaped stencil for dilation, a minimal skewing of H grid points is required in each spatial dimension

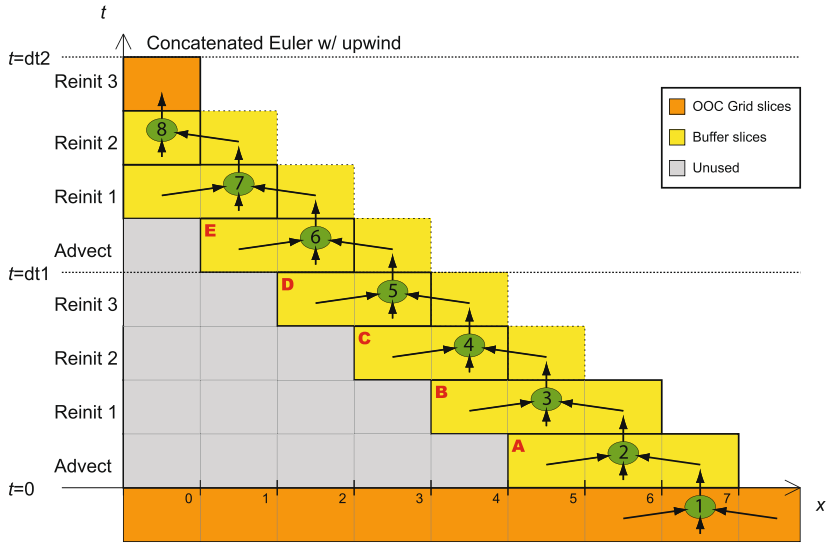


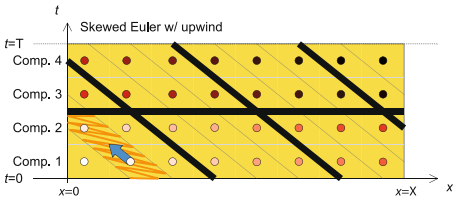
Fig. 13 Two concatenated Euler level set steps. Notice how all intermediate values are stored in-core while out-of-core grids are used only as input and for the final output. Also note that we have abstracted away the sub-steps of the rebuild step for simplicity

at this level of computation. When including a rebuild step on the wavefront of execution, where both the prior and the subsequent steps operate in the buffer of temporaries as opposed to out-of-core, the number of entries in the buffer of temporaries is increased by $(2H+1)+(2\lfloor\frac{w}{2}\rfloor+1)$ for a simulation employing forward Euler. In particular step 1 requires $2H+1$ additional entries corresponding to the width of the dilation stencil, and steps 2 and 3 require $2\lfloor\frac{w}{2}\rfloor+1$ additional entries corresponding the width of the propagation stencil. Similar arguments hold for a simulation employing TVD RK or BFECC time stepping thereby obtaining $(2H+1)+(3\lfloor\frac{w}{2}\rfloor+1)$ additional entries. If the result of the rebuild step is placed out-of-core, the number of entries is only increased by $(2H+1)$, independent of the time stepping approach, since there is no propagation stencil.

3.1.5 Concatenating Multiple Level Set Steps

In the previous sections, we have only dealt with performing one level set step during a single pass over the level set. If the amount of main memory allows it, several level set steps can be concatenated into one pass. Figure 13 shows a concatenation of two level set steps with advection and re-initialization steps (for simplicity the rebuild steps have been omitted). Note how the intermediate results of the first level set step are kept in memory as opposed to in Fig. 8. This lowers the disk traffic and lessens the load on the bandwidth to the disk, which facilitates using slower disks and/or faster processors with no performance penalty as well as running several simulations or computing on several tiles at the same time.

This approach requires that a time-step size is chosen for some level set steps in the simulation *before* the previous steps have been completed, which is not always possible. In most cases, however, one can perform conservative estimates of or exactly determine the allowed time-step size. This is the case for e.g. mean curvature flows where the latter is possible and for analytical flows where only the former is an option. One notable exception is fluid simulation flows where a guaranteed stable estimate can not be made.




```
1: for t2 ← 0, T, BT do
2:     for x2 ← 0, X + T, BX do
3:         for t ← t2, min(T, t2 + BT) do
4:             for x ← max(t, x2), min(X + t, x2 + BX) do
5:                 A[t + 1, x − t] ← step(A[t, x − t], A[t, x − 1 − t], A[t, x + 1 − t])
6:             end for
7:         end for
8:     end for
9: end for
```

Fig. 14 (Color online) *Leftmost*: Shows the geometric outline (*thick black lines*) of the *skewed* tiles resulting from tiling the iteration space. In this case a tiling of $B_T = 2$ was used in the temporal dimension and a tiling of $B_X = 3$ was used in the spatial dimension. The wavefront of execution which is independent of the grid dimensions is sketched in *orange* in the lower left tile. *Rightmost*: Shows the code corresponding to the iteration space traversal on the *left*

Even when performing only a single level set step per pass, one does not want to perform a separate pass over the data to determine the time-step size for the next step. If the velocity field for the next step is available, one can combine the rebuild algorithm with an evaluation of the field for all grid points in the new narrow band and thus determine the allowed time-step size.

3.2 Tiling

In this section we first describe the tiling of the skewed iteration space and next explain the storage mapping that tiles the actual data layout.

3.2.1 Tiling the Iteration Space

Whereas skewing of the iteration space is required to facilitate permutation of the iteration directions, tiling is in general required to ensure locality of references to data. This is caused by the fact that along a direction in the iteration space we may reference more data than can fit into memory. Hence, even though later computations reuse data items, omitting skewing will in the worst case have to load them into memory every time they are referenced. Tiling, or blocking, is conceptually simple and illustrated in 1D for a first order spatial method combined with Forward Euler in Fig. 14. The iteration along each direction of the iteration space is divided into tiles of equal size by splitting the corresponding loop into two loops, where the loop that steps over the tiles is called the controlling loop. As shown in the rightmost part of Fig. 14 tiling is combined with loop interchange to place the controlling loops as the outermost loops in order to ensure locality. More concretely, if a given level set simulation takes T substeps then data has to be streamed to and from memory T times without skewing and tiling. By introducing skewing and a tile size of B_T in the temporal direction, the number of times data has to be streamed is reduced to T/B_T. In order for this to work, the size of the wavefront of temporaries described in Sect. 3.1 must fit into memory. This poses restrictions on how large B_T can be and additionally a tiling in the spatial dimensions may be required as well. In fact, for a simulation on a N-dimensional grid, tiling is required in at most $N - 1$ spatial dimensions in order to make the wavefront of execution independent of the grid dimensions. Consider the 1D example in Fig. 14. In this case it is not necessary to tile in the x-direction to ensure locality because the dimensions of the wavefront of execution is independent of the spatial direction in which it travels (in this case the x-direction), see Appendix .1 for a more elaborate theoretical justification of this. For the 3D simulations considered in this article, we at most tile in the y- and z-directions and always leave the

x-direction untiled. However, we note that in practice the optimal choice of the direction to leave untiled depends on the level set geometry. Since the level set evolves dynamically over time, the untiled direction should be able to change over time as well. Furthermore the orientation of the level set on the computational grid will also impact the choice of direction. We leave these issues as future work.

3.2.2 Tiled Storage Mapping

Skewing and tiling the iteration space optimizes locality for level set computations. However, these transformations are not sufficient to yield a fast practical implementation. Recall that in order to obtain computational and storage efficiency we implement our proposed framework on the DT-Grid data structure. Accessing data in tiles on a DT-Grid will result in a large number of non-sequential access operations that have logarithmic time complexities. In an out-of-core implementation the non-sequential access operation furthermore results in a logarithmic IO complexity, as well as disk search operations which are expensive operations relative to sequential disk access. This suggests that in order to obtain feasible run times a transformation of the layout of data on secondary storage is required in addition to iteration space tiling. In particular, a tiling of the narrow band is required. Figure 15(a) shows a 1D untiled grid with three connected components, and Fig. 15(b) shows a tiled version of the same grid, where each tile is stored as a separate grid, and the tiles are indicated by the thick vertical lines. The tile boundaries of the grid correspond to the tile boundaries of the iteration space. Inside each tile, computations are skewed with respect to time and temporary storage allocated as described in Sect. 3.1 and illustrated in Fig. 15(c). In order to

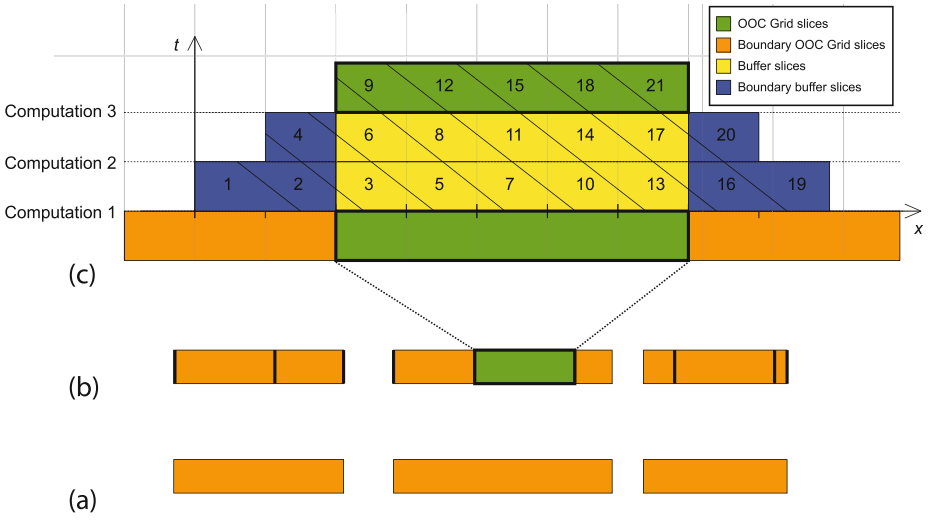


Fig. 15 (Color online) (**a**) Untiled 1D grid consisting of three connected components. (**b**) The grid in (**a**) tiled using a fixed tile size. Note that due to the connected components the part of the narrow band inside each tile may vary in size. (**c**) The enlargement of a single tile, shown in *green* (labeled OOC Grid). The boundary data from neighboring tiles required for three complete computations on the tile is shown in *orange* (labeled Boundary OOC Grid). The values allocated as temporaries are shown in *yellow* and *blue* (labeled Buffer and Boundary buffer, respectively). Note that only the wavefront of temporaries are stored in-core at any time during the simulation. The output from three complete computations on the tile is shown in *green* at the top. The wavefronts are indicated by the *skewed lines*, and the exact order of computations is indicated by the numbering

complete several iterations inside each tile, boundary data from neighboring tiles is required which is also illustrated in Fig. 15(c). A straightforward way to do this would be to also stream parts of neighboring tiles through memory when performing computations on a specific tile. However, this is infeasible due to the aforementioned penalties of random access, and further impeded by the fact that large page sizes are used when transferring data from disk to memory. Thus we may in the worst case end up streaming all of the neighboring tiles through memory when in fact only a relatively thin band of boundary data is needed. The situation becomes more intractable for higher dimensional grids, e.g. in 3D we may end up streaming the grid to and from memory nine times if tiling in two spatial dimensions. The solution to this is to store separate boundary grids that contain only the boundary data needed to complete the computations inside each tile. Note that the width of the boundary grids is equal to the width of the wavefront of execution, and proportional to the number of fictitious time-steps. In particular, a wavefront including M Forward Euler fictitious time-steps has a width of $(M-1)\lfloor\frac{w}{2}\rfloor$, whereas a wavefront including M TVD RK or BFECC fictitious time-steps has a width of $(M-1)(w-1)$, where w is the width of the stencil. Each rebuild step included in the wavefront adds one to the width. In all of our simulations the boundary grids are very small compared to the size of the tiles, as B_X is orders of magnitude larger than B_T, where B_X and B_T are the tile sizes in space and time respectively. The reason for this is that B_T in most cases only includes a single level set step. If several level set steps are concatenated as discussed in Sect. 3.1.5, there is a tradeoff between lowering the requirements on IO bandwidth and the increased overhead arising from boundary grids.

We employ a data structure dubbed the *Tiled DT-Grid*. An example in 3D is depicted in Fig. 2. The tiled DT-Grid consists of three components: A *coarse grid*, several *tile grids* and several *boundary grids*. Each cell in the coarse grid corresponds to a tile in the spatial dimensions. Since we always leave the x-direction untiled, the spatial extent of such a cell will be infinite in at least one direction. A cell in the coarse grid essentially holds pointers to a single tile grid and to a boundary grid for each boundary element (in 3D either an edge or a face) along which two cells are adjacent. Hence for a 3D grid the number of boundary grids for each cell will vary between zero and eight; zero if no tiling is applied and eight if tiling in the y- and z-directions since there will be eight neighboring grids; one along each of the four faces and one along each of the four edges of the tile (see Fig. 3). Both the coarse grid, the tile grids and the boundary grids can be stored separately as out-of-core DT-Grids. Since the intersection of the narrow band with the boundaries of a tile may result in a level set that is not closed, special algorithmic care has to be taken for a DT-Grid implementation, and how to handle iteration with a stencil and narrow band rebuild is described in [41].

Interleaved with computations on a specific tile, the grid points of the tile itself as well as the grid points of adjacent boundary grids are merged into a single DT-Grid when streamed into memory. This merging is performed on demand one slice of the computational grid at a time as requested by the computation. Due to the lexicographic storage order of the DT-Grids used for both tile and boundary grids, the merging can be performed in time linear in the number of grid points. Specifically this is achieved using a heap which contains one element for the tile and one for each boundary grid. Since we leave one direction untiled, at most nine elements will be present in the heap at once. In each iteration of the merging, the minimal element is deleted from the heap and inserted into the current slice being constructed. The minimal element is then replaced by the next element of the grid it originates from and the heap is reordered. In practice we do not merge on a grid point per grid point basis. In particular we can exploit that our grids are disjoint and have a simple blocked structure. Once we know that a certain grid point is the minimal element in the heap we can insert not only that grid point but all grid points from the same column and grid (tile or boundary)

into the current slice we are constructing. Note that the heap adds an overhead compared to streaming alone.

Notice from Fig. 15(c) that computations on grid points that fall inside a boundary grid will be performed twice on each tile if processed independently. The computational overhead will be proportional to the narrow band volume inside the boundary grids. Generally this will be small compared to the narrow band volume inside the tile grids, and this strategy also has the advantage that each tile can be processed in parallel. If duplication of computations on boundary grids is not desirable, the tiles can be processed sequentially and computations performed on a boundary grid can be stored to disk temporarily and used for initialization in adjacent tiles.

4 Results and Discussion

4.1 Single Threaded Performance

In this section we present benchmark evaluations of our out-of-core level set framework and compare its throughput to the throughput of the out-of-core framework by Nielsen et al. [44] as well as to the throughput obtainable for in-core simulations on the original DT-Grid data structure [42]. We define *throughput* as the number of gridpoints/voxels computed per second. Furthermore we define the *relative throughput* of a given simulation as the throughput of the simulation divided by the throughput of a similar simulation on an in-core DT-Grid at effective resolution 1000^3. The latter definition makes it easier to compare the throughput of our proposed method against the peak thoughput of a fully in-core simulation and data structure.

For the benchmark tests we have employed the following methodology. We consider three different level set flows: (1) Constant normal propagation (exemplified by an erosion), $\phi_t - |\nabla\phi| = 0$. (2) Advection in a velocity field where the maximal velocity is known prior to each level set step (exemplified by a translation), $\phi_t + \vec{V} \cdot \nabla\phi = 0$. (3) Mean curvature flow, $\phi_t - \kappa|\nabla\phi| = 0$. Each of these flows are simulated using two different numerical schemes: (1) A forward Euler temporal discretization combined with a spatial discretization of first order one-sided differences for the hyperbolic terms and second order central differences for the parabolic terms. (2) A third order TVD RK discretization in time combined with a spatial discretization of HJ WENO for the hyperbolic terms and second order central differences for the parabolic terms. We evaluate each combination of flow and numerical scheme on a narrow band DT-Grid based level set representation of the Stanford Bunny (see Fig. 2) at increasing resolution. For the high order numerical scheme (2), the maximal width of the narrow band corresponds to 7 grid cells, and for the low order numerical scheme (1) a width of 4 grid cells is used. For numerical scheme 1, we have run the simulations on narrow band grids ranging roughly over effective resolutions from 1000^3 (0.085 GB and 20.10 M voxels) to 14000^3 (17 GB and 4188 M voxels). For numerical scheme 2, the effective resolution of the narrow band grids range roughly from 1000^3 (0.13 GB and 34.27 M voxels) to 10000^3 (14 GB and 3568 M voxels). In each case the highest resolution corresponds roughly to the narrow band that allows for the maximal number of degrees of freedom representable in 32 bit. Since the narrow band is wider for numerical method 2, the largest model employed here is smaller in terms of resolution. Note that during simulation roughly twice the storage is required, since an input and an output out-of-core grid must be represented simultaneously. Hence our benchmark computations require up to roughly 34 GB of storage. The simulations were run on a computer with 32Bit Windows XP Pro, a single-core 2.41 GHz AMD

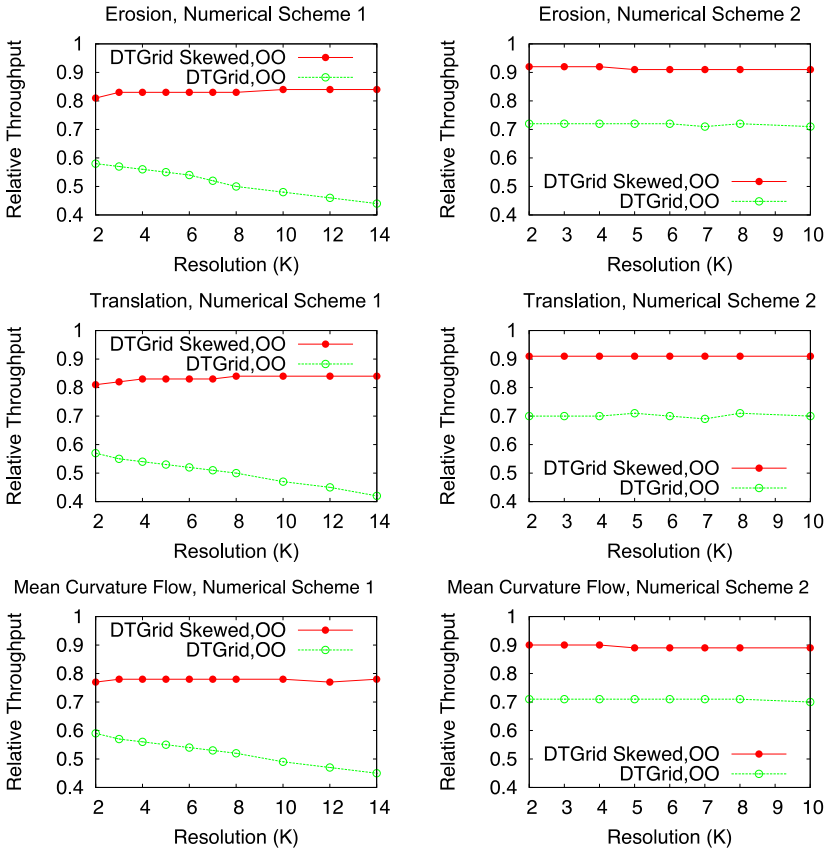


Fig. 16 Benchmarks from a propagation in the normal direction (*top*), advection *(middle)* and mean curvature flow (*bottom*). The *left column* shows results from numerical scheme 1 and the *right column* shows results from numerical scheme 2 (Sect. 4.1). Results are reported as the ratio of throughputs obtained by our framework (DTGrid Skewed, OO) and the framework in [44] (DTGrid, OO) to the throughput obtained by an in-core DT-Grid simulation at 1000^3

CPU, 1 GB of memory and a 10000 RPM Western Digital Raptor disk. The benchmark programs were implemented in C++ and compiled using Visual Studio 2005 with maximal optimization enabled. In our implementations of the framework proposed in this paper and the framework in [44], only the code for the level set algorithms differed. The underlying DT-Grid as well as the prefetching and page-replacement schemes all utilized the exact same code. For the single threaded benchmarks we utilized only a single tile and hence no boundary grids.

4.1.1 Performance of Skewed Simulations

As illustrated in Fig. 17, the performance of the DT-Grid drops notably as the main memory limit is reached. The out-of-core framework in [44] improves the performance, but remains IO limited as shown in Fig. 18 left. In contrast the framework proposed in this paper is CPU limited as depicted in Fig. 18 right.

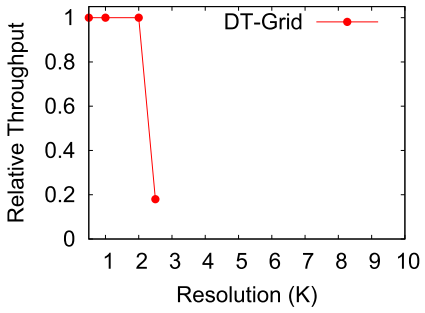


Fig. 17 The performance of the in-core DT-Grid drops when the main memory limit is exceeded

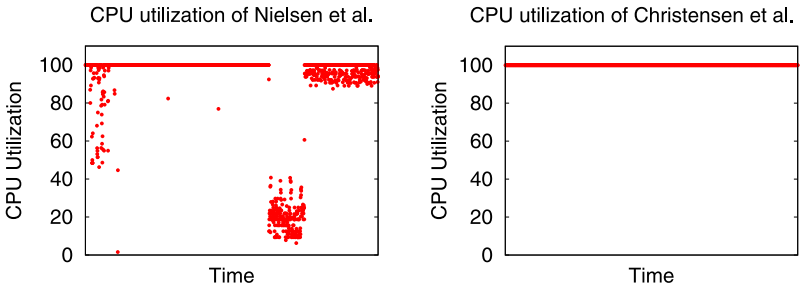


Fig. 18 This figure shows the CPU utilization over the course of a single time-step of an out-of-core simulation on the Stanford Bunny at resolution $\approx 8000^3$. *Left:* The framework of [44] is IO limited for both low and high order methods. *Right:* Our framework utilizes the CPU 100% and is thus CPU limited although the data is streamed to and from memory

Results from the benchmarks are shown in Fig. 16. The left column of Fig. 16 shows results obtained using numerical scheme 1. A relative performance of 1.0 corresponds to absolute throughputs of 1.4 M, 1.4 M and 1.2 M for the erosion, translation and mean curvature flow tests, respectively. For numerical scheme 1 there seems to be the following tendency: For our proposed framework, the relative throughput is roughly constant and appears to rise slightly for the propagation and advection tests. In all cases, our new framework outperforms that in [44] and the relative throughput of our method stays within 77–84 percent of peak in-core performance (a relative throughput of 1.0). In particular the performance of the framework in [44] drops as the resolution is increased, a consequence of an increasing IO bottleneck. We expect the performance to converge asymptotically to some fixed ratio of throughputs not reached within the resolutions spanned by our tests.

The right column of Fig. 16 shows results obtained using numerical scheme 2. A relative performance of 1.0 corresponds to absolute throughputs of 0.11 M, 0.12 M and 0.12 M for the erosion, translation and mean curvature flow tests, respectively. For numerical method 2, performance is roughly constant for both frameworks, but our framework outperforms that in [44] and the relative throughput of our method stays within 89–92 percent of peak in-core performance (a relative throughput of 1.0). The reason for this is that the framework in [44] is IO limited. Contrary to the case of numerical method 1, the ratio of throughputs has in this case converged for the framework in [44]. Performance is higher in the case of numerical method 2 than numerical method 1. This is due to the fact that numerical method 2 is of higher order, and the CPU overhead associated with our proposed framework is constant per byte streamed to and from disk. Since the higher order schemes require more CPU time, the relative overhead of our framework is lower for a higher order than a lower order method.

We conclude that for both numerical methods, the throughput of our framework appears to be sustained, independently of resolution and numerical method.

As we have argued above, the current framework is CPU bound with an overhead between 8–23% compared to the DT-Grid in-core narrow band level set method. The question is if this overhead can be reduced further. From analyzing the framework, it appears that about two thirds of the introduced overhead arises from checks in the code that ensure that iterators are updated correctly whenever they move to data in a new disk page. Because these checks have to be done for each access into the topology and value data, they comprise a substantial part of the overhead compared to a fully in-core method which does not have to perform these checks. We believe that most of the overhead associated with these checks can be eliminated on a 64 bit operating system, if the narrow band level set is not larger than the virtual address space. The strategy is to essentially memory map the file into the virtual address space, but in such a way that only active pages are actually allocated and that pre-fetching and page-replacement is done using the methods in [44]. We are currently investigating this.

4.2 Multi Threaded Performance

Parallelization techniques are required in order to run simulations within feasible time constraints. In this section we evaluate the performance of our framework in a multi-threaded environment. In particular we evaluate its performance when running several out-of-core simulations on the same disk and compare performance to the out-of-core framework by Nielsen et al. [44]. Furthermore we evaluate the parallelization overhead introduced by combining skewing and tiling transformations and benchmark the parallel performance of our framework. For single threaded applications we have not found it necessary to tile the grid in practice. The reason is that the computation-time becomes infeasible before the memory limit is exceeded by the number of slices required in-core by the computation. For this reason we only apply tiling for multi-threaded applications.

The multi-threaded experiments were run on a computer with 64Bit Windows Vista Business, two Intel Xeon 2.8 GHz quadcore processors (8 cores in total) and three 7200 RPM disks. All experiments utilized less than 2 GB of memory. Our framework was parallelized using Intel Thread Building Blocks [48], and boundary grids were kept in-core for these tests.

4.2.1 Performance of Multiple Simulations on the Same Disk

In this experiment we compare the performance of the framework in [44] to our framework in the situation where several concurrently running simulations utilize and share the bandwidth of the same disk. As test case we consider mean curvature flow on the Stanford Bunny at resolution $\approx 8000^3$ combined with numerical method 1 described in Sect. 4.1. In our framework we concatenated two level set steps as described in Sect. 3.1.5 to obtain better IO-efficiency. In Fig. 19 we report the change in simulation performance when increasing the number of instances of the same simulation running simultaneously on the same computer and using the same disk. The framework of Nielsen et al. is IO-limited for a single simulation, and the throughput drops significantly as the number of simulations utilizing the same disk is increased. On the contrary, the throughput of our framework by and large stays constant, except for a slight drop ($\approx 1\%$) at the beginning.

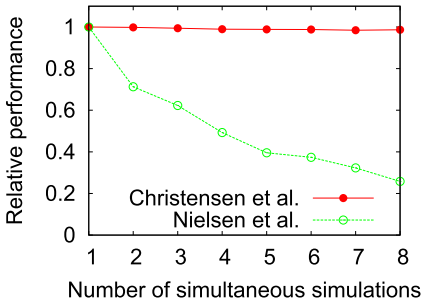


Fig. 19 This figure shows the change in performance as the number of instances of the same simulation running simultaneously on the same computer and utilizing the same disk is increased

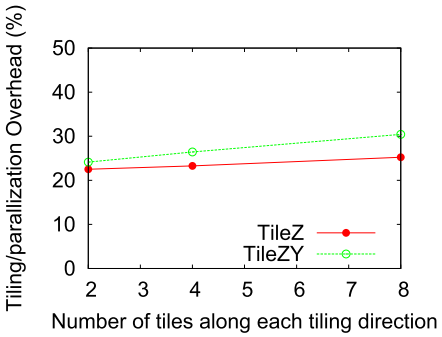


Fig. 20 This figure shows the tiling/parallelization overhead of our framework as a function of the number of tiles along *each* tiling-direction. *TileZ* tiles only in the Z-direction whereas *TileZY* tiles in both the Y- and Z-directions. The tiling/parallelization overhead is defined as percentage increase in execution time resulting from a single-threaded run of our framework using both tiling and skewing versus a single-threaded run of our framework using only skewing transformations. As test case we consider advection of the Stanford Bunny at resolution $\approx 8000^3$ combined with numerical method 1 described in Sect. 4.1

4.2.2 Parallelization Overhead

The best serial algorithm is seldom the best parallel algorithm [48], and a parallelization overhead is introduced by the tiling transformation which in turn is required in order to facilitate multi-threading. For a given dataset, the overhead grows as the number of tiles grows, as this will cause an increase in the number of boundary grids and hence redundant computations. Additionally, the overhead depends on the size of the boundary grids which depends on the number of concatenated level set steps as well as the size of the numerical stencils. Finally the cost of merging grid points from boundary grids and tiles will increase since the heap used to sort these will contain more elements. Figure 20 shows the tiling/parallelization overhead for two different blocking schemes. The overhead in this case lies between 23% and 30% when compared to single-threaded performance. A conclusion to be drawn from Fig. 20 is that tiling in only one direction is preferable over tiling in two directions, if memory requirements permit it.

4.2.3 Performance of Skewed and Tiled Simulations

Once the simulation is set up, there are virtually no serial sections in our framework implementation, except for the code that logs performance and saves level set data to disk. Hence

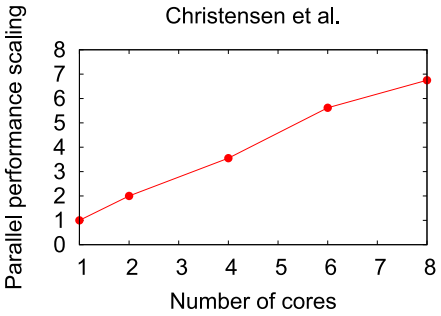


Fig. 21 This figure shows the parallel performance scaling as a function of the number of cores. To emphasize the scaling trend, the parallel performance scaling is computed as the execution time divided by the execution time of the parallel version of our framework (using both skewing and tiling) running on a single core. As test case we consider advection of the Stanford Bunny at resolution $\approx 8000^3$ combined with numerical method 1 described in Sect. 4.1

according to Amdahl's law, our framework should have good theoretical parallel performance scaling properties. As can be seen from Fig. 21, the parallel performance of our current framework implementation scales sub-optimally as the number of cores are increased, obtaining a parallel speedup of 6.75 using 8 cores and using roughly 104 seconds per level set time-step. The reason for not obtaining $8\times$-performance is not due to IO limitations, since our framework remains CPU limited. In fact we observed similar scaling properties when keeping all components (values and topology) in-core on smaller data sets. For the tests in this section we also attempted to diminish load-imbalancing by applying a simple non-uniform tiling strategy in which the number of tiles equals the number of cores used for the computations. Each core is then assigned to a specific tile, and the tiles are constructed in such a way that the number of grid points in each tile is roughly constant. Furthermore the data was distributed evenly on the computer's three disks. We leave an investigation of using more cores, a different architecture as well as improvements of the parallel performance scaling of our framework for future work.

5 Applications

5.1 The Divergence-Free Advection Test

In this section we demonstrate an extreme level set deformation by advecting 8 spheres through the incompressible, periodic velocity field originally proposed in [11, 28]:

$$u(x,y,z) = 2\sin^2(\pi x)\sin(2\pi y)\sin(2\pi z)\cos\left(\frac{t\pi}{T}\right)$$

$$v(x,y,z) = -\sin(2\pi x)\sin^2(\pi y)\sin(2\pi z)\cos\left(\frac{t\pi}{T}\right)$$

$$w(x,y,z) = -\sin(2\pi x)\sin(2\pi y)\sin^2(\pi z)\cos\left(\frac{t\pi}{T}\right)$$

where $T = 3$ is the period of t. The velocity field is reversed at $t = 1.5$, and the advected level set should return to its original shape at time $t = 3$. The spheres have radius 0.125 and

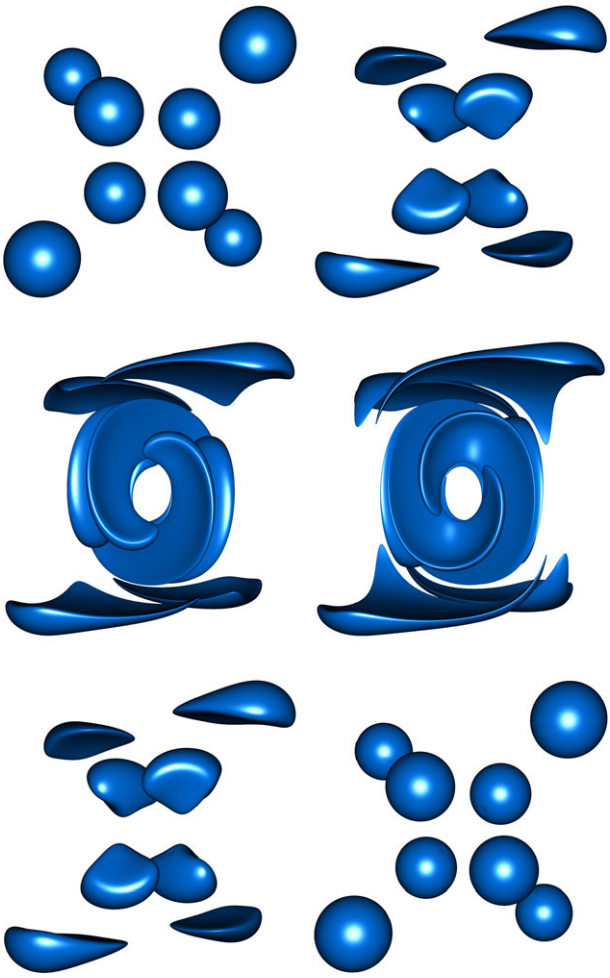


Fig. 22 Results of the divergence-free advection test on 8 spheres at resolution 2048^3 requiring up to 1.4 GB of storage (and twice as much during simulation). At the peak ($t = 1.5$), the narrow band contains approximately 343 million voxels. From top left to bottom right the images are from $t = 0, 0.3, 0.8, 1.5, 2.7, 3$, i.e. the *lower right image* shows the result of advecting forwards and then backwards

are placed in a unit computational domain at positions $(0.15, 0.15, 0.85)$, $(0.15, 0.85, 0.15)$, $(0.35, 0.35, 0.35)$, $(0.35, 0.65, 0.65)$, $(0.65, 0.35, 0.65)$, $(0.65, 0.65, 0.35)$, $(0.85, 0.15, 0.15)$ and $(0.85, 0.85, 0.85)$. The advection equation $\partial\phi/\partial t + \nabla\phi \cdot (u, v, w) = 0$ where ϕ is the level set function and t is time is solved using a third order accurate TVD RK discretization in time and a three—fifth order accurate HJ WENO discretization in space. Figure 22 shows the surfaces at various times during the deformation with the lower right image showing the final result after advecting forwards and backwards, i.e. a full period. For the simulation in this figure, the unit computational domain is sampled at resolution 2048^3, and the DT-Grid narrow band peaks at approximately 343 million voxels requiring 1.4 GB of storage (and roughly twice as much is needed during simulation). We have also run this simulation at resolution 4096^3, where the DT-Grid narrow band takes up 1.4 GB of storage at the beginning ($t = 0$). At $t = 1.5$ the storage requirements peak at 6.1 GB (i.e. around 12.2 GB during simulation) and approximately 1.4 billion voxels are contained in the narrow band. Both simulations were run on a Mac Pro work station with two Intel Xeon quad core 2.80 GHz CPUs, 4 GB of memory and 4 7200 rpm hard drives, which were all utilized. Tiling was performed in the Z-direction with a core assigned to each tile (8 cores/tiles in total), and

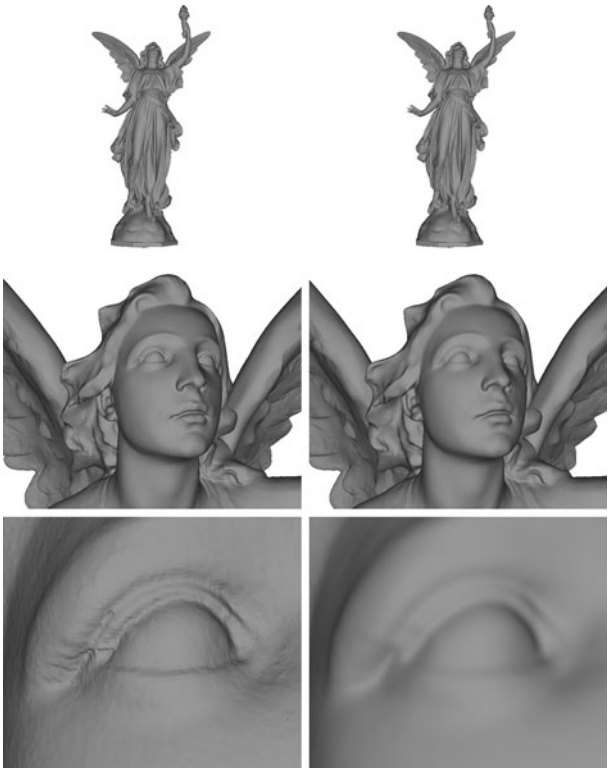


Fig. 23 Results of the mean curvature motion on the lucy statue scan converted into a DT-grid at resolution $17149 \times 9987 \times 5734$ taking up 11.5 GB of storage (note that during simulation roughly twice the storage is required). Initial surface to the left and after 200 iterations to the right. The *first row* shows the whole statue, the *second* a zoom in on the head region and the *third* depicts an even closer zoom to the eye

each iteration of the former (latter) simulation took from 103 (339) seconds in the beginning and end ($t = 0$ and $t = 3$) to 390 (1436) seconds around the peak ($t = 1.5$). The rendering of the former simulation was performed by in-core ray tracing on a different machine.

5.2 Mean Curvature Flow of Surfaces

The applications of curvature-based surface flows are vast. In this section we illustrate the use of mean curvature motion for surface smoothing expressed by the simple level set equation $\partial\phi/\partial t = \kappa|\nabla\phi|$, where κ is the mean curvature of the surface, ϕ is the level set function and t is time. We discretized the equation using first order accurate Forward Euler in time and second order accurate central differences in space. Figure 23 shows the Lucy statuette from the Stanford Scanning Repository [54] scan converted into a DT-grid narrow band level set at resolution $17149 \times 9987 \times 5734$ using the method for manifold meshes proposed in [17]. Each instance of the DT-Grid is 11.5 GB, and during simulation roughly twice the storage is required. The simulation was run on a Dell Inspiron 8600 Laptop with 1 GB of memory and a 7200 rpm hard drive. Even on this hardware configuration, our out-of-core framework remains CPU limited, one iteration taking approximately 108 minutes. While not necessarily feasible for simulations requiring many iterations, due to time constraints, this illustrates that our framework is applicable even on architectures with limited resources. Notice that in the initial surface on the left, the triangulation of the original Lucy polygonal model, from which the level set was scan converted, is visible. However, since the model is super-sampled using the level set, and hence has a larger number of degrees of freedom, the mean curvature motion effectively results in a smooth interpolation of this model to higher

resolution. The rendering was done by Gouraud shading of triangles, streamed to the graphics card and extracted from the level set using the marching cubes algorithm [27]. Since the marching cubes algorithm will produce many more triangles for this 11.5 GB level set than can be stored on the graphics card, a draw command is repeatedly issued whenever a fixed number of triangles have been streamed to the graphics card.

6 Conclusion and Future Work

We have presented a fast, storage efficient and parallelizable out-of-core framework for performing computations on level sets at resolutions only limited by the size of disk space. The framework utilizes code transformations to allow the combination of multiple passes over the data into a single pass. As a result, the level set algorithms become CPU limited and maintain a throughput between 77% and 92% of peak in-core performance, independently of the level set resolution.

The framework still incurs an overhead compared to a strictly in-core level set method. As mentioned in Sect. 4.1.1, we believe that a careful engineering effort can reduce this overhead. In the future, we wish to further investigate and improve the performance of our out-of-core framework for parallelization and multi-threading. In particular, given the high resolutions enabled by our framework, computation times are now the main bottleneck in achieving results at a desired resolution within a desired time frame. Investigating our framework in the context of parallelization over more cores and CPUs as well as in combination with implicit methods for solving PDEs, which would allow for larger time-steps, are promising directions for future work. Additionally, an automatic determination of the number of computations that can be concatenated in order to fully utilize internal memory would ensure that no excess disk bandwidth is used. It would probably require an initial pass through the data coupled with a statistical model of the upcoming memory requirements resulting from the computation. Similarly, determining and adjusting tiling-directions as well as tile sizes automatically depending on the amount of memory available, would be useful. Currently, we assume it is possible to set up a configuration at the beginning that will remain valid throughout the lifespan of the simulation. This has been the case for all the simulations presented in this paper, however the ability to adjust the tile sizes dynamically could prove useful for load balancing the CPUs.

While we have focused entirely on level set computations, our techniques are also relevant for general out-of-core stencil based computations, and it would be interesting to investigate similar strategies for stencil-based computations such as fluid simulation and the solution of linear equation systems arising from an implicit discretization.

Acknowledgements The authors wish to thank Ola Nilsson for help with Figs. 1 and 2. This work was partially funded by the Danish Agency for Science, Technology and Innovation.

Appendix: Data Locality Analysis

In the following appendices we perform data locality analysis of level set FD schemes using the model of Wolf and Lam [62]. For the sake of completeness, we briefly introduce the model and explain how to use it for analysis.

Considering a perfectly nested loop of depth n, we look at the iteration space which corresponds to a convex polyhedron in $\mathbb{Z}^n$ bounded by the loop bounds. We can identify

each iteration by a node inside this polyhedron using a vector $\vec{p} = (p_1, p_2, \ldots, p_n)$, where p_i is the loop index of the i'th loop in the nest. An execution of the loop-nest corresponds to visiting all nodes in the polyhedron in lexicographic order. We have reuse of a data item if it is accessed in several iterations of the loop. Thus, reuse is inherent in a computation and does not depend on the execution order of the loops in the nest. However, reuse does not guarantee temporal or spatial memory locality since accesses to a particular data item might be separated by many accesses to other data. This means that in the worst case the data item will have to be loaded each time it is used. The space spanned by the iteration space directions in which reuse is found is called the *reuse vector space*. We can transform our iteration space which corresponds to performing loop transformations such as skewing. These transformations change the way the iteration space is traversed and thus the way we exploit reuse. We must however be sure that the data dependencies of the algorithm are not violated. The dependence vectors, which define dependencies between two nodes $\vec{p}_1$ and $\vec{p}_2$ in the iteration space, must be transformed as well. A dependence vector points from $\vec{p}_1$ to $\vec{p}_2$ if the execution of the statement at $\vec{p}_2$ depends on the result from $\vec{p}_1$. Hence a valid code transformation T must satisfy $T(\vec{p}_1) < T(\vec{p}_2)$, where $<$ is the lexicographic ordering.

While transformations might improve our utilization of reuse, they cannot alone exploit reuse in multiple dimensions. Therefore we also need to perform tiling. We can tile loops i through j (for $i < j$) if they are *fully permutable*, i.e. can be interchanged freely: A property satisfied if the dependence vectors are non-negative and have either lexicographically positive components d_1 through d_{i-1}, or components d_i through d_j which are non-negative. Thus we can also use transformations to enable tiling since the dependence vectors change.

The result of applying the transformations is a vector space spanned by the iteration directions in which reuse can be exploited. This vector space is called the *localized vector space*. The goal of our data locality analysis thus is: Given an iteration space and corresponding data dependence vectors, we want to apply skewing and tiling transformations in order to obtain a localized vector space which completely contains the reuse vector space. By ensuring this, data reuse will result in data locality allowing data to be accessed multiple times whilst in memory, thus avoiding data to be loaded from disk multiple times.

.1 Forward Euler

Algorithm 1 Euler with first order upwind (original)

```
1: for t ← 0, T do
2:     for x ← 0, X do
3:         A[t + 1, x] ← step(A[t, x], A[t, x − 1], A[t, x + 1])
4:     end for
5: end for
```

In this appendix we analyze the *forward Euler* algorithm in the model of Wolf and Lam [62]. In particular we derive the reuse vector space, propose specific loop transformations and then show that these transformations result in a localized vector space that includes the reuse vector space, hence exploiting maximal reuse. For the sake of simplicity, we carry out the analysis in one spatial dimension and explain how this analysis generalizes to higher spatial dimensions.

Consider the pseudo-code for forward Euler in Algorithm 1. To facilitate the analysis, we assume the existence of an array A that includes both the spatial and temporal dimensions.

Furthermore we assume that the initial data is present in $A[0, x]$ for all x in the spatial dimension, and that the upper loop bounds are not included in the iteration space (i.e. zero-based indexing). Note that in the actual implementation, our algorithms retain the iteration order illustrated in Algorithm 1, but only a thin narrow band is traversed, and iterators into the DT-Grid data structure are used to abstract away the details of topology encoding and streaming to and from disk. Furthermore we use the storage allocation scheme described in Sect. 3.1.

As explained in the previous appendix, we need to adhere to the dependencies of the algorithm when we transform the iteration space, i.e. we must ensure that an iteration does not execute before an iteration which it depends on. The dependence vectors are $\{(1, 0), (1, -1), (1, 1)\}$. To find the reuse vector space, we classify the memory references in terms of how they reuse memory. To facilitate this, we put two references $A[f(\vec{i})]$ and $A[g(\vec{i})]$ in the same equivalence class, called a *uniformly generated set*, if it for some linear transformation H and constant vectors $\vec{c_f}$ and $\vec{c_g}$ holds that $f(\vec{i}) = H\vec{i} + \vec{c_f}$ and $g(\vec{i}) = H\vec{i} + \vec{c_g}$.

All references in Algorithm 1 are uniformly generated with $H = \mathrm{Id}$ and constant vectors

$$c_1 = (1, 0)^T, \qquad c_2 = (0, 0)^T, \qquad c_3 = (0, -1)^T, \qquad c_4 = (0, 1)^T,$$

respectively. We wish to compute the reuse vector space containing all the directions of reuse, and therefore we look at group-spatial reuse (i.e. reuse as a result of references that are close in either time or space), as the associated reuse space R_{GS} contains all the other types of reuse. R_{GS} is defined as $\mathrm{span}\{r_2, \ldots, r_4\} + \ker H_S$, where H_S denotes H with the last row replaced by the zero-vector, and r_j for $j = 2, \ldots, 4$ is a particular solution of the linear equation $H\vec{r_j} = \vec{c_{S,1}} - \vec{c_{S,j}}$ where $\vec{c_{S,i}}$ is $\vec{c_i}$ with the last component set to 0. Since $\ker H_S = \mathrm{span}\{(0, 1)\}$, and the particular solutions all lie in $\mathrm{span}\{(1, 0)\}$, we get that $R_{GS} = \mathrm{span}\{(1, 0), (0, 1)\}$. We can thus see that there is reuse in the entire iteration space. The localized vector space for the original loops is however only $L = \mathrm{span}\{(0, 1)\}$. Hence R_{GS} is not fully contained in L, and in particular we only exploit reuse in the innermost loop. To improve this, we want to tile our loop nest in all necessary directions. However, it is not possible to do tiling in both loops as they are not fully permutable. Therefore, we have to skew our loops using the following transformation $T : [t, x] \to [t, x + t]$. The result is shown in Algorithm 2. This also transforms our dependence vectors: $\{(1, 1), (1, 0), (1, 2)\}$, and we can now clearly see, that our loops are fully permutable. Tiling the t and x loops with tile sizes B_T and B_X respectively, reveals the pseudocode in Algorithm 3 for which the localized vector space completely coincides with R_{GS} thus demonstrating that our skewing and tiling scheme exploits all reuses.

To minimize the IO latency and bandwidth of our computations in practice, we do not tile in the x-direction, independent of the number of spatial dimensions involved in the computations. Note that by not tiling in the x-direction we do not alter the localized vector space. This is due to the fact that the loop within the tile in the x-direction can become the outermost loop—among those loops that iterate *within* a tile—by simple loop-interchange. Furthermore, the innermost controlling loop can trivially be coalesced with the loop over its individual tiles since all loops are fully permutable. Algorithm 4 shows our final algorithm for forward Euler.

Generalization to higher order spatial schemes and higher spatial dimensions is straightforward. To use the *Hamilton-Jacobi Weighted ENO* (HJ WENO) [20, 21, 32] scheme, the transformation would skew by three instead of one, and thus becomes $T : [t, x] \to [t, x + 3t]$. In higher spatial dimensions, we skew identically and tile in each additional spatial dimension.

Algorithm 2 Euler with first order upwind (skewed)

```
1: for t ← 0, T do
2:     for x ← t, X + t do
3:         A[t + 1, x − t] ← step(A[t, x − t], A[t, x − 1 − t], A[t, x + 1 − t])
4:     end for
5: end for
```

Algorithm 3 Euler with first order upwind (skewed and tiled)

```
1: for t2 ← 0, T, BT do
2:     for x2 ← 0, X + T, BX do
3:         for t ← t2, min(T, t2 + BT) do
4:             for x ← max(t, x2), min(X + t, x2 + BX) do
5:                 A[t + 1, x − t] ← step(A[t, x − t], A[t, x − 1 − t], A[t, x + 1 − t])
6:             end for
7:         end for
8:     end for
9: end for
```

Algorithm 4 Euler with first order upwind (final)

```
1: for t2 ← 0, T, BT do
2:     for x ← 0, X + T do
3:         for t ← t2, min(T, t2 + BT) do
4:             if max(t, x) < min(X + t, x + 1) then
5:                 A[t + 1, x − t] ← step(A[t, x − t], A[t, x − 1 − t], A[t, x + 1 − t])
6:             end if
7:         end for
8:     end for
9: end for
```

.2 BFECC and TVD RK

We now proceed to analyze the *Back and Forth Error Compensation and Correction* (BFECC) [7] and *Total Variation Diminishing Runge Kutta* (TVD RK) [52] algorithms. The procedure of the analysis is similar for the two algorithms as they both consist of a number of propagation and weighted averaging steps. Due to the relative simplicity of the BFECC algorithm we will focus on this algorithm and only provide an outline of the TVD RK analysis along with the proposed transformations. Again, we carry out the analysis in one spatial dimension and then generalize it to higher spatial dimensions.

Consider the pseudo-code for the BFECC algorithm shown in Algorithm 5. To facilitate the analysis we have formulated the BFECC algorithm as a perfect loop nest by introducing the fictitious time variable t such that $\lfloor \frac{t}{4} \rfloor$ denotes the time-step and t mod 4 uniquely identifies one of the four assignment statements in the loop body. Furthermore, we have again assumed the existence of an array A that includes both the spatial and temporal dimension, and holds the initial data in $A[0, x]$ for all x in the spatial dimension. The storage mapping scheme can be found in Sect. 3.1.

Assuming the j'th array reference is expressed as $H[t, x]^T + \vec{c}_j$, only a single uniformly generated set of references is present as represented by the identity matrix $H =$

Algorithm 5 BFECC with first order upwind (original)

```
1: for t ← 0, 4T do
2:     for x ← 0, X do
3:         if t mod 4 = 0 then
4:             A[t + 1, x] ← step(A[t, x − 1], A[t, x], A[t, x + 1])
5:         else if t mod 4 = 1 then
6:             A[t + 1, x] ← backstep(A[t, x − 1], A[t, x], A[t, x + 1])
7:         else if t mod 4 = 2 then
8:             A[t + 1, x] ← average(A[t, x], A[t − 2, x])
9:         else
10:            A[t + 1, x] ← step(A[t, x − 1], A[t, x], A[t, x + 1])
11:        end if
12:    end for
13: end for
```

Algorithm 6 BFECC with first order upwind (skewed)

```
1: for t ← 0, 4T do
2:     x_start ← 3⌊t/4⌋ + min(t mod 4, 1) + ⌊(t mod 4)/3⌋
3:     for x ← x_start, X + x_start do
4:         w ← x − x_start
5:         if t mod 4 = 0 then
6:             A[t + 1, w] ← step(A[t, w − 1], A[t, w], A[t, w + 1])
7:         else if t mod 4 = 1 then
8:             A[t + 1, w] ← backstep(A[t, w − 1], A[t, w], A[t, w + 1])
9:         else if t mod 4 = 2 then
10:            A[t + 1, w] ← average(A[t, w], A[t − 2, w])
11:        else
12:            A[t + 1, w] ← step(A[t, w − 1], A[t, w], A[t, w + 1])
13:        end if
14:    end for
15: end for
```

Id. Since ker $H_S = \text{span}\{(0,1)\}$, and $\text{span}\{\vec{r}_j\} = \text{span}\{(1,0)\}$, we conclude that $R_{\text{GS}} = \text{span}\{(1,0),(0,1)\}$, hence the BFECC algorithm has reuse in both the temporal and spatial dimensions.

To fully exploit reuse, we must find loop transformations that result in a localized vector space L, such that $R_{\text{GS}} \subset L$ without violating the dependencies of the BFECC algorithm. Note that since the loop body has four separate cases, $t \bmod 4 = \{0,1,2,3\}$, we must consider the dependence vectors of *each* of these cases locally. For cases 0 and 2 they are $\{(1,-1),(1,0),(1,1)\}$, for case 1 the dependence vector is $\{(1,0)\}$ and for case 3 the dependence vectors are $\{(1,-1),(1,0),(1,1),(3,0)\}$ as depicted to the left in Fig. 24. Recall that the loops are fully permutable, and hence tilable, if all entries in the transformed dependence vectors are non-negative. Thus we see from the dependence vectors that by skewing by one in the x-direction from one case to the next, except from case 1 to case 2, we obtain a fully permutable loop nest. Specifically, we propose the skewing transformation $T : [t,x] \rightarrow [t, x + 3\lfloor \frac{t}{4} \rfloor + \min(t \bmod 4, 1) + \lfloor \frac{t \bmod 4}{3} \rfloor]$ which results in dependence vectors, (t,x), equal to $\{(1,0),(1,1),(1,2),(3,2)\}$ for case 3, $\{(1,0),(1,1),(1,2)\}$ for cases 0 and 2, and $\{(1,0)\}$

Algorithm 7 BFECC with first order upwind (skewed and tiled)

```
for t2 ← 0, 4T, Bt do
    for x2 ← 0, X + 3⌊(4T−1)/4⌋ + 2, Bx do
        for t ← t2, min(4T, t2 + Bt) do
            xstart ← 3⌊t/4⌋ + min(t mod 4, 1) + ⌊(t mod 4)/3⌋
            for x ← max(x2, xstart), min(x2 + Bx, X + xstart) do
                w ← x − xstart
                if t mod 4 = 0 then
                    A[t + 1, w] ← step(A[t, w − 1], A[t, w], A[t, w + 1])
                else if t mod 4 = 1 then
                    A[t + 1, w] ← backstep(A[t, w − 1], A[t, w], A[t, w + 1])
                else if t mod 4 = 2 then
                    A[t + 1, w] ← average(A[t, w], A[t − 2, w])
                else
                    A[t + 1, w] ← step(A[t, w − 1], A[t, w], A[t, w + 1])
                end if
            end for
        end for
    end for
end for
```

Algorithm 8 BFECC with first order upwind (final)

```
for t2 ← 0, 4T, Bt do
    for x ← 0, X + 3⌊(4T−1)/4⌋ + 2 do
        for t ← t2, min(4T, t2 + Bt) do
            xstart ← 3⌊t/4⌋ + min(t mod 4, 1) + ⌊(t mod 4)/3⌋
            if max(x, xstart) < min(x + 1, X + xstart) then
                w ← max(x, xstart) − xstart
                if t mod 4 = 0 then
                    A[t + 1, w] ← step(A[t, w − 1], A[t, w], A[t, w + 1])
                else if t mod 4 = 1 then
                    A[t + 1, w] ← backstep(A[t, w − 1], A[t, w], A[t, w + 1])
                else if t mod 4 = 2 then
                    A[t + 1, w] ← average(A[t, w], A[t − 2, w])
                else
                    A[t + 1, w] ← step(A[t, w − 1], A[t, w], A[t, w + 1])
                end if
            end if
        end for
    end for
end for
```

for case 1 as depicted to the right in Fig. 24. Note that the proposed transformation is not constant throughout the iteration space. However the proposed transformation is invariant in the x-direction for each case, since $T((t_1, x_1)) - T((t_2, x_2)) = T((t_1, x_1 + d)) - T((t_2, x_2 + d))$, so we can transform the end-points of the dependence vectors independent of their absolute position in the x-direction.

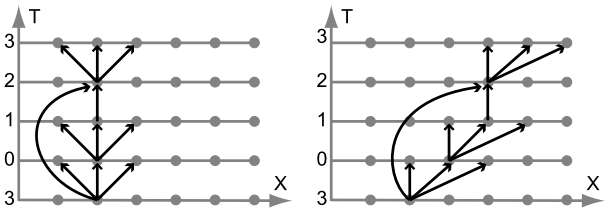


Fig. 24 Dependence vectors for the BFECC algorithm with first order spatial derivatives. The numering on the T axis is t mod 4. *Left*: Before skewing transformation. *Right*: After skewing transformation

Algorithm 6 shows the code resulting from skewing the iteration space. In particular the loop bounds are transformed using T and the array indices are transformed using T^{-1}. To obtain an optimal localized vector space, tiling and loop interchange is performed in Algorithm 7. To do the actual loop interchange in Algorithm 7 between the controlling loop in the x-direction and the loop within the tile for the temporal dimension we do the following: The bounds of the controlling loop in the x-direction are made independent of t by exchanging the minimum value for t in the lower bound and the maximum value for t in the upper bound. When doing this it is important to change the lower bound of the loop within the tile in the x-direction to $\max(x_2, x_{\text{start}})$, since this ensures that the loop within the tile starts at either the lower bound of the iteration space or at the lower bound of a tile within the iteration space. As in the case of the forward Euler algorithm, we do not tile in the x-direction, and the localized iteration space remains unchanged. In Algorithm 8 we show the BFECC algorithm that results from only tiling in the temporal dimension.

Generalization to higher order spatial schemes and higher spatial dimensions is again straightforward. To use the HJ WENO scheme, the transformation would skew by three instead of one in each case (except case 3 in which there is still no skewing), and thus becomes $T : [t, x] \to [t, x + 9\lfloor \frac{t}{4} \rfloor + 3\min(t \bmod 4, 1) + 3\lfloor \frac{t \bmod 4}{3} \rfloor]$. In higher spatial dimensions, we skew identically and tile in each additional spatial dimension, in two dimensions for example we obtain $T : [t, x, y] \to [t, x + 3\lfloor \frac{t}{4} \rfloor + \min(t \bmod 4, 1) + \lfloor \frac{t \bmod 4}{3} \rfloor, y + 3\lfloor \frac{t}{4} \rfloor + \min(t \bmod 4, 1) + \lfloor \frac{t \bmod 4}{3} \rfloor]$.

For a TVD RK method the analysis proceeds analogously. Take for example the third order accurate TVD RK scheme which consists of three advection steps and two averaging steps, as opposed to three advection steps and one averaging step in the case of the BFECC scheme. In particular the fictitious time includes a multiplicative factor of 5 (instead of 4 for BFECC), and the transformation for 1D TVD RK combined with a first order upwind scheme in the spatial dimension becomes $T : [t, x] \to [t, x + 3\lfloor \frac{t}{5} \rfloor + \min(t \bmod 5, 1) + \lfloor \frac{t \bmod 5}{3} \rfloor]$. This can readily be seen from a diagram similar to the one shown in Fig. 24. The generalization to higher spatial dimensions and higher order spatial schemes is identical to that of the BFECC scheme. In particular skewing is applied in each spatial dimension and the magnitude of skewing increases respectively.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988)
2. Adalsteinsson, D., Sethian, J.A.: A fast level set method for propagating interfaces. J. Comput. Phys. **118**(2), 269–277 (1995)
3. Bertalmío, M., Cheng, L.-T., Osher, S., Sapiro, G.: Variational problems and partial differential equations on implicit surfaces. J. Comput. Phys. **174**(2), 759–780 (2001)
4. Bibireata, A., Krishnan, S., Baumgartner, G., Cociorva, D., Lam, C., Sadayappan, P., Ramanujam, J., Bernholdt, D.E., Choppella, V.: Memory-constrained data locality optimization for tensor contractions.

In: Rauchwerger, L. (ed.) Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC'03). Lecture Notes in Computer Science, vol. 2958, pp. 93–108. Springer, Berlin (2004)
5. Bridson, R.: Computational aspects of dynamic surfaces. Ph.D. thesis, Stanford University (2003)
6. Chopp, D.L.: Computing minimal surfaces via level set curvature flow. J. Comput. Phys. **106**, 77–91 (1993)
7. Dupont, T.F., Liu, Y.: Back and forth error compensation and correction methods for removing errors induced by uneven gradients of the level set function. J. Comput. Phys. **190**(1), 311–324 (2003)
8. Droske, M., Rumpf, M.: A level set formulation for willmore flow. Interfaces Free Bound. **6**(3), 361–378 (2004)
9. Dervieux, A., Thomasset, F.: A finite element method for the simulation of Raleigh-Taylor instability. Lect. Notes Math. **771**, 145–158 (1979)
10. Dervieux, A., Thomasset, F.: Multifluid incompressible flows by a finite element method. In: Reynolds, W., MacCormack, R. (eds.) Seventh International Conference on Numerical Methods in Fluid Dynamics. Lecture Notes in Physics, vol. 141, pp. 158–163 (1980)
11. Enright, D., Fedkiw, R., Ferziger, J., Mitchell, I.: A hybrid particle level set method for improved interface capturing. J. Comput. Phys. **183**(1), 83–116 (2002)
12. Foster, N., Fedkiw, R.: Practical animation of liquids. In: Proceedings of ACM SIGGRAPH 2001. Computer Graphics Proceedings, Annual Conference Series, pp. 23–30. ACM, New York (2001)
13. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: 40th Annual Symposium on Foundations of Computer Science, New York, October 17–19, pp. 285–297 (1999)
14. Gibou, F., Fedkiw, R., Caflisch, R., Osher, S.: A level set approach for the numerical simulation of dendritic growth. J. Sci. Comput. **19**(1–3), 183–199 (2003)
15. Harten, A., Engquist, B., Osher, S., Chakravarthy, S.R.: Uniformly high order accurate essentially non-oscillatory schemes, iii. J. Comput. Phys. **131**(1), 3–47 (1997)
16. Hieber, S.E., Koumoutsakos, P.: A Lagrangian particle level set method. J. Comput. Phys. **210**(1), 342–367 (2005)
17. Houston, B., Nielsen, M., Batty, C., Nilsson, O., Museth, K.: Hierarchical RLE level set: a compact and versatile deformable surface representation. ACM Trans. Graph. **25**(1), 1–24 (2006)
18. Pen, U.-L., Trac, H.: Out-of-core hydrodynamic simulations for cosmological applications. New Astron. (2006)
19. Jiang, G.-S., Peng, D.: Weighted ENO schemes for Hamilton–Jacobi equations. SIAM J. Sci. Comput. **21**(6), 2126–2143 (1999)
20. Jiang, G.-S., Peng, D.: Weighted ENO schemes for Hamilton-Jacobi equations. SIAM J. Sci. Comput. **21**(6), 2126–2143 (1999)
21. Jiang, G.-S., Shu, C.-W.: Efficient implementation of weighted ENO schemes. J. Comput. Phys. **126**(1), 202–228 (1996)
22. Jeong, W.-K., Whitaker, R.T.: A fast iterative method for eikonal equations. SIAM J. Sci. Comput. **30**(5), 2512–2534 (2008)
23. Kandemir, M., Choudhary, A., Ramanujam, J.: An i/o-conscious tiling strategy for disk-resident data sets. J. Supercomput. **21**(3), 257–284 (2002)
24. Kandemir, M., Choudhary, A., Ramanujam, J., Kandaswamy, M.A.: A unified framework for optimizing locality, parallelism, and communication in out-of-core computations. IEEE Trans. Parallel Distrib. Syst. **11**(7), 648–668 (2000)
25. Kamil, S., Datta, K., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Implicit and explicit optimizations for stencil computations. In: MSPC'06: Proceedings of the (2006) Workshop on Memory System Performance and Correctness, pp. 51–60. ACM, New York (2004)
26. Kowarschik, M.: An overview of cache optimization techniques and cache-aware numerical algorithms. In: Algorithms for Memory Hierarchies. LNCS, vol. 2625, pp. 213–232. Springer, Berlin (2003)
27. Lorensen, W.E., Cline, H.E.: Marching cubes: a high resolution 3d surface construction algorithm. In: SIGGRAPH'87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, pp. 163–169. ACM, New York (1987)
28. LeVeque, R.: High-resolution conservative algorithms for advection in incompressible flow. SIAM J. Numer. Anal. **33**, 627–665 (1996)
29. Leventhal, A.: Flash storage memory. Commun. ACM **51**(7), 47–51 (2008)
30. Losasso, F., Fedkiw, R., Osher, S.: Spatially adaptive techniques for level set methods and incompressible flow. Comput. Fluids **35** (2005)
31. Lefohn, A.E., Kniss, J.M., Hansen, C.D., Whitaker, R.T.: Interactive deformation and visualization of level set surfaces using graphics hardware. In: VIS'03: Proceedings of the 14th IEEE Visualization (VIS'03), p. 11. IEEE Comput. Soc., Los Alamitos (2003)

32. Liu, X.D., Osher, S.J., Chan, T.: Weighted essentially nonoscillatory schemes. J. Comput. Phys. **115**, 200–212 (1994)
33. Liu, X.-D., Osher, S., Chan, T.: Weighted essentially non-oscillatory schemes. J. Comput. Phys. **115**(1), 200–212 (1994)
34. Li, Z., Song, Y.: Automatic tiling of iterative stencil loops. ACM Trans. Program. Lang. Syst. **26**(6), 975–1028 (2004)
35. Museth, K., Breen, D., Whitaker, R., Barr, A.: Level set surface editing operators. ACM Trans. Graph. (Proc. SIGGRAPH) **21**(3), 330–338 (2002)
36. Mckinley, K.S., Carr, S.: Improving data locality with loop transformations. ACM Trans. Program. Lang. Syst. **18**, 424–453 (1996)
37. Museth, K., Clive, M.: Cracktastic: fast 3d fragmentation in "the mummy: Tomb of the dragon emperor". In: SIGGRAPH'08: ACM SIGGRAPH, pp. 1–1. ACM, New York (2008)
38. Milne, R.B.: An adaptive level-set method. Ph.D. thesis, University of California, Berkeley (1995)
39. Min, C.: Local level set method in high dimension and codimension. J. Comput. Phys. **200**, 368–382 (2004)
40. Museth, K.: An efficient level set toolkit for visual effects. In: SIGGRAPH'09: SIGGRAPH (2009) Talks, pp. 1–1. ACM, New York (2009)
41. Nielsen, M.B.: Efficient and high resolution level set simulations. Ph.D. thesis, Aarhus University (2006)
42. 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)
43. Nemitz, O., Nielsen, M.B., Rumpf, M., Whitaker, R.: Finite element methods on very large, dynamic tubular grid encoded implicit surfaces. SIAM J. Sci. Comput. **31**(3), 2258–2281 (2009)
44. Nielsen, M.B., Nilsson, O., Söderström, A., Museth, K.: Out-of-core and compressed level set methods. ACM Trans. Graph. **26**(4), 26 (2007)
45. Osher, S., Cheng, L.-T., Kang, M., Shim, H., Tsai, Y.-H.: Geometric optics in a phase-space-based level set and Eulerian framework. J. Comput. Phys. **179**(2), 622–648 (2002)
46. Osher, S., Sethian, J.A.: Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. J. Comput. Phys. **79**, 12–49 (1988)
47. 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)
48. Reinders, J.: Intel Threading Building Blocks, 1st edn. O'Reilly, Sebastopol (2007)
49. Sussman, M., Almgren, A.S., Bell, J.B., Colella, P., Howell, L.H., Welcome, M.L.: An adaptive level set approach for incompressible two-phase flows. J. Comput. Phys. **148**(1), 81–124 (1999)
50. Sethian, J.A.: A fast marching level set method for monotonically advancing fronts. Proc. Natl. Acad. Sci. USA **93**(4), 1591–1595 (1996)
51. Song, Y., Li, Z.: New tiling techniques to improve cache temporal locality. In: SIGPLAN Conference on Programming Language Design and Implementation, pp. 215–228 (1999)
52. Shu, C.W., Osher, S.: Efficient implementation of essentially non-oscillatory shock capturing schemes. J. Comput. Phys. **77**, 439–471 (1988)
53. Sussman, M., Smereka, P., Osher, S.: A level set approach for computing solutions to incompressible two-phase flow. J. Comput. Phys. **114**(1), 146–159 (1994)
54. Stanford scanning repository. http://graphics.stanford.edu/data/3Dscanrep/
55. Strain, J.: Tree methods for moving interfaces. J. Comput. Phys. **151**(2), 616–648 (1999)
56. Salmon, J.K., Warren, M.S.: Parallel, out-of-core methods for n-body simulation. In: PPSC (1997)
57. Toledo, S.: A survey of out-of-core algorithms in numerical linear algebra, pp. 161–179 (1999)
58. Tsitsiklis, J.N.: Efficient algorithms for globally optimal trajectories. In: Proceedings of the 33rd Conference on Decision and Control, Lake Buena Vista, LF, pp. 1368–1373 (1994)
59. Tsitsiklis, J.N.: Efficient algorithms for globally optimal trajectories. IEEE Trans. Autom. Control **40**, 1528–1538 (1995)
60. Vitter, J.S.: External memory algorithms and data structures: dealing with massive data. ACM Comput. Surv. **33**(2), 209–271 (2001)
61. Whitaker, R.T.: A level-set approach to 3d reconstruction from range data. Int. J. Comput. Vis. **29**(3), 203–231 (1998)
62. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: PLDI'91: Proceedings of the ACM SIGPLAN (1991) Conference on Programming Language Design and Implementation pp. 30–44. ACM, New York (1991)
63. Wonnacott, D.: Achieving scalable locality with time skewing. Int. J. Parallel Program. **30**(3), 181–221 (2002)