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, University of Bonn, Bonn 53115, Germany.*

Abstract

We present scalable parallel algorithms for the level-set method on adaptive Quadtree and Octree Cartesian grids. The algorithms are based on a domain decomposition technique and implemented using MPI and the open-source p4est library. An important contribution is a scalable parallel 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 Quadtree and Octree 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 limited to 4096 cores at most. Finally a relevant application of the algorithms is presented in modeling the crystallization phenomena by solving a Stefan problem, illustrating a level of detail that would be impossible 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.

Keywords: Quadtree/Octree Grids, Parallel Computing, Space Filling Curves, Semi-Lagrangian Method, Level-set Method

1. Introduction

The level-set method, originally proposed by Sethian and Osher [31], 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 [29, 38]. 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 [24, 44], the level-set method is typically less accurate and mass conservation could be a problem, although progress has been made in resolving this issue [18]. Second, the level-set function has to be defined in a higher dimensional space that that of the interface. If only the location of the 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 [11].

Another approach that can address both problems is the use of local grid refinement. In [39] the idea of using tree-based grids for level-set calculations was first introduced and later extended in [32, 26] for fluid simulations. More recently, authors in [28] proposed second-order accurate level-set methods on Quadtree (two spatial dimensions) and Octree (three spatial 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,

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

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, which allows for arbitrarily large time steps. This is particularly important when using adaptive grids, as higher grid resolutions translate into impractical small time steps.

In general, an infinite number of level-set functions can describe the same zero contour and thus an interface. However, it is desirable to chose a function with the signed distance property, $|\nabla\phi| = 1$. As detailed in section 1, we solve the pseudo-time transient reinitialization equation [40, 30] to achieve this property,

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

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 appropriate 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 algorithms presented in section 3 are based on the sequential methods presented in [28] and refer the interested reader to the aforementioned articles 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 [39]. Moreover, adaptive tree-based grids are easy to generate in the presence of a signed-distance level-set function [28] and can efficiently be encoded using a tree datastructure [34]. In order to develop scalable parallel algorithms on these grids, it is necessary to parallelize the datastructure 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 [1] is a collection of such parallel algorithms that has recently emerged and shown to scale up to 200,000 cores [13] and more recently to 450,000 (my arXiv preprint)

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. figure 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 this article, however, we limit discussions to simple uniform Cartesian macromeshshes. 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 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:

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

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:

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

We refer to section 3.2 of [13] for details on the 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 processes. This is achieved 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-indices are also geometrically close in the physical domain. This is beneficial since it leads to both 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

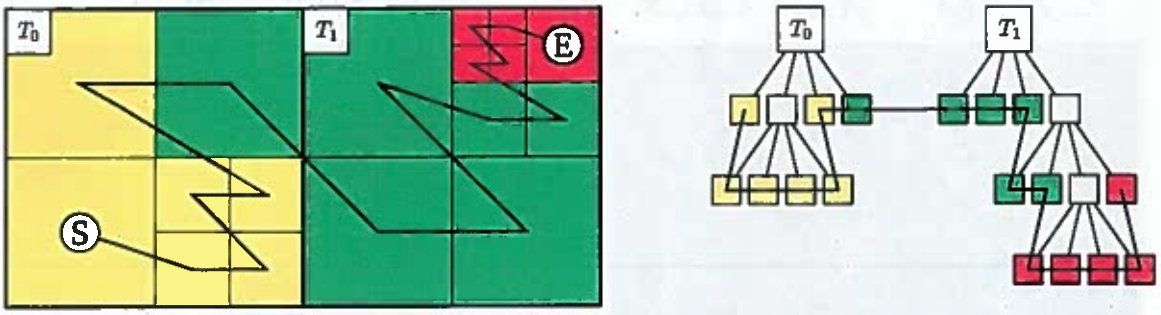


Figure 1: Left: a "forest" made up of two trees T_0 and T_1. Parallel partitioning is achieved by first constructing a Z-curve starting at cell "S" and ending at cell "E". Next, the one dimensional curve is split up among processes either uniformly or by assigning different weights to cells. Here processes 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 the forest and its partitioning.

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]. 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 the most 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 [8] 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. Figure 2 illustrates an example where Algorithm 1 is applied. Note how each process 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:

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

The semi-Lagrangian method integrates equations (5) backward in time, i.e. starting from the grid G^{n+1} (computed in 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 curves are chosen such that $\underline{\mathbf{X}}(t^{n+1})$ are ~~simply~~ 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 [28]:

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

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

Algo 1 goes top-down and is strictly $O(N_p \log N_p)$.
I can think of a recursion that is $O(N_p)$.
Would you want me to write this? Let's talk.

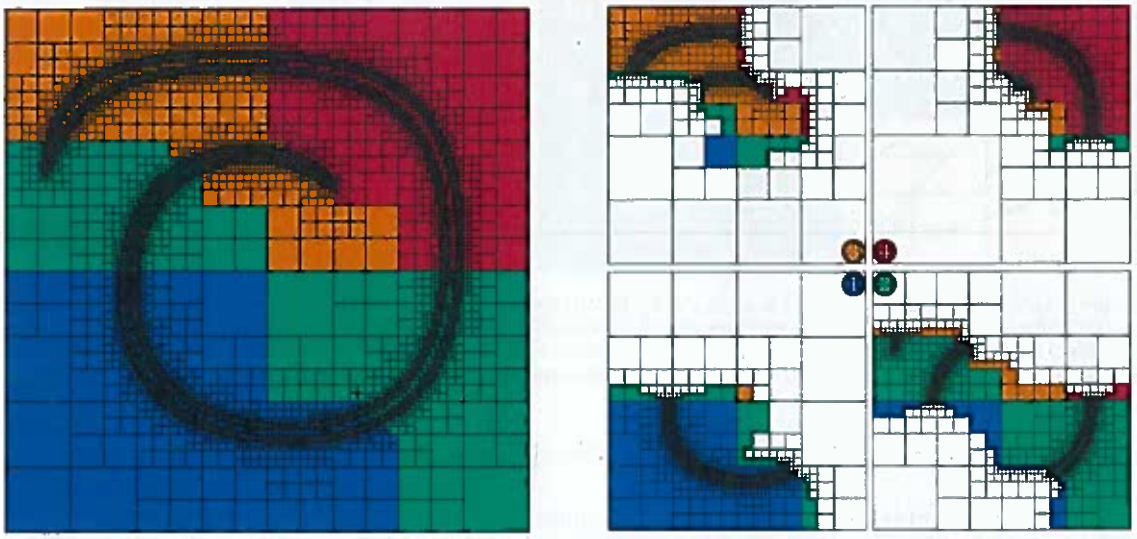


Figure 2: Left: a forest refined close to an interface and partitioned among four processes, as indicated by color. Right: each process independently recreates a local forest that is refined to match the local grid and is as coarse as possible elsewhere. Note that empty cells are fictitious, i.e. they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

I do not see how H is modified.

Algorithm 1 $H \leftarrow$ `Reconstruct` (G): construction of the local tree hierarchy H from the grid G

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

What does it do? Explanation missing

Ah sorry.

What is the result of this?

So now $c_l = c$ and we are done?

Compare this to Algorithm 1 in Burstedde, Ghattas, Stadler, Tan, Wilcox [8] — what is different? Need to cite/discuss.

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

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

Note that all values at the intermediate point, $\underline{\mathbf{X}}^*$, 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 $\underline{\mathbf{u}}^{n+\frac{1}{2}}(\underline{\mathbf{X}}^*)$ [28]. 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 procedure given in Algorithm 2 is probably the most important contribution of this article since the 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 $w \leq \text{CFL}\,\Delta x_{\min}$ around the local partition, where $x_{\min}$ is the size of the smallest cell in the forest. Naturally, if one enforces $\text{CFL} \leq 1$, 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 values.

One remedy to this problem, proposed in [43] 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 (c.f. 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 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 [22]. 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.

To solve the communication problem, we first compute the adjacency matrix of the communication pattern, i.e. we construct the matrix $A_{P\times P}$, 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 sent and received messages, 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.

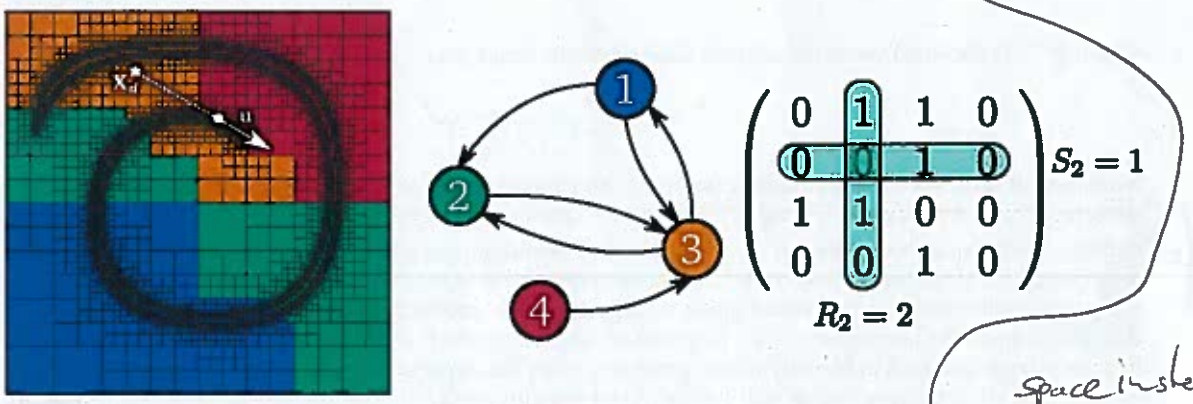


Figure 3: Left: the location of back-traced points depends on the magnitude of the local velocity and on the time-step. Although the distance to the departure point is bounded by $\mathrm{CFL}\Delta x_{\min}$, one cannot predict the receiving rank without explicitly searching the entire Z-curve. Moreover, 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.

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).

Using the interpolation Algorithm 2, we close this section by presenting the final semi-Lagrangian Algorithm 3. 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 [28]. For completeness, we briefly review the scheme. First, we write equation (2) in the following semi-discrete form:

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

Algorithm *values* ← Interpolate $(H, F, \underline{X})$: interpolate the value of F, defined on the local tree hierarchy H, at coordinates $\underline{X}$

```
 1: col ← 0, buff ← null                                   ▷ Phase I – buffering
 2: for p : X do
 3:     [r, cell] ← H.search(p)                            ▷ search for the owner's rank and cell
 4:     if r ≠ mpirank then
 5:         buff[r].push_back(p, cell)
 6:     else
 7:         buff[r].push_back(p)
 8:         col[r] ← 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 ← sum(col)
17: R ← MPI_Reduce_scatter(col, MPI_SUM)
18: done ← false, it ← buff[mpirank].begin()
19: while !done do                                         ▷ Phase III – main loop
20:     if it ≠ buff[mpirank].end() then
21:         values ← process_local(it)                     ▷ process local interpolations
22:         ++it
23:     end if
24:     if R > 0 then                                      ▷ process queries sent from remote processes
25:         [msg, st] ← MPI_Iprobe()
26:         if msg then
27:             val_buff ← 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] ← MPI_Iprobe()
34:         if msg then
35:             values ← process_replies(st.MPI_SOURCE)    ▷ receive remotely interpolated values
36:             S--
37:         end if
38:     end if
39:     done ← S = 0 & R = 0 & it = buff[mpirank].end()
40: end while
41: return values
```

Algorithm 3 $[G^{n+1}, \phi^{n+1}] \leftarrow$ SemiLagrangian $(G^n, \phi^n, \underline{u}^n, \underline{u}^{n-1}, \text{CFL})$: 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}

$\Delta t_l \leftarrow \text{CFL} \times G^n.\text{hmin}()/\max\{\underline{u}^n\}$
$\Delta t \leftarrow \text{MPI\_Allreduce}(\Delta t_l, \text{MPI\_MIN})$
$H^n \leftarrow \text{Reconstruct}(G^n)$
$G_0^{n+1} \leftarrow G^n$
while *true* **do**
  $\underline{X}_d \leftarrow \text{ComputeDeparturePoints}(G_0^{n+1}, \underline{u}^n, \underline{u}^{n-1}, \Delta t)$ ▷ using equations 6 – 8
  $\phi^{n+1} \leftarrow \text{Interpolate}(H^n, \phi^n, \underline{X}_d)$
  $G^{n+1} \leftarrow G_0^{n+1}.\text{refine\_and\_coarsen}(\phi^{n+1})$ ▷ using equations 3 and 4 as criteria
  if $G^{n+1} \neq G_0^{n+1}$ **then**
    $G^{n+1}.\text{partition}()$
    $G_0^{n+1} \leftarrow G^{n+1}$
  else
    break
  end if
end while
return $[G^{n+1}, \phi^{n+1}]$

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

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

where $a^+ = \max(a, 0)$ and $a^- = \min(a, 0)$. Similar to [28], 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. 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 [28], one-sided derivatives $D_i^+\phi$ and $D_i^-\phi$ are computed using second order discretization which requires to compute the second-order derivatives. To enable overlap between computation and communication 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. $B_p = \bigcup_{r \neq p} G_r$. 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.

4. Scaling results

In this section we present some results that demonstrate the scalability of our algorithms. All of our tests were ran on the Stampede cluster at the Texas Advanced Computing Center (TACC) 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

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

```
1: for i : B_p do                                  ▷ I – perform computation on boundary points
2:     y_i ← F(x_i)
3: end for
4: send_req ← MPI_Isend(y_B)                       ▷ II – begin updating ghost values
5: recv_req ← MPI_Irecv(y_G)
6: for i : P_p do                                  ▷ III – perform computation on private points
7:     y_i ← F(x_i)
8: end for
9: MPI_Waitall(send_req, recv_req)                 ▷ IV – wait for ghost update to finish
10: return y
```

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 common 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 33M nodes and a larger test on a level 13 tree with roughly 280M 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 [28] 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 figure 4. We also present a third row of results for a much larger problem with roughly 1.66B 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 $\alpha = 95\%$ (c.f. table 1). 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 figure 4 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 scalablable for sufficiently large problems.

	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%

Table 1: Parallel efficiency of the total runtime of the interpolation algorithm for the small (33M nodes), large (280M nodes), and very large (1.66B nodes) tests. Reported efficiencies are based on the lowest number of processes for each test.

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

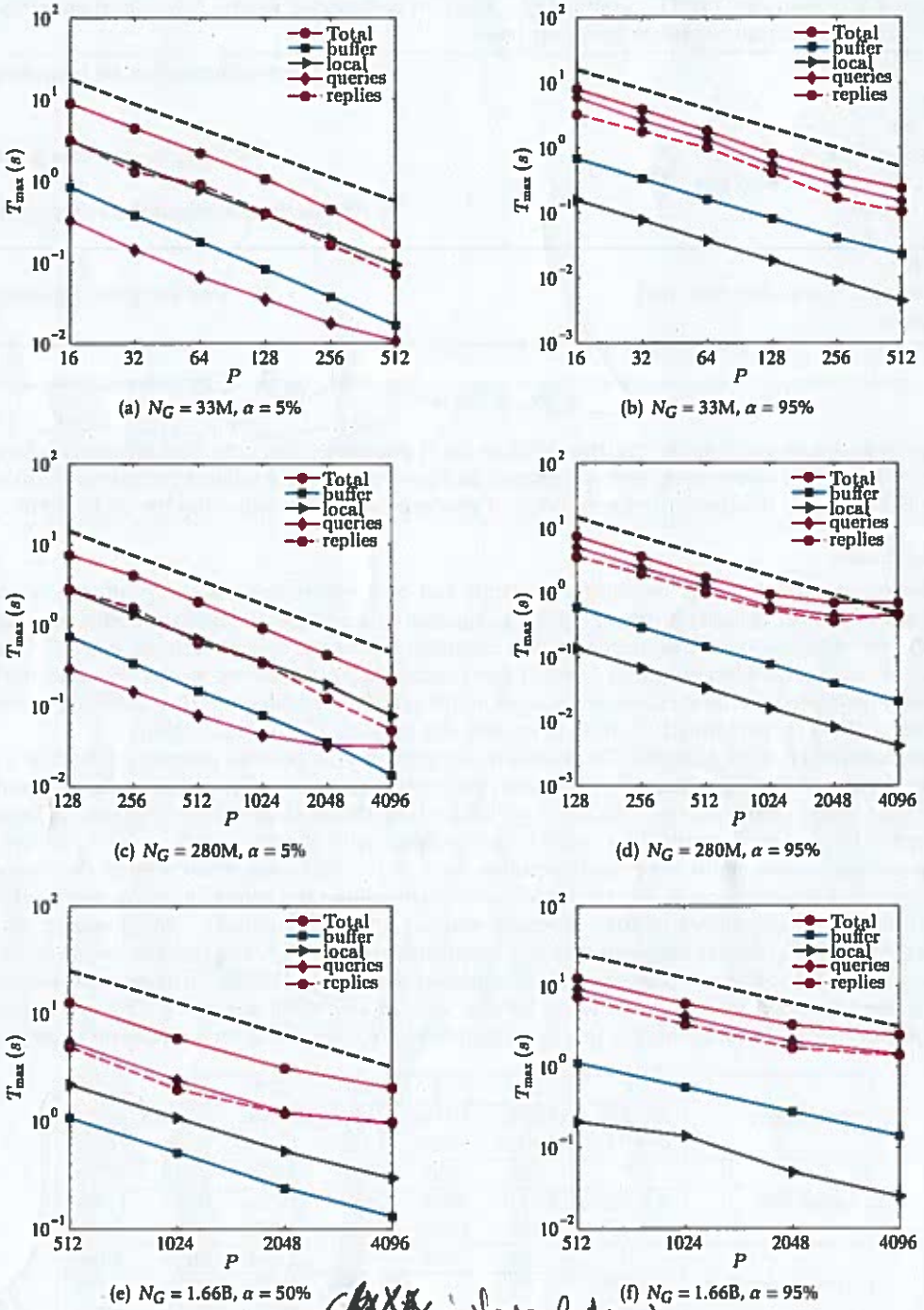


Figure 4: Strong scaling of Algorithm 2 for several tests where N_G denotes the number of random interpolation points (which is the same as the number of nodes in the octree) and α denotes the percentage of these points that are remote for each process. Here "Total" represents the total time spent in the interpolation while "buffer", "local", "queries", and "replies" represent the the timing for different sections (c.f. 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 α = 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 = 1.66B points (e-f).

4.2. *Semi-Lagrangian* method

To test the scalability of the semi-Lagrangian scheme of Algorithm 3, we consider a slightly modified version of the Enright's rotation test [18], 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:

This is not so "small"

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

To understand the effect of the CFL number on the scalability of the algorithm we perform one step of the semi-Lagrangian algorithm for CFL = 1, CFL = 10, and 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 15M and 255M nodes, respectively.

2 levels should give a factor 64 ?

I prefer tables without the [h]

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) $l_{max} = 12$

Table 2: 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 15M and 255M 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.

What (context)?

Table 2 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. Note that the apparent dependence on the number of processes is most likely due to round-off errors during the interpolation and/or backtracking steps. Figures 5 and 6 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 2. For both problems, excellent scalability is observed for CFL = 1 and CFL = 10. The algorithm even shows good scalability when taken to the extreme, i.e. for CFL = 100.

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. Figure 7 illustrates the effects of the CFL number on different metrics, namely the number of interpolation points[2], N_p, the number of sent and

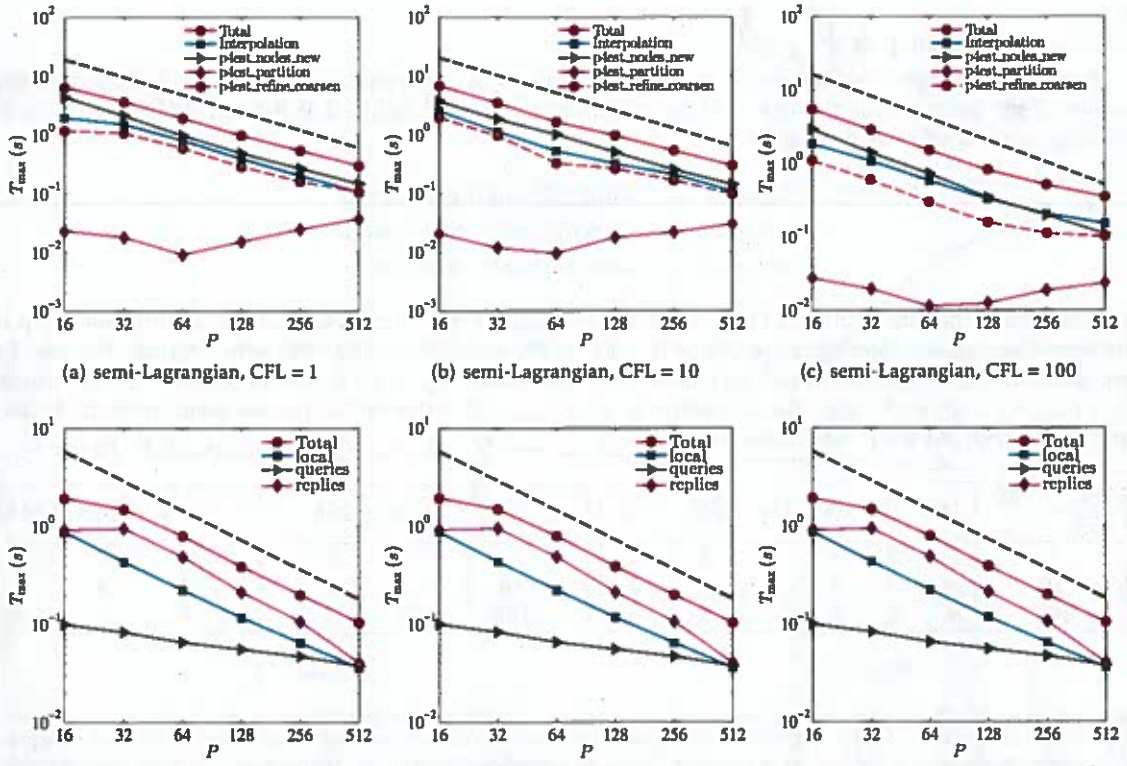
[2]Note that this includes both the local points and the points queried by other processes

This is a strong point in our favor: we require a very fast octree. Mention in the intro that our method is very demanding in terms of ...

Sub-iterations on the grid means that

Do we need an explicit Algorithm to explain p4est_refine_coarsen? It's hard to figure out what exactly is being done.

Referring to [13] is not enough to understand it.



(a) semi-Lagrangian, CFL = 1 (b) semi-Lagrangian, CFL = 10 (c) semi-Lagrangian, CFL = 100

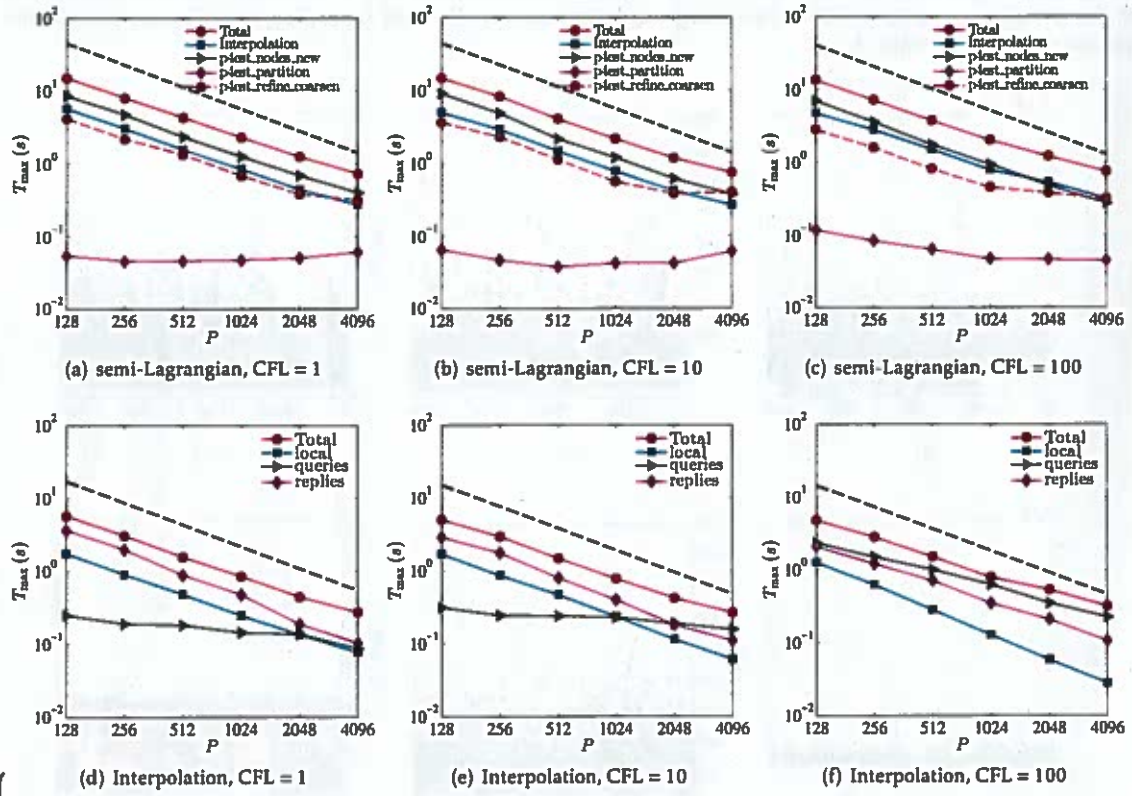
(d) Interpolation, CFL = 1 (e) Interpolation, CFL = 10 (f) Interpolation, CFL = 100

Figure 5: Strong scaling of a single time step of Algorithm 3 for the rotation test on a level-10 octree with approximately 15M 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 (c.f. table 2). 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].

	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%

fairly close values

Table 3: Parallel efficiency of the runtime of a single semi-Lagrangian step. Reported efficiencies are based on the lowest number of processes for each test.



replies > local: why?

Figure 6: Strong scaling of a single time step of Algorithm 3 for the rotation test on a level-12 octree with approximately 255M 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 (c.f. table 2).

Comment on absolute run times: 10^1s seems to be the limit of the network.

What are the components of p4est_refine_coarsen? Which is the most expensive?

More details needed on Algorithm.

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 figure 7(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 (c.f. figure 7(c)). Interestingly, however, the number of sent and received messages do not seem to affected by the CFL number. The bottom row of figure 7 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 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 4.

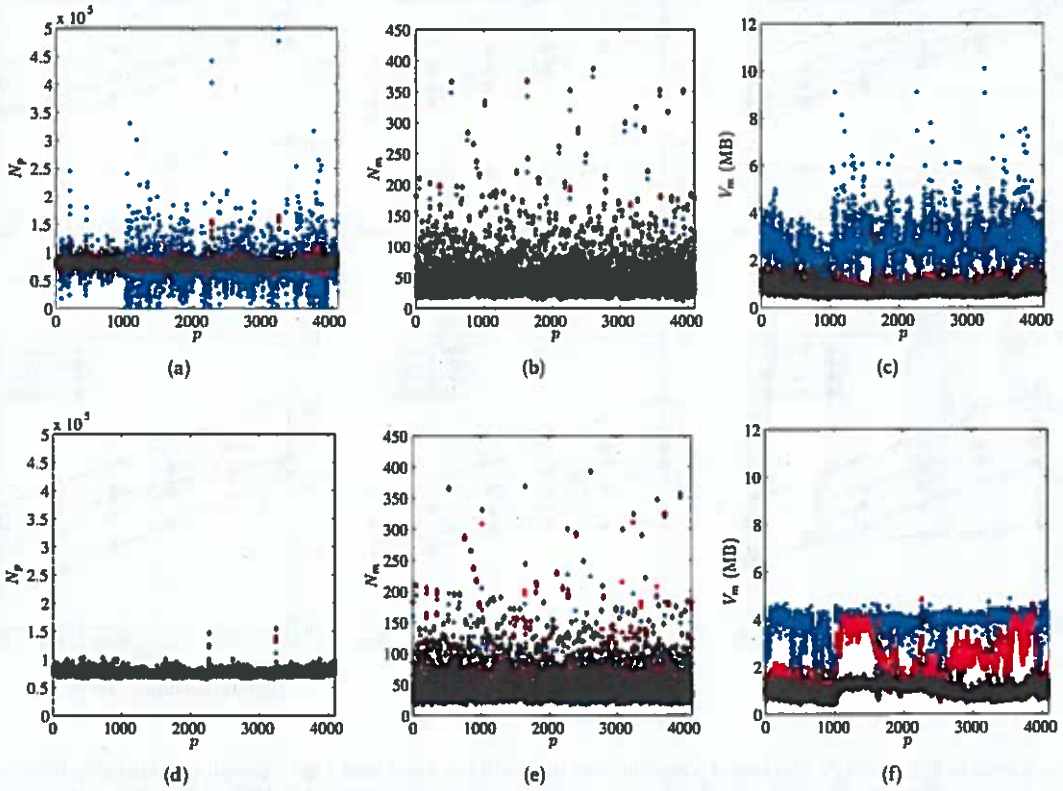


Figure 7: Performance indicators of the first (top row) and last (bottom row) sub-iterations of the semi-Lagrangian algorithm for the level-set advection on 4096 processes with CFL = 1 (◆), CFL = 10 (■), and CFL = 100 (●). Increasing the CFL number ~~greatly~~ causes load imbalance during interpolation (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 (c.f. 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).

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~~ of computing the signed distance function to a collection of 100

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			

Table 4: Detailed load balancing and communication information for the advection test for CFL = 1, CFL = 10, and CFL = 100. 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.

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%

Table 5: Parallel efficiency of the total runtime for the reinitialization test based on the lowest number of processes for each test.

spheres whose radii and centers are chosen randomly. The test is performed on a small, level-8 octree with about 21M and a larger, level-10 octree with about 337M grid points. In both cases the forest is built on a $3\times 3\times 3$ macro-mesh. Figure 8 illustrates that our reinitialization algorithm, and in particular the overlapping strategy presented in Algorithm 4, scales very well (c.f. table 5). 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 crystaline 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

$$\frac{\partial T_l}{\partial t} = D_l \Delta T_l \quad \text{in } \Omega_l, \tag{10}$$

$$\frac{\partial T_s}{\partial t} = D_s \Delta T_s \quad \text{in } \Omega_s. \tag{11}$$

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 [4, 3]:

$$T_s = T_l = T_\Gamma = -\epsilon_c \kappa - \epsilon_v (\underline{\mathbf{V}} \cdot \underline{\mathbf{n}}), \tag{12}$$

where κ is the local interface curvature, $\underline{\mathbf{V}}$ 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

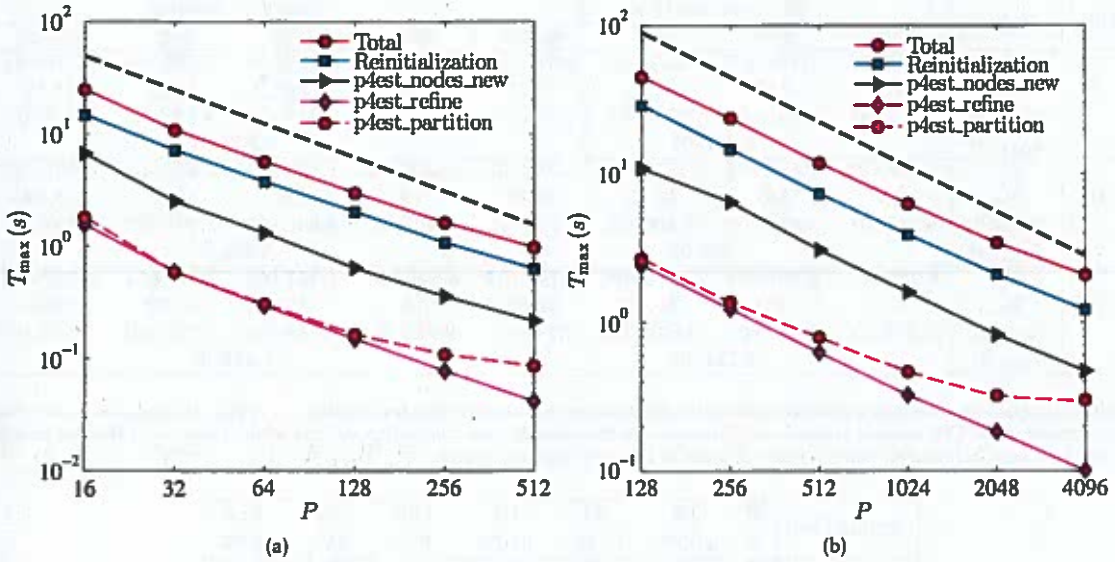


Figure 8: Scalability of the reinitialization test for a small (left) and large (right) octree with roughly 21M and 337M grid points, respectively. The black dashed line represents ideal scaling. Excellent results are obtained in both cases, illustrating the scalability of the overlapping strategy (c.f. algorithm 4).

velocity $\underline{\mathbf{V}}$ is defined from the jump in the heat flux at the interface,

$$(\underline{\mathbf{V}} \cdot \underline{\mathbf{n}}) = -\left[D_l \frac{\partial T_l}{\partial \underline{\mathbf{n}}} - D_s \frac{\partial T_s}{\partial \underline{\mathbf{n}}} \right]. \tag{13}$$

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

$$\Delta t = 5\, \Delta x_{min}\, \min(1, 1/\max\|\underline{\mathbf{V}}\|), \tag{14}$$

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.

Algorithm 5 General procedure for solving the Stefan problem

1: Initialize the forest and ϕ given the initial geometry.
2: Initialize T_s in Ω^+ and T_l in Ω^-.
3: Reinitialize ϕ and compute the local interface curvature κ.
4: Compute T_l^{n+1} and T_s^{n+1} by solving the heat equations (10) and (11).
5: Extrapolate T_s^{n+1} from Ω^+ to Ω^- and T_l^{n+1} from Ω^- to Ω^+.
6: Compute the velocity field $\underline{\mathbf{V}}$ according to (13).
7: Compute the time step dt following (14).
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.

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

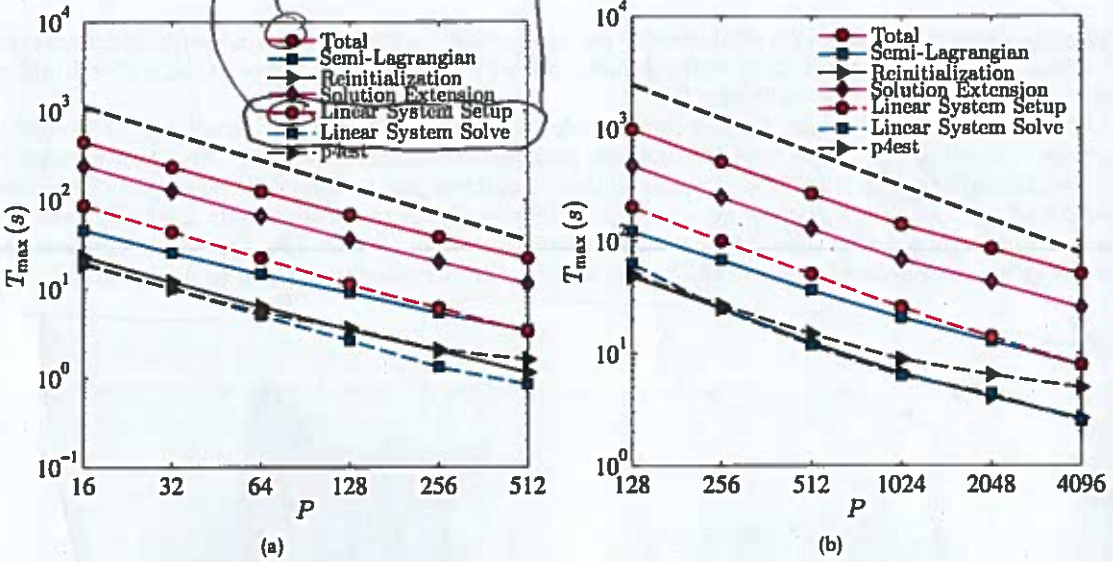


Figure 9: Scalability of the Stefan problem for small (left) and large (right) octrees with roughly 7M and 105M grid points, respectively. The solid dashed line represents perfect scaling. As expected from the scalability analysis of the individual components, we observe excellent results, illustrating the potential of our algorithms.

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%

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

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 7M grid points, and the maximum resolution for the large test is 11, corresponding to 105M grid points. The results are presented in figure 9. As expected from the results obtained for each component in the previous sections, our implementation of the Stefan problem exhibits very satisfactory scaling (c.f. table 6).

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 (12) are defined as

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

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

$$\epsilon_1 = 2\,(\sin(x) + \cos(y) + 2) \cdot 10^{-6}, \qquad \epsilon_2 = 2\,(\sin(x) + \cos(z) + 2) \cdot 10^{-6},$$
$$\alpha_1 = \frac{1}{4}(\cos(x) + \sin(y) + 2), \qquad \alpha_2 = \frac{1}{4}(\cos(x) + \sin(z) + 2),$$

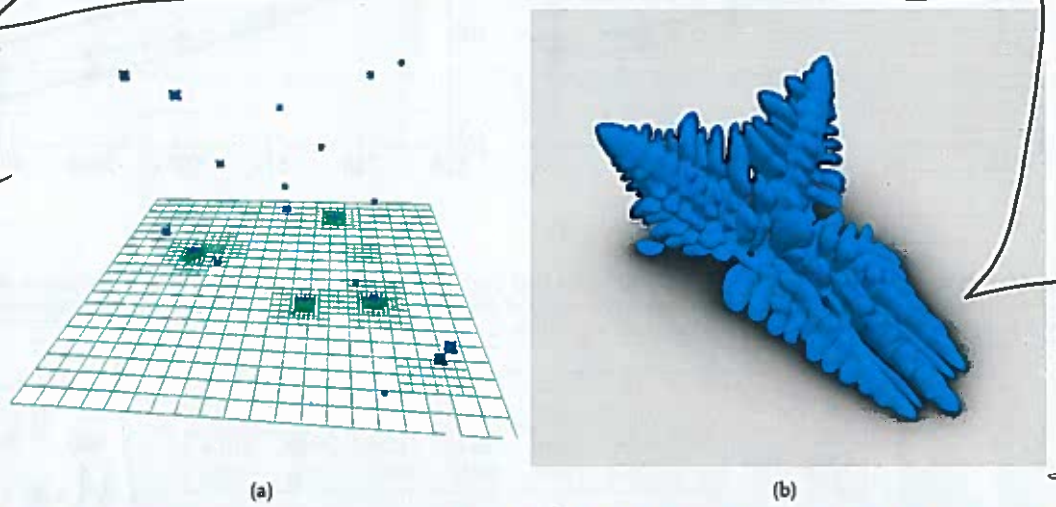
by who? Is this our decision? → active

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 figures 10 and 11. The final iteration of the simulation consisted of 167M 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.

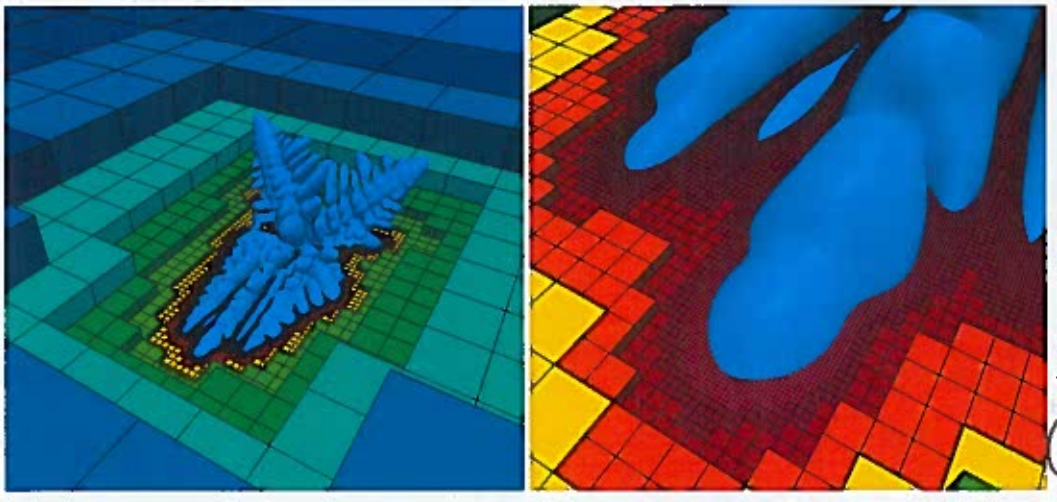
Need a bit more background on the physics/values?

Refer, see above



(a) (b)

between what? Linear solve needs to be mentioned in the paper.



(c) (d)

Or does this refer to semi-Lagrangian?

Figure 10: Visualization of the computational mesh (a, c, d) and the ~~temperature field~~ (b) for the Stefan problem simulation.

interface

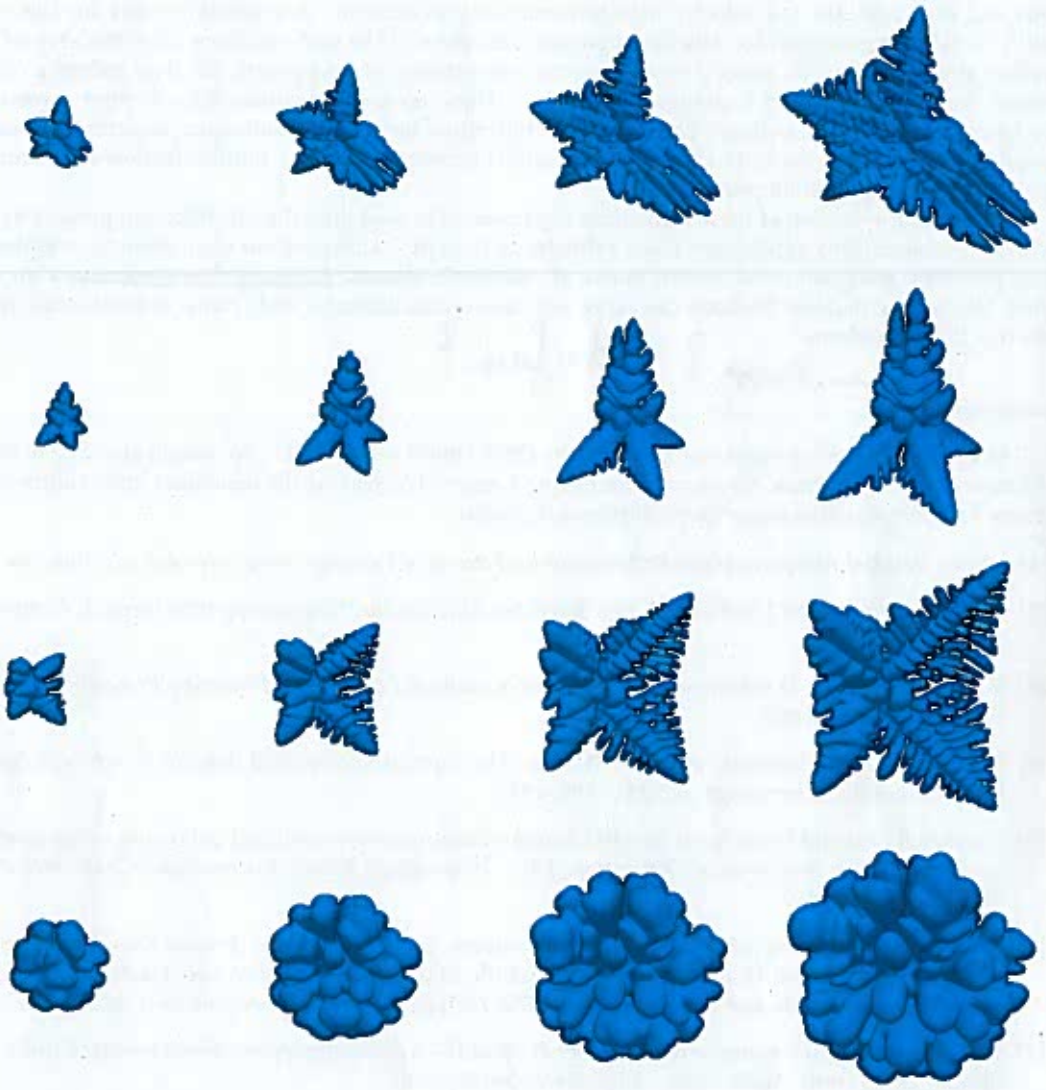


Figure 11: Time evolution of four of the crystals obtained for the Stefan problem simulation. The snapshots represent, from left to right, iterations 96, 196, 296 and 396.

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.

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 problems and free body problems.

Acknowledgment

The funding for this project was provided by ONR N00014-11-1-0027. We would also like to thank the technical staff at the Texas Advanced Computing Center (TACC) and the developer community of PETSc library for their valuable suggestions during this project.

[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] V Alexiades and A D Solomon. *Mathematical Modeling of Melting and Freezing Processes*. Hemisphere, Washingotn, DC, 1993.

[4] V Alexiades, A D Solomon, and D G Wilson. The formation of a solid nucleus in supercooled liquid. I. *J, Non-Equilib. Thermodyn.*, 13:281–300, 1988.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. *ACM SIGPLAN Notices*, 45(5):159–168, 2010.

[23] WK Jeong and RT Whitaker. A Fast Iterative Method for Eikonal Equations. *SIAM Journal on Scientific Computing*, 30(5):2512–2534, 2008.

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

[25] 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.

[26] F. Losasso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. *ACM Trans. Graph. (SIGGRAPH Proc.)*, pages 457–462, 2004.

[27] Frank Losasso, Ron Fedkiw, and Stanley Osher. Spatially Adaptive Techniques for Level Set Methods and Incompressible Flow. *Computers and Fluids*, 35:995–1010, 2006.

[28] C. Min and F. Gibou. A second order accurate level set method on non-graded adaptive Cartesian grids. *J. Comput. Phys.*, 225:300–321, 2007.

[29] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer-Verlag, 2002. New York, NY.

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

[31] S. Osher and J.A. Sethian. Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations. *Journal of Computational Physics*, 79:12–49, 1988.

[32] S. Popinet. Gerris: A tree-based adaptive solver for the incompressible euler equations in complex geometries. *J. Comput. Phys.*, 190:572–600, 2003.

[33] 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.

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

[35] 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.

[36] 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.

[37] J. Sethian. A fast marching level set method for monotonically advancing fronts. *Proc. Natl. Acad. Sci.*, 93:1591–1595, 1996.

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

[39] J. Strain. Tree methods for moving interfaces. *J. Comput. Phys.*, 151:616–648, 1999.

[40] M. Sussman, P. Smereka, and S. Osher. A level set approach for computing solutions to incompressible two-phase flow. *J. Comput. Phys.*, 114:146–159, 1994.

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

[42] M. Theillard, F. Gibou, and T. Pollock. A sharp computational method for the simulation of the solidification of binary alloys. *J. Sci. Comput.*, 2014.

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

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

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

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