Tree-based Adaptive Level-set Methods on Highly Parallel Machines

Mohammad Mirzadeh[a,*], Arthur Guittet[a], Carsten Burstedde[c], Frédéric 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, University of Bonn, Bonn 53115, Germany.*

Abstract

Abstract will be filled in after finalization of the paper.

Keywords: Level-set Method, Adaptive Grids, Parallel Computing, Oct-/Quadtree datastructures

1. Introduction

The level-set method, originally proposed by Sethian and Osher [27], is a popular and powerful framework for tracking arbitrary interfaces that undergo complicated topological changes. As a result, the level-set method has wide range of application such as multiphase flows, moving boundary problems, image segmentation, and computer graphics graphics [25, 33]. An important feature which makes this method powerful and easy to use is that the location of 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 [21, 39], level-set method is typically less accurate and mass conservation could be a problem; although progress has been made in resolving this issue [16]. Second, the level-set function has to be defined in a higher dimensional space compared to the interface. If only the location of interface is needed, the added dimension greatly increases the overall computational cost. 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 [2] or, more recently, by using a hash table to restrict both computation and storage requirements [9].

Another approach that can address both problems is the use of local grid refinement. In [34] the idea of using tree-based grids for level-set calculations was first introduced and later extended in [23] for graphics application. More recently, authors in [24] proposed second-order accurate level-set methods on Quadtree (two dimensions) and Octree (three dimensions) grids. Use of tree-based adaptive grids in the context of level-set method is quite advantageous because: 1) It gives fine-grain control over error, which typically occurs close to the interface and 2) It can effectively reduce the dimensionality of the problem by focusing most of the grid cells close to the interface. Fortunately, construction of the tree is quite simple in the presence of an interface that naturally defines an ideal metric for refinement. Although 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 [37], could take a very long time in serial. In this paper we extend these algorithms by proposing parallel algorithms for distributed memory machines using domain decomposition technique.

One of the main challenges in parallelizing level-set algorithms on adaptive grids is handling the grid itself. One option is to replicate the entire grid on each processor and use off-the-shelf graph partitioners, e.g. ParMetis [22] or Zoltan [7], for load balancing and domain decomposition. For instance, this was the approach originally taken by the `deal.II` library [5]. This approach, however, is only scalable to a few hundred processor at best and is limited by the size of the grid itself. Moreover, use of a general-purpose graph partitioner adds extra overhead that can limit the overall scalability even further. Interestingly, tree-based grids have nice spatial ordering that naturally lead to the concept of space-filling curves (SFCs) which can be efficiently exploited for parallel load balancing [3, 12].

[*]Corresponding author: m.mirzadeh@engineering.ucsb.edu

The idea of using SFCs for parallel partitioning of Quad-/Octrees is not new in itself and has been tried by many researchers. For instance, `Octor` [40] 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 cores [10]. `Dendro` [30] is another example of an octree code in which similar ideas are used for parallel partitioning and development of a parallel geometric multigrid that has been scaled up to about 32,000 cores [31]. More recently, authors in [11] extended these ideas to a collection, or a "forest", of octrees that are connected through a common, potentially unstructured, hexahedral grid. This forest is then partitioned in parallel using a global Morton curve. Implementation of these algorithms, which were shown to scale to more than 200,000 processors, are publicly available through simple API provided by the `p4est` library [1]. In fact the algorithms presented in this paper are directly implemented on top of `p4est` library and we do not discuss any algorithm that is already covered in [11]. We also make use of the popular `PETSc` [4] library for linear algebra and its parallel primitives, such as parallel ghosted vector and scatter/gather operations, which simplifies the implementation.

Parallel level-set algorithms can be categorized in two groups: 1) Parallel advection algorithms and 2) 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 before [24]. 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 processors that are potentially far away. This requires a very dynamic and nonuniform communication pattern which is complicated to implement. For adaptive grids situation is even more complicated due to the asymmetric nature of communication (c.f. 3). Second, load balancing could be an issue for large CFL numbers and nonuniform velocity fields due to clustering of departure points and can 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 semi-Lagrangian algorithm in the first place.

Nonetheless, several parallel semi-Lagrangian algorithms have been proposed in the past. A simple domain decomposition technique was used in [38] where the width of ghost layer is fixed based on the maximum CFL number to ensure all departure points are covered by the ghost layer. At large CFL numbers, this leads to large communication volume that can limit the scalability. Nonetheless good scaling was report for small CFL numbers (CFL ≤ 2). In [15] 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, however, this approach seems to suffer from excessive communication overhead at larger number of processors as well. More recently, authors in [43] used a domain decomposition strategy on a cubed sphere but with a single layer of ghost nodes. Interpolation on remote processors is then handled by sending query points to the corresponding processor 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. Although in this article 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 has been proposed with applications in modeling dendritic crystal growth [42], multiphase flows [35, 17, 28], and atomization process [19].

In many applications of the level-set method, it is desirable for the level-set function to be a signed distance function, i.e. $|\nabla\phi| = 1$. There are generally two approaches to enforce this property: 1) Solving the pseudo-time transient reinitialization equation [36, 25]:

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

or 2) 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 in a domain decomposition approach. Moreover, only a few iterations may be needed if the signed-distance property is only required close to the interface [24].

This is the approach we have chosen in this paper. If, however, the signed-distance property is required in the entire domain solving the Eikonal equation is more computationally efficient. Unfortunately, however, the most popular algorithm for solving the Eikonal equation, i.e. the Fast Marching Method [32, 33], is inherently sequential due to causal relationship between grid points and cannot be easily parallelized. The Fast Sweeping Method (FSM) [44] is an alternative method 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 [18] where a domain decomposition algorithm was introduced. Unlike the serial FMM, however, parallel FMM potentially requires multiple iteration or “rollback operations” to enforce the causality across processors. Similar ideas are described in details in [41]. It should be noted that number of iterations for the parallel FMM to converge greatly depends on the complexity of the interface and also the parallel partitioning and, in general, fewer iterations are required if domains are aligned with the normals to the interface. Due to the nature of Eikonal equation, shared memory machines might be a better environment for parallelization. For instance, in [8] authors use an “adaptive” technique where individual threads implicitly define a domain decomposition at runtime. Unfortunately, however, this approach does not seem to be more effective than a simple static decomposition. In [45] a parallel FSM method was presented for the fist time. However, a more scalable FSM was more recently proposed in [14] where the Cuthill-McKee numbering was utilize to expose more parallelism. More recently a two-scale, hybrid FMM-FSM was presented in [13] which, albeit being more complicated to implement, promises even better scalability. Finally, a parallel Fast Iterative Method (FIM) was proposed in [20]. The FIM is similar to FMM in that it also maintains a list of “active nodes”. Unlike FMM, however, FIM avoids sorting the list and allows for concurrent updating of all nodes in an iterative fashion.

The rest of 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 using a domain decomposition method. In section **??** we provide several examples that illustrate the scalability of our algorithms. Finally, we close by providing an important application of our algorithms in modeling the solidification process by solving a Stefan problem in **??**.

2. The level-set method

The level-set method, introduced in [27], 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 dimension can be described as $\Gamma = \{(x, y)|\phi(x, y) = 0\}$ where $\phi(x, y)$ is the level-set function. Evolution of the curve under a velocity field $\mathbf{V}$ is then obtained by solving the level-set equation:

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

When 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 regular finite difference method is its unconditional stability which allows for arbitrarily large time steps. This is specially of interest when using adaptive grids that allow for higher grid resolutions.

In general, an infinite number of level-set functions can describe the same zero contour and thus interface. However, it is desirable to chose functions with signed distance property, i.e. $|\nabla\phi| = 1$. As detailed in section 1, we solve the pseudo-time transient reinitialization equation [36, 25] to achieve this property,

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

where ϕ_0 is any level-set function that correctly describes the interface location and $S(\phi_0)$ is an appropriate approximation of the sign function [26]. Here, we do not go into the details of the sequential algorithms for solving equations (1) and (2). Instead, we note that the algorithms presented in section 3 are based on the sequential methods presented in [24] and refer the interested reader to the aforementioned article and references therein for more details.

3. Parallel algorithms

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 [34]. Moreover, adaptive tree-based grids are easy to generate in the presence of a signed-distance level-set function [24] and can efficiently be encoded using a tree datastructure [29]. To develope scalable parallel algorithms on these grids, it is necessary to parallelize the datastructure and grid manipulations method such as refinement and coarsening of cells as well as provide with a fast method for grid partitioning and load balancing. `p4est` library [1] is a collection of such parallel algorithms that has recently emerged and shown to scale up to 200,000 processors [11].

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 (c.f. figure 1). This common coarse grid, which we will refer to as the "macromesh", can in general be an unstructured quadrilateral, in two spatial dimensions, or hexahedral mesh, in three spatial dimensions. In this article, however, we shall limit ourselves to simple uniform Cartesian macromeshes. Moreover it is implicitly assumed that the macromesh is small enough that can be entirely replicated on all processors. For instance, in many of the applications that we are interested in this paper the macromesh simply a single 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:

$$\min_{\forall v \in V(C)} |\phi| \leq \frac{LD}{2}. \tag{3}$$

Conversely an existing cell is marked for coarsening if:

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

Here $V(C)$ 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. We refer to section 3.2 of [11] for details on parallel refinement and coarsening algorithms used in `p4est`.

Once the gird is adapted to the interface, it must be partitioned to ensure load balancing across processors. This is achieve by constructing a Z-curve that traverses the leaves of all trees in order of tree index (c.f. figure 1). A Z-curve is a Space Filling Curve (SFC) with the important property that cells with close Z-indecies are also geometrically close in the physical domain. This is beneficial since it both leads to reduction in MPI communication and improves 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 [11]. Aside from grid manipulation and partitioning, we use two additional features of `p4est`, namely

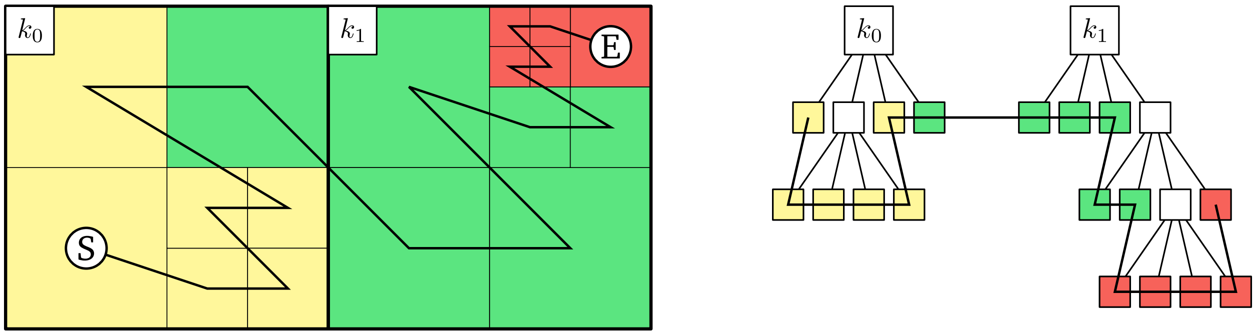


Figure 1: Left: A "forest" made up of two trees k_0 and k_1. Parallel partitioning is achieved by first constructing a Z-curve which starts at cell "S" and ends at cell "E". Next this tree is split up among processors either uniformly or by assigning different weights to cells. Here processors are represented via different colors. Note how use of Z-curve naturally leads to clustering of most cells in each processor domain. Right: Schematic of a tree data structure representing the forest and its partitioning.

generation of ghost layer cells and creating a globally unique node indexing. These algorithms are detailed in sections 3.5 and 3.6 of [11]. An important feature of our algorithms is that they are designed for non-graded trees. This is important because we can entirely skip tree balancing which was shown to be one of most time consuming parts of grid adaptation in `p4est` [11].

Finally, in `p4est` trees are linearized, i.e. only the leaves are explicitly stored. However, all of algorithms considered here require explicit knowledge of the entire tree. As a result we introduce a simple algorithm, called `Reconstruct`, which 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 [6] and our tests show that in a typical application they usually 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 forest by recursively refining from the root until reaching the same level and location of all leaves in `p4est` and `ghost`. Note that algorithm 1 does not involve any communication and is load balanced. Figure 2 illustrates an application of algorithm 1. Note how each processor 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.

Algorithm 1 $H \leftarrow$ `Reconstruct` (G)

```
 1: $H \leftarrow G$.macromesh()
 2: for $tr : G$.local_trees() do                          ▷ traverse local cells
 3:     for $q : tr$.cells() do
 4:         $H$.update_tree($tr, q$)
 5:     end for
 6: end for
 7: for $q : G$.ghost_cells() do                           ▷ traverse ghost cells
 8:     $H$.update_tree($q$.tree(), $q$)
 9: end for
10: return $H$
11:
12: function $H$.UPDATE_TREE($tr, q$)                      ▷ recursive reconstruction
13:     $q_l \leftarrow H$.root($tr$)
14:     while $q_l$.level() $\neq q$.level() do
15:         if $q_l$.is_leaf() then $q_l$.split()
16:         end if
17:         $h \leftarrow q_l$.length()/2          ▷ select the next child based on direction
18:         $i \leftarrow q.x \geq q_l.x + h$
19:         $j \leftarrow q.y \geq q_l.y + h$
20:         $k \leftarrow q.z \geq q_l.z + h$
21:         $q_l \leftarrow q_l$.child($i, j, k$)
22:     end while
23: end function
```

3.2. Semi-Lagrangian method

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

$$\left\{ \begin{array}{rcl} \dfrac{\mathrm{d}\mathbf{X}}{\mathrm{d}t} & = & \mathbf{V}, \\ \dfrac{\mathrm{d}\phi(\mathbf{X}(t),t)}{\mathrm{d}t} & = & 0. \end{array} \right. \tag{5}$$

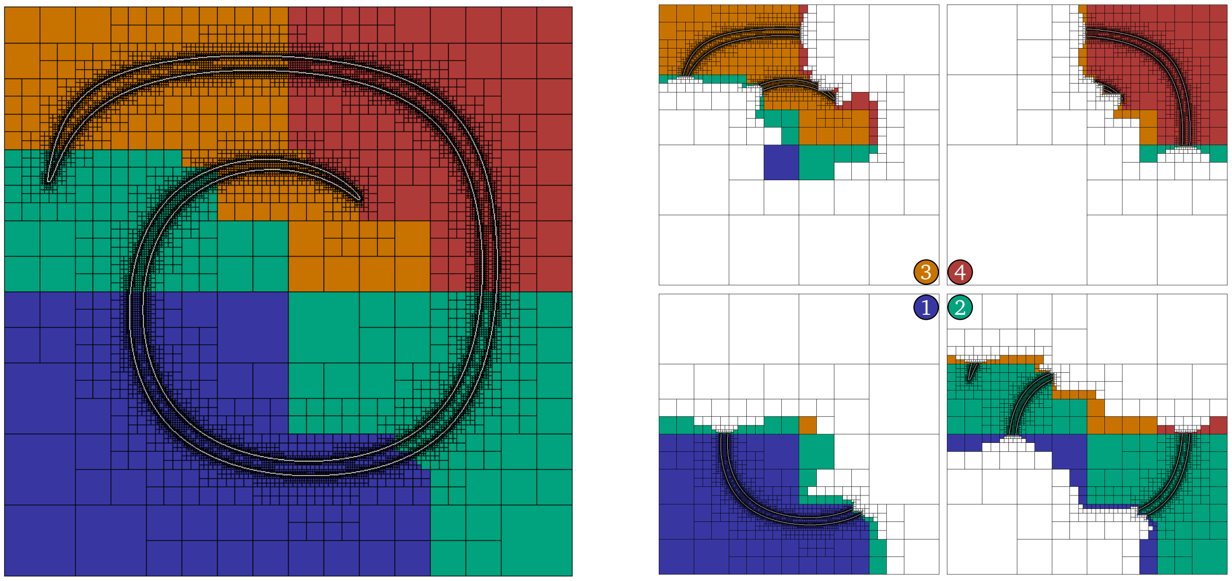


Figure 2: Left: A forest refined close to an interface and partitioned among four processors as indicated by colors. Right: Each processor independently recreates a local forest which is refined to match the global grid and is as coarse as possible elsewhere. Note that empty cells are fictitious, i.e. they are only required to generate the hierarchal structure and are not matched by any corresponding cell in the global forest.

The Semi-Lagrangian method integrates equations (5) backward in time, i.e. starting from the grid G^{n+1} [1], we simply write $\phi^{n+1}(\mathbf{X}^{n+1}) = \phi(\mathbf{X}(t^{n+1}), t^{n+1}) = \phi(\mathbf{X}(t^n), t^n) = \phi^n(\mathbf{X}_d)$. Here the characteristic curves are chosen such that $\mathbf{X}(t^{n+1})$ are simply the coordinates of grids points of G^{n+1}. Moreover $\mathbf{X}_d$ are the departure points which are computed using the second-order midpoint method [24]:

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

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

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

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

Note that all values at the intermediate point, $\mathbf{X}^\star$, and departure point, $\mathbf{X}_d$, must be calculated via interpolation from previous grids, G^n and G^{n-1}. Here we use the stabilized second-order interpolation for $\phi(\mathbf{X}_d)$ and the multi-linear interpolation for $\mathbf{V}^{n+\frac{1}{2}}(\mathbf{X}^\star)$ [24]. 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 algorithm 2 is probably the most important contribution of this article. Complication arises because not all calculated departure points will reside in the domain of current processor. Moreover, due to the irregular shape of partitions, we cannot be certain that they are entirely owned by neighboring processors. At best we can only expect that their location are bounded by a halo of width $w \leq \text{CFL}\,\Delta x_{\min}$ around the local partition. Of course 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 Semi-Lagrangian in the first place since we are interested in large values of CFL number.

One remedy to this problem, proposed in [38] for uniform grids, is to increase the size of ghost layer to $\lceil \text{CFL} \rceil$. For large values of 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

[1] G^{n+1} itself is computed in an iterative fashion as explained later on.

to repartitioning. 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 processor searches for all departure points inside the local tree. If the point is owned by a local cell, it is added to a local buffer, otherwise we find the processor 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, similar to the `UpdateLocalTree` function in algorithm 1. The owner's rank is found by computing the Z-index of the point and then using binary search on the Z-curve. This is already implemented in `p4est` and explained in details in section 2.5 of [11].

Once buffering is done, every processor knows exactly how many messages it needs to send and to which processors. This also implicitly defines processors that will later on send a reply message to this processor. However, at this point no one knows which processors they should expect a message from. We solve this problem using a simple *communication matrix* (see figure 3). Other approaches for solving this problem could include using non-blocking collective or one-sided communication operations introduced in the MPI-3 standard [**? ?**]. Although these algorithms have better theoretical communication complexities, we did not see any difference in the runtime or scalability of algorithm 2 when using them.

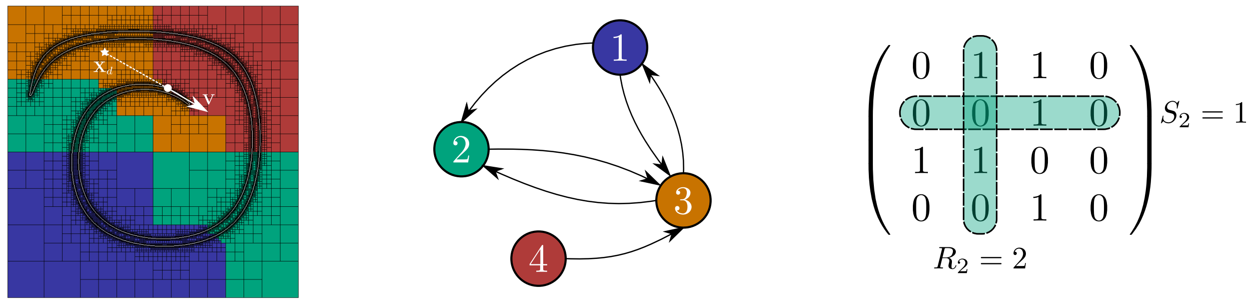

x_d
v
1
2
3
4
$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$
$S_2 = 1$
$R_2 = 2$

Figure 3: Left: Location of back-traced points depends on the magnitude of local velocity and time-step. Although departure distance is bounded by CFL $\Delta x_{\min}$, one cannot make any special assumption about receiving rank without explicit searching of the entire Z-curve. Moreover, the receiving processor has no prior knowledge about which processors 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 processors with arrows representing the direction that messages are sent. Right: The adjacency matrix of communication graph. For each row, sum of all columns represents the number of sent messages. Conversely, for each column, sum of all rows represents number of messages that need to be received. As detailed in algorithm 2 this information is enough to build a parallel interpolation scheme.

To solve the communication problem, we first compute the adjacency matrix of the communication pattern, i.e. we construct the matrix $A_{P\times P}$ such that:

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

Note that this matrix is also distributed among processors, 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 sent and received messages respectively. While S_i can be computed trivially, a reduction operation is required for computing R_i. For instance, this can be achieved using a single `MPI_Reduce_scatter` function call. The last phase of interpolation involves overlapping the computation of interpolated values for local points with the communication of data between processors. This is done by alternating between local calculations and probing for incoming messages from other processors. Interpolation is finished once values for all local points have been calculated and all remote requests have been processed (see algorithm 2).

Using the interpolation algorithm 2, we close this section by presenting the final Semi-Lagrangian algorithm 3. The basic idea here is to start from an initial guess G_0^{n+1} and modify the grid using refinement

Algorithm 2 *values* ← `Interpolate` $(H, \mathbf{X})$

```
1:  $col \leftarrow 0, buff \leftarrow null$                                   ▷ Phase I – buffering
2:  for $\mathbf{p} : \mathbf{X}$ do
3:      $[r, cell] \leftarrow H$.search($\mathbf{p}$)                 ▷ search for the owner's rank and cell
4:      if $r = mpirank$ then
5:          $buff[r]$.push_back($\mathbf{p}, cell$)
6:      else
7:          $buff[r]$.push_back($\mathbf{p}$)
8:          $col[r] \leftarrow 1$
9:      end if
10: end for
11: for $r : mpisize$ do          ▷ Phase II – initiate communication and compute number of messages
12:     if $col[r]$ then
13:         MP_Isend($r, buff[r]$)
14:     end if
15: end for
16: $S \leftarrow$ sum($col$)
17: $R \leftarrow$ MPI_Reduce_scatter($col$, MPI_SUM)
18: $done \leftarrow false, it \leftarrow buff[mpirank]$.begin()
19: while $!done$ do                                                ▷ Phase III – main loop
20:     if $it \neq buff[mpirank]$.end() then
21:         $values \leftarrow$ process_local($it$)                    ▷ process local interpolations
22:         ++$it$
23:     end if
24:     if $R > 0$ then                                ▷ process queries sent from remote processors
25:         $[msg, st] \leftarrow$ MPI_Iprobe()
26:         if $msg$ then
27:             $val\_buff \leftarrow$ process_queries($st$)          ▷ receive, search, and interpolate values
28:             MPI_Isend($st$.MPI_SOURCE, $val\_buff$)               ▷ send back interpolated values
29:             $R$--
30:         end if
31:     end if
32:     if $S > 0$ then                                            ▷ process replies sent to our queries
33:         $[msg, st] \leftarrow$ MPI_Iprobe()
34:         if $msg$ then
35:             $values \leftarrow$ process_replies($st$.MPI_SOURCE)     ▷ receive remotely interpolated values
36:             $S$--
37:         end if
38:     end if
39:     $done \leftarrow S = 0 \quad \& \quad R = 0 \quad \& \quad it = buff[mpirank]$.end()
40: end while
41: return $values$
```

and coarsening criteria in (3) and (4) 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 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 value 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 $l_{\max}$ steps where $l_{\max}$ denotes the maximum final depth of trees across all processors.

Algorithm 3 $[G^{n+1}, \phi^{n+1}] \leftarrow$ `SemiLagrangian` $(G^n, \phi^n, \mathbf{V}^n, \mathbf{V}^{n-1}, \text{CFL})$

```
1:  $\Delta t_l \leftarrow \text{CFL} \times \max\{\mathbf{V}^n\}/G^n$.hmin()
2:  $\Delta t \leftarrow$ MPI_Allreduce($\Delta t_l$, MPI_MIN)
3:  $H^n \leftarrow$ Reconstruct($G^n$)
4:  $G_0^{n+1} \leftarrow G^n$
5:  while true do
6:      $\mathbf{X}_d \leftarrow$ ComputeDeparturePoints($G_0^{n+1}, \mathbf{V}^n, \mathbf{V}^{n-1}, \Delta t$)      ▷ using equations 6 – 8
7:      $\phi^{n+1} \leftarrow$ Interpolate($H^n, \mathbf{X}_d$)
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 $[G^{n+1}, \phi^{n+1}]$
```

3.3. Reinitialization

Successive application of algorithm 3, especially for large values of CFL number, potentially lead degrades the signed distance property of level-set function. It is thus important to reinitialize the level-set 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 [24]. For completeness, we briefly review the scheme here. First equation (2) is written 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:

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

where $a^+ = \max(a, 0)$ and $a^- = \min(a, 0)$. Similar to [24], equation (9) is integrated in time using the TVD-RK2 scheme using adaptive time-stepping to accelerate convergence to steady state. Since all computation is grid based, parallel implementation of this scheme is mostly trivial. However, one minor point requires further explanation. As suggested in [24], one-sided derivatives $D_i^+\phi$ and $D_i^-\phi$ are computed using second order discretizations which require computation of second derivatives. To enable overlap between computation and communications when computing second derivatives and also integrating (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, boundary points are the collection of all local points that are regarded as ghost point, G_r, on all other processors, i.e.

$B_p = \bigcup_{r,\, r \neq p} G_r$. Private points are fined 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 computation/computation overlap of an arbitrary local operation $\phi^{n+1} \leftarrow \mathcal{F}(\phi^n)$. Note that `p4est` library already includes all the primitives required for labeling local points without any further communication.

Algorithm 4 $\phi^{n+1} \leftarrow$ `Overlap` $(\phi^n, \mathcal{F})$

```
 1: for $i : B_p$ do                                   ▷ I – perform computation on boundary points
 2:     $\phi_i^{n+1} \leftarrow \mathcal{F}(\phi_i^n)$
 3: end for
 4: $send\_req \leftarrow$ MPI_Isend($\phi_B^{n+1}$)           ▷ II – begin updating ghost values
 5: $recv\_req \leftarrow$ MPI_Irecv($\phi_G^{n+1}$)
 6: for $i : P_p$ do                                   ▷ III – perform computation on private points
 7:     $\phi_i^{n+1} \leftarrow \mathcal{F}(\phi_i^n)$
 8: end for
 9: MPI_Waitall($send\_req, recv\_req$)                ▷ IV – wait for ghost update to finish
10: return $\phi^{n+1}$
```

4. Scaling

In this section we present results that show the scalability of our algorithms. All of our tests have been run on the Stampede cluster at Texas Advanced Computing Center (TACC). Each node of Stampede has 2 eight-core Xenon E5-2680 processors clocked at 2.7 GHz with 32 GB of DDR3-1600 MHz memory and interconnected using an InfiniBand network. Unless explicitly mentioned otherwise, in all tests we use 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.

4.1. Interpolation

In this section we show 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 dimensions. We consider two cases. A small test on a level 9 tree with roughtly 20M cells and 33M nodes and larger test on a level 13 tree with roughly 128M cells and 280M nodes. In both cases number of randomly generated points are chosen to be equal to the number of nodes and the stablized second-order interpolation of [24] was performed 10 times to smooth out possible timing fluctuations.

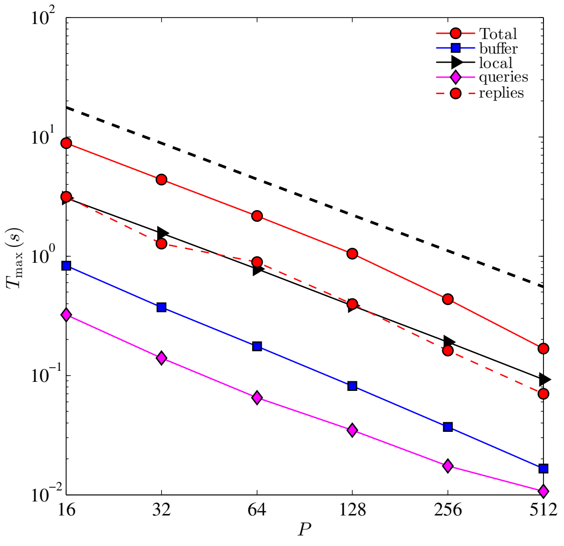
To simulate the effect of different CFL numbers, we generate the random points such that on each processor α percentage of them are located outside the processor boundary and thus will initiate communication. Scaling results are presented for $\alpha = 5\%$ and $\alpha = 95\%$ for both small and large problems in figure 4. Excellent scaling is obtained for the small problem for $P = 16 - 512$ even when 95% of interpolation points are remote to processors. For the larger problem, communication overhead prevents further scaling of algorithm beyond 4096 processors when $\alpha = 95\%$. Note that for $\alpha = 5\%$, even though some sections of algorithm stop scaling, the overall algorithm still scales since the local work dominates the timing. This illustrates the effectiveness of non-blocking communication pattern in algorithm 2.

4.2. Semi-Lagrangian

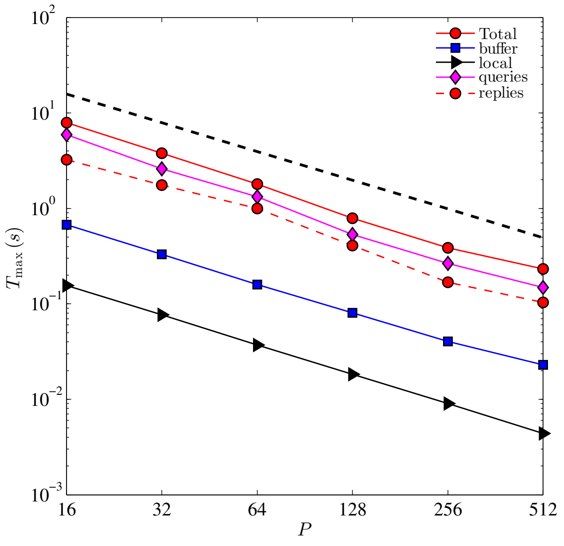
[1] `p4est`: Parallel Adaptive Mesh Refinement on Forests of Octrees. `http://p4est.github.io`.

[2] D Adalsteinsson and J Sethian. A Fast Level Set Method for Propagating Interfaces. *J. Comput. Phys.*, 118:269–277, 1995.

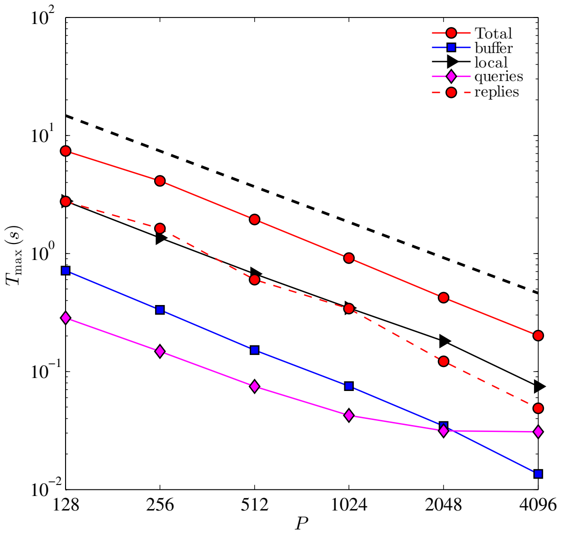
[3] Srinivas Aluru and F Sevilgen. Parallel domain decomposition and load balancing using space-filling curves. In *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*, pages 230–235. IEEE, 1997.



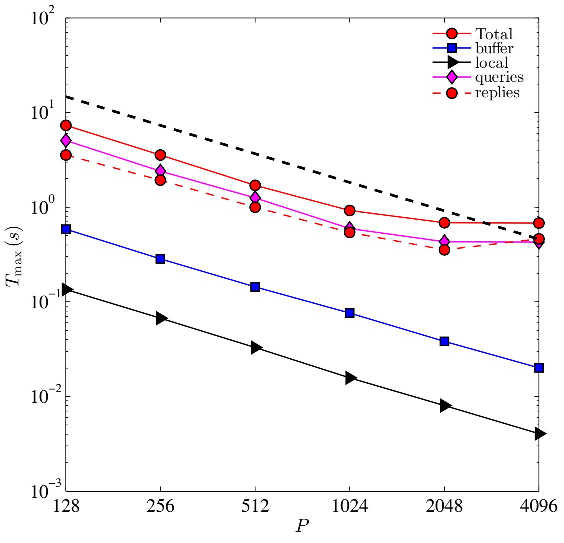
(a) $N_G = 33 \times 10^6$, $\alpha = 5\%$



(b) $N_G = 33 \times 10^6$, $\alpha = 95\%$



(c) $N_G = 280 \times 10^6$, $\alpha = 5\%$



(d) $N_G = 280 \times 10^6$, $\alpha = 95\%$

Figure 4: Strong scaling of algorithm 2 for several tests where N_G denotes the number of random interpolation points (same as number of nodes in the octree) and α denotes the percentage of these points that are remote to each processor. Here "Total" represents the total time spent in the interpolation while "buffer", "local", "queries", and "replies" represent the the timing for different sections (c.f. algorithm 2). 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 4096 processors due to communication overhead.

[4] 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, and Hong Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2014.

[5] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[6] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, and Martin Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Transactions on Mathematical Software (TOMS)*, 38(2):14, 2011.

[7] Erik G Boman, Ümit V Çatalyürek, Cédric Chevalier, and Karen D Devine. The zoltan and isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[8] Michael Breuß, Emiliano Cristiani, Pascal Gwosdek, and Oliver Vogel. An adaptive domain-decomposition technique for parallelization of the fast marching method. *Applied Mathematics and Computation*, 218(1):32–44, 2011.

[9] Emmanuel Brun, Arthur Guittet, and Frederic Gibou. A local level-set method using a hash table data structure. *J. Comp. Phys.*, 231:2528–2536, 2012.

[10] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Georg Stadler, Eh Tan, Tiankai Tu, Lucas C Wilcox, and Shijie Zhong. Scalable adaptive mantle convection simulation on petascale supercomputers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 62. IEEE Press, 2008.

[11] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[12] Paul M Campbell, Karen D Devine, Joseph E Flaherty, Luis G Gervasio, and James D Teresco. Dynamic octree load balancing using space-filling curves. *Williams College Department of Computer Science, Tech. Rep. CS-03-01*, 2003.

[13] Adam Chacon and Alexander Vladimirsky. A parallel heap-cell method for eikonal equations. *arXiv preprint arXiv:1306.4743*, 2013.

[14] Miles Detrixhe, Frédéric Gibou, and Chohong Min. A Parallel Fast Sweeping Method for the Eikonal Equation. *Journal of Computational Physics*, 237:46–55, March 2013.

[15] John Drake, Ian Foster, John Michalakes, Brian Toonen, and Patrick Worley. Design and performance of a scalable parallel community climate model. *Parallel Computing*, 21(10):1571–1591, 1995.

[16] D Enright, R Fedkiw, J Ferziger, and I Mitchell. A Hybrid Particle Level Set Method for Improved Interface Capturing. *J. Comput. Phys.*, 183:83–116, 2002.

[17] Oliver Fortmeier and 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*, pages 200–209. Springer, 2011.

[18] M Herrmann. A domain decomposition parallelization of the fast marching method. Technical report, DTIC Document, 2003.

[19] Marcus Herrmann. A parallel Eulerian interface tracking/Lagrangian point particle multi-scale coupling procedure. *Journal of Computational Physics*, 229(3):745–759, 2010.

[20] Won-Ki Jeong and Ross T. Whitaker. A fast iterative method for eikonal equations. *SIAM J. Sci. Comput.*, 30(5):2512–2534, 2008.

[21] D Juric. A Front-Tracking Method for Dendritic Solidification. *Journal of Computational Physics*, 123(1):127–148, January 1996.

[22] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[23] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. Simulating Water and Smoke with an Octree Data Structure. *ACM Transaction on Graphics (SIGGRAPH Proc.)*, pages 457–462, 2004.

[24] Chohong Min and Frederic Gibou. A second order accurate level set method on non-graded adaptive Cartesian grids. *Journal of Computational Physics*, 225(1):300–321, 2007.

[25] S Osher and R P Fedkiw. Level Set Methods: An Overview and Some Recent Results. *Journal of Computational Physics*, 169:463–502, 2001.

[26] Stanley Osher and Ronald Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer-Verlag, 2002.

[27] Stanley Osher and James A. Sethian. Fronts propagating with curvature dependent speed: Algorithms based on hamilton-jacobi formulations. *Journal of Computational Physics*, 79(1):12–49, 1988.

[28] Joseph M Rodriguez, Onkar Sahni, Richard T Lahey Jr, and Kenneth E Jansen. A parallel adaptive mesh method for the numerical simulation of multiphase flows. *Computers & Fluids*, 87:115–131, 2013.

[29] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*. Addison-Wesley, New York, 1990.

[30] Rahul S Sampath, Santi S Adavani, Hari Sundar, Ilya Lashuk, and 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*, page 18. IEEE Press, 2008.

[31] Rahul S Sampath and George Biros. A parallel geometric multigrid method for finite elements on octree meshes, siam journal on scientific computing. *SIAM J. of Scientific Comput.*, 32(3):1361–1392, 2010.

[32] J A Sethian. A Fast Marching Level Set Method for Monotonically Advancing Fronts. *Proceedings of the National Academy of Sciences of the United States of America*, 93(4):1591–5, March 1996.

[33] James A. Sethian. *Level set methods and fast marching methods*. Cambridge University Press, 1999.

[34] J Strain. Tree Methods for Moving Interfaces. *J. Comput. Phys.*, 151:616–648, 1999.

[35] Mark Sussman. A parallelized, adaptive algorithm for multiphase flows in general geometries. *Computers & structures*, 83(6):435–444, 2005.

[36] Mark Sussman, Peter Smereka, and Stanley Osher. A level set approach for computing solutions to incompressible two-phase flow. *Journal of Computational Physics*, 114:146–159, 1994.

[37] Maxime Theillard, Frédéric Gibou, and Tresa Pollock. A sharp computational method for the simulation of the solidification of binary alloys. *Journal of Scientific Computing*, pages 1–25, 2014.

[38] S Thomas and J Côté. Massively parallel semi-Lagrangian advection. *Simulation Practice and Theory*, 3(4):223–238, 1995.

[39] G Tryggvason, B Bunner, A Esmaeeli, D Juric, N Al-Rawahi, W Tauber, J Han, S Nas, and Y.-J. Jan. A Front-Tracking Method for the Computations of Multiphase Flow. *J. Comput. Phys.*, 169:708–759, 2001.

[40] Tiankai Tu, David R O'Hallaron, and Omar Ghattas. Scalable parallel octree meshing for terascale applications. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 4–4. IEEE, 2005.

[41] Maria Cristina Tugurlan. *Fast marching methods-parallel implementation and analysis*. PhD thesis, Louisiana State University, 2008.

[42] Kai Wang, Anthony Chang, Laxmikant V Kale, and Jonathan A Dantzig. Parallelization of a level set method for simulating dendritic growth. *Journal of Parallel and Distributed Computing*, 66(11):1379–1386, 2006.

[43] JB White III and Jack J Dongarra. High-performance high-resolution semi-Lagrangian tracer transport on a sphere. *Journal of Computational Physics*, 230(17):6778–6799, 2011.

[44] Hongkai Zhao. A fast sweeping method for eikonal equations. *Mathematics of computation*, 74(250):603–627, 2005.

[45] Hongkai Zhao. Parallel implementations of the fast sweeping method. *Journal of Computational Mathematics*, 25:421–429, 2007.