Contents lists available at ScienceDirect
Journal of Computational Physics
www.elsevier.com/locate/jcp

Parallel level-set methods on adaptive tree-based grids

Mohammad Mirzadeh [a,*], Arthur Guittet [a], Carsten Burstedde [c], Frederic Gibou [a,b]

[a] Department of Mechanical Engineering, University of California, Santa Barbara, CA 93106-5070, United States
[b] Department of Computer Science, University of California, Santa Barbara, CA 93106-5110, United States
[c] Institute for Numerical Simulation and Hausdorff Center for Mathematics, University of Bonn, 53115 Bonn, Germany

A R T I C L E   I N F O

Article history:
Received 28 June 2015
Received in revised form 18 February 2016
Accepted 10 June 2016
Available online xxxx

Keywords:
Quadtree/Octree grids
Parallel computing
Semi-Lagrangian method
Level-set method
Adaptive mesh refinement
Space filling curves
Stefan problem

A B S T R A C T

We present scalable algorithms for the level-set method on dynamic, adaptive Quadtree and Octree Cartesian grids. The algorithms are fully parallelized and implemented using the `MPI` standard and the open-source `p4est` library. We solve the level set equation with a semi-Lagrangian method which, similar to its serial implementation, is free of any time-step restrictions. This is achieved by introducing a scalable global interpolation scheme on adaptive tree-based grids. Moreover, we present a simple parallel reinitialization scheme using the pseudo-time transient formulation. Both parallel algorithms scale on the Stampede supercomputer, where we are currently using up to 4096 CPU cores, the limit of our current account. Finally, a relevant application of the algorithms is presented in modeling a crystallization phenomenon by solving a Stefan problem, illustrating a level of detail that would be impossible to achieve without a parallel adaptive strategy. We believe that the algorithms presented in this article will be of interest and useful to researchers working with the level-set framework and modeling multi-scale physics in general.

1. Introduction

The level-set method, originally proposed by Sethian and Osher [37], is a popular and powerful framework for tracking arbitrary interfaces that undergo complicated topological changes. As a result, the level-set method has been used to a wide range of applications such as multiphase flows, image segmentation, and computer graphics [35,46]. An important feature of this method is that the location of the interface is defined implicitly on an underlying grid. This convenience, however, comes at a price. First, compared to an explicit method, e.g. front tracking [28,54], the level-set method is typically less accurate and mass conservation could be a problem, although progress has been made in resolving this issue [19]. Second, the level-set function has to be defined in a one dimension higher space than that of the interface. If only the location of the interface is needed, the added dimension greatly increases the overall computational cost for uniform grids. One way to avoid this problem is by computing the level-set only close to the interface, e.g. as in the narrow-band level-set method [1] or, more recently, by using a hash table to restrict both computation and storage requirements [10].

Another approach that can address both problems is the use of local grid refinement. In [48] the idea of using tree-based grids for level-set calculations was first introduced and later extended in [38,31] for fluid simulations. More recently, authors in [33] proposed second-order accurate level-set methods on Quadtree (two spatial dimensions) and Octree (three spatial

* Corresponding author.
E-mail address: m.mirzadeh@engineering.ucsb.edu (M. Mirzadeh).

http://dx.doi.org/10.1016/j.jcp.2016.06.017
0021-9991/© 2016 Elsevier Inc. All rights reserved.

dimensions) grids. The use of adaptive tree-base grids in the context of the level-set method is quite advantageous because (i) it gives fine-grain control over errors, which typically occur close to the interface and (ii) it can effectively reduce the dimensionality of the problem by focusing most of the grid cells close to the interface. Fortunately, constructing the tree is quite simple in the presence of an interface that naturally defines an ideal metric for refinement. However, even though the use of adaptive grids can dramatically reduce the computational cost, performing high-resolution three dimensional calculations of complex interfacial problems, e.g. crystal growth in binary alloys [51], could be prohibitively expensive or even impossible on a serial machine. In this paper we extend the level-set technology on Quad-/Octrees by proposing parallel algorithms for distributed memory machines using a domain decomposition technique.

One of the main challenges in parallelizing algorithms on adaptive grids is handling the grid itself. One option is to replicate the entire grid on each process and to employ serial ordering techniques, as implemented in an earlier version of the `deal.II` library [6], or to use serial graph partitioners such as METIS [29]. This approach, however, is only scalable to a few hundred processes at best and is limited by the size of the grid itself that can fit in memory. Even though parallel general-purpose partitioners have since been popularized [30,8] and the scalability of partitioning algorithms for unstructured grids has been improved (see e.g. [41]), their use adds extra overhead that can limit the overall scalability. Refining a grid consisting of multiple trees using recursive coordinate bisection has been implemented in the SIERRA framework [47]. Interestingly, tree-based grids have a nice spatial ordering that naturally leads to the concept of space-filling curves (SFCs) and can be efficiently exploited for parallel load balancing [4,21,14].

The idea of using SFCs for parallel partitioning of Quad-/Octrees is not new in itself and has been used by many researchers. For instance, `Octor` [55] uses a Morton curve (also known as Z-curve) for traversing the leaves of an Octree for indexing and load balancing and has been scaled up to 62 000 CPU cores [12]. `Dendro` [43] is an example of a so-called linear Octree code in which new algorithms are introduced for parallel partitioning and the development of a parallel geometric multigrid that has been scaled up to about 32 000 cores [44]. More recently, authors in [13] extended these ideas by optionally allowing for a collection, or a "forest", of connected Octrees, which is partitioned in parallel using a global Morton curve. The `p4est` library [11] provides a publicly available implementation of these algorithms that is equally efficient for a single tree as well as multiple trees and has been shown to scale to more than 458 k CPU cores [26]. Applications built with `p4est` have scaled to 1.57 M cores [40] and 3.14 M cores [34]. In fact, the algorithms presented in this paper are implemented on top of the `p4est` API. Due to the need for multiple adaptation and partitioning operations in each time step, the semi-Lagrangian method we describe below presents a stringent test of the algorithms and implementation both in terms of scalability and absolute run time.

Parallel level-set algorithms can be categorized into two groups: parallel advection algorithms and parallel reinitialization algorithms. Eulerian advection schemes can easily be parallelized but unfortunately are limited by the CFL condition, which could be very restrictive for adaptive grids. Semi-Lagrangian methods combine the unconditional stability of Lagrangian methods and the ease of use of Eulerian grids and have been successfully used for advecting the level-set function on tree-based grids [31–33]. However, parallelizing the semi-Lagrangian algorithm in a domain decomposition context is not an easy task. The reason for this is twofold. First, depending on the CFL number, the departure points may end up outside the ghost region and in remote processes that are potentially far away. This requires a very dynamic and nonuniform communication pattern that is challenging to implement. For an adaptive grid, the situation is even more complex due to the asymmetric nature of the communication pattern (cf. section 3). Second, load balancing could be an issue for large CFL numbers and nonuniform velocity fields, due to clustering of departure points, which can thus considerably restrict the scalability of the algorithm. Both of these problems, of course, could be avoided by choosing CFL ≤ 1 but that would defeat the purpose of using the semi-Lagrangian algorithm in the first place.

Nonetheless, several parallel semi-Lagrangian algorithms have been proposed. A simple domain decomposition technique was used in [52] where the width of the ghost layer is fixed based on the maximum CFL number to ensure that all departure points are covered by the ghost layer. Good scalings were reported for small CFL numbers (CFL ≤ 2). However, for large CFL numbers, this leads to a large volume of communication that can limit the scalability. In [18] the authors propose a more sophisticated domain decomposition approach which uses a "dynamic ghost layer". Here the width of the ghost layer is dynamically determined at runtime based on information from previous time steps. Unfortunately, this approach also suffers from excessive communication overhead for large numbers of processes. More recently, the authors in [58] used a domain decomposition strategy on a cubed sphere but with a single layer of ghost nodes. Interpolation on remote processes is then handled by sending query points to the corresponding process and asking for the interpolated result. This approach seems to provide good scalability for transporting a single tracer up to about 1000 cores for CFL ~ 10. At higher CFL numbers, the method begins to loose scalability due to an increase in communication volume. Finally, note that although we are mainly interested in parallel semi-Lagrangian methods, one could resort to finite difference or finite element discretization methods if small CFL numbers are acceptable. Indeed several algorithms of this type have been proposed with applications to modeling dendritic crystal growth [57], multiphase flows [50,20,39], and atomization process [23].

In many applications, it is desirable that the level-set function has the signed-distance property, i.e., $|\nabla\phi| = 1$. Generally, there are two approaches to enforce this property, either by solving the pseudo-time transient reinitialization equation [49, 36]

$$\phi_\tau + S(\phi_0)\,(|\nabla\phi| - 1) = 0,$$

or by solving the Eikonal equation

$$F(x)|\nabla\phi| = 1$$

with constant speed function $F(x) \equiv 1$. The transient reinitialization equation can be solved using explicit finite differences and thus can easily be parallelized using a domain decomposition approach. Moreover, only a few iterations may be needed if the signed-distance property is only required close to the interface [33]. This is the approach we have chosen in this paper. However, if the signed-distance property is required in the entire domain, solving the Eikonal equation is more computationally efficient. Unfortunately, the most popular algorithm for solving the Eikonal equation, i.e. the Fast Marching Method (FMM) [45,46], is inherently sequential due to causal relationship between grid points and cannot be easily parallelized. The Fast Sweeping Method (FSM) [59] is an alternative for solving the Eikonal equation iteratively. The FSM can be more computationally efficient for simple choices of speed function, e.g. as in this context, and for simple interfaces. Moreover, FSM has more potential for parallelization compared to the FMM.

One of the earliest attempt in parallelizing the FMM is reported in [22] where a domain decomposition algorithm was introduced. Unlike the serial FMM, however, parallel FMM potentially requires multiple iterations or "rollback operations" to enforce causality across processes. Similar ideas are described in detail in [56]. It should be noted that the number of iterations needed for the parallel FMM to converge greatly depends on the complexity of the interface and on the parallel partitioning and, in general, fewer iterations are required if the domains are aligned with the normals to the interface. Due to the nature of the Eikonal equation, shared memory machines might be a better environment for parallelization. For instance, in [9] the authors use an "adaptive" technique where individual threads implicitly define a domain decomposition at runtime. Unfortunately, this approach does not seem to be any more effective than a simple static decomposition. In [60] a parallel FSM method was presented for the first time, which suffered from a plateau in the speedup. A scalable FSM was more recently proposed in [17], where the Cuthill–McKee numbering was utilized to improve scalability. A two-scale, hybrid FMM-FSM was presented in [15] which, albeit being more complicated to implement, promises even better scalability. Finally, a parallel Fast Iterative Method (FIM) was proposed in [27]. The FIM is similar to FMM in that it also maintains a list of "active nodes". However, unlike FMM, FIM avoids sorting the list and allows for concurrent updating of all nodes in an iterative fashion. In this article we choose the pseudo-time transient formulation for two reasons: 1) it is considerably easier to parallelize on Quadtrees and Octrees and 2) we are merely interested in the signed-distance property close to the interface, which only requires a few iterations.

This article is organized as follows: In section 2, we briefly review the sequential algorithms and discretization methods for the level-set equation on adaptive tree-based grids. These ideas are then extended in section 3 to parallel environments using a domain decomposition method. In section 4, we provide several examples that illustrate the scalability of our algorithms. Finally, we close by providing an application of our method by considering the simulation of the solidification process by solving a Stefan problem in section 5.

2. The level-set method

The level-set method, introduced in [37], is an implicit framework for tracking interfaces that undergo complicated topological changes. In this framework, an interface is represented by the zero contour of a higher dimensional function, e.g. a curve in two spatial dimensions can be described as $\Gamma = \{(x, y)|\phi(x, y) = 0\}$, where $\phi(x, y)$ is the level-set function. The evolution of the curve under a velocity field $\underline{\mathbf{u}}$ is then obtained by solving the level-set equation:

$$\phi_t + \underline{\mathbf{u}} \cdot \underline{\nabla}\phi = 0. \tag{1}$$

When the velocity field does not depend on the level-set function itself, equation (1) can be solved using the semi-Lagrangian method. An important advantage of the semi-Lagrangian method over the regular finite difference method is its unconditional stability that allows for arbitrarily large time steps. This is particularly important when using adaptive grids since higher grid resolutions translate into impractically small time steps.

In general, an infinite number of level-set functions can describe the same zero contour and thus the same interface. However, it is desirable to choose a function with the signed distance property $|\underline{\nabla}\phi| = 1$. As detailed in section 1, we solve the pseudo-time transient reinitialization equation [49,36] to achieve this property,

$$\phi_\tau + S(\phi_0)\left(|\underline{\nabla}\phi| - 1\right) = 0, \tag{2}$$

where τ is a pseudo time step, ϕ_0 is any level-set function that correctly describes the interface location and $S(\phi_0)$ is an approximation of the sign function. Here, we do not go into the details of the sequential algorithms for solving equations (1) and (2). Instead, we note that the parallel algorithms presented in section 3 are based on the sequential methods presented earlier in [33] and refer the interested reader to the aforementioned articles and references therein for more details.

3. Parallel algorithms

To achieve good parallel performance, we must ensure sufficient scalability of all components. This observation prompted the dedicated development and optimization of several techniques, which we discuss in the present section.

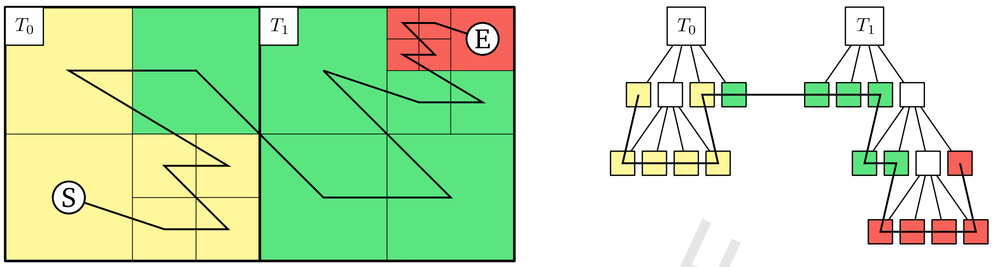


Fig. 1. Left: a "forest" made up of two trees T_0 and T_1. Parallel partitioning is achieved by first constructing a Z-curve starting at cell "S" and ending at cell "E". Next, the one dimensional curve is split up among processes either uniformly or by assigning different weights to cells. Here processes are represented via different colors. Note how using the Z-curve naturally leads to clustering of most cells in each domain. Right: schematic of a tree data structure representing this forest and its partitioning. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

3.1. Grid management

Adaptive tree-based grids can significantly reduce the computational cost of level-set methods by restricting the fine grid close to the interface where it is most needed [48]. Moreover, these grids are easy to generate in the presence of a signed-distance level-set function [33] and can efficiently be encoded using a tree data structure [42]. In order to develop scalable parallel algorithms, it is necessary to parallelize the data structure and grid manipulation methods such as refinement and coarsening of cells as well as to provide a fast method for grid partitioning and load balancing. The p4est library [11] is a collection of such parallel algorithms that has recently emerged and shown to scale up to 458 752 cores [26].

In p4est, the adaptive grid is represented as a non-overlapping collection of trees that are rooted in individual cells of a common coarse grid (cf. Fig. 1). This common coarse grid, which we will refer to as the "macromesh", can in general be an unstructured quadrilateral mesh, in two spatial dimensions, or hexahedral mesh, in three spatial dimensions. (In [47] such a coarsest possible grid has been introduced as the "genesis" mesh.) In this article, however, it is sufficient to limit discussions to simple uniform Cartesian macromeshes. Moreover, it is implicitly assumed that the macromesh is small enough that it can be entirely replicated on all processes. For instance, in many of the applications that we are considering in this paper the macromesh is simply a single cubical cell. p4est allows for arbitrary refinement and coarsening criteria through defining callback functions. In this article the refinement criteria is chosen based on the distance of individual cells to the interface. Specifically, a cell C is marked for refinement if

$$\min_{v\in V(C)} |\phi(v)| \le \frac{LD}{2}, \tag{3}$$

where $V(C)$ denotes the set of all vertices of cell C, L denotes the Lipschitz constant of the level-set function, and D denotes the diagonal size of cell C. Conversely, an existing cell is marked for coarsening if

$$\min_{v\in V(C)} |\phi(v)| > LD. \tag{4}$$

We refer to section 3.2 of [13] for details on the parallel refinement and coarsening algorithms implemented in p4est.

Once the grid is adapted to the interface, it must be partitioned to ensure load balancing across processes. This is achieved by constructing a Z-curve that traverses the leaves of all trees in order of the tree index (cf. Fig. 1). A Z-curve is a Space Filling Curve (SFC) with the important property that cells with close Z-indices are also geometrically close (on average) in the physical domain. This is beneficial since it leads to both a reduction in MPI communications and improvements of the cache performance of several algorithms such as interpolation and finite difference calculations. For more details on parallel partitioning in p4est one may refer to section 3.3 of [13]. Aside from grid manipulation and partitioning, we use two additional features of p4est, namely the generation of ghost layer cells and the creation of a globally unique node indexing. These algorithms are detailed in sections 3.5 and 3.6 of [13]. We have specifically extended the latter algorithm such that it can be applied to a non-graded refinement pattern. This is important because we can entirely skip the 2:1 balance function, which was shown to be one of the more time consuming parts of grid adaptation in p4est [13].

Finally, in p4est trees are linearized, i.e. only the leaves are explicitly stored. However, explicit knowledge of the hierarchal structure of the tree is greatly beneficial in several algorithms, e.g. in search operations needed for the interpolation algorithm. Thus, we introduce a simple reconstruction algorithm that recreates a local representation of the entire "forest" that is only adapted to local cells and, potentially, the ghost layer. This approach is similar to the ideas introduced in [7] and our tests show that in a typical application they amount to less that 1% of the entire runtime. Algorithm 1 illustrates how this reconstruction is performed. Given a forest and a layer of ghost cells from p4est, the algorithm generates a local representation of the forest by recursively refining from the root until reaching the same level and location of all leaves in the local forest and the ghost layer. Note that Algorithm 1 does not involve any communication and is load balanced provided that the initial forest is balanced. Fig. 2 illustrates an example where Algorithm 1 is applied. Note how each process

Algorithm 1 $H \leftarrow$ `Reconstruct`(G): Construction of the local tree hierarchy, H, from the parallel grid, G, supplied by `p4est`. The algorithm starts at the coarsest level, i.e. the macromesh, and recursively splits the each cell to match the finest local and ghost cells generated by `p4est`.

```
1: $H \leftarrow G$.macromesh()                                          ▷ start with the coarsest grid
2: for $tr$ in $G$.local_trees() do                          ▷ update the hierarchy to match local cells
3:     for $c$ in $tr$.local_cells() do
4:         $H$.update_tree($tr, c$)
5:     end for
6: end for
7: for $c$ in $G$.ghost_cells() do                            ▷ update the hierarchy to mach local cells
8:     $H$.update_tree($c$.get_parent_tree(), $c$)
9: end for
10: return $H$
11:
12: function $H$.update_tree($tr, c$)                                    ▷ recursive tree reconstruction
13:     $c_l \leftarrow H$.get_root_cell($tr$)
14:     while $c_l$.level() $\neq c$.level() do            ▷ recursively search for a cell of the same size as $c$
15:         if $c_l$.is_leaf() then $c_l$.split()    ▷ if the current cell is leaf, split the cell and continue search
16:         end if
17:         $h \leftarrow c_l$.length()                          ▷ select the search path based on cell coordinate.
18:         $i \leftarrow c.x \geq c_l.x + h/2$
19:         $j \leftarrow c.y \geq c_l.y + h/2$
20:         $k \leftarrow c.z \geq c_l.z + h/2$
21:         $c_l \leftarrow c_l$.get_child($i, j, k$)
22:     end while
23: end function
```

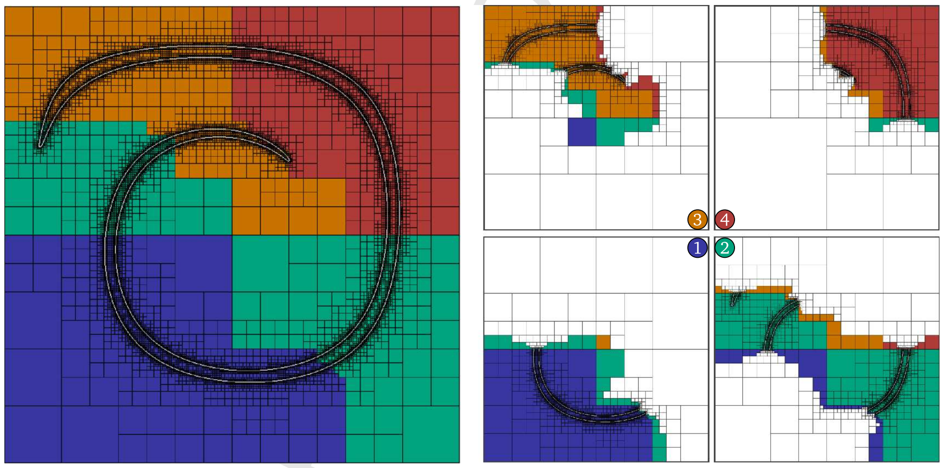


Fig. 2. Left: a forest refined close to an interface and partitioned among four processes, as indicated by colors. Right: each process independently recreates a local forest that is refined to match the local grid and is as coarse as possible elsewhere. Note that empty cells are fictitious, i.e. they are only required to generate the hierarchal structure and are not matched by any corresponding cell in the global forest. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

has independently generated a local representation of the forest that is refined to match the same leaves as in the global forest and ghost layer.

3.2. Interpolation and semi-Lagrangian methods

As indicated earlier, we use the semi-Lagrangian method to solve equation (1) when the velocity field is externally generated, i.e. when it does not depend explicitly on the level-set function itself. Let us rewrite equation (1) along the characteristic curve $\underline{\mathbf{X}}(t)$ as:

$$\begin{cases} \dfrac{\mathrm{d}\underline{\mathbf{X}}}{\mathrm{d}t} = \underline{\mathbf{u}}, \\ \dfrac{\mathrm{d}\phi(\underline{\mathbf{X}}(t), t)}{\mathrm{d}t} = 0. \end{cases} \tag{5}$$

The semi-Lagrangian method integrates equations (5) backward in time, i.e. starting from the grid G^{n+1} (computed iteratively as explained later on), we simply write $\phi^{n+1}(\underline{\mathbf{X}}^{n+1}) = \phi(\underline{\mathbf{X}}(t^{n+1}), t^{n+1}) = \phi(\underline{\mathbf{X}}(t^n), t^n) = \phi^n(\underline{\mathbf{X}}_d)$. Here, the characteristic

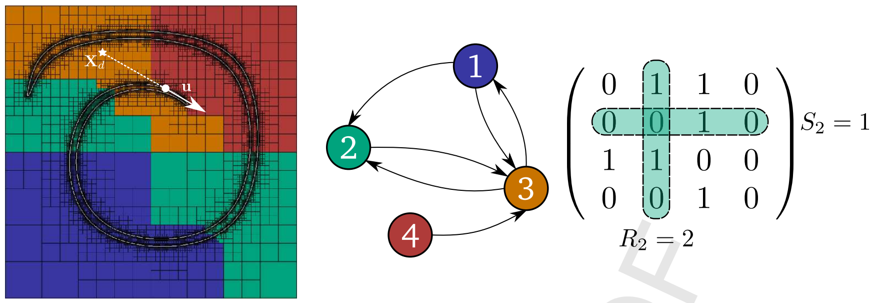


Fig. 3. Left: the location of back-traced points depends on the magnitude of the local velocity and on the time-step. Although the distance to the departure point is bounded by CFL $\Delta x_{\min}$, one cannot predict the receiving rank without explicitly searching the entire Z-curve. Moreover, the receiving process has no prior knowledge about which processes to check for incoming messages nor does it know anything about the possible message length (i.e. number of points). Middle: a directed graph illustrating the communication pattern among processes with arrows representing the direction in which messages are sent. Right: the adjacency matrix of the communication graph. For each row, the sum of all columns represents the number of messages that need to be sent. Conversely, for each column, the sum of all rows represents the number of messages that need to be received. As detailed in Algorithm 2, this information is enough to build a parallel interpolation scheme.

curves are chosen such that $\underline{\mathbf{X}}(t^{n+1})$ are the coordinates of grid points of G^{n+1}, and $\underline{\mathbf{X}}_d$ are the departure points, which are computed using the second-order midpoint method [33]:

$$\underline{\mathbf{X}}^\star = \underline{\mathbf{X}}^{n+1} - \frac{\Delta t}{2}\underline{\mathbf{u}}^n(\underline{\mathbf{X}}^n), \tag{6}$$

$$\underline{\mathbf{X}}_d = \underline{\mathbf{X}}^{n+1} - \Delta t \underline{\mathbf{u}}^{n+\frac{1}{2}}(\underline{\mathbf{X}}^\star), \tag{7}$$

where $\underline{\mathbf{u}}^{n+\frac{1}{2}}$ is obtained via extrapolation from previous times, i.e.:

$$\underline{\mathbf{u}}^{n+\frac{1}{2}} = \frac{3}{2}\underline{\mathbf{u}}^n - \frac{1}{2}\underline{\mathbf{u}}^{n-1}. \tag{8}$$

Note that all values at the intermediate point, $\underline{\mathbf{X}}^\star$, and departure point, $\underline{\mathbf{X}}_d$, must be calculated via interpolation from the previous grids G^n and G^{n-1}. Here, we use the stabilized second-order interpolation for $\phi(\underline{\mathbf{X}}_d)$ and the multi-linear interpolation for $\underline{\mathbf{u}}^{n+\frac{1}{2}}(\underline{\mathbf{X}}^\star)$ [33]. Although parallelization of the interpolation process on a shared-memory machine is trivial, the same cannot be said for distributed-memory machines. In fact, the parallel interpolation given in Algorithm 2 is probably the most important contribution of this article since this procedure, which is trivial on uniform grids, is challenging in the case of trees because it is not straightforward to identify which processes owns the departure points. Indeed, complications arise because not all departure points will reside in the domain owned by the current process. Moreover, due to the irregular shapes of the partitions, one cannot even ensure they are entirely owned by neighboring processes. At best we can only expect that their locations are bounded by a halo of width $w \le \text{CFL}\,\Delta x_{\min}$ around the local partition, where $\Delta x_{\min}$ is the size of the smallest cell in the forest. Naturally, if one enforces CFL ≤ 1, one can ensure that the halo is bounded by the ghost layer, which significantly simplifies the communication problem. This assumption, however, defeats the purpose of using a semi-Lagrangian approach, whose purpose is to enable large CFL calculations.

One remedy to this problem, proposed in [52] for uniform grids, is to increase the size of ghost layer to $\lceil \text{CFL} \rceil$. For large values of the CFL number, however, this approach can substantially increase the communication volume. Moreover, this simple approach does not work in the process of generating G^{n+1} due to repartitioning. Indeed, G^{n+1} is built iteratively and load balancing is enforced by repartitioning at each sub-iteration. Therefore, after one such sub-iteration, the backtracked points can end up outside of the initial ghost layer. An alternative approach would be to handle local and remote interpolations separately. Our remote interpolation algorithm is composed of three separate phases. In the first phase, which we call buffering, every process searches for all departure points inside the local trees. If the point is owned by a local cell, it is added to a local buffer, otherwise we find the process which owns the point and add the point to a separate buffer belonging to the found rank. Note that searching the point in the local tree is performed recursively using the hierarchal reconstruction (cf. Algorithm 1). Moreover, the owner's rank is found by computing the Z-index of the point and then using a binary search on the Z-curve. This is already implemented in `p4est` and explained in details in section 2.5 of [13].

Once buffering is done, every process knows exactly how many messages it needs to send and to which processes. This also implicitly defines processes that will later on send a reply message to this process. However, at this point no process knows which processes to expect a message from. We solve this problem using a simple *communication matrix* (see Fig. 3). Our approach is very similar to the "Personalized Census ($\mathcal{PCX}$)" algorithm described in [24]. Another similar approach is the "Notify" algorithm introduced in [25]. Furthermore, the MPI-3 standard introduces non-blocking collectives and Remote Memory Access (RMA) operations which enable new ways of solving the communication problem. For instance, authors in [24] descried the "Non-blocking Consensus ($\mathcal{NBX}$)" and "Remote Summation ($\mathcal{RSX}$)" algorithms which make use of such operations and have better theoretical communication complexities. With the exception of $\mathcal{RSX}$ algorithm, which was not tested in this study, all remaining algorithms produced similar timing and scaling. Thus we have decided to describe our algorithm based on the idea of the *communication matrix*.

To solve the communication problem, we first compute the adjacency matrix of the communication graph, i.e. we construct the matrix $A_{P\times P}$, where P is the number of processes, such that

$$a_{ij} = \begin{cases} 1 & \text{if process '}i\text{' sends a message to process '}j\text{'}, \\ 0 & \text{otherwise.} \end{cases}$$

Note that this matrix is also distributed among processes, i.e. each row is owned by a separate rank. Next, we compute

$$S_i = \sum_j a_{ij} \quad \text{and} \quad R_i = \sum_j a_{ji},$$

where S_i and R_i denote the number of messages sent and received, respectively. While S_i can be computed trivially, a reduction operation is required to compute R_i. For instance, this can be achieved using a single `MPI_Reduce_scatter` function call. The last phase of the interpolation procedure involves overlapping the computation of interpolated values for local points with the communication of data between processes. This is done by alternating between local calculations and probing for incoming messages from other processes. The interpolation is finished once the values for all local the points have been calculated and all the remote requests have been processed (see Algorithm 2).

We close this section by presenting the final semi-Lagrangian Algorithm 3 based on the interpolation Algorithm 2. The basic idea is to start from an initial guess G_0^{n+1} for the grid and modify it using the refinement (3) and coarsening (4) criteria until convergence is obtained. Various options are available for G_0^{n+1}. For instance it is possible to start from the macromesh and only perform refinement steps until convergence. This choice, however, is not suitable since the first few iterations do not contain many cells and there is little work for parallelism. Here we simply take the previous grid as the starting point, i.e. $G_0^{n+1} = G^n$. Note that this iterative process is essentially unavoidable since the grid is based on the values of the level-set function at t^{n+1}, which itself is unknown and is to be defined on G^{n+1}. Nonetheless the process converges to the final grid in at most $l_{\max} - l_{\min}$ steps where $l_{\min}$ and $l_{\max}$ denote the maximum and minimum depth of all trees in the forest, receptively.

3.3. Reinitialization

Successive application of Algorithm 3, especially for large values of the CFL number, eventually degrades the signed distance property of the level-set function. Thus, it is important to reinitialize the level-set function every few iterations, especially because the quality of generated grid heavily depends on the signed distance property. To achieve this property we solve the pseudo-time transient equation (2) using the discretization scheme detailed in [33]. For completeness, we briefly review the scheme. First, we write equation (2) in the following semi-discrete form:

$$\frac{\mathrm{d}\phi}{\mathrm{d}\tau} + S(\phi_0)\left(\mathcal{H}_G(D_i^+\phi, D_i^-\phi) - 1\right) = 0, \tag{9}$$

where $D_i^+\phi$ and $D_i^-\phi$ are the forward and backward derivatives in the x_i direction and $\mathcal{H}_G$ is the Godunov Hamiltonian defined as:

$$\mathcal{H}_G(a_i, b_i) = \begin{cases} \sqrt{\sum_i \max\left(|a_i^+|^2, |b_i^-|^2\right)} & \text{if } S(\phi_0) \le 0, \\ \\ \sqrt{\sum_i \max\left(|a_i^-|^2, |b_i^+|^2\right)} & \text{if } S(\phi_0) > 0, \end{cases}$$

where $a^+ = \max(a, 0)$ and $a^- = \min(a, 0)$. Similar to [33], equation (9) is integrated in time using the TVD-RK2 scheme with adaptive time-stepping in order to accelerate the convergence to the steady state. Generally it has been observed that adaptive time-stepping considerably improves the convergence rate and only a few iterations are needed if the signed-distance property is desired in a small band around the interface [33]. In this work, and based on previous findings, we use a fixed number of 20 iterations for the reinitialization equation. Of course, it should be noted that it is quite easy to define a custom tolerance as the termination criteria.

Since the computation is based on a local stencil, the parallel implementation of this scheme is mostly trivial. However, one minor point requires further explanation. As suggested in [33], one-sided derivatives $D_i^+\phi$ and $D_i^-\phi$ are computed using second order discretization which require to compute the second-order derivatives. To enable overlap between computation and communications when computing second-order derivatives and also integrating equation (9), we use the following common technique. First, we label all local points, L_p, as either private, P_p, or boundary, B_p. Here, the boundary points are the collection of all local points that are regarded as a ghost point, G_r, on at least one other process, i.e. $B_p = \bigcup_{r,\, r\neq p} G_r$. Private points are defined as the collection of all local points that are not a boundary point, i.e. $P_p = L_p \setminus B_p$. Algorithm 4 illustrates how this labeling can help with overlapping the computation and the communication associated to an arbitrary local operation $y \leftarrow \mathcal{F}(x)$. Note that the `p4est` library already includes all the primitives required for labeling local points without any further communication.

Algorithm 2 *values* ← Interpolate($H, F, \underline{\mathbf{X}}$): interpolate the value of F, defined on the local tree hierarchy H, at coordinates $\underline{\mathbf{X}}$. This is achieved in three phases as detailed in the text. I) First, interpolation points are buffered in two arrays, those than can be interpolated locally and those that should be sent to other processes. II) In the second phase, each process initiates the remote buffer exchange and computes the number of messages it should expect to receive using the communication matrix idea described in section 3.2. III) Finally, each process alternates between computing local interpolation and receiving and performing remote interpolation requests from other processes.

```
1:  col ← 0, local_buff ← null, remote_buff ← null                ▷ Phase I – buffering
2:  for p in X do
3:      [owners_rank, cell] ← H.search(p)                          ▷ search for the owner's rank and cell
4:      if owners_rank = mpirank then                              ▷ can interpolate locally
5:          local_buff.push_back(p, cell)
6:      else                                                       ▷ requires remote interpolation
7:          remote_buff[owners_rank].push_back(p)
8:          col[owners_rank] ← 1
9:      end if
10: end for
11:
12: for r in [0, mpisize) do                  ▷ Phase II – initiate communication and compute number of messages
13:     if col[r] == 1 then
14:         req ← MP_Isend(r, QUERY_TAG, remote_buff[r])           ▷ initiate a non-blocking send to process r
15:         query_requests.push_back(req)
16:     end if
17: end for
18: S ← sum(col)                                                   ▷ compute number of messages to send
19: R ← MPI_Reduce_scatter(col, MPI_SUM)                           ▷ compute number of messages to receive
20:
21: done ← false                                                   ▷ Phase III – interpolation
22: it ← local_buff.begin()
23: while !done do
24:     if it ≠ local_buff.end() then
25:         values ← process_local_interpolation(it)               ▷ process local interpolations
26:         ++it
27:     end if
28:     if R > 0 then                              ▷ search for interpolation query in the message queue
29:         message ← MPI_Iprobe(MPI_ANY_SOURCE, QUERY_TAG)
30:         if message.is_pending() then
31:             values ← process_remote_queries(message)           ▷ receive, search, and interpolate values
32:             req ← MPI_Isend(message.MPI_SOURCE, REPLY_TAG, values)   ▷ send back interpolated values
33:             reply_requests.push_back(req)
34:             R−−
35:         end if
36:     end if
37:     if S > 0 then                              ▷ search for interpolation reply in the message queue
38:         message ← MPI_Iprobe(MPI_ANY_SOURCE, REPLY_TAG)
39:         if message.is_pending() then
40:             values ← process_replies(message)                  ▷ receive remotely interpolated values
41:             S−−
42:         end if
43:     end if
44:     done ← S == 0 & R == 0 & it == local_buff.end()
45: end while
46: MPI_Waitall(query_requests, reply_requests)                    ▷ make sure all messages have been received
47: return values
```

3.4. Accuracy

The numerical methods detailed in the previous sections are widely used in the level-set community and their accuracy is studied for example in [33]. However, in order to validate our implementation, we present a brief convergence analysis.

The advection of an irregular boundary using the semi-Lagrangian method is the perfect candidate to demonstrate the accuracy of our implementation as it makes use of the interpolation routine as well as the reinitialization procedure. We select the benchmark problem proposed in [19]. Consider a sphere centered at (0.35, 0.35, 0.35) and with radius 0.15 in a domain $[0, 1]^3$ and deformed under the divergence free velocity field

$$\begin{cases} u(x,y,z) = 2\sin^2(\pi x)\sin(2\pi y)\sin(2\pi z), \\ v(x,y,z) = -\sin^2(\pi y)\sin(2\pi x)\sin(2\pi z), \\ w(x,y,z) = -\sin^2(\pi z)\sin(2\pi x)\sin(2\pi y). \end{cases} \tag{10}$$

forward in time until $t_{1/2} = 1$ and then backward to its original state at $t_f = 2$. We set the time step to $\Delta t = 5\Delta x_{min}$, where Δx_{min} is the size of the smallest cell in the forest. The level-set function is reinitialized at every time step by applying 20

Algorithm 3 $[G^{n+1}, \phi^{n+1}] \leftarrow$ SemiLagrangian$(G^n, \phi^n, \underline{\mathbf{u}}^n, \underline{\mathbf{u}}^{n-1}, \text{CFL})$: update ϕ^{n+1} from ϕ^n using a semi-Lagrangian scheme and construct the new forest G^{n+1} that is consistent with the zero level-set of ϕ^{n+1}.

```
1: $\Delta t_l \leftarrow \text{CFL} \times G^n$.hmin()/ max{$\underline{\mathbf{u}}^n$}
2: $\Delta t \leftarrow$ MPI_Allreduce($\Delta t_l$, MPI_MIN)        ▷ compute $\Delta t$ based on CFL condition across all processes
3: $H^n \leftarrow$ Reconstruct($G^n$)        ▷ using Algorithm 1
4: $G_0^{n+1} \leftarrow G^n$
5: while true do
6:     $\underline{\mathbf{X}}_d \leftarrow$ ComputeDeparturePoints($G_0^{n+1}, \underline{\mathbf{u}}^n, \underline{\mathbf{u}}^{n-1}, \Delta t$)        ▷ using equations (6)–(8)
7:     $\phi^{n+1} \leftarrow$ Interpolate($H^n, \phi^n, \underline{\mathbf{X}}_d$)        ▷ using Algorithm 2
8:     $G^{n+1} \leftarrow G_0^{n+1}$.refine_and_coarsen($\phi^{n+1}$)        ▷ using equations (3) and (4) as criteria
9:     if $G^{n+1} \neq G_0^{n+1}$ then
10:        $G^{n+1}$.partition()
11:        $G_0^{n+1} \leftarrow G^{n+1}$
12:    else
13:        break
14:    end if
15: end while
16: return $[G^{n+1}, \phi^{n+1}]$
```

Algorithm 4 $y \leftarrow$ Overlap$(x, \mathcal{F})$: compute $y_i = \mathcal{F}(x_i)$ for all nodes x_i, where $\mathcal{F}$ is a local operation, while hiding the communication to update the ghost layer.

```
1: for $x_i$ in $B_p$ do        ▷ I – perform computation on boundary points
2:     $y_i \leftarrow \mathcal{F}(x_i)$
3: end for
4: $send\_req \leftarrow$ MPI_Isend($y_B$)        ▷ II – begin updating ghost values
5: $recv\_req \leftarrow$ MPI_Irecv($y_G$)
6: for $x_i$ in $P_p$ do        ▷ III – perform computation on private points
7:     $y_i \leftarrow \mathcal{F}(x_i)$
8: end for
9: MPI_Waitall($send\_req, recv\_req$)        ▷ IV – wait for ghost update to finish
10: return $y$
```

Table 1
Study of the convergence of the level-set algorithm using Enright's test [19]. The L^∞ error on ϕ is computed only close to the interface as this is the relevant observable for an advection procedure.

Finest resolution	L^∞ error on ϕ	Rate	Volume loss (%)	Rate
128^3	1.53E-01	–	1.85E-01	–
256^3	1.18E-01	0.38	3.73E-02	2.31
512^3	9.82E-03	3.58	6.87E-03	2.44
1024^3	2.56E-03	1.94	1.92E-03	1.84
2048^3	1.60E-03	0.67	5.17E-04	1.89

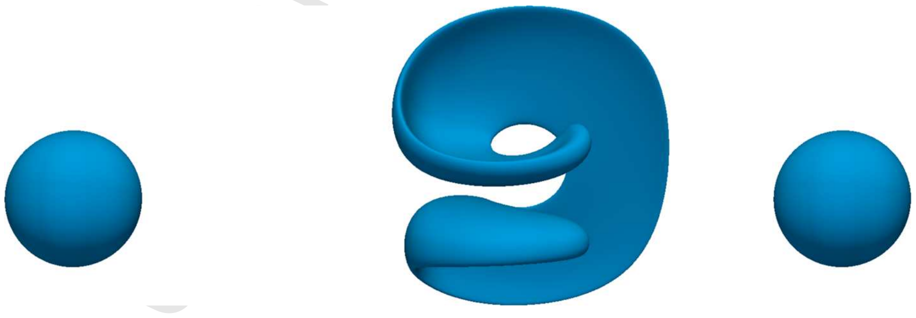


Fig. 4. Visualization of the deformation undergone by the sphere for the Enright's test at time 0, $t_{1/2}$ and t_f. The forest's finest resolution is equivalent to a uniform grid with 2048^3 cells and contains between 17 (at initial and final times) and 63 (at $t_{1/2}$) million nodes. These results were obtained on the Comet supercomputer using 960 processes (40 compute nodes).

iterations of the reinitialization procedure. We monitor the volume loss and the error in the interface location at the final time, when the original shape should be recovered, as the finest resolution of the forest increases. The results are reported in Table 1 and Fig. 4 shows a visualization of the sphere at the initial time, at $t_{1/2}$ when the deformation is maximal, and at the final time. The results are consistent with those reported previously in [33] and indicate second order accuracy for the mass loss.

Table 2
Parallel efficiency of the total runtime of the interpolation algorithm for the small (33 M nodes), large (280 M nodes), and very large (1.66 B nodes) tests. Reported efficiencies are based on the lowest number of processes for each test.

	P	16	32	64	128	256	512
Small Test	$\alpha = 5\%$	100%	101%	102%	106%	127%	165%
	$\alpha = 95\%$	100%	104%	110%	125%	127%	106%
	P	128	256	512	1024	2048	4096
Large Test	$\alpha = 5\%$	100%	89%	95%	101%	109%	114%
	$\alpha = 95\%$	100%	103%	108%	99%	67%	34%
	P	128	256	512	1024	2048	4096
Very Large Test	$\alpha = 50\%$	–	–	100%	108%	102%	78%
	$\alpha = 95\%$	–	–	100%	103%	94%	64%

4. Scaling results

In this section we present some results that demonstrate the scalability of our algorithms. Unless stated otherwise, all of our tests were ran on the Stampede cluster at the Texas Advanced Computing Center (TACC), a resource accessible through the Extreme Science and Engineering Discovery Environment (XSEDE) [53], where we are limited to 4096 cores at most. Each node of Stampede has 2 eight-core Xenon E5-2680 processes clocked at 2.7 GHz with 32 GB of DDR3-1600 MHz memory and interconnected using an InfiniBand network card. Unless mentioned otherwise, in all the tests we have used all 16 cores of every node. Finally, in all cases we report the maximum wall time recorded using PETSc's logging interface which has a temporal resolution of roughly 0.1 µs.

We define parallel efficiency as $e = s \cdot P_1/P$ where $s = t_1/t_P$ is the speed-up, P_1 is the smallest number of processes for which the test was run, t_1 is the time to run the problem on P_1 processes, P is the number of processes and t_P is the time to run the problem on P processes. We note that efficiencies larger than 100% are reported for some cases. This is not uncommon and can be hardware related, for instance linked to the problem being locally smaller for larger number of processes and thus exploiting the cache better.

4.1. Interpolation

In this section we show the results for a simple test to measure the scalability of the interpolation Algorithm 2. The test consists of interpolating a function at a number of random points on a randomly refined Octree in three spatial dimensions. We consider two cases, a small test on a level[1] 9 tree with roughly 33 M nodes and a larger test on a level 13 tree with roughly 280 M nodes. In both cases the number of randomly generated points is chosen to be equal to the number of nodes and the stabilized second-order interpolation of [33] is performed 10 times to smooth out possible timing fluctuations.

To simulate the effect of different CFL numbers, we generate the random points such that on each process α percentage of them are located outside the process boundary and thus will initiate communication. Scaling results are presented for $\alpha = 5\%$ and $\alpha = 95\%$ for both the small and large problems in Fig. 5. We also present a third row of results for a much larger problem with roughly 1.66 B nodes on a level 14 tree. Excellent scaling is obtained for the small problem for $P = 16$–512 even when 95% of the interpolation points belong to a remote process. For the larger problem, however, the communication overhead prevents the algorithm from scaling beyond 2048 processes when $\alpha = 95\%$ (cf. Table 2). Note, however, that this is expected since the total time is dominated by communication for $\alpha = 95\%$ and there is very little local work in this case. Indeed, the last row of Fig. 5 show much better scaling behavior on a larger problem size, e.g. efficiencies are increased from $e = 34\%$ to $e = 68\%$ for $\alpha = 95\%$ on 4096 processes. This is a typical result with strong scaling and simply implies that our algorithms are scalable for sufficiently large problems.

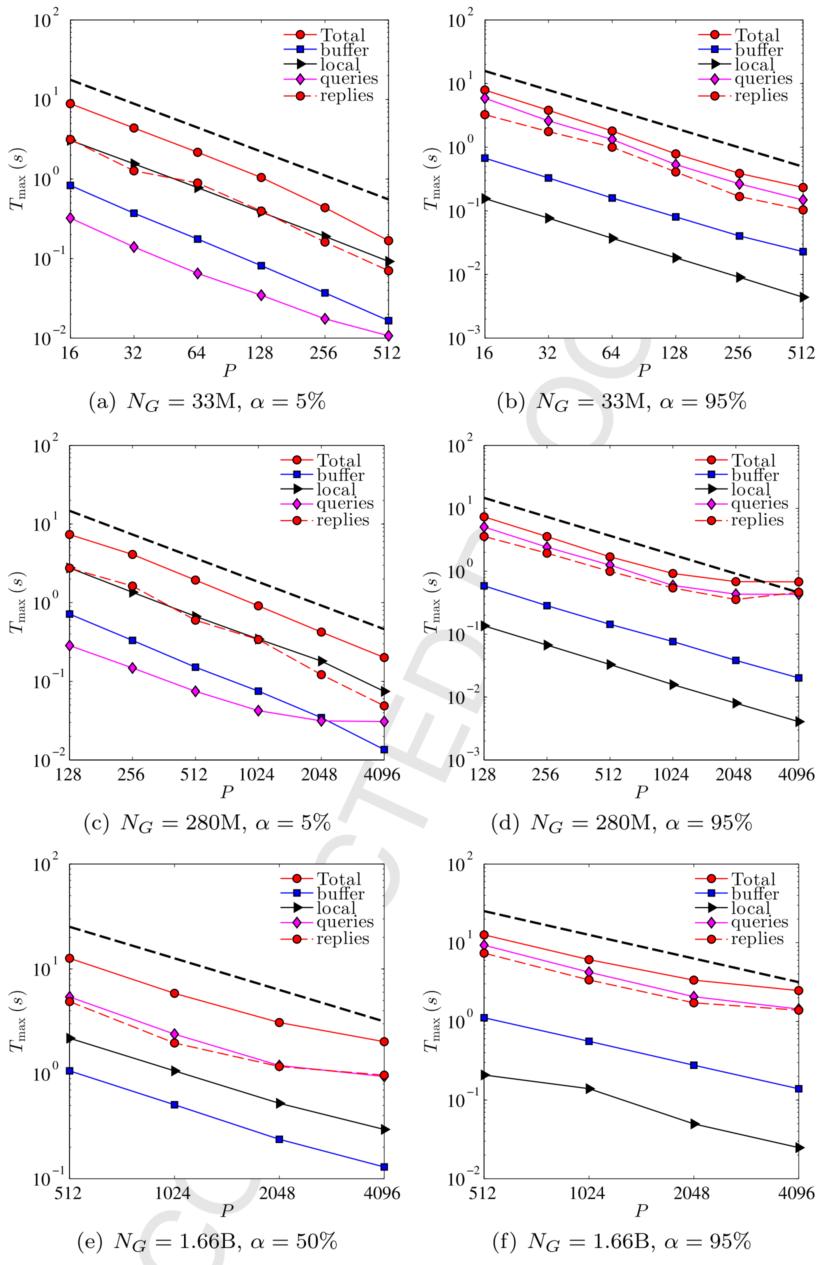
4.2. Semi-Lagrangian

To test the scalability of the semi-Lagrangian scheme of Algorithm 3, we consider a slightly modified version of the Enright's rotation test [19] presented in section 3.4, i.e. we advect a sphere of radius 0.35 located at $(0.4, 0.4, 0.4)$ with a divergence free velocity field given by equation (10).

To understand the effect of the CFL number on the scalability of the algorithm we perform one step of the semi-Lagrangian algorithm for $\mathrm{CFL} = 1$, $\mathrm{CFL} = 10$, and $\mathrm{CFL} = 100$. We also perform the test for two different initial girds, a small grid with maximum level $l_{\max} = 10$ and a large grid with maximum level $l_{\max} = 12$. In both cases, the minimum level is $l_{\min} = 0$. After one advection step, these grids have approximately 15 M and 255 M nodes, respectively.

Unlike many existing applications where the mesh is changed infrequently, our semi-Lagrangian algorithm requires several sub-iterations of the refinement and coarsening operations. As a result, it is expected that refinement and coarsening

[1] The level is the number of recursive splits allowed for each tree.



(a) $N_G = 33\text{M}$, $\alpha = 5\%$
(b) $N_G = 33\text{M}$, $\alpha = 95\%$
(c) $N_G = 280\text{M}$, $\alpha = 5\%$
(d) $N_G = 280\text{M}$, $\alpha = 95\%$
(e) $N_G = 1.66\text{B}$, $\alpha = 50\%$
(f) $N_G = 1.66\text{B}$, $\alpha = 95\%$

Fig. 5. Strong scaling of Algorithm 2 for several tests where N_G denotes the number of random interpolation points (which is the same as the number of nodes in the Octree) and α denotes the percentage of these points that are remote for each process. Here "Total" represents the total time spent in the interpolation while "buffer", "local", "queries", and "replies" represent the timing for different sections (cf. Algorithm 2). The black dashed line represents the ideal scaling. The results indicate excellent scaling for the small test (a–b) and for the large test when $\alpha = 5\%$ (c). For the extreme case (d) the algorithm stops scaling at 2048 processes due to communication overhead. Note, however, that this merely indicates that the problem size is not large enough for this test case. Indeed much better scaling is obtained when the problem size is increased to $N_G = 1.66$ B points (e–f).

steps constitute a significant portion of the total runtime which puts stringent scalability requirements on these algorithms. We refer the interested reader to section 3.2 of [13] for detailed description of scalable refinement and coarsening algorithms in `p4est`.

Table 3 illustrates the dependence of the number of sub-iterations required to build the grid on the CFL number; as the CFL is increased, the interface travels a farther distance, which necessitates more sub-iterations to generate the grid. Figs. 6 and 7 illustrate the scalability of the algorithm for the small and large problems, respectively. To enable meaningful comparisons between different CFL numbers and number of processes, the maximum time has been scaled by the number of sub-iterations required for the grid construction as reported in Table 3. For both problems, excellent scalability is observed

Table 3
Number of sub-iterations required for the grid construction in Algorithm 3 for the rotation test on a (a) level-10 and (b) level-12 Octree with approximately 15 M and 255 M nodes, respectively. Note how the sub-iteration count increases with the CFL number but is almost independent of the number of processes. The slight dependence between the number of sub-iterations and the number of processes is most likely due to the dependence of round-off errors on the number of processes. Nonetheless, close examination of the Octrees generated (data not shown) reveals that they are identical and independent of the number of processes used to perform the test.

CFL	#p					
	16	32	64	128	256	512
1	2	2	3	3	3	3
10	3	3	3	3	3	3
100	6	6	6	6	6	6

(a) $l_{\max} = 10$

CFL	#p					
	128	256	512	1024	2048	4096
1	3	3	3	3	3	3
10	3	3	4	4	4	4
100	6	6	6	6	6	7

(b) $l_{\max} = 12$

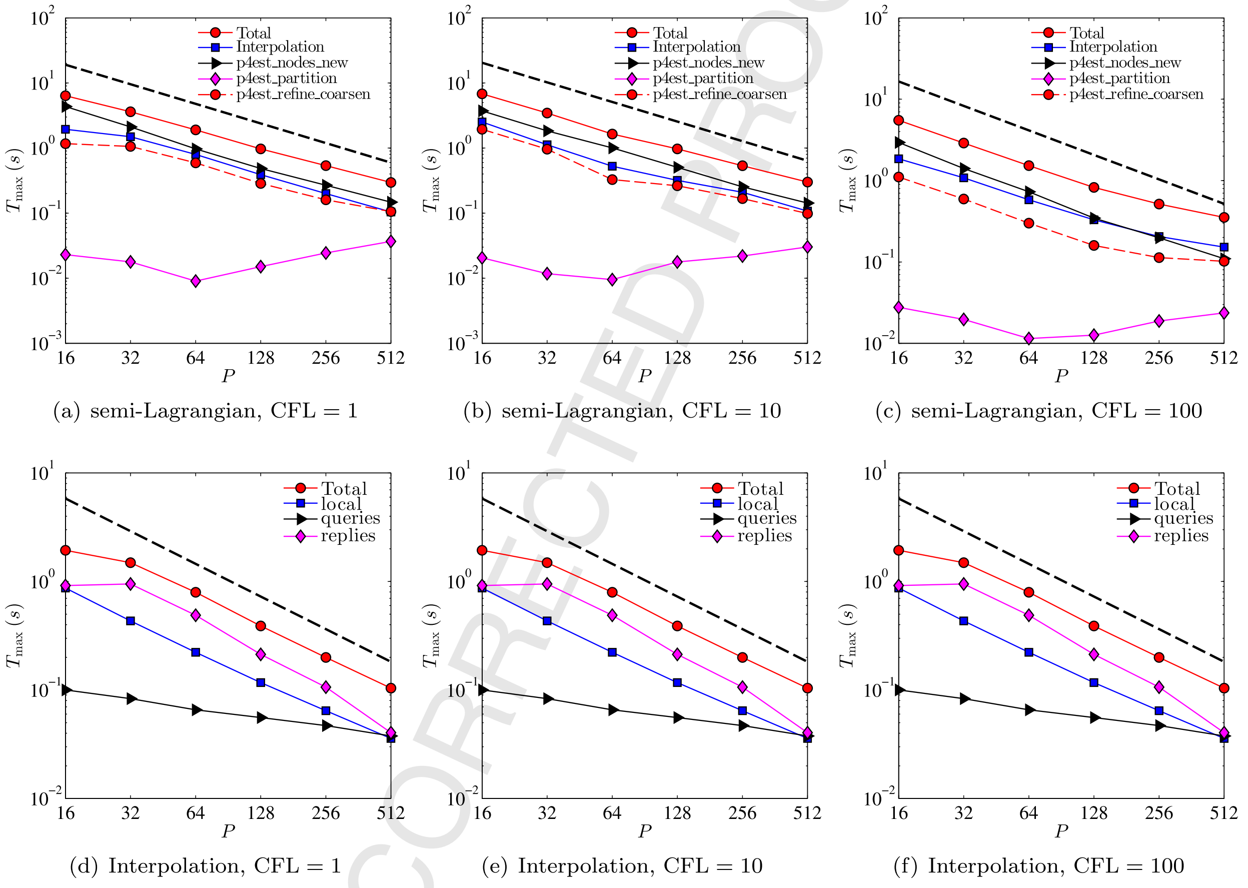


Fig. 6. Strong scaling of a single time step of Algorithm 3 for the rotation test on a level-10 Octree with approximately 15 M nodes. Top row: scaling of the various components of the algorithm for (a) CFL = 1, (b) CFL = 10, and (c) CFL = 100. Bottom row: breakdown of the various components of the interpolation phase for the same CFL numbers. The solid dashed line represents the ideal scaling. Note that the maximum time has been scaled by the number of sub-iterations required to build the tree (cf. Table 3). Here `p4est_nodes_new`, `p4est_partition` and `p4est_refine_coarsen` refer to constructing the global indexing for nodes, partitioning the forest, and the refining/coarsening operation, respectively [13].

for CFL = 1 and CFL = 10. The algorithm even shows good scalability when taken to the extreme, i.e. for CFL = 100 (cf. Table 4).

An increase in the CFL number has two effects on the algorithm. First, a larger fraction of the departure points lands in the domains of remote processes. Moreover, these points are potentially dispersed across a larger number of processes. This means that the communication volume should increase with the CFL number. Second, as more points are shipped to remote processes for interpolation, there is a greater chance that the interpolation load is imbalanced across processes. This is especially true for regions of space in which the streamlines cluster. Both factors can contribute to reducing the scalability of the algorithm at large CFL numbers.

To better understand the importance of the CFL number on the scalability, we have recorded a complete history of the communication pattern in the interpolation step. Fig. 8 illustrates the effects of the CFL number on different metrics,

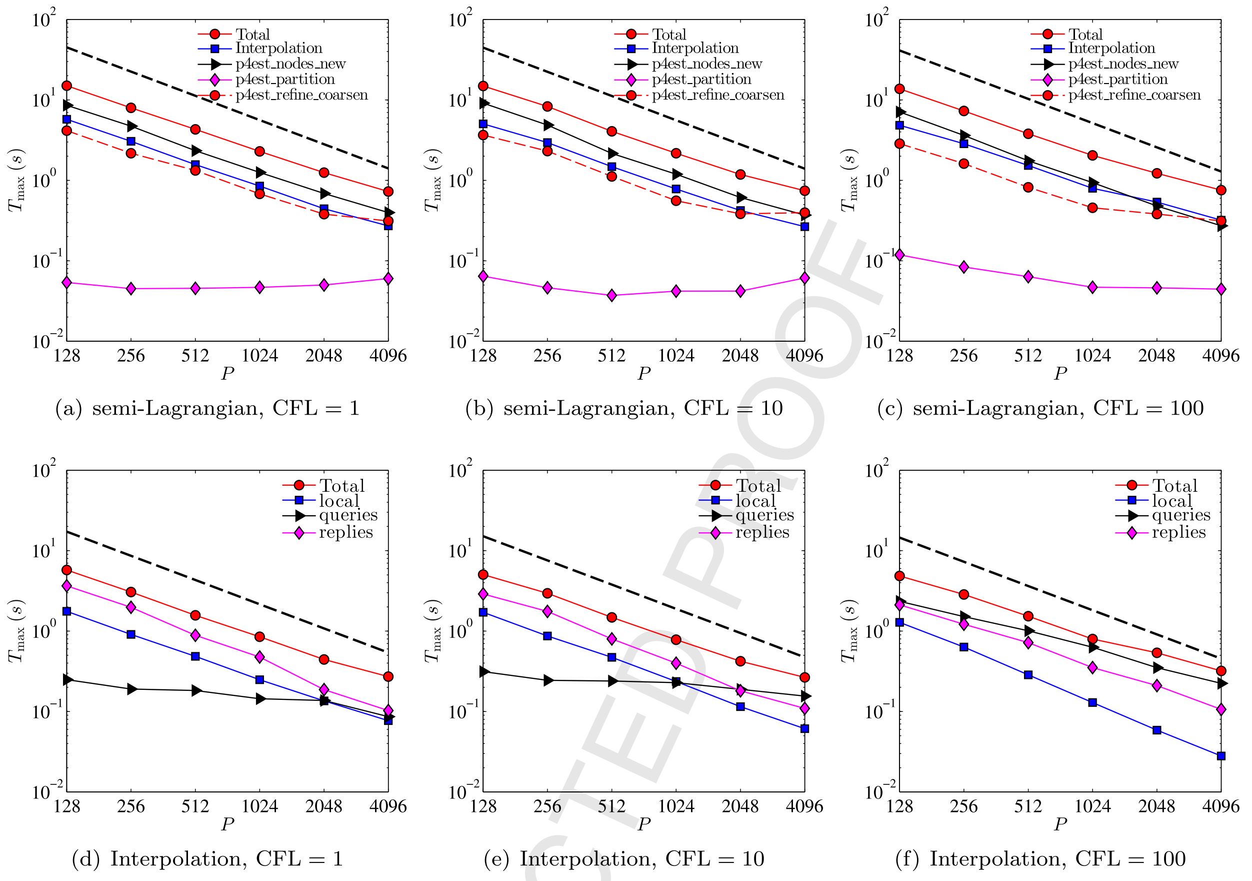


Fig. 7. Strong scaling of a single time step of Algorithm 3 for the rotation test on a level-12 Octree with approximately 255 M nodes. Top row: scaling of the various components of the algorithm for (a) CFL = 1, (b) CFL = 10, and (c) CFL = 100. Bottom row: breakdown of the various components of the interpolation phase for the same CFL numbers. The solid dashed line represents the ideal scaling. Note that the maximum time has been scaled by the number of sub-iterations required to build the tree (cf. Table 3).

Table 4
Parallel efficiency of the runtime of a single semi-Lagrangian step. Reported efficiencies are based on the lowest number of processes for each test.

	P	16	32	64	128	256	512
Small Test	CFL = 1	100%	88%	84%	82%	74%	67%
	CFL = 10	100%	99%	104%	88%	80%	71%
	CFL = 100	100%	95%	90%	84%	67%	49%
	P	128	256	512	1024	2048	4096
Large Test	CFL = 1	100%	94%	87%	82%	75%	65%
	CFL = 10	100%	90%	92%	86%	79%	63%
	CFL = 100	100%	94%	90%	84%	70%	57%

namely the number of interpolation points,[2] N_p, the number of sent and received messages, $N_m = S + R$, and the total communication volume, V_m in megabytes (MB), for $P = 4096$ processes. Furthermore, these values are reported for the first (top row) and last (bottom row) sub-iterations of the semi-Lagrangian algorithm. There are several points to make. First, increasing the CFL number greatly increases the load imbalance, as shown by the spread of the data in Fig. 8(a). This is because at higher CFL numbers, it is more likely that some processes will receive a larger portion of the backtracked points. Second, increasing the CFL number increases both the communication volume and its spread across processes (cf. Fig. 8(c)). Interestingly, however, the number of sent and received messages do not seem to be affected by the CFL number. The bottom row of Fig. 8 exhibits a better balance both in the computation and communication volume in the last sub-iteration of the semi-Lagrangian algorithm. This can be justified by noting that for the final sub-iteration, the partitioning of G^{n+1} is

[2] Note that this includes both the local points and the points queried by other processes.

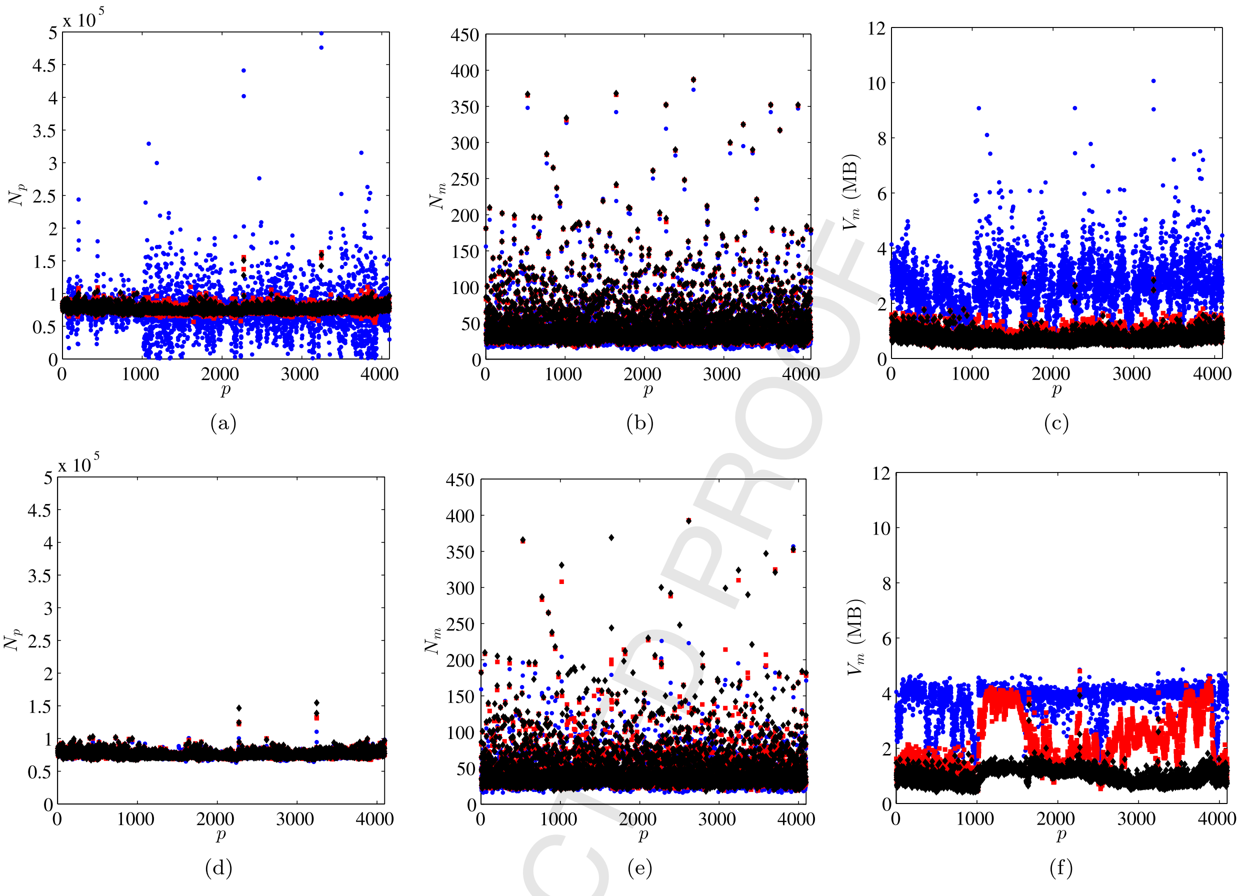


Fig. 8. Performance indicators of the first (top row) and last (bottom row) sub-iterations of the semi-Lagrangian algorithm for the level-set advection on 4096 processes with CFL = 1 (◆), CFL = 10 (🟥), and CFL = 100 (●). Increasing the CFL number causes load imbalance during interpolation (a) and increases the communication volume (c). However, the CFL number does not seem to appreciably affect the number of messages sent by the processes (b). During the last semi-Lagrangian sub-iteration, the initial grid G_0 (cf. Algorithm 3) is very close to the final grid. As a result, the load imbalance is considerably improved (d). Curiously, however, the communication pattern does not seem to be change much between first and last sub-iterations (e, f). (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

Table 5
Detailed load balancing and communication information for the advection test for CFL = 1, CFL = 10, and CFL = 100. Here N_p is the number of interpolation points, $N_m = S + R$ is the number of sent (S) and received (R) messages, and V_m is the total communication volume in megabytes (MB). Note how increasing the CFL number causes load imbalance and increases the communication volume while it does not affect the number of messages sent and received during a sub-iteration of the semi-Lagrangian step.

CFL	Metric	First sub-iteration				Last sub-iteration			
		min	max	avg	stddev	min	max	avg	stddev
1	N_p	6.67E+04	1.59E+05	7.57E+04	4.86E+03	6.69E+04	1.55E+05	7.57E+04	4.73E+03
	N_m	18	387	47.55	31.72	18	392	47.55	31.19
	V_m (MB)	4.01E-01	2.93E+00	7.25E-01	1.95E-01	4.13E-01	3.91E+00	9.68E-01	2.62E-01
	$T_{\max}$ (s)	6.96E-01				3.67E-01			
10	N_p	5.56E+04	1.63E+05	7.57E+04	6.07E+03	6.65E+04	1.33E+05	7.57E+04	4.50E+03
	N_m	17	387	46.73	31.78	19	393	46.68	27.93
	V_m (MB)	4.01E-01	3.06E+00	8.40E-01	2.21E-01	4.40E-01	4.80E+00	2.14E+00	9.88E-01
	$T_{\max}$ (s)	7.55E-01				3.30E-01			
100	N_p	8.28E+02	4.98E+05	7.57E+04	3.14E+04	6.30E+04	1.10E+05	7.57E+04	4.20E+03
	N_m	11	373	41.18	30.85	16	357	41.77	22.66
	V_m (MB)	5.28E-01	1.01E+01	2.65E+00	9.24E-01	9.76E-01	4.86E+00	3.78E+00	5.69E-01
	$T_{\max}$ (s)	9.25E-01				3.55E-01			

more consistent with the partitioning of the departure points on G^n. Detailed information about the load balancing and the communication patterns is listed in Table 5.

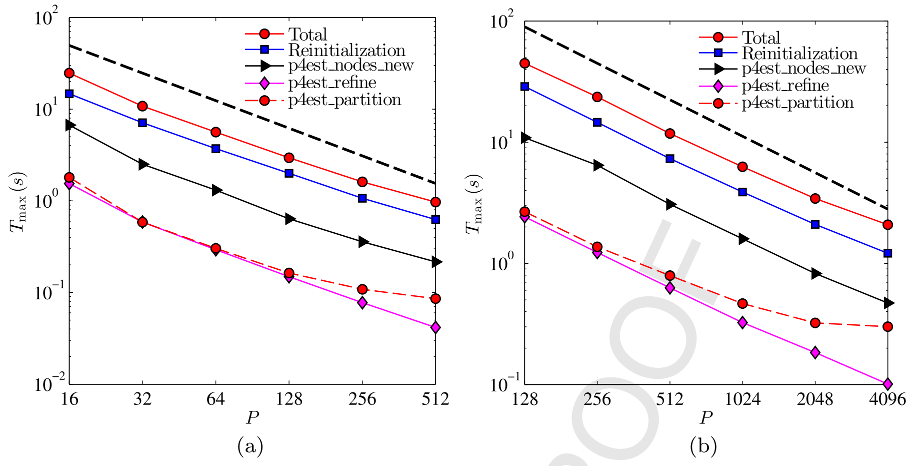


Fig. 9. Scalability of the reinitialization test for a small (left) and large (right) Octree with roughly 21 M and 337 M grid points, respectively. The black dashed line represents ideal scaling. Excellent results are obtained in both cases, illustrating the scalability of the overlapping strategy (cf. Algorithm 4).

Table 6
Parallel efficiency of the total runtime for the reinitialization test based on the lowest number of processes for each test.

Small Test	P	16	32	64	128	256	512
	e	100%	115%	110%	105%	96%	80%
Large Test	P	128	256	512	1024	2048	4096
	e	100%	95%	95%	89%	82%	67%

4.3. Reinitialization

Finally we present the scaling results of our parallel reinitialization algorithm where we extensively make use of Algorithm 4 for overlapping the computations with the communications when computing spatial derivatives. Our test consists in computing the signed distance function to a collection of 100 spheres, whose radii and centers are chosen randomly. The test is performed on a small, level-8 Octree with about 21 M and a larger, level-10 Octree with about 337 M grid points. In both cases the forest is built on a $3 \times 3 \times 3$ macro-mesh. Fig. 9 illustrates that our reinitialization algorithm, and in particular the overlapping strategy presented in Algorithm 4, scales very well (cf. Table 6). In general we expect similar scaling results for any local, finite-difference based calculations on Octrees that can efficiently utilize Algorithm 4.

5. Application to the Stefan problem

5.1. Presentation of the problem

In this section we apply our approach to the study of the phase transition of a liquid melt to a solid crystalline structure. In the case of a single component melt, and in the absence of convection, the process is dominated by diffusion and can be modeled as a Stefan problem. We decompose the computational domain Ω into two subdomains Ω_l and Ω_s, separated by an interface Γ. The Stefan problem describes the evolution of the temperature T, decomposed into T_s in the solid phase Ω_s and T_l in the liquid phase Ω_l, as

$$\frac{\partial T_l}{\partial t} = D_l \Delta T_l \quad \text{in } \Omega_l, \tag{11}$$

$$\frac{\partial T_s}{\partial t} = D_s \Delta T_s \quad \text{in } \Omega_s. \tag{12}$$

The diffusion constants D_l and D_s can be discontinuous across the interface. We prescribe homogeneous Neumann boundary conditions on the edge of the computational domain, $\nabla T \cdot \underline{\mathbf{n}}|_{\partial\Omega} = 0$. At the interface between the solid and the liquid phases, the temperature is given by the Gibbs–Tompson boundary condition [3,2]:

$$T_s = T_l = T_\Gamma = -\epsilon_c \kappa - \epsilon_v (\underline{\mathbf{u}} \cdot \underline{\mathbf{n}}), \tag{13}$$

where κ is the local interface curvature, $\underline{\mathbf{u}}$ is the velocity of the interface, $\underline{\mathbf{n}}$ is the outward normal to the solidification front and ϵ_c and ϵ_v are the surface tension and kinetic undercooling coefficients. The interface velocity $\underline{\mathbf{u}}$ is defined from the jump in the heat flux at the interface,

Algorithm 5 General procedure for solving the Stefan problem.

```
1: Initialize the forest and $\phi$ given the initial geometry.
2: Initialize $T_s$ in $\Omega^+$ and $T_l$ in $\Omega^-$.
3: Reinitialize $\phi$ and compute the local interface curvature $\kappa$.
4: Compute $T_l^{n+1}$ and $T_s^{n+1}$ by solving the heat equations (11) and (12).
5: Extrapolate $T_s^{n+1}$ from $\Omega^+$ to $\Omega^-$ and $T_l^{n+1}$ from $\Omega^-$ to $\Omega^+$.
6: Compute the velocity field $\underline{\mathbf{u}}$ according to (14).
7: Compute the time step dt following (15).
8: Evolve the interface and construct the new forest using the Semi-Lagrangian procedure.
9: Interpolate $T_s^{n+1}$ and $T_l^{n+1}$ from the old forest to the new forest.
10: Go to 3 with $n = n + 1$.
```

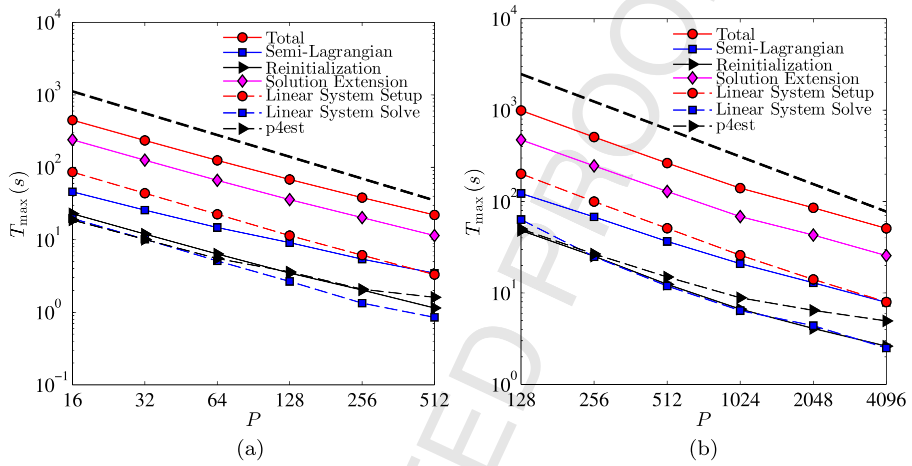


Fig. 10. Scalability of the Stefan problem for small (left) and large (right) Octrees with roughly 7 M and 105 M grid points, respectively. The solid dashed line represents perfect scaling. As expected from the scalability analysis of the individual components, we observe excellent results, illustrating the potential of our algorithms.

$$\underline{\mathbf{u}} \cdot \underline{\mathbf{n}} = -\left[D_l \frac{\partial T_l}{\partial \underline{\mathbf{n}}} - D_s \frac{\partial T_s}{\partial \underline{\mathbf{n}}} \right]. \tag{14}$$

We choose to use an adaptive time step with a CFL = 5, i.e.

$$\Delta t = 5\ \Delta x_{min}\ \min(1, 1/\max\|\underline{\mathbf{u}}\|), \tag{15}$$

where Δx_{min} is the size of the smallest cell of the forest. The general procedure to solve the Stefan problem is presented in Algorithm 5 and we refer the interested reader to [16] for the details of implementation. In implementing the numerical solver, we make use of the popular `PETSc` [5] library for linear algebra and its parallel primitives, such as parallel ghosted vector and scatter/gather operations, which simplifies the implementation.

5.2. Scalability

The implementation of the Stefan problem relies on the components described in the previous sections, and it is therefore a good synthesis of the performance of the various algorithms. We monitored the performance of the code over five time iterations, as presented in Algorithm 5, for two different maximum resolutions. In both cases, the forest is built on a $20 \times 20 \times 20$ macro-mesh. The maximum tree resolution for the small test is 9, leading to approximately 7 M grid points, and the maximum resolution for the large test is 11, corresponding to 105 M grid points. The level-set function is reinitialized at every time-step by applying 20 iterations of the reinitialization procedure. The results are presented in Fig. 10, where "Solution Extension" refers to extrapolation procedure (see Algorithm 5 step 5). As expected from the results obtained for each component in the previous sections, our implementation of the Stefan problem exhibits very satisfactory scaling (cf. Table 7).

5.3. Numerical experiments

We now present the results from a large simulation of the Stefan problem on a $20 \times 20 \times 20$ macro-mesh and with level-10 Octrees. The Gibbs–Tompson anisotropy undercooling coefficients in equation (13) are defined as

$$\epsilon_c = [\epsilon_1\,(1 + \alpha_1 \cos(3\theta_1)) + \epsilon_2\,(1 + \alpha_2 \cos(3\theta_2))]\,\kappa,$$
$$\epsilon_v = 0,$$

Table 7
Parallel efficiency of the total runtime for the Stefan test based on the lowest number of processes for each test.

Small Test	P	16	32	64	128	256	512
	e	100%	95%	90%	82%	73%	64%
Large Test	P	128	256	512	1024	2048	4096
	e	100%	98%	94%	89%	73%	61%

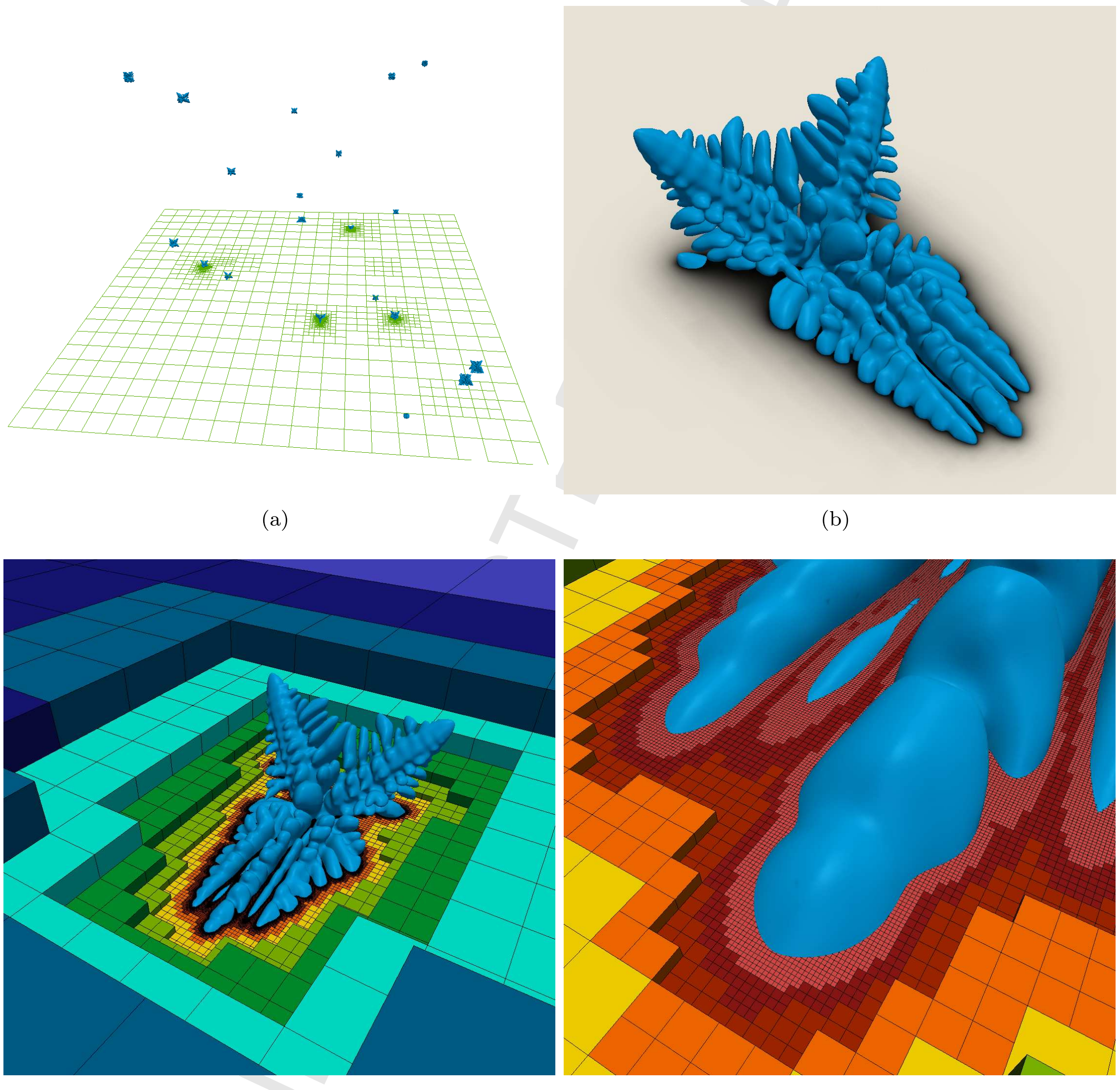


Fig. 11. Visualization of the computational mesh (a, c, d) and the temperature field (b) for the Stefan problem simulation.

with θ_1 the angle between the normal to the interface $\underline{\mathbf{n}}$ and the x-axis in the (x, y) plane and θ_2 the angle between $\underline{\mathbf{n}}$ and the x-axis in the (x, z) plane. The coefficients

$$\epsilon_1 = 2\,(\sin(x) + \cos(y) + 2) \cdot 10^{-6}, \qquad \epsilon_2 = 2\,(\sin(x) + \cos(z) + 2) \cdot 10^{-6},$$

$$\alpha_1 = \frac{1}{4}(\cos(x) + \sin(y) + 2), \qquad \alpha_2 = \frac{1}{4}(\cos(x) + \sin(z) + 2),$$

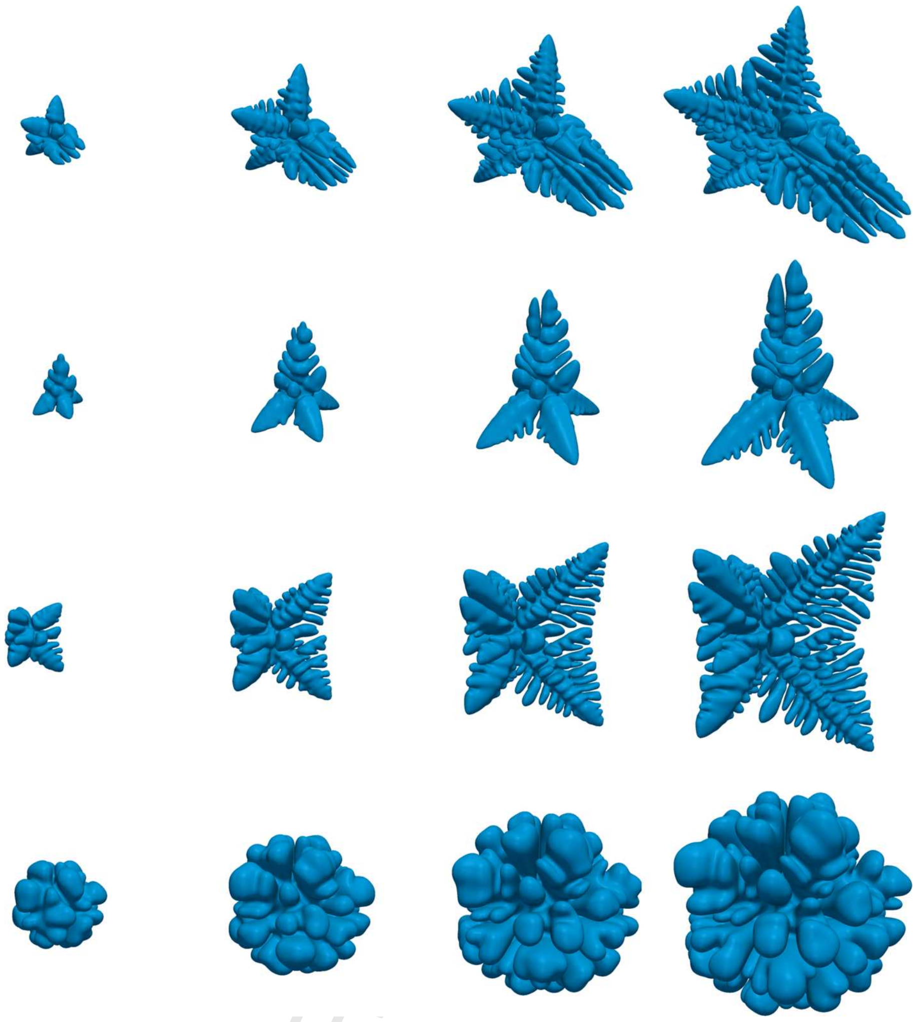


Fig. 12. Time evolution of four of the crystals obtained for the Stefan problem simulation. The snapshots represent, from left to right, iterations 96, 196, 296 and 396.

are used to enforce a variety of crystal shapes. The computation is initialized with twenty spherical seeds of radius $1.5 \cdot 10^{-3}$ placed randomly in the domain. We take the diffusion coefficients $D_s = D_l = 1$ and set the initial temperatures $T_l^0 = -0.25$ and $T_s^0 = 0$.

The simulation was ran on 256 MPI processes for 6 hours and 30 minutes, resulting in 396 time iterations. Visualizations of the final iteration are presented in Figs. 11 and 12. The final iteration of the simulation consisted of 167 M grid points whereas a uniform grid with the equivalent finest resolution would lead to $8.59 \cdot 10^{12}$ grid points, i.e. over eight trillion grid points. Our simulation used only 0.002% of the number of grid points needed for the same simulation on a uniform grid. This application demonstrates the ability of our approach to resolve small scale details, while accounting for long range interactions.

6. Conclusions

In this article we have presented parallel algorithms related to the level-set technology on adaptive Quadtree and Octree grids using a domain decomposition approach. These algorithms are implemented using a combination of `MPI` and the open-source `p4est` library. In order to preserve the unconditional stability property of the semi-Lagrangian scheme while enabling scalable computations, we introduced an asynchronous interpolation algorithm using non-blocking point-to-point communications, and demonstrated its scalability.

In particular we showed that the scalability of the semi-Lagrangian algorithm, depends on the CFL number. Great scalability is observed for intermediate CFL numbers, e.g. CFL ~ 10. At higher CFL numbers, however, the departure points are potentially further dispersed across processors, which limits the scalability. This is because the domain decomposition technique used here is based on the Z-ordering of cells and does not take the velocity field information into account. A possible remedy for this problem could be assigning weights to cells based on some estimate of the grid structure after one step of the advection algorithm, e.g. by using a forward-in-time integration of grid points. Such an estimate could also reduce the number of semi-Lagrangian iterations. These ideas are postponed for further investigations. We have also presented a simple parallelization technique for the reinitialization algorithm based on the pseudo-time transient formulation. Both the semi-Lagrangian and the reinitialization algorithms show good scalability up to 4096 processors, the current limit of our account.

Finally, an application of these algorithms is presented in modeling the solidification process by solving a Stefan problem. This application clearly illustrates the applicability of our algorithms to complex multi-scale problems that cannot be treated practically using the domain decomposition techniques on uniform grids. We believe that our findings can serve as a basis to simulating a wide range of multi-scale and free boundary problems.

Acknowledgements

The funding for this project was provided by ONR N00014-11-1-0027. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575. The authors acknowledge the Texas Advanced Computing Center (TACC) at the University of Texas at Austin and the San Diego Supercomputing Center (SDSC) at the University of California San Diego for providing HPC resources that have contributed to the research results reported within this paper. The third author acknowledges support by the Hausdorff Center for Mathematics (HCM) at Bonn University, an initiative funded by the German Research Foundation (DFG). Finally, we would like to thank the developer community of PETSc library for their valuable suggestions during this project.

References

[1] D. Adalsteinsson, J. Sethian, A fast level set method for propagating interfaces, J. Comput. Phys. 118 (1995) 269–277.
[2] V. Alexiades, A.D. Solomon, Mathematical Modeling of Melting and Freezing Processes, Hemisphere, Washington, DC, 1993.
[3] V. Alexiades, A.D. Solomon, D.G. Wilson, The formation of a solid nucleus in supercooled liquid. I, J. Non-Equilib. Thermodyn. 13 (1988) 281–300.
[4] Srinivas Aluru, F. Sevilgen, Parallel domain decomposition and load balancing using space-filling curves, in: Proceedings of Fourth International Conference on High-Performance Computing, 1997, IEEE, 1997, pp. 230–235.
[5] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Hong Zhang, PETSc Web page, http://www.mcs.anl.gov/petsc, 2014.
[6] W. Bangerth, R. Hartmann, G. Kanschat, deal.II – a general purpose object oriented finite element library, ACM Trans. Math. Softw. 33 (4) (2007) 24/1–24/27.
[7] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, Martin Kronbichler, Algorithms and data structures for massively parallel generic adaptive finite element codes, ACM Trans. Math. Softw. 38 (2) (2011) 14.
[8] Erik G. Boman, Ümit V. Çatalyürek, Cédric Chevalier, Karen D. Devine, The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: partitioning, ordering and coloring, Sci. Program. 20 (2) (2012) 129–150.
[9] Michael Breuß, Emiliano Cristiani, Pascal Gwosdek, Oliver Vogel, An adaptive domain-decomposition technique for parallelization of the fast marching method, Appl. Math. Comput. 218 (1) (2011) 32–44.
[10] Emmanuel Brun, Arthur Guittet, Frederic Gibou, A local level-set method using a hash table data structure, J. Comp. Physiol. 231 (2012) 2528–2536.
[11] C. Burstedde, p4est: parallel adaptive mesh refinement on forests of octrees, http://www.p4est.org/, June 19, 2015, last accessed.
[12] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Georg Stadler, Eh Tan, Tiankai Tu, Lucas C. Wilcox, Shijie Zhong, Scalable adaptive mantle convection simulation on petascale supercomputers, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE Press, 2008, p. 62.
[13] Carsten Burstedde, Lucas C. Wilcox, Omar Ghattas, p4est: scalable algorithms for parallel adaptive mesh refinement on forests of octrees, SIAM J. Sci. Comput. 33 (3) (2011) 1103–1133.
[14] Paul M. Campbell, Karen D. Devine, Joseph E. Flaherty, Luis G. Gervasio, James D. Teresco, Dynamic octree load balancing using space-filling curves, Tech. Rep. CS-03-01, Williams College Department of Computer Science, 2003.
[15] Adam Chacon, Alexander Vladimirsky, A parallel heap-cell method for Eikonal equations, arXiv preprint arXiv:1306.4743, 2013.
[16] Han Chen, Chohong Min, Frederic Gibou, A numerical scheme for the Stefan problem on adaptive Cartesian grids with supralinear convergence rate, J. Comput. Phys. 228 (16) (2009) 5803–5818.
[17] Miles Detrixhe, Frédéric Gibou, Chohong Min, A parallel fast sweeping method for the Eikonal equation, J. Comput. Phys. 237 (March 2013) 46–55.
[18] John Drake, Ian Foster, John Michalakes, Brian Toonen, Patrick Worley, Design and performance of a scalable parallel community climate model, Parallel Comput. 21 (10) (1995) 1571–1591.
[19] D. Enright, R. Fedkiw, J. Ferziger, I. Mitchell, A hybrid particle level set method for improved interface capturing, J. Comput. Phys. 183 (2002) 83–116.
[20] Oliver Fortmeier, H. Martin Bücker, A parallel strategy for a level set simulation of droplets moving in a liquid medium, in: High Performance Computing for Computational Science–VECPAR 2010, Springer, 2011, pp. 200–209.
[21] Michael Griebel, Gerhard W. Zumbusch, Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves, Parallel Comput. 25 (1999) 827–843.
[22] M. Herrmann, A domain decomposition parallelization of the fast marching method, Technical report, DTIC Document, 2003.
[23] Marcus Herrmann, A parallel Eulerian interface tracking/Lagrangian point particle multi-scale coupling procedure, J. Comput. Phys. 229 (3) (2010) 745–759.
[24] Torsten Hoefler, Christian Siebert, Andrew Lumsdaine, Scalable communication protocols for dynamic sparse data exchange, ACM SIGPLAN Not. 45 (5) (2010) 159–168.
[25] Tobin Isaac, Carsten Burstedde, Omar Ghattas, Low-cost parallel algorithms for 2:1 octree balance, in: Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, IEEE, 2012, pp. 426–437.
[26] Tobin Isaac, Carsten Burstedde, Lucas C. Wilcox, Omar Ghattas, Recursive algorithms for distributed forests of octrees, SIAM J. Sci. Comput. 37 (5) (2015) C497–C531.
[27] W.K. Jeong, R.T. Whitaker, A fast iterative method for Eikonal equations, SIAM J. Sci. Comput. 30 (5) (2008) 2512–2534.
[28] D. Juric, A front-tracking method for dendritic solidification, J. Comput. Phys. 123 (1) (January 1996) 127–148.
[29] George Karypis, Vipin Kumar, METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0, 1995.

[30] George Karypis, Vipin Kumar, A parallel algorithm for multilevel graph partitioning and sparse matrix ordering, J. Parallel Distrib. Comput. 48 (1) (1998) 71–95.
[31] F. Losasso, F. Gibou, R. Fedkiw, Simulating water and smoke with an octree data structure, in: ACM Trans. Graph., SIGGRAPH Proc., 2004, pp. 457–462.
[32] Frank Losasso, Ron Fedkiw, Stanley Osher, Spatially adaptive techniques for level set methods and incompressible flow, Comput. Fluids 35 (2006) 995–1010.
[33] C. Min, F. Gibou, A second order accurate level set method on non-graded adaptive Cartesian grids, J. Comput. Phys. 225 (2007) 300–321.
[34] Andreas Müller, Michal A. Kopera, Simone Marras, Lucas C. Wilcox, Tobin Isaac, Francis X. Giraldo, Strong scaling for numerical weather prediction at petascale with the atmospheric model NUMA, http://arxiv.org/abs/1511.01561, 2015.
[35] S. Osher, R. Fedkiw, Level Set Methods and Dynamic Implicit Surfaces, Springer-Verlag, New York, NY, 2002.
[36] S. Osher, R.P. Fedkiw, Level set methods: an overview and some recent results, J. Comput. Phys. 169 (2001) 463–502.
[37] S. Osher, J.A. Sethian, Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations, J. Comput. Phys. 79 (1988) 12–49.
[38] S. Popinet, Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries, J. Comput. Phys. 190 (2003) 572–600.
[39] Joseph M. Rodriguez, Onkar Sahni, Richard T. Lahey Jr, Kenneth E. Jansen, A parallel adaptive mesh method for the numerical simulation of multiphase flows, Comput. Fluids 87 (2013) 115–131.
[40] Johann Rudi, A. Cristiano I. Malossi, Tobin Isaac, Georg Stadler, Michael Gurnis, Peter W.J. Staar, Yves Ineichen, Costas Bekas, Alessandro Curioni, Omar Ghattas, An extreme-scale implicit solver for complex PDEs: highly heterogeneous flow in Earth's mantle, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2015, p. 5.
[41] O. Sahni, M. Zhou, M.S. Shephard, K.E. Jansen, Scalable implicit finite element solver for massively parallel processing with demonstration to 160 K cores, in: SC09: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2009.
[42] H. Samet, Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS, Addison-Wesley, New York, 1990.
[43] Rahul S. Sampath, Santi S. Adavani, Hari Sundar, Ilya Lashuk, George Biros, Dendro: parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE Press, 2008, p. 18.
[44] Rahul S. Sampath, George Biros, A parallel geometric multigrid method for finite elements on octree meshes, SIAM J. Sci. Comput. 32 (3) (2010) 1361–1392.
[45] J. Sethian, A fast marching level set method for monotonically advancing fronts, Proc. Natl. Acad. Sci. 93 (1996) 1591–1595.
[46] J.A. Sethian, Level Set Methods and Fast Marching Methods, Cambridge University Press, 1999.
[47] James R. Stewart, H. Carter Edwards, A framework approach for developing parallel adaptive multiphysics applications, Finite Elem. Anal. Des. 40 (12) (2004) 1599–1617.
[48] J. Strain, Tree methods for moving interfaces, J. Comput. Phys. 151 (1999) 616–648.
[49] M. Sussman, P. Smereka, S. Osher, A level set approach for computing solutions to incompressible two-phase flow, J. Comput. Phys. 114 (1994) 146–159.
[50] Mark Sussman, A parallelized, adaptive algorithm for multiphase flows in general geometries, Comput. Struct. 83 (6) (2005) 435–444.
[51] M. Theillard, F. Gibou, T. Pollock, A sharp computational method for the simulation of the solidification of binary alloys, J. Sci. Comput. (2014).
[52] S. Thomas, J. Côté, Massively parallel semi-Lagrangian advection, Simul. Pract. Theory 3 (4) (1995) 223–238.
[53] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, Nancy Wilkens-Diehr, Xsede: accelerating scientific discovery, Comput. Sci. Eng. 16 (5) (2014) 62–74.
[54] G. Tryggvason, B. Bunner, A. Esmaeeli, D. Juric, N. Al-Rawahi, W. Tauber, J. Han, S. Nas, Y.-J. Jan, A front-tracking method for the computations of multiphase flow, J. Comput. Phys. 169 (2001) 708–759.
[55] Tiankai Tu, David R. O'Hallaron, Omar Ghattas, Scalable parallel octree meshing for terascale applications, in: SC'05: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, IEEE, 2005.
[56] Maria Cristina Tugurlan, Fast marching methods-parallel implementation and analysis, PhD thesis, Louisiana State University, 2008.
[57] Kai Wang, Anthony Chang, Laxmikant V. Kale, Jonathan A. Dantzig, Parallelization of a level set method for simulating dendritic growth, J. Parallel Distrib. Comput. 66 (11) (2006) 1379–1386.
[58] J.B. White III, Jack J. Dongarra, High-performance high-resolution semi-Lagrangian tracer transport on a sphere, J. Comput. Phys. 230 (17) (2011) 6778–6799.
[59] Hongkai Zhao, A fast sweeping method for Eikonal equations, Math. Comput. 74 (250) (2005) 603–627.
[60] Hongkai Zhao, Parallel implementations of the fast sweeping method, J. Comput. Math. 25 (2007) 421–429.

Sponsor names

Do not correct this page. Please mark corrections to sponsor names and grant numbers in the main text.

ONR, *country*=United States, *grants*=N00014-11-1-0027

National Science Foundation, *country*=United States, *grants*=ACI-1053575

German Research Foundation, *country*=Germany, *grants*=