Scalable adaptive algorithms for next-generation multiphase simulations

Kumar Saurabh§
Iowa State University, USA
khanwale@stanford.edu

Masado Ishii§
University of Utah, USA
masado@cs.utah.edu

Makrand A. Khanwale
Stanford University, USA
khanwale@stanford.edu

Hari Sundar
University of Utah, USA
hari@cs.utah.edu

Baskar Ganapathysubramanian
Iowa State University, USA
baskarg@iastate.edu

Abstract—The accuracy of multiphysics simulations is strongly contingent on the finest resolution of mesh used to resolve the interface. However, the increased resolution comes at a cost of inverting a larger matrix size. In this work, we propose algorithmic advances that aims to reduce the computational cost without compromising on the physics by selectively detecting the key regions of interest (droplets/filaments) that requires significantly higher resolution. The overall framework uses an adaptive octree-based mesh generator, which is integrated with PETSc’s linear algebra solver. We demonstrate the scaling of the framework up to 114,688 processes on TACC Frontera. Finally we deploy the framework to simulate primary jet atomization on an equivalent 35 trillion grid points – 64× greater than the state-of-the-art simulations.

Index Terms—multiphase simulations, octree, massively parallel algorithms.

I. INTRODUCTION

Understanding the mechanisms of phase interactions in two-phase flows is critical to answering fundamental and pressing questions in science and technology. In particular, insights into phase interactions may lead to accurate, low-cost coarse-scale models for large systems (e.g., chemical/biological reactors). Further, they can be used for the optimization-based design of micro-scale systems (e.g., bio-microfluidics and advanced manufacturing using multiphase flows). We need physically accurate models and simulations that capture the varying time and length scales of phase interactions to achieve this understanding.

Interface resolved two-phase modeling can be characterized into two main categories – sharp-interface methods and diffuse–interface methods. The sharp interface methods rely on representing the interface with a sharp, discontinuous function (e.g., Volume of Fluids (VOF)), whereas the diffuse interface method involves using a diffuse continuous representation of the interface (e.g., conservative diffuse interface, Cahn–Hilliard Navier–Stokes (CHNS)).

The class of sharp interface methods has been state-of–the–art for high-fidelity simulations of turbulent multiphase flows [1, 2, 3, 4]. These methods involve solving a PDE for transporting the discontinuous sharp volume fraction function. An interface reconstruction procedure is generally required to construct normals. The sharp interface methods suffer from artificial breakup – also known as numerical surface tension, when the interfacial features (droplets/filaments) are comparable in length scale $r$ to the grid size $\Delta x$ [5]. There has been some progress in the development of computational techniques that reduces this effect involving interface reconstruction techniques [6, 7], Moment-of-Fluid methods [8], Manifold death algorithms [9]. However, these methods are limited to sharp interface methods and are still computationally complex.

Diffuse interface methods, particularly CHNS, provide a diffuse interface representation and avoid any interface reconstruction process. However, diffuse interface methods—under low mesh resolutions—also suffer from artificial breakup and mass loss, but for completely different reasons. The limitation arises because of the thickness of the diffuse interface $\epsilon$. In the case of CHNS, when the ratio of $\epsilon/r$ is $O(1)$, it can lead to bound violations in addition to artificial breakup, and these structures might disappear and get absorbed into the larger structures (due to coarsening of Cahn-Hilliard).

So, both the sharp and the diffuse interface methods for insufficient resolution face the challenge of artificial breakup. The main contributions of this work are as follows:

1) We propose a novel scalable algorithms to identify regions of interest, i.e., regions where $\epsilon/r \approx O(1)$ and selectively increase the mesh resolution in these identified regions. Providing targeted resolution is an important feature for performing cost-effective simulations

2) In the case of the multiphase simulations, the levels of the mesh can vary by several orders of magnitudes to resolve the interfacial and flow physics. We tailor existing octree refinement and coarsening algorithms to accelerate remeshing and decrease the associated overhead, especially for multi-level refinement, where the element sizes drop substantially. This contrasts existing approaches, where refinement or coarsening of the octrees is done level by level. [10, 11, 12, 13, 14, 15]

3) Further, we analyze the data movement during matrix and vector assembly of FEM operators and propose a new

---

§ Equal contribution
data layout for local matrix and vector assembly that is more amenable to avoiding cache misses.

4) We showcase the application of the proposed algorithm for the canonical problem of primary jet atomization, where we perform a simulation equivalent to 35 trillion grid points on a uniform mesh – 64× more resolved than the current state–of–the–art simulation.

II. Methods

In this work, we present algorithmic developments that are applicable for all interface capturing methods, however, we specifically deploy a thermodynamically consistent Cahn–Hillard Navier–Stokes (CHNS) for the numerical simulations.

A. Numerical methods

Governing equations:

Let \( \phi \) be the phase field variable that tracks the location of the phases and varies smoothly between +1 to -1 with a characteristic diffuse interface thickness \( \epsilon \), and \( v_i \) be the \( i^{\text{th}} \) component of the mixture velocity of the two phases. A thermodynamically consistent coupled CHNS equations [16] are written as follows:

1) Momentum equation:

\[
\frac{\partial (\rho(\phi)v_i)}{\partial t} + \frac{\partial (\rho(\phi)v_j v_i)}{\partial x_j} + \frac{1}{Pe} \frac{\partial (J_i v_i)}{\partial x_j} + C_n \frac{\partial}{\partial x_i} \left( \frac{\partial \phi}{\partial x} \frac{\partial \phi}{\partial x_j} \right) + \frac{1}{We} \frac{\partial \phi}{\partial x} \left( \frac{\eta(\phi)}{\partial x_j} \right) \frac{\partial \phi}{\partial x} + \phi \rho_0 \frac{\partial \phi}{\partial x} = 0,
\]

where, \( J_i = \frac{\rho_+ - \rho_-}{2 \rho_+ C_n} m(\phi), \)

(1)

2) Solenoidality and Continuity:

\[
\frac{\partial \phi}{\partial x_i} = 0, \quad \frac{\partial \rho(\phi)}{\partial t} + \frac{\partial (\rho(\phi)v_i)}{\partial x_i} + \frac{1}{Pe} \frac{\partial J_i}{\partial x_i} = 0,
\]

(2)

3) Cahn–Hilliard Equations

\[
\frac{\partial \phi}{\partial t} + \frac{\partial (v_i \phi)}{\partial x_i} - \frac{1}{Pe C_n} \frac{\partial}{\partial x_i} \left( m(\phi) \frac{\partial \phi}{\partial x_i} \right) + \mu = \sigma(-\phi^2),
\]

where \( \rho(\phi) \) is the non–dimensional mixture density given by \( (\rho_+ - \rho_-/2\rho_+ + \rho_+ + \rho_-/2\rho_+) \), the non–dimensional mixture viscosity \( \eta(\phi) \) is given by \( (\eta_+ - \eta_-/2\eta_+) \). We use a degenerate mobility \( m(\phi) \) given by \( \sqrt{1 - \phi^2} \).

Non-dimensional parameters are as follows: Peclet, \( Pe = \frac{u L}{\sigma} \); Reynolds, \( Re = \frac{u L}{\nu} \); Weber, \( We = \frac{\rho_+ u^2 L}{\sigma} \); Cahn, \( C_n = \frac{\epsilon}{L} \); and Froude, \( Fr = \frac{u^2}{g L} \), with \( u_r \) and \( L_r \) denoting the reference velocity and length, respectively.

Numerical discretization:

In this work, we utilize the two–block projection scheme proposed in Khanwale et al. [16]. The proposed scheme is based on projection-based pressure splitting of Navier–Stokes (NS). Overall, the scheme involves four solutions per block:

1) CH-Solve: As the first step, we solve the advective Cahn–Hillard equation. This is solved as a non–linear solver in a fully implicit manner.

2) NS-Solve: We compute mixture density and viscosity as a function of \( \phi \) from the CH-Solve in the previous step and use it to solve the momentum equations. We use a semi–implicit discretization based on the Crank–Nicolson scheme that results in linearized momentum equations.

3) PP-Solve: The projection-based splitting results in a variable density pressure Poisson equation that allows us to compute pressure at the current time level.

4) VU-solve: As the last step for each block, we correct the velocity predicted in the NS-solve using the Helmholtz equations to get the solenoidal velocity.

Remark. To minimize the memory footprint, we split the final VU-solve to update velocity once per direction. This reduces the assembled matrix size from \( N \times DIM \times k \) to \( N \times k \), where \( N \) is the total number of independent DOFs, \( k \) is the number of non–zeros per row, and \( DIM \) is the spatial dimension (2D/3D). We note that the mass matrix, as a result of VU-solve, does not need to be recomputed for each of the DIM separately and is reused till the mesh does not change. Once the mesh is assembled, no subsequent call to Mat_Assembly_Begin and Mat_Assembly_End is made, reducing the preconditioner computation cost.

Remark. Although the proposed numerical scheme can be integrated with arbitrary order basis functions with Continuous Galerkin (CG) elements, we have restricted the deployment of the numerical algorithm to the linear basis functions (spatially second-order convergence).

Local Cahn:

The diffuse interface methods, including CHNS, under low mesh resolution, suffers from artificial breakup and mass loss. Particularly for CHNS, this limitation arises when the interface thickness \( \epsilon \) has a comparable length scale to the length scale of fluid features (droplets/filaments/sheets), leading to the artificial breakup. Our simulations use the \( C_n \) number as a non-dimensional \( \epsilon \). A smaller \( C_n \) represents a thinner diffuse interface which allows the capture of more accurate physics. A smaller \( C_n \) compared to the other interface features would prevent an artificial breakup. However, a decrease in \( C_n \) everywhere comes at an additional cost of increased mesh resolution. A rough estimate indicates that a 2× decrease in \( C_n \) leads to approximately 8× increase in mesh resolution. In this work, we develop novel scalable algorithms to tackle this issue, where we selectively decide the regions where the interface thickness has become comparable to the length scale of interfacial features and reduce \( C_n \) only in those regions. We refer to this approach as “local Cahn”. The current state–of–the–art approaches for CHNS simulation relies on a constant Cahn number throughout the domain. This approach balances
the accurate capture of relevant physics and the controlled increase in mesh resolution, which is of utmost importance in tackling engineering problems.

B. Identification of key features

As we have discussed earlier, identifying key features such as thin filaments, sheets, and small drops is quintessential for accurate physics modeling. Sec. II-A discusses the importance of accurately detecting these key features and decreasing the Cahn locally. Decreasing \( Cn \) at all interfacial regions can lead to intractable problems due to increased mesh resolution.

We now present algorithms to detect such regions of interest. We first highlight the salient features of the algorithm on a uniform mesh before discussing the parallel deployment on octree meshes.

1) Uniform mesh case:

Our algorithm is inspired by the classic image processing idea of erosion and dilation. Fig. 1 briefly shows the algorithmic idea behind the approach for a drop (Fig. 1a) and an elongated filament (Fig. 1b).

We perform a series of following steps to achieve the desired goal:

1) The phases in CHNS are described by a continuously changing variable \( \phi \) ranging from -1 to 1. As a first step, we convert the image from continuous \( \phi \) to a binary 0/1 representation \( \phi_{BW} \) by thresholding it at a certain value (\( \delta \)): (denoted by \( T(\phi) \) in Fig. 1).

\[
\phi_{BW} = \begin{cases} 
1 & \phi \leq \delta, \\
0 & \phi > \delta.
\end{cases}
\]

The value of \( \delta \) is decided such that the immersed phase gets the value of 1 and the bulk phase gets 0. We used a value of \( \delta \) of 0.8 or -0.8 depending on the value of immersed phase.

2) We then perform a series of erosion steps to shrink the region of immersed (\( \phi_{BW} = 1 \)) phase. (denoted by \( E(\phi) \) in Fig. 1).

3) Subsequently, a series of dilation steps are performed to expand the region of \( \phi_{BW} \) (denoted by \( D(\phi) \) in Fig. 1). We note that we need to perform more dilation than erosion to incorporate the shrinking of the interface encountered in the first step. Our experiments showed that 3 – 4 extra dilation steps are typically sufficient.

4) Finally we compute the difference of the pixels after the dilation (\( \phi_{BW,d} \)) and \( \phi_{BW} \) (denoted by \( S(\phi) \) in Fig. 1). The pixels marked as 1 in \( \phi_{BW} \) and 0 on \( \phi_{BW,d} \) constitute the region of interest.

5) As the final step, we decrease the Cahn in the regions identified in the previous step and refine the interface region (|\( \phi | < \delta^* \)) with the appropriate resolution.

2) Challenges with octree mesh:

The above steps work well on a uniform mesh with accessible neighbors; however, performing these steps in a parallel distributed setting on adaptive meshes poses the following challenges:

1) The octree-based mesh has elements with non– uniform sizes.

2) CG-based finite element (FEM) data–structures may not have neighbor information. This is an issue with unstructured mesh, as elements can have a varying number of neighbors with no plausible upper limits.

3) Hanging nodes involve the interpolation of values from parent to child elements. So, the \( \phi_{BW} \) value will not have only two distinct values, 0 and 1, but can have arbitrary values between 0 and 1, depending on the type of hanging nodes – edge hanging or face hanging in 3D.

3) Parallel deployment on octree meshes

We extend the idea of the approach illustrated in Sec. II-B1 to the octree meshes in parallel settings, albeit with some modifications.

- The first modification involves changing the limits of \( \phi_{BW} \). We modify \( \phi_{BW} \) as:

\[
\phi_{BW,o} = \begin{cases} 
1 & \phi \leq \delta, \\
-1 & \phi > \delta.
\end{cases}
\]

The above modification in the definition of \( \phi_{BW,o} \) is purely a mathematical convenience in detecting the interface elements and helps in subsequent erosion and dilation algorithms.

- Interface elements: An element contains interface when:

\[
\sum_{i=1}^{\text{nodes}} \phi_{BW,o} \neq \text{nodes}
\]  

• Erosion: Erosion step involves visiting the interfacial element (Eq. (5)) and making all the nodal values to be -1.

• Dilation: Dilation step involves the reverse of the erosion step. It involves visiting the interface element and making all the nodal values +1.

Remark. We note that erosion and dilation involve the call to INSERT_VALUES rather than ADD_VALUES. So, the algorithm can proceed without any issue, even if multiple elements try to write to the same nodal location.

With these modifications, we are ready to define the key steps associated with identifying the features of interest in octree-based meshes. Algorithm 1 briefly the major steps involved in identifying the local regions of interest and locally decreasing the \( Cn \) number. As the first step, we begin by converting the continuous phase field variable \( \phi \) to \( \phi_{BW,o} \). We then impose erosion and dilation as a series of MATVEC operations. (Algorithm 2) 1 1 To perform each erosion step, we make a single pass over the elements in a distributed fashion, with each processor looping over its own set of local elements. When the element containing the interface is visited, the erosion step is triggered, which involves converting all the nodal values of that element to -1. As previously

1 MATVEC operations are at the heart of FEM computations and has been shown to have excellent scaling [11, 17]. Each MATVEC operation involves a single pass over all the local elements with associated ghost exchange. We note that these ghost exchange communication is typically overlapped with computation [11, 17, 18, 19].
mentioned, the octree mesh contains elements at various levels of refinement. To address this challenge, we define a base level \(b_1\), typically the finest level in the octree mesh. Additionally, we maintain a counter on how often the element has been visited for erosion. An erosion is triggered when the counter equals the \((b_1 - l)\), where \(l\) is the current level of the octree mesh. For example, an element two times coarser than the finest mesh, i.e., \(b_1 - l = 2\), will have to wait for two iterations before erosion can finally be triggered. This gives a way to balance the difference in levels of octree mesh. Similarly, as before, we then perform a series of dilation steps to follow the erosion steps. The dilation is also triggered similarly to account for the difference in octree levels. We perform a larger number of dilation steps compared to the erosion steps to compensate for the thresholding encountered during \(\phi_{BW,o}\), thus ensuring that no region that is a part of a larger region gets marked for high refinement, leading to a drastic increase in the problem size. Finally, we make a last pass over the elements and mark it with reduced Cahn based on the following condition:

\[
\sum_{i=1}^{\text{nodes}} \phi_{BW,o} = \text{nodes} \quad \& \quad \sum_{i=1}^{\text{nodes}} \phi_{BW,d} = \text{nodes}
\]  

Eq. (6) states that we need to reduce Cahn in the element, which has all the nodes marked as +1 after thresholding and -1 after extra dilation. This is analogous to the view in the uniform mesh, except we are looking at element by element -1 after extra dilation. This is analogous to the view in the elemental vector (Algorithm 4). This is done by first converting the elemental \(C_n\) vector to the nodal \(C_n\) vector and performing the subsequent erosion and dilation steps on the nodal vectors, as previously discussed. The erosion steps ensure that all the very small zones of reduced \(C_n\) are removed. Similar to erosion, we perform more dilation steps than erosion. The dilation steps pad the region of local \(C_n\) elements. These padded elements serve as a potential candidate that would contain the key features of interest till the next level of local \(C_n\) identification is performed. We note that padding the elements does not increase refinement as the refinement is only performed at the interface. Even if the pure phase region (away from the interface) gets the decreased \(C_n\), no refinement is triggered as we refine only near the interface. Once we have identified the elements for reducing the local \(C_n\), we then proceed to refine the regions of the interface marked by \(|\phi| < \delta\). We note that the mesh is refined only in the region near the interface, even on the elements with reduced Local \(C_n\).

The overall complexity of the proposed approach still scales as \(O(N)\) per dilation and erosion step, which is of the same complexity as with the uniform case. Thus, even without neighbor information, we are able to achieve the same asymptotic complexity. Although we only demonstrate the algorithm deployment for octree-based meshes, this can also be extended for any other class of unstructured meshes. The algorithm proposed relies on efficient \textsc{matvec} operations, which also form the core kernel for FEM operation. Thus, it can be applied to any unstructured FEM code without any major changes to the framework and does not rely on any additional data-structure like neighbors information, which can be difficult to get in the case of unstructured meshes. Furthermore, although we have demonstrated the applicability

\textbf{Algorithm 1} LOCALCAHNNIDENTIFIER: Identify regions of Local Cahn

\begin{algorithm}
\begin{itemize}
  \item [Require:] Phase field vector \(\phi\), Reference level \((b_1)\)
  \item [Ensure:] Elemental Local \(C_n\)
  \begin{itemize}
    \item [1:] \(\phi_{BW,o} \leftarrow \text{THRESHOLD}(\phi_{BW})\) \hfill \text{\&} \text{ Eq. (4)}
    \item [2:] \(\phi_{BW,e} \leftarrow \text{ERODELATE}(\phi_{BW,e}, \text{EROSION, numErodeSteps}, b_1)\) \hfill \text{\&} \text{ Algorithm 2}
    \item [3:] \(\phi_{BW,d} \leftarrow \text{ERODELATE}(\phi_{BW,d}, \text{DILATE,numDilateSteps}, b_1)\) \hfill \text{\&} \text{ Algorithm 2}
    \item [4:] \(\text{elemental_Cahn} \leftarrow \text{ELEMENTALCAHNN}(\phi_{BW,o}, \phi_{BW,d}, \text{Cahn1,Cahn2})\) \hfill \text{\&} \text{ Algorithm 3}
    \item [5:] \(\text{elemental_Cahn} \leftarrow \text{ERODELATECAHNN}(\text{elemental_Cahn, b_1, Cahn1, Cahn2})\) \hfill \text{\&} \text{ Algorithm 4}
  \end{itemize}
\end{itemize}
\textbf{return} elemental\(_\text{C_n}\)
\end{algorithm}
of the algorithm to identify the regions with only two levels of local $C_n$, it can be easily extended to multi-level $C_n$. Each level of $C_n$ will have its own set of numbers of erosion and dilation steps.

C. Refinement & Intergrid transfer

1) Adaptive refinement algorithms

As the interface moves and becomes more complex over time, certain domain regions require greater resolution to maintain accuracy. The needed level of detail can also change suddenly due to a large timestep.

We enforce the adaptive mesh resolution in a scalable manner with an octree-based approach. Adaptive refinement in octrees amounts to substituting sets of leaves by their ancestors or descendants. We adopt similar principles as state-of-the-art octree meshing libraries [11, 17, 19, 20]. Namely, linearization, 2:1-balancing, supporting incomplete octrees and deducing element-to-nodal relations hierarchically on the fly. This paper focuses on the aspects of octrees relevant to refining by multiple levels at once. Due to the drastic changes in the interface, the transformed octree may have leaves that are not direct children of leaves in the input tree but descendants at an arbitrarily greater depth.

a) Refinement in serial

Algorithm 5 takes a sorted leaf set and replaces each leaf with its descendants at the level specified, in sorted order. This algorithm is elegant in that, even without pointers marking the leaves of subtrees, the octree traversal passes through the input leaves only once.

The traversal begins at the root. If the current input octant needs refining below the current traversal depth, then child subtrees are traversed; otherwise, the current subtree root is emitted. After processing the descendant subtree of the current octant, the input pointers are advanced. The next subtree is traversed only if it overlaps the next input octant.

Note that $o_{in}$, $l_{in}$, and $o_{out}$ are passed by reference, while $R$ and sfc are passed by value.

\begin{algorithm}
\caption{ERODE\textsc{DILATE}: Erosion and dilation step}
\begin{algorithmic}
\Require Nodal vector $\phi_{BW,o}$, Stage (EROSEION/DILATION)\\
\Ensure Nodal vector $\phi_{BW,e}$ or $\phi_{BW,d}$, depending on the stage
\end{algorithmic}
\begin{algorithmic}
\Function{eroodeilate}{vec, $\phi_{BW}$}
\State $\phi_{BW,o}$ \Comment{Value to set for Erosion}
\If{$\text{Stage} == \text{EROSION}$}
\State $\text{val} \leftarrow -1$
\Else
\State $\text{val} \leftarrow +1$
\EndIf
\For{$i \leftarrow 0$ to $\text{num}_\text{steps}$}
\State $\text{vec}_\text{ghosted} \leftarrow \text{GhostRead}(\text{vec})$
\State $\text{vec}_\text{temp} \leftarrow \text{vec}_\text{ghosted}$ \Comment{Temporary vector}
\State $\text{counter} \leftarrow 0$
\For{elem \leftarrow local\_elems}
\State $\phi_{\text{elem}} \leftarrow \text{vec}_\text{ghosted}[\text{elem}]$ \Comment{Copy the nodal values}
\State $\phi_{\text{Sum}} \leftarrow 0$
\For{node \leftarrow $\text{num}$\_local\_nodes}
\State $\phi_{\text{Sum}} \leftarrow \phi_{\text{elem}}$ \Comment{Copy the nodal values}
\EndFor
\EndFor
\If{$\text{has}\_\text{Interface} \leftarrow (abs(\phi_{\text{sum}}) \neq \text{num}\_\text{local}\_\text{nodes})}$}
\If{$\text{has}\_\text{Interface}$}
\State $l_c \leftarrow \text{level}[\text{elem}]$
\If{$\text{counter}[\text{elem}] == l_c - b_i$}
\Comment{Check for level}
\EndIf
\EndIf
\EndIf
\For{node \leftarrow $\text{num}$\_local\_nodes}
\State $\text{vec\_temp}[\text{elem}][\text{node}] \leftarrow \text{val}$
\EndFor
\State $\text{vec}_\text{ghosted} \leftarrow \text{vec}_\text{temp}$
\State $\text{vec} \leftarrow \text{GhostWrite}(\text{vec}_\text{ghosted})$
\Return vec
\EndFunction
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\caption{ELEMENTAL.CAHN: Identify the element with local $C_n$}
\begin{algorithmic}
\Require Binary phase field vector $\phi_{BW,o}$, Phase field vector after dilation $\phi_{BW,d}$, $C_n$ values ($C_{n1}$, $C_{n2}$): ($C_{n1}$ < $C_{n2}$)
\Ensure Elemental Local $C_n$
\Function{elemental_cahn}{vec, $\phi_{BW,o}$}
\State $\phi_{BW,o}$ \Comment{Copy the nodal values}
\For{elem \leftarrow local\_elems}
\State $\phi_{\text{elem}} \leftarrow \text{vec}_\text{ghosted}[\text{elem}]$ \Comment{Copy the nodal values}
\EndFor
\For{node \leftarrow $\text{num}$\_local\_nodes}
\State $\phi_{\text{elem}} \leftarrow \phi_{\text{ghosted}}[\text{node}]$
\State $\phi_{\text{elem}} \leftarrow \text{vec}_\text{ghosted}[\text{elem}]$ \Comment{Copy the nodal values}
\EndFor
\State $\phi_{\text{elem}} \leftarrow 0$
\State $\phi_{d} \leftarrow 0$
\For{node \leftarrow $\text{num}$\_local\_nodes}
\State $\phi_{\text{elem}} \leftarrow \phi_{\text{ghosted}}[\text{node}]$
\If{$\phi_{\text{elem}} == \text{num}_\text{local}\_\text{nodes}$}
\State $\text{elemental}$\_\text{C}_n[\text{elem}] = $C_{n2}$
\Else
\State $\text{elemental}$\_\text{C}_n[\text{elem}] = $C_{n1}$
\EndIf
\EndFor
\Return $\text{elemental}$\_\text{C}_n
\EndFunction
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\caption{ERODE\textsc{DILATE}CAHN: Expand $C_n$ regions and remove islands}
\begin{algorithmic}
\Require elemental\_\text{C}_n, Reference level ($b_i$)
\Ensure Elemental Local $C_n$
\Function{eroodeilatecahn}{vec, $\phi_{BW}$}
\State $\phi_{BW,o}$ \Comment{Copy the nodal values}
\For{elem \leftarrow local\_elems}
\State $\phi_{\text{elem}} \leftarrow \text{vec}_\text{ghosted}[\text{elem}]$ \Comment{Copy the nodal values}
\EndFor
\For{node \leftarrow $\text{num}$\_local\_nodes}
\State $\phi_{\text{elem}} \leftarrow \text{vec}_\text{ghosted}[\text{node}]$
\State $\phi_{\text{elem}} \leftarrow \phi_{\text{ghosted}}[\text{node}]$
\EndFor
\State $\phi_{\text{elem}} \leftarrow 0$
\State $\phi_{d} \leftarrow 0$
\For{node \leftarrow $\text{num}$\_local\_nodes}
\State $\phi_{\text{elem}} \leftarrow \phi_{\text{ghosted}}[\text{node}]$
\EndFor
\State $\phi_{\text{elem}} \leftarrow \text{vec}_\text{ghosted}[\text{elem}]$ \Comment{Copy the nodal values}
\State $\phi_{d} \leftarrow \phi_{\text{ghosted}}[\text{node}]$
\If{$\phi_{\text{elem}} == \text{num}_\text{local}\_\text{nodes}$}
\State $\phi_{\text{elem}} \leftarrow \text{num}_\text{local}\_\text{nodes}$
\Else
\State $\phi_{\text{elem}} \leftarrow \text{vec}_\text{ghosted}[\text{elem}]$
\EndIf
\State $\text{elemental}$\_\text{C}_n[\text{elem}] = $C_{n2}$
\Else
\State $\text{elemental}$\_\text{C}_n[\text{elem}] = $C_{n1}$
\EndIf
\Return $\text{elemental}$\_\text{C}_n
\EndFunction
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\caption{REFINE: Replace leaf octants by sorted descendents}
\begin{algorithmic}
\Require List of leaf octants and desired levels, with pointers $o_{in}$ and $l_{in}$.
\Ensure $o_{out}$ contains the descendants of $o_{in}$ at level $l_{in}$.
\Function{refine}{vec, $R$, and sfc}
\State $o_{out} \leftarrow 0$
\State $o_{in} \leftarrow 0$
\EndFunction
\end{algorithmic}
\end{algorithm}
Obtaining a refined set of octants can be done independently on each process, so we do not discuss a parallel version. However, once the refinement is completed, the 2:1-balance condition must be restored after the refinement.

Refining octants near the domain boundary can create void descendants, either directly due to subdivision or as an aftereffect of 2:1-balancing. Void descendants of boundary-intercepted octants need to be discarded.

b) Coarsening in serial

Analogously to refinement, the octree can be coarsened by various levels throughout the domain. An integer is attributed for the coarsest acceptable level to which each octant can be promoted. Unlike refinement, the decision to coarsen a subtree depends on the inputs of all descendants. Consensus is defined based on two requirements: An ancestor \( A \) of an input leaf is output if (i) no descendants of \( A \) in the input require \( A \) to be refined and if (ii) the same cannot be said of the parent of \( A \).

Algorithm 6 traverses the tree while iterating in order over the input—just as for refinement—but here, the output is pushed and popped at every subtree, rather than the leaves only. Per requirement (i), the inputs within the current subtree must be read before deciding whether the root of the subtree is too coarse. Therefore, the root is emitted in the post-order action only after checking all descendants. On the other hand, by (ii), the root of the current subtree could be too fine, depending on inputs outside the subtree. Therefore, we leave it to the parent to retract emitted roots of child subtrees if needed.

In [19] serial coarsening and refinement algorithms are discussed, including multi-level variants. Compared to our algorithm (6), theirs accepts a slightly more general coarsening criterion, expressed as a Boolean callback function \texttt{DoCoarsen} that is dynamically evaluated on every subtree. In contrast, our algorithm is tailored to the case that the application declares up front the maximum change in level that should be allowed for each octant. Our algorithm, however, removes the requirement that subtrees be complete. Otherwise, both algorithms have an equivalent structure, featuring a post-order tree traversal, a single pass through the input, and writing outputs through a pure stack interface (push/pop).

Next, we present algorithms to coarsen in parallel and transfer functions across multiple levels of refinement in parallel.

c) Coarsening in parallel

For the same reasons that coarsening in serial is more complicated than refinement, coarsening in parallel requires extra communication to guarantee consensus.

Algorithm 7 proceeds as follows. The globally sorted input octree undergoes a local coarsening pass resulting in a locally linearized incomplete tree. The results are tentative, as the output octants achieve consensus locally on a process, but may not represent the global consensus. Furthermore, coarse octants might be duplicated even if the results agree across processes. The input must be slightly repartitioned before the second coarsening pass to prevent conflicts. Note that the required partitioning resolution for multi-level coarsening is known only after the first coarsening attempt.

After the first local coarsening pass, the next step is finding the overlap region between consecutive processes. Ideally, the tentative coarse octants do not overlap between two consecutive processes \( p_i \) and \( p_{i+1} \). If so, the inputs on processes \( p_{j \leq i} \) are independent from those on processes \( p_{j > i} \). Otherwise, the worst case for two processes is for a single coarse octant on one process to overlap one or more octants on the other process.

Note that incomplete trees require special handling compared to complete trees. For complete trees, conflicts can be handled using a simple merge strategy. However, for incomplete trees, a more sophisticated algorithm is required to ensure consistency.

Algorithm 7 \texttt{PARCOARSEN}: Replace distributed leafs by partitioned and sorted ancestors

\textbf{Require}: List of leaf octants and desired levels, \texttt{octin} and \texttt{levelin}, MPI communicator \texttt{comm}.

\textbf{Ensure}: Returns the ancestors of \texttt{octin} at the consensus of \texttt{levelin}, with duplicates removed.

\begin{algorithm}
\begin{algorithmic}
\State \textbf{first coarsening pass}
\State \textbf{exchange tentative coarse octants at partition endpoints}
\State \textbf{find overlapped inputs}
\State \textbf{repartition overlapped inputs (omitted: also repartition levels)}
\State \textbf{second coarsening pass}
\end{algorithmic}
\end{algorithm}
be resolved by globally sorting the tentative coarse octants, then locally splitting any coarse octants until all overlaps are removed. For incomplete trees, the same splitting procedure could produce octants in void regions, leading to unnecessary fill-in. There are several possible solutions.

Option one is to reapply the domain test after splitting coarse octants. The advantage here is that no further communication is needed after the partitioning step. But, the cost of the domain test may not be optimal for pinpoint tests at partition endpoints; rather we treat the cost as amortized over bulk queries during startup and refinement. Also, the domain test is completed redundant for coarsening because the input octree already contains the needed information.

Option two is to send each split octant back to the process(es) whose input could overlap that octant and have the originating process(es) figure out whether the octant is void or not. Equivalently, the overlapped input could be requested, with void octants discarded on the same process that performed the splits. (Multiple processes might need to be consulted, but only if the input grain size is very small). Either way, the disadvantage is that a round of ping-pong messages is needed after the partitioning step.

Option three is to repartition the input directly toward the coarsest contender of each conflict. This solution cuts out the lag time needed to fully partition and split tentative coarse octants before resolving conflicts. As a result, communication is limited to an exchange of tentative coarse octants at the partition endpoints and a repartition of overlapped inputs. The new partition makes it just as if there were no overlaps to begin with. Coarsening can be finished independently of other processes, and only the exchanged regions need to be recomputed after the exchange.

We have taken the approach of option three. Although option two can perform partitioning at a finer granularity, it does not promise to restore load balance any better than option three. We consider proper load balancing a separate step in any case, especially since the subsequent restoration of 2:1-balance tends to affect the load. Therefore, we choose option three, which both avoids redundant domain tests and streamlines communication.

In the rare case that a tentative octant is so aggressively coarsened that it overlaps multiple remote partitions, the initial coarse octant exchange can be considered the first step of a distributed exponential search to find the processes containing the region of overlap. The details of this adjusted algorithm are omitted for brevity.

Proof of worst-case overlap: It is claimed above that, after local coarsening, the worst case overlap for two processes is that one or more octants are overlapped by a single remote coarse octant. The proof hinges on the assumption that the input to local coarsening is globally sorted: Suppose process \( p \) precedes process \( q \), and last two octants of \( p \) are \( s_p < t_p \) while the first two octants of \( q \) are \( s_q < t_q \). Now suppose, contrary to the claim, that \( s_p \) and \( s_q \) overlap, and \( t_p \) and \( t_q \) overlap. It must be that \( s_q \) does not overlap \( t_p \); for otherwise \( s_q \) would be an ancestor of \( t_p \) and thus would overlap \( t_q \), which is impossible because \( s_q \) and \( t_q \) belong to the same locally linearized tree. Since \( s_q \) overlaps \( s_p \) and \( s_p < t_p \), the pre- or post-order implies \( s_q \) precedes \( t_p \) (still without overlapping). But, this implies that the inputs descending from \( s_q \) precede the inputs descending from \( t_p \). In other words, the input is not globally sorted, contradicting the assumption that it is.

2) Distributed multi-level inter-grid transfer

After updating the resolution of the octree mesh on a new time step (and restoring 2:1-balance), data from the previous grid must be transferred to the resolution and partition of the new grid.

Octree-based meshing frameworks typically support inter-grid transfers only between successive levels [21]. This constraint simplifies the implementation because it means overlapping cells differ in size by a factor of at most two. However, for the problems with moving interfaces and suddenly changing refinement patterns, constructing intermediate grids one level at a time creates significant overhead. This section describes our methods to transfer data after refining or coarsening an arbitrary number of levels.

a) Serial algorithms

Transferring coarse-to-fine cell-centered data is the simplest case. We traverse the octants of both grids in the SFC order simultaneously. Once a leaf is reached in the coarse grid, all cells overlapped in the fine grid are accessible. The value in the coarse cell is imposed on all the overlapped fine cells.

Coarse-to-fine interpolation of node-centered data begins similarly. Octants of both grids are traversed together. Then, at a coarse leaf, nodal values on all overlapped fine cells can be interpolated from the coarse cell nodes. Using mesh-free techniques [17], the coordinates of the relevant nodes are available in a contiguous list at this point in the traversal. The transfer is done by evaluating the field in the coarse element at each of the fine nodes. If neighboring subtrees redundantly interpolate a node, the results are consistent in each instance; the final value is arbitrarily picked from one of the instances.

Fine-to-coarse transfers follow essentially the same structure, except mirrored. As multiple fine unknowns are aggregated into fewer coarse unknowns at a coarse leaf, the exact aggregation function depends on the differential operator and the type of transfer. Cell-centered values might be averaged. Node-centered values might be simply injected or otherwise weighted by the transpose of the interpolation operator.

The advantage of using mesh-free grid traversals here is that the nodes need not be initially arranged in any particular order. The only requirement is that nodal values are tagged by their unique location code key. The keys make it possible to route the appropriate nodes to each subtree during the traversal. This property is needed by our distributed memory approach of sending nodes—with their keys—to interested processes in a detached manner.

b) Parallel algorithm

The serial grid transfer algorithms can be applied in distributed memory after the partitions are somehow matched between the two grids.
Our inter-grid transfer algorithms are structured in four steps: First, search for grid-grid overlaps in the splitter table. Second, detach and send coarse element nodes to the fine grid. Third, execute serial inter-grid interpolation or aggregation. Fourth, if coarsening, receive coarse element data from the fine grid and attach it.

It is possible to communicate either before or after the local inter-grid transfer. We choose the order that communicates the coarse data. This helps reduce the total message size. Also, since the number of fine elements dominates the local inter-grid transfers, this strategy uses the fine partition to keep the computational workload balanced.

c) Extension of rank search from octants to overlaps

The function \( \text{rank}_X(e) \equiv |x \in X : x \sim e| \) on a totally ordered set \((Y,\sim), X \subset Y\) guarantees that rank is non-decreasing on a sorted subset of \(Y\). This property suggests the desired key can be found by narrowing down an interval, one of the main ideas behind the binary search. To apply binary search algorithms to find overlaps between two arbitrary leaf sets, extending the notion of rank from octants to a kind of overlapping region is useful. Specifically, from the total ordering of octants, this section constructs a new total ordering over equivalence classes whose members share a common ancestor.

This section also uses the properties of hierarchical ordering and leaf sets:

1. **Hierarchical octant ordering**: Let \(a, x, y\) be octants such that \(a\) is an ancestor of \(x\) but not of \(y\). Then \(y < a \iff y < x\). Equivalently, \(a < y \iff x < y\). (Both preorders and postorders have this property.)

2. **Leaf set**: If \(G\) is a grid and \(L = \text{leaves}(G)\), then \(\forall x \in L\), the only ancestor of \(x\) in \(L\) is \(x\) itself.

Let \(G\) and \(H\) be two leaf sets. Define \(L = G \cup H\).

For any \(x, y \in L\), define the equivalence relation \(x \sim y\) to mean there is \(a \in L\) that is an ancestor of both \(x\) and \(y\). Reflexive: \(\forall x \in L, x \sim x\). Symmetric: The definition only uses the expression of both \(x\) and \(y\), which is symmetric. Transitive: Let \(x, y, z \in L\) with \(x \sim y\) via ancestor \(u \in L\), while \(y \sim z\) via ancestor \(v \in L\). Since \(u\) and \(v\) are both ancestors of \(y\), one of \(u\) or \(v\) is an ancestor of the other, and thus of \(x\) and \(z\) as well.

Each equivalence class of \(\sim\) has a coarsest representative. This can be seen by cumulatively repeating the transitivity argument over the whole class, producing an ancestor shared by all members of the class.

The picture is especially clear-cut for two leaf sets, as \(L\) is defined above. For two leaf sets, the coarsest element belongs to one grid, and any overlapped finer elements are disjoint from each other and belong to the other grid. Additionally, for two leaf sets, it is immediately apparent whether elements from different grids belong to the same class: If so, they overlap. The common ancestor of two class members comes from one of the leaf sets; thus it must equal the member from the same leaf set.

Next is a proof that the equivalence classes are totally ordered. Take a total ordering \(\le\) that has the hierarchical property (such as a pre- or post-order SFC traversal) and define a new relation: For any \(x, y \in L\), define \(x \le y\) to mean that \(x < y\) or \(x \sim y\). This is a quasiorder whose equivalence relation is \(\sim\) (see antisymmetry below). Thus the quotient of \(\equiv\) partially orders the equivalence classes of \(\sim\) by \(\le\), which is a total order. Reflexive: \(\sim\) is reflexive. Antisymmetric: If \(x \sim y\) and \(y \equiv x\), then \(x \sim y\). Otherwise, using the symmetry of \(\sim\), it must be that \(x < y\) and also \(y < x\), which is impossible. Transitive: Let \(x, y, z \in L\) with \(x \le y\) and \(y \le z\). There are four cases. (i) \(x < y\) and \(y < z\). Then \(x < z\). (ii) \(x \sim y\) and \(y \sim z\). Then \(x \sim z\), by transitivity of \(\sim\). (iii) \(x < y\) and \(y \sim z\), via ancestor \(v \in L\). If \(v\) is an ancestor of \(x\), then \(x \sim z\) via \(v\). Else, by the hierarchy property, \(x < y \leftrightarrow x < v \leftrightarrow x < z\). (iv) \(x \sim y\), via ancestor \(u \in L\), and \(y < z\). If \(u\) is an ancestor of \(z\), then \(x \sim z\) via \(u\). Else, by the hierarchy property, \(y < z \leftrightarrow u < z \leftrightarrow x < z\).

The inclusive rank function, \(\text{rank}_X(e)\), maps all members of a class to the same rank. This also holds for the exclusive rank, \(\text{rank}_X(e)\). (The irreflexive kernel of \(\equiv\) is \(x \sim y\), which means that \(x < y\) without \(x \sim y\)). The inclusive rank of \(e\) is the exclusive rank of the next class after that of \(e\).

Finally, it is apparent that searching for grid overlaps is nearly the same as for \(<\) rank. A leaf set sorted by \(\le\) is also sorted by \(\equiv\). Double-rank search then only requires evaluating \(x \equiv y\) for \(x\) and \(y\) from distinct leaf sets. This is possible since, for two grids, \(x \equiv y\) can be evaluated between distinct leaf sets.

d) Intervals of components

Denote every partition’s first and last leaves in grid \(G\) by the arrays \(G^-\) and \(G^+\), such that on process \(r\), any leaf \(g_r\) would fall in the interval \(G^-_r \le g_r \le G^+_r\). Similarly, denote arrays \(H^-\) and \(H^+\) for grid \(H\). Each interval of octants maps to an interval of overlap regions with the total ordering \(\subseteq\). A \(\subseteq\)-interval \(G^-_p \ldots G^+_p\) intersects \(H^-_q \ldots H^+_q\) if, and only if, both \(G^-_p \subseteq H^-_q\) and \(H^+_q \subseteq G^+_p\).

These observations show that the range of processes \(q\) overlapped in \(H\) by the interval in \(G\) owned by process \(p\) would be found with two binary searches: \(\text{rank}_{H^-_q \subseteq (G^+_p)} \le q < \text{rank}_{H^-_q \subseteq (G^+_p)}\). Because the searches only involve partition endpoints, not process-local data, the intersections are detected consistently across processes.

Given one of the processes \(q\) and its partition endpoints in \(H\), the range of overlapping local octants \(G_p[i]\) is found by searching: \(\text{rank}_{G_p[i]}(H^-_q) \le i < \text{rank}_{G_p[i]}(H^+_q)\). In practice, one range starts no earlier than one octant before the end of the preceding range. Then, the coarse elements are selected from each range of octants.

Cell-centered data is already in the correct order and can be sent and received once the partition overlap search is complete. Node-centered data, on the other hand, must be clipped out from the nodal ordering separately for each destination process before it can be communicated.

e) Detaching and attaching coarse node-centered data

The process-to-element overlap range is combined with the element-to-node relation to find the remote interpolation source nodes for each process. The nodes are not detached
element-by-element. That would inflate the message size more than necessary since many elements destined for the same process neighbor each other and share nodes. Instead, for each process, we loop just over the range of elements destined for that process, setting a flag for all nodes on those elements. After the loop, the flagged nodes are gathered and ready to be sent in a contiguous message. This is repeated for all destination processes.

The process of attaching node-centered data onto coarse cells after aggregation is similar. The flags from the elemental loops are reused from the detachment phase. The earlier gather movement is reversed, as the packed nodes are scattered to the positions of the flags.

3) Scalability Checkout

Preliminary runs on up to 30K cores alerted us to some suboptimal usage of MPI collective communication in the previous versions of the code.

a) Sorting octree keys (Allreduce and Alltoally)

Sorting octree keys in distributed memory is a building block of our meshing routines, such as for repartitioning, 2:1-balancing, and enumeration of nonhanging nodes. In our old implementation of the distributed octree sorting algorithm, we experienced poor scalability of splitter search, Allreduce, and Alltoall exchange. In the new version, we adopt a hierarchical k-way staged communication pattern, similar to other hypercube exchange sorting algorithms [18, 22]. In the k-way hierarchical scheme, the number of (super)partitions is kept below a constant k for each of $O(\log_k(p))$ stages. By default $k = 128$, so at most, three stages are required up to 2M processes. Dissociating the partitions from process count reduces storage cost of splitter selection from $O(p)$ to $O(k)$, and data transfer of Allreduce from $O(p)$ to $O(k \log_k(p))$. Performing the Alltoall exchange in stages is also a standard defense against network congestion.

b) Hierarchical communication (Comm_split)

During the setup phase of the k-way hierarchical exchange, the global communicator is subdivided recursively with MPI_Comm_split, such that the final stage involves k or fewer processes. Splitting a communicator is a costly global operation. Doing it repeatedly poses a serious threat to scalability. Since the arguments to MPI_Comm_split are not dependent on the data to be sorted, we memoize the sequence of communicators in an MPI user cache attribute attached to the root communicator. On later executions of distributed octree sort, the saved sequence of communicators is recalled without extra splitting.

c) Remote processing (Sparse Alltoally)

Our nodal enumeration algorithm contains an outsourcing pattern, whereby candidate nodes are sorted globally, checked for duplication and hangingness on remote processes, and sent back to the owners of the originating elements. This algorithm, although not necessarily optimal, is simple and has scaled adequately, following a communication fixup.

The last step of the algorithm uses a concept of return address to return nodes to originating processes. Due to the high-locality heuristic of SFC sorted orders, the number of destination processes is actually sparse. At the same time, the return addresses are treated as if arbitrary. Our first implementation used a raw MPI_Alltoall collective to obtain the receive counts at the destinations. MPI_Alltoall tends to produce heavy network congestion and requires time to populate the send count array, which is linear in process count. We noticed that this step of the algorithm had low overhead up to 28K cores, at which point scaling halted and the overhead blew up 15x from 28K to 56K cores. To fix this we adopted the NBX sparse exchange algorithm from [23], which uses non-blocking collectives and point-to-point messages to eliminate any $O(p)$ primitives like MPI_Alltoall.

D. Matrix & Vector assembly

In this work, we interface our framework with PETSc data-structure to perform linear algebra operations. Specifically, we store the matrix in the form of block storage MATMPIBAIJ. This format has been demonstrated to be much more efficient than the non-block version MATMPIAIJ, specifically for the multi-dof system. The block size, here, refers to the number of DOFs per node.

This storage format, although ideal for the global format, is not suitable for local assembly. This is primarily because of the fact that any operator of the form $L(dof_i, dof_j)$ tends to have the strided memory access. The top row Fig. 2 shows the memory view for the global vector. Any loop over $\{dof_i\}$ in vector assembly will write to memory in strided manner. For example, $(i = 0)$ will visit $0,2,4,6$ and $(i = 1)$ will write to $1,3,5,7$. This is not ideal for optimal performance.

In order to circumvent this issue, we perform a sequence of the following operations:

1) zip: We perform the zip operations for the DOFs, where similar DOFs are arranged contiguously in memory. This is done by making just a single pass over the elemental vector.

2) Perform elemental vector assembly.

3) unzip: Finally, we perform the unzip operations to revert back to the original global format.

Similar to the vector, matrix assembly faces the same issue. The assembly of each DOF writes in the strided assembly. For example, in case of 2D with 2 dof system, $L(dof_0, dof_0)$ would write to

$$d_0 \quad d_1 \quad d_2 \quad d_3 \quad d_0 \quad d_1 \quad d_0 \quad d_1 \quad d_0 \quad d_1$$

Fig. 2. Vector memory layout for a 2 DOF system in 2D: The top row depicts the vector memory layout where DOFs are written in strided memory. zip operations zips all the DOFs together so that they are contiguous in memory whereas unzip operation reverts back into the original memory layout.
0, 2, 4, 6, 16, 18, 20, 22, 32, 34, 36, 38, 42, 44, 46, 48\(^3\). Clearly, we can see that there is a lot of strided access while writing the values in the local vector. We perform a similar \textit{zip} operation to pack all the DOFs together. A good way to look the \textit{zip} operation is that, after the \textit{zip} operation, the matrix size is 16 \times 4, where \((dof_0, dof_0)\) writes to \((0 \cdots 15)\), \((dof_0, dof_1)\) to \((16 \cdots 31)\), \((dof_1, dof_0)\) to \((32 \cdots 47)\) and \((dof_1, dof_1)\) to \((48 \cdots 63)\).

**Remark.** In the case of matrix assembly, an explicit \textit{zip} operation is never performed. We begin each elemental assembly with a matrix filled with 0 entries. So, we perform only \textit{unzip} operations once the local matrix is completely assembled and written back to the global matrix.

Additionally, we extended the idea proposed by Saurabh et al. [10] and exposed each matrix assembly operator as a matrix–matrix multiplication and each vector assembly operator as matrix–vector multiplication. This way of expressing the FEM operator allows us to leverage the benefits of vendor–optimized GEMM and GEMV kernels. We note the \textit{zip} and \textit{unzip} operation is quintessential to express the operator as matrix–matrix or matrix–vector multiplication. This also ensures the portability of the code across different hardware platforms. It forms a middle ground between the portability and explicit vectorization of the FEM operator.

**E. Challenges with parallel settings**

The parallel deployment of multiphase simulation comes up with its own set of challenges. As the simulation progress in time, the interface blooms, break into droplets, and forms complex regions and shapes. This leads to a significant increase in the mesh resolution as the time and simulation progress. We showcase one such simulation of primary jet atomization later in Sec. IV. With the increase in the mesh resolution, it is desirable to increase the number of processes accordingly.

We dump the files checkpoint files at frequent intervals. Once the checkpoint is reloaded, we can load it either from the same or larger number of processes. Specific care must be taken while loading with increased processor count. When the processor count is increased, MPI:\textunderscore Comm is split into active and inactive communicators. The checkpoint is loaded with active communicators, and the mesh is created only within the active communicator. The active communicator has the size of the processes that dump the checkpoint files. Once a repartition or remeshing is triggered, the new mesh is then redistributed among all the available processes. The inactive communicators become active once the repartition or remeshing function is called.

**III. Results \& Discussion**

**A. Region Identification: MATVEC scaling**

The proposed algorithm for erosion and dilation scheme relies solely on the MATVEC operations, which involve performing the local elemental traversal with associated ghost exchange. Therefore, we show the performance of MATVEC operations of our framework for linear basis functions. Fig. 4 shows the strong and weak scaling of MATVEC for our framework. For strong scaling, we consider an adaptive mesh of around 13M elements and 13.7 M DOFs. Fig. 4a shows the MATVEC execution time as a function of increasing number of cores. MATVEC execution time decreased from 2.87 s on 224 processes to 0.027 s on 28K processes, resulting in 81% parallel efficiency for 128 fold increase in processor count. For the weak scaling runs, we created grids with a fixed grain size of about 35K elements per core and timed MATVEC execution.
time. The coarsest mesh consists of 981K elements on 28 processes with 1.02M DOFs for linear and 8.01M DOFs. Fig. 4b shows the MATVEC averaged over 100 iterations as a function of the number of cores. A constant execution time would imply ideal weak scaling efficiency. We observed a slowly growing weak-scaled execution time. Overall the time increased from about 1.58 s on 28 cores to 1.9 s on 14 K cores for linear elements, resulting in 82% weak scaling efficiency.

B. Application performance scaling

Fig. 5 shows the scaling performance of the full framework on TACC Frontera. The overall mesh consists of around 700 M mesh elements. The simulation was run for 11 timesteps. We have timed each of the solvers and remeshing separately. Overall, we see a good scaling behavior in all the solvers with NS-solve, resulting in 6.6× speedup for 8× increase in the processor from $O(14K)$ processor to $O(112K)$ processor. Similarly, PP–solve showed a 5.3× reduction, whereas VU–solve showed 5× and CH–solve showed a 4× reduction in total solve time. The remeshing cost showed 2.5× reduction for 4× increase in processes till $O(57K)$ processes but then began to grow with further increase in processor. This increased cost in the remeshing needs further investigation.

The variable density PP–solve is the most time-consuming step till the point where remeshing cost begins to dominate. Solving pressure Poisson efficiently, especially with variable coefficients, is still a current area of research. Scalable solvers, like Geometric multigrid (GMG), promise to yield a better solve time but relies on optimized algorithms for creating different mesh hierarchy and MATVEC operation. This is

---

A. Application to Primary Jet Atomization

We finally demonstrate the ability of our algorithm to simulate the most resolved simulation of primary jet atomization. The finest resolution in our application problem consists of octree at level 15. This is equivalent to solving it on a 35 trillion grid point – a 64× larger than the current state–of–the–art. Fig. 6 shows the simulation result at 6.3 $\mu$s. We can see that the simulation framework can capture tiny droplets by selectively identifying the key reasons of interest and selectively increasing the resolution. Fig. 7 shows the progressive refinement of the mesh. We can see that the algorithm proposed in Sec. II-B is able to detect complicated structures like filaments and drops and selectively refine those regions. The overall interface is resolved at level 13 with the key features resolved at level 15. Fig. 8 shows the fraction of the element at different levels. We see that a significant portion of the elements is at levels 13 and 14 ($\approx$ 25%), with a maximum fraction at level 15. A rough

---

4 All simulation was performed using 56 cores per node.
5 We have tested the framework with Algebraic multigrid via PETSc and Hypre interface. Our study showed that iterative solver yields a better solve time at a large processor count due to the setup cost associated with AMG. Therefore, we used iterative solvers for the current study.

---

6 An octree of level $l$ implies $\Delta x = \Delta y = \Delta z = L/2^l$, where $L$ is the length of the domain.
Fig. 7. Adaptive mesh refinement: 2D slice with mesh overlayed. Note the tip of the filament and bubble is much more resolved then the interface. This region is identified by the erosion and dilation algorithm described in Sec. II-B. The octree level differs by 11 levels, with coarsest mesh at level 4 and the finest at level 15, resulting in a $10^3 \times$ difference in the elemental volume between the finest and the coarsest elements.

Fig. 8. Variation of the element fraction with respect to the levels. An estimate indicates that resolving the complete interface at level 15 would result in at least an $8 \sim 10 \times$ increase in the overall mesh count, resulting in approximately $20 \sim 25 \times$ higher time to solve, $^\text{7}$ yielding the simulation impractical in a reasonable time. We note that level 15 has the maximum element fraction but covers only $0.01\%$ of total volume. This illustrates the importance of adaptivity in resolving multiphase simulations. We defer the more detailed physics discussion for a future paper.

V. RELATED WORK

In this work, we have shown the importance of identifying the key regions of interest and selectively refining those specific regions. The alternative approaches for accompanying the task can be through the route of determining the connected components [24]. However, the connected component labeling is computationally more expensive and non-trivial to implement compared to the current approach. Additionally, connected component labeling is not enough to identify all the regions of interest, such as the thin filament, which is a part of the big structure (for example, Fig. 1b). Furthermore, the proposed approach by Harrison et al. [24] is limited to octree-based meshes, whereas the current approach in this work can be easily integrated into other numerical libraries with structured and unstructured meshes, as it solely relies on efficient MATVEC computations – a key component of scientific computations.

In the context of sharp interface capturing methods, there are advances in techniques, for example, in interface reconstruction [25, 26], moment-of-fluid [27], manifold death [28] that also resolve try to resolve small structures. These techniques are, however, limited to sharp interface methods at the moment.

VI. CONCLUSION & FUTURE WORK

In this work, we present algorithmic advances in the field of multiphase modeling. We proposed a novel scalable algorithm to identify the key regions of interest that need sufficiently high resolution for capturing the physics accurately. This approach is quintessential for efficient simulation of multiphase simulation in a reasonable time. Further, we modify the octree refinement algorithm to refine to an arbitrary level. We also accelerate the matrix and vector assembly by carefully analyzing the data movement. We showcase the application of these algorithmic advances by simulating the canonical problem of primary jet atomization simulation and push the current state–of–the–art by a factor of $64 \times$. The approach proposed in this work can serve as a reference for the next–generation multiphase modeling for diffuse as well as sharp interface modeling.

In future, we plan to extend these algorithms, by integrating the framework with GPU accelerated linear algebra solvers like AMAT [29], MAGMA [30]. Further, we plan to improve the DA scaling and utilize GMG to improve the solve–time, specifically for the variable coefficient pressure Poisson problem. We also plan to integrate the setup with Domain

$^7$assuming $O(N\log N)$ scaling of KSP solve.
Specific Language (DSL) like FINCH [31], FENICS [32] and extend FEM framework to Finite Volume and Discontinuous Galerkin schemes.

REFERENCES

[1] D. Fuster, A. Bagué, T. Boeck, L. Le Moyne, A. Leboissetier, S. Popinet, P. Ray, R. Scardovelli, and S. Zaleski, “Simulation of primary atomization with an octree adaptive mesh refinement and vof method,” International Journal of Multiphase Flow, vol. 35, no. 6, pp. 550–565, 2009.

[2] G. Tomar, D. Fuster, S. Zaleski, and S. Popinet, “Multiscale simulations of primary atomization,” Computers & Fluids, vol. 39, no. 10, pp. 1864–1874, 2010.

[3] C. I. Pairetti, S. M. Damián, N. M. Nigro, S. Popinet, and S. Zaleski, “Mesh resolution effects on primary atomization simulations,” Atomization and Sprays, vol. 30, no. 12, 2020.

[4] M. Herrmann, “Detailed numerical simulations of the primary atomization of a turbulent liquid jet in crossflow,” Journal of Engineering for Gas Turbines and Power, vol. 132, no. 6, 2010.

[5] M. Gorokhovski and M. Herrmann, “Modeling primary atomization,” Annu. Rev. Fluid Mech., vol. 40, pp. 343–366, 2008.

[6] J. Lu and G. Tryggvason, “Direct numerical simulations of multifluid flows in a vertical channel undergoing topology changes,” Physical Review Fluids, vol. 3, no. 8, p. 084401, 2018.

[7] R. M. Chiodi, Advancement of numerical methods for simulating primary atomization. Cornell University, 2020.

[8] M. Jemison, M. Sussman, and M. Shashkov, “Filament capturing with the multimaterial moment-of-fluid method,” Journal of Computational Physics, vol. 285, pp. 149–172, 2015.

[9] L. Chirco, J. Maarek, S. Popinet, and S. Zaleski, “Manifold death: a volume of fluid implementation of controlled topological changes in thin sheets by the signature method,” Journal of Computational Physics, vol. 467, p. 111468, 2022.

[10] K. Saurabh, B. Gao, M. Fernando, S. Xu, M. A. Khanwale, B. Khara, M.-C. Hsu, A. Krishnamurthy, H. Sundar, and B. Ganapathysubramanian, “Industrial scale large eddy simulations with adaptive octree meshes using immerseogeometric analysis,” Computers & Mathematics with Applications, vol. 97, pp. 28–44, 2021.

[11] K. Saurabh, M. Ishii, M. Fernando, B. Gao, K. Tan, M.-C. Hsu, A. Krishnamurthy, H. Sundar, and B. Ganapathysubramanian, “Scalable adaptive pde solvers in arbitrary domains,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–15.

[12] M. Fernando, D. Neilsen, H. Lim, E. Hirschmann, and H. Sundar, “Massively parallel simulations of binary black hole intermediate-mass-ratio inspirals,” arXiv preprint arXiv:1807.06128, 2018.

[13] S. Popinet, “ Gerris: a tree-based adaptive solver for the incompressible euler equations in complex geometries,” Journal of computational physics, vol. 190, no. 2, pp. 572–600, 2003.

[14] ——, “An accurate adaptive solver for surface-tension-driven interfacial flows,” Journal of Computational Physics, vol. 228, no. 16, pp. 5838–5866, 2009.

[15] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, “Algorithms and data structures for massively parallel generic adaptive finite element codes,” ACM Transactions on Mathematical Software (TOMS), vol. 38, no. 2, pp. 1–28, 2012.

[16] M. A. Khanwale, K. Saurabh, M. Ishii, H. Sundar, J. A. Rossmanith et al., “A projection-based, semi-implicit time-stepping approach for the cahn-hilliard navier-stokes equations on adaptive octree meshes,” arXiv preprint arXiv:2107.05123, 2021.

[17] M. Ishii, M. Fernando, K. Saurabh, B. Khara, B. Ganapathysubramanian, and H. Sundar, “Solving pdes in spacetime: 4d tree-based adaptivity, mesh-free and matrix-free approaches,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–61.

[18] M. Fernando, D. Duplyakin, and H. Sundar, “Machine and application aware partitioning for adaptive mesh refinement applications,” in Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, 2017, pp. 231–242.

[19] C. Burstedde, L. C. Wilcox, and O. Ghattas, “p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees,” SIAM Journal on Scientific Computing, vol. 33, no. 3, pp. 1103–1133, 2011.

[20] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas, “Recursive algorithms for distributed forests of octrees,” SIAM Journal on Scientific Computing, vol. 37, no. 5, pp. C497–C531, 2015.

[21] H. Sundar, G. Biros, C. Burstedde, J. Rudi, O. Ghattas, and G. Stadler, “Parallel geometric-algebraic multigrid on unstructured forests of octrees,” in SC’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE, 2012, pp. 1–11.

[22] H. Sundar, D. Malhotra, and G. Biros, “HykSort: A new variant of hypercube quicksort on distributed memory architectures,” in Proceedings of the 27th international ACM conference on international conference on supercomputing, 2013, pp. 293–302.

[23] T. Hoefler, C. Siebert, and A. Lumsdaine, “Scalable communication protocols for dynamic sparse data exchange,” ACM Sigplan Notices, vol. 45, no. 5, pp. 159–168, 2010.

[24] C. Harrison, J. Weiler, R. Bleile, K. Gaither, and A. Lumsdaine, “Scalable communication protocols for dynamic sparse data exchange,” in ACM conference on international conference on supercomputing, 2013, pp. 293–302.
[25] J. Lu and G. Tryggvason, “Direct numerical simulations of multifluid flows in a vertical channel undergoing topology changes,” Physical Review Fluids, vol. 3, no. 8, p. 084401, 2018.

[26] R. M. Chiodi, Advancement of numerical methods for simulating primary atomization. Cornell University, 2020.

[27] M. Jemison, M. Sussman, and M. Shashkov, “Filament capturing with the multimaterial moment-of-fluid method,” Journal of Computational Physics, vol. 285, pp. 149–172, 2015.

[28] L. Chirco, J. Maarek, S. Popinet, and S. Zaleski, “Manifold death: a volume of fluid implementation of controlled topological changes in thin sheets by the signature method,” Journal of Computational Physics, vol. 467, p. 111468, 2022.

[29] H. D. Tran, M. Fernando, K. Saurab, B. Ganapatysubramanian, R. M. Kirby, and H. Sundar, “A scalable adaptive-matrix spmv for heterogeneous architectures,” in 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2022, pp. 13–24.

[30] S. Tomov, J. Dongarra, V. Volkov, and J. Demmel, “Magma library,” Univ. of Tennessee and Univ. of California, Knoxville, TN, and Berkeley, CA, 2009.

[31] E. Heisler, A. Deshmukh, and H. Sundar, “Domain specific language and code generation for finite element and finite volume in julia,” in International Conference on Computational Science. Springer, 2022, pp. 118–132.

[32] M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells, “The fenics project version 1.5,” Archive of Numerical Software, vol. 3, no. 100, 2015.