2018

A scalable software framework for solving PDEs on distributed octree meshes using finite element methods

Alec Lofquist
Iowa State University

A scalable software framework for solving PDEs on distributed octree meshes using finite element methods

by

Alec Dale Lofquist

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Iowa State University

Ames, Iowa

2018

DEDICATION

I would like to dedicate this thesis to my parents, Lori and Steven, whose constant love and support opened the path to where I am today.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr. Baskar Ganapathysubramanian for taking a chance on me as an undergrad and giving me the opportunity to work on problems at such an awesome scale. I hope your enthusiasm continues to inspire others as it did me.

I would also like to thank everybody in Baskar Group for all the time you spent teaching me math and science. It was thanks to your patient help that I was able to make a contribution (however small!) to a field I had no business working in.

Finally, this work would not have been possible without the support from Dr. Songzhe Xu and Dr. Hari Sundar. Thank you both for all the late nights spent debugging.

ABSTRACT

Tracking particle motion in inertial flows (especially in obstructed geometries) is a computationally daunting proposition. This is further complicated by that fact that the construction of migration maps for particles (as a function of particle location, flow conditions, and particle size) requires several thousands of simulations tracking individual particles. This calls for the development of an efficient, scalable approach for single particle tracking in fluids. We bring together three distinct elements to accomplish this: (a) a parallel octree based adaptive mesh generation framework, (b) a variational multiscale (VMS) based treatment that enables flow condition agnostic simulations (laminar or turbulent) Bazilevs et al. (2007a), and (c) a variationally consistent immersed boundary method (IBM) to efficiently track moving particles in a background octree mesh Xu et al. (2016). This project builds on our existing codes for adaptive meshing (Dendro) and finite elements (TalyFEM). We present our adaptive meshing framework that is tailored for the immersed boundary method and experiments demonstrating the scalability of our code to over 16,000 processes.

CHAPTER 1. OVERVIEW

1.1 Introduction

This work combines our existing in-house FEM library (TalyFEM) with an existing distributed octree meshing library (Dendro) to create a third framework we call Dendrite. In chapter 2, we examine particle tracking in a channel, a target problem to solve as a first milestone for this framework. In chapter 3, we conclude with some additional in-progress projects that build on this work.

1.2 TalyFEM

TalyFEM is our in-house FEM library which we have used for several past works (Dyja et al. (2015), Xu et al. (2016)). This library is written in C++ and is built on top of MPI and the PETSc framework Balay et al. (2001). TalyFEM provides an API for generating, loading, and solving PDEs on unstructured meshes using the finite element method. A constant pain point has been adaptive meshing: problems that involve moving objects require us to regularly regenerate the entire finite element mesh and interpolate nodal data onto the new mesh. For more complicated meshes, we use the third party mesh generator Gmsh Geuzaine and Remacle (2009), which is implemented using sequential algorithms. This mesh generator can take a long time to run and does not scale with more compute resources. In this work, we integrate TalyFEM with Dendro, a distributed octree library, to take advantage of an octree-based adaptive meshing algorithm.

While a number of impressive FEM libraries have been developed since the inception of TalyFEM (over ten years ago), we have spent a lot of resources building, testing, documenting, and training scientists on this library. We would prefer not to abandon it for an entirely new platform as some projects are still being actively developed on this codebase. One of the design goals for Dendrite (this work) is to allow us to reuse FEM kernel code from TalyFEM with minimal changes.

Enabling this code reuse requires us to keep the API that the FEM kernel code uses the same. This API encompasses basis function evaluation and access to interpolated nodal values at the Gauss points. As we will briefly discuss in chapter 2, this is not quite straightforward as our implementation is tightly coupled with our underlying mesh data structure.

1.3 Dendro

Dendro is an existing framework for solving PDEs using finite element methods on distributed octree meshes. Octree meshes strike a nice balance between the simplicity of structured meshes and the flexibility of unstructured meshes, allowing Dendro to implement mesh adaptivity in a computationally efficient, scalable way. Dendro has already been shown to scale to over 4,000 processes and supports multigrid methods Sampath et al. (2008), which we may explore using in the future.

1.3.1 Tradeoffs of Octree Meshes

One advantage of the finite element method is that it works with a variety of element shapes and sizes. The most obvious drawback of using an octree mesh is that we are limited to regular hexahedral elements in sizes of $L/2^i$, where L is a domain size (scaling factor) and i is an integer that ranges from 0 to a max octree depth. This also means that all elements must be aligned to $L/2^i$. This makes octree meshes impractical for curved body-fitted meshes without a complicated warping scheme. However, by using an immersed boundary method (IBM), we are able to relax the requirement that our domain boundaries perfectly align with $L/2^i$ in exchange for a surface integration step; during assembly we integrate the effect of the boundary into the "background elements" intersecting it. Chapter 2 will examine the specifics of the immersed boundary method and Dendro implementation details.

Another challenge with octree meshes is that they must be square. This complicates solving on a non-square domain, such as a channel. While we can introduce a per-axis scaling factor to create a rectangular domain, this creates heavily skewed elements. To address this problem, we also extend Dendro to support meshes with holes in them, allowing us to ”carve out” a rectangular channel from the square octree. This ”carving out” still has limitations: we can still only ”cut” at $L/2^i$ boundaries, and the mesh must be smoothly refined to depth i at the boundary. In practice, this limitation is acceptable for our target problem. We do not require extremely high resolution for our channel aspect ratio, and we often want to refine near the domain boundaries anyway.

1.4 Target Problem: Particle Tracking in a Channel

Dendrite was originally conceived to solve a particular problem of interest: tracking lateral motion of a rigid particle as it travels in a microchannel decorated with obstacles. A solution to this problem that can be computed quickly has applications in the design of biomedical devices. We have previously approached this problem using a variational multiscale (VMS) based treatment combined with a variationally consistent immersed boundary method (IBM) in Xu et al. (2016) using TalyFEM. While promising, our implementation was too slow to run on a full 3D problem due to a naive approach to adaptive meshing. Dendrite allows us to reuse much of the code from this previous work with a new octree-based adaptive meshing system. Chapter 2 will examine this problem in detail.

1.5 Summary

Dendrite combines TalyFEM, our in-house FEM library, and Dendro, an octree meshing library. The first milestone for this project is the particle tracking in a channel problem. Scalability is necessary for us to be able to solve our target problem, which requires us to run thousands of simulations to generate our dataset of interest. In chapter 2, we will demonstrate strong scaling results up to 16k processes for our target problem.

CHAPTER 2. CASE STUDY: MOVING PARTICLE IN A CHANNEL

This chapter is based on a conference paper originally submitted to Supercomputing 2018, written as a collaboration with Dr. Songzhe Xu, myself, Dr. Hari Sundar, and Dr. Baskar Ganapathy Subramanian.

2.1 Introduction

Control and localization of particles (cells, precipitates) in aqueous flow is useful in biological processing, chemical reaction control, and for creating structured materials. The controlled motion and localization of cells and particles can automate cellular sample preparation and bio-sensing. Some examples include fast identification of *e. coli* in water, robust removal of circulating tumor cells from the blood plasma and fast separation of cells types for rapid flow cytometry and subsequent identification/tagging for genomic analysis. The precise, efficient and cheap localization of a heterogeneous collection of cells in a fluid medium is a foundational challenge in science and engineering. A general (computationally informed) strategy for passive control of particle localization in microfluidic channels will be transformative to this field.

Researchers have recently discovered Amini et al. (2013) and demonstrated Stoecklein et al. (2014, 2016) the ability to passively engineer the cross-sectional shape of a fluid *(without particles in it)* using the notion of inertial flow deformations induced by **sequences of pillars** that disrupt the flow. This is a *purely passive approach* for flow control that relies on flow physics around bluff bodies. Since these transformations provide a deterministic mapping of fluid elements from upstream to downstream of a pillar, one can sequentially arrange pillars to apply the associated nested maps and therefore program complex fluid structures. This idea has been rapidly picked up by the microfluidics and manufacturing community to make structured particles and fibers Nunes et al. (2014); Paulsen et al. (2015); Wu et al. (2015); Paulsen and Chung (2016) and for reagent recovery in medical diagnostic devices Amini et al. (2014).

Studies have shown that this passive flow control paradigm can be extended to passively localize ***particles*** in fluid flow using a sequence of obstacles that differentially act on various sized particles (based on size and location). While the localization (or 'focusing') of particles in unobstructed microfluid channels is well known, the behavior (and control) of localization of particles in **obstructed microfluidic channels** is a very novel problem with a rich physical underpinning. Segre and Silberberg Segre and Silberberg (1962); Segré and Silberberg (1962) first experimentally observed the phenomena of focusing of particles in a straight channel (or tube) flow. Particles moving in such flows undergo a lateral motion across the flow streamlines until they reach a stable equilibrium located between the channel centerline and the confining walls. Subsequent theoretical studies provided a general understanding of the lift forces and how the structure of lift forces structure depends on the particle size, channel dimensions and flow rate (or Reynolds number). However, the precise calculation needed to design devices that can exploit the migration of particles in flow, such as the separation, concentration, and sorting of cells and biomolecules with high specificity requires highly accurate force calculations, which essentially becomes a computational exercise.

Target problem: Our particular problem of interest is to track the lateral migration of a single, rigid particle as it traverses a microchannel that is decorated with a pillar obstacle (see Fig. 2.1). Our intent is to understand how initial release location, as well as particle size and pillar geometry affect migration patterns.

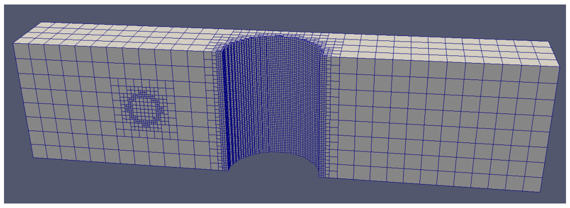

Figure 2.1: An illustrative example of a rigid particle traversing a microchannel decorated with obstacles. Figure shows a slice cut through the geometry.

Tracking particle motion in inertial flows (especially in obstructed geometries) is a computationally daunting proposition. This is further complicated by that fact that the construction of migration maps for particles (as a function of particle location, flow conditions, and particle size) requires **several thousands of simulations** tracking individual particles. This calls for the development of an efficient, scalable approach for single particle tracking in fluids. We bring together three distinct elements to accomplish this: (a) a parallel octree based adaptive mesh generation framework, (b) a variational multiscale (VMS) based treatment that enables flow condition agnostic simulations (laminar or turbulent) Bazilevs et al. (2007a), and (c) a variationally consistent immersed boundary method (IBM) to efficiently track moving particles in a background octree mesh Xu et al. (2016). This project builds on our existing codes for adaptive meshing (Dendro) and Finite Elements (TalyFEM). In the next section, we give a brief introduction to the formulation of our target problem followed by a brief introduction of the immersed boundary method. We then present our adaptive meshing framework that is tailored for the immersed boundary method. We wrap up with experiments demonstrating the scalability of our code.

2.2 Target Problem

We are interested in tracking the motion and lateral forces acting on a single particle—of size a—released at discrete points of the inlet (points shown in green in Fig 2.2). As an individual particle flows down the channel (of width W), it is affected by the spatially varying flow field caused by obstruction due to the pillar (of diameter D). Note that for typical particle sizes ($a/W \geq 0.1$), the moving particle itself causes changes to the flow field (so called *blockage effect*). We are particularly interested in reporting the net lateral displacement as a function of initial release location at the inlet. We consider a finite distance downstream (typically $6D$ downstream of the pillar, due to manufacturability constraints) across which we track the particle motion. The particle displacement is reported as a vector field (Fig 2.2). Additionally, the time history of the lateral forces acting on the particle will be reported.

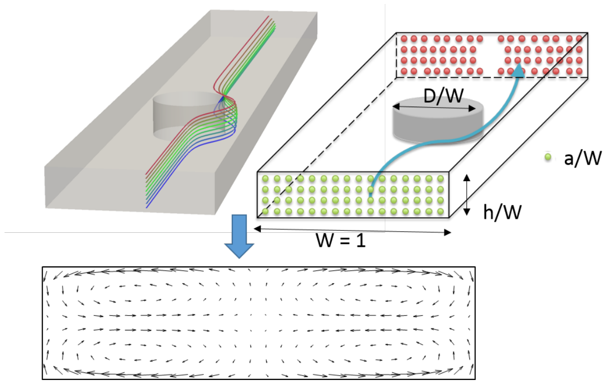

Figure 2.2: An illustration of the canonical target problem. Following standard practice in fluid dynamics, we normalize length scales by the channel width, W, and consider all physical variables in dimensionless quantities. This allows broad usability of the resulting computations, due to kinematic and dynamic similarity principles. The canonical problem is parametrized by 5 variables: (a) the size of the particle (a), (b) the location, δ and diameter, D of the pillar, (c) the flow speed, characterized in terms of the Reynolds number ($\Re$), and (d) the height of the microchannel, h.

For each set of parameters $(a, W, D, \Re, h)$, we hope to simulate the time evolution of the particle-fluid interaction in the domain. Our preliminary results show that it is necessary to have a refined mesh close to the pillar surface, the particle surface as well as the channel walls to fully resolve the fluid velocity features. Based on the Dendro framework, we anticipate requiring `256x256x1024` $\sim 70\times10^6$ hexahedral elements to discretize the domain. Each time step requires solving the Navier-Stokes equations with no-slip boundary conditions on the particle and the channel walls. Once the velocity field (due to the interaction between the particle and pillar and fluid) is computed, the inertial forces on the particle are computed by performing a surface integration of the fluid stress on the particle surface. This is then used to update the location of the particle for the next time step.

The dimensionless Navier–Stokes equations for incompressible flow is written as

$$\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} - \mathbf{f}\right) - \nabla\cdot\boldsymbol{\sigma} = \mathbf{0}\ , \tag{2.1}$$

$$\nabla\cdot\mathbf{u} = 0\ , \tag{2.2}$$

where $\mathbf{u}$ and $\mathbf{f}$ are the flow velocity and the external force, respectively. The stress and strain-rate tensors are defined respectively as

$$\boldsymbol{\sigma}(\mathbf{u}, p) = -p\,\mathbf{I} + 2\frac{1}{\Re}\boldsymbol{\varepsilon}(\mathbf{u}) \ , \tag{2.3}$$

$$\boldsymbol{\varepsilon}(\mathbf{u}) = \frac{1}{2}\left(\boldsymbol{\nabla}\mathbf{u} + \boldsymbol{\nabla}\mathbf{u}^{\mathrm{T}}\right) \ , \tag{2.4}$$

where p is the pressure, $\mathbf{I}$ is an identity tensor. The problem (2.1)–(2.4) is accompanied by suitable boundary conditions, defined on the boundary of the fluid domain, $\Gamma = \Gamma^{\mathrm{D}} \cup \Gamma^{\mathrm{N}}$:

$$\mathbf{u} = \mathbf{u_g} \qquad \text{on } \ \Gamma^{\mathrm{D}} \ , \tag{2.5}$$

$$-p\,\mathbf{n} + 2\frac{1}{\Re}\boldsymbol{\varepsilon}(\mathbf{u})\,\mathbf{n} = \mathbf{h} \qquad \text{on } \ \Gamma^{\mathrm{N}} \ , \tag{2.6}$$

where $\mathbf{u_g}$ denotes the prescribed velocity at the Dirichlet boundary Γ^{D}, $\mathbf{h}$ is the traction vector at the Neumann boundary Γ^{N}, and $\mathbf{n}$ is the unit normal vector pointing in the wall-outward direction.

Consider a collection of disjoint elements $\{\Omega^e\}$, $\cup_e \Omega^e \subset \mathbb{R}^d$. The fluid domain is covered by the closure of the collection: $\Omega \subset \cup_e \overline{\Omega^e}$. Note that Ω^e is not necessarily a subset of Ω with the immersed boundary method. Let $\mathcal{V}_u^h$ and $\mathcal{V}_p^h$ be the finite-dimensional spaces of discrete test functions and trial solutions for velocity and pressure, which are denoted as superscript h, and represent resolved scales (coarse scale) produced by the finite element discretization. The strong problem (2.1)–(2.6) may be recast in a weak form and posed over these discrete spaces to produce the following semi-discrete problem (using the VMS modeling approach): Find $\mathbf{u}^h \in \mathcal{V}_u^h$ and $p^h \in \mathcal{V}_p^h$ such that for all $\mathbf{w}^h \in \mathcal{V}_u^h$ and $q^h \in \mathcal{V}_p^h$:

$$B^{\mathrm{VMS}}\left(\{\mathbf{w}^h, q^h\}, \{\mathbf{u}^h, p^h\}\right) - F^{\mathrm{VMS}}\left(\{\mathbf{w}^h, q^h\}\right) = 0 \ . \tag{2.7}$$

The bilinear form B^{VMS} and the load vector F^{VMS} are given as

$$\begin{aligned}
&B^{\text{VMS}}\left(\{\mathbf{w}^h, q^h\}, \{\mathbf{u}^h, p^h\}\right) \\
&= \int_\Omega \mathbf{w}^h \cdot \left(\frac{\partial \mathbf{u}^h}{\partial t} + \mathbf{u}^h \cdot \boldsymbol{\nabla} \mathbf{u}^h\right) \, d\Omega \\
&+ \int_\Omega \boldsymbol{\nabla} \mathbf{w}^h : \boldsymbol{\sigma}\left(\mathbf{u}^h, p^h\right) \, d\Omega \\
&+ \int_\Omega q^h \boldsymbol{\nabla} \cdot \mathbf{u}^h \, d\Omega \\
&- \sum_e \int_{\Omega^e \cap \Omega} \left(\mathbf{u}^h \cdot \boldsymbol{\nabla} \mathbf{w}^h + \boldsymbol{\nabla} q^h\right) \cdot \mathbf{u}' \, d\Omega \\
&- \sum_e \int_{\Omega^e \cap \Omega} p' \boldsymbol{\nabla} \cdot \mathbf{w}^h \, d\Omega \\
&+ \sum_e \int_{\Omega^e \cap \Omega} \mathbf{w}^h \cdot (\mathbf{u}' \cdot \boldsymbol{\nabla} \mathbf{u}^h) \, d\Omega \\
&- \sum_e \int_{\Omega^e \cap \Omega} \boldsymbol{\nabla} \mathbf{w}^h : \left(\mathbf{u}' \otimes \mathbf{u}'\right) \, d\Omega,
\end{aligned} \tag{2.8}$$

and

$$F^{\text{VMS}}\left(\{\mathbf{w}^h, q^h\}\right) = \int_\Omega \mathbf{w}^h \cdot \mathbf{f} \, d\Omega + \int_{\Gamma^N} \mathbf{w}^h \cdot \mathbf{h} \, d\Gamma \ , \tag{2.9}$$

where primes denote the unsolved scales (fine scale) that need to be modeled, and their effect needs to be added onto the coarse scale. $\mathbf{u}'$ is defined as

$$\mathbf{u}' = -\tau_{\text{M}} \left(\frac{\partial \mathbf{u}^h}{\partial t} + \mathbf{u}^h \cdot \boldsymbol{\nabla} \mathbf{u}^h - \mathbf{f} - \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}\left(\mathbf{u}^h, p^h\right)\right) \ , \tag{2.10}$$

and p' is given by

$$p' = -\tau_{\text{C}} \boldsymbol{\nabla} \cdot \mathbf{u}^h \ . \tag{2.11}$$

$\mathbf{u}'$ and p' are approximated by the residuals of momentum equation and continuity equation, respectively, and τ_M and τ_C are corresponding coefficients with the definitions in Bazilevs et al. (2007a). Equations (2.8)–(2.11) feature the VMS formulation of Navier-Stokes equations of incompressible flows Bazilevs et al. (2007a). The additional terms added onto the standard weak Galerkin form can be interpreted as a combination of streamline/upwind Petrov Galerkin (SUPG) stabilization and VMS large-eddy simulation of turbulence modeling.

The particle evolution is written as

$$m\frac{d\boldsymbol{V}}{dt} = \boldsymbol{F}$$
$$J\frac{d\boldsymbol{\omega}}{dt} = \boldsymbol{T} \tag{2.12}$$

where $\boldsymbol{V} = [u_p, v_p, w_p]^T$ and $\boldsymbol{\omega} = [\omega_x, \omega_y, \omega_z]^T$ are the particle linear and angular velocities, and $\boldsymbol{F} = [F_x, F_y, F_z]^T$ and $\boldsymbol{T} = [\tau_x, \tau_y, \tau_z]^T$ are the force and torque acting on the particle. The force is computed as the surface integral of the fluid stress over the particle surface and an explicit time-stepper is used to update the particle location and velocity.

2.3 Immersed Boundary Method

The immersed boundary method (IBM) was first introduced by Peskin in the context of fluid-structure interaction (FSI) for a heart simulation with associated blood flow to avoid remeshing when the solid body deformed Peskin (1972, 1973). The IBM embeds the solid geometry into a background Cartesian mesh without conforming them to each other, and the effect of the immersed boundary on the fluid field has to be formulated by imposing the boundary conditions of the immersed geometry and distributed on the background Cartesian mesh. Since the IBM does not require a conforming mesh, it becomes computationally convenient to track the motion of particles of arbitrary shape while avoiding a cumbersome boundary fitted (re)meshing process.

The implementation of the IBM requires some refinement of the background mesh across the immersed surface to better capture the shape of the interface as well as to resolve the no-slip boundary condition. This is accomplished by using selective quadrature (i.e. only using those Gauss points that lie in the fluid and not inside the immersed particle). This necessitates performing an "in/out test" to determine the Gauss points inside the fluid domain (red dots) on which we assemble, while discarding the Gauss points inside the object (green dots), as shown in Fig. 2.3.

The no-slip boundary condition (which is a Dirichlet boundary condition) is converted into an equivalent Neumann condition (in the sense of the Nitsche method Nitsche (1971)). Thus, we perform a surface integral over the immersed boundary to weekly impose the Dirichlet boundary

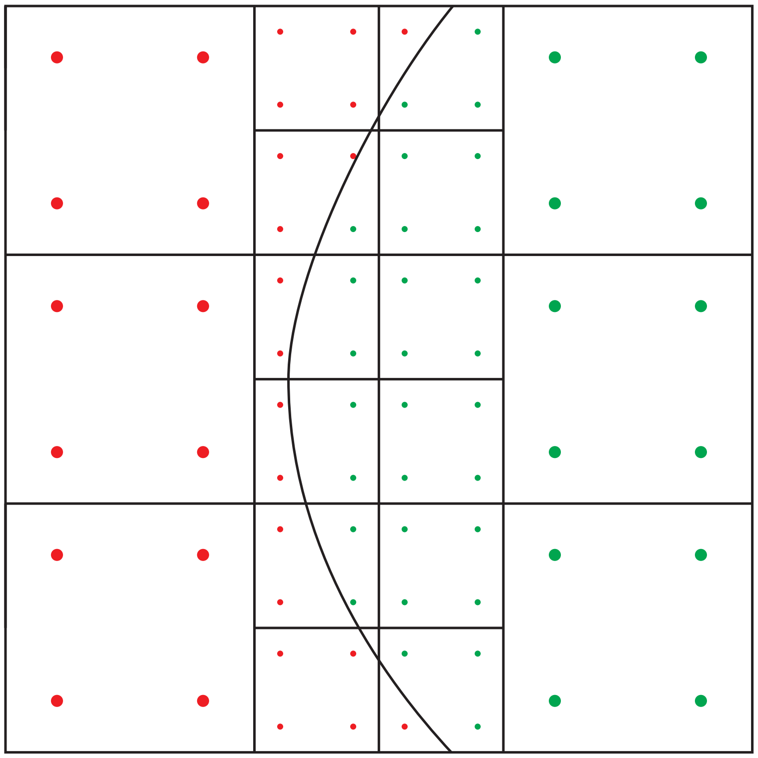

Figure 2.3: A schematic of the volume assembly in the IBM method. We loop over each element and each Gauss point within each element. An in-out test is performed to identify weather that Gauss point is lies inside the particle (red points) or inside the fluid (green points). Only the Gauss points in the fluid domain are used to assemble the elemental matrices.

condition of the immersed boundary Bazilevs and Hughes (2007); Bazilevs et al. (2007b, 2010). Assuming the immersed boundary Γ_{I} is decomposed into N_{eb} surface elements each denoted by $\Gamma_{\mathrm{I}}^{\mathrm{b}}$, the semi-discrete problem becomes

$$\begin{aligned}
&B^{\mathrm{VMS}}\Big(\{\mathbf{w}^h, q^h\}, \{\mathbf{u}^h, p^h\}\Big) - F^{\mathrm{VMS}}\Big(\{\mathbf{w}^h, q^h\}\Big) \\
&\quad - \sum_{b=1}^{N_{eb}} \int_{\Gamma_{\mathrm{I}}^{\mathrm{b}} \bigcap \Gamma_{\mathrm{D}}} \mathbf{w}^h \cdot \left(-p^h\,\mathbf{n} + 2\frac{1}{\Re}\boldsymbol{\varepsilon}(\mathbf{u}^h)\,\mathbf{n} \right)\, d\Gamma \\
&\quad - \sum_{b=1}^{N_{eb}} \int_{\Gamma_{\mathrm{I}}^{\mathrm{b}} \bigcap \Gamma^{D}} \left(2\frac{1}{\Re}\boldsymbol{\varepsilon}(\mathbf{w}^h)\,\mathbf{n} + q^h\,\mathbf{n} \right) \cdot \left(\mathbf{u}^h - \mathbf{u}_{\mathbf{g}} \right)\, d\Gamma \\
&\quad + \sum_{b=1}^{N_{eb}} \int_{\Gamma_{\mathrm{I}}^{\mathrm{b}} \bigcap \Gamma^{D}} \tau^B \mathbf{w}^h \cdot \left(\mathbf{u}^h - \mathbf{u}_{\mathbf{g}} \right)\, d\Gamma \;=\; 0\,. \qquad (2.13)
\end{aligned}$$

The boundary terms added to the governing equation are the second, third and last line in Eq. 2.13, and a detailed interpretation of different terms can be found in Bazilevs and Hughes (2007). Only the penalty-like stabilization parameter, τ^B, is a heuristic that has to be appropriately chosen. We use the definition proposed in Wu et al. (2017), which scales the stabilization parameter as $\tau^{\mathrm{B}} = \mathrm{C}h/\Delta t$, where C is a positive constant, h is the size of the cut element, and Δt is the time step. The boundary terms are imposed onto the surface Gauss points, which are then interpolated by their background Cartesian grids as shown in Fig. 2.4. In this way we can apply the Dirichlet boundary condition on the immersed boundary of the object to the fluid field.

The immersed boundary method has previously been deployed on unstructured meshes to track particle motion in micro-channels and has shown better computational efficiency than a body-fitted method Xu et al. (2016). We integrate the moving IBM with an octree mesh to achieve even better performance.

2.4 Scalable IBM on Octree Meshes

While the concept of adaptive space partitions is well studied, developing such methods for the immersed boundary method on large distributed systems presents significant challenges. This work builds on our existing methods for performing large-scale finite element computations using

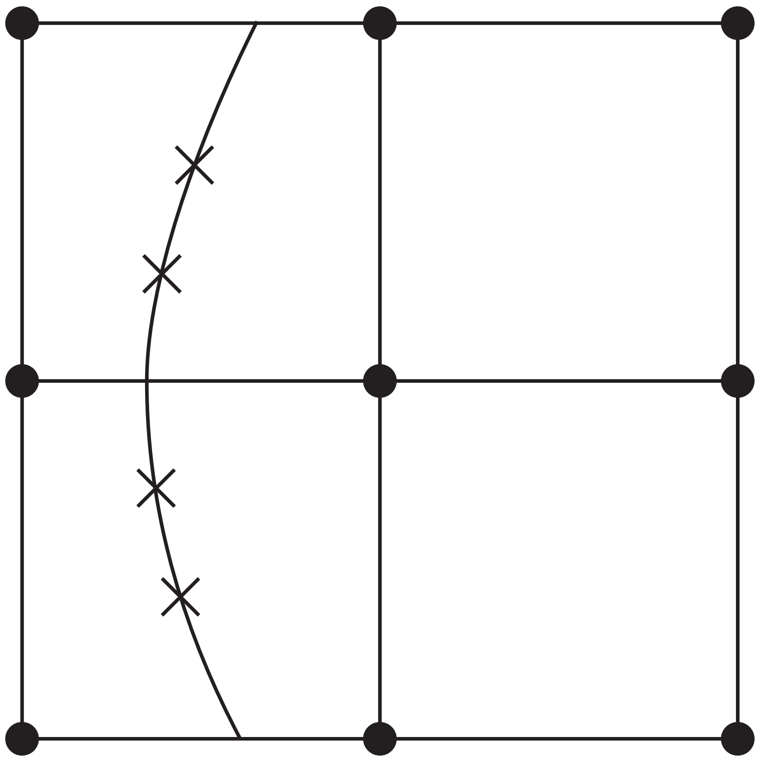

Figure 2.4: Schematic showing how the surface assembly of IBM is performed. The triangulated surface mesh is used to identify surface Gauss points (the 'X' locations). The surface integral terms (i.e. the last three terms in Eq. 2.13) are computed at these surface Gauss points, and then distributed to the nodal locations.

octree-refined meshes. We have extended this work to support the particle localization simulations. We provide a brief description on building the octree mesh in parallel and performing FEM computations and refer to Sundar et al. (2008) for additional details.

While the elemental matrix assemblies are done using TalyFEM described in the next section, Dendro provides the adaptive mesh refinement (AMR) and all parallel data-structures. For this project, Dendro was extended to support meshes with *holes* in it. This is because of the presence of pillars in the channels where we do not need to solve. An example of such a mesh is shown in Fig. 2.1.

The main steps in building and maintaining an adaptive mesh in Dendro are:

Refinement: The sparse grid is constructed based on the geometry. Proceeding in a top-down fashion, a cell is refined if a surface (pillar/particle) passes though it. During the same step, we also determine if the cell is completely inside the pillar, and eliminate it from the mesh in that case. Since the refinement happens in a element-local fashion, this step is embarrassingly parallel. The user passes a function that given coordinates, x, y, z returns the distance from the pillar(s). The eight corners of an octant are tested using this function. If all 8 points have a positive distance (outside), then we retain this element, but do not refine further. If all 8 points have a negative distance (inside), then this element is removed from the mesh. If some of the corners of the octant are inside and others outside, then this octant is refined. This is repeated till we reach the desired level of refinement is achieved. In distributed memory, all processes start from the root and refine until at least p octants requiring further refinement are produced. Then using a weighted space-filling-curve based partitioning, we partition the octants. Note that we do not communicate the octants as every process has a copy of the octants, and all that needs to be done at each process is to retain a subset of the current octants and recurse. A 2:1 balancing is enforced following the refinement operation.

2:1 Balancing: We enforce a condition in our distributed octrees that no two neighboring octants differ in size by more than a factor of two. This makes subsequent operations simpler without affecting the adaptive properties. Our balancing algorithm is similar to existing approaches for balancing octrees Bern et al. (1999); Sundar et al. (2007); Burstedde et al. (2011) with the added aspect that it does not generate octants if the ancestor does not exist in the input. This is done to ensure that the *holes* are not filled in. The algorithm proposed by Bern Bern et al. (1999) is easily extensible to support this case, as we simply need to skip adding balancing octants that violate the criteria.

Partition: Refinement and the subsequent 2:1-balancing of the octree can result in a non-uniform distribution of elements across the processes, leading to load imbalance. This is particularly common in the presence of holes. We use a Hilbert space-filling curve to equipartition the elements by performing a parallel scan on the number of elements on each process followed by point-to-point communication to redistribute the elements. The presence of holes in the domain does not adversely affect this as the partition only tries to equally divide the elements across the processes.

Elements that intersect the immersed boundary require extra computation to assemble (§2.5.1). This can cause a load-imbalance during assembly, as the immersed boundary is typically localized on a small subset of processes and these processes will be the only ones performing surface assembly. This can be accommodated by estimating the relative cost of volume *vs.* surface assembly and performing a weighted partition of the elements. This is not currently done for the results presented here, but we hope to have this completed soon.

Meshing: By meshing we refer to the construction of the (numerical) data structures required for FEM computations from the (topological) octree data. Dendro already has efficient implementations for building the required neighborhood information and for managing overlapping domains between processors (*ghost* or *halo* regions). The key difference with previous applications is the requirement to handle meshes with holes, as all neighbors might not be present in the mesh. This also complicates the process of applying boundary conditions. We added support for defining *subdomains* within Dendro. The subdomains are defined using a function that takes a coordinate (x, y, z) as input and returns `true` or `false` depending on whether that coordinate is part of the subdomain or not. The subdomain leverages the core mesh data-structure and additionally defines a unique mapping for nodes that are part of the subdomain. It also keeps track of which nodes belong to subdomain boundaries. Therefore, subdomains have a small overhead and store significantly less data that the main mesh data-structure. For our target application, it is important to identify both the external (domain) boundary as well as the internal boundary (the pillar surface). The subdomain stores two `bits` to keep track of whether a node is non-boundary, external, or internal boundary.

2.5 Integration with TalyFEM

We previously developed code for calculating the elemental matrix and vector for solving Navier-Stokes with IBM in our in-house FEM framework, TalyFEM, which is designed for arbitrary unstructured meshes. We chose to integrate the core of our in-house framework with Dendro to avoid re-implementing the NS+IBM kernel.

Ideally, we would be able to write an adapter that would provide TalyFEM's API but pass through to Dendro's mesh data structure. However, Dendro does not support random access to octant data - since the node coordinates and elemental connectivity are implicit in the octree's structure, Dendro calculates these values on the fly as the octree is traversed, instead of storing the data persistently as is typically done for an unstructured mesh. Since TalyFEM was designed for unstructured meshes, support for random element access was assumed in its API. A naive solution would be to traverse the octree once and build a random access compatible data structure, but this would impose a significant memory overhead and need to be rebuilt after every remesh.

Instead, we create an unstructured TalyFEM mesh containing a single hexahedral element. As we iterate through the octree mesh for assembly we re-position the nodes in the unstructured element to match the octree element. We also copy nodal data (velocity and pressure) from Dendro's buffers to support the assembly code. This allows us to reuse our existing assembly implementation from TalyFEM with virtually no changes with little overhead.

Basis Functions Since unstructured meshes typically have a great number of element shapes, TalyFEM repeatedly recalculates the isoparametric to physical mapping at each Gauss point. As these values change depending on the shape of the element, it is not feasible to cache them on a large unstructured mesh. However, the octree mesh has only one possible element shape; we take advantage of this by pre-calculating these values during initialization. We create a fake element at the origin for each level in the octree and cache the evaluated basis functions. When the assembly code needs to access these values, we pull them from the corresponding level in the cache and offset the position appropriately.

2.5.1 Sampling the immersed boundary & adding corrections

The immersed boundary (in our problem, the surface of the particle) is defined by a triangulated mesh. Surface integration points are then calculated for each triangle using standard Gaussian quadrature. We also calculate other necessary parameters such as the unit normal and the boundary value of velocity at each Gauss point.

The surface Gauss points are then sorted and distributed to the process that owns the background element containing them. We do this by first mapping each surface Gauss point to a ”virtual” element in the most refined octree mesh possible. This ”most refined mesh” is effectively just a Cartesian grid, which makes the mapping trivial. These ”virtual” elements, which may or may not exist in the real mesh, are still ordered by the same space-filling curve and thus use the same partitioning. We sort and distribute each point to the process that owns the real ancestor of our ”virtual” octant using point-to-point communication.

Now that the surface Gauss points are sorted and on the right processes, we need to visit them with their background element in context to calculate the IBM corrections for the elemental matrix and vector. Since the surface Gauss points are already sorted in the same order as the mesh elements, we can loop over both elements and surface Gauss points simultaneously in linear time; at each element, we only check if the earliest unvisited surface point is contained by the current element.

2.5.1.1 The In/Out Test

Nodes that only belong to elements which are fully inside the geometry are marked and set to have a Dirichlet boundary condition of zero. Since calculating this list of nodes requires information about neighboring elements and the mesh is partitioned, some communication is necessary. We use Dendro’s distributed vector data structure to associate an 8-bit integer with each node. This value represents the number of elements that expect to contribute to this node.

To fill this vector, each process loops over its elements. If an element is not fully inside the immersed geometry, it adds 1 to this value for all nodes in the element. After each process finishes its loop, we synchronize the distributed vector, summing the values in overlapping regions on each processor. This gives each process a consistent nodal vector, where a value of 0 means the node will not be solved for by any element on any process. We use this data to apply our Dirichlet boundary conditions.

2.5.2 Timestepping and particle evolution

The time-dependent Navier–Stokes equation is solved with an implicit scheme. We show results for a backward Euler time-stepping scheme. Once the fluid field is solved, a surface integral over the immersed boundary is then performed to calculate the surface force that is exerted on the object by the fluid.

$$\begin{aligned}\mathbf{F} = &\sum_{b=1}^{N_{eb}} \int_{\Gamma_{\mathrm{I}}^{\mathrm{b}} \bigcap \Gamma^D} \boldsymbol{\sigma}(\mathbf{u}^h, p^h) \cdot \mathbf{n} d\Gamma \\ &- \sum_{b=1}^{N_{eb}} \int_{\Gamma_{\mathrm{I}}^{\mathrm{b}} \bigcap \Gamma^D} \tau^B (\mathbf{u}^h - \mathbf{u_g}) d\Gamma,\end{aligned} \tag{2.14}$$

$$\begin{aligned}\mathbf{T} = &\sum_{b=1}^{N_{eb}} \int_{\Gamma_{\mathrm{I}}^{\mathrm{b}} \bigcap \Gamma^D} \mathbf{r} \times \left(\boldsymbol{\sigma}(\mathbf{u}^h, p^h) \cdot \mathbf{n} \right) d\Gamma \\ &- \sum_{b=1}^{N_{eb}} \int_{\Gamma_{\mathrm{I}}^{\mathrm{b}} \bigcap \Gamma^D} \mathbf{r} \times \tau^B (\mathbf{u}^h - \mathbf{u_g}) d\Gamma.\end{aligned} \tag{2.15}$$

The last terms in Eq. 2.14 and Eq. 2.15 are the penalty-like term that are added onto the surface force calculation. The total force acting on the object is the summation of the surface force and any external body forces (gravity & buoyancy). The particle evolution is computed using an explicit Euler solve.

The backward Euler time-stepper for the NS equation is given as

$$\frac{\partial \boldsymbol{u}}{\partial t} = \frac{\boldsymbol{u}^n - \boldsymbol{u}^{n-1}}{\Delta t} = \mathcal{L}(\boldsymbol{u}^n, p^n), \tag{2.16}$$

where the operator $\mathcal{L}(\boldsymbol{u}^n, p^n)$ represents all the other terms except the time-dependent term evaluated at the current time step in the Navier–Stokes Eq (2.1). Δt is selected to follow the CFL condition. The (non)linear solution procedure is taken care by PETSC Arge et al. (1997). We utilize PETSc's Newton-based line search non-linear solver (newtonls) with the BiCGSTAB linear solver (bcgs). An additive Schwarz preconditioner (asm) is also used for parallel preconditioning and solving on decomposed sub-domains.

The particle is modeled as a rigid body. We denote the velocity of the objects as $\mathbf{v}$, with the motion described as

$$\frac{d\mathbf{x}^{\mathrm{c}}}{dt} = \mathbf{v}^{\mathrm{c}}, \qquad \frac{d\mathbf{v}^{\mathrm{c}}}{dt} = \frac{\mathbf{F}}{m}, \tag{2.17}$$
$$\frac{d\boldsymbol{\theta}^{\mathrm{c}}}{dt} = \boldsymbol{\omega}^{\mathrm{c}}, \qquad \frac{d\boldsymbol{\omega}^{\mathrm{c}}}{dt} = \frac{\mathbf{T}}{J}, \tag{2.18}$$

where $\mathbf{x}^{\mathrm{c}}$ and $\boldsymbol{\theta}^{\mathrm{c}}$ are the linear and angular locations of the patricle, $\mathbf{v}^{\mathrm{c}}$ and $\boldsymbol{\omega}^{\mathrm{c}}$ are linear and angular velocities, $\mathbf{F}$ and $\mathbf{T}$ are the integral of force and torque acting on the particle surface, and m and J are the particle mass and moment of inertia, respectively. $\mathbf{F}$ and $\mathbf{T}$ are computed from the solution of the fluid field, and defined as follows

$$\mathbf{F} = \oint_{\Gamma} \boldsymbol{\sigma}(\mathbf{u}, p) \cdot \mathbf{n} d\Gamma, \qquad \mathbf{T} = \oint_{\Gamma} \mathbf{r} \times (\boldsymbol{\sigma}(\mathbf{u}, p) \cdot \mathbf{n})\, d\Gamma, \tag{2.19}$$

Where Γ is the boundary of particle, $\boldsymbol{\sigma}(\mathbf{u}, p)$ is the stress tensor acting on the particle, $\mathbf{r}$ is the distance vector from the particle centroid to any point on its surface, and the coordinates $\mathbf{x}$ and velocities $\mathbf{v}$ at any point on the particle surface is computed as

$$\mathbf{x} = \mathbf{x}^{\mathrm{c}} + \mathbf{r}, \qquad \mathbf{v} = \mathbf{v}^{\mathrm{c}} + \boldsymbol{\omega}^{\mathrm{c}} \times \mathbf{r}. \tag{2.20}$$

Finally, $\mathbf{n}$ is the unit normal vector that points outward from the particle surface. In the discrete form, assuming the integral of the force and torque over the particle surface are constant during one time step, we have

$$\frac{(\mathbf{x}^{\mathrm{c}})^{n+1} - (\mathbf{x}^{\mathrm{c}})^{n}}{\Delta t} = \frac{(\mathbf{v}^{\mathrm{c}})^{n+1} + (\mathbf{v}^{\mathrm{c}})^{n}}{2}, \tag{2.21}$$
$$\frac{(\mathbf{v}^{\mathrm{c}})^{n+1} - (\mathbf{v}^{\mathrm{c}})^{n}}{\Delta t} = \frac{(\mathbf{F})^{n}}{m_i}, \tag{2.22}$$
$$\frac{(\boldsymbol{\theta}^{\mathrm{c}})^{n+1} - (\boldsymbol{\theta}^{\mathrm{c}})^{n}}{\Delta t} = \frac{(\boldsymbol{\omega}^{\mathrm{c}})^{n+1} + (\boldsymbol{\omega}^{\mathrm{c}})^{n}}{2}, \tag{2.23}$$
$$\frac{(\boldsymbol{\omega}^{\mathrm{c}})^{n+1} - (\boldsymbol{\omega}^{\mathrm{c}})^{n}}{\Delta t} = \frac{(\mathbf{T})^{n}}{m_i}. \tag{2.24}$$

$(\mathbf{F})^n$ and $(\mathbf{T})^n$ are discretized in space and computed with weakly imposed boundary condition as follows

$$(\mathbf{F})^n = \sum_{b=1}^{N_{eb}} \int_{\Gamma^{\mathrm{b}} \bigcap \Gamma} \boldsymbol{\sigma}(\mathbf{u}^n, p^n) \cdot \mathbf{n} d\Gamma \tag{2.25}$$

$$- \sum_{b=1}^{N_{eb}} \int_{\Gamma^{\mathrm{b}} \bigcap \Gamma} \tau^B(\mathbf{u}^{\mathbf{n}} - \mathbf{v}^n) d\Gamma, \tag{2.26}$$

$$(\mathbf{T})^n = \sum_{b=1}^{N_{eb}} \int_{\Gamma^{\mathrm{b}} \bigcap \Gamma} \mathbf{r} \times (\boldsymbol{\sigma}(\mathbf{u}^n, p^n) \cdot \mathbf{n})\, d\Gamma \tag{2.27}$$

$$- \sum_{b=1}^{N_{eb}} \int_{\Gamma^{\mathrm{b}} \bigcap \Gamma} \mathbf{r} \times \tau^B(\mathbf{u}^{\mathbf{n}} - \mathbf{v}^n) d\Gamma \tag{2.28}$$

The particle velocity is evaluated by an explicit forward Euler scheme, which requires a small Δt to ensure accuracy and stability. Each object location is updated by the average velocities.

2.5.3 Intergrid transfers

An essential requirement is to adapt the spatial mesh as the particle moves across the domain. In the distributed memory setting, this also indicates a need to repartition and load-balance. Every few timesteps, we remesh. This is similar to the initial mesh generation and refinement, except that it is now based on the current position of the particle. This is followed by the 2:1 balance enforcement and meshing. Once the new mesh is generated, we transfer the velocity and pressure fields from the old mesh to the new mesh using interpolation as needed. Since the integrid transfer happens only between parent and child (for coarsening and refinement) or are unchanged, this can be performed on the old mesh using standard linear interpolation, followed by a simple repartitioning based on the new mesh. An example of the adaptive mesh refinement following the moving particle is shown in Figure 2.5.

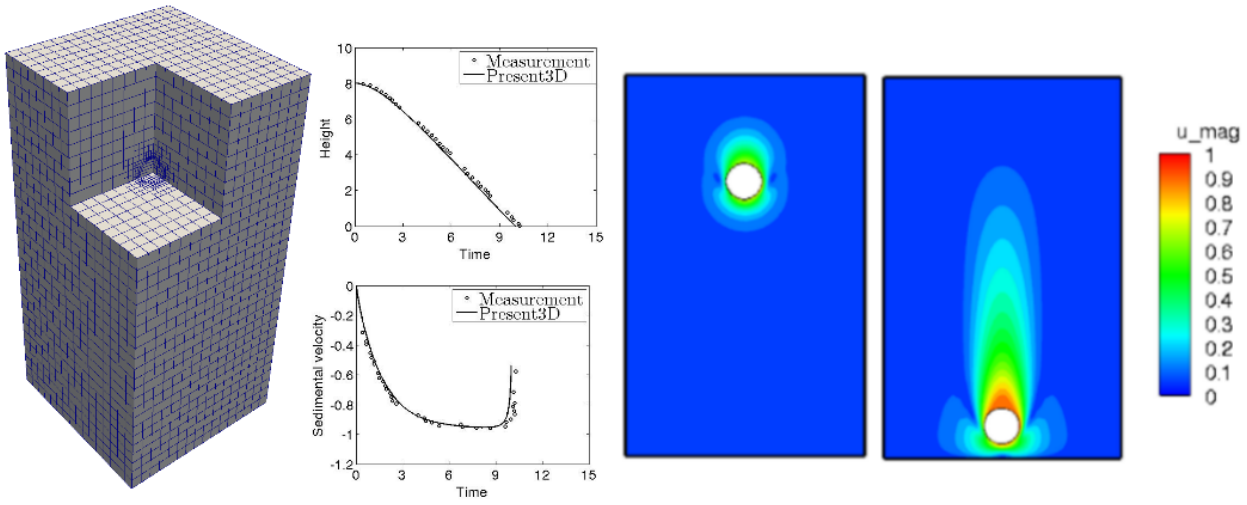

Figure 2.5: Validation of the framework against an experimental benchmark of a particle setting due to gravity Ten Cate et al. (2002). (left) A representative mesh illustrating the refinement around the particle, (middle) A comparison of the height evolution and velocity evolution of the particle as it settles downwards. Notice the particle reaching a terminal velocity. As the particle nears the bottom surface its velocity rapidly zeros out. (Right) Contours of total velocity at two representative time instances.

2.6 Experiments & Results

2.6.1 Implementation details

The Dendro framework implemented in `C++` using `MPI` for distributed memory parallelism and `OpenMP` for shared memory parallelism. The TalyFEM framework is also implemented in `C++` with `MPI` and is used for evaluating basis functions and interpolating nodal data to support assembly as described in 2.5. Our code is tightly integrated with `PETSc v3.7` Arge et al. (1997)'s distributed matrix and vector data-structures and utilizes its SNES and KSP solvers.

These tests were compiled and run on Oak Ridge's Titan supercomputer. PETSc, Dendro, TalyFEM, and the main program were compiled with the GNU 4.9.3 compiler with `-O2` optimization flags. Timing information was reported using PETSc's logging framework.

2.6.2 Validation

Before discussing the scaling behavior, we first validate the framework by comparing the particle trajectory and velocity against a benchmark experimental data of a sphere dropped in a quiescent fluid Ten Cate et al. (2002). We consider a container of dimensions $(0.1\,m \times 0.16\,m \times 0.1\,m)$. We simulate a sphere released at $(0.05\,m, 0.12\,m, 0.05\,m)$ with a diameter of $D = 0.015\,m$. The fluid has a density of $\rho_f = 960\,kg/m^3$, and a dynamic viscosity of $\mu = 0.058\,kg/(m \cdot s)$. The density of the sphere is $\rho_s = 1120\,kg/m^3$. Reynolds number, defined as $\frac{\rho_f \mathbf{u}_0 D}{\mu}$, is $Re = 31.9$ with a reference velocity $\mathbf{u}_0 = 0.128\,m/s$. Initial conditions are set as zero velocity in the whole fluid domain. No slip boundary condition is imposed on lateral and bottom walls, and no velocity gradient and zero pressure boundary conditions are imposed on the top wall. The validation results are presented in Figure 2.5.

2.6.3 Meshes/domains

We next focus on showing scaling of the framework. We collect timing for the case of a dropping sphere (of size 1) in a domain of size $8 \times 8 \times 8$. We run each case for 3 time steps. We adaptively refine the mesh around the interface of the sphere three levels deeper than the rest of the background mesh, remeshing after each timestep as the sphere moves. Note that such frequent remeshing is one of the challenges of our target application. The mesh is defined by a pair of minimum refinement l and maximum refinement h, where the background mesh element size ranges from $8/2^l$ to $8/2^h$ at the interface. We adjust the characteristic length of the surface mesh in sync with the refinement of the interface mesh, keeping a ratio of 1:2 for the surface triangle size to the interface element size. We run this experiment on five background/interface refinement levels: 4/7, 5/8, 6/9, 7/10, and 8/11. Each refinement level has roughly seven to eight times more degrees of freedom to solve for than the previous level, with 4/7 having 29,000 degrees of freedom and 8/11 reaching 70.2 million.

We note that given specific l and h and the same initial conditions, the overall problem size in spite of mesh-refinement is consistent independent of the number of processes being used for the simulation. To this effect, we believe presenting performance for different l/h combinations for different number of processes in the style of a strong scaling is appropriate. Indeed, performing weak scaling for such real-world applications is harder, and therefore, given the somewhat fixed increase in problem size with increasing l/h, we derive the weak-scalability from a set of strong scaling experiment (Figure 2.7).

2.6.4 Parallel Scalability

For our target application, the key goal is to be able to perform the simulations quickly, given the sheer number of simulations we need to perform. Given this, and the relatively moderate size of our problems, the focus is on strong scalability. We first present strong scalability results for the overall simulation, including the cost of remeshing in Fig. 2.6 for two problem sizes. Overall our code scales well, with continued reductions in solve time. We can combine multiple strong scaling results to get approximate weak scaling results for the overall solve times. Note that in general this is much harder than the strong scaling results, since it is much harder to ensure that N/p, i.e., the grain size stay relatively constant. The approximated weak scaling results are presented in Fig. 2.7.

We report additional results to get a deeper understanding of the performance and scalability of the different parts of our code. We present strong scalability results for Matrix assembly Fig. 2.8

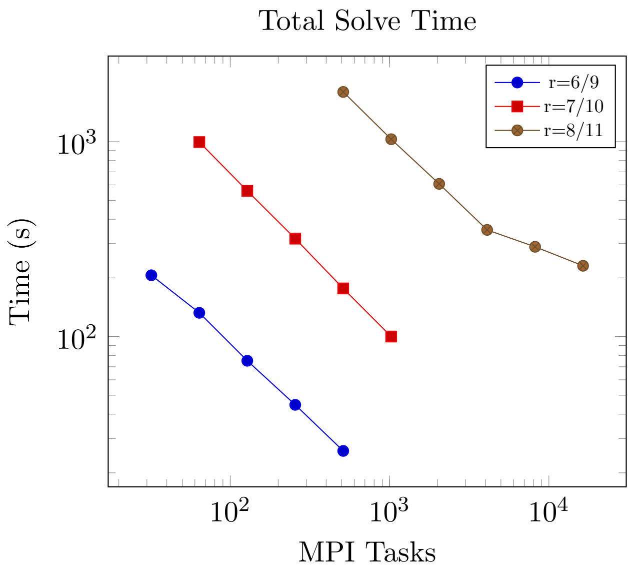

Figure 2.6: Total time to solve three full timesteps (including remeshing) for different problem sizes for a single sphere of size 1 dropping in a channel of size $8 \times 8 \times 8$.

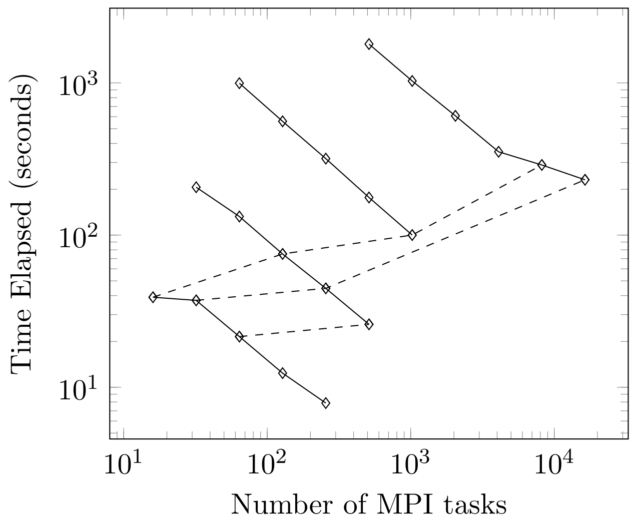

Figure 2.7: Weak scalability approximated from multiple strong-scaling experiments. Since it is difficult to perform weak scalability experiments, with such frequent mesh-refinements and consequently changes in problem size, we instead approximate the weak (dashed lines) scaling from the strong (solid lines) scaling results for $r = (5/8, 6/9, 7/10, 8/11)$ and p up to $16,384$ on Titan. Note that minor fluctuations in the approximation of the weak scalability are expected due to the inconsistent grain size.

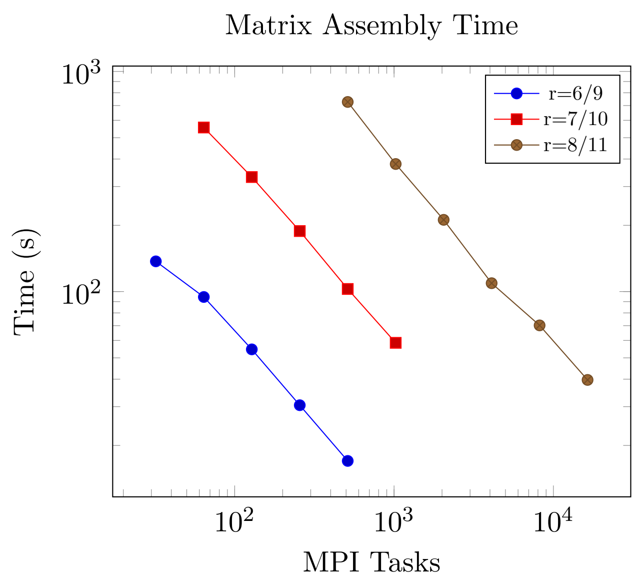

Figure 2.8: Matrix assembly time (volume + surface + Dirichlet BC + communication) for various mesh refinements.

and Vector assembly Fig. 2.9. Both methods scale reasonably well, but the overall time for matrix assemblty is more expensive compared to the vector assembly. This is largely due to the complexity of the operator. We also report just the time spent in the remeshing stage. The remeshing stage refers to the combination of generating a new mesh, interpolating between two meshes and reinitializing the matrix, vector and solver. Effectively, this is the overhead paid for having good adaptivity. The scaling of remeshing, shown in Fig. 2.11, is not as good as the other parts of the code, but the magnitude of time it takes is much smaller than solving the NS equations. Again, note that this is strong scaling, and the meshing code is sufficiently optimized, making it much harder to demonstrate strong scalability across the full range.

2.6.5 Overhead of immersed boundary corrections

After each remesh we perform the in/out test described in 2.5.1.1 in order to identify which nodes in the mesh belong to the fluid. We must also redistribute the surface Gauss points to the appropriate processes as the mesh has been re-partitioned, as described in 2.5.1. In our experiments, we see this bookkeeping time taking up to 10% of our total solve time and scaling well with the

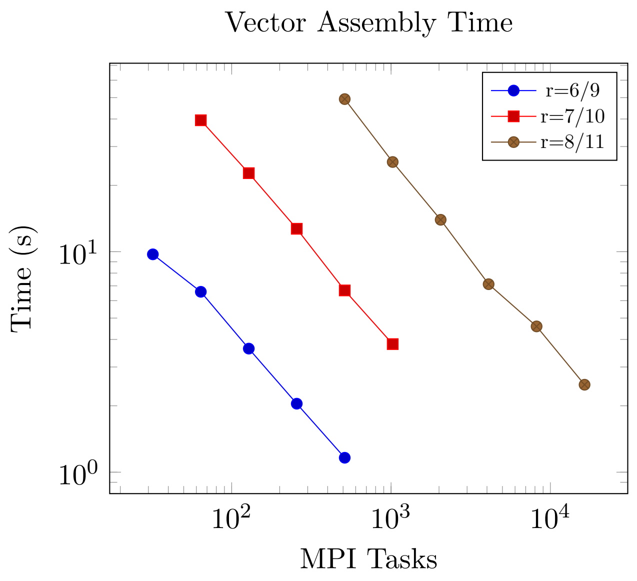

Figure 2.9: Vector assembly time (volume + surface + Dirichlet BC + communication) for various mesh refinements.

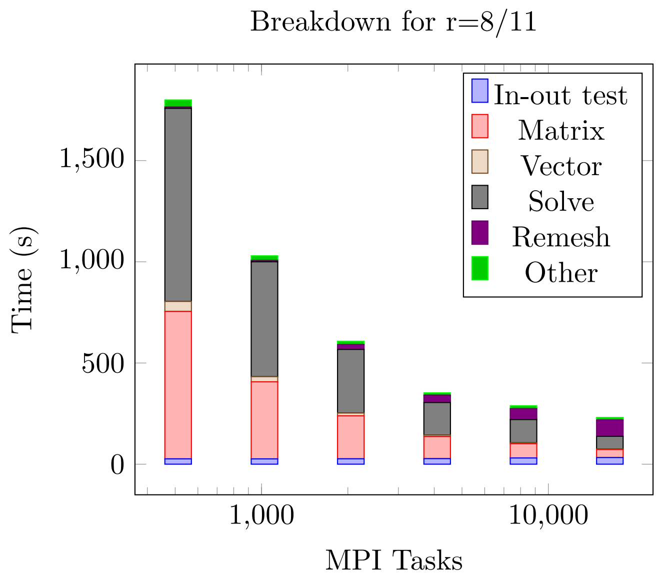

Figure 2.10: Total solve time broken down by category for refinement level 8/11. In-out is described in section 2.6.5. Matrix/vector refer to the time it takes to build the Jacobian matrix/residual vector (volume assembly + surface assembly + BC + communication). Solve refers to the time taken to actually solve the system (i.e. BiCGStab + ASM preconditioner). Remesh refers to time taken to create the next timestep's mesh and interpolate data onto it.

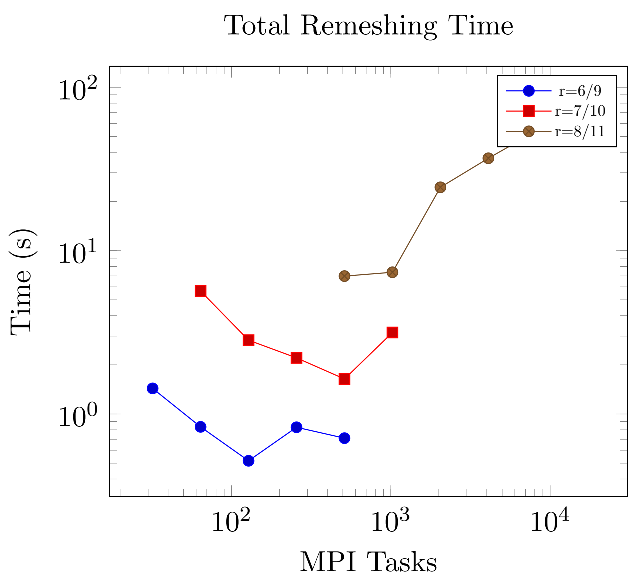

Figure 2.11: Total time for adaptive remeshing for various mesh refinements.

number of processes, as shown in Fig. 2.10 ("In-out"). This figure also highlights that that the overall runtime is dominated by matrix assembly followed by the solver. It can also be seen that the overhead of the AMR is negligible (listed under other). Applying the actual IBM corrections to the matrix and vector takes a more significant amount of time - sometimes more than volume assembly, as shown in Fig. 2.12, 2.13 - but appears to scale better than volume assembly. This is likely because we weight non-interface mesh elements the same as elements containing surface Gauss points when partitioning the mesh. This leads to a work imbalance where processes all perform roughly equal parts of volume assembly, but only some participate in surface assembly. We plan to introduce an elemental "work factor" to the partitioning algorithm to address this imbalance in the future. This issue is also affects the surface force integral over the immersed boundary which is used to update the particle velocity. Surface assembly also does not use the cached basis function values (as volume assembly does), as each surface Gauss point may have a unique position relative to its background element.

IBM corrections also currently happen in a separate step, isolated from the normal Navier-Stokes assembly to keep the code modular. We could reduce communication by combining the

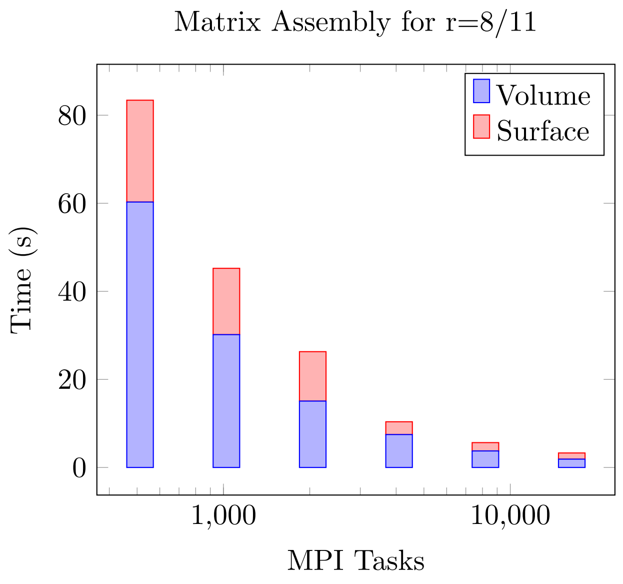

Figure 2.12: Total time spent in matrix assembly broken down by volume vs surface for refinement level 8/11.

IBM corrections with normal assembly, which would avoid duplicate communications for shared nodes. This might improve our run time, but would not affect our scaling behavior.

2.7 Summary

We showcased the performance of a scalable, IBM framework based on octree meshes for tracking particle localization in complex geometry microfluid channels. This framework allows us to efficiently construct the deformation maps for particles under a broad range of experimentally accessible parameters (as illustrated in Fig. 2.2), which will result in a passive approach for particle localization. Our approach demonstrate excellent strong scalability for the overall solve time, even with frequent remeshing. Our framework keeps the overhead of AMR and immersed boundary corrections relatively low, making the overall approach scaleable - one of our design goals. Our immediate goals are to improve the performance for the matrix assembly and incorporate a dynamic load balancing that accounts for the additional work involved in the surface computations. We are

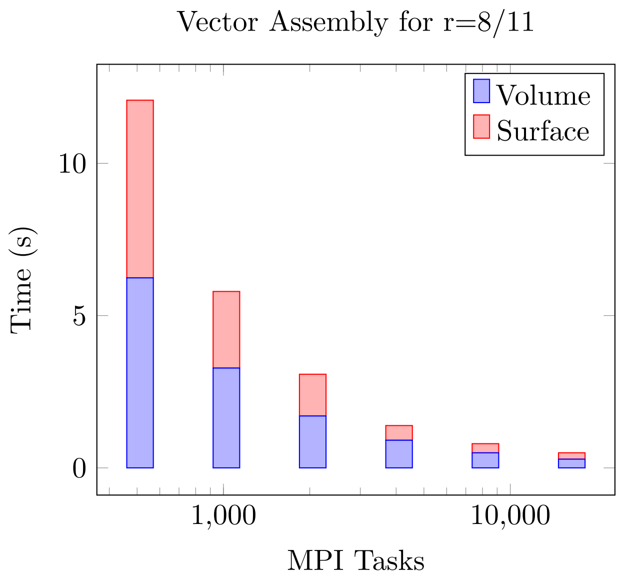

Figure 2.13: Total time spent in vector assembly broken down by volume vs surface for refinement level 8/11.

also working on a couple of extensions to the Dendrite framework which we will discuss in the next chapter.

CHAPTER 3. FUTURE DIRECTIONS

This chapter describes a couple of in-progress projects using Dendrite.

3.1 Support for Arbitrary Immersed Geometries

The primary challenge with arbitrary immersed meshes is refining the octree along the immersed boundary. In this work, we enhanced Dendro to support meshing near the interface of a level-set function. For a spherical particle, the level-set function is trivial; for an arbitrary mesh, perhaps designed in a CAD program, deriving such a function may not be straightforward. To support such meshes, Dendrite also contains a level-set function "generator" that takes a triangulated mesh and provides a function to test if a point is inside the mesh using ray-tracing. If the point is outside the mesh, it returns 1; if it is on the surface of the mesh, it returns 0; if it is inside the mesh, it returns -1.

There is a well-known algorithm for checking if a polygon contains a point: we cast a ray in a random direction from the point of interest and count how many times it intersects the triangles of the mesh Shimrat (1962). If the number of intersections is even, the point is outside the geometry. We chose to use the algorithm given by Tomas Moller Möller and Trumbore (2005) to detect ray-triangle intersection for its simplicity. Since our surface meshes can be very fine, we also pre-calculate a rectangular bounding box for the entire mesh to quickly answer queries for points far outside the immersed geometry. We also pre-calculate bounding spheres for each triangle to skip the ray-triangle test if the ray is not near the triangle. Figure 3.1 shows an example of an airplane mesh with the octree refined around its surface.

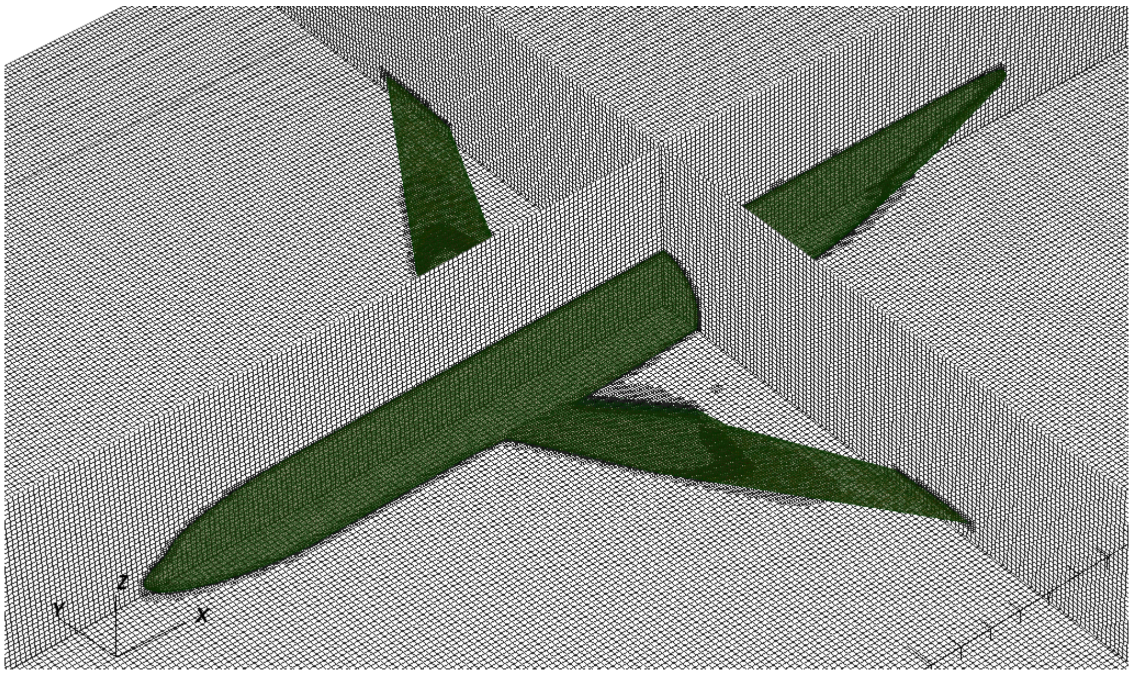

Figure 3.1: The octree mesh refined around the surface of a simplified airplane model.

3.2 Non-rigid Particles: Refining by a Field

Another problem originally started in TalyFEM involves studying the evolution of interfaces in two-phase flows using a thermodynamically consistent coupled Cahn-Hilliard Navier-Stokes based formulation Khanwale et al. (2018). This work uses different equations than the particle-in-a-channel problem outlined in chapter 2 and does not use the immersed boundary method.

For this problem, we need to track (potentially multiple) non-rigid particles as they evolve over time. These particles may deform or even split into multiple bodies. We use a smooth scalar field on the finite element mesh which represents the mixture of the two phases. A value of -1 means the fluid is entirely phase A, a value of 1 means entirely phase B, and 0 represents the interface between the two phases. We solve for this phase field and the fluid velocity and pressure field using a block iterative method.

The phase field we are solving is nearly the level set function we want to refine with, but we are not able to use the level-set-based mesh generator directly due to the lack of random access to mesh data. Instead, we once again extend Dendro with a mesh generator that takes an existing "base" mesh and a scalar elemental vector represents what level each element should be refined to. Dendro iterates over the base mesh and checks the value in the elemental vector for each octant. If an element has a higher value than its current refinement, it is split into more octants. If an element and all its neighbors have a lower value, the octant is coarsened.

This is very similar to the level-set mesh generator, but allows the application to specify the desired refinement once, globally, instead of on-the-fly for each octant. This is a trade-off: since we cannot recurse into octants with this method (since the mesh generator is only given refinement values for the base mesh), we may end up with significant over-refinement if the base mesh is coarse and the desired refinement is deep and localized. A two-pass approach can be used to correct this over-refinement if necessary, although this essentially requires us to pay the cost of remeshing twice. We currently only perform this second pass when creating the initial mesh, as the change in refinement levels is relatively small after the first solve.

To put into perspective how important adaptive meshing is, in TalyFEM, we needed a uniform mesh with nearly 1.5 billion elements to simulate a full 3D problem. With Dendrite's adaptive remeshing, we can simulate the same problem with less than a million elements - only 0.07% of the TalyFEM uniform mesh!

Since the equations for this project are carefully tuned and went through many iterations, the portability of the FEM kernel code between TalyFEM and Dendrite has shined for this project. Our version of Dendro only supports 3D meshes, which are extremely resource-intensive to simulate for this particular problem. Our interoperability with TalyFEM has allowed us to quickly prototype changes to the underlying equations in TalyFEM, validate them in 2D, then copy the changes into the Dendrite framework with almost no changes.

3.3 Summary and Conclusion

We successfully used Dendrite to solve our target problem. We have shown that our solution scales to 16k processes, following through on our scalability goal. We have been able to share code with multiple existing TalyFEM projects, and the interoperability was immediately helpful for us, following through on our flexibility goal. We have preliminary results for extending Dendrite to work with arbitrary meshes using IBM and raytracing-based meshing, as well as refining our meshes by variables we solve for. We hope to use this framework to publish more work in the future.

REFERENCES

Amini, H., Lee, W., and Di Carlo, D. (2014). Inertial microfluidic physics. *Lab on a Chip*, 14(15):2739–2761.

Amini, H., Sollier, E., Masaeli, M., Xie, Y., Ganapathysubramanian, B., Stone, H. A., and Di Carlo, D. (2013). Engineering fluid flow using sequenced microstructures. *Nature communications*, 4:1826.

Arge, E., Bruaset, A. M., and Langtangen, H. P., editors (1997). *Efficient Management of Parallelism in Object Oriented Numerical Software Libraries*. Birkhäuser Press.

Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. (2001). Petsc web page.

Bazilevs, Y., Calo, V. M., Cottrel, J. A., Hughes, T. J. R., Reali, A., and Scovazzi, G. (2007a). Variational multiscale residual-based turbulence modeling for large eddy simulation of incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 197:173–201.

Bazilevs, Y. and Hughes, T. J. R. (2007). Weak imposition of Dirichlet boundary conditions in fluid mechanics. *Computers & Fluids*, 36:12–26.

Bazilevs, Y., Michler, C., Calo, V. M., and Hughes, T. J. R. (2007b). Weak Dirichlet boundary conditions for wall-bounded turbulent flows. *Computer Methods in Applied Mechanics and Engineering*, 196:4853–4862.

Bazilevs, Y., Michler, C., Calo, V. M., and Hughes, T. J. R. (2010). Isogeometric variational multiscale modeling of wall-bounded turbulent flows with weakly enforced boundary conditions on unstretched meshes. *Computer Methods in Applied Mechanics and Engineering*, 199:780–790.

Bern, M., Eppstein, D., and Teng, S.-H. (1999). Parallel construction of quadtrees and quality triangulations. *International Journal of Computational Geometry & Applications*, 9(06):517–532.

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

Dyja, R., Gawronska, E., Grosser, A., Jeruszka, P., and Sczygiol, N. (2015). Estimate the impact of different heat capacity approximation methods on the numerical results during computer simulation of solidification. In *The World Congress on Engineering and Computer Science*, pages 1–14. Springer.

Geuzaine, C. and Remacle, J.-F. (2009). Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International journal for numerical methods in engineering*, 79(11):1309–1331.

Khanwale, M., Lofquist, A., Hur, S. C., Sundar, H., and Ganapathysubramanian, B. (2018). Simulating two-phase flows using a thermodynamically consistent coupled cahn-hilliard navier-stokes framework. *Bulletin of the American Physical Society.*

Möller, T. and Trumbore, B. (2005). Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, page 7. ACM.

Nitsche, J. (1971). Über ein Variationsprinzip zur Lösung von Dirichlet-Problemen bei Verwendung von Teilräumen, die keinen Randbedingungen unterworfen sind. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 36:9–15.

Nunes, J. K., Wu, C.-Y., Amini, H., Owsley, K., Di Carlo, D., and Stone, H. A. (2014). Fabricating shaped microfibers with inertial microfluidics. *Advanced Materials*, 26(22):3712–3717.

Paulsen, K. S. and Chung, A. J. (2016). Non-spherical particle generation from 4d optofluidic fabrication. *Lab on a Chip*, 16(16):2987–2995.

Paulsen, K. S., Di Carlo, D., and Chung, A. J. (2015). Optofluidic fabrication for 3d-shaped particles. *Nature communications*, 6.

Peskin, C. S. (1972). Flow patterns around heart valves: a numerical method. *Journal of computational physics*, 10(2):252–271.

Peskin, C. S. (1973). Flow patterns around heart valves: a digital computer method for solving the equations of motion. *IEEE Transactions on Biomedical Engineering*, BME-20(4):316–317.

Sampath, R. S., Adavani, S. S., Sundar, H., Lashuk, I., and Biros, G. (2008). 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.

Segre, G. and Silberberg, A. (1962). Behaviour of macroscopic rigid spheres in poiseuille flow part 1. determination of local concentration by statistical analysis of particle passages through crossed light beams. *Journal of fluid mechanics*, 14(1):115–135.

Segré, G. and Silberberg, A. (1962). Behaviour of macroscopic rigid spheres in poiseuille flow part 2. experimental results and interpretation. *Journal of fluid mechanics*, 14(1):136–157.

Shimrat, M. (1962). Algorithm 112: Position of point relative to polygon. *Commun. ACM*, 5(8):434–
.

Stoecklein, D., Wu, C.-Y., Kim, D., Carlo, D. D., and Ganapathysubramanian, B. (2016). Optimization of micropillar sequences for fluid flow sculpting. *Physics of Fluids*, 28(1):012003.

Stoecklein, D., Wu, C.-Y., Owsley, K., Xie, Y., Di Carlo, D., and Ganapathysubramanian, B. (2014). Micropillar sequence designs for fundamental inertial flow transformations. *Lab on a Chip*, 14(21):4197–4204.

Sundar, H., Sampath, R., and Biros, G. (2008). Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708.

Sundar, H., Sampath, R. S., Adavani, S. S., Davatzikos, C., and Biros, G. (2007). Low-constant parallel algorithms for finite element simulations using linear octrees. In *SC'07: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE.

Ten Cate, A., Nieuwstad, C., Derksen, J., and Van den Akker, H. (2002). Particle imaging velocimetry experiments and lattice-boltzmann simulations on a single sphere settling under gravity. *Physics of Fluids*, 14(11):4012–4025.

Wu, C.-Y., Owsley, K., and Di Carlo, D. (2015). Rapid software-based design and optical transient liquid molding of microparticles. *Advanced Materials*, 27(48):7970–7978.

Wu, M. C. H., Kamensky, D., Wang, C., Herrema, A. J., Xu, F., Pigazzini, M. S., Verma, A., Marsden, A. L., Bazilevs, Y., and Hsu, M.-C. (2017). Optimizing fluid–structure interaction systems with immersogeometric analysis and surrogate modeling: Application to a hydraulic arresting gear. *Computer Methods in Applied Mechanics and Engineering*, 316:668–693.

Xu, F., Schillinger, D., Kamensky, D., Varduhn, V., Wang, C., and Hsu, M.-C. (2016). The tetrahedral finite cell method for fluids: Immersogeometric analysis of turbulent flow around complex geometries. *Computers & Fluids*, 141:135–154.