Scalable Distributed Feature Tracking and Remapping on Adaptive Unstructured Meshes for Finite Element Simulations

Presented in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

with a Major in

Computer Science

in the

College of Graduate Studies

University of Idaho

by

Cody James Permann

Major Professor
Robert Hiromoto, Ph.D.

Committee
Michael Haney, Ph.D.
Richard Martineau, Ph.D.
Michael Tonks, Ph.D.

Department Administrator
Rick Sheldon, Ph.D.

May 2017

Authorization to Submit Dissertation

This dissertation of Cody James Permann, submitted for the degree of Doctor of Philosophy with a Major in Computer Science and titled "Scalable Distributed Feature Tracking and Remapping on Adaptive Unstructured Meshes for Finite Element Simulations", has been reviewed in final form. Permission, as indicated by the signatures and dates below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor:	Robert Hiromoto, Ph.D.	Date
Committee Members:	Michael Haney, Ph.D.	Date
	Richard Martineau, Ph.D.	Date
	Michael Tonks, Ph.D.	Date
Department Administrator:	Rick Sheldon, Ph.D.	Date

Abstract

Phase field modeling (PFM) is a well-known technique for simulating microstructural evolution. To model grain growth using PFM, typically each grain feature is assigned a unique non-conserved spatial variable known as an order parameter. Each order parameter field is then evolved in time. Traditional approaches for modeling these individual grains uses a one-to-one mapping of grains to order parameters since the interactions among grains is not known a priori. This presents a challenge when modeling large numbers of grains due to the computational expense of using many order parameters. This problem is exacerbated when using common numerical solution schemes including the fully-implicit finite element method (FEM), as the global matrix size is proportional to the number of order parameters squared. While previous work has developed methods to reduce the number of required variables and thus the computational complexity, none of the existing approaches can be applied to an implicit FEM implementation of PFM. Additionally, polycrystal modeling with grain growth and other coupled physics requires careful tracking of each grain's position and orientation, which is lost when using a reduced number of variables. Here, we present a modular, scalable distributed feature tracking and remapping algorithm suitable for solving these deficiencies. The algorithm presented in this dissertation maintains a unique ID for each grain even after variable remapping without restricting the underlying modeling method. This approach enables fully-coupled multiphysics using a fully generalized finite element method. Implementation details and comparative results of using this approach are presented.

Acknowledgments

I would like to thank several of my colleagues who have helped to create an environment and the tools to make computation research possible and exciting at the Idaho National Laboratory (INL); the current staff of the Multiphysics Object Oriented Simulation Environment (MOOSE) Team in alphabetical order: Andrew Slaughter, Brian Alger, David Andrš, Derek Gaston, Fande Kong, Jason Miller, John Peterson, and Robert Carlsen. I'd like to extend a special thanks to Michael Tonks for accidentally starting me on this project several years ago. He has been a great mentor throughout the duration of this project. My adviser, Robert Hiromoto always exercised patience as we debated ideas in his office. I want to thank him for his time and guidance during this work. I'd like to thank Richard Martineau, the director of modeling and simulation at the INL for the emphasis he placed on the importance of working through this degree program and the support to get it done. None of this work would be possible without the loving support of my family: my wife, Amber and my daughter, Karina, and the newest addition, Breckin, as the balance of a full-time job and the added burden of school was challenging for everyone. Finally, I'd like to thank the INL and the Department of Energy (DOE). This work was funded in part first as a Laboratory Directed Research and Development (LDRD) project, and later through the DOE programmatic funding.

Dedication

To my late mother, Carol Ann Permann (1949-2016) and my father, Steven Harold Permann. You believed in me, encouraged me, and provided me with the environment and means to become the man I am today.

Table of Contents

List of Tables

List of Figures

Chapter 1 INTRODUCTION

"Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity."
- Alan Turing

Simulations consisting of multiple evolving features are common in the computational science community. A feature in this context refers to a topologically connected region within the domain that changes over time. Examples include the tracking of gas bubbles in a fluid, the formation of individual cracks in rocks or other media, and the evolution of grains in a polycrystal material. However, analyzing the raw spatial variable values that represent each of these features in these examples isn't sufficient for quantifying the number, sizes, or distribution information as important connection information must also be extracted. The efficient discovery and extraction of feature information is a well studied problem in the computer science field and is known as connected component identification [41, 66, 25, 14, 65]. These techniques have in turn been applied to create distributed feature extraction and tracking techniques for high performance visualization workloads [83, 84, 16, 18].

Polycrystal grain evolution is an important problem in the materials science community. Understanding material microstructure evolution under a wide variety of environmental conditions leads to the design and development of stronger, safer, and longer lasting materials. These materials can then be used in the manufacture of improved vehicles, stronger buildings, and safer power generation. The numerical methods for studying this problem have been continuously improving over the last 30 years. The earliest polycrystal models were Monte Carlo Potts models [3, 87], but since then, researchers have employed front tracking [27, 70], phase field (PF) [24, 49], cellular automata [60], and level set methods [100]. While all of the various models have been shown to predict similar behavior, Monte Carlo Potts models and the phase-field method are the most popular due to their flexibility and computational efficiency.

While the phase-field method was originally implemented using a finite-differencing scheme, the finite element method (FEM) has gained in popularity in recent years due to the myriad of features offered by this approach: irregular domain shapes, unstructured mesh, multiphysics coupling, and mesh adaptivity. However, through all of these advances in numerical techniques, a barrier has remained to running the polycrystal grain evolution problem using the phase-field method with a pure finite element implementation due to the more complex global basis function continuity and consistency requirements of this method. Many scalable approaches to solving the polycrystal evolution problem using a finite-differencing scheme do exist [57, 50, 39, 96, 97, 68]. These methods reduce the problem size by either reducing the number of field variables required through clever remapping techniques or by restricting the computational domain through bounding box techniques. We refer to the former technique as "reduced order parameter" modeling.

The work in this dissertation focuses on the application of a scalable extraction and tracking method along with a solution remapping algorithm to create a novel reduced order parameter finite element phase-field modeling method. The algorithms presented have been developed in a modular fashion to work on a wide variety of problems with an extensive feature set. These capabilities includes the ability to run on fully-unstructured meshes with adaptivity, and the use of arbitrary periodic boundary conditions while remaining agnostic to the underlying physics being simulated.

1.1 Problem Description

Traditional polycrystal evolution models employ individual order parameter variables for each modeled grain that are defined over the entire simulation domain. This is due to the fact that the size and location of the grains undergoing evolution is unknown. This approach lacks scalability as larger systems of grains are modeled as the vast spatial area of each order parameter's representation is unused. However, these variables still need to be stored and computed wasting both memory and computational resources. While there have been several solutions to this problem, their applicability is limited to specific modeling methods. We seek a method that can be completely generalized to run with a fully-implicit, fully-unstructured finite element method with mesh adaptivity.

In order to reduce the number of variables required to represent a large number of grains (features) that can potentially occupy the entire domain, we must employ a technique to reuse variables to represent several grains while maintaining the integrity of each grain. That is, we must assure that grains represented on the same variable are never allowed to interact. A method to remap entire grains from one variable to another dynamically will be used to minimize the number of variables needed for each simulation while maintaining a unique identifier to each grain as it moves to different solution fields. This approach must not interfere with the underlying modeling method, in this case the finite element method. Finally, this approach must be scalable both in terms of the number of grains represented, but also in the efficient use of parallel computational resources.

1.2 Purpose and Scope

In this research we propose a novel extraction, tracking and remapping algorithm and its corresponding implementation within the Multiphysics Object Oriented Simulation Environment (MOOSE) [30] phase-field module. The correctness and performance of this algorithm is also evaluated versus identical phase-field models that do not utilize this new algorithm. All findings and contributions in this research are published as open-source software, free to the public to enable larger and more complex modeling efforts to advance the state of the art. Several parallel algorithms are used for scalability and optimized for efficient execution.

In order to completely generalize the tracking and remapping algorithm, several requirements have been identified that must be satisfied to solve this problem completely:

1. The algorithm shall function when using parallel mesh decomposition.
2. The algorithm shall support arbitrary periodic boundary constraints.
3. The algorithm will provide unique and consistent identities to grains throughout the simulation.
4. The algorithm shall support mesh (h-) adaptivity
5. The algorithm shall function with simulations using fully-implicit time integration.
6. The algorithm shall be dimension agnostic.
7. The algorithm shall function under generalized phase-field problems (no assumptions).
8. The algorithm shall not limit parallel scaling.
9. The algorithm shall work with and not restrict the use of a general finite element method.

1.3 Original Contribution

The primary original contribution of this work is a scalable, general purpose, feature extraction, tracking, and remapping algorithm designed to operate on a finite element mesh. The purpose is to provide a mechanism for maintaining clean separation of several overlapping features represented by a small number of independent piecewise continuous finite element solution fields for computational efficiency. These algorithms have been designed to work on fully unstructured meshes using either nodal or elemental centric solution fields and can handle both (*h*-) adaptivity and arbitrary periodic boundary conditions.

The number of variables required to represent several competing features has been further reduced when compared to previous algorithms for both 2D and 3D simulations. Additionally several mechanisms aimed at providing robust material microstructure simulation capabilities have been added. These capabilities include selective feature recombination, feature separation, and tunable algorithms for handling "noisy" feature creation detection. These algorithms have been extensively tested on various "grain-growth" simulations with and without a multiphysics solid mechanics component. At the time of publication these algorithms are currently being used by other researchers to explore advanced simulations involving microstructure recrystallization and sintering processes.

A novel method for handling special cases of feature appearance and disappearance in tracking methods has been created. Specifically for microstructure modeling, we have created an algorithm that can detect the differences between grain splitting and grain nucleation based on each grain's halo integrity. This algorithm also handles the case where a grain may be absorbed while another is split leaving the total number of features unchanged. This method has been used on different data sets containing both of these behaviors.

Parts of this work have been published in *Computation Material Science* [74]. Further papers have been submitted [75, 76].

Chapter 2

MICROSTRUCTURE MODELING

"A mathematician . . . has no material to work with but ideas, and so his patterns are likely to last longer, since ideas wear less with time than words."
- G. H. Hardy

Microstructure modeling is an important aspect of materials science. It is a tool for understanding the mesoscale structure of engineering materials to predict chemical, mechanical, electrical and other properties of materials used in manufacturing and engineering. Advances in modeling and simulation reduces the costs of analysis which can lead to stronger and lighter materials with better properties to resist environmental impacts that lead to the degradation of that material. The applicability of material modeling and simulation cross cuts several industries such as power generation, civil engineering, and transportation. The interest of the research is focused mainly on the study of advanced nuclear fuels being designed to produce the next generation of safe nuclear reactors, but the ideas and algorithms here are widely applicable to each of these fields.

One focus area of microstructure modeling involves the analysis of polycrystalline materials which consist of several crystallites or "grains", which are commonly encountered in ceramics and medals. Within one of these grains, the atoms that form the material are highly structured in a lattice configuration. Each grain is situated within the material at a specific crystallographic orientation. However, neighboring grains are situated at a completely different orientation. In a physical material the number of possible orientations is effectively unbounded. To capture the correct behavior of each grain in a model, these specific orientations must be accounted for in a simulation and is part of the reasoning behind using separate order parameters to model individual grains.

2.1 Polycrsytal Phase-field modeling

In the phase field approach, microstructural features are described using continuous variables. These variables take two forms: conserved variables representing physical properties such as atom concentration or material density, and non-conserved order parameters describing the microstructure of the material, including grains [94]. The evolution of these continuous variables is a function of the Gibbs free energy and can be defined as a system of PDEs.

In [19], a general form of the phase field PDEs is defined. The evolution of all conserved variables is defined using a modified Cahn-Hilliard equation:

$$\frac{\partial c_i}{\partial t} = \nabla \cdot M_i \nabla \frac{\partial F}{\partial c_i}, \tag{2.1}$$

where c_i is a conserved variable and M_i is the associated mobility. The evolution of non-conserved order parameters is represented with an Allen-Cahn equation:

$$\frac{\partial \eta_j}{\partial t} = -L_j \frac{\partial F}{\partial \eta_j}, \tag{2.2}$$

where η_j is an order parameter and L_j is the order parameter mobility. The Gibbs free energy of the system is defined by the functional F.

The free energy functional, for a phase field model using N conserved variables c_i and M order parameters η_j, is described by

$$\begin{aligned} F = \int_V \big[& f_{loc}(c_1, \ldots, c_N, \eta_1, \ldots, \eta_M) + \\ & f_{gr}(c_1, \ldots, c_N, \eta_1, \ldots, \eta_M) + E_d \big] \, dV, \end{aligned} \tag{2.3}$$

where f_{loc} defines the local free energy density as a function of all concentrations and order parameters. A system for solving these PDEs was originally created in a MOOSE-based application called MARMOT [94] and later open sourced as part of the MOOSE framework's phase_field module.

Many of the examples in this dissertation use the grain growth formulation presented in [69]. In this model, each grain is represented by a continuous order parameter η_i equal to one within the grain and zero in all other grains. The evolution of each grain's order parameter is defined by

$$\begin{aligned}\left(\frac{\partial \eta_j}{\partial t}, \phi_m\right) &= -L\left(\kappa_j \nabla \eta_j, \nabla \phi_m\right) \\ &- L\left(\frac{\partial f_{loc}}{\partial \eta_j} + \frac{\partial E_d}{\partial \eta_j}, \phi_m\right) \\ &+ L\left\langle \kappa_j \nabla \eta_j \cdot \vec{n}, \phi_m \right\rangle . \end{aligned} \tag{2.4}$$

from the original Marmot paper [94]. The free energy functional from Eq. 2.3 is defined by

$$\frac{\partial f_{loc}}{\partial \eta_i} = \mu\left(\eta_i^3 - \eta_i + 2\sum_{j=1}^{N} \eta_i \eta_j^2\right). \tag{2.5}$$

This model requires that each grain be modeled by a separate order parameter. This effectively means that the number of grains in a simulation dictates the number of variables needed to realistically predict the physical microstructural evolution. While this restriction is not problematic for small systems, modeling larger systems quickly becomes infeasible even on large scale cluster systems due to the lack of memory scalability.

2.1.1 *Reduced Order Parameter Modeling*

Reduced order parameter modeling in the context of microstructure modeling is the practice of reusing order parameter variables to represent more than one grain in a model. Setting up this kind of simulation simply requires the researcher to distribute the limited number of order parameters among the much larger number of grains being simulated. While this can be done using a simplistic algorithm, an improper assignment can lead to incorrect simulation results. The problem of assigning a valid initial condition is thoroughly discussed in §5. The reduced order parameter modeling method drastically reduces the size of the equation system being solved as it turns the linear growth of variables into a constant overhead as the number of simulated grains is increased. The consequence of using fewer variables becomes apparent almost immediately however as the simulation will produce abnormally large grains very quickly and artificially due to coalescence. During the evolution of the simulation, whenever two grains represented by the same order parameter come into contact, they immediately fuse or coalesce into a larger grain. While this behavior causes no numerical issues and doesn't affect convergence of the simulation, it is non-physical and incorrect. To successfully employ reduced order parameter modeling, one must correctly assign initial conditions and avoid the coalescence problem throughout the simulation.

2.2 Finite Element Solution Data

In the finite element method, a simulation domain is discretized into a finite number of discrete subdomains. These subdomains are typically simple shapes such as lines in one dimension, triangles and rectangles in two dimensions, or tetrahedrons and hexahedrons in three dimensions. These subdomains called "elements" are connected together in such a way as to reconstruct the desired domain geometry. Each element type is given a specific vertex ordering which makes up the element's connectivity. Another data structure then maps the global node ordering to the local vertex ordering which yields information necessary to construct the orientation of each finite element throughout the domain. We then setup a series of basis functions on each element and solve for coefficients that are used to scale those basis functions, which we store in a solution vector. We can use those solution coefficients to reconstruct the original piecewise functions on the elements, evaluating the actual interpolated solution at any point throughout the domain. With this information, it is possible to construct a complete image of any finite element solution.

In the context of PFM, the mesh and solution must be used to find grains for the purpose of feature extraction [16]. One must first understand the layout of finite element solution data. Generally there is a solution vector which contains a coefficient for each globally numbered degree of freedom (DOF). The number and locations of the DOFs depend upon the basis functions being used to represent each variable in the finite element simulation. The basis functions fall into two categories, "nodal" and "elemental", where the locations of the DOFs correspond to the mesh nodes or are positioned within elements respectively. These coefficients are used to reconstruct the basis functions for each variable yielding a solution that spans the domain [8]. The most common basis functions used for finite element analysis come from the "Lagrange" family. These are overlapping piece-wise continuous functions that have a property where only one basis function has a value of one at every node in the mesh while every other the basis function at that node has a value of zero. This property makes it so that the nodal coefficients exactly correspond to the solution at every node in the domain. Additionally, Lagrange functions are typically simple polynomial functions making them very easy to evaluate and integrate over elements.

2.2.1 *Nodal and Elemental information*

As explained in the previous section. Finite element solution data is typically stored at the mesh nodes. There is however a need for capturing information per element too. Any quantity which is only defined, or only makes sense in the context of an element cannot be stored at the nodes. This is due to the fact that nodes are generally shared among several elements in the interior of the mesh. Information stored at a node cannot belong to any given element uniquely, thus it makes sense to store additional information per element in these cases.

The mesh nodes represent a single point in space and therefore do not represent any volume or mass. Only the elements in a finite element simulation represent these quantities. Given this knowledge, one example where element data is typically desirable is for the purpose of representing simulated material properties. Physical materials have mass and volume and map nicely to elemental data. Adjacent elements can be assigned different material properties representing fixed interfaces between materials. These capabilities are useful for simulating real geometries and these material property differences can have large effects on simulation behavior. Since there are many different use cases for storing information at nodes and elements, the algorithms in this dissertation have been designed to work with both for maximum simulation flexibility. For generality, all discussions regarding mesh nodes or elements where the differentiation isn't necessary will use the terms "entity" or "entities".

2.3 Periodic Boundary Conditions

Periodic boundary conditions (PBCs) are a commonly used modeling tool for simulating quasi-infinite domains [21, 20, 91, 94]. They are often employed to reduce computational requirements by reducing domain sizes of models with symmetry. They also conserve quantities of interest such as mass, momentum, or energy, which can be essential in many physical models.

In the finite element library that these algorithms are built upon. PBCs are implemented as constraints applied to a boundary to force the DOFs on that boundary to match those on some other boundary in the domain. Usually PBCs are applied on opposite boundaries for every dimension in the domain to create a quasi-infinite domain. Any non-zero flux through a PBC has an equal flux on the matching boundary. These properties make the application of PBCs in a simulation useful for approximating the macro-scale behavior of materials.

Chapter 3

ALGORITHM DESIGN

"When I am working on a problem I never think about beauty. I only think about how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong."
- Buckminster Fuller

If we consider the full list of requirements given in §1.2, one might consider a post-processing approach for solving this problem. That is, the algorithm should work agnostic of the simulation physics, working only with the solution and geometric information available during the simulation. This approach has the advantage of working on a wide variety of problems and doesn't require the user of the software to know or understand the inner workings of the algorithm. The main goal of this work is to overcome the problem observed in reduced order modeling of grain coalescence within a completely implicit finite element solution scheme. The idea is to observe the solution process, only intervening when necessary to prevent grain coalescence. This is done by assuring that no two grains represented by the same order parameter can ever come into contact.

This idea is closely related to the concept of graph coloring [10]. One can view the connectivity among grains in the microstructure as an undirected graph with the order parameters representing the available coloring in traditional graph coloring problem (Figure 3.1). Avoiding coalescence is then analogous to maintaining a proper coloring as the microstructure evolves. Continuing the analogy, as the microstructure evolves, several vertices will be removed and added and the connectivity of the graph will change requiring that one or more vertices change color to maintain a proper coloring.

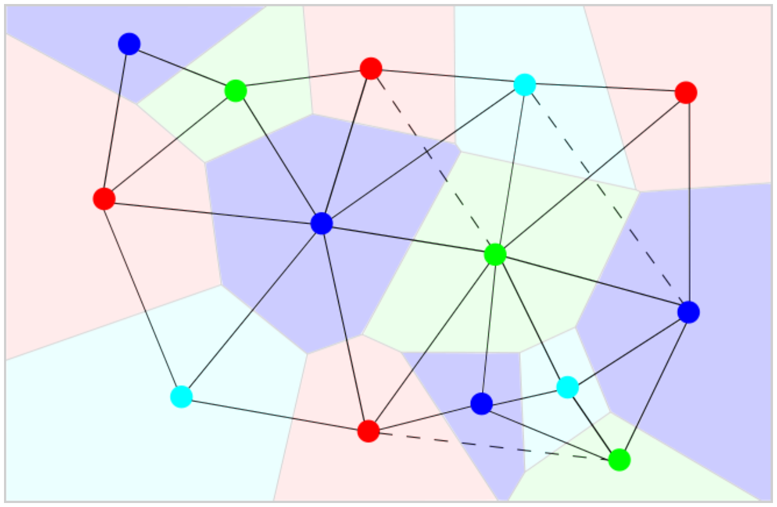


FIGURE 3.1: Grain structure colored by order parameter with the corresponding adjacency graph overlaid. The solid edges in the graph show physical neighbors while the dotted edges show extra edges added due to feature halos (not pictured).

3.1 Grain Identification

The first step in tracking grains is identification. Given a finite element solution we must be able to identify the physical extents of each grain throughout the domain. In a traditional phase-field polycrystal simulation, each grain is represented by a unique variable so grain identification is trivial. The size and shape of each grain can simply be recovered by inspecting the values of each independent variable. However when utilizing reduced order parameter modeling §2.1.1, several grains can be represented by each order parameter. Recall, that the value of the order parameter represents the location of the grain. A grain interior is located anywhere where the order parameter value is equal to one. No grain is represented by a value of zero and the grain boundary is represented by all values between zero and one. When the same order parameter is reused to represent multiple grains, what differentiates once grain from another is the proximity of one set of ones which are disjoint from another set of ones. In other words, the value of an order parameter must vary from one to zero and back up to one again over some finite region within the domain.

Grain identification occurs by finding all connected regions of ones for each order parameter. This is done by iterating over the mesh entities looking for values above a chosen threshold that indicate grain interiors. Once a value is found, a flood (or seed) fill algorithm is used to find all neighboring values above a given threshold on the unstructured mesh.

Note that direct inspection of solution values given at the nodes does not work for the general case. For many finite element types, the solution value at a node directly corresponds to the value at that location, but that property is not guaranteed by the FEM method. For the general case, the value is found by evaluating all of the shape functions at the given point and summing them.

3.2 Grain Tracking

Microstructure evolution is complex and grain shapes and orientations may change quite drastically over the course of a simulation depending on the boundary conditions, material properties, and other forces, both internal and external. Rather than creating an algorithm that attempts to predict grain structure from step to step, the approach taken with this work is to deduce the positions of each grain based on information from the previous step. The first invocation of a tracking algorithm would record the size and positions of all grains in the simulation. Each subsequent invocation would then compare the current step to the previous step to determine the evolution of grains, handling processes such as growth, shrinkage, absorption, splitting, and nucleation.

The first step in tracking grains is to employ feature extraction techniques and component identification [66, 25, 14, 65] choosing suitable information to uniquely identify the locations of all grains in a given step. Each grain is identified by its centroid and the current order parameter representing the grain. Both pieces of information are necessary to assure positive identification from one step to the next for cases where multiple grains have nearly coincidental centroids. This may happen where grains are highly elongated or slightly concave and concentric. Tracking is accomplished by simply comparing and minimizing centroid movement of all grains for each order parameter separately.

3.3 Graph Discovery and Validity checks

In order to maintain a clean separation of individual grains represented in the same manner by the same spatial variables, we must detect features that will potentially interact with one and other. We frame this problem in terms of a graph coloring problem and implement a solution that builds on techniques for solving that type of problem. After grains have been identified, graph discovery occurs based on the microstructure and a validity check on the proper coloring is performed (i.e. no neighboring grains may have the same order parameter). The connectivity of the graph is defined by neighboring grains. It is not necessary to build the graph explicitly. Instead it's sufficient to determine if any grains represented by the same order parameter are neighbors. If all pairs of grains represented by the same order parameter are not neighbors, then the graph coloring is proper and no action further action is required by the algorithm.

3.4 Grain Remapping

Should grains represented by the same spatial variable be determined as potentially interacting, we must employ a mechanism to dynamically remap one or more grains to maintain a valid system state. Recognizing that a simple single remap operation may not be sufficient for recoloring a graph, an algorithm must be designed to handle several remapping operations to maintain a valid state. We employ a recursive recoloring algorithm that may explore a chain of remapping operations to dynamically recolor the graph as the simulation evolves. Special cases must also be handled for grain absorption and nucleation.

3.5 Scalable Parallel Design

It is relatively easy to build a serial algorithm to meet the requirements for a dynamic tracking and remapping algorithm to satisfy the run time requirements of the phase field model. In fact, this dissertation started out of a prototype implementation that was completed in just a few days to handle this problem. The challenge in this work lies in building a scalable parallel algorithm. If the size, shape, and location of each feature is unknown and unrestricted, a sufficient amount of information to uniquely identify each feature must be shared among processors without replicating too much information. The algorithm design in this dissertation uses an asymmetric parallel design, which means that the algorithm works differently on different ranks. All of the global information is maintained and processed on a single rank with all of the remaining ranks having minimal local to global mapping information. The detailed implementation of this design are discussed thoroughly in the next chapter.

Chapter 4
Implementation

"I often compare open source to science. To where science took this whole notion of developing ideas in the open and improving on other peoples' ideas and making it into what science is today and the incredible advances that we have had. And I compare that to witchcraft and alchemy, where openness was something you didn't do."
- Linus Torvalds

All of the work in this dissertation is based upon the open-source Multiphysics Object Oriented Simulation Environment (MOOSE) package and is now included as part of the main distribution. MOOSE leverages several other high performance libraries: libMesh [51], PETSc [4, 6, 5], MPI [38, 28, 56] and Hypre [61]. This list represents only the solver stack and not the complete list of third-party tools leveraged by each of these libraries. The implementation details require a brief overview of the MOOSE architecture as well as pluggable systems leveraged for this work. These systems are covered followed by a detailed discussion of the design and implementation of the algorithms themselves.

4.1 MOOSE Pluggable Systems

The MOOSE framework provides several pluggable systems for implementing new functionality via standardized interfaces. These interfaces are consistently designed so that creating new functionality is similar regardless of which system a developer is leveraging. Some of the key highlights of these systems are standardized constructors with the ability to retrieve arbitrary numbers and types of parameters, rich inherited class members, and prolific coupling interfaces. MOOSE typically does a good job of hiding the complexities of parallel programming and concurrent execution, both in terms of shared memory and distributed memory when possible. The main pluggable systems are designed around the support for solving nonlinear multiphysics partial differential equation (PDE) operators focusing on straightforward coupling capabilities and modularization. There are also several interfaces designed for auxiliary calculations and on-line post-processing functionality.

The algorithms described in this dissertation are implemented using several of these interfaces but only a small subset of the total available interfaces. The evolution of the algorithmic implementations of this work have also impacted some of the core designs and capabilities of the framework itself. These enhancements are described in §4.6. Each of the systems utilized by by this project are described in §4.1.1, §4.1.2, §4.1.3, and §4.1.4.

4.1.1 *MOOSE UserObjects*

The MOOSE `UserObject` system is arguably the most flexible system in the framework. It is typically utilized any time a developer needs to implement a new capability that doesn't match up well with the highly tuned interfaces available in the other MOOSE systems. It is usually employed to maintain user-defined data structures and can be executed over a wide range of mesh related entities such as elements, nodes, sides, and boundaries in a parallel agnostic fashion. It can also just be used for general purpose calculations that may or may not relate to any temporal or spatial data at all. The execution control of UserObjects and several other systems in MOOSE is highly customizable. UserObjects are normally designed to be executed once per time step, but can be tuned to be execute more or less often if required. The available modes of execution are `INITIAL`, `TIMESTEP_BEGIN`, `TIMESTEP_END`, `NONLINEAR`, `LINEAR`, and `CUSTOM`.

Like many MOOSE systems, UserObjects can be coupled to almost every other system. However, the UserObject system is unique in that it is designed to be coupled by direct type, not just through a base class interface. This means that developers are allowed to, and encouraged to implement new interfaces on the objects that they create which can be accessed directly by the objects that couple to them. This allows developers the freedom to build, aggregate, and maintain rich data structures that can be accessed by one or more objects in the system for the purpose of performing calculations outside of the core design of the framework. The `UserObject` system serves as the basis for several other systems such as the post-processing systems (§4.1.2, §4.1.3).

4.1.2 *MOOSE Postprocessors*

`Postprocessors` in MOOSE are objects designed to consume information from elsewhere in the simulation to produce a single scalar value at each invocation. These objects are typically employed to produce aggregate values based on solution data. They have interfaces and helper methods for aggregating data across threads and processes when appropriate. The scalar values they produce get output to the screen and most supported file formats automatically. They are an extension of the `UserObject` system so they inherit all of the member data and interface functionality of their parent objects.

The core algorithms for this dissertation have been implemented as Postprocessors. The convenience of being able to output the count of features or grains while having the full flexibility of the UserObjects makes this a logical choice. It also fulfills the core design of this implementation as proposed in §3: In order to create a general purpose tool that is physics independent, it should be implemented as a post-processing capability to avoid interfering or restricting the simulation process.

4.1.3 *MOOSE VectorPostprocessors*

`VectorPostprocessors` are the "vector" equivalent of Postprocessors. These objects can produce multiple arbitrary length, named vectors at each invocation. The output of these objects is managed through the framework into a series of comma separated values (CSV) files separate from the normal mesh-based output. The interface is similar to that seen in the UserObject and post-processing systems. The output format has no restriction on the length of the vectors and they may differ from invocation to invocation. The information in VectorPostprocessors can also be retrieved via a coupling mechanism so that objects may access data live, during a simulation.

4.1.4 *MOOSE AuxKernels*

The `AuxKernel` system is used to calculate spatial field data based on other simulation information (generally solution data) that can vary over time. The calculations performed in AuxKernels produce coefficients that form the basis of shape functions in the domain. This means that AuxKernel fields can generally be evaluated over the entire domain with continuity restrictions based on the shape function families chosen to support these coefficients.

These objects are generally employed to perform simple arithmetic calculations for variables that are not directly a part of the PDEs that are being solved. For instance, stress and strain calculations are typically a function of displacement, which is the variable being solved for in solid mechanics models. They can also be used as a "spatial post-processor" output capability for several types of mesh related data. Many different spatial fields are produced as part of the work in this dissertation such as the ability to view unique grain information, or the variable assignments to every grain over time. Each of these fields can be visualized through MOOSE's AuxKernel system.

4.1.5 *Scalable Distributed Mesh Terminology*

Before we discuss the distributed extraction algorithm that makes up the basis of the main algorithms in this dissertation, a few terms need to be defined. As we inspect portions of the FEM mesh and look and entities and neighboring entities, we must ensure that portions of the mesh and solution data structures that we need are actually available on the local processor. For scalability, it cannot be assumed that the solution for a given element is available, nor can it be assumed that the geometric information about the element itself is even available [51]. The notion of entity ownership must first be discussed. When the FEM mesh is partitioned, elements are assigned to different partitions which are then assigned to different processors. Each element is given exactly one owner and that attribute is saved for every element. If repartitioning occurs, elements may be reassigned to different processors but they can only have a single owner at any given time. All information relating to an element that is owned by the local processor is always available. This includes any solution information or portions of field variables that have degrees of freedom on those elements and the nodes that make up those elements.

The presence of information on any element that is not owned by the current processor must be verified before access. It's possible that when we query an element for a list of its neighboring elements, we instead get pointers to proxy objects telling us that there are indeed neighboring elements, but that they're not available on the current processor. To differentiate between these two cases, we use the term "ghosted" to indicate that an element is geometrically available on the local processor. However, even if the element is geometrically available, the solution information on that non-local element may not be. Here, we use the term "evaluable" to indicate that the solution information is also available. Furthermore, evaluable automatically implies that the element is "ghosted" since we cannot interpolate a solution across an element without having the geometric information of the element present on the local processor. The underlying FEM library is assumed to always maintain one level of evaluable elements surrounding each partition. These elements are necessary for the normal finite element assembly processor and can be assumed. This means that all of our first level edge-neighbor elements of the current partition can always be inspected without explicitly checking if they are first “ghosted” or “evaluable”[1].

Each node in the mesh also has an owner which can differ from one or more of the attached elements since nodes can be shared among several elements. While the same rules apply to the availability of information for local versus non-local nodes there is one notable exception. All nodal solution information for a non-local node is also available as long as one of the local attached elements shares ownership of that node.

4.2 The FeatureFloodCount Postprocessor

When analyzing field results from a simulation, it is straightforward to compute statistical information such as medians, means, modes, extreme values and fluxes. This is done by inspecting all of the data in any order and maintaining running calculations. Obtaining other types of information such as the number of “features” like gas bubbles or crystallites (grains) can be much more challenging. This is especially true when working with complex geometries, adaptive meshes, parallelism, and periodic constraints.

A method for identifying solution features must first be implemented before more advanced processing can occur. This is the purpose of the `FeatureFloodCount` Postprocessor. A feature here is defined as a series of similar values (usually above or below some desired threshold) collocated geometrically without any discontinuities. Physical isolation of a pocket of values from another pocket of values is what defines the individual features represented in a solution field. The `FeatureFloodCount` Postprocessor distinctly identifies the extents of every feature in any number of field variables. As previously mentioned, it may be used to provide the statistics on single variable fields such as the number and size distribution of gas bubbles in a domain or it can uniquely identify the individual grains contained in multiple order parameter

[1] These terms are not my own and do not exist in published work, they are part of the libMesh library.

fields. This Postprocessor functions properly in parallel, handles fully-unstructured meshes with or without *h*-adaptivity, as well as periodic boundary constraints. The FeatureFloodCount object implementation is very similar to the work in [16, 18, 17, 82]. However, this implementation also supports unstructured mesh and arbitrary domain shapes.

4.2.1 *Distributed Feature Extraction*

Before we are able to track or perform operations on simulation features, we must first define the identifying criteria for a "feature" and implement a process for extracting them from a finite element solution in some distributed fashion. We define a feature to a be a topologically connected region in the solution domain that has some common attribute. In a finite element setting, this usually means a connected region of elements with a similar variable value or values above or below some pre-chosen threshold. In parallel unstructured mesh simulations, typically each processor works on a subset of the domain, or a partition [85]. Each partition consists of several non-overlapping elements which can be processed independently to advance the simulation. To begin, the distributed extraction process, each processor iterates over the elements of its partition and inspects the values of the variable of interest. When a variable value is encountered that matches the feature identification criteria, that element becomes the starting point or seed location for the start of a new feature. A standard seed or flood fill algorithm [63] is then used to recursively visit all of the neighboring elements until the extent of the feature is explored, or an edge of the processor's partition is reached. This approach is similar to that described in [16].

When a seed is identified, the flood algorithm builds a list of all the face-neighbor elements relative to the current element. If mesh *h*-adaptivity is being used, there may be multiple neighbors for a given side if the adjacent elements are finer than the current element. There are no modifications necessary for this scenario, each of the refined elements is visited in the same fashion as if there were only a single neighboring element on a given side. Handling all of this neighbor visitation recursively allows us to construct and update a data structure containing all of the relevant information about these geometrically connected regions on each individual processor for further processing.

4.2.2 *Features Data Structure*

Each feature is represented by a data structure, shown in code listing 1, containing several attributes that are created and manipulated as the algorithm progresses. When the flood fill algorithm identifies the start of a feature as described in §4.2.1, a new instance of the `FeatureData` struct is created and populated. During construction several attributes are populated. First, the current entity ID is inserted into the `_local_ids` set. The `_var_idx` attribute is set to the current variable representing the newly constructed feature. An initial entry is made into the `_orig_ids` list to track the original processor's local feature ID and processor ID indicating which processors own a portion of the feature once it's been merged by the master MPI rank (§4.3). This list length always remains at one on all non-master processes since they never contain the full global feature map. Finally, the feature's status is set to either `MARKED` if the initial entry is above the "starting threshold", or `INACTIVE` if it's only above the "connecting threshold" (§4.3.3). Additional statistical information may also be recorded when appropriate (§4.5.1). After the recursion completes, the structure contains all of entities on the local processor belonging to the current feature along with additional attributes.

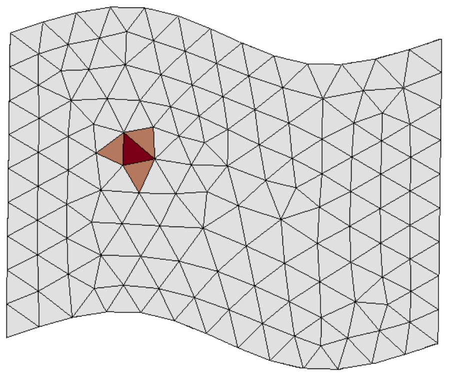
```
enum class STATUS
{
  CLEAR = 0x0,
  MARKED = 0x1,
  DIRTY = 0x2,
  INACTIVE = 0x4
};

struct FeatureData
{
  std::set<dof_id_type> _ghosted_ids;
  std::set<dof_id_type> _local_ids;
  std::set<dof_id_type> _halo_ids;
  std::set<dof_id_type> _periodic_nodes;
  unsigned int _var_idx;
  unsigned int _id;
  std::vector<MeshTools::BoundingBox> _bboxes;
  std::list<std::pair<processor_id_type,
    unsigned int>> _orig_ids;
  dof_id_type _min_entity_id;
  unsigned int _vol_count;
  Point _centroid;
  STATUS _status;
  bool _intersects_boundary;
}
```

CODE 1: Data structure describing the attributes for an individual feature.

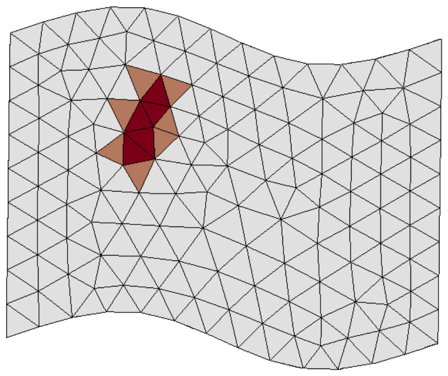
During the flood routine, additional attributes in this structure are also updated. The `_halo_ids` set is used extensively in the tracking (§4.4.1) and remapping (§4.4.4) algorithms discussed later. Also the `_ghosted_ids` set is updated which is used for the "stitching" routine (§4.3.2) when information from all of the parallel ranks is consolidated. The status of the neighboring entities determine when these additional sets are updated. If the ownership of the neighboring entity differs from that of the current processor, we insert the neighboring element's ID into the ghosted set. The recursion itself is performed if and only if the current entity is owned by the local processor. It's not a function of the ownership of the neighboring entity.

If it's determined that a recursive call is to be made in the flooding routine, we save the neighboring ID into the halo set just before the recursive call. This detail is vitally important in avoiding spurious halo entries in the middle of our features. If we save every neighboring entity of the current feature into the halo set up front, but then the recursive call isn't made, we end up with an extra halo interface along the partition boundary that cannot be masked off later because the root processor lacks the local ID information necessary to make that correction (§4.2.4, §4.2.5).

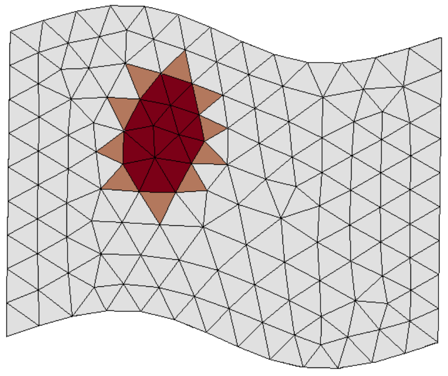
The in-progress discovery of a feature and its halo elements is shown in figure 4.1. Several additional attributes are also recorded for use by later stages of the tracking and remapping algorithms. When we visit a ghosted element, we save that information as it is used to stitch features together. Each feature data structure also contains a variable ID, sorting ID, bounding box information, and flags indicating the state of the feature. Each of these attributes is described in more detail later. Depending on the type of simulation being run, it may be necessary to expand the thickness of the halo (number of element edge neighbors from any interior element in the feature), this is discussed in §4.4.4. When applicable, this thickness is expanded after the flood stage is complete and we have the single level of halo elements created during the flooding process (Figure 4.1(c)). This expansion is accomplished by appending all neighboring elements of the halo elements set for each additional layer of thickness desired. After the additional elements are added to the halo set, the original interior



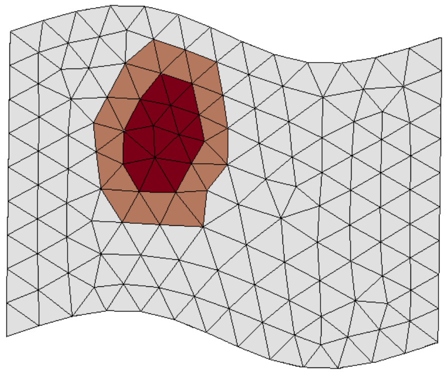
(a)



(b)



(c)



(d)

FIGURE 4.1: The feature and halo identification stage on an unstructured mesh. The feature markings are shaded dark and the halo elements are shaded light. (a) shows the seed element and the corresponding halo elements. (b) shows the feature as it's being identified. (c) shows the completed feature and corresponding halo elements. (d) shows the halo after it's been expanded.

elements representing the feature are subtracted (set difference) from the halo set to maintain distinct feature and halo sets. An expanded halo is shown in figure 4.1(d).

4.2.3 *Halo and Ghosted Set Construction*

The `_halo_ids` set contained within each feature is used to hold a list of entities immediately surrounding the feature excluding IDs that make up the feature (local IDs). This information is used for tracking grains (§4.4.1) and during the remapping operations described in §4.4.4. The construction of this set occurs during the recursive portion of the flooding stage. After the addition of the current entity ID to the `_local_ids` set the algorithm constructs a list of active neighbors relative to the current entity. This list handles mesh adaptivity so it's possible to have multiple neighbors on each side when there is a mismatch in the adaptivity level.

Algorithm 1 Expand Halos

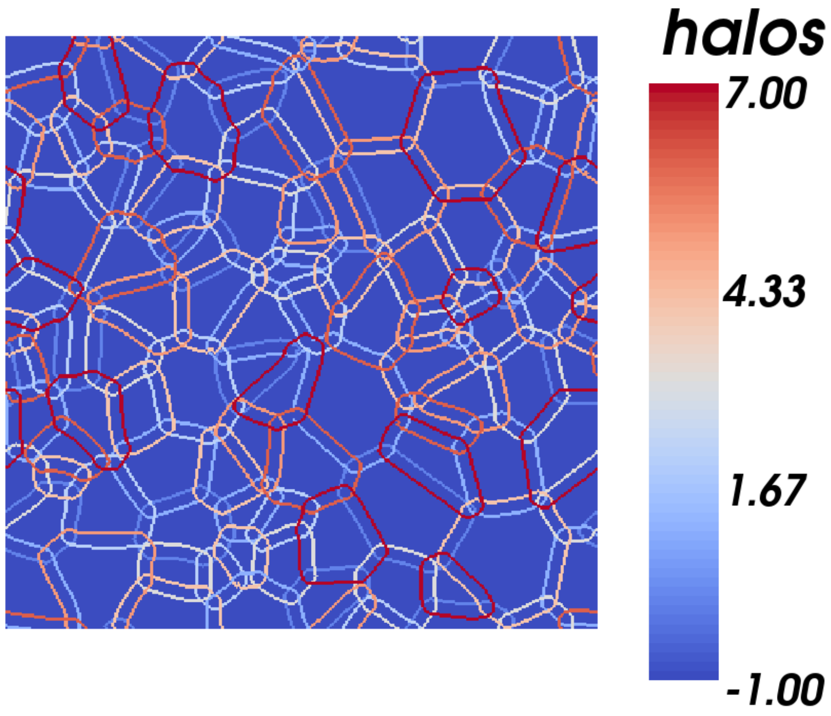
```
1: for $level$ in 1..$level$ do
2:     for all $f$ in L do
3:         for all $entities$ in $f.halo\_ids$ do
4:             $f.halo\_ids \leftarrow f.halo\_ids \cup entities.neighbors$
5:         end for
6:     end for
7: end for
```

The halo "thickness" can be expanded after the full extent of the feature is explored on the local processor. This is accomplished by adding all neighboring entities of all current halo entities for each additional layer of thickness desired. This is done to allow a sufficient interface width for the phase-field method. Without knowing the required thickness, we performed several experiments to determine this value empirically. We found that a minimum thickness of two was required for quadrilateral and hexahedral elements when using a linear Lagrange basis. Moving to triangle and tetrahedral elements bumped the required thickness to three. A thickness of one failed to prevent coalescence in several cases. This was due to an insufficient interface width required by the phase-field method. However, since pairs of halos are always compared against one and other, the total number of elements in the interface is always greater than the thickness of an individual halo. For quadrilateral and hexahedral elements, the effective separation of two features is always double the halo width. For triangle and tetrahedral elements the separation of two features can vary depending on the alignment of halo comparisons.

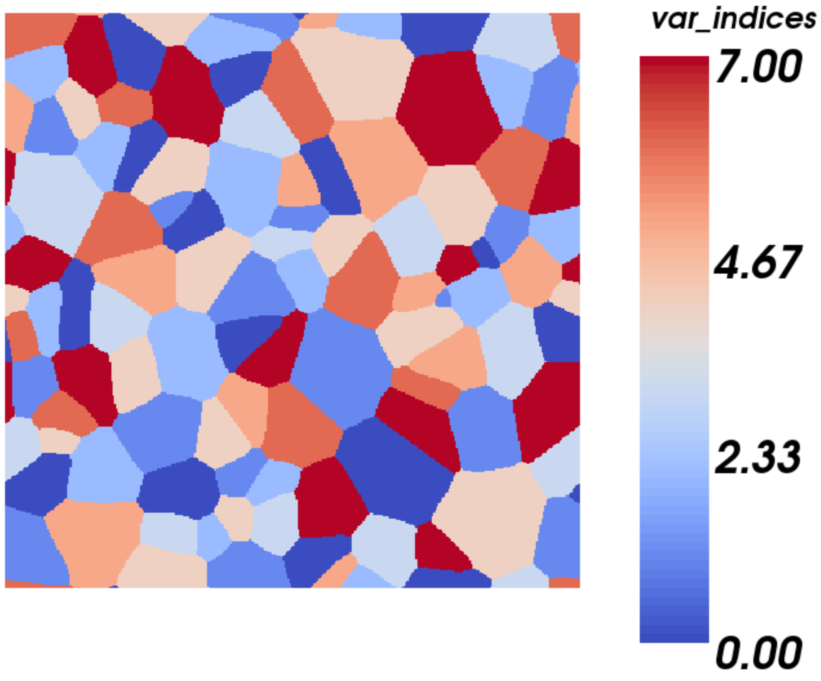
Note that the halo set is always a superset of the local IDs set. This is enforced by beginning each new feature with an entry in the halo set. Every other addition to the local IDs set is always preceded by an addition to the halo set just before the recursive flood call. This allows the halo to be properly created by using a set difference method described in §4.2.4.

Figure 4.2 shows the halos and corresponding microstructure from a 2D grain-growth simulation constructed from a parallel simulation run on a 12-core workstation. Four levels of adaptive mesh refinement were used on this quadrilateral mesh to give the halos a very fine, high fidelity appearance. The very tight halos allow for very dense clustering of features for maximum reuse of variables to represent the features in this microstructure. Note the difference in scales between figures 4.2(a) and 4.2(b). The halo image has a low value of -1, while the low value on the grain image begins at 0. The reason for the difference is that every element in the grain image is occupied by a grain with indices in the range $[0 - 8)$. The halos however do not occupy the interiors of any grain so they are assigned a value of -1 in the resulting image. These images are produced by using an `AuxKernel` to report information from the `FeatureFloodCount` object.

Figure 4.3 shows the volume rendered halos from a 3D grain-growth simulation on a hexahedral mesh without adaptive mesh refinement. Two hundred elements were used along each dimension for a total of eight million elements.

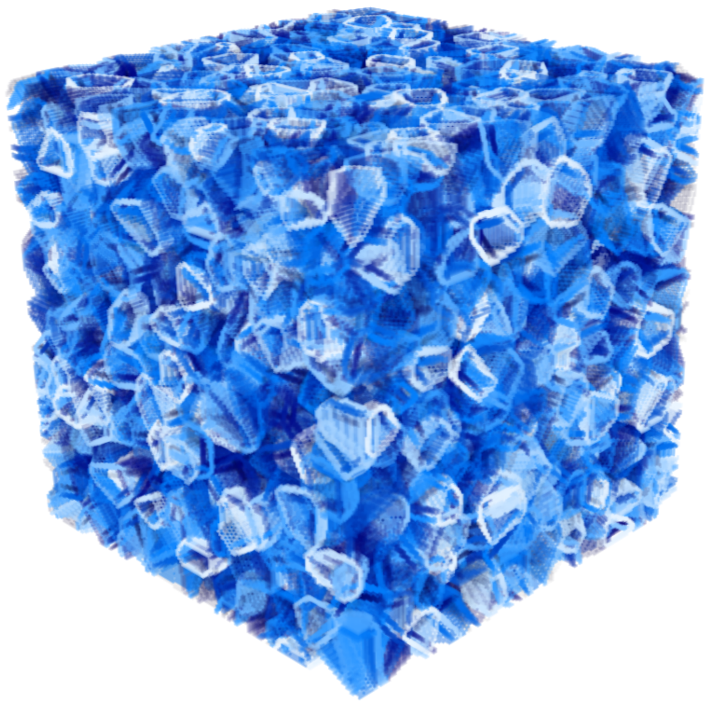


(a)

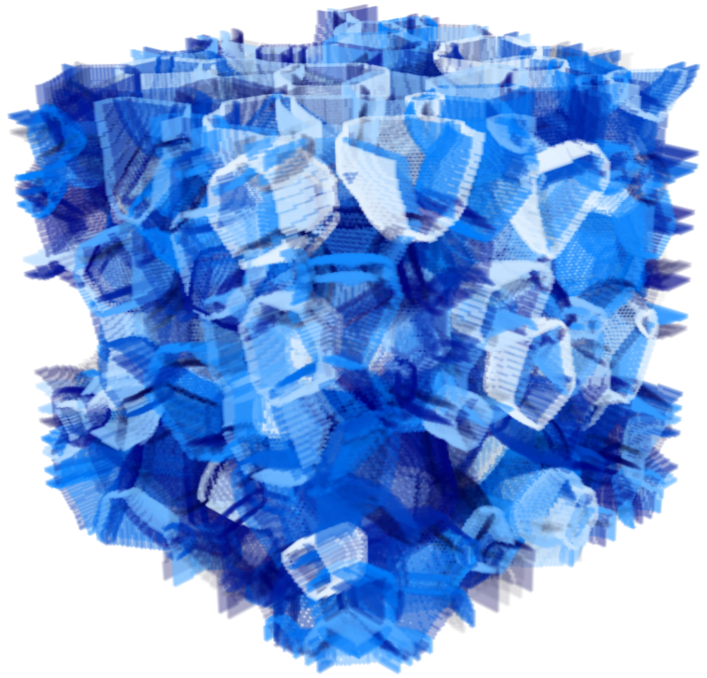


(b)

FIGURE 4.2: 2D simulation with 100 Grains represented by 8 order parameters. Grain halos are shown in (a) with corresponding order parameter (variable indices) assignments shown in (b).



(a)



(b)

FIGURE 4.3: Volume rendered halos from a 3D run containing 6000 grains. (a) shows the halos in the initial condition. (b) shows the halos of the larger grains after several time steps.

4.2.4 *Local Feature Data Processing*

To maintain a scalable algorithm, only a subset of the local feature data from each processor needs to be serialized and gathered on the root processor using the MPI protocol [38]. While the creation of this remaining information could technically be handled during the recursion stage, it adds unnecessary complication and is therefore handled separately. A loop over each feature is performed to populate the `_periodic _nodes`, `_bboxes`, and `_min_entity_id` variables. The periodic nodes set is used to hold feature entities that lie on a periodic boundary constraint (§2.3). These nodes represent an additional non-topological stitching interface. This information is used in a similar fashion as the ghosted entities are used to stitch feature pieces together on physically overlapped regions. The bounding box (`_bboxes`) variable holds axis-aligned bounding boxes that completely enclosing the individual feature pieces. Finally the `_min_entity_id` variable is populated to create a processor independent feature numbering mechanism. The local feature pieces seen by each processor are dependent on the partitioning and entity traversal order. In order to maintain consistent labeling for a given problem as processor counts vary, each feature is tagged with a piece of mesh dependent information. Each mesh entity contains a globally unique ID. The lowest ID of every entity making up a feature is chosen as the sorting tag for a feature. When the `FeatureFloodCount` Postprocessor is operating on a single variable this guarantees a strict ordering among identified features. When multiple variables are used, a consistent ordering can be derived by using this value in conjunction with the `_var_idx` value (i.e. `_var_idx=0`, `_min_entity_id=0` would come before `_var_idx=1`, `_min_entity_id=0`).

To populate these additional pieces of information a loop over all of the local entities is performed on each feature and each of these variables is simply updated based on the current value and incoming value. For the bounding box value, the minimum or maximum position in each dimension is updated according to any new extremes encountered in the incoming value. The minimum entity id is updated if the incoming value has lower value than the current value. The periodic nodes have one additional special case to handle. Periodic constraints are always enforced on nodes, never elements in the libMesh library. If the `FeatureFloodCount` Postprocessor is operating on elements, the nodes have to first be recovered from the element and compared to the periodic constraint nodes. Any matches are then added to the periodic nodes set. For generality in the stitching algorithm and communication efficiency discussed in §4.2.5, all matches are added to the periodic nodes set, not just the corresponding match or matches. For example, if node 0 and node 100 are periodically constrained to one and other, then if either one of them is encountered directory in the local IDs set, they are both added to the periodic nodes set.

Finally, the halo set that was discussed in §4.2.3 is adjusted to remove all interior entries. All of the local ids are removed from the halo set except for those contained within the ghosted ids set:

$$halo = halo \setminus (local \setminus ghost) \tag{4.1}$$

This set operations for computing the final halo entries is described in equation (4.1) and depicted in figure 4.4.

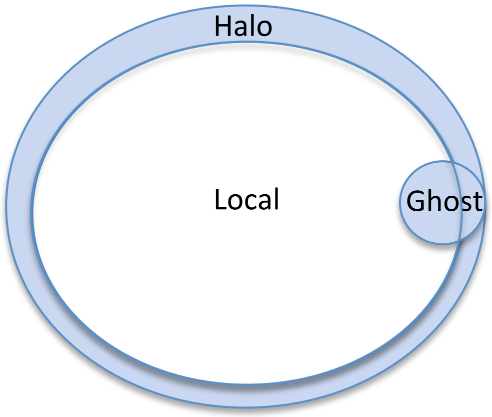


FIGURE 4.4: A Venn diagram showing the relationship between halo IDs, local IDs, and ghosted IDs prior to the adjustment described in §4.2.4. The shaded region shows the IDs remaining in the halo set after the adjustment.

4.2.5 *Data Serialization, Deserialization, Backup, and Recovery*

The parallel communication of feature data takes advantage of multiple MOOSE and libMesh technologies requiring very little code or effort to accomplish. The serialization and deserialization of data occurs by specializing MOOSE's built-in templated `dataStore` and `dataLoad` routines for this new type. These routines are normally used to serialize user-defined data into a byte stream to be stored in memory or on disk and restored as needed by the framework for the purpose of checkpointing or advanced execution strategies. We take advantage of the serialization process to simplify the parallel communication of the FeatureData structures introduced in §4.2.2. These routines can be used to efficiently pack several variable-length records into a single buffer for communicating via MPI.

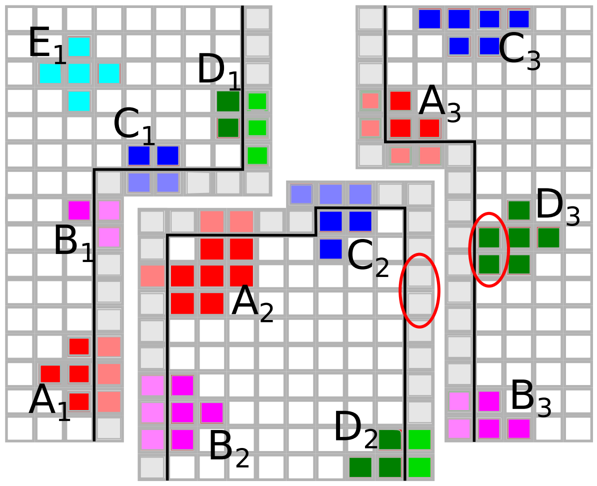
To reduce the amount of data communicated, we avoid serializing the local IDs portion of the data structure, which is usually the largest single piece of information. These local IDs represent the interior of every feature and are not useful in reconstructing the global feature map since they, by definition, are not used in any stitching operation. After the data is serialized into a single byte stream, every processor participates in a parallel "gather" operation where every rank communicates their buffers with the root process. Each buffer is unpacked and deserizlized into a large linked-list data structure on the root process where all stitching takes place.

4.3 Connected Component Identification

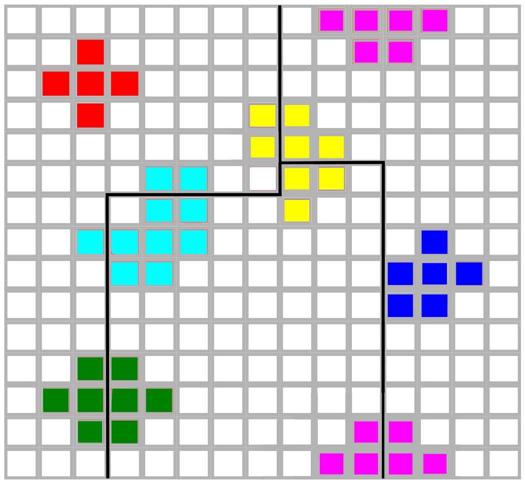
After all feature information has been extracted, serialized, and communicated with the root process, that data must be stitched together to recover a consistent global representation of all features. This stitching process is known as connected component identification [41]. While there have been several efficient connected component algorithms that work on both serial and distributed graphs [66, 25, 14, 65], the lack of a readily available graph data structure in this algorithm make the implementation of those complex algorithms less advantageous in this context. Furthermore, the approach taken in both [66, 25] assume that graph vertices and edges can be inspected at random and used in repeated set operations to find connected components. In our implementation neither the complete set of vertices or edges are readily available in our distributed finite element mesh data structures. As discussed in §4.2.1, each processor instead only runs the flood algorithm on a subset of the mesh and the depth first search algorithm consumes only a very small amount of the overall run time of the algorithm (typically $< 1\%$ in measured runs). However, since each processor only contains a small portion of the solution at any given time, the challenge of joining the partially identified features cannot be avoided.

Figure 4.5(a) shows a mesh partitioned three ways with several distinct elemental features illustrated as colored regions. The alpha characters used in the labeling represent a possible local identification order. The numeric subscripts represent the processor IDs. The label B_1 would represent the second identified feature on processor one. These separate features on the individual partitions represent partial features prior to stitching. The processor partitions are shown as solid bold lines with a layer of ghost elements, which are shown on the opposite side of each partition (lightly shaded elements). The ghost elements mirror corresponding elements and their current variable solutions on neighboring processors. This information is automatically maintained by the framework. Note that the features labeled B_1 and C_1 in figure 4.5(a) are distinct. They only share a common node; they are not edge neighbors so flooding fails to see that they are part of the same feature without information from the neighboring processor.

Each processor only iterates over its partition's elements (unshaded elements) as it looks for feature seeds to initiate the flood fill algorithm. Once the flood fill begins, both local and ghosted elements are inspected so that overlapping elements will be



(a)



(b)

FIGURE 4.5: Regular grid with 6 features partitioned 3 ways. Ghost elements and local identifications are shown in (a). The alpha character represents a possible feature identification ordering. The subscript represents the processor ID. Global identification is shown in (b)

marked appropriately for stitching. The elements circled in Figure 4.5(a) illustrate the effect of only beginning the flood fill on local elements. Feature D_3 has values that appear in processor two's ghosted region but not anywhere in processor two's local region. Note the substantial difference in feature counts between figures 4.5(a) and 4.5(b). Prior to communicating the data structures to the root processor and stitching the partial features, the global feature count in this example is thirteen (five on the first processor, and four on both the second and third processors). The final global count is six (seven if periodic boundary conditions are not used).

4.3.1 *Handling Periodic Boundary Conditions*

If periodic boundaries are used in the simulation, grains intersecting any of the domain edges that are on a periodic boundary are partially represented at multiple disjoint locations on the domain. During the flooding stage these disjoint pieces are generally recognized separately (multiple feature data structures), and are almost always on separate processor partitions. A typical feature intersecting a periodic boundary is shown as the magenta color in Figure 4.5(b). The individual pieces are labeled as D_2, B_3, and C_3 in Figure 4.5(a). Periodic constraints can be applied in a variety of ways. On regular grids they most often appear on opposite sides, however for some 2D and 3D simulations, only a subset of the axes might be made periodic while others are not. Additionally it is possible to apply periodic constraints on orthogonal sides instead of parallel sides. With all of the generalities available in applying periodic constraints no assumptions can be made about how offsets are applied to join separated pieces of a feature. Features split by a periodic constraint are never physically joined or treated as a single continuous feature. Instead, each piece of a periodic feature maintains its own axis-aligned bounding box. This greatly simplifies not only stitching (§4.3.2) but also tracking (§4.4.1), since bounding box comparison logic does not require special logic cases when addressing periodic features. For cuboids, the maximum number of disjoint pieces of a single feature is a function of the physical dimension: 2^{dim}.

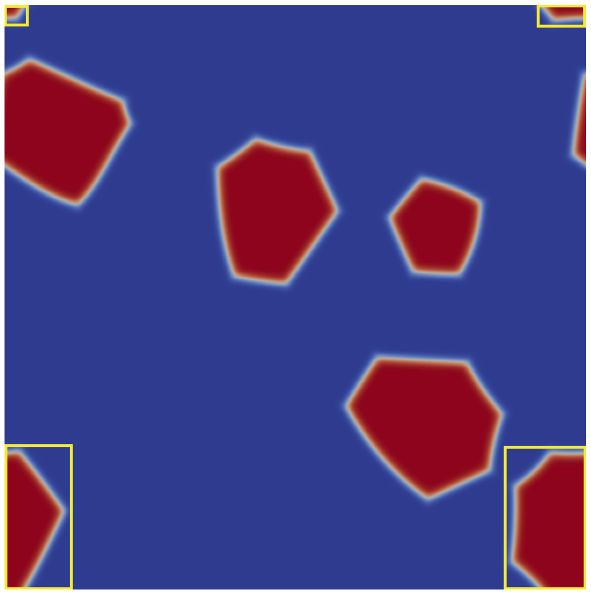


FIGURE 4.6: Solution field for one order parameter (η_i) of a grain-growth simulation where periodic boundary conditions are applied. The feature appearing on all four corners of the square domain is a single grain. Its representation includes four bounding boxes highlighted in yellow.

Figure 4.6 shows an order parameter field on a square domain from a simulation where periodic constraints have been applied to both pairs of opposite sides (top/bottom, left/right). In the figure, a single physical grain appears on all four corners of the domain and is represented by four bounding boxes (highlighted). In three dimensions, a grain appearing on the corner of the domain would appear on all eight corners and would require eight corresponding bounding boxes to completely represent the feature.

4.3.2 *Feature Stitching*

After the feature information has been gathered on the root process, it has all of the information needed to identify and uniquely label all global features. It is important to re-emphasize that the information on the root process is not complete. None of the local identifiers are sent during the data serialization routine (§4.2.5) so all partial features not identified by the root process do not contain any local ids (the volumetric interior element ids). All other data structure members populated during the flooding routine (§4.2.1) and local processing (§4.2.4) are available.

There are two types of stitchings that can occur: "physical stitching" and "logical stitching". The former is the normal case where features physically overlap and are split only due to domain partitioning. The latter is the case that arises due to the use of periodic boundaries. Each of these cases requires special treatment in the way that they are handled. The algorithm is illustrated in algorithm 2. The root processor begins the stitching of the partial feature structures by comparing all possible pairs and checking to see if each pair are merge candidates. A quick check is performed to make sure that the current pair of features are identified by the same variable number. Information identified by different variables fields is never merged.

The merge candidacy first performs a coarse level check on all pairs of axis-aligned bounding boxes (the multiplicity of bounding boxes comes from logical merging which is discussed shortly) to see if the two partial features physically overlap. This check is relatively inexpensive and avoids the more expensive fine-grained checks for the vast majority of the partial features. If the intersecting bounding box check succeeds, the fine-grained check is performed to see if the partial features are actually intersecting within the coarser bounding boxes. This operation is a set intersection check of the ghosted identifier sets. If these checks fail, we still must check to see if the features are logically intersecting so we perform a set intersection check on the partial feature's periodic nodes. If either the physical or logical checks succeed, then the feature structures are merged. The simplest case of a physical intersection is illustrated in 4.5(a) as A_1 and B_2. A logical intersection case is shown as B_3 and C_3. It is also possible for both checks to succeed when intersections occur "along" a periodic boundary instead of "across" it. This case is shown by D_2 and B_3. These two partial features overlap on both periodic nodes and also physically on ghosted entities. In this case the physical case takes precedence.

If two partial features require merging, each one of the member sets within the feature data structure is merged by performing a set union operation. The type of intersection between partial feature pieces dictates how the bounding box information is stitched. If physical intersection occurs, the intersecting pair of bounding boxes is also stitched. If only logical intersection occurs, then all bounding boxes from both partial features are preserved in the stitched feature. Finally the `_min_entity_id` is updated to contain the lowest identifier of both pieces for sorting purposes.

For efficiency, we actually perform some of these set operations multiple times: once during the intersection check, and then again during the actual merge operation. While this may seem counter-intuitive, it turns out that the C++ standard library creates temporary sets for most set operations including unions, intersections, and differences, which creates significant memory churn. For this reason, a custom set intersection method was developed that can return the Boolean result of a set intersection without creating any temporary data structures in memory.

Algorithm 2 Feature Stitching

```
1: $it1 \leftarrow feature\_sets.begin$
2: while $it1 != feature\_sets.end$ do
3:     $feature1 \leftarrow *it1$
4:     for all $it2$ in $feature\_sets$ do
5:         $feature2 \leftarrow *it2$
6:         if $it1$ != $it2$ and $feature1.intersects(feature2)$ then
7:             $feature2.merge(feature1)$
8:             $feature\_sets.push\_back(feature2)$
9:             $feature\_sets.erase(it2)$
10:            $old\_it \leftarrow it1$
11:            $it1 \leftarrow it1 + 1$
12:            $feature\_sets.erase(old\_i1)$
13:        else
14:            $it1 \leftarrow it1 + 1$
15:        end if
16:    end for
17: end while
```

Algorithm 3 Feature Intersection

```
1: if $a.var\_idx = b.var\_idx$ and
        $setsIntersect(a.periodic\_entities, b.periodic\_entities)$ or
        $(boundaryBoxesIntersect(a.bboxes, b.bboxes)$ and
        $setsIntersect(a.ghosted\_entities, b.ghosted\_entities))$ then
2:     return true
3: else
4:     return false
5: end if
```

After a merge occurs, the two partial features are removed from the partial feature list and the newly merged feature is appended to the end of the list. This action ensures that all list entries will be included in a subsequent comparison ensuring that all possible merges are performed at the termination of a double loop. Consider the pathological case of a long spiral feature in a 2D domain or a helical feature in a 3D domain. If the all of the partial segments in the list occur in reverse order relative to the iteration direction, then a full traversal of the segments yields only a single intersection. If the newly merged piece is left in place, some possible merges will be missed in this scenario and similar scenarios requiring additional passes over the data structure.

The runtime of the stitching algorithm has been analyzed extensively and its implementation profiled as it is one of the most performance critical regions of the algorithm. The upper bound performance of this algorithm is $\mathcal{O}(n^2)$, where n is the number of partial features. The lower bound however is $\Omega(n)$ when every comparison results in a stitch (which can occur if the entire domain is a single feature). In real simulations however, there are often several spatial variables used to represent features that are in close proximity to one and other so that they can be identified and tracked separately. If m variables are used to represent n partial features evenly, then the runtime of the algorithm can be bounded by Eq. 4.2:

$$\mathcal{O}\left(\left(\frac{n}{m}\right)^2\right). \tag{4.2}$$

While this algorithm is still quadratic, the practical numbers for real feature based simulations are very manageable and far from the asymptotic limits for scaling purposes. Our very largest simulations contain only thousands of features and require a few dozen variables to represent them spatially. There are alternative implementations that could result in a better asymptotic growth rate for larger numbers, such as the tree based communication method in [16]. This communication method may be considered for future enhancements.

Spatial partitioning algorithms including KD-trees [9] and Bounding Volume Hierarchies [52], have been explored to further reduce the rum time cost of this algorithm. While these data structures certainly would reduce the number of required comparisons the cost of construction and memory overhead may reduce the effectiveness of these techniques since each data structure is only queried once before needing to be rebuilt by subsequent invocations of this algorithm. The largest model executed to date (§7.3.4) contained only 6000 grains and required 30 order parameters to successfully run. These numbers are still relatively small for the largest models. Furthermore this algorithm consumed well under 10% of the total runtime making it a poor candidate for further optimization [2].

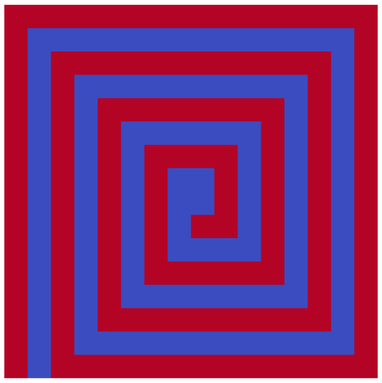
Several test cases were developed to ensure proper functionality of the stitching algorithm. These test cases are illustrated in figure 4.7. Each of these test cases was run on several different numbers of processors (domain partitionings). These images reflect the 16 processor cases (4x4 tiled partition). Figure 4.7(a) shows a single global spiral feature. The 4x4 partitioning splits this feature into a total of 36 pieces that must be re-assembled into a single feature. Each processor initially sees multiple features that are not mergeable since each processor owns different partial disconnected strips of the spiral. The ghost elements of the spiral are illustrated in figure 4.7(b). These are the identifying elements of the partial features gathered on the root processor for stitching. Figures 4.7(c) and 4.7(d) show concentric but non-intersecting boxes and their associated ghosting pattern on a 4x4 tiled partition. This case ensures that the bounding box intersection checks are exercised while making sure that non-overlapping regions are not merged. Figures 4.7(e) and 4.7(f) show a series of nested "l-shapes". This pattern is similar to the boxes in that the features do not overlap but many of the bounding boxes do intersect.

4.3.3 *Multiple Flooding Thresholds*

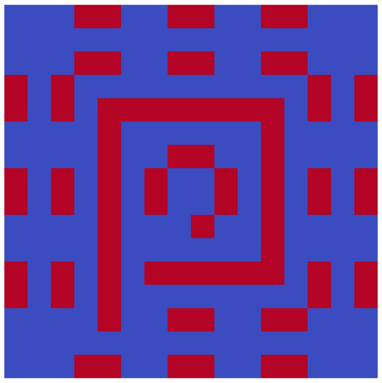
In §4.2.1 we discuss locating and flooding features by sampling the solution field on each mesh entity and comparing it to a threshold value. In this section we talk about the addition of a second threshold to improve the robustness of several of the algorithms in this work. We introduce the concept of a "starting threshold" and a "connecting threshold" to create a range where solution values may fall without triggering the start of new features or stopping the flooding for the identification of an existing feature. This dual threshold approach removes the jitter or numerical noise which can cause failure of a simulation. This idea is similar to concepts found in signal analysis for filtering [79].

The multiple flooding threshold approach has another application as well. In simulations involving grain nucleation, such as those involving recrystallization [13, 78], there are many small features being formed and merged into one and other while they are still very small. If we wish to ignore these nucleating features while simultaneously tracking and remapping our full-sized features, we require another mechanism. Using a very high starting threshold that only larger features can obtain while leaving the connecting threshold very low is one approach to solving this issue. At the time of publication, this approach is being investigated with promising results.

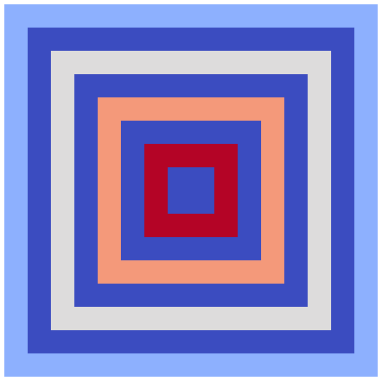
Unfortunately, the multiple flooding threshold approach creates another difficulty for the distributed feature extraction algorithms. It's highly likely with increasing numbers of features and processors that a single feature's high entities may be separated



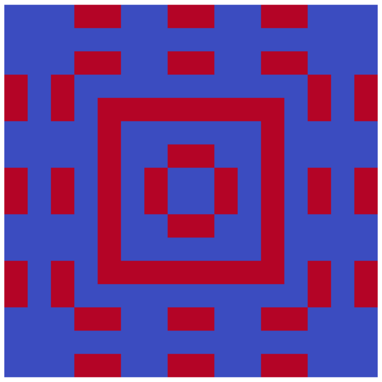
(a)



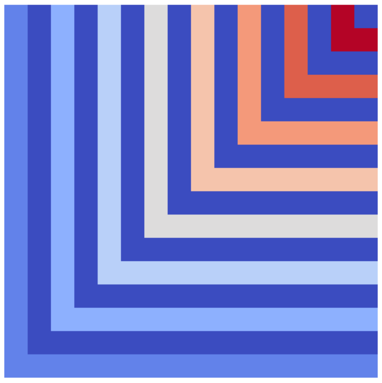
(b)



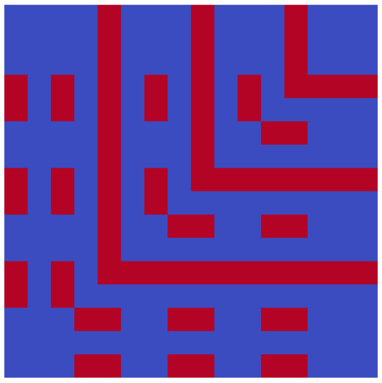
(c)



(d)



(e)



(f)

FIGURE 4.7: Feature identification test cases and associated 4x4 ghosting patterns: Figure (a) shows a single spiral which appears as multiple features in each partition (b) until merged. Figures (c), (d) show concentric boxes where bounding boxes intersect or are contained within one and other. Figures (e), (f) show disconnected features with intersecting bounding boxes.

from the complete set of low entities by a processor partition. To solve this problem, the flood fill algorithm uses the `STATUS` flag introduced in §4.2.2 to keep track of whether or not each partial feature on each processor contains a starting threshold value. Partial features that do not have any high threshold values are marked as "inactive" but maintained and communicated to the root processor for merging. If there are still no starting threshold entities after a feature has been fully merged, it may be discarded.

4.3.4 *Updating Non Root Ranks*

After the feature stitching operation has been performed, the master rank contains a complete global map of all features along with their corresponding partition independent IDs. However, all of the remaining ranks in the parallel run still have only their partial feature information that was last updated just before the serialization routine (§4.2.4 and §4.2.5). Fortunately, each of the fully merged features on the root processor contain a data structure with all of the original processor IDs and corresponding local IDs ("original IDs") that make up any portion of the fully merged feature. The root processor iterates over each of these lists in each of the features and constructs a stacked "local to global" feature vector to be distributed to the remaining ranks. This vector is sized as the total number of partial features in the entire simulation. For each feature, the stacked index is updated with the current feature's ID. The stacked index s, is calculated as follows:

$$s = \sum_{k=0}^{orig_j.p-1} |features_k| + orig_j.l \tag{4.3}$$

$$global\_features[s] = feature_i.id \tag{4.4}$$

where i is used to index over the global features, j indexes over each feature's original IDs, $|features_k|$ represents the total number of partial features on the k^{th} processor, $orig_j$ represents the current original ID containing both a local ID, $orig_j.l$ and processor ID,

$orig_j.p$. After this vector is built, an MPI "scatter" operation is performed to send a slice of that vector to the remaining ranks. Each non-root processor then updates its local partial feature information with the correct global ID. Additionally, an auxiliary "global to local" map is constructed for handling queries where the global ID is known. Note that we use an associative map as opposed to a vector for the "global to local" map because there may be significant breaks in global feature indices.

4.3.5 *Updating Reportable Data*

After the feature maps are completely constructed on each processor, additional post processing is performed to update auxiliary data structures that are queryable from the user API. The features as they are constructed in §4.3.2 are not well suited for algorithms that need feature information at a specific node or element in the mesh. Auxiliary data structures are constructed by looping over all of the various sets in each feature and creating maps, keyed by entity ID. Note that several features may overlap at a given location in the domain if several variables are coupled into the `FeatureFloodCount` Postprocessor. This information may also change quite significantly depending on the chosen flooding threshold §4.2.1. Rather than storing several feature values for a given entity in the mesh, only the feature with the highest variable value is chosen instead. This practice removes any bias that could occur by processing the various variables in a specific order creating a case where some variables where over-represented (e.g. last value processed wins).

4.4 The GrainTracker Postprocessor

In this section we begin using the terminology "grain" instead of "feature". A grain (§2.1) is a specific type of feature. Whenever you see the term grain in the remainder of this chapter, you may envision it as a feature. Similarly, we begin using the terminology "order parameter" instead of "variable". An order parameter is simply the name given to variables that represent non-conserved quantities in the Allen-Cahn model (§2.1). The `GrainTracker` Postprocessor is what makes accurate reduced order parameter (ROP) modeling (§2.1.1) possible. There is only a single requirement in making this occur: no two grains represented by the same order parameter may ever come into contact as the simulation evolves. Generally however, it is also useful to maintain unique grain identifiers that are fixed over the lifetime of the simulation, which can be used to query individual grain properties such as orientation. The work to identify grains is completely handled by the `FeatureFloodCount` Postprocessor (§4.2). There are however several additional functions that need to be performed to track and prevent the merging of grains over the course of a simulation. The process flow of the grain tracker is summarized in figure

4.4.1 *Grain Tracking*

In order to support the ability to query grain information throughout a simulation, a unique and unchanging identifier must be assigned to each grain. Unless external data is being used where identifiers may already be pre-chosen, the choice of identifiers for grains is insignificant. However, the assignment of identifiers should be invariant of domain partitioning or the number of processors used in a simulation. This property aids in comparing simulations performed using different numbers of processors. The `GrainTracker` Postprocessor supports each of these requirements. If the grain structure is supplied from an external data source such as "Electron Back-Scatter Data" (EBSD), the `GrainTracker` will simply query the external data object for grain identifier information once the grains are extracted and merged. When generating an initial condition, the grain identifiers are assigned in groups based on the order parameter variable numbers. The grains represented by each order parameter are sorted by the `_min_entity_id` stored in their respective data structures (1). Since grains may over-

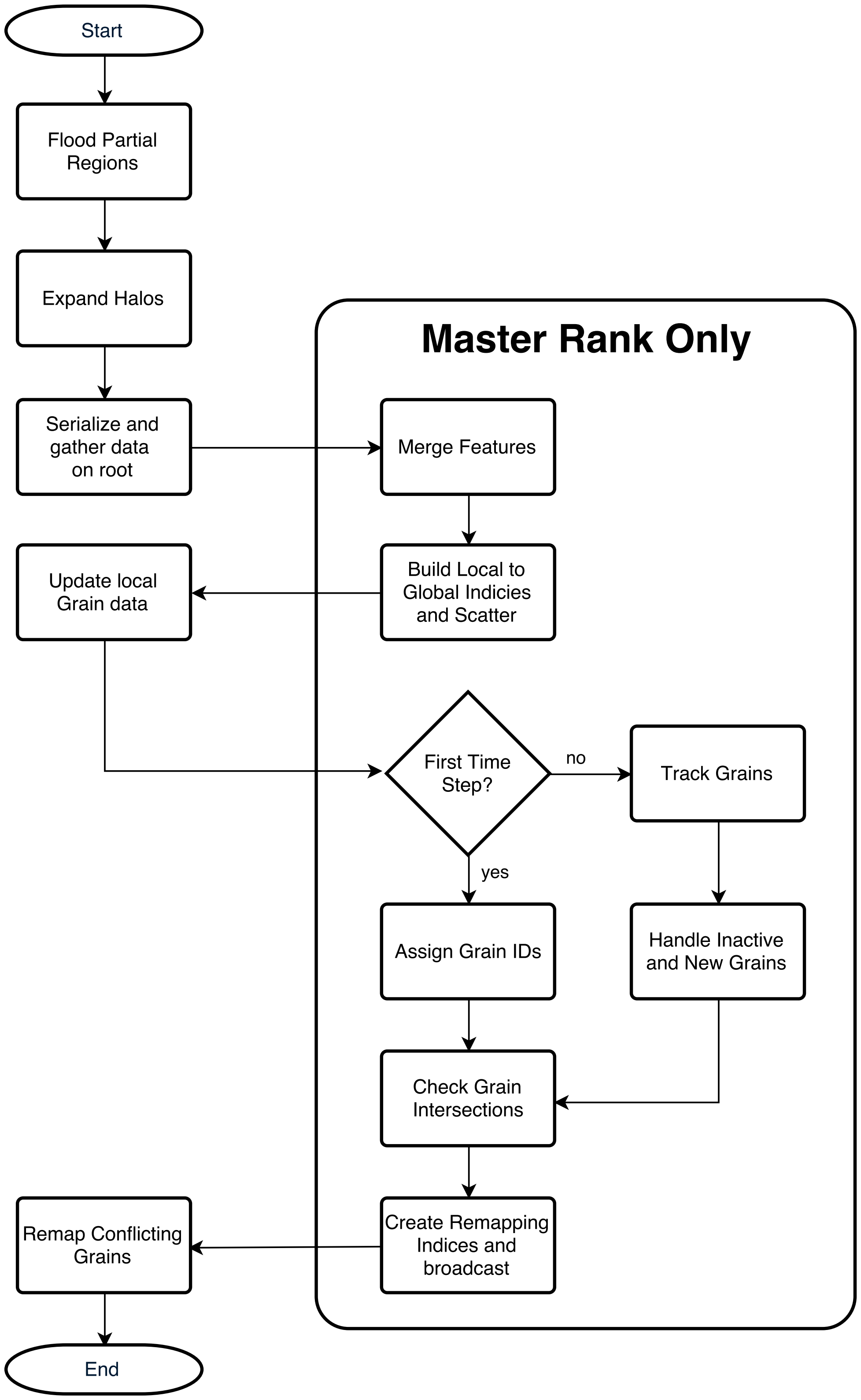

Start
Flood Partial Regions
Expand Halos
Serialize and gather data on root
Update local Grain data
Master Rank Only
Merge Features
Build Local to Global Indicies and Scatter
First Time Step?
no
Track Grains
yes
Assign Grain IDs
Handle Inactive and New Grains
Check Grain Intersections
Create Remapping Indices and broadcast
Remap Conflicting Grains
End

FIGURE 4.8: Grain Tracker Flow Chart: The actions on the right are performed only on the master MPI process to save memory.

lap, it's possible for multiple grains to have the same `_min_entity_id`. However, this cannot occur for grains represented by the same order parameter so the sorting is guaranteed to be deterministic.

After the initial assignment is made, it is vitally important that each grain maintains its original identifier for the duration of the simulation whether that grain grows, shrinks, migrates through the domain, or even fractures. This is accomplished by a careful comparison and analysis of each pair of time steps as the simulation progresses to identify each grain over its lifetime. The general strategy to tracking grains is to create a mapping that minimizes global grain movement. Any mapping created must not permit any jumps or large translations of individual grains. The global mapping must also support and correctly handle the movement of grains across periodic boundaries while maintaining global minimal grain movements.

The tracking stage begins on the `GrainTracker`'s second "active" time step. Active here is used to indicate that the `GrainTracker` need not begin tracking at the beginning of the simulation. Like any other MOOSE Object, it may be turned on or activated at some point after the simulation has started rather than being active at the start of the simulation. This capability is useful for certain types of initial conditions that need to evolve over a period of several steps such as those observed in the `PolycrystalRandomIC` object (an intial condition which evolves a grain structure from random noise). The grain structure from the previous time step is compared to that of the current time step using a movement minimization algorithm to construct a "new to existing" grain mapping (Algorithm 4).

There are several special cases that need to be handled during the construction of the new to existing grain map to ensure proper model integrity under different simulation scenarios. Grains may appear (nucleate), disappear (be absorbed), split or join together (rapidly coalesce). While detecting single events is straightforward as will be discussed next, there are pathological combinations of events that can be very

Algorithm 4 Construct New To Existing Grain Map

```
for each $grain$ in $unique\_grains$ do
    $min$ ←MAX
    $min\_grain$ ←NULL
    for each $new\_grain$ in $new\_grains$ do
        if grain.var_idx = new_grain.var_idx then
            $curr \leftarrow distance(grain.\_bboxes, new\_grain.\_bboxes, true)$
            if $curr < min$ then
                $min \leftarrow curr$
                $min\_grain \leftarrow new\_grain$
            end if
        end if
    end for
    $new\_to\_existing[new\_grain].push\_back(grain)$
end for
```

difficult to detect. Consider the case where a single grain appears while another grain disappears in a single time step. Similarly, consider the case where a grain may split while another grain disappears leaving the overall count of grains unchanged.

For the purpose of discussing the grain tracking algorithm, we'll call our grain data structure from the previous time step "existing", and the grain data structure from the current time step "new". To construct the new to existing map we loop over the existing grains looking for best matches by distance in the new data structure. We take advantage of the fact that the new grain data structure is constructed (sorted) by variable index to reduce the number of comparisons the algorithm must make. We only need to consider matching up grains with matching variable indices. The distance routine in Line 6 is shown in Algorithm 5, but for the purposes of this discussion can be assumed to return the centroid distance between its two grain arguments. As we loop over the new grains, we make sure that its bounding box overlaps with the existing grain to participate in the distance comparisons. This inexpensive check helps reduce the chances of hitting a pathological edge case previewed earlier. Before saving the "new to existing" pair to the map, we make sure that there isn't already an existing key (duplicate "new" grain). This case sometimes occurs when a grain disappears in the new time step and there is no best match for the existing grain. The closer of the two grains to the old grain is chosen as the match and the other grain is marked inactive.

When the loop iteration over the existing grains has finished, the new to existing data structure will contain the best mapping for most simulated grains. The grain IDs for this subset are propagated from the existing grains to the new grains. After the unique IDs have been transferred, there are two cases left to handle:

1. Any grain in the new set with an unset status is a new grain. Since we are matching up grains by looking at the existing set and finding the closet matches in the new set, any grain in the new set that isn't matched up has to be new since every other existing grain found a better match in the new set.

2. Any grain in the existing set that has an unset status is inactive. We can only fall into this case when the very last grain on a given variable disappears during the current time step. In that case we never have a matching variable index in the comparison loop so this grain never competes for any new grains leaving its status unset (i.e. it doesn't even compete for an incorrect match).

The new grain case requires additional special handling since there are multiple ways for new grains to be detected. The `GrainTracker` is currently designed to handles two types of new grain discoveries: splitting and nucleating. Splitting grains are grains that break apart or fracture in a time step. This may happen if a grain is exposed to high mechanical stress. Nucleating grains are grains that are created spontaneously in a simulation generally in a region of unstructured atoms. These phenomena occur under a wide range of conditions ranging from mechanical fatigue to radiation damage. The criteria for differentiating between these two conditions is summarized as follows:

1. Splitting Grain: A new grain that is unmatched by any existing grain but has an overlapping halo with another grain on the same variable index.

2. Nucleating Grain: A new grain whose halo does not intersect with any other grain on the same variable.

These new grain cases rely heavily on the integrity of halos from step to step to differentiate between them. The fact that halos are also used by the intersection and remapping algorithms discussed in §4.4.3, and §4.4.4 require that all halos are out of contact at the end of each time step to ensure simulation integrity.

Algorithm 5 Grain Distance Algorithm

```
1: procedure DISTANCE($bboxes_1, bboxes_2, use\_centroids\_only$)
2:     $min \leftarrow$ MAX
3:     for each $bbox_1$ in $bboxes_1$ do
4:         $centroid_1 \leftarrow (bbox_1.max() + bbox_1.min())/2.0$
5:         for each $bbox_2$ in $bboxes_2$ do
6:             $centroid_2 \leftarrow (bbox_2.max() + bbox_2.min())/2.0$
7:             $curr \leftarrow minPeriodicDistance(centroid_1, centroid_2)$
8:             if $curr < min$ then
9:                 $min \leftarrow curr$
10:            end if
11:        end for
12:    end for
13:    return $min$
14: end procedure
```

4.4.2 *Handling Nucleation and Splitting*

As a simulation evolves, the grain tracker will encounter situations where there are mismatches between the number of grains identified in the current and previous steps. When the overall number decreases, this indicates the absorption or "joining" of grains. When this number increases this indicates the presence of a new grain typically through a nucleation process. It can also happen in simulations where nucleation is not being simulated due to grain "splitting". Both of these cases are illustrated in figures 4.9 and 4.10.

Figure 4.9 shows a case where a concave grain splits during a typical grain growth simulation. While grains do not normally break or split physically during evolution, numerically these cases are somewhat common for some grain shapes. The rounded grain in figure 4.9 will typically shrink under normal circumstances. As the size of the grain decreases, the thinning neck in the center of the grain will eventually disappear leaving two distinct features in place of the larger feature. If we refer back to the previous section on grain tracking, we can see that this case may be handled by checking to see if the halo of this new feature intersects the halo of any existing feature with a matching variable index. There are additional highly unlikely cases possible that are not currently implemented. These cases are discusses in §8.1.

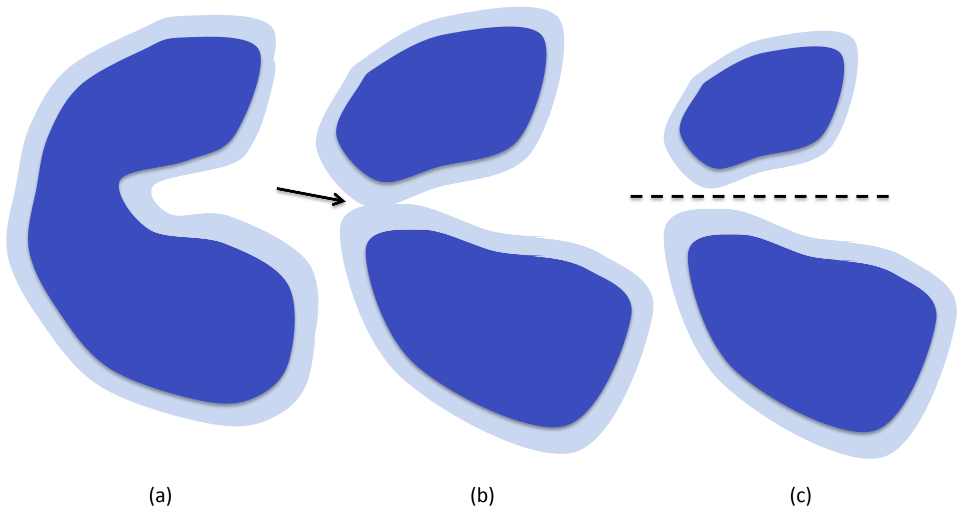


FIGURE 4.9: A possible progression of the absorption of a non-convex grain. The grain halo is shaded around the grain body. Figure (a) shows the original shape of the grain as it begins to be absorbed. Figure (b) shows the time step where the neck of the grain becomes narrow enough that it eventually causes separation of the two larger parts of the grain body. Figure (c) shows a complete separation of the grain and halo regions. Note the general curved shape of the grain. This is typical for a grain that is being absorbed by neighboring grains.

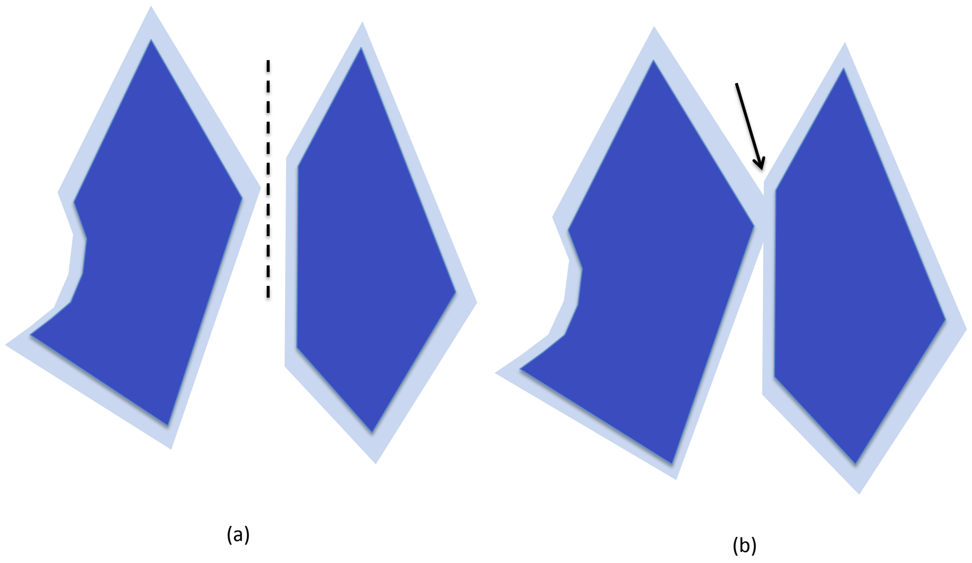


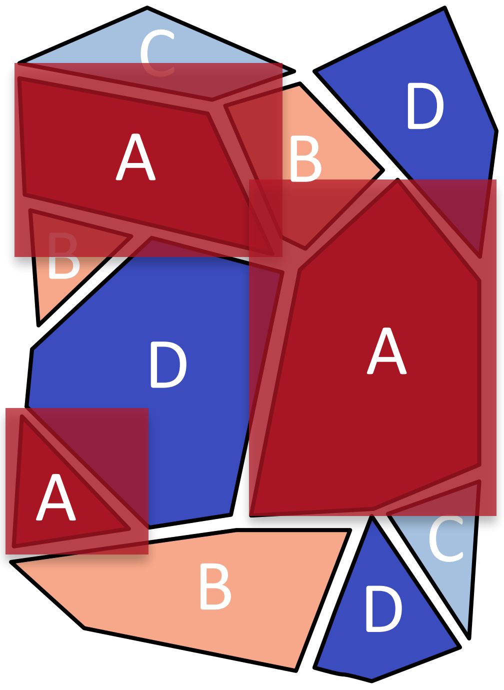
FIGURE 4.10: A possible progression of the coalescence of a “split grain”. Figure (a) shows two separate grains each with non-overlapping halo regions. Figure (b) shows the initial contact of the halo regions of these grains. Grains sharing the same grain ID are remapped to the same order parameter if they are not already represented by a single variable in a given time step. They are allowed to coalesce if they come into contact.

Figure 4.10 shows a case where grains are beginning to join. Normally as two grains represented by the same order parameter begin to come into contact (indicated by touching halos), one of them is remapped to another order parameter. This algorithm is discussed thoroughly in §4.4.4. However, if the two grains have the same ID, they are allowed to coalesce into a single larger grain. It is important to note that this is generally an artificial case that the grain tracker must handle due to the reduction from a real 3D data set to a 2D simulation.

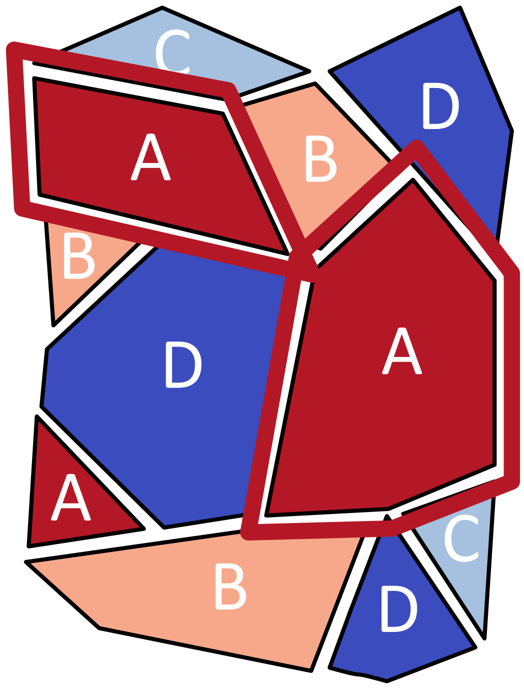
4.4.3 *Grain Intersection Checks*

After the grain tracking stage has completed, the updated grain information can be immediately used to report unique grain identifiers and grain statistics. However due to grain evolution, additional work may be needed to prevent grain coalesce due to reduce order parameter modeling discussed in §4.4. The remapping algorithm assumes that the order parameters in the previous time step are assigned in a valid configuration. It is then reasonable to assume that most of the grain's order parameters are still in a valid configuration in the current time step. Rather than recoloring the whole grain map, we only need to handle new grain neighbors that are in conflict.

The grain integrity check stage begins by looping over all grain pairs with matching variable indices looking for conflicts. Grains that are represented by different order parameters are by definition never in conflict so those comparisons are immediately skipped. For grains that are represented by the same order parameter, a coarse check of the grains' bounding boxes is made first to see if the grains may be in contact. If any of either grains' bounding boxes are overlapping as shown in Fig 4.11(a), then a fine level check is performed to see if there is a set intersection of the two grain's halo sets as shown in Fig 4.11(b). If an intersection occurs, this indicates that two grains have become neighbors from a graph coloring point of view so grain remapping must occur.



(a)



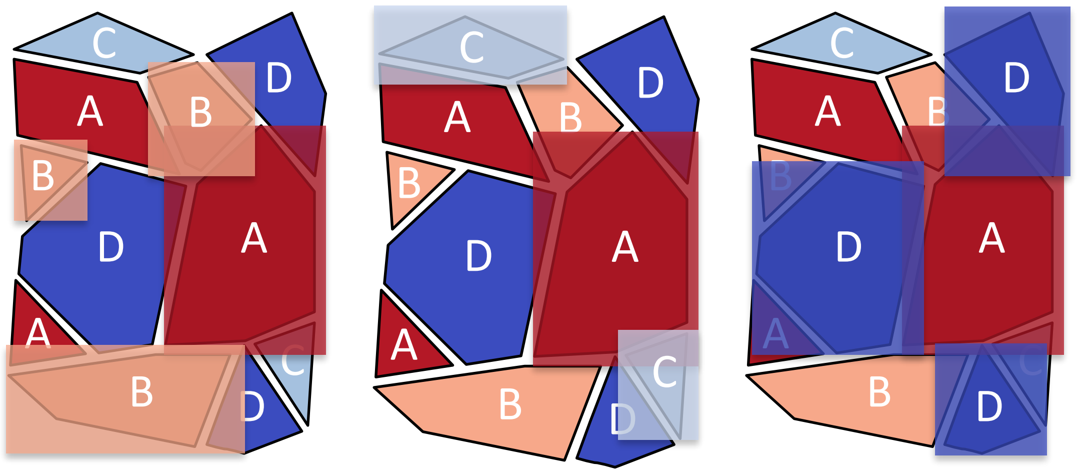
(b)

FIGURE 4.11: Illustration showing two features in close proximity triggering a remapping operation. Figure (a) shows two features with intersecting bounding boxes (coarse intersection check). This triggers a fine-grained "halo" check illustrated in (b).

4.4.4 *Grain Remapping*

Grain remapping is implemented using a recursive backtracking algorithm capable of performing several variable swaps to transform the improperly colored grain graph into a proper one. This backtracking algorithm runs only on the root process which is the only processor that contains the complete global grain graph. There are several stages and utilities that make up this portion of the algorithm which are discussed here. When a pair of grains are located that are in close proximity defined by the fine intersection check covered in the previous section, one of them is arbitrarily chosen and designated as the "target" grain indicating that we seek to remap its defining variable values to a different solution variable. Depending on the number of neighbors a graph has and the variables representing each of those neighbors, it may or may not be possible to create a valid graph by remapping only the target grain. In this case a depth-limited, depth-first search is performed seeking a series of neighbor swaps to leave the graph in a valid state.

To begin, an array of lists of size m is built and populated, where m is the number of variables (colors) in use. For each variable the nearest grain represented by that variable (as determined by the bounding box distance) is located and its distance is stored in the list at the corresponding array position along with the grain ID itself. In cases where the nearest bounding boxes for a given variable overlap the target grain, we maintain a negative count representing the total number of overlaps and the ID of each grain which overlaps. Otherwise we store the closest bounding box edge to bounding box edge distance for the given variable. We don't bother to calculate or store any information for grains with matching variable indices, or for grains that live on a reserved order parameter (§6.1) since those variables are ineligible for remapping. If there are any empty order parameters (an order parameter representing zero grains), a distance of infinity (∞) is entered into the corresponding position prioritizing those variables for remapping. This "color distance" array is then sorted in reverse order putting the grains furthest away near the front and leaving those with several overlaps near the back. A case with all negative distances is illustrated in figure 4.12. In this example, the target grain is chosen as the large grain labeled A, centered on the right side of the image. All of the other colors have at least one bounding box that overlaps the large A grain: 2 for B, 1 for C, and 3 for D. The empty list ($\varnothing$) is used for the variable represented by the target grain to ensure that the same variable is never considered as a possible remapping option.

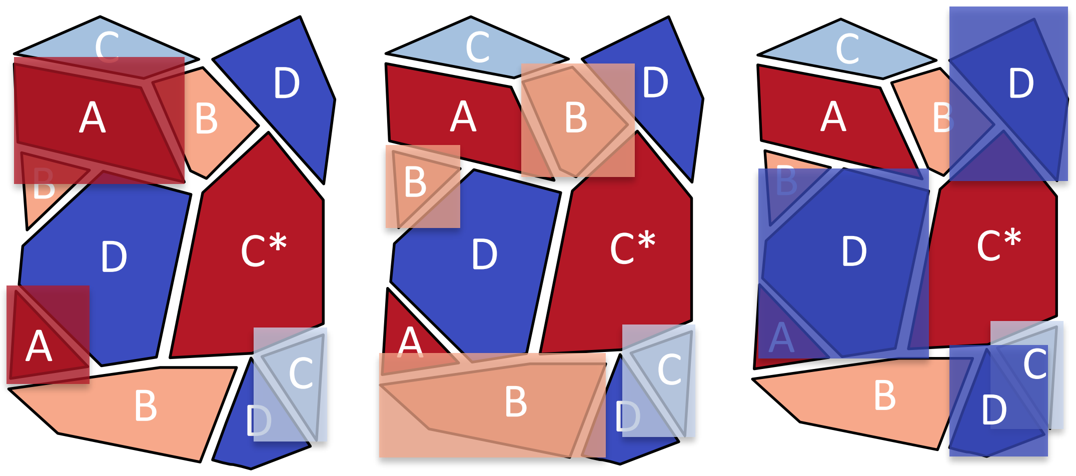


(a) (b) (c)

Variable	Distance
A	$\varnothing$
B	-2.0
C	-1.0
D	-3.0

FIGURE 4.12: Illustration showing the grain distance heuristic being calculated for all variables (colors) in the simulation for the target "A" grain located on the right-hand side. Figures (a), (b), and (c) show that all available variables in this simulation are potentially neighbors (overlapping bounding boxes) with the current target. The heuristic function penalizes each variable based on the number of overlaps by maintaining an increasingly more negative value for each grain with one or more overlapping bounding boxes. $\varnothing$ is used to prevent variables from attempting to remap to themselves.

We iterate over the array of distances looking for available variables suitable for remapping the target grain. If a positive value is encountered, the grain can be imme-



Variable	Distance
A	52.6
B	4.2
C	∅
D	-1.0

FIGURE 4.13: Illustration showing the grain distance heuristic being calculated for all variables (colors) in the simulation for the target "C" grain located in the lower right corner. Figure (c) shows that one variable is still a potential neighbor. Figures (a) and (b) however give us options for immediate remapping. The order parameter with the larger positive distance is selected since it is further away.

diately remapped and the algorithm returns "success". If however a negative value is encountered, we must first perform a fine-level check on each of the corresponding grain halos to see if these grains actually overlap. If they do not, we can immediately remap the target grain and return "success". If we encounter a case where there is only a single truly overlapping grain (bounding boxes and halos intersect), the algorithm tentatively marks the target grain with the other grain's variable effectively simulating a remapping operation. It then recurses on the other neighboring grain making it the new target. If the algorithm is able to find a successful remap in the recursive call, the returned "success" value indicates to the caller that the tentative mark can be removed. The "success" value can then be propagated on up the call stack. If all items in the "color distance" array are exhausted without finding a successful swap or set of swaps, the algorithm returns "fail". If we are in a recursive call, the tentative mark is removed and the next value in the array is inspected. We find that limiting the depth-first search to a relatively small depth (2 or 3) works reasonably well to fail out of impossibly tightly colored graphs faster. This also helps avoid the huge runtime penalty and exponential growth rate possible with an unlimited backtracking algorithm. Note: Tentative markings are indicated by turning on the `DIRTY` status flag in the feature's data structure. The `DIRTY` status uses an independent bit so it can exist simultaneously with another status.

4.4.5 *Remapping the Finite Element Solution*

As previously noted, The grain remapping algorithm detailed in the previous section runs only on the root processor since it is the only processor with the complete global map of grain structure information. This represents a significant challenge since solution variable information is typically distributed for scalability in many finite element frameworks. To overcome this challenge, instead of performing the solution remapping operation during the remapping algorithm, the root processor maintains

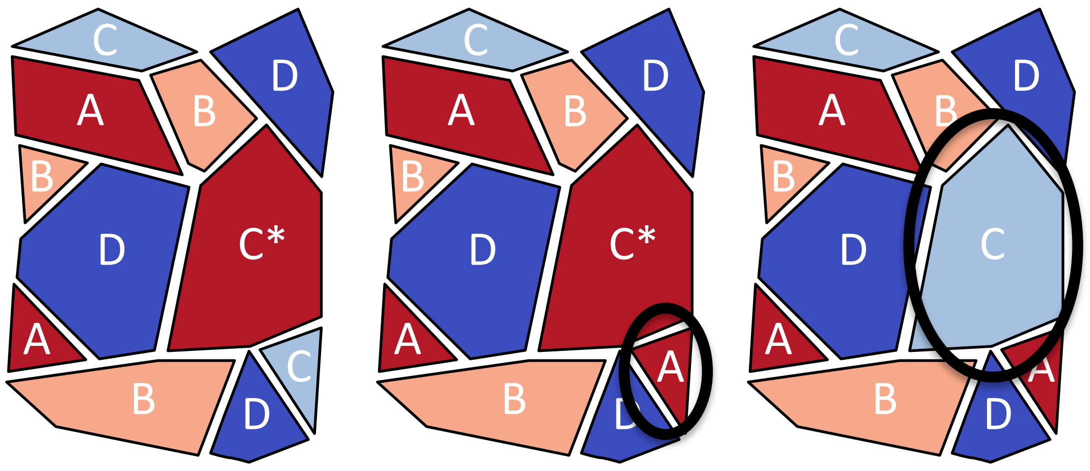


FIGURE 4.14: Illustration showing the final remapping stages. Figure (a) shows the grain colors after the table in Figure 4.13 is built prior to any remaps. The presence of a positive distance means that we can immediately remap to one of those colors. We choose A, the largest distance and remap (Figure (b)). As the algorithm backs out of recursive call, it remaps the tentatively marked grain to C (Figure (c)).

a list of grain remapping operations that need to be performed. This information is broadcast to all of the remaining ranks after the remapping algorithm has completed. All of the ranks (including the root process) then iterate over this map and use the remapping information contained within to update their individual local feature structures and corresponding solution values.

We abstract the solution value swapping algorithm into a little helper routine described in §4.4.5. The actual solution value swapping must be performed in two passes to handle any neighboring feature swaps (i.e. neighboring vertices that exchange colors). This case is illustrated in figure 4.15. This is due to the fact that a single element in the finite element will have variable values for all field variables in the simulation and features are permitted to overlap in the phase-field model. During the first pass, each rank that owns any portion of any of the remapped feature uses its current variable index and local IDs to look-up corresponding degree of freedom (DOF) indices. These DOF indices are then used to look-up the variable values in

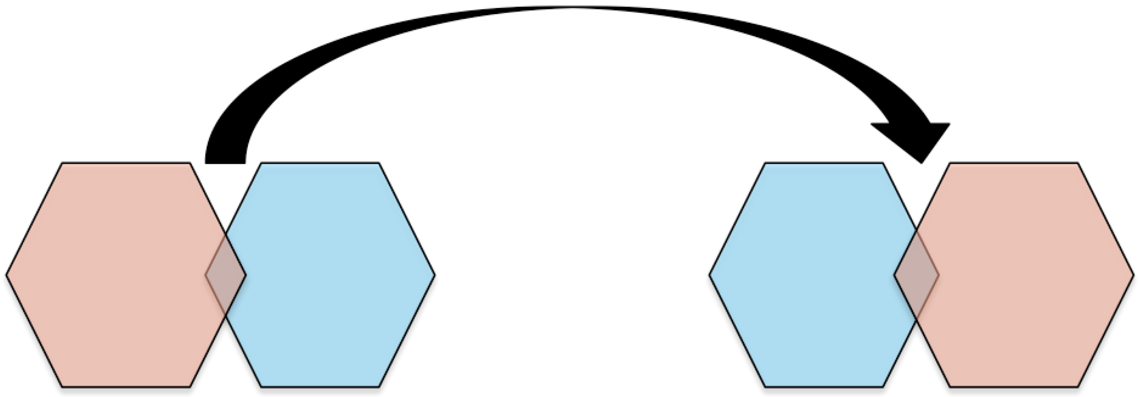


FIGURE 4.15: Solution remapping on neighboring features: The overlapping region between features requires that the solution information is read and cached for all variables before being re-written.

the solution vector. That information is cached in a local data structure. During the second pass, the new variable index (color) is used to look-up the new DOF indices where that cached information should be written. The cached data is copied into the corresponding positions in the processor's local solution vector. Finally the old variable values are zeroed out effectively completing the transfers.

The remapping routine interacts with the finite element solution vectors. Since the finite element method defines a solution based on field of global basis functions, the solution modifications must be made precisely to preserve the integrity of the simulation. The purpose of the grain tracker narrows the scope of what kinds of moves are performed. We are only ever moving the values of order parameter fields within a phase-field simulation and the properties and characteristics of these variables are well-understood. First order parameters are non-conversed, greatly simplifying the rigor needed in copying values around in the solution vector. Second, we are always moving non-zero values from one order parameter (representing a grain) to any other order parameter where there is a region containing only zeros in the target area. The target area must completely encompass the volume of the grain being moved so that no values are lost when the transfers are made. Finally, to preserve the integrity of the simulation, any corresponding degrees of freedom in solution vectors from previous time steps (known as "old" and "older" in the MOOSE framework) must also be updated when the current solution vector is changed. This is because many physics simulations use information from previous time steps when calculating current residual values. An astute observer may note that the graph coloring properties of the previous solution vectors may not hold if information from the current time step is propagated backwards in this manner. This is not a problem in practice since graph and grain information is not used in calculating current residuals.

SOLUTION VALUE CACHING — The solution value rewrites that take place following the remapping algorithm are handled by a cache aware utility for convenience. This utility has three modes of operation: `FILL`, `USE`, and `BYPASS`. The `BYPASS` mode is used for performing live updates on the solution vector. The cache isn't filled or used in this case. The values are simply transferred from the source variable to the target variable. It turns out that we can only use this method in single swap conditions as an optimization. The other two modes for working with the solution vector are used in concert. We first use the `FILL` mode, meaning that we read information from the target variable into a cache indexed on the source variable. If we use the USE mode, we read information out of the cache for the target variable into the DOFs on the solution vector corresponding to the destination variable.

4.4.6 *Updating Variable Field Information*

After the remapping operation is complete, the `GrainTracker` Postprocessor needs to update and create several data structures that can be queried by the remaining portion of the simulation. These data structures provide unique IDs where they are needed and are used to produce fields for visualization purposes. Multiple maps that are indexed by entity IDs are created to produce the unique IDs, current order parameter assignments, halo fields (§4.2.3), and ghosted entities. Since the `FeatureFloodCount` Postprocessor produces overlapping grain fields, the highest value at an entity always determines the information displayed at that entity. This removes any bias that may arise from looping over entity information in each of the grains and having either the first or last value set determine the value at an entity.

This section is similar to the functionality provided by the similarly named section in the `FeatureFloodCount` discussion 4.3.5. Its underlying purpose is indeed the same but there are several additional attributes needed by grains that are not needed by more generic features.

4.5 Feature Statistics

The ability of the `FeatureFloodCount` and `GrainTracker` objects to produce information about individual features throughout a given domain can be used to provide statistical information for further post-processing analyzes. The `FeatureFloodCount` object can provide the total number of features observed as well as the number of features per solution variable. Several pieces of information can be obtained for each identified feature: the centroid, the bounding boxes, the number of elements making up the volume or the halo around the feature and a boundary intersecting indicator. In addition to the physical statistics available, several simulation related statics can be retrieved: The list of processors owning a piece of feature, a partition independent id useful for stable sorting when running the same simulation over different numbers of processors, and a status variable indicating the various states a feature may be in during the simulation such as active, inactive, marked, and others.

One notable piece of information that is not directly available in these objects is the actual computed volume of each feature. This value is not normally calculated during a typical simulation because it is slightly more expensive to calculate since it requires additional “shape function” evaluation and quadrature point loop to obtain the integral values. Even if these values were readily available, neither one of these objects would have a natural ability to output this information in any useful way since the values are neither spatial or scalar. Fortunately, the MOOSE framework has a vector post-processing system (§4.1.3) for managing output of arbitrary "vector" data such as these statistical value arrays.

4.5.1 *Feature Volume Vector Postprocessor*

The `FeatureVolumeVectorPostprocessor` is responsible for outputting vectors of data from the `FeatureFloodCount` object that are not directly related to the mesh. These include lists of feature volumes, corresponding variable indices, and other relevant feature information. Creating several independent objects to implement the algorithms in this dissertation limits the scope of each object, allowing for better software design and maintenance. `VectorPostprocessors` discussed in §4.1.3 handles the output of information which is often error prone and challenging in parallel computing environments.

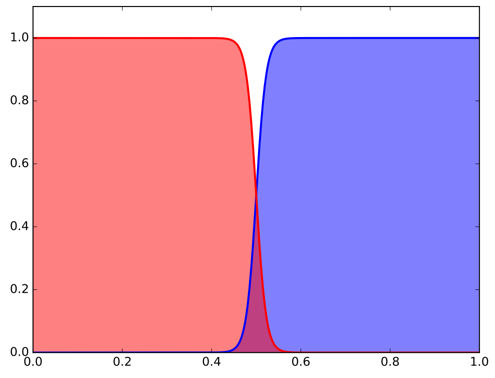


FIGURE 4.16: 1D view of the intersection between two order parameters (η_i) from a phase-field simulation. The volume integral calculation of each "grain" is shaded.

As noted in §4.5, neither the `FeatureFloodCount` nor the `GrainTracker` objects calculate the feature volumes beyond maintaining a list of elements where the feature is present. This object couples to either the `FeatureFloodCount` or `GrainTracker` objects. As the vector post-processor is performing an elemental loop, it is able to retrieve an array of all active variables to the corresponding feature that they are representing for the given element. As the integral value is calculated on every element, the appropriate volume contributions are summed into the corresponding feature's volume. This procedure allows for accurate volume calculations based on the integrals of the calculated variable values. This capability is important in the phase-field method where the interface between adjacent grains is non-zero as seen in Figure 4.16. In the interface regions, multiple variables will be non-zero but whose sum is approximately equal to one.

4.6 MOOSE Core enhancements

MOOSE contains several systems where the execution frequency is user-controlled. This concept is briefly introduced in the §4.1.1. This is useful in reducing expensive computations where calculated fields are only used for things like output instead of feeding back into a coupled solution computation. The algorithms presented in this dissertation generally only need to execute once per solve to inspect and impact the converged solution. This is accomplished by setting the `execute_on` flag for the target objects all to the same value such as `TIMESTEP_END`. However, this doesn't specify the relative ordering among objects from different systems such as `Postprocessors` and `AuxKernels`. During the development of the feature identification and grain tracking capabilities, it became apparent that some amount of inter system dependency resolution would be necessary for the purpose of visualization. The `FeatureFloodCount` and `GrainTracker` objects both create several data structures which can be used to populate a number of auxiliary variable fields (`AuxVariables`). However, post processing is usually the last operation run during a time step so that aggregation can be performed on the field variables. This was indeed an assumption that was designed into the MOOSE framework.

To support the output of information to field variables, MOOSE was enhanced as part of the grain tracking development effort so that user object and post processors could be executed before `AuxKernels`. This enhancement was designed so that the dependency resolution would be automatic based on the data dependency. This capability has become an important part of the MOOSE ecosystem and is continually being expanded to incorporate more ad hoc resolutions in the future.

Chapter 5
POLYCRYSTAL INITIAL CONDITIONS

"People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones."
- Donald Knuth

In order for the GrainTracker to function properly in a simulation, it must begin with valid initial conditions. There are two types of grain structure initial conditions: those that are dictated by real experimental data, and those that are generated by the simulation to mimic real microstructures. In either case, an order parameter assignment for the grain structure must be produced for running a reduced order modeling simulations. To ensure the validity of the simulation, the initial assignment of these order parameters to the grain structure must satisfy the same conditions as those being held during the simulation execution. That is, no two grains represented by same order parameter may be in contact for the same definition of contact. Recall that run-time contact is defined in §4.4.3 as the intersection of halos of different grains represented by the same order parameter.

5.1 Initial Conditions for Nodal FEM Basis Functions

When creating an initial condition for a finite element mesh, one must first choose a suitable set of basis functions to represent the finite element solution on the domain depending on the properties and continuity requirements of the PDE for which a solution is being sought. For many types of FEM analysis, the Lagrange shape function family is chosen for several of its desirable properties: simple fast numerical calculations, C0 continuity, linear interpolary solutions, and nodal values that correspond exactly to the solution at those points [8, 67]. However, when projecting cell-centered or volumetric data, some smoothing or averaging technique must be applied to set the values at the nodal positions between elements with different values. One option is to average all of the elemental values connected to each node.

5.2 Initial Conditions from Experimental Data

Electron Backscatter Data (EBSD) is a very common experimental technique for obtaining crystallographic microstructure and texture data, which is suitable for use as initial conditions and for validation of phase-field simulations [42, 15, 99, 81, 88]. The MOOSE phase-field module has a rich interface for reading and querying EBSD files exported by the de facto standard EBSD software DREAM3D. This EBSD data is assumed to be cell-centered or an "elemental" data format so some projection method must be used to produce initial condition values for nodal basis functions on the finite element mesh. First, we must pick an order parameter to represent each grain identified in the EBSD data set (§5.4 and §5.5). After we have selected an order parameter for a given grain, we must decide how to project the value onto the nodal basis. Consider for example a regular structured 2D rectangular mesh and assume that we have a grain that's represented by a single element in that grid. If we set a value of 1.0 for each of the four node points we will effectively initialize the phase-field with no interface widths anywhere in the domain as opposed to the diffuse interface that the phase-field method normally expects (Figure 4.16). This kind of sharp initial condition has a few disadvantages: First, it creates larger, non-conservative grains throughout the domain since every grain overlaps every other grain on at least one node. It may also cause the first nonlinear FEM system assembled to fail to converge since the sharp interface is a poor initial guess for the diffuse interface solution to the phase-field equations.

A better initial condition approximation is to set the nodal value for each variable to the average of all neighboring elemental values for a given variable. This approach creates a linear profile spreading away from each grain. Since every elemental value is either zero or one by definition, this approach assigns each node a value equal to the percentage of attached elements with a value on each variable. Thus, in our 2D regular grid example, each element contributes 25% of its value to each interior node, 50% to each side node, and the full value to the corner nodes on the domain. In this manner, we can initialize the nodal basis with an interface width while capturing every grain in the original EBSD data set.

5.2.1 *GrainTracker EBSD Initial Conditions*

When setting the initial conditions following the approach in the previous section we find that the interpolation of the nodal data is insufficient for recovering the original elemental data. This severally impedes the grain tracker's ability to successfully identify grains less than three elements across their smallest dimension accurately. To reason why this is the case, consider the small 3x3, 3 grain data set illustrated in table 5.1. In this scenario, the center element will contribute 25% of its value to each of its nodes. However, one of the surrounding grains will contribute at least 50% of its values to each of those same nodes. When we sample the shape functions in the center element the grain represented by B will have the largest value making it difficult to locate the proper grain center for the smaller grain, C.

A	A	B
A	C	B
B	B	B

TABLE 5.1: Table representing a small EBSD data set. Using the initial condition from §5.2 results in the center grain being missed by the feature extraction algorithm.

To overcome this problem, the grain tracking algorithm must consult the EBSD data directly during the initial flood stage (§4.2.1) instead of relying solely on the interpolated order parameter information. The first time that the flood algorithm is called, the EBSD data for the given element is directly retrieved to determine if the current region is part of an existing feature or the start of a new feature. This allows the initial identification of grains to exactly match the EBSD initial condition regardless of the size, shape, and discontinuities in the grain regions. On subsequent tracking invocations, we resume using the interpolated nodal solution information. Through empirical observation, this approach works reasonably well. While there may be several absorption events and other changes detected by the grain tracking algorithm, this approach is far less problematic than beginning with a condition that simply doesn't match at all. For modularity in the grain tracker implementation, this special case logic for handling EBSD data sets is abstracted away from the normal logic of the extraction and tracking in a virtual callback method and may be applied to any elemental polycrystal initial condition.

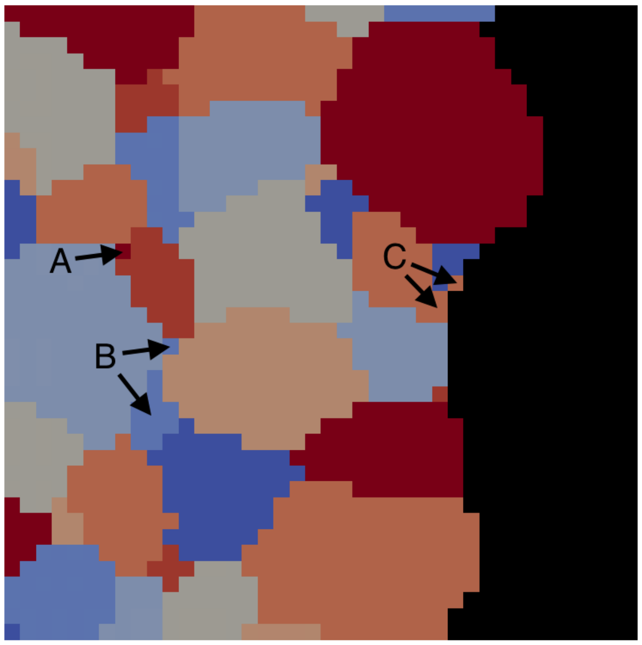


FIGURE 5.1: Initial condition from an EBSD data set containing several small grains. (A) shows one of the several single element grains present in the sample. (B) and (C) both show disjoint grains regions. The colors represent the reduced order parameter assignments.

Figure 5.1 shows the elemental initial condition constructed from an EBSD data file. It contains several single element grains whose nodal values are set to a relatively low order parameter value of 0.25 using the criterion from section 5.2. However, if we were to use a projection method instead, these small grains would be completely missed by the initial grain tracker invocation due to the larger surrounding grains that would dominate the elemental values due to the projection operation. Additionally this data set contains two grains which are not topologically connected when viewing only a 2D slice of this 3D sample. §4.4.2 discusses how these disjoint grains are handled. When any experimental data format is used, it's possible and quite common to have spatially disjoint regions appear in the data file that share the same "unique" grain identifier. The GrainTracker honors these repeated IDs even when they are disjoint and permits regions represented by the same ID to coalesce if they should come into contact during the simulation. The handling of these disjoint grain cases are discussed in section 4.4.2 and illustrated in figures 4.9 and 4.10.

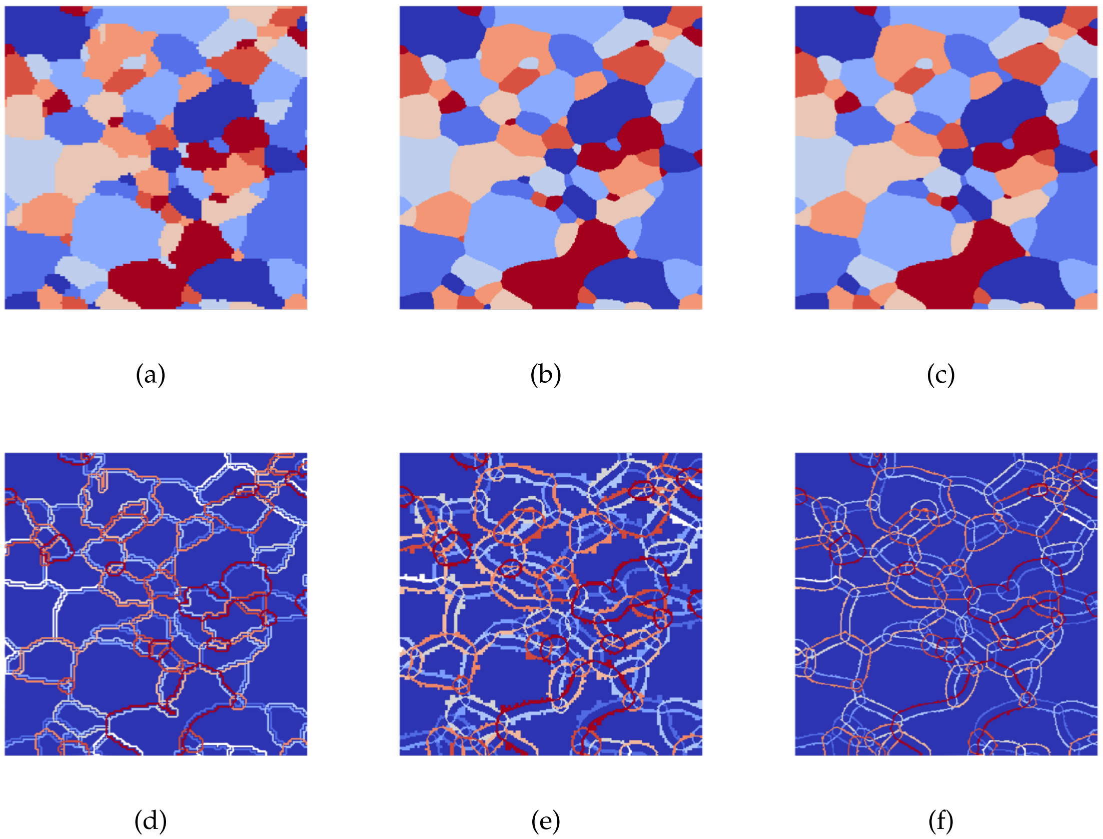


FIGURE 5.2: A microstructure reconstructed from an irradiated sample of UO_2. The physics solution process and GrainTracker algorithms work nearly identically to simulations using generated microstructures.

Figure 5.2 shows the reconstructed initial condition from an irradiated sample of Uranium Dioxide and the evolution of that sample under grain growth physics. The corresponding grain halos are also illustrated. Figure 5.3 shows the reconstructed initial condition from a high temperature alloy. Note the significant differences in the microstructures between these two materials. Once the initial conditions have been reconstructed for a material. The GrainTracker treats them identically for the duration of the simulation.

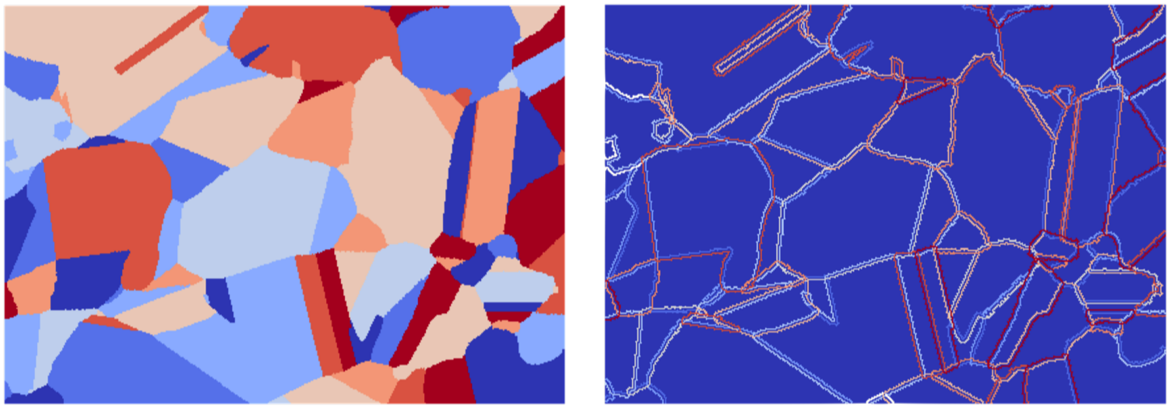

(a) (b)

FIGURE 5.3: A sample of a high temperature alloy from a commercial company. This microstructure of of this sample is very different from that observed in Figure 5.2.

5.3 Generated Initial Conditions

If experimental data is not available, one must generate a representative microstructure for the simulation. Generating realistic microstructures is a rather complex problem and there have been several proposed methodologies [37, 89, 62, 7, 98]. There most common method for generating a polycrystal microstructure is by creating a Voronoi Tessellation [11, 58, 94]. Voronoi Tessellations are produced by laying down a set of random points within the domain and then assigning each element to the closest point.

We have also implemented an experimental random initial condition where we use the phase-field simulation to evolve a set of grains beginning with random noise on each order parameter field. While this method is non-physical it can create very complex microstructures with ease. The biggest drawback of the random approach is that it is highly dependent on the number of order parameters used in the simulation. Since the initial conditions evolve through a series of small cells coalescing, fewer order parameters correspond with more coalescence. We demonstrate the use of a random initial condition in §7.3.2.

When using a Voronoi Tessellation or starting from an EBSD data set with a reduced order parameter simulation, one must distribute the smaller number of order parameters in some valid fashion to the grains in the simulation. If this is done incorrectly, it can lead to simulations with very different and quite possibly incorrect behaviors. An example of this change in behavior is discussed in §5.6 and illustrated in figure 5.4. A proper distribution requires that neighboring grains are always assigned different order parameters from any of their neighboring grains.

5.3.1 *Reduced Order Modeling as a Graph Coloring Problem*

To build a valid initial condition, we must first determine each grain's neighbors and then distributed the order parameters among all of the grains such that no two adjacent grains have the same variable. This is analogous to solving a graph coloring problem [43], which is NP-complete [47]. To visualize this problem, refer back to the figure 3.1 in §3. There are a few key differences in the traditional graph coloring problem and the assignment of initial conditions for a grain simulation. Namely, we aren't looking for a solution using only the number of order parameters allowed by the given graph's chromatic number [71], instead we are looking for a valid coloring using some slightly higher number of order parameters. This allows the simulation to evolve with headroom to allow for new edges to appear in the graph without having to worry about changing the fixed number of order parameters in a simulation.

Using a higher number of order parameters has several advantages. First, we don't spend a lot of computing resources determining the minimum number of required variables for a given problem, we simply pick some higher pre-chosen value and immediately use it to color the graph. The use of extra order parameters can also improve the performance of stochastic and more rudimentary deterministic coloring algorithms since there are many more valid colorings possible with additional available variables. Finally, less remapping is necessary with more order parameters as grains are more spaced out throughout the domain. This helps to reduce truncation errors in the solution since a finite number of elements containing a non-zero solution can be remapped for a given remapping operation. This is due to the "sigmoid-like" shape of the solution field (Figure 4.16). While, this truncation does not appear to effect the bulk-behavior of the simulation, it can increase localized variations in the solution field in the tail ends of the shape functions representing the solution fields. Two different polycrystal initial condition algorithms are compared in §5.4 and §5.5.

Before any order parameter assignment algorithm can be implemented, we first need to construct a graph indicating all neighboring grains using our finite element mesh and polycrystal structure. For maximum flexibility in implementing algorithms in this work, we use a traditional adjacency matrix [35] to represent the grain adjacency graph. This algorithm is discussed in the next section.

5.3.2 *Adjacency Matrix Construction*

To construct the adjacency matrix, a well-defined grain structure Application Program Interface (API) must be defined. Regardless of whether the grain structure is recreated from experimental data or generated the spatial information, the API needs to be queryable to determine vertex connections. Since our work is always constrained by the underlying unstructured finite element mesh representing our domain, a natural API to support for all data types is simply a method that returns the current grain ID given an element ID. We can then use this interface to construct an element to grain map which will be used to construct a graph.

The element to grain interface is naturally supported for the commonly used EBSD format (§5.2). This format already consists of a list of positions and corresponding grain identifiers. In the MOOSE framework, the `EBSDMesh` format captures the full fidelity of this format and can optionally further refine the mesh beyond the initial resolution to capture lower length physical effects. For a generated Voronoi Tessellation, one can sample a point anywhere in the domain and determine the corresponding grain ID. Additional microstructure algorithms can easily be plugged in to support this interface if needed.

Armed with this interface, we begin by constructing the element to grain ID map, the adjacency matrix can then be readily constructed. This data structure is created by iterating over all elements in the mesh and inspecting neighboring mesh elements. We consult our map and whenever adjacent elements have different grain IDs, this indicates an "edge" in our graph. Those grain IDs can be inserted directly into the adjacency matrix. In this manner, all physical neighbors can be determined. Handling periodic boundary conditions is straight-forward in this phase as well. Instead of just inspecting physical neighbors, we look at "topological" neighbors which may point to corresponding elements along a corresponding periodic interface. This algorithm however is not sufficient to satisfy the run time intersection conditions discussed in §4.4.3 due to the lack of visibility of neighbors that can only be detected through the addition of grain halos (§4.2.3), so additional work is needed to fill in the missing edge information in our adjacency matrix.

To find find the missing edges in our adjacency matrix, we have to construct grain halos as we build the adjacency graph. When we encounter a grain interface, we track both the “local IDs” and the “halo IDs” of the two elements in question in two auxiliary data structures. We append the element ID to a list keyed on the current grain ID. This is because the current element is an actual interior element since that’s what the API is designed to report. We then insert the neighboring element ID (the one with a different grain ID) into a halo list also keyed by the current grain ID. Since that element is neighboring *this* grain, it is a first level halo element because it does **not** have a matching grain ID. To generate the symmetric map for the neighboring element we insert a local ID and halo ID into our two data structures using the neighbor element as a frame of reference too. This step is not strictly necessary if one loops over every element as we’d insert this information for the neighboring grain eventually. However, In our implementation we insert the element as an optimization to handle the expensive of calculating which neighbors are active only once for the adaptive mesh refinement case.

Once the elemental loop is complete, we have a data structure containing all of the first level halo elements for all grains and a data structure containing the outer rim of interior local ids for all grains. With these two structures, the halos can be expanded in a similar fashion as the algorithm described in §4.2.3. The difference being that additional set differences are performed to avoid building the halos into the interior of each grain since we don’t have the full grain interiors mapped. After a matching halo depth is constructed the halos can be used to construct a full adjacency graph. Each pair of halos is inspected to check for intersections indicating an edge in our graph. This procedure constructs a full adjacency matrix with the exact same definition as the intersection definition used during run-time §4.4.3. The advantage of building the adjacency matrix is that it abstracts the graph information away from the finite element mesh and details of the initial condition generation. Once the graph is succinctly represented in an adjacency matrix, any number of advanced graph coloring algorithms may be employed to assign order parameters to the grains [23, 44, 26, 55, 29].

The algorithm described in this section has not been parallelized and does not currently work with the distributed mesh functionality. Each rank in a simulation using MPI builds the full adjacency matrix. While there are not any significant concerns in terms of memory usage since the adjacency matrix is a simple integer matrix, there may be opportunities for speedup in the construction of this matrix and the use of a parallel graph coloring algorithm [1, 45, 33, 34].

5.4 Backtracking Algorithm Assignment

The backtracking algorithm [53, 54, 36] is a simple algorithm that can be applied to the graph coloring problem with a high likelihood of a successful coloring. This is especially true as the space of possible valid colorings increases through the addition of extra colors. The most significant drawback to using the backtracking algorithm for the purpose of order parameter assignment is the potential exponential complexity of using this algorithm on the graph coloring problem. However, for real graphs based on polycrystal microstructures where the number of physical neighbors can be bounded and the number of colors can be sufficiently over-estimated, the run-time probabilistic expected run-time can be drastically reduced to polynomial time [64].

The backtracking algorithm takes an adjacency matrix (§5.3.2), the number of grains, and the number of available order parameters for the simulation. It assigns a color and recursively visits each graph vertex to assign the remaining colors. If an invalid configuration is created at any point in time, the recursion is terminated and a new color is attempted for a vertex at some earlier recursive invocation. The algorithm terminates when either all vertices are assigned and the graph coloring is valid or all colors for all nodes are exhausted (a highly unlikely scenario resulting in the full exponential run time in the number of grains). This algorithm has been implemented as part of the MOOSE phase field module as a starting point for analysis.

5.5 Greedy Algorithm Assignment

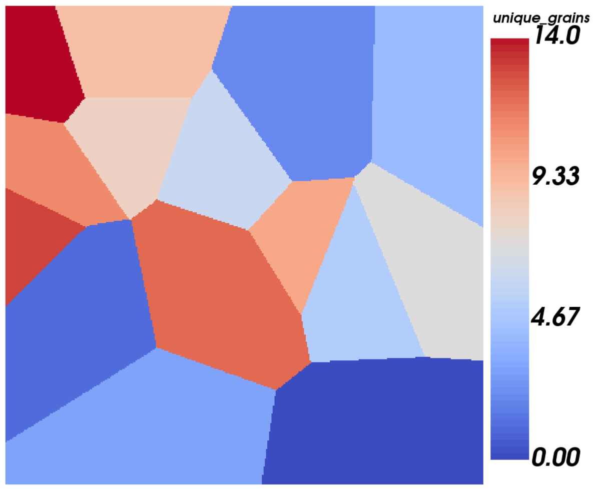
One of the most straightforward deterministic algorithms for assigning order parameters to a microstructure is to use a greedy algorithm [59]. We loop over all of the unsorted grains in the simulation. For each grain we loop over all of the remaining grains and keep track of which order parameter is furthest away by distance to the other grain centroids. If there are a sufficient number of order parameters and each grain has a relatively small number of neighbors, this algorithm can work reasonably well. The problem with this algorithm is it only makes the best choice available for each grain instead of verifying that the choice of assignment is valid (i.e. if the assignment of order parameters to grains is chosen poorly, it's possible to wind up with two neighboring grains being represented by the same order parameter as illustrated in center of Figure 5.4(c). The advantages of using this algorithm include its trivial implementation and its ability to produce a valid initial condition for a wide range of smaller test problems. Unfortunately, the likelihood of it producing valid coloring drastically decreases with the reduction in number of variables (or colors) used or if the average degree of connectivity within the graph increases [12]. It is possible to use the greedy algorithm as the starting point of a stochastic algorithm where invalid edges are removed in a later stage. The PETSc package provides a greedy coloring algorithm with these characteristics.

5.6 Greedy Algorithm vs Backtracking

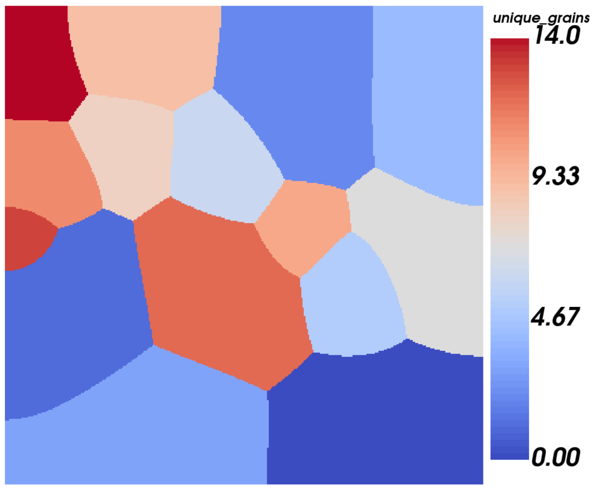
The initial order parameter assignment is very important to reduced order parameter modeling. Using a simple "greedy" algorithm or another naïve approach may yield an invalid initial condition which can have significant impacts on a simulation. Consider a simple grain growth simulation where an analyst would like to measure the largest grain under various conditions. If an invalid initial condition (i.e. invalid graph coloring) is produced, one or more abnormally large grains would exist in the initial condition with abnormal shapes, significantly impacting the physics of such a simulation. A comparison of using a proper order parameter assignment versus a naive assignment is show in figure 5.4. A valid initial condition is shown in figure 5.4(a), with the corresponding evolution at 40 time steps is shown in figure 5.4(b). Compare this with the initial condition created with the greedy algorithm (§5.5) shown in figure 5.4(c). Note the two grains in the center of the image with the same color. Since they are assigned the same order parameter, the physics simulation has no way to differentiate them and treats them as a single larger body. The incorrect assignment produces a single grain with a concave shape in the otherwise exclusive convex Voronoi initial condition. This leads to the evolution of outwardly curved surfaces on the grain as can been seen in Figure 5.4(d) leading to accelerated grain growth of the incorrectly assigned grains. This effect completely invalidates any analysis involving the sizes or distributions of grains.

5.7 Advanced Graph Coloring Algorithms

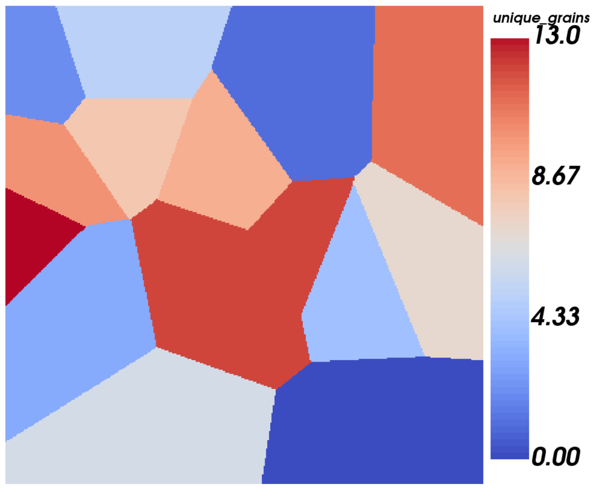
In addition to the two built-in graph coloring algorithms presented in §5.4 and §5.5, additional graph coloring algorithms are available if the PETSc package is being used as the underlying solver (§4). PETSc contains several advanced graph coloring algorithms for the purpose of building an approximate Jacobian matrix through a finite differencing technique [22]. These algorithms however, can be re-used for determining a proper polycrystal initial condition coloring by utilizing the adjacency matrix constructed in §5.3.2. The only drawback of re-using these existing routines is that we have little control over their implementation and characteristics. For instance, some of



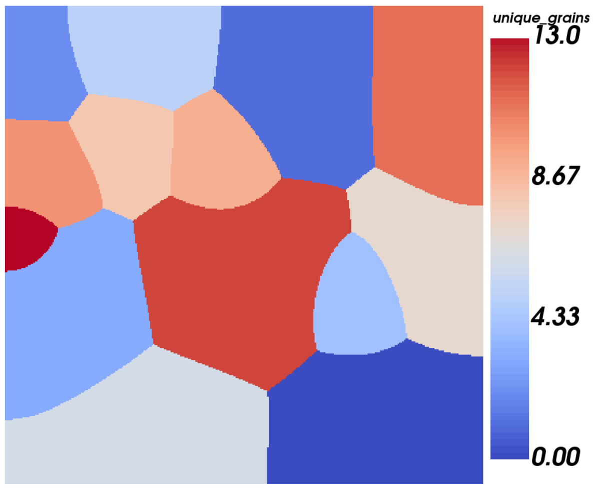
(a)



(b)



(c)



(d)

FIGURE 5.4: Comparison of two different order parameter assignment algorithms at the initial condition and after 40 time steps. Figure (a) shows a proper assignment using a backtracking algorithm with the corresponding evolution in (b). Figure (c) shows an incorrect assignment using a greedy algorithm with the corresponding evolution in (d) after 40 time steps. The concave edges created on the center grain due to incorrect assignment result in very rapid growth.

them produce a coloring that is as dense as possible regardless of the number of order parameters requested at the start of the simulation. Also, these algorithms are not designed to produce relatively even distributions of colors if several possible colorings exist. Both of these characteristics generally force the grain tracker to perform several more remapping operations in the first few time steps as the diffuse interface begins to form. Both of these differences are illustrated in figure (figure 5.5). However, these algorithms generally perform much better than either of the built-in algorithms at scale and should be used for larger 3D simulations provided PETSc is available.

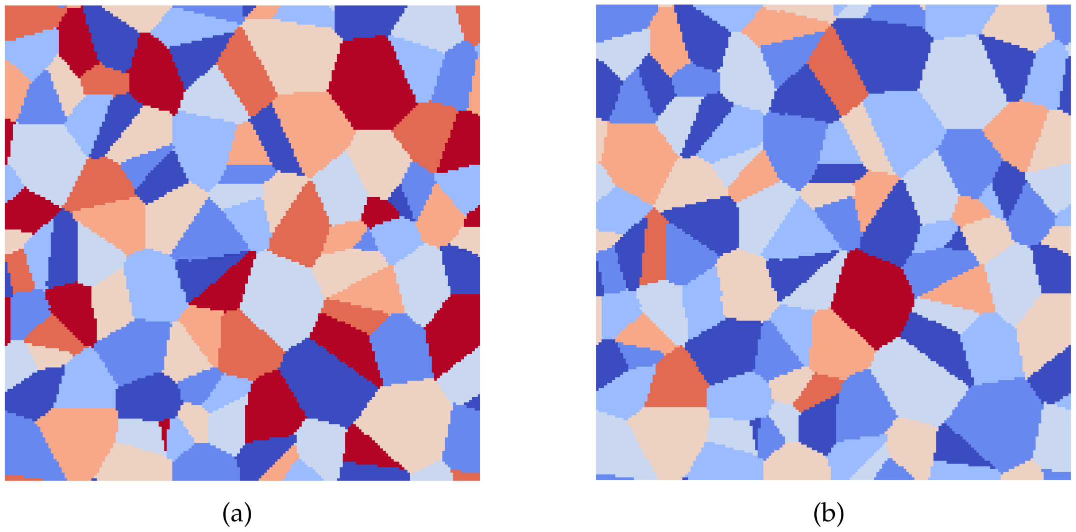


FIGURE 5.5: Comparison of a hand coded back tracking coloring algorithm designed to distribute colors somewhat evenly (a) versus an advanced coloring algorithm designed to produce a valid coloring quickly (b).

5.8 Further Initial Condition Challenges

Depending on the type of analysis being run, it may be of interest to the researcher that the specified number of grains will always be reliably created when using artificially generated initial conditions. Much of this chapter is devoted to discussing valid order parameter assignment which solves much of this problem but still does not guarantee that the desired number of grains are created using one of the stochastic methods in §5.3. The problem arises from the use of randomly generated center points in $\mathbb{R}$ space and the potential loss of information once transferred to the finite element mesh. Consider the method used in generating a Voronoi initial condition; Several points are chosen randomly within the domain. Once those points are chosen, the mesh elements are assigned to the Voronoi cell in a discrete fashion based on proximity to the nearest center point. If by chance a cluster of points fall close to one and other with respect the mesh element sizes. It's possible that very few, or even no elements will be assigned to one of the Voronoi cells. While this is perfectly valid, it may not be desirable.

One approach to ensuring that the desired number of grains are generated is given here: We can query the mesh data structure as each center point is generated to avoid certain situations. The Mesh's built-in space querying capability can be used to locate the elements containing each grain center (plural here because a center can fall exactly on an element edge or even on a node creating multiple owners). Those element IDs can then be saved to a list (or hash map for fast look-ups) for the purpose of disallowing any other center to fall within the same element for a given simulation. While this method may have an adverse impact on the random distribution properties of the grain center points, it does solve the problem of having a simulation begin with fewer than the expected number of grains. Another workaround to this problem is simply to use finer elements as the number of grains is increased. The likelihood of encountering this problem is proportional to the number of grains being simulated divided by the total number of elements in the simulation.

Chapter 6

Advanced Capabilities

"If you're teaching today what you were teaching five years ago, either the field is dead or you are."
- Noam Chomsky

6.1 Reserved Order Parameters

Irradiation induced recrystallization [77, 73, 72, 95] is another phenomenon being modeled with the MOOSE phase-field module. This process is much more challenging to model with a reduced order parameter formulation than the grain-growth model used extensively throughout this dissertation due to the rapidly changing microstructure. From an observation point of view, the irradiation damage in this process essentially causes the initial microstructure to be "replaced" by an entirely new microstructure. Instead of individual grain boundaries slowly evolving over several time steps, the grain boundaries in one of these recrystallization simulations appear to open where several new smaller grains appear and absorb the existing grain structure in all directions. With the core of the grain tracking algorithm being designed upon step to step comparisons, new capabilities where added to handle these types of simulations.

The notion of a "reserve" order parameter was added to GrainTracker object. These reserve order parameters do not differ from regular order parameter to the physics simulation but have several different attributes in the context of the GrainTracker that enable more robust handling of recrystallization simulations.

1. A separate threshold value for the flooding (grain identification) stage.

2. Grains are immediately remapped once identified on a reserve order parameter.

3. Reserved order parameters are never considered as a remapping target.

4. Grain IDs are pre-chosen and fixed for reserve order parameter grains.

The idea of the reserved order parameter is that they may serve as an nucleation and incubation area for the nucleation and initial growth of new grains. These fields are necessarily numerically “noisy”, as they undergo transient randomly placed source terms to simulate some underlying physical process such as irradiation damage. The higher threshold of the reserve field allows the GrainTracker to ignore spurious feature identification and also allows it to ignore very small grains that have nucleated but have yet to reach some critical mass allowing them to stabilize. As these new grains incubate for a few time steps, they eventually contain a point, whose value is large enough to keep an identified grain as discussed in §4.3.3. These grains are immediately remapped away from the reserve order parameter allowing new grains to be nucleated in their place.

There are no restrictions on the number of reserve or regular order parameters that may exist in a phase-field simulation. This allows researchers the flexibility to add varying numbers of each type to support a wide range of nucleation rates while balancing the associated matrix sizes.

6.2 Distributed Mesh Functionality

The libMesh [51] library upon which MOOSE [31, 32, 86] is built has support for a distributed mesh data structure. This mesh structure maintains only local and ghosted elements on each rank discarding all remaining portions of the mesh. The vast majority of users of MOOSE and libMesh do not leverage this mode because replicating the mesh data structure has speed advantages and does not consume large amounts of memory until element counts reach well into the millions and simulations are spread out to several hundred to thousands of ranks. The GrainTracker capability is fully parallelized for solution variable information as discussed throughout §4. The core grain tracking algorithms also support distributed mesh and can use this capability when used with libMesh's "ghosting functor" system.

The halo construction algorithm discussed in section 4.2.3 requires additional elements around the surface of each element. When grains are split on processor boundaries, no additional information is required to properly construct a complete halo around the grain. Each processor owning any piece of a grain is able to contribute its portion of the corresponding halo elements to the master rank for final assembly. The cases where additional information is necessary with the current implementation is when a grain's extent approaches a boundary region but does not extend onto the neighboring processor (Refer to grain D_3 in Figure 4.5 in §4.2.1). If each processor only stores the parts of the mesh it owns (plus ghosted regions), we cannot construct the complete halo without significant increases in communication. One simple solution to this issue is to have libMesh increase the thickness of the ghosted regions along all processor boundaries. The GrainTracker halos are normally sized to a thickness of at least two for all simulation types. The desired halo thickness should also be used as the ghosting thickness for full functionality. While the grain tracker has been made to support distributed mesh capability, more work remains to be done before we can successfully run larger simulations. Additional work and analysis is planned for the future (§8.1).

6.2.1 *Mesh Pre-splitting*

Distributed mesh can consume substantial amounts of memory during problem start-up when reading traditional mesh formats. This is because the framework focuses on ease of use and maximum flexibility in its supported mesh formats. Fortunately, the choice of mesh format and the mesh mode are independent. The framework is capable of reading in any serial format and broadcasting it to all ranks where several mesh setup steps are performed such as partitioning and neighbor discovery. When each processor performs the same steps on the complete mesh data structure, the amount of necessary communication and algorithmic complexity can be drastically decreased. Furthermore, the subroutines for preparing a replicated or distributed mesh can remain largely the same. The primary difference is that the processors delete all of their non-local mesh information after the meshes are setup and prepared for use in the simulation. This strategy however comes at a significant memory cost. When reading in a very large serial format, it is first replicated on all ranks before it is finalized. While each rank may have some amount of reserve headroom before the solvers, matrices and vectors are built for the problem, there is an upper limit on the amount of replicated mesh information that can be stored on all ranks. This is where pre-splitting can be used.

Pre-splitting the mesh is performed by using a utility to read in a serial mesh on some number of ranks (typically far fewer than will be used for the simulation). These ranks each work on a full replicated copy of the mesh but write out a pre-chosen number of separate non-overlapping partitions to be used by the actual simulation. The simulation is then run on the pre-chosen number of ranks exactly matching the same number of partitions previously written out. Each rank then reads in its designated mesh partition number processing only the bare minimum portion of the mesh that it owns before running the simulation.

6.3 Checkpoint Recovery Support

The MOOSE framework features a checkpoint and recovery system that can be used to split up a long running job over multiple batch runs when using computing system with limited wall-time resources. This same system also serves as a fault-tolerance system for spurious failures that may occur at scale on massively parallel simulations due to problems involving networking, memory, CPU cores or other hardware issues. When using the check-pointing system, MOOSE automatically saves the necessary mesh data structures along with all finite element data and solution fields. Developers creating objects that maintain state must request memory locations from MOOSE that will be backed up and restored as needed under a variety of conditions without user intervention.

The FeatureFloodCount object 4.2 requires no stateful data despite holding large data structures of all features in a given time step in addition to storing several maps containing "field" (spatial variable) data. The GrainTracker object has a single data structure which must be maintained across time steps and is therefore declared as restartable data. This is the "existing grains" data structure discussed in Section 4.4.1. All other data structures in both the FeatureFloodCount and GrainTracker objects are generated before each use. Simulations using either of these objects can therefore be stopped and restarted ("recovered") on demand.

Chapter 7
RESULTS

"Success is a science; if you have the conditions, you get the result."
- Oscar Wilde

The grain tracking algorithms covered in this dissertation are designed to make 3D finite element phase-field modeling feasible without impeding the advanced multi-physics capabilities present in the MOOSE phase-field module. These same algorithms drastically accelerate and reduce the sizes of 2D models making larger models possible on workstation class machines. Numerous simulations demonstrating the capabilities of the these grain tracking algorithms are presented in this section showing the drastic memory and solution time reductions possible with this tool while retaining a high degree of confidence in the correctness of the underlying model.

7.1 Reduced Computational Resources

It is difficult to make a direct comparison of the required number of order parameters between phase-field implementations using finite difference (FD) and finite element methods (FEM). While the solution values in a finite difference method are formed from discrete stencil based calculations, in FEM, the solution represents coefficients to element shape functions which make up larger basis functions that span multiple elements. As a result, FD remapping techniques are able to remap solution values on individual points without interfering with the global solution. The same approach cannot be used with FEM. Still, the minimum number of order parameters required for coalescence-free conditions presented in this work are comparable despite the additional constraints imposed when using the finite element method.

Table 7.1 shows the number of required variables for a few recent remapping implementations. The values from [50] in the second column represent the absolute minimum number of active order parameters at each simulation point without concern for neighboring values or basis functions. This method however is only applicable to explicit time integration schemes. In contrast, the halo method we are using to surround each grain allows for the closest possible interaction among entire grains while still respecting the diffuse interface requirements of the phase-field method. The author's previous work, which used a remapping method triggered by a bounding sphere collision detection algorithm [74] required several more order parameters than the current state-of-the-art FD methods. However, the values present in this dissertation (4th column) are more comparable with the FD approach in [57].

	FD: Krill [57]	FD: Kim [50]	FD: Vanherpe [96]	FEM: Permann [74]	FEM: Halo
2D	17	5		20	**8**
3D	20	6	7	120	**24**

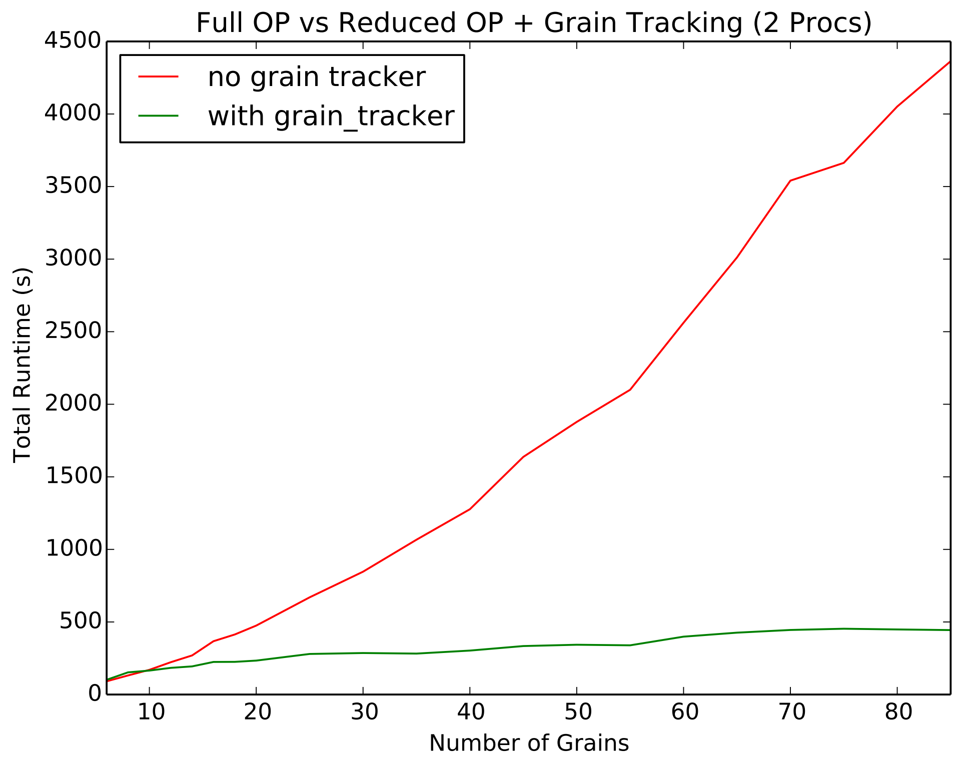
TABLE 7.1: Table representing the number of order parameter (or independent phase-field variables) required for coalescence free conditions. The first two columns contain values for recent finite difference methods, while the final two columns are numbers applicable to finite element methods. The FEM Halo method values come from the work presented in this dissertation.

Comparing memory usage and reduction is a more difficult task. The general FEM method with unstructured mesh arguably uses significantly more memory than does a stencil based finite difference method due to several extra data structures which must be maintained by the former. We can however make a meaningful comparison between the implementations written as part of this dissertation. The first implementation performed the tracking and remapping portions of this algorithm on all ranks for simplicity. While no noticeable or significant increase in memory was noted on large 2D runs, a significant jump in memory was observed on the largest 3D problems. The final implementation was written so that all tracking and remapping was performed on the master rank only with corresponding local to global maps distributed to the remaining ranks. This resulted in a memory savings of more than two terabytes of combined system memory on the 6000 grain simulation that was run on 1600 processors (§7.3.4).

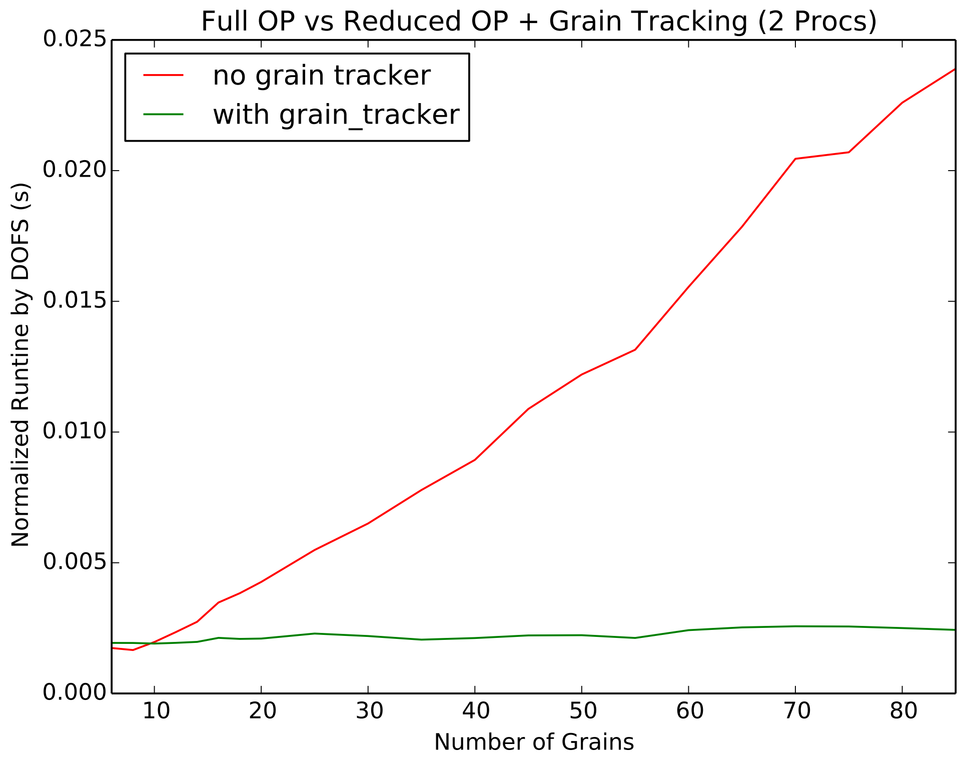
7.2 Run time Reduction

To begin verification of the GrainTracker implementation we begin with a square 2D domain and vary the number of grains in the modeled polycrystal from 6 to 100 with 24 separate runs. In the control model, we use the same number of order parameters as we do grains and observe the evolution of the system over 10 time steps. In the test model, we begin we use the same number of order parameters on the runs containing exactly 6 and 8 grains. We then hold the number of order parameters fixed at 8 while continuing to increase the number of grains modeled. For both models we also enable mesh adaptivity as would typically be done in a production run to further reduce the run time of the model. A standard "Kelly Error Estimator" [48, 101] is used to control the mesh adaptivity of the simulation. This estimator adds more mesh where solution error is higher, which in the case of a phase-field simulation of the solution of several order parameters is effectively on the boundary interfaces between grains. As the number of simulated grains is increased, additional mesh is added to maintain the desired level of error for the simulation.

Figure 7.1 shows the run time comparison of a "small" simulation with and without the the use of the GrainTracker. Figure 7.1(a) shows the total run time of both simulation types while figure 7.1(b) shows the run time normalized by the maximum number of degrees of freedom used by the adaptive mesh simulation. This normalization allows us to compare runs with different numbers of grains to one and other in a fair manner. The normalized figure shows that the simulation run time is constant with respect to the number of grains simulated, while the full order parameter simulations without the grain tracker scale with a rate $\Omega(n)$.



(a)



(b)

FIGURE 7.1: Plots showing the run time of a 2D adaptive mesh phase-field simulation on a workstation using 2 processors as a function of the number of simulated grains. Figure (a) shows the raw run time with and without the use of the grain tracker while figure (b) shows the run time normalized by the maximum number of degrees of freedom used in the adaptive mesh simulation.

Figure 7.2 shows the same simulation as shown in figure 7.1 run on 12 processors of a workstation. The overall run time is reduced for both simulation types as expected while maintaining the same growth rates as before. Note how both the 2 processor and 12 processor runs have a kink in the raw runs around the 70-75 grain cases. Upon further examination of these cases, these particular runs required slightly fewer solver iterations to obtain a converged solution thus impacting the overall run time. These kinds of variations are possible when using iterative methods as was done for these simulations.

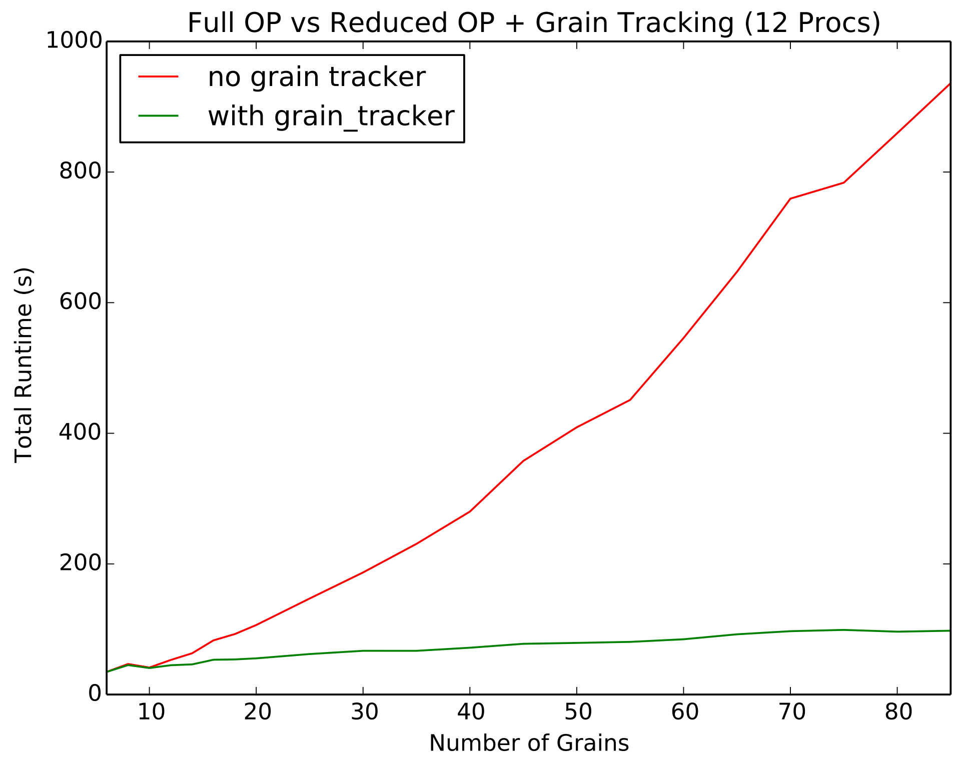
7.3 Simulation Results

We investigate the performance of the grain remapping algorithm using various 2D and 3D simulations. Every simulation is run in parallel on at least 24 processors using a 16,000 core cluster at Idaho National Laboratory.[2]

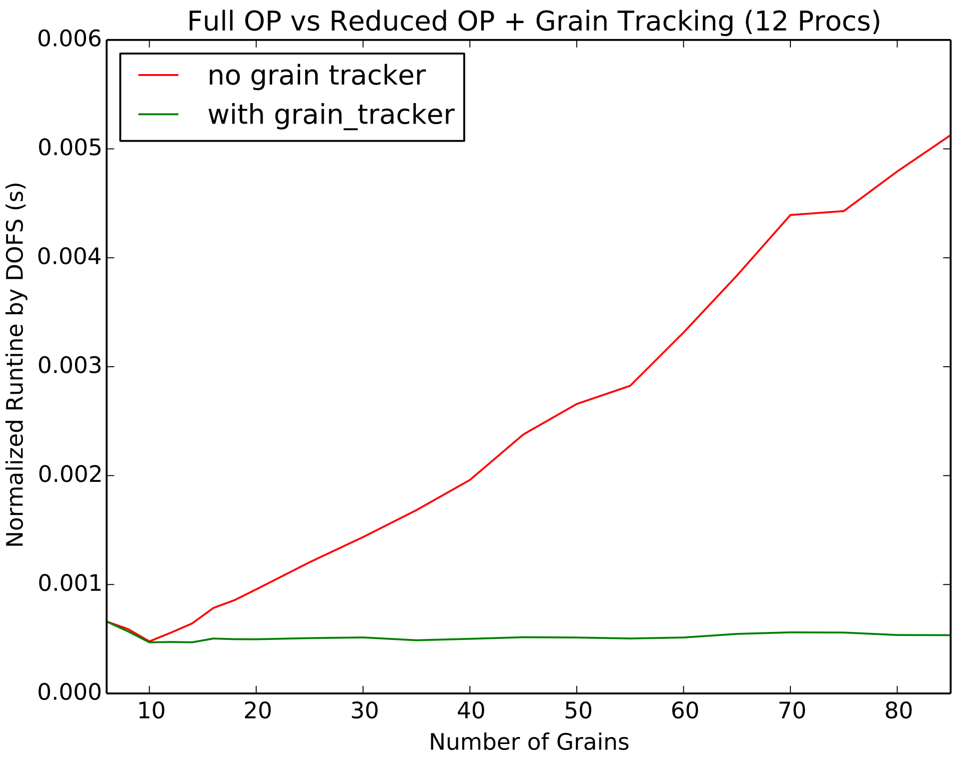
7.3.1 *Grain growth in a 2D copper polycrystal using periodic boundary conditions*

We model a 2D copper polycrystal on a 1 $\mu m \times 1$ μm domain with 450 initial grains. The grain growth is predicted using the model from [69] and implemented using the MOOSE `Phase Field` module. The initial grain structure is created using a Voronoi tessellation and the simulation is run at $T = 500$ K until 25 grains remain (see Figs. 7.3(a) and 7.3(b)). To represent copper, we assume isotropic GB properties and use a grain boundary mobility of $M_{GB} = 2.5 \times 10^{-6} e^{-0.23/k_b T}$ m^4/(Js) and GB energy $\gamma_{GB} = 0.708$ taken from [80] for a $\Sigma 29$ twist boundary. The solution was computed using implicit backward-Euler time integration with periodic boundary conditions. The simulations utilized mesh and time step adaptivity, to reduce computational expense.

[2]Each of the examples in this section were originally published in Computational Materials Science [74].



(a)



(b)

FIGURE 7.2: Plots showing the run time of a 2D adaptive mesh phase-field simulation on a workstation using 12 processors as a function of the number of simulated grains. Figure (a) shows the raw run time with and without the use of the grain tracker while figure (b) shows the run time normalized by the maximum number of degrees of freedom used in the adaptive mesh simulation.

450 grains were represented within the grain tracker using various numbers of order parameters ranging from 10 to 30 in increments of two. To evaluate the impact of the number of order parameters on the simulation, we plotted the average grain area over time (Fig. 7.3(c)). Using the original bounding sphere collision detection approach, simulations with less than 14 order parameters failed due to an insufficient number of order parameters for remapping. Simulations with 16 or more order parameters predicted the same grain growth irrespective of the number of order parameters used. The run containing 14 parameters initially failed to predict the same microstructure evolution due to an invalid initial condition where adjacent grains where assigned to the same order parameter. The average grain area over time is also shown for a simulation with 16 order parameters run without the remapping algorithm. The increase in the average grain area is significantly faster due to coalescence which is non-physical for this model.

A valid initial assignment of the order parameters is necessary to avoid coalescence during the first time step. If the initial condition assigns two adjacent grains to the same order parameter, a comparative study cannot be performed. We realized this defect and implemented a more advanced initial condition assignment algorithm (§5.4), which resolved this issue. After the halo collision detection mechanism was developed (§4.4.3), this example was re-run resulting in a model that required no more than 8 order parameters to predict the same microstructure as the reference run.

The scalable grain tracker algorithm doesn't increase the computational expense of 2D simulations by any significant amount. In 2D 450 grain run, the grain tracker used less than 2 percent of the total computational time for the 8 order parameter simulation when compared to a reduced order model without the grain tracker algorithm. However, the use of the grain tracking algorithm guarantees the same behavior of a simulation running on a non-reduced order parameter set (correct evolution without coalescence). When comparing to a non-reduced 2D simulation on the same microstructure, the savings is substantial in terms of both memory and time. Typical simulations run about two orders of magnitude faster and consume roughly two orders of magnitude less memory.

7.3.2 *Grain growth in a circular 2D copper polycrystal*

The grain tracker algorithm works for all domain sizes and shapes. In this example, we model grain growth in a circular polycrystal with 1735 initial grains, where the initial grain structure was randomly generated (see Fig. 7.4(a)). The simulation was initially conducted with 18 order parameters using the bounding sphere approach. Since this simulation uses a random evolution initial condition the number of order parameters cannot be significantly reduced without changing the properties of the grain sizes and distributions. However, using the bounding sphere and halo approaches both work reasonably well at this number of order parameters. We simulated the grain growth at $T = 500$ K and the same model and material parameters were used as were used in the previous example. The domain has a 500 nm radius and zero flux boundary conditions were applied to all order parameters on the outer edge.

After 1.23 μs, 111 grains remain in the system, as shown in Fig. 7.4(b). The average grain area over time can be computed from the number of grains recorded by the grain tracker algorithm, showing the expected linear relationship (Fig. 7.4(c)). The grain tracker also records the grain area, from which we calculated the grain size distribution. The size distribution quickly reaches a steady state value, as shown in Fig. 7.4(d). The Hillert distribution [40] is shown for reference. The grain tracker algorithm took less than 2 percent of the total computation time.

7.3.3 *Grain growth in a deformed 2D copper polycrystal*

Because the grain tracker algorithm maintains a unique ID for every grain throughout the simulation, it can be used with models that consider the impact of crystal

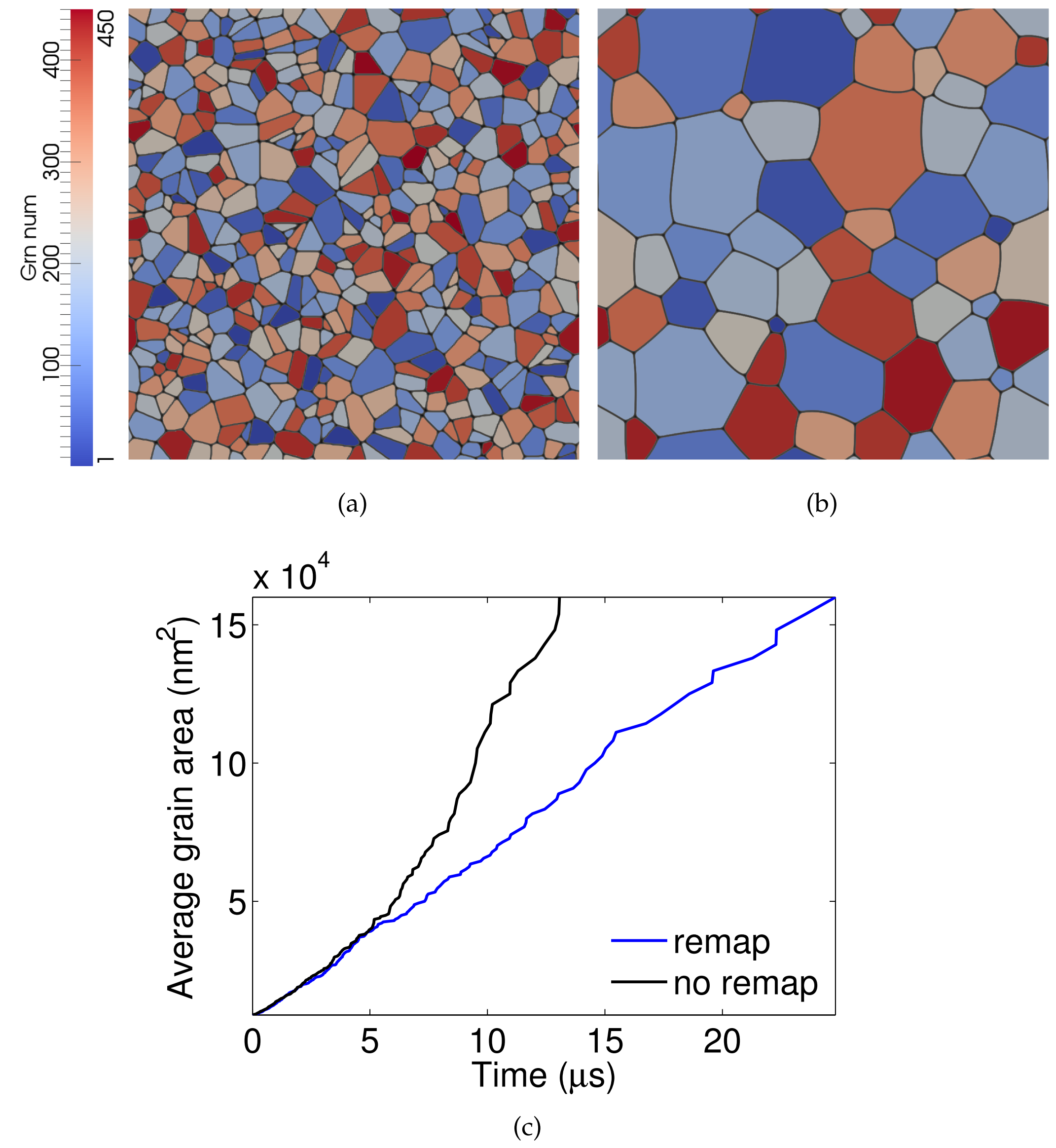


FIGURE 7.3: 2D simulation with grains spanning periodic boundary conditions. (a) initial grain structure, (b) final grain structure after 6000 ns, (c) plot of the average grain area versus time. (a) and (b) are shaded by unique grain ID. Average grain area behaves the same for sixteen or more order parameters once the grain tracker is activated. Therefore, the model is evolving correctly (without coalescence).

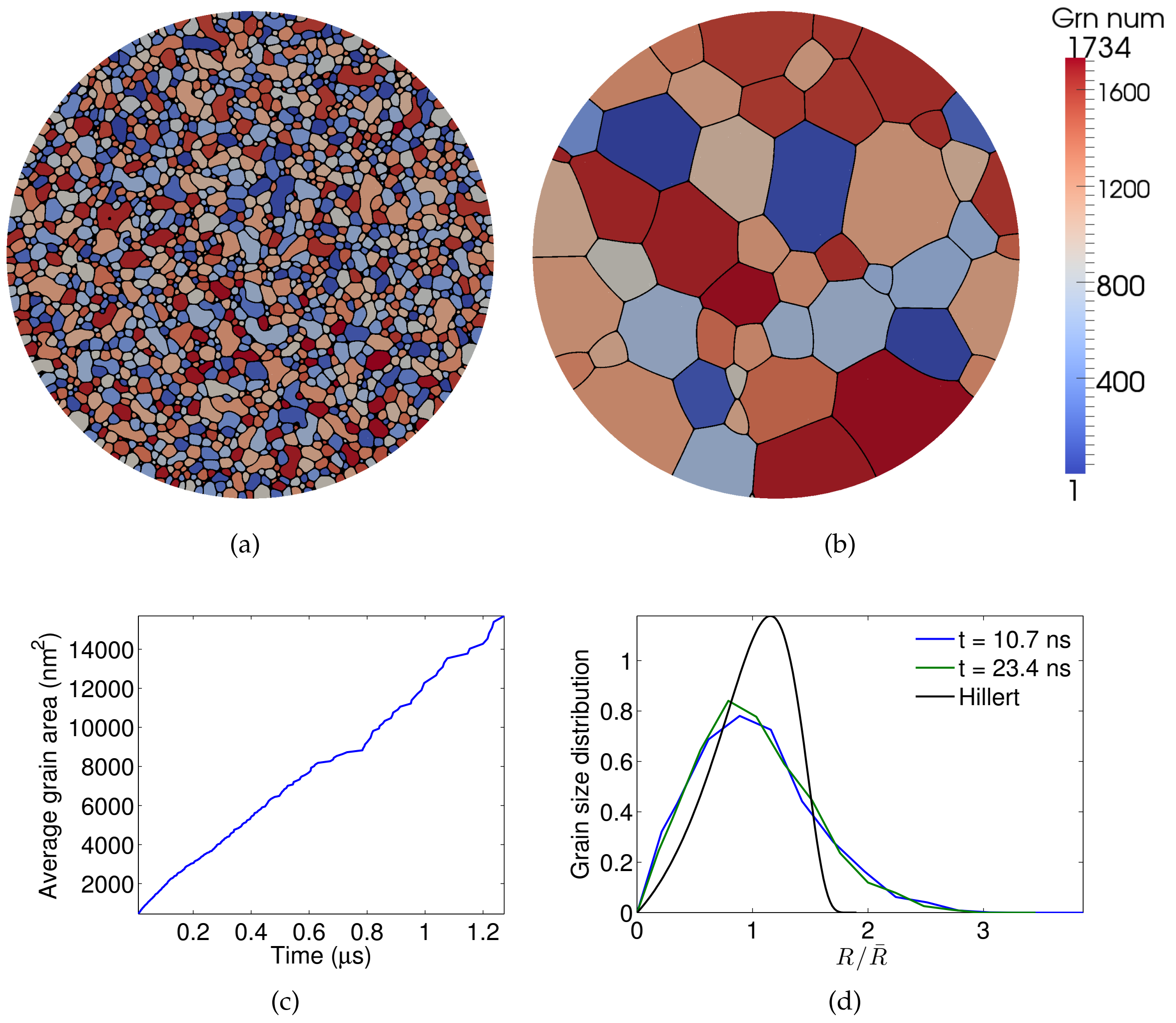
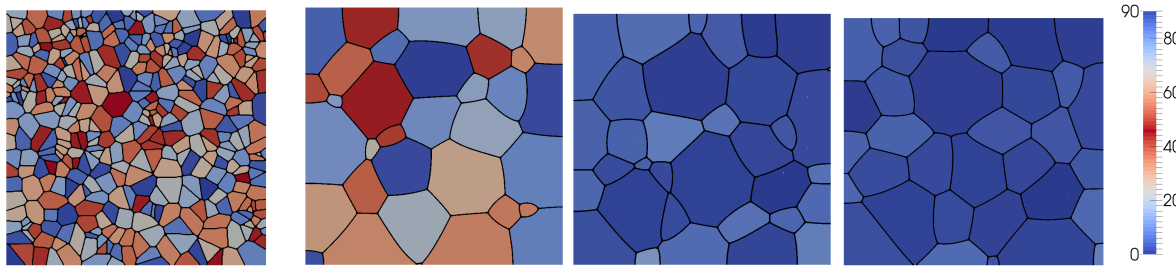


FIGURE 7.4: 2D simulations on a circular domain with zero flux boundary conditions for all order parameters in a copper polycrystal. The initial grain structure with 1723 grains, created by randomly seeding each order parameter and running for ten time steps, is show in (a). The final grain structure after 1.23 μs is shown in (b). Using information collected by the algorithm, the average grain area over time is shown in (c) and the grain size distribution is shown in (d).

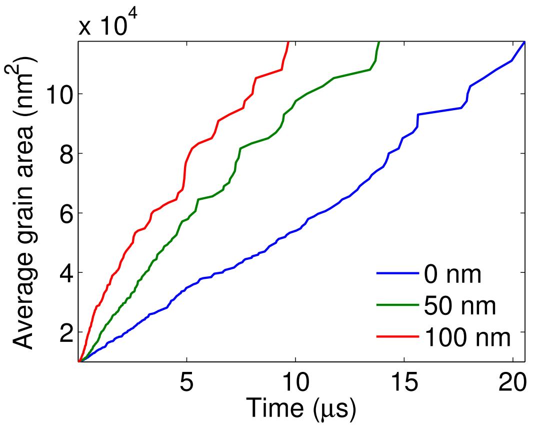
orientation. In this simulation we model grain growth in a 2D copper polycrystal that is undergoing deformation using the model from [93] and [92]. The polycrystal has 400 grains with the initial structure created by a Voronoi tessellation, as shown in Fig. 7.5(a). We assume isotropic GB properties, though we consider elastic anisotropy. The initial texture is random and only involves rotations around the $z-$axis ranging from $0°$ to $45°$, due to the cubic symmetry. Sixteen order parameters were used to represent the 400 grains.

To account for the mechanical deformation, we couple the phase field equations to a solution of a linear elastic mechanics problem, in the manner shown in [90]. The values for the three cubic elastic constants for copper are $C_{11} =$, $C_{12} =$, and $C_{14} =$ [80]. The square domain is 1 $\mu m \times 1$ μm and it is deformed downward on the top boundary by 50 and 100 nm, in two simulations respectively. The applied deformation begins at zero at time equals zero and increases linearly until it reaches the maximum value at $t = 25$ ns. For the rest of the simulation, the applied deformation is constant. The bottom surface is fixed in the y-direction, while the left and right boundaries are fixed in the x-direction. Again, the simulation is solved using implicit time integration and mesh and time step adaptivity to reduce the computational expense. The grain tracker algorithm took only a fraction of 1% of the total computation time, due to additional costs from the mechanics calculation.

As shown in the previous work [92], the slope of the plot of grain area over time increases with increasing magnitude of the applied deformation (see Fig. 7.4(d)). As the grains grow under the applied load, specific orientations grow preferentially over other grains in order to reduce the total elastic energy in the system. With no applied load, the final texture is random. As the applied load increases, the final texture becomes more and more pronounced (7.5(b), 7.5(c), and 7.5(d)). Note that this simulation would not be possible using 10 order parameters without the grain tracker algorithm, as there



(a) (b) (c) (d)



(e)

FIGURE 7.5: 2D simulations on a square domain of a 2D copper polycrystal with 400 grains under an applied load, where the initial grain structure is shown in (a). The final grain structures are shown in (b) to (d), where the grains are shaded by Euler angle. The structure with no applied strain is shown in (b), with a 50 nm applied load in (c), and with a 100 nm applied load in (d). The final texture becomes more pronounced with increasing load. The average grain area is shown over time for the three applied deformations in (e).

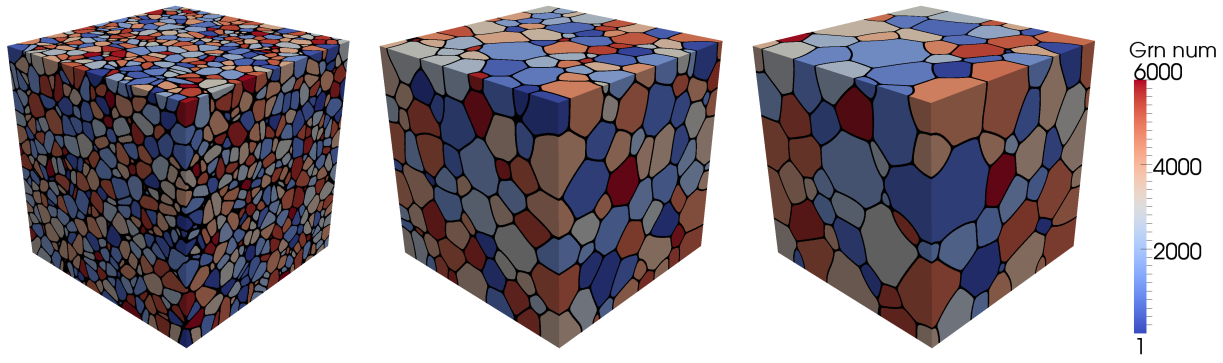
would be no unique ID to determine which crystal orientation should be assigned to each grain.

7.3.4 *Grain growth in a 3D copper polycrystal*

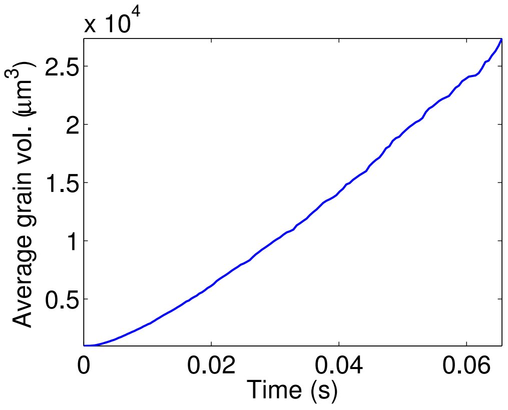
The final demonstration of the grain tracker algorithm is in a 3D copper polycrystalline cube with each side having a length of 15 μm. The number of order parameters required to represent 3D grain structures is significantly larger in 3D than in 2D because of the many more interactions between grains. The 6000 initial grains are generated using a Voronoi tessellation, as shown in Fig. 7.6(a). The evolved microstructure is also shown. As was done in the 2D circle example 7.3.2, we use the grain tracker information to compute the average grain volume over time (Fig. 7.6(b)). In addition, we compute the grain size distribution from the grain volume data taken from the grain tracker, as shown in Fig. 7.6(c). As has been shown in previous simulations, the grain size distribution rapidly changes until it reaches steady state. The steady state distribution deviates from the Hillert distribution [40] in a similar manner to that reported in [46].

In the 3D simulation, 100 order parameters were required to successfully model the grain growth using the bounding sphere approach. However, with the implementation of the halo approach, we were able to successfully run this model with only 25 order parameters. The grain tracker algorithm consumed about 7% of the computation time in 3D depending on the number of steps executed. We noted that the grain tracker consumed roughly 10% of the time in the early portion of the simulation, but as the number of grains dropped the computational requirements of the grain tracker did as well. Towards the final steps of the simulation the grain tracker was consuming about 5% of the computation time per step.

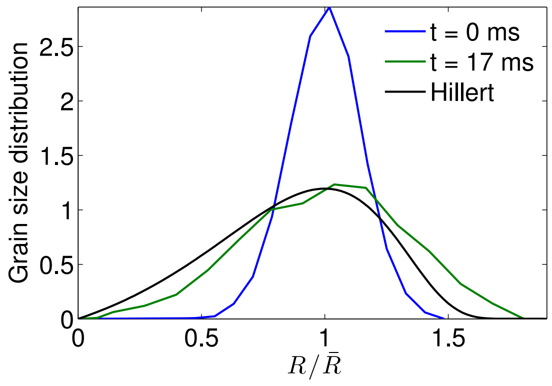
These results are significant when compared to the original grain tracker implementation. The bounding sphere implementation accounting for 72.6% of the total computation time. In either case, these simulations are only computationally feasible due to the reduced order parameter algorithm in this dissertation. Using a non-reduced order parameter set, the 6000 grain polycrystal would require 6000 order parameters to model. The grain tracker makes this simulation possible with significantly fewer variables while still maintaining a unique ID for each grain. The number of degrees of freedom have been reduced by a factor of 240 by the grain tracker algorithm. This grain tracking capability will enable many new types of 3D polycrystal simulations using FEM.



(a)



(b)



(c)

FIGURE 7.6: 3D simulations on a cubic domain of a 3D copper polycrystal with 6000 grains created with a Voronoi tessellation. The evolving grain structure is shown in (a), the the initial structure on the left, the structure after 0.03 seconds in the center and the structure after 0.065 s on the right. The average grain volume vs time is shown in (b) and the grain size distribution is shown in (c). The steady state distribution deviates slightly from the Hillert distribution, as expected [46].

Chapter 8

Conclusion and Future Work

"...if you aren't, at any given time, scandalized by code you wrote five or even three years ago, you're not learning anywhere near enough"
- Nick Black

The grain tracker is an advanced mesoscale modeling utility useful for drastically reducing the number of nonlinear solution variables needed in a finite element phase-field simulation. This reduction in model sizes allows developers to execute existing models at greatly reduced simulation times on few processing cores or alternatively, enables the execution of significantly larger simulations given the same resources over current approaches. The grain tracker algorithm in MOOSE advances previous work by moving grain tracking and remapping techniques on reduced order parameter simulations into a fully-implicit multiphysics FEM framework. The current contribution supports all time integration schemes while providing unique IDs without simulation restrictions presented in previous work. The algorithm constantly monitors the locations and potential interactions of all grains in a simulation and ensures that grains represented by the same order parameters remain out of contact at all times. This is accomplished by remapping one or more order parameters of a grain nearing an interaction to another variable in the system that is not involved in the interaction while simultaneously maintaining a unique, unchanging identifier for every grain in the simulation. The grain tracking algorithm was designed to run completely independently of the mesoscale physics used in any particular simulation. This design enables the grain tracker to work seamlessly with advanced mesoscale simulations involving heat transfer, solid mechanics, or fission gas studies without requiring any knowledge of the physics implemented in those models nor does it require any modification of those models.

Significant effort has been put into a scalable implementation, so that the grain tracker may be used on thousands of processors with mesh sizes reaching well into the millions of elements. This was accomplished with a two-pronged approach: First, the amount of data that is communicated was reduced so that only the minimum amount of information necessary to track and detect collisions is sent. Secondly, the global grain map is only maintained on one processor regardless of the size of the problem. While this creates extra complexity in the implementation of the algorithms in this dissertation, the reduction in memory and computational resources is substantial.

Several examples of the grain tracker algorithm working with various grain evolution models on different domain shapes under different conditions were presented and verified by comparing results (when possible) to non-reduced order parameter models. Finally, the grain tracker's impact to application run time was discussed and compared against other reduced order parameter models.

8.1 Future Work

The grain tracker project will usher in many new simulation possibilites and will spawn more work within the grain tracker itself as new features and capbilites are requested. The most significant and important upcoming work is the continued reduction in memory footprint possible through the use of the distributed mesh feature which requires additional work with the grain tracker's expanded halos. The initial implementation is already complete and is being debugged at the time of publication. While we were able to run distributed mesh on smaller problems, the true advantage of this system will not be fully-realized until it functions properly on very large problems. The completed and debugged functionality of the grain tracker under the distributed mesh capability will allow researchers to unlock even larger simulations than what are currently possible today.

Recently advanced initial condition coloring algorithms were added to the polycrystal initial conditions system. These algorithms are currently being tested and compared against the existing algorithms to ensure that they are functioning properly and as intended. These new algorithms will enable the proper distribution of order parameters on the largest problems without coalesce. However, before these algorithms can be fully employed, additional capabilities are required in the polycrystal initial conditions systems to avoid creating and coloring the same graph for each order parameter.

The grain tracking system has been designed with many safeguards to prevent incorrect identification of grains over a wide range of conditions. However, there will likely always exist a few pathelogical cases that will cause the system to fail. A few highly unlikely scenarios have already been identified that would not be handled properly should they be encountered. These cases involve things like simultaneous nucleation and absorbtion of nearby grains or highly concentric grains represented by the same order parameter being mismatched. Even though these cases are unlikely to happen in any simulation scenario. Proper safeguards should be added to prevent failure should they ever be encountered.

Finally, verification and validation of the phase-field method using an FEM implementation both with and without the grain tracker is an important area of research that needs to be completed before the full potential of the software can be applied to real world problems. This is a substantial amount of work that may take several years and involve many researchers. However, the fact that the grain tracker capabilty enables the software to be used in this capacity at all speaks to the successful path that this implementation is already on. Phase-field modeling beginning from EBSD initial conditions is already beginning to show its promise as a viable tool for both validating and predicting real-world experiments. These capabilities will need to be thoroughly tested on a much larger cache of data sets that were not easily obtainable through this research project. At the time of publication we are also investigating the use of creating initial conditions from other more readily available microstructure data such as images and grain orientation maps.

As this dissertation goes to publication, there are several researchers actively using the capabilities already in the grain tracker which makes this probject even more exciting. The ongoing recrystallization and sintering work will likely yeild ongoing development and improved modeling techniques.

Bibliography

[1] J. Allwright, R. Bordawekar, P. Coddington, K. Dincer, and C. Martin. A comparison of parallel graph coloring algorithms. *Technical report, SCCS-666, Northeast Parallel Architectures Center at Syracuse University*, 1995.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[3] M. Anderson, D. Srolovitz, G. Grest, and P. Sahni. Computer-Simulation Of Grain-Growth .1. Kinetics. *Acta Metallurgica*, 32(5):783–791, 1984.

[4] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016.

[5] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2016.

[6] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[7] J. Barker, G. Bollerhey, and J. Hamaekers. A multilevel approach to the evolutionary generation of polycrystalline structures. *Computational Materials Science*, 114:54–63, 2016.

[8] E. B. Becker, G. F. Carey, and J. T. Oden. Finite elements, an introduction: Volume i. *., 258*, page 1981, 1981.

[9] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[10] N. Biggs, E. K. Lloyd, and R. J. Wilson. *Graph Theory, 1736-1936*. Oxford University Press, 1976.

[11] C. M. Bishop and W. C. Carter. Relating atomistic grain boundary simulation results to the phase-field model. *Computational materials science*, 25(3):378–386, 2002.

[12] J. A. Bondy and U. S. R. Murty. *Graph theory with applications*, volume 290. Citeseer.

[13] J. Burke and D. Turnbull. Recrystallization and grain growth. *Progress in metal physics*, 3:220IN11245IN13267IN15275–244IN12266IN14274292, 1952.

[14] L. Buš and P. Tvrdík. A parallel algorithm for connected components on distributed memory machines. In *European Parallel Virtual Machine/Message Passing Interface UsersâĂŹ Group Meeting*, pages 280–287. Springer, 2001.

[15] M. Calcagnotto, D. Ponge, E. Demir, and D. Raabe. Orientation gradients and geometrically necessary dislocations in ultrafine grained dual-phase steels studied by 2d and 3d ebsd. *Materials Science and Engineering: A*, 527(10):2738–2746, 2010.

[16] J. Chen, Y. Kusurkar, and D. E. Silver. Distributed feature extraction. In *Electronic Imaging 2002*, pages 189–195. International Society for Optics and Photonics, 2002.

[17] J. Chen, D. Silver, and L. Jiang. The feature tree: Visualizing feature tracking in distributed amr datasets. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 14. IEEE Computer Society, 2003.

[18] J. Chen, D. Silver, and M. Parashar. Real time feature extraction and tracking in a computational steering environment. In *Proceedings of the 11th high performance computing symposium (HPC 2003), Orlando, FL*, 2003.

[19] L. Chen. Phase-field models for microstructure evolution. *Annu. Rev. Mater. Res.*, 32:113–40, 2002.

[20] L.-Q. Chen. Phase-field models for microstructure evolution. *Annual review of materials research*, 32(1):113–140, 2002.

[21] L. Q. Chen and J. Shen. Applications of semi-implicit fourier-spectral method to phase field equations. *Computer Physics Communications*, 108(2-3):147–158, 1998.

[22] T. F. Coleman and J. J. Moré. Estimation of sparse jacobian matrices and graph coloring blems. *SIAM journal on Numerical Analysis*, 20(1):187–209, 1983.

[23] D. Eppstein. Small maximal independent sets and faster exact graph coloring. *J. Graph Algorithms Appl.*, 7(2):131–140, 2003.

[24] D. Fan and L. Chen. Computer simulation of grain growth using a continuum field model. *Acta mater*, 45(2):611–622, 1997.

[25] L. K. Fleischer, B. Hendrickson, and A. Pınar. On identifying strongly connected components in parallel. In *International Parallel and Distributed Processing Symposium*, pages 505–511. Springer, 2000.

[26] C. Fleurent and J. A. Ferland. Genetic and hybrid algorithms for graph coloring. *Annals of Operations Research*, 63(3):437–461, 1996.

[27] H. Frost, C. Thompson, C. Howe, and J. Whang. A 2-Dimensional Computer-Simulation Of Capillarity-Driven Grain-Growth - Preliminary-Results. *Scripta Metallurgica*, 22(1):65–70, Jan 1988.

[28] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface UsersâĂŹ Group Meeting*, pages 97–104. Springer, 2004.

[29] P. Galinier and J.-K. Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of combinatorial optimization*, 3(4):379–397, 1999.

[30] D. R. Gaston, C. J. Permann, D. Andrsš, J. W. Peterson, A. E. Slaughter, J. Miller, B. Alger, F. Kong, and R. Carlsen. MOOSE Github page. `http://www.github.com/idaholab/moose`, 2016.

[31] D. R. Gaston, C. J. Permann, J. W. Peterson, A. E. Slaughter, D. Andrš, Y. Wang, M. P. Short, D. M. Perez, M. R. Tonks, J. Ortensi, et al. Physics-based multiscale coupling for full core nuclear reactor simulation. *Annals of Nuclear Energy*, 84:45–54, 2015.

[32] D. R. Gaston, J. W. Peterson, C. J. Permann, D. Andrs, A. E. Slaughter, and J. M. Miller. Continuous integration for concurrent computational framework and application development. *Journal of Open Research Software*, 2(1):e10, 2014.

[33] A. H. Gebremedhin. *Parallel graph coloring*. PhD thesis, University of Bergen Norway, 1999.

[34] A. H. Gebremedhin and F. Manne. Scalable parallel graph coloring algorithms. *Concurrency - Practice and Experience*, 12(12):1131–1146, 2000.

[35] C. Godsil and G. F. Royle. *Algebraic graph theory*, volume 207. Springer Science & Business Media, 2013.

[36] S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of the ACM (JACM)*, 12(4):516–524, 1965.

[37] M. Groeber, S. Ghosh, M. D. Uchic, and D. M. Dimiduk. A framework for automated analysis and simulation of 3d polycrystalline microstructures. part 2: Synthetic structure generation. *Acta Materialia*, 56(6):1274–1287, 2008.

[38] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[39] J. Gruber, N. Ma, Y. Wang, A. D. Rollett, and G. S. Rohrer. Sparse data structure and algorithm for the phase field method. *Modelling and simulation in materials science and engineering*, 14(7):1189, 2006.

[40] M. Hillert. On the theory of normal and abnormal grain growth. *Acta metallurgica*, 13(3):227–238, 1965.

[41] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[42] F. Humphreys. Characterisation of fine-scale microstructures by electron backscatter diffraction (ebsd). *Scripta materialia*, 51(8):771–776, 2004.

[43] T. R. Jensen and B. Toft. *Graph coloring problems*, volume 39. John Wiley & Sons, 2011.

[44] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part ii, graph coloring and number partitioning. *Operations research*, 39(3):378–406, 1991.

[45] M. T. Jones and P. E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993.

[46] R. D. Kamachali and I. Steinbach. 3-d phase-field simulation of grain growth: Topological analysis versus mean-field approximations. *Acta Materialia*, 60(6âĂŞ7):2719 – 2728, 2012.

[47] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[48] D. Kelly, D. S. Gago, O. Zienkiewicz, I. Babuska, et al. A posteriori error analysis and adaptive processes in the finite element method: Part iâĂŤerror analysis. *International journal for numerical methods in engineering*, 19(11):1593–1619, 1983.

[49] S. Kim, D. Kim, W. Kim, and Y. Park. Computer simulations of two-dimensional and three-dimensional ideal grain growth. *Phys. Rev. E*, 74(6):061605, 2006.

[50] S. G. Kim, D. I. Kim, W. T. Kim, and Y. B. Park. Computer simulations of two-dimensional and three-dimensional ideal grain growth. *Physical Review E*, 74(6):061605, 2006.

[51] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. libmesh: a c++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3-4):237–254, 2006.

[52] J. T. Klosowski, M. Held, J. S. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[53] D. Knuth. The art of computer programming, volume 4a, enumeration and backtracking, 1968.

[54] D. E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of computation*, 29(129):122–136, 1975.

[55] M. Koivisto. An o*(2ˆ n) algorithm for graph coloring and other partitioning problems via inclusion–exclusion. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 583–590. IEEE, 2006.

[56] M. J. Koop, T. Jones, and D. K. Panda. Mvapich-aptus: Scalable high-performance multi-transport mpi over infiniband. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[57] C. Krill and L. Chen. Computer simulation of 3-d grain growth using a phase-field model. *Acta Materialia*, 50(12):3059–3075, 2002.

[58] C. Krill Iii and L.-Q. Chen. Computer simulation of 3-d grain growth using a phase-field model. *Acta materialia*, 50(12):3059–3075, 2002.

[59] L. Kučera. The greedy coloring is a bad probabilistic algorithm. *Journal of Algorithms*, 12(4):674–684, 1991.

[60] Y. Lan, D. Li, and Y. Li. A mesoscale cellular automaton model for curvature-driven grain growth. *Metallurgical and Materials Transactions B*, 37B:119–29, 2006.

[61] Lawrence Livermore National Laboratory. *hypre: High Performance Preconditioners*. `http://www.llnl.gov/CASC/hypre/`.

[62] A. Leonardi, P. Scardi, and M. Leoni. Realistic nano-polycrystalline microstructures: beyond the classical voronoi tessellation. *Philosophical Magazine*, 92(8):986–1005, 2012.

[63] M. Levoy. Area flooding algorithms. 1981.

[64] Z. Á. Mann and A. Szajkó. Determining the expected runtime of an exact graph coloring algorithm. 2011.

[65] R. McColl, O. Green, and D. A. Bader. A new parallel algorithm for connected components in dynamic graphs. In *20th Annual International Conference on High Performance Computing*, pages 246–255. IEEE, 2013.

[66] W. Mclendon Iii, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[67] E. Meijering. A chronology of interpolation: from ancient astronomy to modern signal and image processing. *Proceedings of the IEEE*, 90(3):319–342, 2002.

[68] N. Moelans, B. Blanpain, and P. Wollants. An introduction to phase-field modeling of microstructure evolution. *Calphad*, 32(2):268–294, 2008.

[69] N. Moelans, B. Blanpain, and P. Wollants. Quantitative analysis of grain boundary properties in a generalized phase field model for grain growth in anisotropic systems. *Phys. Rev. B*, 78:024113, 2008.

[70] D. Moldovan, D. Wolf, S. Phillpot, and A. Haslam. Mesoscopic simulation of two-dimensional grain growth with anistropic grain-boundary properties. *Phil. Mag. A*, 82(7):1271–97, 2002.

[71] M. Morse et al. George david birkhoff and his mathematical work. *Bulletin of the American Mathematical Society*, 52(5, Part 1):357–391, 1946.

[72] K. Nogita and K. Une. Radiation-induced microstructural change in high burnup uo2 fuel pellets. *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, 91(1-4):301–306, 1994.

[73] K. Nogita and K. Une. Irradiation-induced recrystallization in high burnup uo 2 fuel. *Journal of nuclear materials*, 226(3):302–310, 1995.

[74] C. J. Permann, M. R. Tonks, B. Fromm, and D. R. Gaston. Order parameter re-mapping algorithm for 3d phase field model of grain growth using fem. *Computational Materials Science*, 115:18–25, 2016.

[75] C. J. Permann, M. R. Tonks, D. Schwen, D. R. Gaston, R. Hiromoto, and R. Martineau. Scalable distributed feature tracking and remapping on unstructured finite element meshes with adaptive refinement.

[76] C. J. Permann, M. R. Tonks, D. Schwen, D. R. Gaston, R. Hiromoto, and R. Martineau. A scalable order parameter remapping algorithm with robust initial condtions for 3d phase field grain growth using fem.

[77] J. Rest. A model for the effect of the progression of irradiation-induced recrystallization from initiation to completion on swelling of uo 2 and u–10mo nuclear fuels. *Journal of nuclear materials*, 346(2):226–232, 2005.

[78] A. Rollett, F. Humphreys, G. S. Rohrer, and M. Hatherly. *Recrystallization and related annealing phenomena*. Elsevier, 2004.

[79] O. H. Schmitt. A thermionic trigger. *Journal of Scientific Instruments*, 15(1):24, 1938.

[80] B. Schönfelder, D. Wolf, S. Phillpot, and M. Furtkamp. Molecular-dynamics method for the simulation of grain-boundary migration. *Interface Science*, 5(4):245–262, 1997.

[81] G. Seward, S. Celotto, D. Prior, J. Wheeler, and R. Pond. In situ sem-ebsd observations of the hcp to bcc phase transformation in commercially pure titanium. *Acta Materialia*, 52(4):821–832, 2004.

[82] D. Silver and X. Wang. Volume tracking. In *Visualization'96. Proceedings.*, pages 157–164. IEEE, 1996.

[83] D. Silver and X. Wang. Tracking and visualizing turbulent 3d features. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):129–141, 1997.

[84] D. Silver and X. Wang. Tracking scalar features in unstructured data sets. In *Visualization'98. Proceedings*, pages 79–86. IEEE, 1998.

[85] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2-3):135–148, 1991.

[86] A. E. Slaughter, J. W. Peterson, D. R. Gaston, C. J. Permann, D. Andrš, and J. M. Miller. Continuous integration for concurrent moose framework and application development on github. *Journal of Open Research Software*, 3(1), 2015.

[87] D. Srolovitz, M. Anderson, P. Sahni, and G. Grest. Computer-Simulation Of Grain-Growth .2. Grain-Size Distribution, Topology, And Local Dynamics. *Acta Metallurgica*, 32(5):793–802, 1984.

[88] T. Takaki, A. Yamanaka, Y. Higa, and Y. Tomita. Phase-field model during static recrystallization based on crystal-plasticity theory. *Journal of Computer-Aided Materials Design*, 14(1):75–84, 2007.

[89] K. Teferra and L. Graham-Brady. Tessellation growth models for polycrystalline microstructures. *Computational Materials Science*, 102:57–67, 2015.

[90] M. Tonks, D. Gaston, P. Millett, D. Andrs, and P. Talbot. An object-oriented finite element framework for multiphysics phase field simulations. *Comp. Mat. Sci.*, 51(1):20–29, 2012.

[91] M. Tonks, G. Hansen, D. Gaston, C. Permann, P. Millett, and D. Wolf. Fully-coupled engineering and mesoscale simulations of thermal conductivity in uo2 fuel using an implicit multiscale approach. In *Journal of Physics: Conference Series*, volume 180, page 012078. IOP Publishing, 2009.

[92] M. Tonks and P. Millett. Phase field simulations of elastic deformation-driven grain growth in 2d copper polycrystals. *Mat. Sci. Eng. A*, 528:4086–91, 2011.

[93] M. Tonks, P. Millett, W. Cai, and D. Wolf. Analysis of the elastic strain energy driving force for grain boundary migration using phase field simulation. *Scripta Materialia*, 63:1049–52, 2010.

[94] M. R. Tonks, D. Gaston, P. C. Millett, D. Andrs, and P. Talbot. An object-oriented finite element framework for multiphysics phase field simulations. *Computational Materials Science*, 51(1):20–29, 2012.

[95] K. Une, K. Nogita, S. Kashibe, and M. Imamura. Microstructural change and its influence on fission gas release in high burnup uo2 fuel. *Journal of nuclear materials*, 188:65–72, 1992.

[96] L. Vanherpe, N. Moelans, B. Blanpain, and S. Vandewalle. Bounding box algorithm for three-dimensional phase-field simulations of microstructural evolution in polycrystalline materials. *Physical Review E*, 76(5):056702, 2007.

[97] L. Vanherpe, N. Moelans, B. Blanpain, and S. Vandewalle. Bounding box framework for efficient phase field simulation of grain growth in anisotropic systems. *Computational Materials Science*, 50(7):2221–2231, 2011.

[98] Y. O. Yildiz and M. Kirca. Atomistic simulation of voronoi-based coated nanoporous metals. *Modelling and Simulation in Materials Science and Engineering*, 25(2):025008, 2016.

[99] N. Zaafarani, D. Raabe, R. Singh, F. Roters, and S. Zaefferer. Three-dimensional investigation of the texture and microstructure below a nanoindent in a cu single crystal using 3d ebsd and crystal plasticity finite element simulations. *Acta Materialia*, 54(7):1863–1876, 2006.

[100] X. Zhang, J. Chen, and S. Osher. A multiple level set method for modeling grain boundary evolution of polycrystalline materials. *Interaction MultiScale Mech*, 1(2):178–91, 2008.

[101] O. C. Zienkiewicz and J. Z. Zhu. A simple error estimator and adaptive procedure for practical engineerng analysis. *International Journal for Numerical Methods in Engineering*, 24(2):337–357, 1987.

Appendix A

C++ Implementation in the MOOSE Framework

The following pages contain the most current source code implementations used for all simulation results contained in this dissertation. The MOOSE open-source project contains the complete revision history of these files. All code is compliant with the C++11 standard and compiles under multiple compilers (*GNU 4.8+*, *Clang 3.4.0*, and *Intel2013+*).

While the algorithms contained within are designed to work with the MOOSE framework built upon libMesh, the algorithms themselves could easily be abstracted to work with other simulation frameworks if a suitable API is made available. The author highly encourages those who wish to use or expand upon the implementation here to contact the MOOSE framework mailing list for assistance. Contributions and extensions to the original code may be accepted into the project. The MOOSE phase-field module contains extensive documentation, test cases and example problems which employ these objects. It is our hope that they continue to function any remain a vital part of the MOOSE suite for many years to come.

The expected external interface for implementing these algorithms into another project is not insurmountable. Access to the mesh data structure and solution variables is essentially all that is required. Periodic boundary constraints require a bit more information but could be worked in easily enough. The MOOSE infrastructure has a nice abstraction to the DOFs in the solution vector but as long as these values can be measured, that's all that's required for using these algorithms. Since these algorithms are designed for unstructured mesh with h-adaptivity, we expect to be able to query the mesh structure for all neighbors (at any level) at every point. In a structured mesh, a simple stencil can be used to determine neighbors at any point.

I've tried to document the code as much as possible. I've attempted to remove duplicated code as much as possible which is why these two separate classes exist in the first place. The feature data structure can be found in the header of the feature flood count object and is used extensively throughout both objects. I used modern C++11 coding standards and made that object movable, not copyable. This allowed me to drop the prolific use of pointers throughout these objects and store entire data structures inside containers. The syntactic sugar of this approach is quite nice.

A.1 FeatureFloodCount.h

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/
#ifndef FEATUREFLOODCOUNT_H
#define FEATUREFLOODCOUNT_H

#include "Coupleable.h"
#include "GeneralPostprocessor.h"
#include "InfixIterator.h"
#include "MooseVariableDependencyInterface.h"
#include "ZeroInterface.h"

#include <iterator>
#include <list>
#include <set>
#include <vector>

#include "libmesh/mesh_tools.h"
#include "libmesh/periodic_boundaries.h"

// External includes
#include "bitmask_operators.h"

// Forward Declarations
class FeatureFloodCount;
class MooseMesh;
class MooseVariable;

template <>
InputParameters validParams<FeatureFloodCount>();

/**
 * This object will mark nodes or elements of continuous regions all with a unique number for the
 * purpose of counting or "coloring" unique regions in a solution.  It is designed to work with
 * either a single variable, or multiple variables.
 *
 * Note:  When inspecting multiple variables, those variables must not have regions of interest
 *        that overlap or they will not be correctly colored.
 */
class FeatureFloodCount : public GeneralPostprocessor,
                          public Coupleable,
                          public MooseVariableDependencyInterface,
                          public ZeroInterface
{
public:
  FeatureFloodCount(const InputParameters & parameters);
  ~FeatureFloodCount();

  virtual void initialSetup() override;
  virtual void meshChanged() override;

  virtual void initialize() override;
  virtual void execute() override;
  virtual void finalize() override;
  virtual Real getValue() override;

  /// Returns the total feature count (active and inactive ids, useful for sizing vectors)
  virtual std::size_t getTotalFeatureCount() const;

  /// Returns a Boolean indicating whether this feature intersects _any_ boundary
  virtual bool doesFeatureIntersectBoundary(unsigned int feature_id) const;

  /**
   * Returns a list of active unique feature ids for a particular element. The vector is indexed by
   * variable number with each entry containing either an invalid size_t type (no feature active at
   * that location) or a feature id if the variable is non-zero at that location.
   */
  virtual const std::vector<unsigned int> & getVarToFeatureVector(dof_id_type elem_id) const;

  /// Returns the variable representing the passed in feature
  virtual unsigned int getFeatureVar(unsigned int feature_id) const;

  /// Returns the number of coupled varaibles
  std::size_t numCoupledVars() const { return _n_vars; }

  ///@{
  /// Constants used for invalid indices set to the max value of std::size_t type
  static const std::size_t invalid_size_t;
  static const unsigned int invalid_id;
  ///@}

  /// Returns a const vector to the coupled variable pointers
  const std::vector<MooseVariable *> & getCoupledVars() const { return _vars; }

  enum class FieldType
  {
    UNIQUE_REGION,
    VARIABLE_COLORING,
    GHOSTED_ENTITIES,
    HALOS,
    CENTROID,
    ACTIVE_BOUNDS,
  };

  // Retrieve field information
  virtual Real
  getEntityValue(dof_id_type entity_id, FieldType field_type, std::size_t var_index = 0) const;

  inline bool isElemental() const { return _is_elemental; }

  /// This enumeration is used to indicate status of the grains in the _unique_grains data structure
  enum class Status : unsigned char
  {
    CLEAR = 0x0,
    MARKED = 0x1,
    DIRTY = 0x2,
    INACTIVE = 0x4
  };

  struct FeatureData
  {
    FeatureData() : FeatureData(std::numeric_limits<std::size_t>::max(), Status::INACTIVE) {}

    FeatureData(std::size_t var_index,
                unsigned int local_index,
                processor_id_type rank,
                Status status)
      : FeatureData(var_index, status)
    {
      _orig_ids = {std::make_pair(rank, local_index)};
    }

    FeatureData(std::size_t var_index, Status status)
      : _var_index(var_index),
        _id(invalid_id),
        _bboxes(1), // Assume at least one bounding box
        _min_entity_id(DofObject::invalid_id),
        _vol_count(0),
        _status(status),
        _intersects_boundary(false)
    {
    }

///@{
/**
 * We do not expect these objects to ever be copied. This is important
 * since they are stored in standard containers directly. To enforce
 * this, we are explicitly deleting the copy constructor, and copy
 * assignment operator.
 */
#ifdef __INTEL_COMPILER
    /**
     * 2016-07-14
     * The INTEL compiler we are currently using (2013 with GCC 4.8) appears to have a bug
     * introduced by the addition of the Point member in this structure. Even though
     * it supports move semantics on other non-POD types like libMesh::BoundingBox,
     * it fails to compile this class with the "centroid" member. Specifically, it
     * supports the move operation into the vector type but fails to work with the
     * bracket operator on std::map and the std::sort algorithm used in this class.
     * It does work with std::map::emplace() but that syntax is much less appealing
     * and still doesn't work around the issue. For now, I'm allowing the copy
     * constructor so that this class works under the Intel compiler but there
     * may be a degradation in performance in that case.
     */
    FeatureData(const FeatureData & f) = default;
    FeatureData & operator=(const FeatureData & f) = default;
#else // GCC CLANG
    FeatureData(const FeatureData & f) = delete;
    FeatureData & operator=(const FeatureData & f) = delete;
#endif
    ///@}

    ///@{
```

```
  // Default Move constructors
  FeatureData(FeatureData && f) = default;
  FeatureData & operator=(FeatureData && f) = default;
  ///@}

  ///@{
  /**
   * Update the minimum and maximum coordinates of a bounding box
   * given a Point, Elem or BBox parameter.
   */
  void updateBBoxExtremes(MeshTools::BoundingBox & bbox, const Point & node);
  void updateBBoxExtremes(MeshTools::BoundingBox & bbox, const Elem & elem);
  void updateBBoxExtremes(MeshTools::BoundingBox & bbox, const MeshTools::BoundingBox & rhs_bbox);
  ///@}

  /**
   * Determines if any of this FeatureData's bounding boxes overlap with
   * the other FeatureData's bounding boxes.
   */
  bool boundingBoxesIntersect(const FeatureData & rhs) const;

  ///@{
  /**
   * Determine if one of this FeaturesData's member sets intersects
   * the other FeatureData's corresponding set.
   */
  bool halosIntersect(const FeatureData & rhs) const;
  bool periodicBoundariesIntersect(const FeatureData & rhs) const;
  bool ghostedIntersect(const FeatureData & rhs) const;
  ///@}

  /**
   * Located the overlapping bounding box between this Feature and the
   * other Feature and expands that overlapping box accordingly.
   */
  void expandBBox(const FeatureData & rhs);

  /**
   * Merges another Feature Data into this one. This method leaves rhs
   * in an inconsistent state.
   */
  void merge(FeatureData && rhs);

  // TODO: Doco
  void clear();

  /// Comparison operator for sorting individual FeatureDatas
  bool operator<(const FeatureData & rhs) const
  {
    if (_id != invalid_id)
    {
      mooseAssert(rhs._id != invalid_id, "Asymmetric setting of ids detected during sort");

      // Sort based on ids
      return _id < rhs._id;
    }
```

```
      else
        // Sort based on processor independent information (mesh and variable info)
        return _var_index < rhs._var_index ||
               (_var_index == rhs._var_index && _min_entity_id < rhs._min_entity_id);
    }

    /// stream output operator
    friend std::ostream & operator<<(std::ostream & out, const FeatureData & feature);

    /// Holds the ghosted ids for a feature (the ids which will be used for stitching
    std::set<dof_id_type> _ghosted_ids;

    /// Holds the local ids in the interior of a feature.
    /// This data structure is only maintained on the local processor
    std::set<dof_id_type> _local_ids;

    /// Holds the ids surrounding the feature
    std::set<dof_id_type> _halo_ids;

    /// Holds the nodes that belong to the feature on a periodic boundary
    std::set<dof_id_type> _periodic_nodes;

    /// The Moose variable where this feature was found (often the "order parameter")
    std::size_t _var_index;

    /// An ID for this feature
    unsigned int _id;

    /// The vector of bounding boxes completely enclosing this feature
    /// (multiple used with periodic constraints)
    std::vector<MeshTools::BoundingBox> _bboxes;

    /// Original processor/local ids
    std::list<std::pair<processor_id_type, unsigned int>> _orig_ids;

    /// The minimum entity seen in the _local_ids, used for sorting features
    dof_id_type _min_entity_id;

    /// The count of entities contributing to the volume calculation
    std::size_t _vol_count;

    /// The centroid of the feature (average of coordinates from entities participating in
    /// the volume calculation)
    Point _centroid;

    /// The status of a feature (used mostly in derived classes like the GrainTracker)
    Status _status;

    /// Flag indicating whether this feature intersects a boundary
    bool _intersects_boundary;
  };

protected:
  /**
   * This method is used to populate any of the data structures used for storing field data (nodal
   * or elemental). It is called at the end of finalize and can make use of any of the data
   * structures created during the execution of this postprocessor.
   */
  virtual void updateFieldInfo();

  /**
   * This method will "mark" all entities on neighboring elements that
   * are above the supplied threshold. If feature is NULL, we are exploring
   * for a new region to mark, otherwise we are in the recursive calls
   * currently marking a region.
   */
  void flood(const DofObject * dof_object, std::size_t current_index, FeatureData * feature);

  /**
   * Return the starting comparison threshold to use when inspecting an entity during the flood
   * stage.
   */
  virtual Real getThreshold(std::size_t current_index) const;

  /**
   * Return the "connecting" comparison threshold to use when inspecting an entity during the flood
   * stage.
   */
  virtual Real getConnectingThreshold(std::size_t current_index) const;

  /**
   * This method is used to determine whether the current entity value is part of a feature or not.
   * Comparisons can either be greater than or less than the threshold which is controlled via
   * input parameter.
   */
  bool compareValueWithThreshold(Real entity_value, Real threshold) const;

  /**
   * Method called during the recursive flood routine that should return whether or not the current
   * entity is part of the current feature (if one is being explored), or if it's the start
   * of a new feature.
   */
  virtual bool isNewFeatureOrConnectedRegion(const DofObject * dof_object,
                                             std::size_t current_index,
                                             FeatureData *& feature,
                                             Status & status,
                                             unsigned int & new_id);

  ///@{
  /**
   * These two routines are utility routines used by the flood routine and by derived classes for
   * visiting neighbors. Since the logic is different for the elemental versus nodal case it's
   * easier to split them up.
   */
  void visitNodalNeighbors(const Node * node,
                           std::size_t current_index,
                           FeatureData * feature,
                           bool expand_halos_only);
  void visitElementalNeighbors(const Elem * elem,
                               std::size_t current_index,
                               FeatureData * feature,
                               bool expand_halos_only);
```

```
///@}

/**
 * The actual logic for visiting neighbors is abstracted out here. This method is templated to
 * handle the Nodal
 * and Elemental cases together.
 */
template <typename T>
void visitNeighborsHelper(const T * curr_entity,
                          std::vector<const T *> neighbor_entities,
                          std::size_t current_index,
                          FeatureData * feature,
                          bool expand_halos_only);

/**
 * This routine uses the local flooded data to build up the local feature data structures
 * (_feature_sets). This routine does not perform any communication so the _feature_sets data
 * structure will only contain information from the local processor after calling this routine.
 * Any existing data in the _feature_sets structure is destroyed by calling this routine.
 *
 * _feature_sets layout:
 * The outer vector is sized to one when _single_map_mode == true, otherwise it is sized for the
 * number of coupled variables. The inner list represents the flooded regions (local only
 * after this call but fully populated after parallel communication and stitching).
 */
void prepareDataForTransfer();

///@{
/**
 * These routines packs/unpack the _feature_map data into a structure suitable for parallel
 * communication operations. See the comments in these routines for the exact
 * data structure layout.
 */
void serialize(std::string & serialized_buffer);
void deserialize(std::vector<std::string> & serialized_buffers);
///@}

/**
 * This routine is called on the master rank only and stitches together the partial
 * feature pieces seen on any processor.
 */
void mergeSets(bool use_periodic_boundary_info);

/**
 * This routine handles all of the serialization, communication and deserialization of the data
 * structures containing FeatureData objects.
 */
void communicateAndMerge();

/**
 * Sort and assign ids to features based on their position in the container after sorting.
 */
void sortAndLabel();

/**
 * Calls buildLocalToGlobalIndices to build the individual local to global indicies for each rank
 * and scatters that information to all ranks. Finally, the non-master ranks update their own data
 * structures to reflect the global mappings.
 */
void scatterAndUpdateRanks();

/**
 * This routine populates a stacked vector of local to global indices per rank and the associated
 * count vector for scattering the vector to the ranks. The individual vectors can be different
 * sizes. The ith vector will be distributed to the ith processor including the master rank.
 * e.g.
 * [ ... n_0 ] [ ... n_1 ] ... [ ... n_m ]
 *
 * It is intended to be overridden in derived classes.
 */
virtual void buildLocalToGlobalIndices(std::vector<std::size_t> & local_to_global_all,
                                       std::vector<int> & counts) const;

/**
 * This method builds a lookup map for retrieving the right local feature (by index) given a
 * global index or id. max_id is passed to size the vector properly and may or may not be a
 * globally consistent number. The assumption is that any id that is later queried from this
 * object that is higher simply doesn't exist on the local processor.
 */
void buildFeatureIdToLocalIndices(unsigned int max_id);

/**
 * Helper routine for clearing up data structures during initialize and prior to parallel
 * communication.
 */
virtual void clearDataStructures();

/**
 * This routine adds the periodic node information to our data structure prior to packing the data
 * this makes those periodic neighbors appear much like ghosted nodes in a multiprocessor setting
 */
void appendPeriodicNeighborNodes(FeatureData & data) const;

/**
 * This routine updates the _region_offsets variable which is useful for quickly determining
 * the proper global number for a feature when using multimap mode
 */
void updateRegionOffsets();

/**
 * This method detects whether two sets intersect without building a result set.
 * It exits as soon as any intersection is detected.
 */
template <class InputIterator>
static inline bool setsIntersect(InputIterator first1,
                                 InputIterator last1,
                                 InputIterator first2,
                                 InputIterator last2)
{
  while (first1 != last1 && first2 != last2)
  {
    if (*first1 == *first2)
```

```
        return true;

      if (*first1 < *first2)
        ++first1;
      else if (*first1 > *first2)
        ++first2;
    }
    return false;
  }

  /*************************************************
   *************** Data Structures *****************
   ************************************************/

  /// The vector of coupled in variables
  std::vector<MooseVariable *> _vars;

  /// The threshold above (or below) where an entity may begin a new region (feature)
  const Real _threshold;
  Real _step_threshold;

  /// The threshold above (or below) which neighboring entities are flooded
  /// (where regions can be extended but not started)
  const Real _connecting_threshold;
  Real _step_connecting_threshold;

  /// A reference to the mesh
  MooseMesh & _mesh;

  /**
   * This variable is used to build the periodic node map.
   * Assumption: We are going to assume that either all variables are periodic or none are.
   *             This assumption can be relaxed at a later time if necessary.
   */
  unsigned long _var_number;

  /// This variable is used to indicate whether or not multiple maps are used during flooding
  const bool _single_map_mode;

  const bool _condense_map_info;

  /// This variable is used to indicate whether or not we identify features with
  /// unique numbers on multiple maps
  const bool _global_numbering;

  /// This variable is used to indicate whether the maps will contain unique region
  /// information or just the variable numbers owning those regions
  const bool _var_index_mode;

  /// Indicates whether or not to communicate halo map information with all ranks
  const bool _compute_halo_maps;

  /// Indicates whether or not the var to feature map is populated.
  const bool _compute_var_to_feature_map;

  /**
```

```
   * Use less-than when comparing values against the threshold value.
   * True by default.  If false, then greater-than comparison is used
   * instead.
   */
  const bool _use_less_than_threshold_comparison;

  // Convenience variable holding the number of variables coupled into this object
  const std::size_t _n_vars;

  /// Convenience variable holding the size of all the datastructures size by the number of maps
  const std::size_t _maps_size;

  /// Convenience variable holding the number of processors in this simulation
  const processor_id_type _n_procs;

  /**
   * This variable keeps track of which nodes have been visited during execution.  We don't use the
   * _feature_map for this since we don't want to explicitly store data for all the unmarked nodes
   * in a serialized datastructures.
   * This keeps our overhead down since this variable never needs to be communicated.
   */
  std::vector<std::map<dof_id_type, bool>> _entities_visited;

  /**
   * This map keeps track of which variables own which nodes.  We need a vector of them for multimap
   * mode where multiple variables can own a single mode.
   *
   * Note: This map is only populated when "show_var_coloring" is set to true.
   */
  std::vector<std::map<dof_id_type, int>> _var_index_maps;

  /// The data structure used to find neighboring elements give a node ID
  std::vector<std::vector<const Elem *>> _nodes_to_elem_map;

  /// The number of features seen by this object per map
  std::vector<unsigned int> _feature_counts_per_map;

  /// The number of features seen by this object (same as summing _feature_counts_per_map)
  unsigned int _feature_count;

  /**
   * The data structure used to hold partial and communicated feature data.
   * The data structure mirrors that found in _feature_sets, but contains
   * one additional vector indexed by processor id
   */
  std::vector<std::list<FeatureData>> _partial_feature_sets;

  /**
   * The data structure used to hold the globally unique features. The outer vector
   * is indexed by variable number, the inner vector is indexed by feature number
   */
  std::vector<FeatureData> _feature_sets;

  /**
   * The feature maps contain the raw flooded node information and eventually the unique grain
   * numbers.  We have a vector of them so we can create one per variable if that level of detail
   * is desired.
   */
  std::vector<std::map<dof_id_type, int>> _feature_maps;

  /// The vector recording the local to global feature indices
  std::vector<std::size_t> _local_to_global_feature_map;

  /// The vector recording the grain_id to local index (several indices will contain invalid_size_t)
  std::vector<std::size_t> _feature_id_to_local_index;

  /// A pointer to the periodic boundary constraints object
  PeriodicBoundaries * _pbs;

  /// Average value of the domain which can optionally be used to find features in a field
  const PostprocessorValue & _element_average_value;

  /// The map for holding reconstructed ghosted element information
  std::map<dof_id_type, int> _ghosted_entity_ids;

  /**
   * The data structure for looking up halos around features. The outer vector is for splitting out
   * the information per variable. The inner map holds the actual halo information
   */
  std::vector<std::map<dof_id_type, int>> _halo_ids;

  /**
   * The data structure which is a list of nodes that are constrained to other nodes
   * based on the imposed periodic boundary conditions.
   */
  std::multimap<dof_id_type, dof_id_type> _periodic_node_map;

  /// The set of entities on the boundary of the domain used for determining
  /// if features intersect any boundary
  std::set<dof_id_type> _all_boundary_entity_ids;

  std::map<dof_id_type, std::vector<unsigned int>> _entity_var_to_features;

  std::vector<unsigned int> _empty_var_to_features;

  /// Determines if the flood counter is elements or not (nodes)
  bool _is_elemental;

  /// Convenience variable for testing master rank
  bool _is_master;
};

template <>
void dataStore(std::ostream & stream, FeatureFloodCount::FeatureData & feature, void * context);
template <>
void dataStore(std::ostream & stream, MeshTools::BoundingBox & bbox, void * context);

template <>
void dataLoad(std::istream & stream, FeatureFloodCount::FeatureData & feature, void * context);
template <>
void dataLoad(std::istream & stream, MeshTools::BoundingBox & bbox, void * context);

```

```
template <>
struct enable_bitmask_operators<FeatureFloodCount::Status>
{
  static const bool enable = true;
};

#endif // FEATUREFLOODCOUNT_H
```

A.2 FeatureFloodCount.C

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/

#include "FeatureFloodCount.h"
#include "IndirectSort.h"
#include "MooseMesh.h"
#include "MooseUtils.h"
#include "MooseVariable.h"
#include "SubProblem.h"

#include "Assembly.h"
#include "FEProblem.h"
#include "NonlinearSystem.h"

// libMesh includes
#include "libmesh/dof_map.h"
#include "libmesh/mesh_tools.h"
#include "libmesh/periodic_boundaries.h"
#include "libmesh/point_locator_base.h"

#include <algorithm>
#include <limits>

template <>
void
dataStore(std::ostream & stream, FeatureFloodCount::FeatureData & feature, void * context)
{
  /**
   * Not that _local_ids is not stored here. It's not needed for restart, and not needed
   * during the parallel merge operation
   */
  storeHelper(stream, feature._ghosted_ids, context);
  storeHelper(stream, feature._halo_ids, context);
  storeHelper(stream, feature._periodic_nodes, context);
  storeHelper(stream, feature._var_index, context);
  storeHelper(stream, feature._id, context);
  storeHelper(stream, feature._bboxes, context);
  storeHelper(stream, feature._orig_ids, context);
  storeHelper(stream, feature._min_entity_id, context);
  storeHelper(stream, feature._vol_count, context);
  storeHelper(stream, feature._centroid, context);
  storeHelper(stream, feature._status, context);
  storeHelper(stream, feature._intersects_boundary, context);
}

template <>
void
dataStore(std::ostream & stream, MeshTools::BoundingBox & bbox, void * context)
{
  storeHelper(stream, bbox.min(), context);
  storeHelper(stream, bbox.max(), context);
}

template <>
void
dataLoad(std::istream & stream, FeatureFloodCount::FeatureData & feature, void * context)
{
  /**
   * Not that _local_ids is not loaded here. It's not needed for restart, and not needed
   * during the parallel merge operation
   */
  loadHelper(stream, feature._ghosted_ids, context);
  loadHelper(stream, feature._halo_ids, context);
  loadHelper(stream, feature._periodic_nodes, context);
  loadHelper(stream, feature._var_index, context);
  loadHelper(stream, feature._id, context);
  loadHelper(stream, feature._bboxes, context);
  loadHelper(stream, feature._orig_ids, context);
  loadHelper(stream, feature._min_entity_id, context);
  loadHelper(stream, feature._vol_count, context);
  loadHelper(stream, feature._centroid, context);
  loadHelper(stream, feature._status, context);
  loadHelper(stream, feature._intersects_boundary, context);
}

template <>
void
dataLoad(std::istream & stream, MeshTools::BoundingBox & bbox, void * context)
{
  loadHelper(stream, bbox.min(), context);
  loadHelper(stream, bbox.max(), context);
}

template <>
InputParameters
validParams<FeatureFloodCount>()
{
  InputParameters params = validParams<GeneralPostprocessor>();
  params.addRequiredCoupledVar(
      "variable",
      "The variable(s) for which to find connected regions of interests, i.e. \"features\".");
  params.addParam<Real>(
      "threshold", 0.5, "The threshold value for which a new feature may be started");
  params.addParam<Real>(
      "connecting_threshold",
      "The threshold for which an existing feature may be extended (defaults to \"threshold\")");
  params.addParam<bool>("use_single_map",
                        true,
                        "Determine whether information is tracked per "
                        "coupled variable or consolidated into one "
                        "(default: true)");
  params.addParam<bool>(
      "condense_map_info",
      false,
      "Determines whether we condense all the node values when in multimap mode (default: false)");
  params.addParam<bool>("use_global_numbering",
                        true,
                        "Determine whether or not global numbers are "
                        "used to label features on multiple maps "
                        "(default: true)");
  params.addParam<bool>("enable_var_coloring",
                        false,
                        "Instruct the Postprocessor to populate the variable index map.");
  params.addParam<bool>(
      "compute_halo_maps",
      false,
      "Instruct the Postprocessor to communicate proper halo information to all ranks");
  params.addParam<bool>("compute_var_to_feature_map",
                        false,
                        "Instruct the Postprocessor to compute the active vars to features map");
  params.addParam<bool>(
      "use_less_than_threshold_comparison",
      true,
      "Controls whether features are defined to be less than or greater than the threshold value.");

  /**
   * The FeatureFloodCount and derived objects should not to operate on the displaced mesh. These
   * objects consume variable values from the nonlinear system and use a lot of raw geometric
   * element information from the mesh. If you use the displaced system with EBSD information for
   * instance, you'll have difficulties reconciling the difference between the coordinates from the
   * EBSD data file and the potential displacements applied via boundary conditions.
   */
  params.set<bool>("use_displaced_mesh") = false;

  params.addParamNamesToGroup("use_single_map condense_map_info use_global_numbering", "Advanced");

  MooseEnum flood_type("NODAL ELEMENTAL", "ELEMENTAL");
  params.addParam<MooseEnum>("flood_entity_type",
                             flood_type,
                             "Determines whether the flood algorithm runs on nodes or elements");
  return params;
}

FeatureFloodCount::FeatureFloodCount(const InputParameters & parameters)
  : GeneralPostprocessor(parameters),
    Coupleable(this, false),
    MooseVariableDependencyInterface(),
    ZeroInterface(parameters),
    _vars(getCoupledMooseVars()),
    _threshold(getParam<Real>("threshold")),
    _connecting_threshold(isParamValid("connecting_threshold")
                              ? getParam<Real>("connecting_threshold")
                              : getParam<Real>("threshold")),
    _mesh(_subproblem.mesh()),
    _var_number(_vars[0]->number()),
    _single_map_mode(getParam<bool>("use_single_map")),
    _condense_map_info(getParam<bool>("condense_map_info")),
    _global_numbering(getParam<bool>("use_global_numbering")),
    _var_index_mode(getParam<bool>("enable_var_coloring")),
    _compute_halo_maps(getParam<bool>("compute_halo_maps")),
    _compute_var_to_feature_map(getParam<bool>("compute_var_to_feature_map")),
    _use_less_than_threshold_comparison(getParam<bool>("use_less_than_threshold_comparison")),
    _n_vars(_vars.size()),
    _maps_size(_single_map_mode ? 1 : _vars.size()),
    _n_procs(_app.n_processors()),
    _entities_visited(_vars.size()), // This map is always sized to the number of variables
    _feature_counts_per_map(_maps_size),
    _feature_count(0),
    _partial_feature_sets(_maps_size),
    _feature_maps(_maps_size),
    _pbs(nullptr),
    _element_average_value(parameters.isParamValid("elem_avg_value")
                               ? getPostprocessorValue("elem_avg_value")
                               : _real_zero),
    _halo_ids(_maps_size),
    _is_elemental(getParam<MooseEnum>("flood_entity_type") == "ELEMENTAL"),
    _is_master(processor_id() == 0)
{
  if (_var_index_mode)
    _var_index_maps.resize(_maps_size);

  addMooseVariableDependency(_vars);
}

FeatureFloodCount::~FeatureFloodCount() {}

void
FeatureFloodCount::initialSetup()
{
  // Get a pointer to the PeriodicBoundaries buried in libMesh
  _pbs = _fe_problem.getNonlinearSystemBase().dofMap().get_periodic_boundaries();

  meshChanged();

  /**
   * Size the empty var to features vector to the number of coupled variables.
   * This empty vector (but properly sized) vector is returned for elements
   * that are queried but are not in the structure (which also shouldn't happen).
   * The user is warned in this case but this helps avoid extra bounds checking
   * in user code and avoids segfaults.
   */
  _empty_var_to_features.resize(_n_vars, invalid_id);
}

void
FeatureFloodCount::initialize()
{
  // Clear the feature marking maps and region counters and other data structures
  for (auto map_num = decltype(_maps_size)(0); map_num < _maps_size; ++map_num)
  {
    _feature_maps[map_num].clear();
    _partial_feature_sets[map_num].clear();

    if (_var_index_mode)
      _var_index_maps[map_num].clear();

    _halo_ids[map_num].clear();
  }

  _feature_sets.clear();

  // Calculate the thresholds for this iteration
  _step_threshold = _element_average_value + _threshold;
  _step_connecting_threshold = _element_average_value + _connecting_threshold;

  _ghosted_entity_ids.clear();

  // Reset the feature count and max local size
  _feature_count = 0;

  _entity_var_to_features.clear();

  clearDataStructures();
}

void
FeatureFloodCount::clearDataStructures()
{
  for (auto & map_ref : _entities_visited)
    map_ref.clear();
}

void
FeatureFloodCount::meshChanged()
{
  _mesh.buildPeriodicNodeMap(_periodic_node_map, _var_number, _pbs);

  // Build a new node to element map
  _nodes_to_elem_map.clear();
  MeshTools::build_nodes_to_elem_map(_mesh.getMesh(), _nodes_to_elem_map);

  /**
   * We need to build a set containing all of the boundary entities
   * to compare against. This will be elements for elemental flooding.
   * Volumes for nodal flooding is not supported
   */
  _all_boundary_entity_ids.clear();
  if (_is_elemental)
    for (auto elem_it = _mesh.bndElemsBegin(), elem_end = _mesh.bndElemsEnd(); elem_it != elem_end;
         ++elem_it)
      _all_boundary_entity_ids.insert((*elem_it)->_elem->id());
}

void
FeatureFloodCount::execute()
{
  const auto end = _mesh.getMesh().active_local_elements_end();
  for (auto el = _mesh.getMesh().active_local_elements_begin(); el != end; ++el)
  {
    const Elem * current_elem = *el;

    // Loop over elements or nodes
    if (_is_elemental)
    {
      for (auto var_num = beginIndex(_vars); var_num < _vars.size(); ++var_num)
        flood(current_elem, var_num, nullptr /* Designates inactive feature */);
    }
    else
    {
      auto n_nodes = current_elem->n_vertices();
      for (auto i = decltype(n_nodes)(0); i < n_nodes; ++i)
      {
        const Node * current_node = current_elem->get_node(i);

        for (auto var_num = beginIndex(_vars); var_num < _vars.size(); ++var_num)
          flood(current_node, var_num, nullptr /* Designates inactive feature */);
      }
    }
  }
}

void
FeatureFloodCount::communicateAndMerge()
{
  // First we need to transform the raw data into a usable data structure
  prepareDataForTransfer();

  /**
   * The libMesh packed range routines handle the communication of the individual
   * string buffers. Here we need to create a container to hold our type
   * to serialize. It'll always be size one because we are sending a single
   * byte stream of all the data to other processors. The stream need not be
   * the same size on all processors.
   */
  std::vector<std::string> send_buffers(1);

  /**
   * Additionally we need to create a different container to hold the received
   * byte buffers. The container type need not match the send container type.
   * However, We do know the number of incoming buffers (num processors) so we'll
   * go ahead and use a vector.
   */
  std::vector<std::string> recv_buffers;
  if (_is_master)
    recv_buffers.reserve(_app.n_processors());

  serialize(send_buffers[0]);

  // Free up as much memory as possible here before we do global communication
  clearDataStructures();

  /**
   * Send the data from all processors to the root to create a complete
   * global feature map.
   */
  _communicator.gather_packed_range(0,
                                    (void *)(nullptr),
                                    send_buffers.begin(),
                                    send_buffers.end(),
                                    std::back_inserter(recv_buffers));

  if (_is_master)
  {
    // The root process now needs to deserialize and merge all of the data
    deserialize(recv_buffers);
    recv_buffers.clear();

    mergeSets(true);
  }

  // Make sure that feature count is communicated to all ranks
  _communicator.broadcast(_feature_count);
}

void
FeatureFloodCount::sortAndLabel()
{
  mooseAssert(_is_master, "sortAndLabel can only be called on the master");

  /**
   * Perform a sort to give a parallel unique sorting to the identified features.
   * We use the "min_entity_id" inside each feature to assign it's position in the
   * sorted vector.
   */
  std::sort(_feature_sets.begin(), _feature_sets.end());

#ifndef NDEBUG
  /**
   * Sanity check. Now that we've sorted the flattened vector of features
   * we need to make sure that the counts vector still lines up appropriately
   * with each feature's _var_index.
   */
  unsigned int feature_offset = 0;
  for (auto map_num = beginIndex(_feature_counts_per_map); map_num < _maps_size; ++map_num)
  {
    // Skip empty map checks
    if (_feature_counts_per_map[map_num] == 0)
      continue;

    // Check the begin and end of the current range
    auto range_front = feature_offset;
    auto range_back = feature_offset + _feature_counts_per_map[map_num] - 1;

    mooseAssert(range_front <= range_back && range_back < _feature_count,
                "Indexing error in feature sets");

    if (!_single_map_mode && (_feature_sets[range_front]._var_index != map_num ||
                              _feature_sets[range_back]._var_index != map_num))
      mooseError("Error in _feature_sets sorting, map index: ", map_num);

    feature_offset += _feature_counts_per_map[map_num];
  }
#endif

  // Label the features with an ID based on the sorting (processor number independent value)
  for (auto i = beginIndex(_feature_sets); i < _feature_sets.size(); ++i)
```

```
    _feature_sets[i]._id = i;
}

void
FeatureFloodCount::buildLocalToGlobalIndices(std::vector<std::size_t> & local_to_global_all,
                                             std::vector<int> & counts) const
{
  mooseAssert(_is_master, "This method must only be called on the root processor");

  counts.assign(_n_procs, 0);
  // Now size the individual counts vectors based on the largest index seen per processor
  for (const auto & feature : _feature_sets)
    for (const auto & local_index_pair : feature._orig_ids)
      // local index                                             // rank
      if (local_index_pair.second >= static_cast<std::size_t>(counts[local_index_pair.first]))
        counts[local_index_pair.first] = local_index_pair.second + 1;

  // Build the offsets vector
  unsigned int globalsize = 0;
  std::vector<int> offsets(_n_procs); // Type is signed for use with the MPI API
  for (auto i = beginIndex(offsets); i < offsets.size(); ++i)
  {
    offsets[i] = globalsize;
    globalsize += counts[i];
  }

  // Finally populate the master vector
  local_to_global_all.resize(globalsize, FeatureFloodCount::invalid_size_t);
  for (const auto & feature : _feature_sets)
  {
    // Get the local indices from the feature and build a map
    for (const auto & local_index_pair : feature._orig_ids)
    {
      auto rank = local_index_pair.first;
      mooseAssert(rank < _n_procs, rank << ", " << _n_procs);

      auto local_index = local_index_pair.second;
      auto stacked_local_index = offsets[rank] + local_index;

      mooseAssert(stacked_local_index < globalsize,
                  "Global index: " << stacked_local_index << " is out of range");
      local_to_global_all[stacked_local_index] = feature._id;
    }
  }
}

void
FeatureFloodCount::buildFeatureIdToLocalIndices(unsigned int max_id)
{
  _feature_id_to_local_index.assign(max_id + 1, invalid_size_t);
  for (auto feature_index = beginIndex(_feature_sets); feature_index < _feature_sets.size();
       ++feature_index)
  {
    if (_feature_sets[feature_index]._status != Status::INACTIVE)
    {
      mooseAssert(_feature_sets[feature_index]._id <= max_id,
                  "Feature ID out of range(" << _feature_sets[feature_index]._id << ')');
      _feature_id_to_local_index[_feature_sets[feature_index]._id] = feature_index;
    }
  }
}

void
FeatureFloodCount::finalize()
{
  // Gather all information on processor zero and merge
  communicateAndMerge();

  // Sort and label the features
  if (_is_master)
    sortAndLabel();

  // Send out the local to global mappings
  scatterAndUpdateRanks();

  // Populate _feature_maps and _var_index_maps
  updateFieldInfo();
}

const std::vector<unsigned int> &
FeatureFloodCount::getVarToFeatureVector(dof_id_type elem_id) const
{
  mooseDoOnce(if (!_compute_var_to_feature_map) mooseError(
      "Please set \"compute_var_to_feature_map = true\" to use this interface method"));

  const auto pos = _entity_var_to_features.find(elem_id);
  if (pos != _entity_var_to_features.end())
  {
    mooseAssert(pos->second.size() == _n_vars, "Variable to feature vector not sized properly");
    return pos->second;
  }
  else
    return _empty_var_to_features;
}

void
FeatureFloodCount::scatterAndUpdateRanks()
{
  // local to global map (one per processor)
  std::vector<int> counts;
  std::vector<std::size_t> local_to_global_all;
  if (_is_master)
    buildLocalToGlobalIndices(local_to_global_all, counts);

  // Scatter local_to_global indices to all processors and store in class member variable
  _communicator.scatter(local_to_global_all, counts, _local_to_global_feature_map);

  std::size_t largest_global_index = std::numeric_limits<std::size_t>::lowest();
  if (!_is_master)
  {
    _feature_sets.resize(_local_to_global_feature_map.size());

    /**
     * On non-root processors we can't maintain the full _feature_sets data structure since
     * we don't have all of the global information. We'll move the items from the partial
     * feature sets into a flat structure maintaining order and update the internal IDs
     * with the proper global ID.
     */
    for (auto & list_ref : _partial_feature_sets)
    {
      for (auto & feature : list_ref)
      {
        mooseAssert(feature._orig_ids.size() == 1, "feature._orig_ids length doesn't make sense");

        auto global_index = FeatureFloodCount::invalid_size_t;
        auto local_index = feature._orig_ids.begin()->second;

        if (local_index < _local_to_global_feature_map.size())
          global_index = _local_to_global_feature_map[local_index];

        if (global_index != FeatureFloodCount::invalid_size_t)
        {
          if (global_index > largest_global_index)
            largest_global_index = global_index;

          // Set the correct global index
          feature._id = global_index;

          /**
           * Important: Make sure we clear the local status if we received a valid global
           * index for this feature. It's possible that we have a status of INVALID
           * on the local processor because there was never any starting threshold found.
           * However, the root processor wouldn't have sent an index if it didn't find
           * a starting threshold connected to our local piece.
           */
          feature._status &= ~Status::INACTIVE;

          // Move the feature into the correct place
          _feature_sets[local_index] = std::move(feature);
        }
      }
    }
  }
  else
  {
    for (auto global_index : local_to_global_all)
      if (global_index != FeatureFloodCount::invalid_size_t && global_index > largest_global_index)
        largest_global_index = global_index;
  }

  buildFeatureIdToLocalIndices(largest_global_index);
}

Real
FeatureFloodCount::getValue()
{
  return static_cast<Real>(_feature_count);
}
```

```

std::size_t
FeatureFloodCount::getTotalFeatureCount() const
{
  /**
   * Since the FeatureFloodCount object doesn’t maintain any information about
   * features between invocations. The maximum id in use is simply the number of
   * features.
   */
  return _feature_count;
}

unsigned int
FeatureFloodCount::getFeatureVar(unsigned int feature_id) const
{
  // Some processors don’t contain the largest feature id, in that case we just return invalid_id
  if (feature_id >= _feature_id_to_local_index.size())
    return invalid_id;

  auto local_index = _feature_id_to_local_index[feature_id];
  if (local_index != invalid_size_t)
  {
    mooseAssert(local_index < _feature_sets.size(), "local_index out of bounds");
    return _feature_sets[local_index]._status != Status::INACTIVE
               ? _feature_sets[feature_id]._var_index
               : invalid_id;
  }

  return invalid_id;
}

bool
FeatureFloodCount::doesFeatureIntersectBoundary(unsigned int feature_id) const
{
  // TODO: This information is not parallel consistent when using FeatureFloodCounter

  // Some processors don’t contain the largest feature id, in that case we just return invalid_id
  if (feature_id >= _feature_id_to_local_index.size())
    return false;

  auto local_index = _feature_id_to_local_index[feature_id];

  if (local_index != invalid_size_t)
  {
    mooseAssert(local_index < _feature_sets.size(), "local_index out of bounds");
    return _feature_sets[local_index]._intersects_boundary;
  }

  return false;
}

Real
FeatureFloodCount::getEntityValue(dof_id_type entity_id,
                                  FieldType field_type,
                                  std::size_t var_index) const
{
  auto use_default = false;
  if (var_index == invalid_size_t)
  {
    use_default = true;
    var_index = 0;
  }

  mooseAssert(var_index < _maps_size, "Index out of range");

  switch (field_type)
  {
    case FieldType::UNIQUE_REGION:
    {
      const auto entity_it = _feature_maps[var_index].find(entity_id);

      if (entity_it != _feature_maps[var_index].end())
        return entity_it->second; // + _region_offsets[var_index];
      else
        return -1;
    }

    case FieldType::VARIABLE_COLORING:
    {
      const auto entity_it = _var_index_maps[var_index].find(entity_id);

      if (entity_it != _var_index_maps[var_index].end())
        return entity_it->second;
      else
        return -1;
    }

    case FieldType::GHOSTED_ENTITIES:
    {
      const auto entity_it = _ghosted_entity_ids.find(entity_id);

      if (entity_it != _ghosted_entity_ids.end())
        return entity_it->second;
      else
        return -1;
    }

    case FieldType::HALOS:
    {
      if (!use_default)
      {
        const auto entity_it = _halo_ids[var_index].find(entity_id);
        if (entity_it != _halo_ids[var_index].end())
          return entity_it->second;
      }
      else
      {
        // Showing halos in reverse order for backwards compatibility
        for (auto map_num = _maps_size;
             map_num-- /* don't compare greater than zero for unsigned */;)
        {
          const auto entity_it = _halo_ids[map_num].find(entity_id);
```

```cpp

          if (entity_it != _halo_ids[map_num].end())
            return entity_it->second;
        }
      }
      return -1;
    }

    case FieldType::CENTROID:
    {
      if (_periodic_node_map.size())
        mooseDoOnce(mooseWarning(
            "Centroids are not correct when using periodic boundaries, contact the MOOSE team"));

      // If this element contains the centroid of one of features, return one
      const auto * elem_ptr = _mesh.elemPtr(entity_id);

      for (const auto & feature : _feature_sets)
      {
        if (feature._status == Status::INACTIVE)
          continue;

        if (elem_ptr->contains_point(feature._centroid))
          return 1;
      }

      return 0;
    }

    default:
      return 0;
  }
}

void
FeatureFloodCount::prepareDataForTransfer()
{
  MeshBase & mesh = _mesh.getMesh();

  std::set<dof_id_type> local_ids_no_ghost, set_difference;

  for (auto & list_ref : _partial_feature_sets)
    for (auto & feature : list_ref)
    {
      /**
       * We need to adjust the halo markings before sending. We need to discard all of the
       * local cell information but not any of the stitch region information. To do that
       * we subtract off the ghosted cells from the local cells and use that in the
       * set difference operation with the halo_ids.
       */
      std::set_difference(feature._local_ids.begin(),
                          feature._local_ids.end(),
                          feature._ghosted_ids.begin(),
                          feature._ghosted_ids.end(),
                          std::insert_iterator<std::set<dof_id_type>>(local_ids_no_ghost,
                                                                      local_ids_no_ghost.begin()));

      std::set_difference(
          feature._halo_ids.begin(),
          feature._halo_ids.end(),
          local_ids_no_ghost.begin(),
          local_ids_no_ghost.end(),
          std::insert_iterator<std::set<dof_id_type>>(set_difference, set_difference.begin()));
      feature._halo_ids.swap(set_difference);
      local_ids_no_ghost.clear();
      set_difference.clear();

      mooseAssert(!feature._local_ids.empty(), "local entity ids cannot be empty");

      /**
       * Save off the min entity id present in the feature to uniquely
       * identify the feature regardless of n_procs
       */
      feature._min_entity_id = *feature._local_ids.begin();

      for (auto & entity_id : feature._local_ids)
      {
        /**
         * Update the bounding box.
         *
         * Note: There will always be one and only one bbox while we are building up our
         * data structures because we haven't started to stitch together any regions yet.
         */
        if (_is_elemental)
          feature.updateBBoxExtremes(feature._bboxes[0], *mesh.elem(entity_id));
        else
          feature.updateBBoxExtremes(feature._bboxes[0], mesh.node(entity_id));
      }

      // Now extend the bounding box by the halo region
      for (auto & halo_id : feature._halo_ids)
      {
        if (_is_elemental)
          feature.updateBBoxExtremes(feature._bboxes[0], *mesh.elem(halo_id));
        else
          feature.updateBBoxExtremes(feature._bboxes[0], mesh.node(halo_id));
      }

      // Periodic node ids
      appendPeriodicNeighborNodes(feature);
    }
}

void
FeatureFloodCount::serialize(std::string & serialized_buffer)
{
  // stream for serializing the _partial_feature_sets data structure to a byte stream
  std::ostringstream oss;

  /**
   * Call the MOOSE serialization routines to serialize this processor's data.
   * Note: The _partial_feature_sets data structure will be empty for all other processors
   */
  dataStore(oss, _partial_feature_sets, this);

  // Populate the passed in string pointer with the string stream’s buffer contents
  serialized_buffer.assign(oss.str());
}

/**
 * This routine takes the vector of byte buffers (one for each processor), deserializes them
 * into a series of FeatureSet objects, and appends them to the _feature_sets data structure.
 *
 * Note: It is assumed that local processor information may already be stored in the _feature_sets
 * data structure so it is not cleared before insertion.
 */
void
FeatureFloodCount::deserialize(std::vector<std::string> & serialized_buffers)
{
  // The input string stream used for deserialization
  std::istringstream iss;

  mooseAssert(serialized_buffers.size() == _app.n_processors(),
              "Unexpected size of serialized_buffers: " << serialized_buffers.size());
  auto rank = processor_id();
  for (auto proc_id = beginIndex(serialized_buffers); proc_id < serialized_buffers.size();
       ++proc_id)
  {
    /**
     * We should already have the local processor data in the features data structure.
     * Don’t unpack the local buffer again.
     */
    if (proc_id == rank)
      continue;

    iss.str(serialized_buffers[proc_id]); // populate the stream with a new buffer
    iss.clear();                          // reset the string stream state

    // Load the communicated data into all of the other processors’ slots
    dataLoad(iss, _partial_feature_sets, this);
  }
}

void
FeatureFloodCount::mergeSets(bool use_periodic_boundary_info)
{
  Moose::perf_log.push("mergeSets()", "FeatureFloodCount");

  // Since we gathered only on the root process, we only need to merge sets on the root process.
  mooseAssert(_is_master, "mergeSets() should only be called on the root process");

  // Local variable used for sizing structures, it will be >= the actual number of features
  for (auto map_num = decltype(_maps_size)(0); map_num < _maps_size; ++map_num)
  {
    for (auto it1 = _partial_feature_sets[map_num].begin();
         it1 != _partial_feature_sets[map_num].end();
         /* No increment on it1 */)
    {
      bool merge_occured = false;
      for (auto it2 = _partial_feature_sets[map_num].begin();
           it2 != _partial_feature_sets[map_num].end();
           ++it2)
      {
        // clang-format off
        if (it1 != it2 &&                             // iters aren't pointing at the same item and
            it1->_var_index ==  it2->_var_index &&    // the sets have matching variable indices and
            ((it1->boundingBoxesIntersect(*it2) &&    //  (if the feature's bboxes intersect and
              it1->ghostedIntersect(*it2))            //   the ghosted entities also intersect)
              ||                                      //   or
             (use_periodic_boundary_info &&           //  (if merging across periodic nodes and
              it1->periodicBoundariesIntersect(*it2) //   those node sets intersect)
            )))
        // clang-format on
        {
          it2->merge(std::move(*it1));

          /**
           * Insert the new entity at the end of the list so that it may be checked against all
           * other partial features again.
           */
          _partial_feature_sets[map_num].emplace_back(std::move(*it2));

          /**
           * Now remove both halves the merged features: it2 contains the "moved" feature cell just
           * inserted at the back of the list, it1 contains the mostly empty other half. We have to
           * be careful about the order in which these two elements are deleted. We delete it2 first
           * since we don't care where its iterator points after the deletion. We are going to break
           * out of this loop anyway. If we delete it1 first, it may end up pointing at the same
           * location as it2 which after the second deletion would cause both of the iterators to be
           * invalidated.
           */
          _partial_feature_sets[map_num].erase(it2);
          it1 = _partial_feature_sets[map_num].erase(it1); // it1 is incremented here!

          // A merge occurred, this is used to determine whether or not we increment the outer
          // iterator
          merge_occured = true;

          // We need to start the list comparison over for the new it1 so break here
          break;
        }
      } // it2 loop

      if (!merge_occured) // No merges so we need to manually increment the outer iterator
        ++it1;

    } // it1 loop
  }   // map loop

  /**
   * Now that the merges are complete we need to adjust the centroid, and halos.
   * Additionally, To make several of the sorting and tracking algorithms more straightforward,
   * we will move the features into a flat vector. Finally we can count the final number of
   * features and find the max local index seen on any processor
   * Note: This is all occurring on rank 0 only!
   */
  // Offset where the current set of features with the same variable id starts in the flat vector
  unsigned int feature_offset = 0;
  // Set the member feature count to zero and start counting the actual features
  _feature_count = 0;

  for (auto map_num = decltype(_maps_size)(0); map_num < _maps_size; ++map_num)
  {
    std::set<dof_id_type> set_difference;
    for (auto & feature : _partial_feature_sets[map_num])
    {
      // If after merging we still have an inactive feature, discard it
      if (feature._status == Status::CLEAR)
      {
        // First we need to calculate the centroid now that we are doing merging all partial
        // features
        if (feature._vol_count != 0)
          feature._centroid /= feature._vol_count;

        _feature_sets.emplace_back(std::move(feature));
        ++_feature_count;
      }
    }

    // Record the feature numbers just for the current map
    _feature_counts_per_map[map_num] = _feature_count - feature_offset;

    // Now update the running feature count so we can calculate the next map's contribution
    feature_offset = _feature_count;

    // Clean up the "moved" objects
    _partial_feature_sets[map_num].clear();
  }

  /**
   * IMPORTANT: FeatureFloodCount::_feature_count is set on rank 0 at this point but
   * we can't broadcast it here because this routine is not collective.
   */

  Moose::perf_log.pop("mergeSets()", "FeatureFloodCount");
}

void
FeatureFloodCount::updateFieldInfo()
{
  for (auto i = beginIndex(_feature_sets); i < _feature_sets.size(); ++i)
  {
    auto & feature = _feature_sets[i];
    decltype(i) global_feature_number;

    if (_is_master)
      /**
       * If we are on processor zero, the global feature number is simply the current
       * index since we previously merged and sorted the partial features.
       */
      global_feature_number = i;
    else
    {
      /**
       * For the remaining ranks, obtaining the feature number requires us to
       * first obtain the original local index (stored inside of the feature).
       * Once we have that index, we can use it to look up the global id
       * in the local to global map.
       */
      mooseAssert(feature._orig_ids.size() == 1, "feature._orig_ids length doesn't make sense");

      // Get the local ID from the orig IDs
      auto local_id = feature._orig_ids.begin()->second;
      mooseAssert(local_id < _local_to_global_feature_map.size(),
                  "local_id : " << local_id << " is out of range ("
                                << _local_to_global_feature_map.size()
                                << ')');
      global_feature_number = _local_to_global_feature_map[local_id];
    }

    // If the developer has requested _condense_map_info we'll make sure we only update the zeroth
    // map
    auto map_index = (_single_map_mode || _condense_map_info) ? decltype(feature._var_index)(0)
                                                              : feature._var_index;

    // Loop over the entity ids of this feature and update our local map
    for (auto entity : feature._local_ids)
    {
      _feature_maps[map_index][entity] = static_cast<int>(global_feature_number);

      if (_var_index_mode)
        _var_index_maps[map_index][entity] = feature._var_index;

      // Fill in the data structure that keeps track of all features per elem
      if (_compute_var_to_feature_map)
      {
        auto map_it = _entity_var_to_features.lower_bound(entity);
        if (map_it == _entity_var_to_features.end() || map_it->first != entity)
          map_it = _entity_var_to_features.emplace_hint(
              map_it, entity, std::vector<unsigned int>(_n_vars, invalid_id));
        map_it->second[feature._var_index] = feature._id;
      }
    }

    if (_compute_halo_maps)
      // Loop over the halo ids to update cells with halo information
      for (auto entity : feature._halo_ids)
        _halo_ids[map_index][entity] = static_cast<int>(global_feature_number);

    // Loop over the ghosted ids to update cells with ghost information
    for (auto entity : feature._ghosted_ids)
      _ghosted_entity_ids[entity] = 1;

    // TODO: Fixme
    if (!_global_numbering)
      mooseError("Local numbering currently disabled");
```

```
  }
}

void
FeatureFloodCount::flood(const DofObject * dof_object,
                         std::size_t current_index,
                         FeatureData * feature)
{
  if (dof_object == nullptr)
    return;

  // Retrieve the id of the current entity
  auto entity_id = dof_object->id();

  // Has this entity already been marked? - if so move along
  if (_entities_visited[current_index].find(entity_id) != _entities_visited[current_index].end())
    return;

  // See if the current entity either starts a new feature or continues an existing feature
  auto new_id = invalid_id; // Writable reference to hold an optional id;
  Status status =
      Status::INACTIVE; // Status is inactive until we find an entity above the starting threshold
  if (!isNewFeatureOrConnectedRegion(dof_object, current_index, feature, status, new_id))
    return;

  /**
   * If we reach this point (i.e. we haven't returned early from this routine),
   * we've found a new mesh entity that's part of a feature. We need to mark
   * the entity as visited at this point (and not before!) to avoid infinite
   * recursion. If you mark the node too early you risk not coloring in a whole
   * feature any time a "connecting threshold" is used since we may have
   * already visited this entity earlier but it was in-between two thresholds.
   */
  _entities_visited[current_index][entity_id] = true;

  auto map_num = _single_map_mode ? decltype(current_index)(0) : current_index;

  // New Feature (we need to create it and add it to our data structure)
  if (!feature)
  {
    _partial_feature_sets[map_num].emplace_back(
        current_index, _feature_count++, processor_id(), status);

    // Get a handle to the feature we will update (always the last feature in the data structure)
    feature = &_partial_feature_sets[map_num].back();

    // If new_id is valid, we'll set it in the feature here.
    if (new_id != invalid_id)
      feature->_id = new_id;
  }

  // Insert the current entity into the local ids map
  feature->_local_ids.insert(entity_id);

  /**
   * See if this particular entity cell contributes to the centroid calculation. We
   * only deal with elemental floods and only count it if it's owned by the current
   * processor to avoid skewing the result.
   */
  if (_is_elemental && processor_id() == dof_object->processor_id())
  {
    const Elem * elem = static_cast<const Elem *>(dof_object);

    // Keep track of how many elements participate in the centroid averaging
    feature->_vol_count++;

    // Sum the centroid values for now, we'll average them later
    feature->_centroid += elem->centroid();

    // Does the volume intersect the boundary?
    if (_all_boundary_entity_ids.find(elem->id()) != _all_boundary_entity_ids.end())
      feature->_intersects_boundary = true;
  }

  if (_is_elemental)
    visitElementalNeighbors(static_cast<const Elem *>(dof_object),
                            current_index,
                            feature,
                            /*expand_halos_only =*/false);
  else
    visitNodalNeighbors(static_cast<const Node *>(dof_object),
                        current_index,
                        feature,
                        /*expand_halos_only =*/false);
}

Real FeatureFloodCount::getThreshold(std::size_t /*current_index*/) const
{
  return _step_threshold;
}

Real FeatureFloodCount::getConnectingThreshold(std::size_t /*current_index*/) const
{
  return _step_connecting_threshold;
}

bool
FeatureFloodCount::compareValueWithThreshold(Real entity_value, Real threshold) const
{
  return ((_use_less_than_threshold_comparison && (entity_value >= threshold)) ||
          (!_use_less_than_threshold_comparison && (entity_value <= threshold)));
}

bool
FeatureFloodCount::isNewFeatureOrConnectedRegion(const DofObject * dof_object,
                                                 std::size_t current_index,
                                                 FeatureData *& feature,
                                                 Status & status,
                                                 unsigned int & /*new_id*/)
{
  // Get the value of the current variable for the current entity
  Real entity_value;
  if (_is_elemental)
  {
    const Elem * elem = static_cast<const Elem *>(dof_object);
    std::vector<Point> centroid(1, elem->centroid());
    _subproblem.reinitElemPhys(elem, centroid, 0);
    entity_value = _vars[current_index]->sln()[0];
  }
  else
    entity_value = _vars[current_index]->getNodalValue(*static_cast<const Node *>(dof_object));

  // If the value compares against our starting threshold, this is definitely part of a feature
  // we'll keep
  if (compareValueWithThreshold(entity_value, getThreshold(current_index)))
  {
    Status * status_ptr = &status;

    if (feature)
      status_ptr = &feature->_status;

    // Update an existing feature's status or clear the flag on the passed in status
    *status_ptr &= ~Status::INACTIVE;
    return true;
  }

  /**
   * If the value is _only_ above the connecting threshold, it's still part of a feature but
   * possibly part of one that we'll discard if there is never any starting threshold encountered.
   */
  return compareValueWithThreshold(entity_value, getConnectingThreshold(current_index));
}

void
FeatureFloodCount::visitElementalNeighbors(const Elem * elem,
                                           std::size_t current_index,
                                           FeatureData * feature,
                                           bool expand_halos_only)
{
  mooseAssert(elem, "Elem is NULL");

  std::vector<const Elem *> all_active_neighbors;

  // Loop over all neighbors (at the the same level as the current element)
  for (auto i = decltype(elem->n_neighbors())(0); i < elem->n_neighbors(); ++i)
  {
    const Elem * neighbor_ancestor = elem->neighbor(i);
    if (neighbor_ancestor)
      /**
       * Retrieve only the active neighbors for each side of this element, append them to the list
       * of active neighbors
       */
      neighbor_ancestor->active_family_tree_by_neighbor(all_active_neighbors, elem, false);
  }

  visitNeighborsHelper(elem, all_active_neighbors, current_index, feature, expand_halos_only);
}

```

```
void
FeatureFloodCount::visitNodalNeighbors(const Node * node,
                                       std::size_t current_index,
                                       FeatureData * feature,
                                       bool expand_halos_only)
{
  mooseAssert(node, "Node is NULL");

  std::vector<const Node *> all_active_neighbors;
  MeshTools::find_nodal_neighbors(_mesh.getMesh(), *node, _nodes_to_elem_map, all_active_neighbors);

  visitNeighborsHelper(node, all_active_neighbors, current_index, feature, expand_halos_only);
}

template <typename T>
void
FeatureFloodCount::visitNeighborsHelper(const T * curr_entity,
                                        std::vector<const T *> neighbor_entities,
                                        std::size_t current_index,
                                        FeatureData * feature,
                                        bool expand_halos_only)
{
  // Loop over all active element neighbors
  for (const auto neighbor : neighbor_entities)
  {
    if (neighbor)
    {
      if (expand_halos_only)
        feature->_halo_ids.insert(neighbor->id());

      else
      {
        auto my_processor_id = processor_id();

        if (neighbor->processor_id() != my_processor_id)
          feature->_ghosted_ids.insert(curr_entity->id());

        /**
         * Only recurse where we own this entity. We might step outside of the
         * ghosted region if we recurse where we don't own the current entity.
         */
        if (curr_entity->processor_id() == my_processor_id)
        {
          /**
           * Premark neighboring entities with a halo mark. These
           * entities may or may not end up being part of the feature.
           * We will not update the _entities_visited data structure
           * here.
           */
          feature->_halo_ids.insert(neighbor->id());

          flood(neighbor, current_index, feature);
        }
      }
    }
  }
}

void
FeatureFloodCount::appendPeriodicNeighborNodes(FeatureData & data) const
{
  if (_is_elemental)
  {
    for (auto entity : data._local_ids)
    {
      Elem * elem = _mesh.elemPtr(entity);

      for (auto node_n = decltype(elem->n_nodes())(0); node_n < elem->n_nodes(); ++node_n)
      {
        auto iters = _periodic_node_map.equal_range(elem->node(node_n));

        for (auto it = iters.first; it != iters.second; ++it)
        {
          data._periodic_nodes.insert(it->first);
          data._periodic_nodes.insert(it->second);
        }
      }
    }
  }
  else
  {
    for (auto entity : data._local_ids)
    {
      auto iters = _periodic_node_map.equal_range(entity);

      for (auto it = iters.first; it != iters.second; ++it)
      {
        data._periodic_nodes.insert(it->first);
        data._periodic_nodes.insert(it->second);
      }
    }
  }
}

void
FeatureFloodCount::FeatureData::updateBBoxExtremes(MeshTools::BoundingBox & bbox,
                                                   const Point & node)
{
  for (unsigned int i = 0; i < LIBMESH_DIM; ++i)
  {
    bbox.min()(i) = std::min(bbox.min()(i), node(i));
    bbox.max()(i) = std::max(bbox.max()(i), node(i));
  }
}

void
FeatureFloodCount::FeatureData::updateBBoxExtremes(MeshTools::BoundingBox & bbox, const Elem & elem)
{
  for (auto node_n = decltype(elem.n_nodes())(0); node_n < elem.n_nodes(); ++node_n)
    updateBBoxExtremes(bbox, *(elem.get_node(node_n)));
}

```

```
void
FeatureFloodCount::FeatureData::updateBBoxExtremes(MeshTools::BoundingBox & bbox,
                                                   const MeshTools::BoundingBox & rhs_bbox)
{
  for (unsigned int i = 0; i < LIBMESH_DIM; ++i)
  {
    bbox.min()(i) = std::min(bbox.min()(i), rhs_bbox.min()(i));
    bbox.max()(i) = std::max(bbox.max()(i), rhs_bbox.max()(i));
  }
}

bool
FeatureFloodCount::FeatureData::boundingBoxesIntersect(const FeatureData & rhs) const
{
  // See if any of the bounding boxes in either FeatureData object intersect
  for (const auto & bbox_lhs : _bboxes)
    for (const auto & bbox_rhs : rhs._bboxes)
      if (bbox_lhs.intersects(bbox_rhs))
        return true;

  return false;
}

bool
FeatureFloodCount::FeatureData::halosIntersect(const FeatureData & rhs) const
{
  return setsIntersect(
      _halo_ids.begin(), _halo_ids.end(), rhs._halo_ids.begin(), rhs._halo_ids.end());
}

bool
FeatureFloodCount::FeatureData::periodicBoundariesIntersect(const FeatureData & rhs) const
{
  return setsIntersect(_periodic_nodes.begin(),
                       _periodic_nodes.end(),
                       rhs._periodic_nodes.begin(),
                       rhs._periodic_nodes.end());
}

bool
FeatureFloodCount::FeatureData::ghostedIntersect(const FeatureData & rhs) const
{
  return setsIntersect(
      _ghosted_ids.begin(), _ghosted_ids.end(), rhs._ghosted_ids.begin(), rhs._ghosted_ids.end());
}

void
FeatureFloodCount::FeatureData::merge(FeatureData && rhs)
{
  mooseAssert(_var_index == rhs._var_index, "Mismatched variable index in merge");
  mooseAssert(_id == rhs._id, "Mismatched auxiliary id in merge");

  std::set<dof_id_type> set_union;

  /**
   * Even though we've determined that these two partial regions need to be merged, we don't
   * necessarily know if the _ghost_ids intersect. We could be in this branch because the periodic
   * boundaries intersect but that doesn’t tell us anything about whether or not the ghost_region
   * also intersects. If the _ghost_ids intersect, that means that we are merging along a periodic
   * boundary, not across one. In this case the bounding box(s) need to be expanded.
   */
  std::set_union(_periodic_nodes.begin(),
                 _periodic_nodes.end(),
                 rhs._periodic_nodes.begin(),
                 rhs._periodic_nodes.end(),
                 std::insert_iterator<std::set<dof_id_type>>(set_union, set_union.begin()));
  _periodic_nodes.swap(set_union);

  set_union.clear();
  std::set_union(_local_ids.begin(),
                 _local_ids.end(),
                 rhs._local_ids.begin(),
                 rhs._local_ids.end(),
                 std::insert_iterator<std::set<dof_id_type>>(set_union, set_union.begin()));
  _local_ids.swap(set_union);

  set_union.clear();
  std::set_union(_halo_ids.begin(),
                 _halo_ids.end(),
                 rhs._halo_ids.begin(),
                 rhs._halo_ids.end(),
                 std::insert_iterator<std::set<dof_id_type>>(set_union, set_union.begin()));
  _halo_ids.swap(set_union);

  set_union.clear();
  std::set_union(_ghosted_ids.begin(),
                 _ghosted_ids.end(),
                 rhs._ghosted_ids.begin(),
                 rhs._ghosted_ids.end(),
                 std::insert_iterator<std::set<dof_id_type>>(set_union, set_union.begin()));

  // Was there overlap in the physical region?
  bool physical_intersection = (_ghosted_ids.size() + rhs._ghosted_ids.size() > set_union.size());
  _ghosted_ids.swap(set_union);

  /**
   * If we had a physical intersection, we need to expand boxes. If we had a virtual (periodic)
   * intersection we need to preserve all of the boxes from each of the regions’ sets.
   */
  if (physical_intersection)
    expandBBox(rhs);
  else
    std::move(rhs._bboxes.begin(), rhs._bboxes.end(), std::back_inserter(_bboxes));

  // Keep track of the original ids so we can notify other processors of the local to global mapping
  _orig_ids.splice(_orig_ids.end(), std::move(rhs._orig_ids));

  // Update the min feature id
  _min_entity_id = std::min(_min_entity_id, rhs._min_entity_id);

  /**
   * Combine the status flags: Currently we only expect to combine CLEAR and INACTIVE. Any other
   * combination is currently a logic error. In this case of CLEAR and INACTIVE though,
   * we want to make sure that CLEAR wins.
   */
  mooseAssert((_status & Status::MARKED & Status::DIRTY) == Status::CLEAR,
              "Flags in invalid state");

  // Logical AND here to combine flags (INACTIVE & INACTIVE == INACTIVE, all other combos are CLEAR)
  _status &= rhs._status;

  _vol_count += rhs._vol_count;
  _centroid += rhs._centroid;
}

void
FeatureFloodCount::FeatureData::clear()
{
  _local_ids.clear();
  _periodic_nodes.clear();
  _halo_ids.clear();
  _ghosted_ids.clear();
  _bboxes.clear();
  _orig_ids.clear();
}

void
FeatureFloodCount::FeatureData::expandBBox(const FeatureData & rhs)
{
  std::vector<bool> intersected_boxes(rhs._bboxes.size(), false);

  auto box_expanded = false;
  for (auto & bbox : _bboxes)
    for (auto j = beginIndex(rhs._bboxes); j < rhs._bboxes.size(); ++j)
      if (bbox.intersects(rhs._bboxes[j]))
      {
        updateBBoxExtremes(bbox, rhs._bboxes[j]);
        intersected_boxes[j] = true;
        box_expanded = true;
      }

  // Any bounding box in the rhs vector that doesn't intersect
  // needs to be appended to the lhs vector
  for (auto j = beginIndex(intersected_boxes); j < intersected_boxes.size(); ++j)
    if (!intersected_boxes[j])
      _bboxes.push_back(rhs._bboxes[j]);

  // Error check
  if (!box_expanded)
  {
    std::ostringstream oss;
    oss << "LHS BBoxes:\n";
    for (auto i = beginIndex(_bboxes); i < _bboxes.size(); ++i)
      oss << "Max: " << _bboxes[i].max() << " Min: " << _bboxes[i].min() << '\n';

    oss << "RHS BBoxes:\n";
    for (auto i = beginIndex(rhs._bboxes); i < rhs._bboxes.size(); ++i)
      oss << "Max: " << rhs._bboxes[i].max() << " Min: " << rhs._bboxes[i].min() << '\n';

    ::mooseError("No Bounding Boxes Expanded - This is a catastrophic error!\n", oss.str());
  }
}

std::ostream &
operator<<(std::ostream & out, const FeatureFloodCount::FeatureData & feature)
{
  static const bool debug = true;

  out << "Grain ID: ";
  if (feature._id != FeatureFloodCount::invalid_id)
    out << feature._id;
  else
    out << "invalid";

  if (debug)
  {
    out << "\nGhosted Entities: ";
    for (auto ghosted_id : feature._ghosted_ids)
      out << ghosted_id << " ";

    out << "\nLocal Entities: ";
    for (auto local_id : feature._local_ids)
      out << local_id << " ";

    out << "\nHalo Entities: ";
    for (auto halo_id : feature._halo_ids)
      out << halo_id << " ";

    out << "\nPeriodic Node IDs: ";
    for (auto periodic_node : feature._periodic_nodes)
      out << periodic_node << " ";
  }

  out << "\nBBoxes:";
  Real volume = 0;
  for (const auto & bbox : feature._bboxes)
  {
    out << "\nMax: " << bbox.max() << " Min: " << bbox.min();
    volume += (bbox.max()(0) - bbox.min()(0)) * (bbox.max()(1) - bbox.min()(1)) *
              (MooseUtils::absoluteFuzzyEqual(bbox.max()(2), bbox.min()(2))
                   ? 1
                   : bbox.max()(2) - bbox.min()(2));
  }

  out << "\nStatus: ";
  if (feature._status == FeatureFloodCount::Status::CLEAR)
    out << "CLEAR";
  if (static_cast<bool>(feature._status & FeatureFloodCount::Status::MARKED))
    out << " MARKED";
  if (static_cast<bool>(feature._status & FeatureFloodCount::Status::DIRTY))
    out << " DIRTY";
  if (static_cast<bool>(feature._status & FeatureFloodCount::Status::INACTIVE))
    out << " INACTIVE";

  if (debug)
  {
    out << "\nOrig IDs (rank, index): ";
    for (const auto & orig_pair : feature._orig_ids)
      out << '(' << orig_pair.first << ", " << orig_pair.second << ") ";
    out << "\nVar_index: " << feature._var_index;
    out << "\nMin Entity ID: " << feature._min_entity_id;
  }
  out << "\n\n";

  return out;
}

const std::size_t FeatureFloodCount::invalid_size_t = std::numeric_limits<std::size_t>::max();
const unsigned int FeatureFloodCount::invalid_id = std::numeric_limits<unsigned int>::max();
```

A.3 GrainTracker.h

```cpp
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/

#ifndef GRAINTRACKER_H
#define GRAINTRACKER_H

#include "FeatureFloodCount.h"
#include "GrainTrackerInterface.h"

// libMesh includes
#include "libmesh/mesh_tools.h"

class GrainTracker;
class EBSDReader;
struct GrainDistance;

template <>
InputParameters validParams<GrainTracker>();

class GrainTracker : public FeatureFloodCount, public GrainTrackerInterface
{
public:
  GrainTracker(const InputParameters & parameters);
  virtual ~GrainTracker();

  virtual void initialize() override;
  virtual void execute() override;
  virtual void finalize() override;

  virtual std::size_t getTotalFeatureCount() const override;

  // Struct used to transfer minimal data to all ranks
  struct PartialFeatureData
  {
    bool intersects_boundary;
    unsigned int id;
    Point centroid;
    Status status;
  };

  struct CacheValues
  {
    Real current;
    Real old;
    Real older;
  };

  enum class RemapCacheMode
  {
    FILL,
    USE,
    BYPASS
  };

  // GrainTrackerInterface methods
  virtual Real getEntityValue(dof_id_type node_id,
                              FieldType field_type,
                              std::size_t var_index = 0) const override;
  virtual const std::vector<unsigned int> &
  getVarToFeatureVector(dof_id_type elem_id) const override;
  virtual unsigned int getFeatureVar(unsigned int feature_id) const override;
  virtual std::size_t getNumberActiveGrains() const override;
  virtual Point getGrainCentroid(unsigned int grain_id) const override;
  virtual bool doesFeatureIntersectBoundary(unsigned int feature_id) const override;
  virtual std::vector<unsigned int> getNewGrainIDs() const override;

protected:
  virtual void updateFieldInfo() override;
  virtual Real getThreshold(std::size_t current_index) const override;
  virtual bool isNewFeatureOrConnectedRegion(const DofObject * dof_object,
                                             std::size_t current_index,
                                             FeatureData *& feature,
                                             Status & status,
                                             unsigned int & new_id) override;

  void communicateHaloMap();

  /**
   * When the tracking phase starts (_t_step == _tracking_step) it assigns a unique id to every
   * FeatureData object found by the FeatureFloodCount object. If an EBSDReader is linked into
   * the GrainTracker the information from the reader is used to assign grain information,
   * otherwise it's ordered by each Feature's "minimum entity id" and assigned a non-negative
   * integer.
   */
  void assignGrains();

  /**
   * On subsequent time_steps, incoming FeatureData objects are compared to previous time_step
   * information to track grains between time steps.
   *
   * This method updates the _feature_sets data structure.
   * This method should only be called on the root processor
   */
  void trackGrains();

  /**
   * This method is called when a new grain is detected. It can be overridden by a derived class to
   * handle setting new properties on the newly created grain.
   */
  virtual void newGrainCreated(unsigned int new_grain_id);

  /**
   * This method is called after trackGrains to remap grains that are too close to each other.
   */
  void remapGrains();

  /**
   * Broadcast essential Grain information to all processors. This method is used to get certain
   * attributes like centroids distributed and whether or not a grain intersects a boundary updated.
   */
  void broadcastAndUpdateGrainData();

  /**
   * Populates and sorts a min_distances vector with the minimum distances to all grains in the
   * simulation for a given grain. There are _vars.size() entries in the outer vector, one for
   * each order parameter. A list of grains with the same OP are ordered in lists per OP.
   */
  void computeMinDistancesFromGrain(FeatureData & grain,
                                    std::vector<std::list<GrainDistance>> & min_distances);

  /**
   * This is the recursive part of the remapping algorithm. It attempts to remap a grain to a new
   * index and recurses until max_depth is reached.
   */
  bool attemptGrainRenumber(FeatureData & grain, unsigned int depth, unsigned int max_depth);

  /**
   * A routine for moving all of the solution values from a given grain to a new variable number. It
   * is called with different modes to only cache, or actually do the work, or bypass the cache
   * altogether.
   */
  void swapSolutionValues(FeatureData & grain,
                          std::size_t new_var_index,
                          std::vector<std::map<Node *, CacheValues>> & cache,
                          RemapCacheMode cache_mode);

  /**
   * Helper method for actually performing the swaps.
   */
  void swapSolutionValuesHelper(Node * curr_node,
                                std::size_t curr_var_index,
                                std::size_t new_var_index,
                                std::vector<std::map<Node *, CacheValues>> & cache,
                                RemapCacheMode cache_mode);

  /**
   * This method returns the minimum periodic distance between two vectors of bounding boxes. If the
   * bounding boxes overlap the result is always -1.0.
   */
  Real boundingRegionDistance(std::vector<MeshTools::BoundingBox> & bboxes1,
                              std::vector<MeshTools::BoundingBox> & bboxes2) const;

  /**
   * This method returns the minimum periodic distance between the centroids of two vectors of
   * bounding boxes.
   */
  Real centroidRegionDistance(std::vector<MeshTools::BoundingBox> & bboxes1,
                              std::vector<MeshTools::BoundingBox> & bboxes2) const;

  /**
   * This method takes all of the partial features and expands the local, ghosted, and halo sets
   * around those regions to account for the diffuse interface. Rather than using any kind of
   * recursion here, we simply expand the region by all "point" neighbors from the actual
   * grain cells since all point neighbors will contain contributions to the region.
   */
  void expandEBSDGrains();

  /**
   * This method colors neighbors of halo entries to expand the halo as desired for a given
   * simulation.
   */
  void expandHalos(unsigned int num_layers_to_expand);

  /**
   * Retrieve the next unique grain number if a new grain is detected during trackGrains. This
   * method handles reserve order parameter indices properly. Direct access to the next index
   * should be avoided.
   */
  unsigned int getNextUniqueID();

  /*************************************************
   *************** Data Structures *****************
   *************************************************/

  /// The timestep to begin tracking grains
  const int _tracking_step;

  /// The thickness of the halo surrounding each grain
  const unsigned int _halo_level;

  /// Depth of renumbering recursion (a depth of zero means no recursion)
  static const unsigned int _max_renumbering_recursion = 4;

  /// The number of reserved order parameters
  const unsigned short _n_reserve_ops;

  /// The cutoff index where if variable index >= this number, no remapping TO that variable
  /// will occur
  const std::size_t _reserve_op_index;

  /// The threshold above (or below) where a grain may be found on a reserve op field
  const Real _reserve_op_threshold;

  /// Inidicates whether remapping should be done or not (remapping is independent of tracking)
  const bool _remap;

  /// A reference to the nonlinear system (used for retrieving solution vectors)
  NonlinearSystemBase & _nl;

  /**
   * This data structure holds the map of unique grains from the previous time step.
   * The information is updated each timestep to track grains over time.
   */
  std::vector<FeatureData> & _feature_sets_old;

  /// Optional ESBD Reader
  const EBSDReader * _ebsd_reader;

  /// Optional EBSD OP variable pointer (required if EBSD is supplied)
  MooseVariable * _ebsd_op_var;

  /// The phase to retrieve EBSD information from
  const unsigned int _phase;

  /// Boolean to indicate that we should retrieve EBSD information from a specific phase
  const bool _consider_phase;

  /**
   * Boolean to indicate the first time this object executes.
   * Note: _tracking_step isn't enough if people skip initial or execute more than once per step.
   */
  bool _first_time;

  /**
   * Boolean to terminate with an error if a new grain is created during the simulation.
   * This is for simulations where new grains are not expected. Note, this does not impact
   * the initial callback to newGrainCreated() nor does it get triggered for splitting grains.
   */
  bool _error_on_grain_creation;

private:
  /// Holds the first unique grain index when using _reserve_op (all the remaining indices are sequential)
  unsigned int _reserve_grain_first_index;

  /// The previous max grain id (needed to figure out which ids are new in a given step)
  unsigned int _old_max_grain_id;

  /// Holds the next "regular" grain ID (a grain found or remapped to the standard op vars)
  unsigned int _max_curr_grain_id;

  /// Boolean to indicate whether this is a Steady or Transient solve
  const bool _is_transient;
};

/**
 * This struct is used to hold distance information to other grains in the simulation. It is used
 * for sorting and during the remapping algorithm.
 */
struct GrainDistance
{
  GrainDistance(Real distance, std::size_t var_index);

  GrainDistance(Real distance,
                std::size_t var_index,
                std::size_t grain_index,
                unsigned int grain_id);

  // Copy constructors
  GrainDistance(const GrainDistance & f) = default;
  GrainDistance & operator=(const GrainDistance & f) = default;

  // Move constructors
  GrainDistance(GrainDistance && f) = default;
  GrainDistance & operator=(GrainDistance && f) = default;

  bool operator<(const GrainDistance & rhs) const;

  Real _distance;
  std::size_t _var_index;
  std::size_t _grain_index;
  unsigned int _grain_id;
};

template <>
void dataStore(std::ostream & stream, GrainTracker::PartialFeatureData & feature, void * context);
template <>
void dataLoad(std::istream & stream, GrainTracker::PartialFeatureData & feature, void * context);

#endif
```

A.4 GrainTracker.C

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/

// MOOSE includes
#include "EBSDReader.h"
#include "GeneratedMesh.h"
#include "GrainTracker.h"
#include "MooseMesh.h"
#include "NonlinearSystem.h"

// LibMesh includes
#include "libmesh/periodic_boundary_base.h"

#include <algorithm>
#include <limits>
#include <numeric>

template <>
void
dataStore(std::ostream & stream, GrainTracker::PartialFeatureData & feature, void * context)
{
  storeHelper(stream, feature.intersects_boundary, context);
  storeHelper(stream, feature.id, context);
  storeHelper(stream, feature.centroid, context);
  storeHelper(stream, feature.status, context);
}

template <>
void
dataLoad(std::istream & stream, GrainTracker::PartialFeatureData & feature, void * context)
{
  loadHelper(stream, feature.intersects_boundary, context);
  loadHelper(stream, feature.id, context);
  loadHelper(stream, feature.centroid, context);
  loadHelper(stream, feature.status, context);
}

template <>
InputParameters
validParams<GrainTracker>()
{
  InputParameters params = validParams<FeatureFloodCount>();
  params += validParams<GrainTrackerInterface>();
  params.addClassDescription("Grain Tracker object for running reduced order parameter simulations "
                             "without grain coalescence.");

  return params;
}

GrainTracker::GrainTracker(const InputParameters & parameters)
```

```
  : FeatureFloodCount(parameters),
    GrainTrackerInterface(),
    _tracking_step(getParam<int>("tracking_step")),
    _halo_level(getParam<unsigned int>("halo_level")),
    _n_reserve_ops(getParam<unsigned short>("reserve_op")),
    _reserve_op_index(_n_reserve_ops <= _n_vars ? _n_vars - _n_reserve_ops : 0),
    _reserve_op_threshold(getParam<Real>("reserve_op_threshold")),
    _remap(getParam<bool>("remap_grains")),
    _nl(_fe_problem.getNonlinearSystemBase()),
    _feature_sets_old(declareRestartableData<std::vector<FeatureData>>("unique_grains")),
    _ebsd_reader(parameters.isParamValid("ebsd_reader") ? &getUserObject<EBSDReader>("ebsd_reader")
                                                        : nullptr),
    _ebsd_op_var(_ebsd_reader
                     ? &_fe_problem.getVariable(0, getParam<std::string>("var_name_base") + "_op")
                     : nullptr),
    _phase(isParamValid("phase") ? getParam<unsigned int>("phase") : 0),
    _consider_phase(isParamValid("phase")),
    _first_time(true),
    _error_on_grain_creation(getParam<bool>("error_on_grain_creation")),
    _reserve_grain_first_index(0),
    _old_max_grain_id(0),
    _max_curr_grain_id(0),
    _is_transient(_subproblem.isTransient())
{
  if (_ebsd_reader && !_ebsd_op_var)
    mooseError("EBSD OP variable must be supplied if the reader is supplied");
}

GrainTracker::~GrainTracker() {}

Real
GrainTracker::getEntityValue(dof_id_type entity_id,
                             FieldType field_type,
                             std::size_t var_index) const
{
  if (_t_step < _tracking_step)
    return 0;

  return FeatureFloodCount::getEntityValue(entity_id, field_type, var_index);
}

const std::vector<unsigned int> &
GrainTracker::getVarToFeatureVector(dof_id_type elem_id) const
{
  return FeatureFloodCount::getVarToFeatureVector(elem_id);
}

unsigned int
GrainTracker::getFeatureVar(unsigned int feature_id) const
{
  return FeatureFloodCount::getFeatureVar(feature_id);
}

std::size_t
GrainTracker::getNumberActiveGrains() const
{
  // Note: This value is parallel consistent, see FeatureFloodCount::communicateAndMerge()
  return _feature_count;
}

std::size_t
GrainTracker::getTotalFeatureCount() const
{
  // Note: This value is parallel consistent, see assignGrains()/trackGrains()
  return _max_curr_grain_id + 1;
}

Point
GrainTracker::getGrainCentroid(unsigned int grain_id) const
{
  mooseAssert(grain_id < _feature_id_to_local_index.size(), "Grain ID out of bounds");
  auto grain_index = _feature_id_to_local_index[grain_id];

  if (grain_index != invalid_size_t)
  {
    mooseAssert(_feature_id_to_local_index[grain_id] < _feature_sets.size(),
                "Grain index out of bounds");
    // Note: This value is parallel consistent, see GrainTracker::broadcastAndUpdateGrainData()
    return _feature_sets[_feature_id_to_local_index[grain_id]]._centroid;
  }

  // Inactive grain
  return Point();
}

bool
GrainTracker::doesFeatureIntersectBoundary(unsigned int feature_id) const
{
  // TODO: This data structure may need to be turned into a Multimap
  mooseAssert(feature_id < _feature_id_to_local_index.size(), "Grain ID out of bounds");

  auto feature_index = _feature_id_to_local_index[feature_id];
  if (feature_index != invalid_size_t)
  {
    mooseAssert(feature_index < _feature_sets.size(), "Grain index out of bounds");
    return _feature_sets[feature_index]._intersects_boundary;
  }

  return false;
}

void
GrainTracker::initialize()
{
  // Don't track grains if the current simulation step is before the specified tracking step
  if (_t_step < _tracking_step)
    return;

  /**
   * If we are passed the first time, we need to save the existing
   * grains before beginning the tracking on the current step. We'll do that
   * with a swap since the _feature_sets contents will be cleared anyway.
   */
  if (!_first_time)
    _feature_sets_old.swap(_feature_sets);

  FeatureFloodCount::initialize();
}

void
GrainTracker::execute()
{
  // Don't track grains if the current simulation step is before the specified tracking step
  if (_t_step < _tracking_step)
    return;

  Moose::perf_log.push("execute()", "GrainTracker");
  FeatureFloodCount::execute();
  Moose::perf_log.pop("execute()", "GrainTracker");
}

Real
GrainTracker::getThreshold(std::size_t var_index) const
{
  // If we are inspecting a reserve op parameter, we need to make sure
  // that there is an entity above the reserve_op threshold before
  // starting the flood of the feature.
  if (var_index >= _reserve_op_index)
    return _reserve_op_threshold;
  else
    return _step_threshold;
}

bool
GrainTracker::isNewFeatureOrConnectedRegion(const DofObject * dof_object,
                                            std::size_t current_index,
                                            FeatureData *& feature,
                                            Status & status,
                                            unsigned int & new_id)
{
  /**
   * When working with the EBSD reader we need to make sure that we get an accurate map
   * of the EBSD initial condition for the physics simulation to be correct. This is
   * incredibly difficult if we can only view the nodal interpolation of the elemental
   * EBSD data. Instead, we'll use the EBSD Reader data directly the first time this
   * object runs. Using EBSD data is only valid if we begin tracking in the zeroeth step
   */
  if (_ebsd_reader && _first_time)
  {
    mooseAssert(_t_step == 0, "EBSD only works if we begin in the initial condition");
    mooseAssert(_is_elemental, "EBSD only works with elemental grain tracker");

    /**
     * First inspect the order parameter assigned to the feature at this
     * element and see if it matches the current_index.
     */
    const Elem * elem = static_cast<const Elem *>(dof_object);
    unsigned int op = static_cast<unsigned int>(std::round(_ebsd_op_var->getElementalValue(elem)));
    if (current_index != op)
      return false;

    // Sample the EBSD Reader and retrieve the global_id or local_id and phase for the current
    // element
    std::vector<Point> centroid = {elem->centroid()};
    const EBSDAccessFunctors::EBSDPointData & d = _ebsd_reader->getData(centroid[0]);
    const auto phase = d._phase;

    // See if we are in a phase that we are actually tracking
    if (_consider_phase && phase != _phase)
      return false;

    // Get the ids from the EBSD reader
    const auto global_id = _ebsd_reader->getGlobalID(d._feature_id);
    const auto local_id = _ebsd_reader->getAvgData(global_id)._local_id;

    /**
     * If we don't have an active feature we'll need to populate new_id with the actual EBSD
     * grain number so that the flood routine will set it after creating the new feature.
     * We'll use that information when assigning the initial grain IDs.
     */
    if (!feature)
    {
      // Set the ID (EBSD ID)
      new_id = _consider_phase ? local_id : global_id;

      // EBSD Grains are _always_ kept
      status &= ~Status::INACTIVE;

      return true;
    }
    else
    {
      mooseAssert(feature->_id != invalid_id, "Expected EBSD ID missing");

      /**
       * If we have an active feature just make sure that the current active feature ID
       * matches the current entities EBSD local_id.
       */
      return feature->_id == (_consider_phase ? local_id : global_id);
    }
  }
  else
    // Just use normal variable inspection on subsequent steps
    return FeatureFloodCount::isNewFeatureOrConnectedRegion(
        dof_object, current_index, feature, status, new_id);
}

void
GrainTracker::finalize()
{
  /**
   * Some perf_log operations appear here instead of inside of the named routines
   * because of multiple return paths.
   */

  // Don't track grains if the current simulation step is before the specified tracking step
  if (_t_step < _tracking_step)
    return;

  Moose::perf_log.push("finalize()", "GrainTracker");

  // Expand the depth of the halos around all grains
  auto num_halo_layers = _halo_level >= 1
                             ? _halo_level - 1
                             : 0; // The first level of halos already exists so subtract one
  if (_ebsd_reader && _first_time)
  {
    expandEBSDGrains();

    /**
     * By expanding the EBSD Grains we've effectively erased one level of halo.
     * We'll just request one additional layer of halo this time around.
     */
    ++num_halo_layers;
  }
  expandHalos(num_halo_layers);

  // Build up the grain map on the root processor
  communicateAndMerge();

  /**
   * Assign or Track Grains
   */
  Moose::perf_log.push("trackGrains()", "GrainTracker");
  if (_first_time)
    assignGrains();
  else
    trackGrains();
  Moose::perf_log.pop("trackGrains()", "GrainTracker");
  _console << "Finished inside of trackGrains" << std::endl;

  /**
   * Broadcast essential data
   */
  broadcastAndUpdateGrainData();

  /**
   * Remap Grains
   */
  Moose::perf_log.push("remapGrains()", "GrainTracker");
  if (_remap)
    remapGrains();
  Moose::perf_log.pop("remapGrains()", "GrainTracker");

  updateFieldInfo();
  _console << "Finished inside of updateFieldInfo" << std::endl;

  // Set the first time flag false here (after all methods of finalize() have completed)
  _first_time = false;

  // TODO: Release non essential memory

  _console << "Finished inside of GrainTracker" << std::endl;
  Moose::perf_log.pop("finalize()", "GrainTracker");
}

void
GrainTracker::broadcastAndUpdateGrainData()
{
  std::vector<PartialFeatureData> root_feature_data;
  std::vector<std::string> send_buffer(1), recv_buffer;

  if (_is_master)
  {
    root_feature_data.reserve(_feature_sets.size());

    // Populate a subset of the information in a small data structure
    std::transform(_feature_sets.begin(),
                   _feature_sets.end(),
                   std::back_inserter(root_feature_data),
                   [](FeatureData & feature) {
                     PartialFeatureData partial_feature;
                     partial_feature.intersects_boundary = feature._intersects_boundary;
                     partial_feature.id = feature._id;
                     partial_feature.centroid = feature._centroid;
                     partial_feature.status = feature._status;
                     return partial_feature;
                   });

    std::ostringstream oss;
    dataStore(oss, root_feature_data, this);
    send_buffer[0].assign(oss.str());
  }

  // Broadcast the data to all ranks
  _communicator.broadcast_packed_range((void *)(nullptr),
                                       send_buffer.begin(),
                                       send_buffer.end(),
                                       (void *)(nullptr),
                                       std::back_inserter(recv_buffer));

  // Unpack and update
  if (!_is_master)
  {
    std::istringstream iss;
    iss.str(recv_buffer[0]);
    iss.clear();

    dataLoad(iss, root_feature_data, this);

    for (const auto & partial_data : root_feature_data)
    {
      // See if this processor has a record of this grain
      if (partial_data.id < _feature_id_to_local_index.size() &&
          _feature_id_to_local_index[partial_data.id] != invalid_size_t)
      {
        auto & grain = _feature_sets[_feature_id_to_local_index[partial_data.id]];
        grain._intersects_boundary = partial_data.intersects_boundary;
        grain._centroid = partial_data.centroid;
        if (partial_data.status == Status::INACTIVE)
          grain._status = Status::INACTIVE;
      }
    }
  }
}

void
GrainTracker::expandHalos(unsigned int num_layers_to_expand)
{
  if (num_layers_to_expand == 0)
    return;

  for (auto & list_ref : _partial_feature_sets)
  {
    for (auto & feature : list_ref)
    {
      for (auto halo_level = decltype(num_layers_to_expand)(0); halo_level < num_layers_to_expand;
           ++halo_level)
      {
        /**
         * Create a copy of the halo set so that as we insert new ids into the
         * set we don't continue to iterate on those new ids.
         */
        std::set<dof_id_type> orig_halo_ids(feature._halo_ids);

        for (auto entity : orig_halo_ids)
        {
          if (_is_elemental)
            visitElementalNeighbors(_mesh.elemPtr(entity),
                                    feature._var_index,
                                    &feature,
                                    /*expand_halos_only =*/true);
          else
            visitNodalNeighbors(_mesh.nodePtr(entity),
                                feature._var_index,
                                &feature,
                                /*expand_halos_only =*/true);
        }
      }
    }
  }
}

void
GrainTracker::expandEBSDGrains()
{
  mooseAssert(_t_step == 0, "EBSD only works if we begin in the initial condition");
  mooseAssert(_is_elemental, "EBSD only works with elemental grain tracker");

  const auto & node_to_elem_map = _mesh.nodeToActiveSemilocalElemMap();
  decltype(FeatureData::_local_ids) expanded_local_ids;
  auto my_processor_id = processor_id();

  /**
   * To expand the EBSD region to the actual flooded
   * region we need to add in all point neighbors of the
   * current local region for each feature. This is
   * because the variable influence spreads from the
   * EBSD data out exactly one element from every
   * point by design (elem_to_node_weight_map)
   */
  for (auto & list_ref : _partial_feature_sets)
  {
    for (auto & feature : list_ref)
    {
      expanded_local_ids.clear();

      for (auto entity : feature._local_ids)
      {
        const Elem * elem = _mesh.elemPtr(entity);
        mooseAssert(elem, "elem pointer is NULL");

        // Get the nodes on a current element so that we can add in point neighbors
        auto n_nodes = elem->n_vertices();
        for (auto i = decltype(n_nodes)(0); i < n_nodes; ++i)
        {
          const Node * current_node = elem->get_node(i);

          auto elem_vector_it = node_to_elem_map.find(current_node->id());
          if (elem_vector_it == node_to_elem_map.end())
            mooseError("Error in node to elem map");

          const auto & elem_vector = elem_vector_it->second;

          expanded_local_ids.insert(elem_vector.begin(), elem_vector.end());

          // Now see which elements need to go into the ghosted set
          for (auto entity : elem_vector)
          {
            const Elem * neighbor = _mesh.elemPtr(entity);
            mooseAssert(neighbor, "neighbor pointer is NULL");

            if (neighbor->processor_id() != my_processor_id)
              feature._ghosted_ids.insert(elem->id());
          }
        }
      }

      // Replace the existing local ids with the expanded local ids
      feature._local_ids.swap(expanded_local_ids);

      // Copy the expanded local_ids into the halo_ids container
      feature._halo_ids = feature._local_ids;
    }
  }
}

void
```

```
GrainTracker::assignGrains()
{
  mooseAssert(_first_time, "assignGrains may only be called on the first tracking step");

  /**
   * When using the EBSD reader, the grain IDs will already be assigned. We'll
   * use that information to sort the grains. Otherwise, we'll use the default
   * sorting that doesn't require grainIDs (relies on _min_entity_id and _var_index).
   * These will be the unique grain numbers that we must track for
   * the remainder of the simulation.
   */
  if (_is_master)
  {
    mooseAssert(!_feature_sets.empty(), "Feature sets empty!");

    // Find the largest grain ID, this requires sorting if the ID is not already set
    if (_ebsd_reader)
    {
      auto grain_num =
          _consider_phase ? _ebsd_reader->getGrainNum(_phase) : _ebsd_reader->getGrainNum();
      _max_curr_grain_id = grain_num - 1;
    }
    else
    {
      sortAndLabel();
      _max_curr_grain_id = _feature_sets[_feature_sets.size() - 1]._id;
    }

    for (auto & grain : _feature_sets)
      grain._status = Status::MARKED; // Mark the grain

    // Set up the first reserve grain index based on the largest grain ID
    _reserve_grain_first_index = _max_curr_grain_id + 1;
  } // is_master

  /*************************************************************
   ****************** COLLECTIVE WORK SECTION ******************
   *************************************************************/

  // Make IDs on all non-master ranks consistent
  scatterAndUpdateRanks();

  // Build up an id to index map
  _communicator.broadcast(_max_curr_grain_id);
  buildFeatureIdToLocalIndices(_max_curr_grain_id);

  // Now trigger the newGrainCreated() callback on all ranks
  for (auto new_id = decltype(_max_curr_grain_id)(0); new_id <= _max_curr_grain_id; ++new_id)
    newGrainCreated(new_id);
}

void
GrainTracker::trackGrains()
{
  mooseAssert(!_first_time, "Track grains may only be called when _tracking_step > _t_step");

  // Used to track indices for which to trigger the new grain callback on (used on all ranks)
  auto _old_max_grain_id = _max_curr_grain_id;

  /**
   * Only the master rank does tracking, the remaining ranks
   * wait to receive local to global indices from the master.
   */
  if (_is_master)
  {
    // Reset Status on active unique grains
    std::vector<unsigned int> map_sizes(_maps_size);
    for (auto & grain : _feature_sets_old)
    {
      if (grain._status != Status::INACTIVE)
      {
        grain._status = Status::CLEAR;
        map_sizes[grain._var_index]++;
      }
    }

    // Print out stats on overall tracking changes per var_index
    for (auto map_num = decltype(_maps_size)(0); map_num < _maps_size; ++map_num)
    {
      _console << "\nGrains active index " << map_num << ": " << map_sizes[map_num] << " -> "
               << _feature_counts_per_map[map_num];
      if (map_sizes[map_num] > _feature_counts_per_map[map_num])
        _console << "--";
      else if (map_sizes[map_num] < _feature_counts_per_map[map_num])
        _console << "++";
    }
    _console << '\n' << std::endl;

    /**
     * To track grains across time steps, we will loop over our unique grains and link each one up
     * with one of our new unique grains. The criteria for doing this will be to find the unique
     * grain in the new list with a matching variable index whose centroid is closest to this
     * unique grain.
     */
    std::vector<std::size_t> new_grain_index_to_existing_grain_index(_feature_sets.size(),
                                                                     invalid_size_t);

    for (auto old_grain_index = beginIndex(_feature_sets_old);
         old_grain_index < _feature_sets_old.size();
         ++old_grain_index)
    {
      auto & old_grain = _feature_sets_old[old_grain_index];

      if (old_grain._status == Status::INACTIVE) // Don't try to find matches for inactive grains
        continue;

      std::size_t closest_match_index = invalid_size_t;
      Real min_centroid_diff = std::numeric_limits<Real>::max();

      /**
       * The _feature_sets vector is constructed by _var_index so we can avoid looping over all
       * indices. We can quickly jump to the first matching index to reduce the number of
       * comparisons and terminate our loop when our variable index stops matching.
       */
      // clang-format off
      auto start_it =
          std::lower_bound(_feature_sets.begin(), _feature_sets.end(), old_grain._var_index,
                           [](const FeatureData & item, std::size_t var_index)
                           {
                             return item._var_index < var_index;
                           });
      // clang-format on

      // We only need to examine grains that have matching variable indices
      for (decltype(_feature_sets.size()) new_grain_index =
               std::distance(_feature_sets.begin(), start_it);
           new_grain_index < _feature_sets.size() &&
           _feature_sets[new_grain_index]._var_index == old_grain._var_index;
           ++new_grain_index)
      {
        auto & new_grain = _feature_sets[new_grain_index];

        /**
         * Don't try to do any matching unless the bounding boxes at least overlap. This is to avoid
         * the corner case of having a grain split and a grain disappear during the same time step!
         */
        if (new_grain.boundingBoxesIntersect(old_grain))
        {
          Real curr_centroid_diff = centroidRegionDistance(old_grain._bboxes, new_grain._bboxes);
          if (curr_centroid_diff <= min_centroid_diff)
          {
            closest_match_index = new_grain_index;
            min_centroid_diff = curr_centroid_diff;
          }
        }
      }

      // found a match
      if (closest_match_index != invalid_size_t)
      {
        /**
         * It's possible that multiple existing grains will map to a single new grain (indicated by
         * finding multiple matches when we are building this map). This will happen any time a
         * grain disappears during this time step. We need to figure out the rightful owner in this
         * case and inactivate the old grain.
         */
        auto curr_index = new_grain_index_to_existing_grain_index[closest_match_index];
        if (curr_index != invalid_size_t)
        {
          // The new feature being competed for
          auto & new_grain = _feature_sets[closest_match_index];

          // The other old grain competing to match up to the same new grain
          auto & other_old_grain = _feature_sets_old[curr_index];

          auto centroid_diff1 = centroidRegionDistance(new_grain._bboxes, old_grain._bboxes);
          auto centroid_diff2 = centroidRegionDistance(new_grain._bboxes, other_old_grain._bboxes);

          auto & inactive_grain = (centroid_diff1 < centroid_diff2) ? other_old_grain : old_grain;

          inactive_grain._status = Status::INACTIVE;
          _console << "Marking Grain " << inactive_grain._id
                   << " as INACTIVE (variable index: " << inactive_grain._var_index << ")\n"
                   << inactive_grain;

          /**
           * If the grain we just marked inactive was the one whose index was in the new grain
           * to existing grain map (other_old_grain). Then we need to update the map to point
           * to the new match winner.
           */
          if (&inactive_grain == &other_old_grain)
            new_grain_index_to_existing_grain_index[closest_match_index] = old_grain_index;
        }
        else
          new_grain_index_to_existing_grain_index[closest_match_index] = old_grain_index;
      }
    }

    // Mark all resolved grain matches
    for (auto new_index = beginIndex(new_grain_index_to_existing_grain_index);
         new_index < new_grain_index_to_existing_grain_index.size();
         ++new_index)
    {
      auto curr_index = new_grain_index_to_existing_grain_index[new_index];

      // This may be a new grain, we'll handle that case below
      if (curr_index == invalid_size_t)
        continue;

      mooseAssert(_feature_sets_old[curr_index]._id != invalid_id,
                  "Invalid ID in old grain structure");

      _feature_sets[new_index]._id = _feature_sets_old[curr_index]._id; // Transfer ID
      _feature_sets[new_index]._status = Status::MARKED;      // Mark the status in the new set
      _feature_sets_old[curr_index]._status = Status::MARKED; // Mark the status in the old set
    }

    /**
     * At this point we have should have only two cases left to handle:
     * Case 1: A grain in the new set who has an unset status (These are new grains, previously
     *         untracked) This case is easy to understand. Since we are matching up grains by
     *         looking at the old set and finding closest matches in the new set, any grain in
     *         the new set that isn't matched up is simply new since some other grain satisfied
     *         each and every request from the old set.
     *
     * Case 2: A grain in the old set who has an unset status (These are inactive grains that
     *         haven't been marked) We can only fall into this case when the very last grain on
     *         a given variable disappears during the current time step. In that case we never have
     *         a matching _var_index in the comparison loop above so that old grain never competes
     *         for any new grain which means it can't be marked inactive in the loop above.
     */
    // Case 1 (new grains in _feature_sets):
    for (auto grain_num = beginIndex(_feature_sets); grain_num < _feature_sets.size(); ++grain_num)
    {
      auto & grain = _feature_sets[grain_num];

      // New Grain
      if (grain._status == Status::CLEAR)
      {
        mooseAssert(!_ebsd_reader || !_first_time,
                    "Can't create new grains in initial EBSD step, logic error");

        /**
         * Now we need to figure out what kind of "new" grain this is. Is it a nucleating grain that
         * we're just barely seeing for the first time or is it a "splitting" grain. A grain that
         * gets pinched into two or more pieces usually as it is being absorbed by other grains or
         * possibly due to external forces. We have to handle splitting grains this way so as to
         * no confuse them with regular grains that just happen to be in contact in this step.
         *
         * Splitting Grain: An grain that is unmatched by any old grain
         *                  on the same order parameter with touching halos.
         *
         * Nucleating Grain: A completely new grain appearing somewhere in the domain
         *                   not overlapping any other grain's halo.
         *
         * To figure out which case we are dealing with, we have to make another pass over all of
         * the existing grains with matching variable indices to see if any of them have overlapping
         * halos.
         */

        // clang-format off
        auto start_it =
            std::lower_bound(_feature_sets.begin(), _feature_sets.end(), grain._var_index,
                             [](const FeatureData & item, std::size_t var_index)
                             {
                               return item._var_index < var_index;
                             });
        // clang-format on

        // Loop over matching variable indices
        for (decltype(_feature_sets.size()) new_grain_index =
                 std::distance(_feature_sets.begin(), start_it);
             new_grain_index < _feature_sets.size() &&
             _feature_sets[new_grain_index]._var_index == grain._var_index;
             ++new_grain_index)
        {
          auto & other_grain = _feature_sets[new_grain_index];

          // Splitting grain?
          if (grain_num != new_grain_index && // Make sure indices aren't pointing at the same grain
              other_grain._status == Status::MARKED && // and that the other grain is indeed marked
              other_grain.boundingBoxesIntersect(grain) && // and the bboxes intersect
              other_grain.halosIntersect(grain))           // and the halos also intersect
          // TODO: Inspect combined volume and see if it's "close" to the expected value
          {
            grain._id = other_grain._id;    // Set the duplicate ID
            grain._status = Status::MARKED; // Mark it
            _console << "Split Grain Detected "
                     << " (variable index: " << grain._var_index << ")\n"
                     << grain << other_grain;
          }
        }

        // Must be a nucleating grain (status is still not set)
        if (grain._status == Status::CLEAR)
        {
          auto new_index = getNextUniqueID();
          grain._id = new_index;          // Set the ID
          grain._status = Status::MARKED; // Mark it
        }
      }
    }

    // Case 2 (inactive grains in _feature_sets_old)
    for (auto & grain : _feature_sets_old)
    {
      if (grain._status == Status::CLEAR)
      {
        grain._status = Status::INACTIVE;
        _console << "Marking Grain " << grain._id
                 << " as INACTIVE (variable index: " << grain._var_index << ")\n"
                 << grain;
      }
    }
  } // is_master

  /*************************************************************
   ****************** COLLECTIVE WORK SECTION ******************
   *************************************************************/

  // Make IDs on all non-master ranks consistent
  scatterAndUpdateRanks();

  // Build up an id to index map
  _communicator.broadcast(_max_curr_grain_id);
  buildFeatureIdToLocalIndices(_max_curr_grain_id);

  /**
   * Trigger callback for new grains
   */
  if (_old_max_grain_id < _max_curr_grain_id)
  {
    for (auto new_id = _old_max_grain_id + 1; new_id <= _max_curr_grain_id; ++new_id)
    {
      // Don't trigger the callback on the reserve IDs
      if (new_id >= _reserve_grain_first_index + _n_reserve_ops)
      {
        // See if we've been instructed to terminate with an error
        if (!_first_time && _error_on_grain_creation)
          mooseError(
              "Error: New grain detected and \"error_on_new_grain_creation\" is set to true");
        else
          newGrainCreated(new_id);
      }
    }
  }
```

```
}

void
GrainTracker::newGrainCreated(unsigned int new_grain_id)
{
  if (!_first_time && _is_master)
  {
    mooseAssert(new_grain_id < _feature_id_to_local_index.size(), "new_grain_id is out of bounds");
    auto grain_index = _feature_id_to_local_index[new_grain_id];
    mooseAssert(grain_index != invalid_size_t && grain_index < _feature_sets.size(),
                "new_grain_id appears to be invalid");

    const auto & grain = _feature_sets[grain_index];
    _console << COLOR_YELLOW
             << "\n*****************************************************************************"
             << "\nCouldn't find a matching grain while working on variable index: "
             << grain._var_index << "\nCreating new unique grain: " << new_grain_id << '\n'
             << grain
             << "\n*****************************************************************************\n"
             << COLOR_DEFAULT;
  }
}

std::vector<unsigned int>
GrainTracker::getNewGrainIDs() const
{
  std::vector<unsigned int> new_ids(_max_curr_grain_id - _old_max_grain_id);
  auto new_id = _old_max_grain_id + 1;

  // Generate the new ids
  std::iota(new_ids.begin(), new_ids.end(), new_id);

  return new_ids;
}

void
GrainTracker::remapGrains()
{
  // Don't remap grains if the current simulation step is before the specified tracking step
  if (_t_step < _tracking_step)
    return;

  _console << "Running remap Grains" << std::endl;

  /**
   * Map used for communicating remap indices to all ranks
   * This map isn't populated until after the remap loop.
   * It's declared here before we enter the root scope
   * since it's needed by all ranks during the broadcast.
   */
  std::map<unsigned int, std::size_t> grain_id_to_new_var;

  // Items are added to this list when split EBSD grains are found
  std::list<std::pair<std::size_t, std::size_t>> ebsd_pairs;

  /**
```

```
   * The remapping algorithm is recursive. We will use the status variable in each FeatureData
   * to track which grains are currently being remapped so we don't have runaway recursion.
   * To begin we need to clear all of the active (MARKED) flags (CLEAR).
   *
   * Additionally we need to record each grain's variable index so that we can communicate
   * changes to the non-root ranks later in a single batch.
   */
  if (_is_master)
  {
    // Build the map to detect difference in _var_index mappings after the remap operation
    std::map<unsigned int, std::size_t> grain_id_to_existing_var_index;
    for (auto & grain : _feature_sets)
    {
      // Unmark the grain so it can be used in the remap loop
      grain._status = Status::CLEAR;

      grain_id_to_existing_var_index[grain._id] = grain._var_index;
    }

    // Make sure that all split pieces of an EBSD grain are on the same OP
    if (_ebsd_reader)
    {
      for (auto i = beginIndex(_feature_sets); i < _feature_sets.size(); ++i)
      {
        auto & grain1 = _feature_sets[i];

        for (auto j = beginIndex(_feature_sets); j < _feature_sets.size(); ++j)
        {
          auto & grain2 = _feature_sets[j];
          if (i == j)
            continue;

          // The first condition below is there to prevent symmetric checks (duplicate values)
          if (i < j && grain1._id == grain2._id)
          {
            ebsd_pairs.push_front(std::make_pair(i, j));
            if (grain1._var_index != grain2._var_index)
            {
              _console << COLOR_YELLOW << "Split EBSD Grain (#" << grain1._id
                       << ") detected on unmatched OPs (" << grain1._var_index << ", "
                       << grain2._var_index << ") attempting to remap to " << grain1._var_index
                       << ".\n"
                       << COLOR_DEFAULT;

              /**
               * We're not going to try very hard to look for a suitable remapping. Just set it to
               * what we want and hope it all works out. Make the GrainTracker great again!
               */
              grain1._var_index = grain2._var_index;
              grain1._status |= Status::DIRTY;
            }
          }
        }
      }
    }

```

```
/**
 * Loop over each grain and see if any grains represented by the same variable are "touching"
 */
bool any_grains_remapped = false;
bool grains_remapped;
do
{
  grains_remapped = false;
  for (auto & grain1 : _feature_sets)
  {
    // We need to remap any grains represented on any variable index above the cuttoff
    if (grain1._var_index >= _reserve_op_index)
    {
      _console << COLOR_YELLOW << "\nGrain #" << grain1._id
               << " detected on a reserved order parameter #" << grain1._var_index
               << ", remapping to another variable\n"
               << COLOR_DEFAULT;

      for (auto max = decltype(_max_renumbering_recursion)(0);
           max <= _max_renumbering_recursion;
           ++max)
        if (max < _max_renumbering_recursion)
        {
          if (attemptGrainRenumber(grain1, 0, max))
           break;
        }
        else if (!attemptGrainRenumber(grain1, 0, max))
        {
          _console << std::flush;
          mooseError(COLOR_RED,
                     "Unable to find any suitable order parameters for remapping."
                     " Perhaps you need more op variables?\n\n",
                     COLOR_DEFAULT);
        }

      grains_remapped = true;
    }

    for (auto & grain2 : _feature_sets)
    {
      // Don't compare a grain with itself and don't try to remap inactive grains
      if (&grain1 == &grain2)
        continue;

      if (grain1._var_index == grain2._var_index && // grains represented by same variable?
          grain1._id != grain2._id &&               // are they part of different grains?
          grain1.boundingBoxesIntersect(grain2) &&  // do bboxes intersect (coarse level)?
          grain1.halosIntersect(grain2))            // do they actually overlap (fine level)?
      {
        _console << COLOR_YELLOW << "\nGrain #" << grain1._id << " intersects Grain #"
                 << grain2._id << " (variable index: " << grain1._var_index << ")\n"
                 << COLOR_DEFAULT;

        for (auto max = decltype(_max_renumbering_recursion)(0);
             max <= _max_renumbering_recursion;
             ++max)
```

```
            if (max < _max_renumbering_recursion)
            {
              if (attemptGrainRenumber(grain1, 0, max))
                break;
            }
            else if (!attemptGrainRenumber(grain1, 0, max) &&
                     !attemptGrainRenumber(grain2, 0, max))
            {
              _console << std::flush;
              mooseError(COLOR_RED,
                         "Unable to find any suitable order parameters for remapping."
                         " Perhaps you need more op variables?\n\n",
                         COLOR_DEFAULT);
            }

          grains_remapped = true;
        }
      }
    }
    any_grains_remapped |= grains_remapped;
  } while (grains_remapped);

  // Verify that EBSD split grains are still intact
  if (_ebsd_reader)
    for (auto & ebsd_pair : ebsd_pairs)
      if (_feature_sets[ebsd_pair.first]._var_index != _feature_sets[ebsd_pair.first]._var_index)
        mooseError("EBSD split grain remapped - This case is currently not handled");

  /**
   * The remapping loop is complete but only on the master process.
   * Now we need to build the remap map and communicate it to the
   * remaining processors.
   */
  for (auto & grain : _feature_sets)
  {
    mooseAssert(grain_id_to_existing_var_index.find(grain._id) !=
                    grain_id_to_existing_var_index.end(),
                "Missing unique ID");

    auto old_var_index = grain_id_to_existing_var_index[grain._id];

    if (old_var_index != grain._var_index)
    {
      mooseAssert(static_cast<bool>(grain._status & Status::DIRTY), "grain status is incorrect");

      grain_id_to_new_var.emplace_hint(
          grain_id_to_new_var.end(),
          std::pair<unsigned int, std::size_t>(grain._id, grain._var_index));

      /**
       * Since the remapping algorithm only runs on the root process,
       * the variable index on the master's grains is inconsistent from
       * the rest of the ranks. These are the grains with a status of
       * DIRTY. As we build this map we will temporarily switch these
       * variable indices back to the correct value so that all
       * processors use the same algorithm to remap.
```

```
         */
        grain._var_index = old_var_index;
        // Clear the DIRTY status as well for consistency
        grain._status &= ~Status::DIRTY;
      }
    }

    if (!grain_id_to_new_var.empty())
    {
      _console << "\nFinal remapping tally:\n";
      for (const auto & remap_pair : grain_id_to_new_var)
        _console << "Grain #" << remap_pair.first << " var_index "
                 << grain_id_to_existing_var_index[remap_pair.first] << " -> " << remap_pair.second
                 << '\n';
      _console << "Communicating swaps with remaining processors..." << std::endl;
    }
  } // root processor

  // Communicate the std::map to all ranks
  _communicator.broadcast(grain_id_to_new_var);

  // Perform swaps if any occurred
  if (!grain_id_to_new_var.empty())
  {
    // Cache for holding values during swaps
    std::vector<std::map<Node *, CacheValues>> cache(_n_vars);

    // Perform the actual swaps on all processors
    for (auto & grain : _feature_sets)
    {
      // See if this grain was remapped
      auto new_var_it = grain_id_to_new_var.find(grain._id);
      if (new_var_it != grain_id_to_new_var.end())
        swapSolutionValues(grain, new_var_it->second, cache, RemapCacheMode::FILL);
    }

    for (auto & grain : _feature_sets)
    {
      // See if this grain was remapped
      auto new_var_it = grain_id_to_new_var.find(grain._id);
      if (new_var_it != grain_id_to_new_var.end())
        swapSolutionValues(grain, new_var_it->second, cache, RemapCacheMode::USE);
    }

    _nl.solution().close();
    _nl.solutionOld().close();
    _nl.solutionOlder().close();

    _fe_problem.getNonlinearSystemBase().system().update();

    _console << "Swaps complete" << std::endl;
  }
}

void
GrainTracker::computeMinDistancesFromGrain(FeatureData & grain,
                                           std::vector<std::list<GrainDistance>> & min_distances)
{
  /**
   * In the diagram below assume we have 4 order parameters. The grain with the asterisk needs to
   * be remapped. All order parameters are used in neighboring grains. For all "touching" grains,
   * the value of the corresponding entry in min_distances will be a negative integer representing
   * the number of immediate neighbors with that order parameter.
   *
   *  Note: Only the first member of the pair (the distance) is shown in the array below.
   *        e.g. [-2.0, -max, -1.0, -2.0]
   *
   * After sorting, variable index 2 (value: -1.0) be at the end of the array and will be the first
   * variable we attempt to renumber the current grain to.
   *
   *        __       ___
   *          \  0  /   \
   *        2  \___/  1  \___
   *           /   \     /   \
   *        __/  1  \___/  2  \
   *          \  *  /   \     /
   *        3  \___/  3  \___/
   *           /   \     /
   *        __/  0  \___/
   *
   */
  for (auto i = beginIndex(_feature_sets); i < _feature_sets.size(); ++i)
  {
    auto & other_grain = _feature_sets[i];

    if (other_grain._var_index == grain._var_index || other_grain._var_index >= _reserve_op_index)
      continue;

    auto target_var_index = other_grain._var_index;
    auto target_grain_index = i;
    auto target_grain_id = other_grain._id;

    Real curr_bbox_diff = boundingRegionDistance(grain._bboxes, other_grain._bboxes);

    GrainDistance grain_distance_obj(
        curr_bbox_diff, target_var_index, target_grain_index, target_grain_id);

    // To handle touching halos we penalize the top pick each time we see another
    if (curr_bbox_diff == -1.0 && !min_distances[target_var_index].empty())
    {
      Real last_distance = min_distances[target_var_index].begin()->_distance;
      if (last_distance < 0)
        grain_distance_obj._distance += last_distance;
    }

    // Insertion sort into a list
    auto insert_it = min_distances[target_var_index].begin();
    while (insert_it != min_distances[target_var_index].end() && !(grain_distance_obj < *insert_it))
      ++insert_it;
    min_distances[target_var_index].insert(insert_it, grain_distance_obj);
  }

  /**
   * See if we have any completely open OPs (excluding reserve order parameters) or the order
   * parameter corresponding to this grain, we need to put them in the list or the grain  tracker
   * won't realize that those vars are available for remapping.
   */
  for (auto var_index = beginIndex(_vars); var_index < _reserve_op_index; ++var_index)
  {
    // Don't put an entry in for matching variable indices (i.e. we can't remap to ourselves)
    if (grain._var_index == var_index)
      continue;

    if (min_distances[var_index].empty())
      min_distances[var_index].emplace_front(std::numeric_limits<Real>::max(), var_index);
  }
}

bool
GrainTracker::attemptGrainRenumber(FeatureData & grain, unsigned int depth, unsigned int max_depth)
{
  // End the recursion of our breadth first search
  if (depth > max_depth)
    return false;

  std::size_t curr_var_index = grain._var_index;

  std::vector<std::map<Node *, CacheValues>> cache;

  std::vector<std::list<GrainDistance>> min_distances(_vars.size());

  /**
   * We have two grains that are getting close represented by the same order parameter.
   * We need to map to the variable whose closest grain to this one is furthest away by bounding
   * region to bounding region distance.
   */
  computeMinDistancesFromGrain(grain, min_distances);

  /**
   * We have a vector of the distances to the closest grains represented by each of our variables.
   * We just need to pick a suitable grain to replace with. We will start with the maximum of this
   * this list: (max of the mins), but will settle for next to largest and so forth as we make more
   * attempts at remapping grains. This is a graph coloring problem so more work will be required
   * to optimize this process.
   *
   * Note: We don't have an explicit check here to avoid remapping a variable to itself. This is
   * unnecessary since the min_distance of a variable is explicitly set up above.
   */
  // clang-format off
  std::sort(min_distances.begin(), min_distances.end(),
            [](const std::list<GrainDistance> & lhs, const std::list<GrainDistance> & rhs)
              {
                // Sort lists in reverse order (largest distance first)
                // These empty cases are here to make this comparison stable
                if (lhs.empty())
                  return false;
                else if (rhs.empty())
                  return true;
                else
                  return lhs.begin()->_distance > rhs.begin()->_distance;
              });
  // clang-format on

  for (auto & list_ref : min_distances)
  {
    const auto target_it = list_ref.begin();
    if (target_it == list_ref.end())
      continue;

    // If the distance is positive we can just remap and be done
    if (target_it->_distance > 0)
    {
      _console << COLOR_GREEN << "- Depth " << depth << ": Remapping grain #" << grain._id
               << " from variable index " << curr_var_index << " to " << target_it->_var_index;
      if (target_it->_distance == std::numeric_limits<Real>::max())
        _console << " which currently contains zero grains." << COLOR_DEFAULT;
      else
        _console << " whose closest grain (#" << target_it->_grain_id << ") is at a distance of "
                 << target_it->_distance << "\n"
                 << COLOR_DEFAULT;

      grain._status |= Status::DIRTY;
      grain._var_index = target_it->_var_index;
      return true;
    }

    // If the distance isn't positive we just need to make sure that none of the grains represented
    // by the target variable index would intersect this one if we were to remap
    auto next_target_it = target_it;
    bool intersection_hit = false;
    std::ostringstream oss;
    while (!intersection_hit && next_target_it != list_ref.end())
    {
      if (next_target_it->_distance > 0)
        break;

      mooseAssert(next_target_it->_grain_index < _feature_sets.size(),
                  "Error in indexing target grain in attemptGrainRenumber");
      FeatureData & next_target_grain = _feature_sets[next_target_it->_grain_index];

      // If any grains touch we're done here
      if (grain.halosIntersect(next_target_grain))
        intersection_hit = true;
      else
        oss << " #" << next_target_it->_grain_id;

      ++next_target_it;
    }

    if (!intersection_hit)
    {
      _console << COLOR_GREEN << "- Depth " << depth << ": Remapping grain #" << grain._id
               << " from variable index " << curr_var_index << " to " << target_it->_var_index
               << " whose closest grain:" << oss.str()
```

```
              << " is inside our bounding box but whose halo(s) are not touching.\n"
              << COLOR_DEFAULT;

      grain._status |= Status::DIRTY;
      grain._var_index = target_it->_var_index;
      return true;
    }

    // If we reach this part of the loop, there is no simple renumbering that can be done.
    mooseAssert(target_it->_grain_index < _feature_sets.size(),
                "Error in indexing target grain in attemptGrainRenumber");
    FeatureData & target_grain = _feature_sets[target_it->_grain_index];

    /**
     * If we get to this case and the best distance is less than -1, we are in big trouble.
     * This means that grains represented by all of the remaining order parameters are
     * overlapping this one in at least two places. We'd have to maintain multiple recursive
     * chains, or just start over from scratch...
     * Let's just return false and see if there is another remapping option.
     */
    if (target_it->_distance < -1)
      return false;

    // Make sure this grain isn't marked. If it is, we can't recurse here
    if ((target_grain._status & Status::MARKED) == Status::MARKED)
      return false;

    /**
     * Propose a new variable index for the current grain and recurse.
     * We don't need to mark the status as DIRTY here since the recursion
     * may fail. For now, we'll just add MARKED to the status.
     */
    grain._var_index = target_it->_var_index;
    grain._status |= Status::MARKED;
    if (attemptGrainRenumber(target_grain, depth + 1, max_depth))
    {
      // SUCCESS!
      _console << COLOR_GREEN << "- Depth " << depth << ": Remapping grain #" << grain._id
               << " from variable index " << curr_var_index << " to " << target_it->_var_index
               << '\n'
               << COLOR_DEFAULT;

      // Now we need to mark the grain as DIRTY since the recursion succeeded.
      grain._status |= Status::DIRTY;
      return true;
    }
    else
      // FAILURE, We need to set our var index back after failed recursive step
      grain._var_index = curr_var_index;

    // ALWAYS "unmark" (or clear the MARKED status) after recursion so it can be used by other remap
    // operations
    grain._status &= ~Status::MARKED;
  }

  return false;
```

```
}

void
GrainTracker::swapSolutionValues(FeatureData & grain,
                                 std::size_t new_var_index,
                                 std::vector<std::map<Node *, CacheValues>> & cache,
                                 RemapCacheMode cache_mode)
{
  MeshBase & mesh = _mesh.getMesh();

  // Remap the grain
  std::set<Node *> updated_nodes_tmp; // Used only in the elemental case
  for (auto entity : grain._local_ids)
  {
    if (_is_elemental)
    {
      Elem * elem = mesh.query_elem(entity);
      if (!elem)
        continue;

      for (unsigned int i = 0; i < elem->n_nodes(); ++i)
      {
        Node * curr_node = elem->get_node(i);
        if (updated_nodes_tmp.find(curr_node) == updated_nodes_tmp.end())
        {
          // cache this node so we don't attempt to remap it again within this loop
          updated_nodes_tmp.insert(curr_node);
          swapSolutionValuesHelper(curr_node, grain._var_index, new_var_index, cache, cache_mode);
        }
      }
    }
    else
      swapSolutionValuesHelper(
          mesh.query_node_ptr(entity), grain._var_index, new_var_index, cache, cache_mode);
  }

  // Update the variable index in the unique grain datastructure after swaps are complete
  if (cache_mode == RemapCacheMode::USE || cache_mode == RemapCacheMode::BYPASS)
    grain._var_index = new_var_index;
}

void
GrainTracker::swapSolutionValuesHelper(Node * curr_node,
                                       std::size_t curr_var_index,
                                       std::size_t new_var_index,
                                       std::vector<std::map<Node *, CacheValues>> & cache,
                                       RemapCacheMode cache_mode)
{
  if (curr_node && curr_node->processor_id() == processor_id())
  {
    // Reinit the node so we can get and set values of the solution here
    _subproblem.reinitNode(curr_node, 0);

    // Local variables to hold values being transferred
    Real current, old = 0, older = 0;
    // Retrieve the value either from the old variable or cache
```

```
if (cache_mode == RemapCacheMode::FILL || cache_mode == RemapCacheMode::BYPASS)
{
  current = _vars[curr_var_index]->nodalSln()[0];
  if (_is_transient)
  {
    old = _vars[curr_var_index]->nodalSlnOld()[0];
    older = _vars[curr_var_index]->nodalSlnOlder()[0];
  }
}
else // USE
{
  const auto cache_it = cache[curr_var_index].find(curr_node);
  mooseAssert(cache_it != cache[curr_var_index].end(), "Error in cache");
  current = cache_it->second.current;
  old = cache_it->second.old;
  older = cache_it->second.older;
}

// Cache the value or use it!
if (cache_mode == RemapCacheMode::FILL)
{
  cache[curr_var_index][curr_node].current = current;
  cache[curr_var_index][curr_node].old = old;
  cache[curr_var_index][curr_node].older = older;
}
else // USE or BYPASS
{
  const auto & dof_index = _vars[new_var_index]->nodalDofIndex();

  // Transfer this solution from the old to the current
  _nl.solution().set(dof_index, current);
  if (_is_transient)
  {
    _nl.solutionOld().set(dof_index, old);
    _nl.solutionOlder().set(dof_index, older);
  }
}

/**
 * Finally zero out the old variable. When using the FILL/USE combination to
 * read/write variables, it's important to zero the variable on the FILL
 * stage and not the USE stage. The reason for this is handling swaps as
 * illustrated in the following diagram
 *       ___  ___
 *      /   \/   \    If adjacent grains (overlapping flood region) end up
 *     /  1 /\ 2  \   swapping variable indices and variables are zeroed on
 *     \  2*\/ 1* /   "USE", the overlap region will be incorrectly zeroed
 *      \___/\___/    by whichever variable is written to second.
 *.
 */
if (cache_mode == RemapCacheMode::FILL || cache_mode == RemapCacheMode::BYPASS)
{
  const auto & dof_index = _vars[curr_var_index]->nodalDofIndex();

  // Set the DOF for the current variable to zero
  _nl.solution().set(dof_index, 0.0);
```

```
    if (_is_transient)
    {
      _nl.solutionOld().set(dof_index, 0.0);
      _nl.solutionOlder().set(dof_index, 0.0);
    }
   }
  }
}

void
GrainTracker::updateFieldInfo()
{
  for (auto map_num = decltype(_maps_size)(0); map_num < _maps_size; ++map_num)
    _feature_maps[map_num].clear();

  std::map<dof_id_type, Real> tmp_map;
  MeshBase & mesh = _mesh.getMesh();

  for (const auto & grain : _feature_sets)
  {
    std::size_t curr_var = grain._var_index;
    std::size_t map_index = (_single_map_mode || _condense_map_info) ? 0 : curr_var;

    for (auto entity : grain._local_ids)
    {
      // Highest variable value at this entity wins
      Real entity_value = std::numeric_limits<Real>::lowest();
      if (_is_elemental)
      {
        const Elem * elem = mesh.elem(entity);
        std::vector<Point> centroid(1, elem->centroid());
        if (_ebsd_reader && _first_time)
        {
          const EBSDAccessFunctors::EBSDPointData & d = _ebsd_reader->getData(centroid[0]);
          const auto phase = d._phase;
          if (!_consider_phase || phase == _phase)
          {
            const auto global_id = _ebsd_reader->getGlobalID(d._feature_id);
            const auto local_id = _ebsd_reader->getAvgData(global_id)._local_id;
            const auto grain_id = _consider_phase ? local_id : global_id;
            if (grain_id == grain._id)
              entity_value = std::numeric_limits<Real>::max();
          }
        }
        else
        {
          _fe_problem.reinitElemPhys(elem, centroid, 0);
          entity_value = _vars[curr_var]->sln()[0];
        }
      }
      else
      {
        Node & node = mesh.node(entity);
        entity_value = _vars[curr_var]->getNodalValue(node);
      }

      if (entity_value != std::numeric_limits<Real>::lowest() &&
          (tmp_map.find(entity) == tmp_map.end() || entity_value > tmp_map[entity]))
      {
        mooseAssert(grain._id != invalid_id, "Missing Grain ID");
        _feature_maps[map_index][entity] = grain._id;

        if (_var_index_mode)
          _var_index_maps[map_index][entity] = grain._var_index;

        tmp_map[entity] = entity_value;
      }

      if (_compute_var_to_feature_map)
      {
        auto map_it = _entity_var_to_features.lower_bound(entity);
        if (map_it == _entity_var_to_features.end() || map_it->first != entity)
        {
          map_it = _entity_var_to_features.emplace_hint(
              map_it, entity, std::vector<unsigned int>(_n_vars, invalid_id));

          // insert the reserve op numbers (if appropriate)
          for (auto reserve_index = decltype(_n_reserve_ops)(0); reserve_index < _n_reserve_ops;
               ++reserve_index)
            map_it->second[reserve_index] = _reserve_grain_first_index + reserve_index;
        }
        map_it->second[grain._var_index] = grain._id;
      }
    }

    if (_compute_halo_maps)
      for (auto entity : grain._halo_ids)
        _halo_ids[grain._var_index][entity] = grain._var_index;

    for (auto entity : grain._ghosted_ids)
      _ghosted_entity_ids[entity] = 1;
  }

  communicateHaloMap();
}

void
GrainTracker::communicateHaloMap()
{
  if (_compute_halo_maps)
  {
    // rank               var_index    entity_id
    std::vector<std::pair<std::size_t, dof_id_type>> halo_ids_all;

    std::vector<int> counts;
    std::vector<std::pair<std::size_t, dof_id_type>> local_halo_ids;
    std::size_t counter = 0;

    if (_is_master)
    {
      std::vector<std::vector<std::pair<std::size_t, dof_id_type>>> root_halo_ids(_n_procs);
      counts.resize(_n_procs);

      auto & mesh = _mesh.getMesh();
      // Loop over the _halo_ids "field" and build minimal lists for all of the other ranks
      for (auto var_index = beginIndex(_halo_ids); var_index < _halo_ids.size(); ++var_index)
      {
        for (const auto & entity_pair : _halo_ids[var_index])
        {
          DofObject * halo_entity;
          if (_is_elemental)
            halo_entity = mesh.elem(entity_pair.first);
          else
            halo_entity = &mesh.node(entity_pair.first);

          root_halo_ids[halo_entity->processor_id()].push_back(
              std::make_pair(var_index, entity_pair.first));
        }
      }

      // Build up the counts vector for MPI scatter
      std::size_t global_count = 0;
      for (const auto & vector_ref : root_halo_ids)
      {
        std::copy(vector_ref.begin(), vector_ref.end(), std::back_inserter(halo_ids_all));
        counts[counter] = vector_ref.size();
        global_count += counts[counter++];
      }
    }

    _communicator.scatter(halo_ids_all, counts, local_halo_ids);

    // Now add the contributions from the root process to the processor local maps
    for (const auto & halo_pair : local_halo_ids)
      _halo_ids[halo_pair.first].emplace(std::make_pair(halo_pair.second, halo_pair.first));

    // Finally remove halo markings from stitch regions
    for (const auto & grain : _feature_sets)
      for (auto local_id : grain._local_ids)
        _halo_ids[grain._var_index].erase(local_id);
  }
}

Real
GrainTracker::centroidRegionDistance(std::vector<MeshTools::BoundingBox> & bboxes1,
                                     std::vector<MeshTools::BoundingBox> & bboxes2) const
{
  /**
   * Find the minimum centroid distance between any to pieces of the grains.
   */
  auto min_distance = std::numeric_limits<Real>::max();
  for (const auto & bbox1 : bboxes1)
  {
    const auto centroid_point1 = (bbox1.max() + bbox1.min()) / 2.0;

    for (const auto & bbox2 : bboxes2)
    {
      const auto centroid_point2 = (bbox2.max() + bbox2.min()) / 2.0;

      // Here we'll calculate a distance between the centroids
      auto curr_distance = _mesh.minPeriodicDistance(_var_number, centroid_point1, centroid_point2);

      if (curr_distance < min_distance)
        min_distance = curr_distance;
    }
  }

  return min_distance;
}

Real
GrainTracker::boundingRegionDistance(std::vector<MeshTools::BoundingBox> & bboxes1,
                                     std::vector<MeshTools::BoundingBox> & bboxes2) const
{
  /**
   * The region that each grain covers is represented by a bounding box large enough to encompassing
   * all the points within that grain. When using periodic boundaries, we may have several discrete
   * "pieces" of a grain each represented by a bounding box. The distance between any two grains
   * is defined as the minimum distance between any pair of boxes, one selected from each grain.
   */
  auto min_distance = std::numeric_limits<Real>::max();
  for (const auto & bbox1 : bboxes1)
  {
    for (const auto & bbox2 : bboxes2)
    {
      // AABB squared distance
      Real curr_distance = 0.0;
      bool boxes_overlap = true;
      for (unsigned int dim = 0; dim < LIBMESH_DIM; ++dim)
      {
        const auto & min1 = bbox1.min()(dim);
        const auto & max1 = bbox1.max()(dim);
        const auto & min2 = bbox2.min()(dim);
        const auto & max2 = bbox2.max()(dim);

        if (min1 > max2)
        {
          const auto delta = max2 - min1;
          curr_distance += delta * delta;
          boxes_overlap = false;
        }
        else if (min2 > max1)
        {
          const auto delta = max1 - min2;
          curr_distance += delta * delta;
          boxes_overlap = false;
        }
      }

      if (boxes_overlap)
        return -1.0; /* all overlaps are treated the same */

      if (curr_distance < min_distance)
        min_distance = curr_distance;
    }
  }

  return min_distance;
}

unsigned int
GrainTracker::getNextUniqueID()
{
  /**
   * Get the next unique grain ID but make sure to respect
   * reserve ids. Note, that the first valid ID for a new
   * grain is _reserve_grain_first_index + _n_reserve_ops because
   * _reserve_grain_first_index IS a valid index. It does not
   * point to the last valid index of the non-reserved grains.
   */
  _max_curr_grain_id = std::max(_max_curr_grain_id + 1,
                                _reserve_grain_first_index + _n_reserve_ops /* no +1 here!*/);

  return _max_curr_grain_id;
}

/*************************************************
 ************** Helper Structures ****************
 *************************************************/
GrainDistance::GrainDistance(Real distance, std::size_t var_index)
  : GrainDistance(distance,
                  var_index,
                  std::numeric_limits<std::size_t>::max(),
                  std::numeric_limits<unsigned int>::max())
{
}

GrainDistance::GrainDistance(Real distance,
                             std::size_t var_index,
                             std::size_t grain_index,
                             unsigned int grain_id)
  : _distance(distance), _var_index(var_index), _grain_index(grain_index), _grain_id(grain_id)
{
}

bool
GrainDistance::operator<(const GrainDistance & rhs) const
{
  return _distance < rhs._distance;
}
```

A.5 FeatureVolumeVectorPostprocessor.h

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/

#ifndef FEATUREVOLUMEVECTORPOSTPROCESSOR_H
#define FEATUREVOLUMEVECTORPOSTPROCESSOR_H

#include "GeneralVectorPostprocessor.h"
#include "MooseVariableDependencyInterface.h"

// Forward Declarations
class FeatureVolumeVectorPostprocessor;
class FeatureFloodCount;

template <>
InputParameters validParams<FeatureVolumeVectorPostprocessor>();

/**
 * This VectorPostprocessor is intended to be used to calculate
 * accurate volumes from the FeatureFloodCount and/or GrainTracker
 * objects. It is a GeneralVectorPostProcessor instead of the
 * more natural elemental kind so that dependency resolution
 * will work properly when an AuxVariable is not depending
 * on the FeatureFloodCount object. It obtains the coupled
 * variables from the FeatureFloodCount object so that there's
 * one less thing for the user of this class to worry about.
 */
class FeatureVolumeVectorPostprocessor : public GeneralVectorPostprocessor,
                                         public MooseVariableDependencyInterface
{
public:
  FeatureVolumeVectorPostprocessor(const InputParameters & parameters);

  virtual void initialize() override;
  virtual void execute() override;
  virtual void finalize() override;

  /**
   * Returns the volume for the given grain number.
   */
  Real getFeatureVolume(unsigned int feature_id) const;

protected:
  /// A Boolean indicating how the volume is calculated
  const bool _single_feature_per_elem;

  /// A reference to the feature flood count object
  const FeatureFloodCount & _feature_counter;

  VectorPostprocessorValue & _var_num;
  VectorPostprocessorValue & _feature_volumes;
  VectorPostprocessorValue & _intersects_bounds;

private:
  /// Add volume contributions to one or entries in the feature volume vector
  void accumulateVolumes(const Elem * elem,
                         const std::vector<unsigned int> & var_to_features,
                         std::size_t num_features);

  /// Calculate the integral value of the passed in variable (index)
  Real computeIntegral(std::size_t var_index) const;

  const std::vector<MooseVariable *> & _vars;
  std::vector<const VariableValue *> _coupled_sln;

  MooseMesh & _mesh;
  Assembly & _assembly;
  const MooseArray<Point> & _q_point;
  QBase *& _qrule;
  const MooseArray<Real> & _JxW;
  const MooseArray<Real> & _coord;
};

#endif
```

A.6 FeatureVolumeVectorPostprocessor.C

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/

#include "FeatureVolumeVectorPostprocessor.h"
#include "FeatureFloodCount.h"
#include "GrainTrackerInterface.h"
#include "MooseMesh.h"
#include "Assembly.h"

#include "libmesh/quadrature.h"

template <>
InputParameters
validParams<FeatureVolumeVectorPostprocessor>()
{
  InputParameters params = validParams<GeneralVectorPostprocessor>();

  params.addRequiredParam<UserObjectName>("flood_counter",
                                          "The FeatureFloodCount UserObject to get values from.");
  params.addParam<bool>("single_feature_per_element",
                        false,
                        "Set this Boolean if you wish to use an element based volume where"
                        " the dominant order parameter determines the feature that accumulates the "
                        "entire element volume");
  return params;
}

FeatureVolumeVectorPostprocessor::FeatureVolumeVectorPostprocessor(
    const InputParameters & parameters)
  : GeneralVectorPostprocessor(parameters),
    MooseVariableDependencyInterface(),
    _single_feature_per_elem(getParam<bool>("single_feature_per_element")),
    _feature_counter(getUserObject<FeatureFloodCount>("flood_counter")),
    _var_num(declareVector("var_num")),
    _feature_volumes(declareVector("feature_volumes")),
    _intersects_bounds(declareVector("intersects_bounds")),
    _vars(_feature_counter.getCoupledVars()),
    _mesh(_subproblem.mesh()),
    _assembly(_subproblem.assembly(_tid)),
    _q_point(_assembly.qPoints()),
    _qrule(_assembly.qRule()),
    _JxW(_assembly.JxW()),
    _coord(_assembly.coordTransformation())
{
  addMooseVariableDependency(_vars);

  _coupled_sln.reserve(_vars.size());
  for (auto & var : _vars)
    _coupled_sln.push_back(&var->sln());
}
```

```

void
FeatureVolumeVectorPostprocessor::initialize()
{
}

void
FeatureVolumeVectorPostprocessor::execute()
{
  const auto num_features = _feature_counter.getTotalFeatureCount();

  // Reset the variable index and intersect bounds vectors
  _var_num.assign(num_features, -1);            // Invalid
  _intersects_bounds.assign(num_features, -1); // Invalid
  for (auto feature_num = beginIndex(_var_num); feature_num < num_features; ++feature_num)
  {
    auto var_num = _feature_counter.getFeatureVar(feature_num);
    if (var_num != FeatureFloodCount::invalid_id)
      _var_num[feature_num] = var_num;

    _intersects_bounds[feature_num] =
        static_cast<unsigned int>(_feature_counter.doesFeatureIntersectBoundary(feature_num));
  }

  // Reset the volume vector
  _feature_volumes.assign(num_features, 0);
  const auto end = _mesh.getMesh().active_local_elements_end();
  for (auto el = _mesh.getMesh().active_local_elements_begin(); el != end; ++el)
  {
    const Elem * elem = *el;
    _fe_problem.prepare(elem, 0);
    _fe_problem.reinitElem(elem, 0);

    /**
     * Here we retrieve the var to features vector on the current element.
     * We'll use that information to figure out which variables are non-zero
     * (from a threshold perspective) then we can sum those values into
     * appropriate grain index locations.
     */
    const auto & var_to_features = _feature_counter.getVarToFeatureVector(elem->id());

    accumulateVolumes(elem, var_to_features, num_features);
  }
}

void
FeatureVolumeVectorPostprocessor::finalize()
{
  // Do the parallel sum
  _communicator.sum(_feature_volumes);
}

Real
FeatureVolumeVectorPostprocessor::getFeatureVolume(unsigned int feature_id) const
{
  mooseAssert(feature_id < _feature_volumes.size(), "feature_id is out of range");
  return _feature_volumes[feature_id];
}

void
FeatureVolumeVectorPostprocessor::accumulateVolumes(
    const Elem * elem,
    const std::vector<unsigned int> & var_to_features,
    std::size_t libmesh_dbg_var(num_features))
{
  unsigned int dominant_feature_id = FeatureFloodCount::invalid_id;
  Real max_var_value = std::numeric_limits<Real>::lowest();

  for (auto var_index = beginIndex(var_to_features); var_index < var_to_features.size();
       ++var_index)
  {
    // Only sample "active" variables
    if (var_to_features[var_index] != FeatureFloodCount::invalid_id)
    {
      auto feature_id = var_to_features[var_index];
      mooseAssert(feature_id < num_features, "Feature ID out of range");
      auto integral_value = computeIntegral(var_index);

      // Compute volumes in a simplistic but domain conservative fashion
      if (_single_feature_per_elem)
      {
        if (integral_value > max_var_value)
        {
          // Update the current dominant feature and associated value
          max_var_value = integral_value;
          dominant_feature_id = feature_id;
        }
      }
      // Solution based volume calculation (integral value)
      else
        _feature_volumes[feature_id] += integral_value;
    }
  }

  // Accumulate the entire element volume into the dominant feature. Do not use the integral value
  if (_single_feature_per_elem && dominant_feature_id != FeatureFloodCount::invalid_id)
    _feature_volumes[dominant_feature_id] += elem->volume();
}

Real
FeatureVolumeVectorPostprocessor::computeIntegral(std::size_t var_index) const
{
  Real sum = 0;

  for (unsigned int qp = 0; qp < _qrule->n_points(); ++qp)
    sum += _JxW[qp] * _coord[qp] * (*_coupled_sln[var_index])[qp];

  return sum;
}
```