IMPROVING PARALLELISM OF SCIENTIFIC AND ENGINEERING APPLICATIONS ON HETEROGENEOUS SUPERCOMPUTERS

Gerrett Diamond

Submitted in Partial Fullfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

Approved by:
Mark S. Shephard, Chair
George Slota
Barb Cutler
Onkar Sahni
Cameron W. Smith

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York

[August 2021]
Submitted August 2021

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

This thesis and the years of research devoted to it owes thanks to many people who have contributed and supported me. Thank you to my committee for helping me along my journey. A special thanks to Cameron Smith who has served as a wonderful mentor for me since my time as an undergraduate researcher. Thanks to my advisor, Mark Shephard for always pushing me to progress in my research and provide the means for me to grow professionally. Many thanks to my colleagues Dan Ibanez, Brian Granzow, Chonglin Zhang, Dhyan Nath and many others who helped me understand concepts involved in our research and for their time in development and debugging.

The opportunity to work on several of the most powerful supercomputers owes thanks to the staff and system administrators at the Computational Center for Innovations, the Texas Advanced Computing Center, the National Energy Research Scientific Computing Center, Argonne National Laboratories, and Oak Ridge National Laboratories.

Finally, I'd like to thank my family, friends, and my cats who have supported me and kept me striving for success these past years. A special thanks to Anthony Trubiano and my two cats Pluot and Dango for keeping me company through the COVID pandemic.

ABSTRACT

The rising usage of heterogeneous supercomputers introduces both opportunities for increased parallelism and challenges for efficient usage of the available hardware. Applications running on heterogeneous supercomputers must adopt new methods to achieve performance across two levels of parallelism. Inter-process parallelism defines coordination between processes and intra-process parallelism within each process. This thesis presents research towards improving inter-process and intra-process parallelism for applications that use complex data structures such as distributed unstructured meshes.

Inter-process parallelism is defined by the coupled costs of the partition of load between processes and the communications between processes required as a result of the partition. To achieve optimal performance, partitions must divide computational load evenly between processes while minimizing the additional costs of communications. This thesis addresses improving inter-process parallelism using multicriteria partition improvement multicriteria methods on a generalized structure for a broad set of potential applications. The partition improvement methods are applied to different unstructured mesh setups with partitions up to half a million processes.

In the case of heterogeneous supercomputers, intra-process parallelism is dictated by the parallel hardware available to each process for performing computations. For most of the current and next generation US systems, Graphic Processing Units (GPUs) are the parallel hardware available on each node. This thesis addresses methods for intra-process parallelism in the scope of particle-in-cell simulations with a novel approach to the storage of the unstructured mesh and the particles for optimized performance on GPUs while utilizing performance-portable methods for performance on future hardware. Scaling studies of these methods are presented up to 4096 nodes of the Summit supercomputer with over a trillion particles simulated.

CHAPTER 1
INTRODUCTION

Scientists and engineers use simulations running on computers to model domains of interest and perform virtual experiments to more efficiently expand our understanding for different aspects of our world. To execute these simulations on computers the domains of interest must be expressed as structures that can be operated on efficiently. Depending on the size and complexity of the domain, different models may be employed. For many engineering simulations including finite-element [1], finite-volume [2], and particle-in-cell [3] simulations, unstructured meshes are commonly employed to model complex domains and accurately examine the physical scenarios driving each simulation.

As simulations study broader domains and more complex physics, larger and faster computers are required to accurately perform the calculations. When the simulations grow beyond the capacity of a single computational unit or CPU, the simulations data and calculations must be partitioned across multiple processes and memory spaces on many CPUs that perform calculations in parallel. The largest parallel computers used for these simulations are referred to as supercomputers.

Simulations executing on massively parallel supercomputers must achieve a high level of performance and scalability to properly utilize the available hardware. Performance on these systems can be broken down into two categories, inter-process performance and intra-process performance. Inter-process performance depends on the cost of communications between processes used to synchronize the simulation. Additionally, the balance of computational work across all processes is important for inter-process performance as processes that have less work will spend more time idle waiting on the processes with more work. Intra-process performance is the performance of the computational work given to each process. This relates to the usage of the available computational hardware and the effective mitigation of the hardware latencies.

1.1 Partitioning and Load Balancing

For gaining optimal inter-process performance, the key is to minimize the need for communications between processes, optimize the necessary communications, and maintain a well balanced load of computational work between processes. While the exact usage of inter-process communications is at the application level, the amount of data that needs to be communicated and the balance of computational work is a direct result of how the simulation data is partitioned across the processes. Applying partitioning and load balancing techniques can successfully distribute work evenly while also minimizing the overlapping data that requires communications. The most powerful of these methods for partitioning unstructured meshes is the multilevel graph methods [4]–[8] that have been shown to scale up to tens of thousands of processes [9]. However, these methods tend to fail at larger process counts due to memory usage [10] and can only target one criteria for partitioning. Diffusive methods have been used to improve the partitions of multilevel graph methods to achieve partitions up to millions of processes and target multiple criteria simultaneously [11]–[13]. Diffusive methods are also useful for evolving simulations where the computational work load changes as the simulation progresses. Evolving simulations such as simulations using mesh adaptation [14]–[17] and particle-in-cell simulations [18]–[21] require dynamic load balancing throughout the simulation to maintain optimal performance. Expensive or memory intensive methods such as multilevel graph methods can not be applied in this case where as fast low-cost diffusive methods are an excellent choice [22], [23].

The partitioning methods used for unstructured meshes are applicable to a range of applications, but require being generalized for other models that exhibit similar characteristics. The N-Graph Partitioner (EnGPar) has been developed to provide the diffusive methods previously used for unstructured meshes to perform partition improvement on a range of domains of interest including parallel unstructured meshes. EnGPar is shown to achieve similar partition improvement for unstructured meshes faster than the direct unstructured mesh diffusive tool on over half a million processes. EnGPar is also applied to perform dynamic load balancing of particles in an unstructured mesh domain for particle-in-cell simulations.

Chapter 2 details methods developed in the EnGPar library and the improvements to previous methods. EnGPar is applied to improve the partitions of unstructured meshes in Chapter 3 including a comparison of EnGPar to its predecessor ParMA [13] and cases that ParMA does not natively support. A method for performing dynamic load balancing of particles for particle-in-cell simulations using a partitioned mesh is detailed in Chapter 4. Usage of this method with results are included in Chapter 5.

1.2 Particle-In-Cell Framework for Efficient Operation on Accelerators

Intra-process performance is achieved through proper usage of system hardware to optimize the necessary calculations and memory accesses a simulation requires. This can include usage of data structures that perform well on the underlying hardware, algorithms that exploit the memory layout and hierarchy, etc. Optimizing intra-process performance is further complicated on computers which include accelerators. Accelerators are additional computational devices that perform many computations simultaneously resulting in much faster calculation than the CPUs can perform. While accelerators can generally speed up calculations, they introduce new challenges and limitations in the algorithms and data structures that can effectively utilize the hardware.

The inclusion of accelerators in supercomputers has been growing over the past generations of high-performance computing systems. The TOP500, a ranking list of the top supercomputers in the world based on the performance of the LINPACK benchmark [24], has seven of the top ten supercomputers as of November 2020 include an accelerator [25]. Graphic processing units (GPUs) have dominated the current and next generation of U.S. based supercomputers including Summit [26]–[28] and Sierra [29], [30], the second and third place respectively on the TOP500, that each use NVIDIA Tesla V100 GPUs. Usage of the accelerators is essential to achieving performance on these machines as most computational power is from the accelerators. For Summit, 98% of the FLOPs are provided by the GPUs [31]. It is also important to design simulations running on accelerators to be performance-portable across different architectures as the next generation of DOE machines each plan to have GPU accelerators from a different vendor: Perlmutter with NVIDIA GPUs [32], Aurora with Intel X^e [33], and Frontier with AMD GPUs [34]. Maintaining performant versions of simulation codes for all of these architectures as well as future supercomputers is infeasible. Performance-portable tools such as Kokkos [35], and RAJA [36] allow routines to be written once and maintain performance on many different architectures.

Supercomputers with accelerators have two key factors that change how computing is done compared to past architectures. First, is that programming for accelerators is different than for CPUs. CPUs are good for complex computation and branching that melded well to Multiple-Instruction Multiple-Data (MIMD) programming. The main hardware improvements for CPUs is good cache performance by operating on data roughly in the order it was stored in memory. Accelerators have many simpler processing units that operate together requiring Single-Instruction Multiple-Data (SIMD) operations and avoid branching when possible. Hardware performance for accelerators include coalescing memory for different threads and utilizing the SIMD width of the device to correctly align computations. In order to utilize the device properly, different types of algorithms and data structures are required that specifically take advantage of the hardware.

One class of high-performance computing simulation that ports well to GPUs is the Particle-In-Cell (PIC) method [37]–[39]. These methods track particles as they move through the simulation domain. Particles can be simultaneously operated on by the accelerators. Unstructured mesh Particle-In-Cell simulations [40]–[44] use an unstructured mesh to represent the complex geometry of the simulation domain. In this case, particles are tracked as they move through the mesh elements and interact with the fields associated to the mesh and other external background fields. The traditional approach to unstructured mesh PIC is to distribute particles across processes and store a copy of the unstructured mesh on each process. The particle data is extended to store the mesh element the particle is currently within know as the particle's parent element. An alternative method researched as part of this thesis is to distribute both the unstructured mesh and the particles across processes. Instead of the particles maintaining knowledge of the parent mesh element, the particles are stored based on parent element so particles with the same parent element are grouped nearby and accessed through the mesh elements. The Parallel Unstructured Mesh Infrastructure for PIC (PUMIPic) is a library developed to support the implementation of PIC simulations using this alternative method we call mesh-based PIC.

Chapter 4 describes the details of the structures and algorithms that make up the PUMIPic library . The developments towards implementing mesh-based versions of the plasma-physics PIC simulations XGC [42] and GITR [44] using the PUMIPic library are included in Chapter 5.

1.3 Contributions

The research that comprises this thesis targets both the inter-process and intra-process performance for massively parallel simulations. Inter-process performance is targeted through the development of partition improvement methods for balancing and minimizing communications between processes. The EnGPar library was developed by the author that includes a multihypergraph data structure and the implementation of a generalized version of dynamic load balancing methods previously developed by Smith [45]. Details of the EnGPar library can be found in Chapter 2. Developments by the author are as follows:

1. The implementation of the N-graph, a multihypergraph allowing multiple edge types and hyperedges that support connecting two or more vertices per edge.

2. Generalization of Smith's dynamic load balancing algorithm to perform on the N-graph for usage in a larger range of applications than element-partitioned unstructured meshes.

3. Improvements to the most expensive operations of the load balancing algorithm for faster performance.

Intra-process parallelism is explored in the context of unstructured mesh particle-in-cell (PIC) simulations running on computers with GPUs. Developments by the author make up the PUMIPic library for implementing scalable unstructured mesh-based PIC simulations. Additional simulation specific algorithm implementations developed by the author are included. These developments are covered in Chapter 4 that include:

1. Several data structures for storing particles in different formats on accelerators.

2. An extension to Omega_h [46], [47], an accelerator based unstructured mesh data structure, for better performance of particle-mesh interactions.

3. General algorithms for the support of PIC simulations.

4. Implementations of PIC calculations for GPUs to support applications.

1.4 Terminology

CPU	Central Processing Unit: The component responsible for computations and execution of programs with one or more processing units.
Accelerator	a component that accelerates computations through massive on-device parallelism.
SIMD width	The amount of data processed by a single unit of an accelerator.
GPU	Graphics processing unit. An accelerator with a large number of simple processing units that perform synchronized calculations.
2D/3D	two-dimensional or three-dimensional
Unstructured Mesh	A data structure made up of vertices, edges, faces, and regions (for 3D meshes).
Mesh Element	The highest dimensional entity in the mesh. Ex: faces in 2D or regions in 3D
Graph	A data structure made up of vertices connected by edges.
Hypergraph	A graph with hyperedges that connect two or more vertices.
Multigraph	A graph with multiple edges between the same vertices.
Part	A subset of a data structure assigned to a process.
Partition	A set of parts that make up the full data structure.
Cut Hyperedge	A hyperedge is cut if it is on the boundary between two or more parts.
Partition-Model Boundary	The boundary between two parts. Ex: The partition-model boundary of one part of a hypergraph is the set of cut hyperedges.
Degree	The degree of an entity is the number of entities adjacent to it. Ex: The degree of a hyperedge is the number of graph vertices connected by the hyperedge.
Imbalance	A qualitative measure of the computational load of a partition. Calculated as max(load)/average(load).
Ki	suffix denoting a multiplier of 1024. Example 16Ki is $16 * 1024 = 16,384$

1.5 Notation

The following details the notation used in this thesis for describing mesh entities, graph entities, partitions, and associated statistics.

$M = (M^0, ..., M^d)$	A d-dimension mesh with mesh entities of dimension 0 to d.
M_i^d	The ith mesh entity of dimension d.
$\{M_i^d\{M^q\}\}$	The set of mesh entities of dimension q adjacent to the ith mesh entity of dimension d.
$G = (V, H^1, ..., H^t)$	A multihypergraph with vertices V and t sets of hyperedges $H^1, ..., H^t$.
$\{H_i^t\{V\}\}$	The set of graph vertices connected by the ith hyperedge of type t.
$\{V_i\{H^t\}\}$	The set of hyperedges of type t that connect vertex i to other vertices.
$P(T) = (P_0, ..., P_{N-1})$	A unique partition of a set T into N parts such that $\bigcup\limits_{i=0}^{N-1} P_i = T$ and $\forall i, j \in \{0, ..., N-1\}$, if $i \neq j$ then $P_i \cap P_j = \emptyset$.
$w_p(T)$	The sum of weights for a set of entities on process p. Example: $w_p(V)$ is the sum of weights associated to the vertices on process p.
$I_p(T)$	The weighted imbalance of a set of entities T on process p. $I_p(T) = w_p(T)/avg(w_p(T)\|_{p=0,1,2...N-1})$.
$I(T)$	The max weighted imbalance across all processes for the set of entities T.
$\bar{P}_i = P_i \cup B_i$	The ith PICpart composed of part $P_i \in P(M^d)$ and buffer $B_i \subset M^d$ where $P_i \cap B_i = \emptyset$.
S_i	The safe mesh elements of a PICpart where all dependencies are satisfied for operating on particles. $S_i \subseteq \bar{P}_i$.

CHAPTER 2
GENERALIZED MULTICRITERIA DIFFUSIVE LOAD BALANCING ON A MULTIHYPERGRAPH

For simulations utilizing complex relational data structures such as graphs and meshes running on massively parallel supercomputers to execute efficiently, the structures must be partitioned across all processes running such that the computational load associated to the structures' entities is evenly distributed. Partitioning the structures across processes incurs an additional cost in terms of communications between the processes in order to synchronize entities that exist on multiple processes. The partitioning of relational data structures is a well studied problem [4]–[6], [8], [50]–[54] with many tools for creating partitions for use in simulations [55]–[57]. One drawback for these partitioning methods is that they only target one criteria for partitioning. For simulations where multiple criteria need to be satisfied simultaneously the partitions are not satisfactory. This is common in unstructured mesh simulations when degrees of freedom are associated to multiple dimensions of entities for example vertices and edges. In these scenarios partitioning for one entity dimension may lead to a poor balance of the other dimensions.

In evolving simulations where the computational load changes over time, repartitioning the structures is required during execution to maintain even distribution of computation and communication costs [14], [15], [18], [19]. In these cases, re-running static partitioning algorithms is too expensive to be performed repetitively [10]. For better performance, efficient dynamic load balancing strategies must be applied to the evolving structure.

An efficient set of algorithms to solve these challenges use diffusive load balancing techniques. These algorithms perform load balancing across part boundaries and can be applied in succession to perform multiple levels of partitioning. One such diffusive load balancer, ParMA [13], [45], works directly on an element-based unstructured mesh data structure

Portions of this chapter previously appeared as: G. Diamond, C. W. Smith, and M. S. Shephard, "Dynamic load balancing of massively parallel unstructured meshes," in *Proc. 8th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst.*, Denver, CO, USA, Nov. 2017, pp. 9:1–9:7. DOI: 10.1145/3148226.3148236

Portions of this chapter previously appeared as: G. Diamond, C. W. Smith, E. Yoon, and M. S. Shephard, "Dynamic load balancing of plasma and flow simulations.," in *Proc. 8th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst.*, Dallas, TX, USA, Nov. 2018, pp. 73–80. DOI: 10.1109/ScalA.2018.00013

PUMI [58]. ParMA utilizes mesh adjacencies to perform localized balancing through diffusive migration from heavy parts to lighter neighbors. It has been shown that these operations can be efficiently used to perform multi-criteria load balancing in order to improve the partition that is returned by graph/hypergraph and geometric methods.

While ParMA has been used to improve partitions for several simulations using mesh databases, there are applications that have different relation based data structures that have similar partitioning requirements. Two examples of these structures are scale-free graphs often used in computational social science to analyze social phenomena [59], [60] and vertex-partitioned unstructured meshes which are sometimes used in computational fluid dynamic simulations [61]. While the latter example can be handled in ParMA by converting the vertex-partitioned mesh to PUMI's element-partitioned mesh, there are some challenges to appropriately account for the switch and it becomes more difficult to accurately approximate the load of each part. To extend ParMA's capabilities to these other applications, we present EnGPar. EnGPar utilizes an expanded graph structure that we call the N-graph to provide a general representation of relation-based data. Using this graph structure, a new implementation of ParMA's diffusive load balancing algorithm is presented.

Existing multilevel, geometric and diffusive partitioning techniques are reviewed in Section 2.1. Section 2.2 introduces the N-graph that load balancing is performed on. Section 2.3 reviews the load balancing method used in ParMA and how it is generalized to the N-graph to be applied to any relation based structure. New contributions to the diffusive load balancer are described in Section 2.3.3 and Section 2.3.4.

2.1 Partitioning and Load Balancing Methods

In order for applications running with multiprocess parallelism to execute efficiently, partitioning the computational units of work across the processes is required. While partitioning work load to more processors decreases the computational costs per process, an additional cost for interprocess communication is incurred. Hendrickson and Devine [62] list two requirements for effective partitioning techniques: 1) the computational work is well balanced across all processes and 2) the time spent performing interprocess communication is small. Since dynamic partitioning methods are executed at the runtime of the application, Hendrickson and Devine list additional requirements for these methods as: 3) run fast in parallel, 4) memory usage is limited, 5) partitions should be updated incrementally, 6) provide the new communication pattern, and 7) should be easy incorporate in the simulation.

For applications with complex structures where data is highly connected and irregular, managing the computation and communication costs while partitioning is a challenging problem. For structures that can be mapped to a graph, the most common existing techniques include multilevel methods or geometric methods if spatial information is available.

2.1.1 Multilevel (Hyper)Graph Partitioning

Multilevel graph/hypergraph methods are powerful partitioning techniques for creating high quality static partitions. The goal of multilevel partitioners is to distribute the vertices across parts evenly while minimizing the edges cut across parts. Multilevel methods consist of three phases. The first phase is coarsening where the graph is successively reduced in complexity until a threshold is hit. The second phase creates an initial partition on the fully-coarsened graph. The final phase reverses the coarsening operations while maintaining the quality of the partition at each stage. Commonly used multilevel graph partitioners include METIS/ParMETIS [50], PT-SCOTCH [55], and Chaco [56].

For structures with more complex connectivity, graphs do not properly account for data that is not directly linked through one-to-one connections. Hypergraphs improve the representation of the data by using hyperedges or nets that connect two or more vertices together. Multilevel methods applied to hypergraphs have been shown to get better quality partitions for structures that can benefit from hyperedges at the cost of longer time to partition [51]–[53].

Multilevel methods are able to create high quality partitions satisfying the first two requirements listed in Section 2.1. There are two challenges that limit the usability of multilevel graph methods. The first is that the methods become memory intensive at large part counts preventing scaling out beyond tens of thousands of parts [10]. As a result requirement 4) is not satisfied by this class of methods for dynamic load balancing. The scaling issue can be partially avoided by performing what is called local partitioning where each part partitions itself into a new partition creating a larger global partition. Figure 2.1 shows a mesh being partitioned globally into four parts and then locally partitioned into two parts each creating an eight-part partition. The problem with this approach is that each local partitioning will at best maintain the same global quality as the original partition. More likely, each additional level of partitioning results in worse global partition quality.

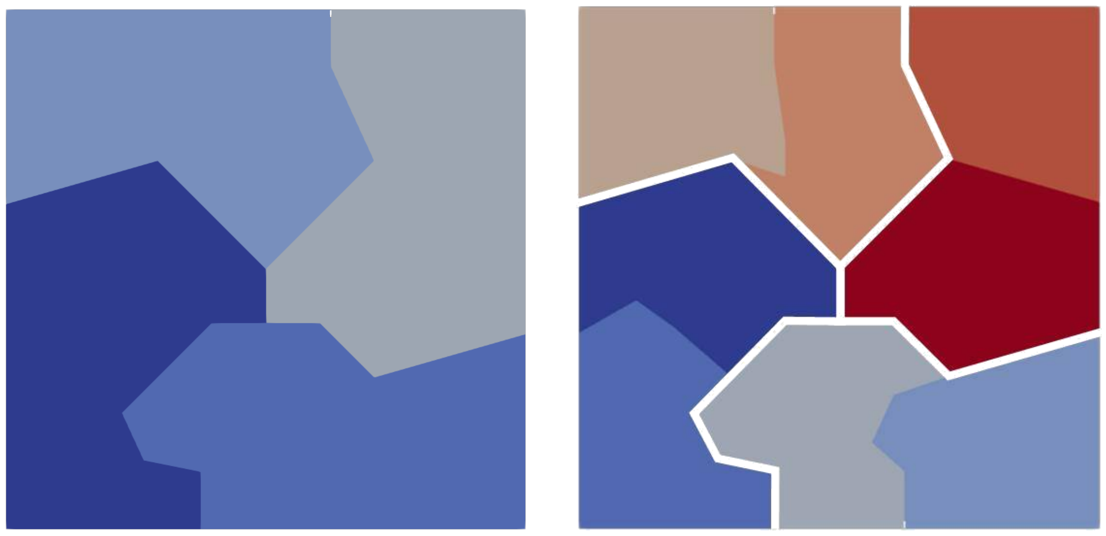

Figure 2.1: A mesh partitioned to four parts using global partitioning (left) and then locally partitioned to eight parts (right).

The second challenge is for applications with a richer data structure than a graph. Multilevel graph methods focus partitioning vertices and minimizing edge cut. If the data structure has computational costs associated with data that are not represented by the graph vertices, then the imbalance of computation across parts will not be properly controlled by the graph partition. This imbalance becomes more severe as the part counts increase. For partitioning unstructured meshes, it has been shown that partitioning mesh elements using a multilevel graph technique has resulted in large mesh vertex imbalances as part size increased [12], [13].

2.1.2 Geometric Partitioning

Geometric methods utilize spatial information tied to the data instead of adjacency information to quickly create partitions. Data is represented with spacial coordinates and the relationship information is defined by distance; closer data is more strongly related. Geometric methods have the advantage of being less memory intensive and computationally cheaper than multilevel methods since only spatial information is required [10].

Common recursive methods, like recursive coordinate bisection [63] (RCB) and recursive inertial bisection (RIB) [64], [65], recursively section the domain to create parts. RCB cuts along coordinate axis and RIB cuts orthogonal to the longest direction. These methods have the same challenge as multilevel techniques as only one criteria is balanced when partitioning.

Space-filling curves (SFC) are another common geometric partitioning technique used [10], [66]. SFC partitioning is a three step operation consisting of encoding, sorting and splitting. Different SFC have been used to perform partitioning for different applications. For unstructured meshes Hilbert and Morton curves are commonly used [67].

Geometric methods are known for their simplicity and fast partitioning of the domain. The main problem with these methods is since there is no representation of connectivity in these methods, communications costs of the resulting partition tend to be much higher than those in multilevel partitions [66]. Similar to multilevel techniques, geometric methods only partition for one goal and can have problems for simulations requiring multicriteria.

2.1.3 Diffusive Partitioning Improvement

Diffusive methods are used to improve partitions through migration of work to neighboring parts. The two categories of diffusive methods are global and local methods. Global diffusion are techniques that make migration decisions based on the entire partition. These methods use global knowledge in order to minimize the total weight transferred or minimize the max weight transferred between any two parts [68]–[71]. Obtaining and managing the information of partitions globally incurs a large communication cost which scales poorly as the number of processes increases.

Local methods make partitioning decisions based on neighborhood information [22], [23], [72]. These methods have less communications required to gather information, but cannot make the high-level partitioning decisions that global methods do. Without global knowledge, poor original partitions can take much longer time to improve. Local methods utilize heuristics to control how work is transferred between processes. Selection based on part quality improvements have had success for partitioning graphs [73], [74]. Label propagation with heuristics have been used to partition graphs with up to one trillion edges [75], [76]. For unstructured meshes, similar heuristics have been used to create partitions for over a million processes [11]–[13].

Since diffusive methods are fast and provide incremental updates to a partition, they are good choices for simulations requiring dynamic load balancing [22], [23]. The usage of multilevel or geometric partitioning methods to create high quality static partitions plus utilizing diffusive partition improvement methods satisfy the requirements of dynamic load balancing of relational data structures[4], [45].

2.2 Partitioned Multihypergraph Representation

EnGPar interfaces to different partitioning procedures through a multihypergraph abstraction called the N-graph. A multigraph [54], [77] is a graph that supports multiple edges between two vertices. A hypergraph allows for hyperedges that connect any number of vertices. The N-graph is defined by a set of vertices V and n sets of hyperedges $H^0, ..., H^{n-1}$. The usage of hyperedges allows for better representation for complex dependencies found in certain relational data structures such as unstructured meshes. Take for instance the construction of an N-graph given a 3D unstructured mesh where mesh elements, M^3, are represented by graph vertices and mesh vertices, M^0, are used for relations between graph vertices. To represent the relationship of mesh vertices with regular graph edges, each graph vertex will have an edge to every other graph vertex for each mesh element surrounding the mesh vertex. With hyperedges, the relationship can be expressed with one hyperedge that connects all of the graph vertices representing the mesh elements surrounding the mesh vertex. To examine these two scenarios, let n be the number of mesh elements that bound a certain mesh vertex. On average in a three-dimensional tetrahedron mesh, n is 23 [78]. To represent this relation between all n graph vertices with regular graph edges would require $\frac{n(n-1)}{2}$ edges. However, when using hyperedges one graph edge is created to represent the mesh vertex with n connections between the graph vertices and hyperedges. This results in a reduction in memory usage. Additionally, the usage of hyperedges has been shown to more accurately represent the communications associated with mesh vertices for usage in partitioning [51], [52]. Figure 2.2 gives an example 2D mesh where seven mesh elements bound a mesh vertex (a). A graph using regular graph edges to represent the mesh is shown in (b). In (c) the mesh is shown in the N-graph format using a hyperedge. In all figures of hypergraphs, hyperedges are represented by boxes and lines stem from the box to the vertices that are connected by the hyperedge. In (b) 21 graph edges are created for the one mesh vertex while in (c) one hyperedge is added and seven connections between the graph vertices and hyperedge.

Supporting multiple edge types in the N-graph allows representing multiple layers of connection between the graph vertices. This is useful to represent applications that use multidimensional data or complex levels of communication. One example is an unstructured

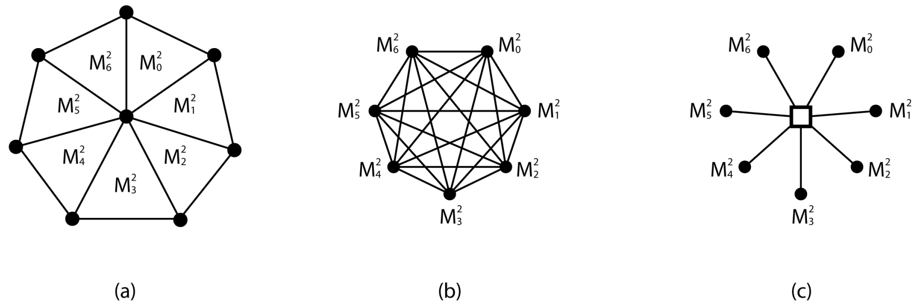

Figure 2.2: (a) A seven triangle mesh surrounding one mesh vertex. (b) A graph using traditional edges to connect the graph vertices whose corresponding mesh elements share the mesh vertex. (c) The N-graph construction using a hyperedge for the mesh vertex and connecting the adjacent mesh elements in the N-graph.

mesh which have mesh vertices, mesh edges, and mesh faces (in three dimensions) that are shared between mesh elements. To represent these data structures for a range of application needs, the N-graph supports the arbitrary use of edge types to allow different configurations for applications. Figure 2.3 depicts the mapping of a 2D unstructured mesh (a) to a representation where mesh elements map to graph vertices and mesh vertices map to graph hyperedges (b). In (c) a second mapping is shown where mesh edges are also mapped to a second edge type in the graph. The labels of the entities in (a) are carried to (b) and (c) to show which mesh entity is represented by the corresponding graph entity.

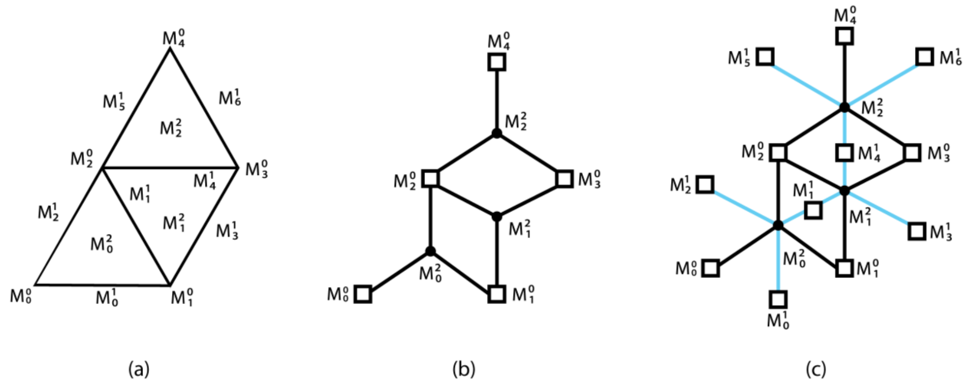

Figure 2.3: (a) A 2D unstructured mesh. (b) N-graph construction with elements→vertices, vertices→hyperedges. (c) Additional mesh edges are used for a second set of hyperedges. Mesh labeling is shared in all three to correlate mesh entities to graph entities.

2.3 Diffusive Load Balancing Method

EnGPar uses the diffusive load balancing techniques from ParMA [45] on the N-graph structure to improve the partition of any relational data structure that can be mapped to the N-graph. The method is to iteratively perform a series of steps until user specified criteria are met or the algorithm can not improve the partition quality further. The metric of interest in ParMA is the load metric used for partition quality is the imbalance, $I(M^d)$, of a set of mesh entities of dimension d defined as the max weight across all processes divided by the average weight. For example, when equal weights are used, the imbalance of mesh elements is the maximum number of mesh elements on a single part divided by the average number of mesh elements per part, across all processes. Algorithm 1 lists a general framework for the multi-criteria load balancing procedures in ParMA. These steps are equivalently implemented on the N-graph for use in EnGPar.

Algorithm 1 ParMA Load Balancing Framework

```
1: procedure BALANCE(mesh,dimensions)
2:     for all $d \in dimensions$ do
3:         while imbalance of $d >$ tolerance do RUNSTEP(mesh,$d$)
4:             if Balancing Stagnates then
5:                 break
6:             end if
7:         end while
8:     end for
9: end procedure
10: procedure RUNSTEP(mesh,$d$)
11:     -Determine the neighboring parts and size of part boundaries.
12:     -Compute the weight of the entities in dimension $d$
13:     -Share the computed weight with each neighbor.
14:     -Determine which neighbors that can receive more weight.
15:     -Calculate how much weight to send to each neighbor.
16:     -Construct an ordering of vertices to traverse the part boundary.
17:     -Create migration plan that reduces the imbalance of dimension $d$.
18:     -Adjust the plan to maintain balance of previous dimensions.
19:     -Perform Migration
20: end procedure
```

ParMA's load balancing begins by calling the BALANCE procedure on lines 1-9 given the *mesh* and a set of priorities to balance in *dimensions*. For each priority, the RUNSTEP procedure is iteratively called for the current priority on lines 3-7 until the imbalance is reduced below the tolerance or stagnation is detected. The RUNSTEP procedure performs six stages to determine the diffusive load balancing of each step. The first step on line 11 determines who are the neighbors of each part and what is the size of the part boundaries between them. The second computes the weight of the target dimension d and sends the weight to the neighboring parts on lines 12 and 13. Then, the heavy parts decide which neighbors to send weight to and how much to send on lines 14 and 15. The fourth stage constructs an ordering of the entities on the boundary for selecting on line 16. Lines 17 and 18 select entities to be migrated to the neighboring parts. The selection of entities is covered in more detail in Section 2.3.1. Finally, migration is performed to send the selected entities to neighboring parts on line 19.

The *dimensions* input to the BALANCE procedure is an ordering of the criteria to be balanced. It is interpreted such that earlier entries have higher priority and thus will be balanced first and the reduced imbalance will be maintained in following steps. Line 18 of Algorithm 1 adjusts the migration plan to ensure that the imbalance of completed dimensions is not increased. An example of multi-criteria load balancing is the "vertex > element" case for finite element methods when degrees of freedom are defined by the mesh vertices. The degrees of freedom contain the largest portion of computational work in these simulations and thus, make balancing the mesh vertices a top priority. However, these simulations also require balancing the mesh elements for efficient linear system assembly. In this example mesh vertices would be the first criteria followed up by mesh elements.

An additional metric of interest in EnGPar is the communication metric. The communication metric is evaluated based on the cut hyperedges in the partition. There are different ways to define this cost [79]. Here we count the number of cut hyperedges to measure the communication. A higher number of cut hyperedges results in more entities that will require communications in the simulation. In general, EnGPar will be applied to partitions created by multilevel methods that result in a very low edge-cut. So, the goal in EnGPar will be to perform multicriteria load balancing with minimal increase to the edge-cut.

Additional consideration to improve the multi-criteria load balancing in EnGPar is focused on two aspects of this algorithm. One is the generalization to support a larger range of applications discussed in Section 2.3.2. The second is an improvement to one of the more computational expensive operations in Section 2.3.3. A third addition to ParMA's method is to control the edge-cut when the degree of graph vertices is larger that is described in Section 2.3.4.

2.3.1 Selection

The selection step of the diffusive algorithm, lines 17-18, is where decisions are made on which entities to migrate across part boundaries. At this stage, each process has determined a set of targets that include the neighboring parts this process will send weight to and how much weight to select. The selection decisions are made by constructing cavities defined by a hyperedge H_i^t that is cut on the boundary of two or more parts. The cavity includes all of the on-part graph vertices in $\{H_i^t\{V\}\}$. Figure 2.4 depicts a hyperedge cut across three parts and the surrounding graph vertices connected by the hyperedge. The color of each graph vertex depicts the part it belongs to. The cavity for part A would have size 4 and could be selected to migrate to either parts B or C. Similar cavities exist on parts B and C for the cut hyperedge with sizes 2 and 3 respectively.

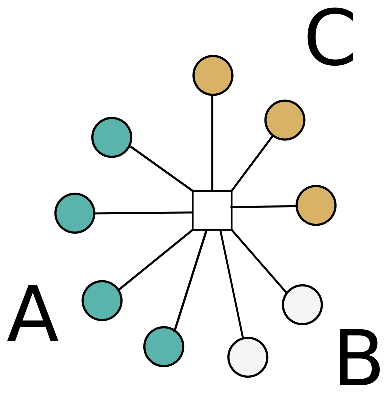

Figure 2.4: A hyperedge cut across three parts A, B, and C. The cavity for each part has size 4,2, and 3 respectively.

The cavities are sorted in two levels that aim to perform better migrations first. The first sort is a topological distance sort of all cavities based on the distance from the topological center of the part to the cut hyperedges. The goal of this sort is to reduce the edge-cut of the part and diameter of the part by creating *rounder* parts with minimal topological surface area [80], [81]. The topological sort is covered in more detail in Section 2.3.3 along with an improvement to the algorithm used in ParMA. The second sort is by lower on-part degree of the hyperedge first. Migrating cavities which have fewer on-part graph vertices first will generally have a smaller increase to the edge-cut than high degree hyperedges. This greedy heuristic is true when the degree of graph vertices is fairly uniform. In the case of ParMA, which operates on element-partitioned unstructured meshes, the degree of the mesh element is constant based on the element type, e.g. 4 mesh vertices per tetrahedron, 5 mesh vertices per pyramid, 8 mesh vertices per hexahedron etc. When the degree of graph vertices is non-uniform, an additional check is used in EnGPar to control the increase in edge-cut. This heuristic is described in Section 2.3.4.

After the cavities are sorted, the cavities are iterated over checking if any of the neighboring processes that the hyperedge is cut across are in the set of targets and still need more weight. If so, the cavity is selected that will set all graph vertices of the cavities to be migrated to the neighboring process and the weight of those vertices is subtracted from the target's weight. Cavities are selected for migration until enough weight is selected for each target or if all cavities have been processed.

2.3.2 Generalization of Multicriteria Load Balancing

Since ParMA works directly on PUMI [58], it can only support applications that can map the structure to the PUMI element-partitioned mesh. Applications that use vertex-partitioned meshes can be converted to the PUMI mesh, however there are challenges with accurately expressing the partition to ParMA which cause the resulting partitions to have worse quality than those produced for an element-partitioned application. Other relational structures such as scale-free graphs [59], [60] cannot be expressed as an unstructured mesh and would not be able to utilize ParMA. Since EnGPar is built on the N-graph, it natively supports other relational data structures that can be expressed as a graph or hypergraph without loss of the power of ParMA's use of adjacencies for fast dynamic load balancing.

Beyond supporting other forms of data, many applications have different priorities of balancing the different criteria. While ParMA is built to perform load balancing targeting finite element applications, EnGPar takes a more general approach allowing users to define the ordering and tolerances of criteria. This is highlighted in Algorithm 1 on line 11 with the *dimensions* argument. This argument controls the ordering in which dimensions are balanced with earlier dimensions having higher priority. In ParMA these inputs would be defined by the balancer that is applied. Meaning that for ParMA to support additional applications new balancers need to be defined for the user to use. In EnGPar the users can use any ordering of priorities without any new development. This is done by providing a richer user interface with inputs to control all necessary components of the diffusive balancing procedure. Examples of using this interface are provided in Appedix 6.2.3.

2.3.3 Graph Distance Computation

One of the optimizations in ParMA targets decreasing surface area across parts by ordering the selection of mesh elements to migrate based on their topological distance from the topological center of the part. The topological center of a part is the set of vertices that are farthest from all partition model boundaries for that part. The depth of the part is the topological distance from the topological center to the partition model boundary. A smaller depth results in less edge-cut [45], [80]. Towards decreasing the depth of the part, entities that have the greatest topological distance from the topological center are migrated first.

Several challenges arise in determining entities with the largest topological distance due to the existence of disconnected components in the part. Disconnected components are generally bad for partitions as there will be more partition model boundaries leading to a higher edge-cut and more communications required. Ideally, we would like to have one connected component or at a minimum decrease the size of other disconnected components so the effect on edge-cut is minimized. To get an optimal ordering, the topological distances of the disconnected components are offset such that components with lower depth get higher priority. This leads to lower depth components being migrated first.

ParMA computes the graph distance using multiple breadth-first traversals (BFTs); one traversal to find disconnected components, another to locate the topological centers of each component, and another to compute the distance to each mesh vertex on the partition model boundary. EnGPar reduces the number of traversals from three to two. This is done by locating the disconnected components during the traversal to compute distances using a disjoint set data structure [82], [83].

The disjoint set data structure is made up of sets that begin disjoint and are joined together by the algorithm operating on them. The structure requires three operations to function: MakeSet creates a new disjoint set, Union joins two disjoint sets together with one set as the primary set, and Find that determines the primary disjoint set of a disjoint set. For N disjoint sets, the runtime of each operation is $O(1)$ for MakeSet and $O(\log(N))$ for Union and Find. To aid the usage of disjoint sets in EnGPar's distance computation an additional operation is defined named Add that adds an element not in any disjoint set to a disjoint set in $O(1)$ time. This is important to note as this operation does not increase the size N for the runtime of Union and Find.

EnGPar's distance computation begins with determining the depths of every hyperedge in the part. Algorithm 2 lists the procedures for this operation. The algorithm begins with the ComputeCenters procedure on lines 1-8. The procedure takes in the graph, G, and the hyperedge type, t, used for creating cavities. The ComputeCenters procedure's primary operation is a BFT on line 6. The starting set for the BFT are all cut hyperedges of type t gathered on line 4. The BFT procedure on lines 9-23 iterates over a queue of hyperedges to be visited on lines 11-22. Each iteration pops the first hyperedge from the queue, line 12 and if the hyperedge has not been visited in a previous iteration then it is visited on lines 13-21. When a hyperedge is visited, the second adjacency hyperedges are iterated by a double for loop, first over $\{H_i^T\{V\}\}$ on line 15 and then over $\{V_i\{H_j^t\}\}$ on line 16. Each second adjacent hyperedge executes a *kernel* on line 17 in order to update any external values for the BFT. For computing the depths the DepthKernel on lines 24-28 is used that sets the depth of the second adjacent hyperedge, H_j^t, to one more than the depth H_i^t if the depth has not been set previously. The final step for the BFT iteration is to add the second adjacent hyperedges to the end of the queue on line 18. After the BFT is completed, every hyperedge will have its depth set to be the smallest distance from any cut hyperedge. The hyperedges are then sorted at the end of ComputeCenters on line 7. Figure 2.5 shows the depth values of every hyperedge for a graph with three disconnected components. The cut hyperedges are denoted with filled boxes. The topological centers of each component, the hyperedges with greatest depth, are represented as boxes with an X in them. Note that at this stage the disconnected components have yet to be determined and the noting of X is for clarity.

To compute the topological distances of each cut hyperedge, the depths are used to determine the starting point for the second BFT. For this BFT, a secondary disjoint set structure is maintained to aid in distinguishing the disconnected components. Algorithm 3 provides pseudocode for the distance computation. The ComputeDistance procedure on lines 1-14 begins by retrieving the *levels* array that contains the list of hyperedges sorted by highest depth first from the ComputeCenters procedure on line 2. Initialization of the arrays for the BFT and a *start_distance* value to 0 is on lines 3-5. The hyperedges that make up the topological center for the deepest disconnected components are determined in the SetSeedEdges procedure on line 6. This procedure on lines 15-29 finds the first hyperedge in the *levels* list that has not been visited and stores the depth of the hyperedge as *max_depth* on lines 17-22. At first, this will be the first hyperedge as no hyperedges have been visited yet and since the array is sorted *max_depth* will be set to the largest depth of

Algorithm 2 Find Topological Center

```
 1: procedure ComputeCenters($G$, $t$)
 2:     $depth \leftarrow array(|H^t|, -1)$
 3:     $visited \leftarrow array(|H^t|, false)$
 4:     $init \leftarrow$ cut hyperedges $\in H^t$
 5:     foreach $H_i^t \in init$: $depth(H_i^t) \leftarrow 0$
 6:     BreadthFirstTraversal($init$,$G$,DepthKernel)
 7:     return $levels \leftarrow$ Sort($G$)        ▷ $levels$ is an array of hyperedges in order of decreasing depth
 8: end procedure
 9: procedure BreadthFirstTraversal($seed$, $G$, $kernel$)
10:     $Queue \leftarrow seed$
11:     while $Queue$ is not empty do
12:         $H_i^t \leftarrow Queue.pop()$
13:         if not $visited(H_i^t)$ then
14:             $visited(H_i^t) \leftarrow true$
15:             for $V_i \in \{H_i^t\{V\}\}$ do
16:                 for $H_j^t \in \{V_i\{H^t\}\}$ do
17:                     $kernel(H_i^t, G)$
18:                     $Queue.add(H_j^t)$
19:                 end for
20:             end for
21:         end if
22:     end while
23: end procedure
24: procedure DepthKernel($H_i^t$, $H_j^t$)
25:     if $depth(H_j^t) == -1$ then
26:         $depth(H_j^t) \leftarrow depth(H_i^t) + 1$
27:     end if
28: end procedure
```

any hyperedge. Then lines 23-27 gathers every hyperedge with $depth = max\_depth$ that has not been visited into *init*. The *init* set will be the initial queue of hyperedges for the second BFT.

After the *init* set is created by SetSeedEdges, the disjoints sets structure is initialized on line 8 where a disjoint set is created for each hyperedge in *init*. Then, the second BFT is performed on line 10 using the same BreadthFirstTraversal procedure in Algorithm 2, but with the DistanceKernel on lines 30-39. This kernel has additional operations for updating the disjoint sets. For the visited hyperedge, H_i^t and a second adjacent hyperedges, H_j^t, there are two cases that need to be handled: H_j^t is not in a disjoint set or H_j^t is in a different set then H_i^t. The first case is handled on lines 31-32 where H_j^t is added to the disjoint set of H_i^t. The second case requires using the disjoint set's Union function to join

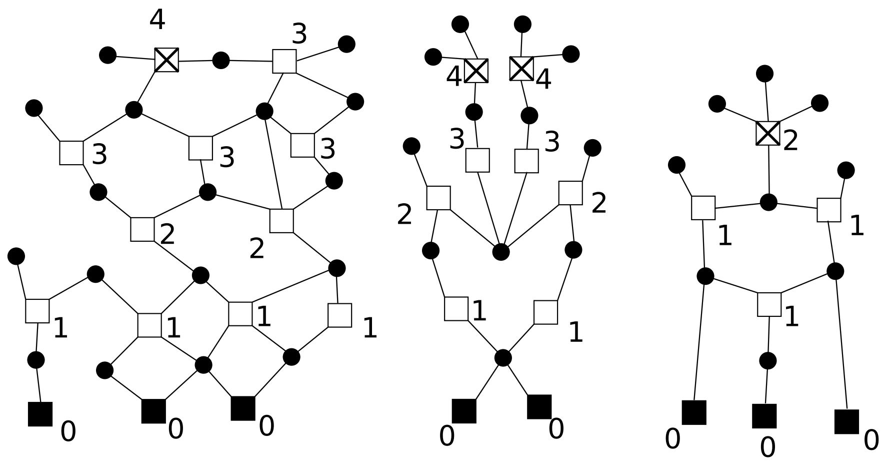

Figure 2.5: The result of executing the ComputeCenters procedure on a hypergraph part with three disconnected components. Filled boxes represent cut hyperedges and boxes with an X are the topological centers for each component. The depths of each hyperedge is listed adjacent to the boxes.

the sets of H_i^t and H_j^t on lines 33-34. The joining of two sets correlates to two of the initial deepest hyperedges being on the same component. There is technically a third case when H_j^t is in the same set as H_i^t, but in this case nothing needs to be done. Besides the logic to handle the disjoint sets, the distance of the hyperedges is updated the same as in the DepthKernel on lines 36-38.

With the completion of the BFT all hyperedges in the same components as the hyperedges in *init* will have their distances calculated. The number of remaining disjoint sets as this stage is equal to the number of disconnected components with the same max depth. Components with the same depth are distinguished by offsets the distances such that the component with the fewest hyperedges are given highest priority in the UpdateDistance procedure on lines 40-49. First, the disjoint sets are sorted by decreasing number size of the sets on line 42. Then each disjoint set is processed to offset the distances of every hyperedge within the set on lines 44-46. The distances are increased by *distance_update−start_distance* where *distance_update* is the max distance value of the previously processed disjoint set. For the first disjoint set none of the distances are increased, but subsequent disjoint sets will be updated due to the changing value of *distance_update* on line 47.

For disconnected components with a lower depth than the max, the steps from lines 6-13 are repeated. After the completion of each iteration, the *start_distance* is updated to one more than the max distance of the visited hyperedges. This results in the disconnected components with lower depth to have a higher distance. In successive iterations the SetSeedEdges procedure will return the next set of unvisited hyperedges with highest depth for a new call to BreadthFirstTraversal. While multiple calls to BreadthFirstTraversal procedure are required in the second BFT, they total to visiting each hyperedge in the graph once so the total runtime is equivalent to a single BFT of the full part. Figure 2.6 depicts the distance computation on the three disconnected components from the previous figure. The distances of each hyperedge is noted next to each hyperedge. The left and middle components have the same depth so both are processed in the same call to BreadthFirstTraversal. Three disjoint sets are created, one for the left component and two for the middle component. While processing the second adjacencies through the graph vertex labeled "union", the two disjoint sets of the middle component are joined to form one set resulting in one set for each component at the end of the BFT. The distances of the middle component are increased because it is smaller. The right component is processed in a second iteration of the while loop with *start_distance* = 11 and one new disjoint set is created.

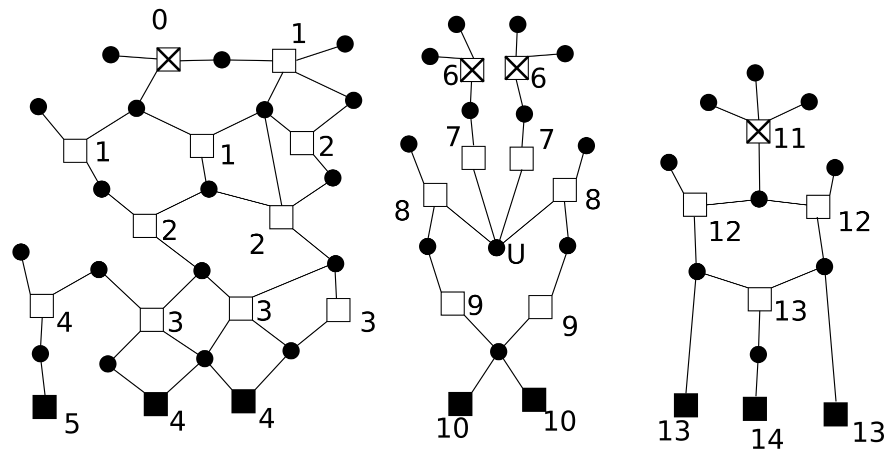

Figure 2.6: The result of executing the ComputeDistance procedure on the hypergraph part from 2.5. Distances of each hyperedge is noted adjacent to each box. The vertex labeled U represents the moment the two disjoint sets of the middle component are unioned to form one set for the component.

2.3.4 Mitigating Edge-Cut Increases

One drawback from performing the diffusive load balancing routine is that there may be an increase in the communication cost of the partition. This is especially common when the

Algorithm 3 Compute Topological Distance

```
1: procedure ComputeDistance($G$, $t$)
2:     $levels \leftarrow$ ComputeCenters($G, t$)
3:     $distance \leftarrow array(|H^t|, -1)$
4:     $visited \leftarrow array(|H^t|, false)$
5:     $start\_distance \leftarrow 0$
6:     while $init \leftarrow$ SetSeedEdges($visited$,$levels$) do    ▷ Label the vertices with the largest remaining depth. If none remain, then break from the while loop.
7:         $disjoint\_sets \leftarrow \emptyset$
8:         foreach $H_i^t \in init$: $disjoint\_sets \leftarrow$MakeSet($H_i^t$)
9:         foreach $H_i^t \in init$: $distance(H_i^t) \leftarrow start\_distance$
10:        BreadthFirstTraversal($init$,$G$,DistanceKernel)
11:        UpdateDistance($disjoint\_sets$, $start\_distance$)
12:        $start\_distance \leftarrow max\{distance(H^t)\} + 1$
13:    end while
14: end procedure
15: procedure SetSeedEdges($visited$,$levels$)
16:    $init \leftarrow \emptyset$
17:    for $H_i^t \in levels$ do
18:        if not $visited(H_i^t)$ then
19:            $max\_depth \leftarrow depth(H_i^t)$
20:            break
21:        end if
22:    end for
23:    for $H_i^t \in levels$ do
24:        if not $visited(H_i^t)$ and $depth(H_i^t) == max\_depth$ then
25:            $init \leftarrow init \cup H_i^t$
26:        end if
27:    end for
28:    return $init$
29: end procedure
30: procedure DistanceKernel($H_i^t$, $G$, $Queue$)
31:    if not Find($H_j^t$) then
32:        Add(Find($H_i^t$), $H_j^t$)
33:    else if Find($H_i^t$) != Find($H_j^t$) then
34:        Union($H_i^t$,$H_j^t$)    ▷ merge sets into one set
35:    end if
36:    if $distance(H_j^t) == -1$ then
37:        $distance(H_j^t) \leftarrow distance(H_i^t) + 1$
38:    end if
39: end procedure
40: procedure UpdateDistance($disjoint\_sets$, $start\_distance$)
41:    $distance\_update \leftarrow start\_distance$
42:    $DS = Sort(disjoint\_sets)$    ▷ Sort by decreasing size of sets
43:    for $D \in DS$ do
44:        for $H_i^t \in D$ do
45:            $distance(H_i^t) \leftarrow distance(H_i^t) + distance\_update - start\_distance$
46:        end for
47:        $distance\_update = \max\{distance(H_i^t \in D)\} + 1$
48:    end for
49: end procedure
```

original partition is created by a multilevel graph method as the main objective for multilevel methods is to minimize the edge-cut in the graph. In ParMA, the heuristics for migrating cavities by distance from the topological center and prioritizing smaller cavities over larger cavities is sufficient because the degree of mesh elements, the number of lower dimentional entities bounding a mesh element, is uniform based on the type of the element. Even in a mixed mesh, where multiple types of mesh elements are used, the difference in degree is relatively small and will not have a large impact on the results of the partition. For EnGPar, the N-graph allows supporting structures that will not have uniform degree. One such case, is for vertex-partitioned meshes where the degree of mesh vertices can change drastically throughout the mesh, especially in the case of a mixed mesh. For a 3D tetrahedral mesh, the average number of mesh elements adjacent to a mesh vertex is 23 [78]. The N-graph will have graph vertices with varying degree meaning that decisions on migrating cavities has to take this into account otherwise poor decisions could result in a large increase to the edge-cut and significantly more communications in the simulation.

To control the edge cut while balancing graphs with varying degree vertices, a new test on the change of edge-cut is introduced in the selection step that cavities must pass in order to be selected for migration. The test is evaluated by creating the set of second adjacent hyperedges from the cut hyperedge that defines the cavity. Namely, for a cavity defined by the cut hyperedge H_i^t with vertices $\{H_i^t\{V\}\}$, the second adjacent hyperedges are defined as $\{H_i^t\{V\{H^t\}\}\}$. When a cavity is migrated, H_i^t will no longer be a cut hyperedge, but the hyperedges in $\{H_i^t\{V\{H^t\}\}\}$ that are not cut hyperedges will become cut hyperedges. The desire is to select cavities that have the fewest non-cut second-adjacent hyperedges. To achieve this, the cut ratio is defined as the ratio of non-cut hyperedges to cut hyperedges in $\{H_i^t\{V\{H^t\}\}\}$. If the cut ratio is small, then the increase to the edge-cut from migrating the cavity will be smaller than a cavity with a larger ratio. A parameter β is defined such that any cavity with a ratio greater than β will not be selected for migration. This test limits the load balancing that can be performed based on how restrictive *beta* is set, but ensures that the edge-cut will not grow uncontrollably.

2.4 Summary

EnGPar provides a generalized appraoch to partition improvement through the multi-hypergraph structure called the N-graph. The diffusive load balancing methods previously used in a parallel unstructured mesh library have been generalized to be applied on a wider range of parallel data structures in EnGPar. Further improvements to the efficiency of the algorithm's distance computation were achieved through the use of disjoint sets. A new test was created in the selection phase of the algorithm to mitigate the increase in edge-cut as the imbalance is decreased for graphs with high degree vertices.

CHAPTER 3
IMPROVING UNSTRUCTURED MESH PARTITIONS USING DIFFUSIVE MULTICRITERIA GRAPH LOAD BALANCING METHODS

Massively parallel unstructured mesh simulations such as finite element [1] and finite volume [2], [61] methods require efficient approaches to partitioning the mesh entities such that computation costs are well balanced across processes and communication costs are minimized. For a partitioned unstructured mesh, the computation costs are related to the number of mesh entities on a process that work is performed on. The communication costs are attributed to the mesh entities that are shared between processes that require updates. There are several factors that drive the partitioning requirements for unstructured meshes including how the mesh is stored in memory, the dimensions of mesh entities that make up the computation and communication costs, the need for mesh adaptation [84]–[86] and differences in the meshing of the domain such as boundary layers [87]–[89]. Multilevel or geometric partitioning methods are sufficient tools for creating good quality partitions for unstructured mesh applications when a single load balancing criteria is considered. When there are multiple mesh entity dimensions that contribute to the computation costs additional partition improvement methods must be applied to create efficient partition especially as the number of processes grows [10]. The extreme of this case is when computational work is associated to each dimension of mesh entities such as in higher-order finite element simulations [90]–[92].

In this chapter, we apply EnGPar to improve the partitions created by multilevel methods for cases stemming from unstructured finite element and finite volume simulations. Section 3.1 introduces the three cases that EnGPar is applied to, namely an element-partitioned mesh, vertex-partitioned mesh, and higher-order finite element case. For each case the con-

Portions of this chapter previously appeared as: G. Diamond, C. W. Smith, and M. S. Shephard, "Dynamic load balancing of massively parallel unstructured meshes," in *Proc. 8th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst.*, Denver, CO, USA, Nov. 2017, pp. 9:1–9:7. DOI: `10.1145/3148226.3148236`

Portions of this chapter previously appeared as: G. Diamond, C. W. Smith, E. Yoon, and M. S. Shephard, "Dynamic load balancing of plasma and flow simulations.," in *Proc. 8th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst.*, Dallas, TX, USA, Nov. 2018, pp. 73–80. DOI: `10.1109/ScalA.2018.00013` struction of the N-graph for the unstructured mesh representation is described. Section 3.2 compares EnGPar to its predecessor ParMA for improving the partitions of an element-partitioned unstructured mesh. Section 3.3 applies EnGPar to a mixed vertex-partitioned mesh. A final case for higher-order finite element simulation in Section 3.4 compares two different definitions of the N-graph that can be used by EnGPar to target the balance of degrees of freedom.

3.1 Representing Unstructured Mesh Partitions as an N-graph

Parallel unstructured mesh simulations like finite volume and finite element methods distribute the unstructured mesh in different ways to efficiently resolve data dependencies. A partition of a mesh is typically defined by one dimension of mesh entities that is uniquely partitioned. Other dimensions of mesh entities are duplicated as needed on the partition model boundary to maintain the closure of mesh elements. For example, two common partitions are element partitions and vertex partitions. In an element-partitioned mesh the mesh elements are uniquely partitioned and lower dimension entities are duplicated to form the closure of the on-part mesh elements. A vertex-partitioned mesh uniquely assigns vertices to parts while the elements, and their closure, are copied along the boundary to any process that shares them.

EnGPar can support any type of mesh partition using the general N-graph structure. For a mesh partition, that uniquely partitions dimension d, for every M_i^d in the mesh, a graph vertex V_i is constructed. The hyperedges in the N-graph are constructed based on the dimensions of mesh entities that require load balancing, have important simulation data dependencies, or mesh entities that require communications. For mesh dimension $t \neq d$, the set of hyperedges H^t will be constructed with one hyperedge H_j^t for every mesh entity M_j^t. Hyperedge H_j^t will connect the set of vertices $\{H_j^t\{V\}\}$ such that for every $M_i^d \in \{M_j^t\{M^d\}\}$, $V_i \in \{H_j^t\{V\}\}$. Multiple different dimensions of mesh entities can be used to construct multiple sets of hyperedges up to including every dimension in the N-graph if a simulation requires balancing all dimensions of the mesh.

In this chapter, three different unstructured mesh cases are presented for EnGPar to improve the partitions of: an element-partitioned mesh, a vertex-partitioned mesh, and a higher-order finite element case on an element-partitioned mesh. Figure 3.1 depicts a 2D mixed mesh of triangles and a quadrilateral (a) along with the N-graphs for each of the three cases (b-d). For the element-partitioned mesh case (b), the mesh elements are represented as graph vertices with mesh vertices represented as hyperedges connecting the mesh elements that are bounded by each mesh vertex. This case is tested in Section 3.2. The vertex-partitioned mesh (c), has graph vertices representing the mesh vertices and hyperedges representing the mesh elements connecting the mesh vertices which bound the same mesh element. The vertex-partitioned mesh case is covered in Section 3.3.

The higher-order finite element case is a more expanded version of the element partitioned case. In this case, the mesh elements are again represented by graph vertices and the mesh entities with degrees of freedom (DOF) also named DOF holders are represented by hyperedges. With the flexibility of the N-graph there are different ways this can be represented in EnGPar. Two approaches are explored, the first is one hyperedge type for each dimension of DOF holders in the mesh. In Figure 3.1 (d) the N-graph has one hyperedge type for mesh vertices labeled as empty boxes and a second for mesh edges depicted as filled boxes. This first approach would be the approach used in ParMA with a "vertex=edge>element" balancer. An alternative approach supported with the N-graph is to treat all DOF holders as one hyperedge type. In this case both the empty and filled boxes would be one hyperedge type for the graph in (d). Both approaches to improving the higher-order finite element partitions are compared in Section 3.4.

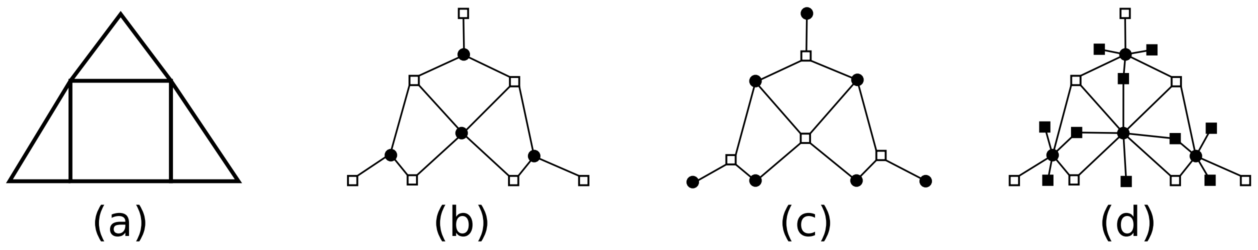

Figure 3.1: An unstructured mesh of triangles and a quadrilateral (a), the N-graph for an element-partitioning of the mesh with mesh vertices as hyperedges (b), the N-graph for a vertex-partitioning of the mesh with mesh elements as hyperedges (c), the N-graph for a higher-order element-partitioning of the mesh where two hyperedge types are used one for mesh vertices (empty boxes) and one for mesh edges (filled boxes) (d).

3.2 Improving Element-Partitioned Meshes

Tests for the finite element case were run with a one billion element tetrahedral mesh of an airplane's vertical tail structure. EnGPar was run on the Mira BlueGene/Q system at the Argonne Leadership Computing Facility [93] on partitions from 128Ki ($128 * 2^{10}$) up to 512Ki parts. These partitions were created by using ParMETIS part k-way [9] globally up to 8Ki parts. Then METIS is run on each part locally to create the 128Ki to 512Ki partitions. The initial mesh element imbalance is 2% and the mesh vertex imbalance ranges from 12% for the 128Ki partition up to 53% at 512Ki parts.

We compare EnGPar to ParMA [13], a diffusive load balancer that works directly on an unstructured mesh. Each tool is run on the partitions with the goal of balancing the mesh vertices down to 5% while keeping the mesh element imbalance below 5%. Figures 3.2 shows the mesh vertex imbalance from ParMETIS and after EnGPar and ParMA are used. Both tools significantly reduce the mesh vertex imbalance. For the 128Ki and 256Ki cases, both reduce the imbalance to the target 5%. In the 512Ki case, EnGPar reduces the mesh vertex imbalance to 6% while ParMA reduces to the target.

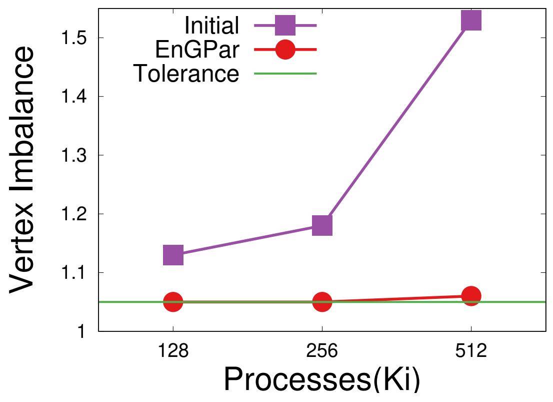

Figure 3.2: Vertex imbalance for the initial partitioning and the partitions created by EnGPar and ParMA. Element imbalance is maintained below the 5% tolerance for all cases. ©2018 IEEE.

Figure 3.3 shows the runtime for each partition of ParMA and EnGPar. The timing for EnGPar includes the construction of the N-graph and subsequent repartition of the mesh after running EnGPar to fairly compare with ParMA which doesn't require any conversions of the mesh. In all cases EnGPar runs faster. The speedup ranges from 25% faster at the 256Ki case up to 54% faster for the 512Ki partition.

Table 3.1 shows the average number of vertices in the mesh for each case. This mea-

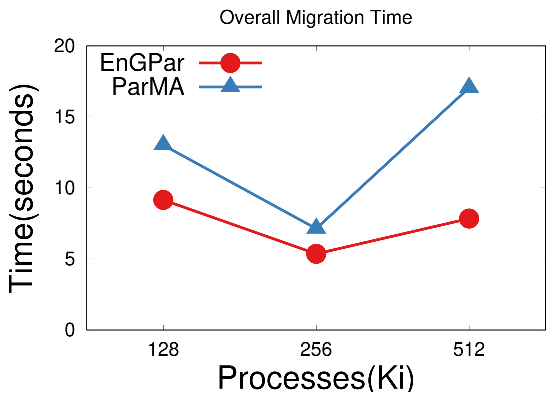

Figure 3.3: Time to balance for EnGPar and ParMA. ©2018 IEEE.

surement is related to edge cut in the graph since more average vertices means more surface area of the part and higher edge cut. ParMA slightly reduces the vertex counts in every case. EnGPar increases slightly by around 1% for each case. As was mentioned for the element-partitioned mesh, the edge cut limit metric was not required to avoid large increases in edge cut.

Table 3.1: Average number of mesh vertices per part. ©2018 IEEE.

	128Ki	256Ki	512Ki
Initial	2146.404	1138.881	611.673
ParMA	2141.965	1137.343	610.959
EnGPar	2148.310	1143.970	619.177

3.3 Improving Vertex-Partitioned Meshes

Experiments for the vertex-partitioned mesh case are performed on a 3D mixed unstructured mesh. In the N-graph constructed for the vertex-partitioned meshes, the degree of graph vertices is nonuniform. As such, the technique to mitigate increases in edge-cut introduced in Section 2.3.4 is used for this case. An additional feature in this mesh that is factored into the creationg of the N-graph is the existance of boundary layer stacks of prism elements growing from the geometric model faces towards the interior. In Section 3.3.1 a method for collapsing each the stacks of boundary layer mesh entities into a smaller set of N-graph entities is discussed. We will analyze applying EnGPar to both the collapsed boundary layer case and the case where the boundary layer is treated the same as the rest of the mesh in Section 3.3.2.

3.3.1 Boundary Layer Stacks

Semi-structured boundary layer element stacks growing from geometric model faces can be used to reduce discretization errors and reduce mesh element count (i.e., versus a full unstructured tetrahedral mesh) when there are strong gradients in fields normal to a geometric model surface. Localizing a stack of elements, or vertices along the growth curve, can reduce communications during a PDE solve that uses line relaxation pre-conditioning methods [94] for improved convergence, and during mesh adaptation coarsening procedures [87]–[89].

Algorithm 4 details steps taken to combine the stacks for a vertex-partitioned mesh with prismatic boundary layers. The algorithm loops over each vertex classified on a geometric model face to see if it bounds a prism on lines 1-2. From each of these vertices, the algorithm searches for the mesh edge that does not bound any triangles on lines 5-9. An edge that only bounds quad faces is guaranteed to be the edge going up the prism elements. If this edge is found, lines 10-14 find the other vertex that bounds this edge and repeats looking for a new edge from this vertex. This process is continued until no edge is found since once a tetrahedron or pyramid element is hit there will be no edges that are not adjacent to a triangle. Note, the algorithm is simplified to assume each stack exists on a single process. If this were not the case, topological information used to stitch parts together (i.e., which processes have a copy of a given mesh entity and the pointer to the entity on the remote process) would be queried and peer-to-peer communications required to complete the stack traversal.

Algorithm 4 Boundary Layer Stack Collapse

```
1: for all vertices, $v$, classified on a geometric model face do
2:     if $v$ bounds a prism then
3:         $prev\_edge = NULL$
4:         $next\_edge = NULL$
5:         for all edges, $e$, bounded by $v$ do
6:             if $e$ bounds no triangles and is not $prev\_edge$ then
7:                 $next\_edge = e$
8:             end if
9:         end for
10:        if $next\_edge$ is not $NULL$ then
11:            $edge\_prev = e$
12:            $v = other\_vertex(v, e)$
13:            goto 4:
14:        end if
15:    end if
16: end for
```

To maintain the correct computational and communication load of the stacks, we assign weights to the vertices and hyperedges that represent the stack. Each boundary layer stack vertex accumulates the weight of the mesh vertices in the stack, while the hyperedges accumulate the weight of the elements that share the same graph vertices. If the application does not supply per-vertex and per element weights than a unit weight is assigned.

3.3.2 Vertex-Based Unstructured Mesh Partition

Experiments for the vertex-partitioned mesh application were done with a 57 million mixed element mesh (i.e., tetrahedra, prisms, and pyramids) on up to 8192 processes. The mesh is comprised of 32 million tetrahedron, 25 million prisms and 150 thousand pyramids. As a vertex-partitioned mixed mesh, the graph vertex degree is non-uniform throughout the mesh, so the edge-cut mitigation strategy is important for this situation. To measure the effect of β, values from 0.5 to 1.2 are compared to the heuristic being disabled. Each test runs 30 iterations of EnGPar's edge balancer while strictly maintaining the vertex imbalance at 5%. Figure 3.4 shows the edge cut and edge imbalances for each test; including the initial values of the ParMETIS partitioned mesh. For the 8192 part case, with β disabled, EnGPar reduces the edge imbalance by 24 percentage points while increasing the cut by 34%. Setting $\beta = 1.2$ results in limiting the increase of the cut to 17% while reducing the imbalance by 18 percentage points. Results for $\beta = 1.0$ and $\beta = 0.8$ have similar effects following the trend that lower values of the limit result in lower edge cuts and higher edge imbalance. When $\beta = 0.5$, the edge cut increases by 1% while reducing the imbalance by 11 percentage points.

The best choice for β is specific to each application. The increase in edge cut means that there will be more elements on each part as well as increased communication between parts. If communication dominates a simulation's scaling then the increase in edge cut, and the associated increase in communications, may negate, or exceed, any savings from improved balance. So, the partition from ParMETIS may be the best choice, or running EnGPar with a low value for β like in the 0.5 case. However, if the application performance is very sensitive to imbalance, then a increase in the edge cut would be worth the larger decreases in imbalance as seen in the $\beta = 0.8$ and $\beta = 1.2$ cases.

For the boundary layer collapse, we use the same 57 million element mixed mesh. The N-graph is built in serial from the mesh with the collapsed boundary layers and partitioned out using global ParMETIS for the 1024 to 8192 partitions. Then, EnGPar is run for 30

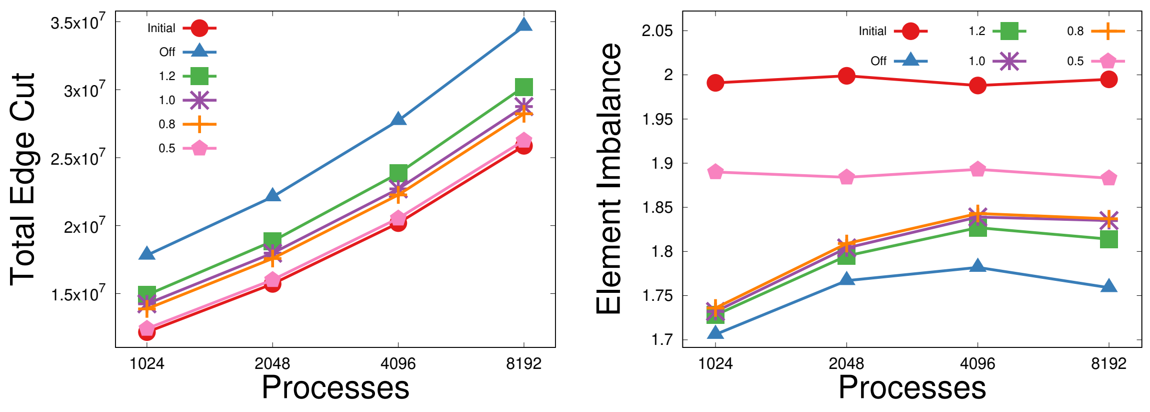

Figure 3.4: Edge cut and imbalance for various values of β used to reduce the growth of edge cut. Initial values from ParMETIS and β disabled are also provided. Vertex imbalance is 5% for all cases. ©2018 IEEE.

iterations to reduce the edge imbalance with the same set of values for β. Figure 3.5 shows the edge cut and edge imbalance after partitioning with ParMETIS and after EnGPar. With β disabled, there is a 41% point reduction in element imbalance with a 38% increase in edge cut. The usage of β has a much larger affect than in the uncollapsed boundary layer case. For a value of 1.2 there is a 35% point drop in imbalance with a 15% increase in edge cut. Similar trends are seen for $\beta = 1.0$ and $\beta = 0.8$. The $\beta = 0.5$ case reduces the imbalance by 11 percentage points with a less than 1% increase in the edge cut.

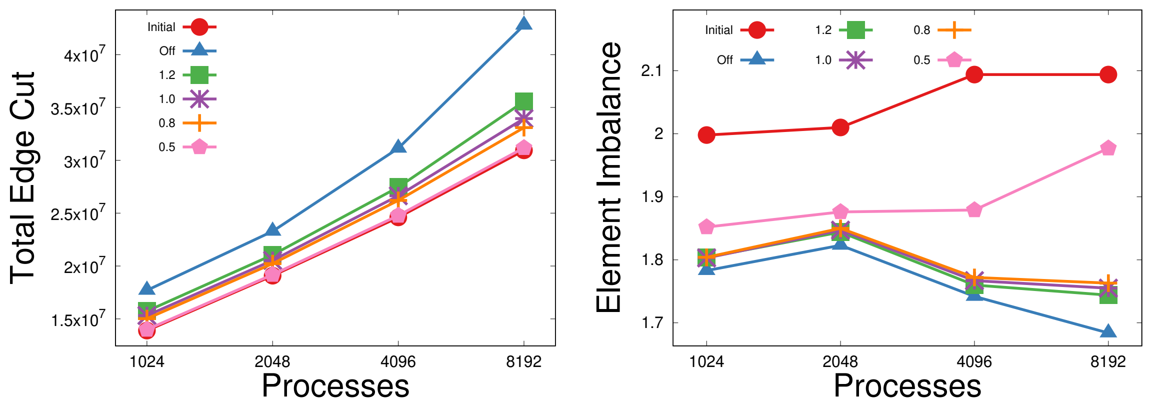

Figure 3.5: Edge cut and imbalance for the graph with collapsed boundary layer stacks using values of β from 0.5 to 1.2 and with β disabled. Initial values from ParMETIS are provided. Vertex imbalance is 5% for all cases. ©2018 IEEE.

3.4 Improving Higher-Order Finite Element Partitions

For higher-order finite element methods, different number of unknowns are defined on each dimension of mesh entity based on the order of the finite element and the type of finite element used. The computation costs for the simulation are directly related to the number of unknowns a process has to solve for, while the communication costs are related to the number of shared unknowns between processes. For these studies, we define degrees of freedom on mesh vertices, mesh edges, and mesh faces. Weights are assigned to each entity as one for mesh vertices, two for mesh edges, one for mesh triangles, and two for mesh quadrilaterals.

Experiments for the higher-order finite element case are performed on an element-partitioning of a 57 million mixed element mesh on up to 8192 processes. The original partitions of the mesh is created by ParMETIS globally to balance the mesh elements. The goal for EnGPar is to balance the number of unknowns and then to rebalance the mesh elements. For the two definitions of the N-graph for higher-order finite elements described in Section 3.1, the number of unknowns associated to a specific mesh entity is set as the weight of the hyperedge in the N-graph for that mesh entity. EnGPar is applied on each partition from 1024 to 8192 processes. Experiments are run on the Theta system at the Argonne Leadership Computing Facility.

For the N-graph with different hyperedge types for each dimension, the priority order is mesh vertices, mesh edges, mesh faces and finally mesh elements. This means that EnGPar will balance the hyperedges associated with the mesh vertices first, then balance the hyperedges for mesh edges while maintaining the imbalance of mesh vertices, then the mesh faces. Finally, the graph vertices that represent the mesh elements will be rebalanced while maintaining the imbalance of the three types of hyperedges. For the second approach, where the N-graph has one type of hyperedges for all degree of freedom holders, the priorities are first to balance the hyperedges followed by the graph vertices. In both cases a tolerance of 1.05 is the goal for all dimensions.

Figure 3.6 shows the imbalance of DOFs and mesh elements before and after applying EnGPar for both the hyperedge types for each lower dimension of mesh entities, labeled “Multiple”, and when one hyperedge type for all DOF holders, labeled “Single”. For the multiple hyperedge type case, EnGPar is able to reduce the imbalance of DOFs down from 1.3-1.55 in the initial partition to between 1.1-1.25. Balancing is unable to improve further due to a combination of stagnation, and balancing each mesh dimension individually may not result in equivalently balancing the number of degrees of freedom. For the single hyperedge type for all DOFs, the imbalance is reduced below the requested tolerance of 1.05 for 1024-4096 parts and down to 1.052 for the 8192 partition. In this case, EnGPar is able to properly balance all lower mesh dimensions simultaneously and achieve the desired imbalance.

For improving the imbalance of the mesh elements after finishing the degrees of freedom, EnGPar reduces the imbalances 1.20-1.24 imbalances down to 1.15-1.18 in both the multiple and single cases. The element imbalances are roughly .01 better for all partitions using the N-graph with a single hyperedge type.

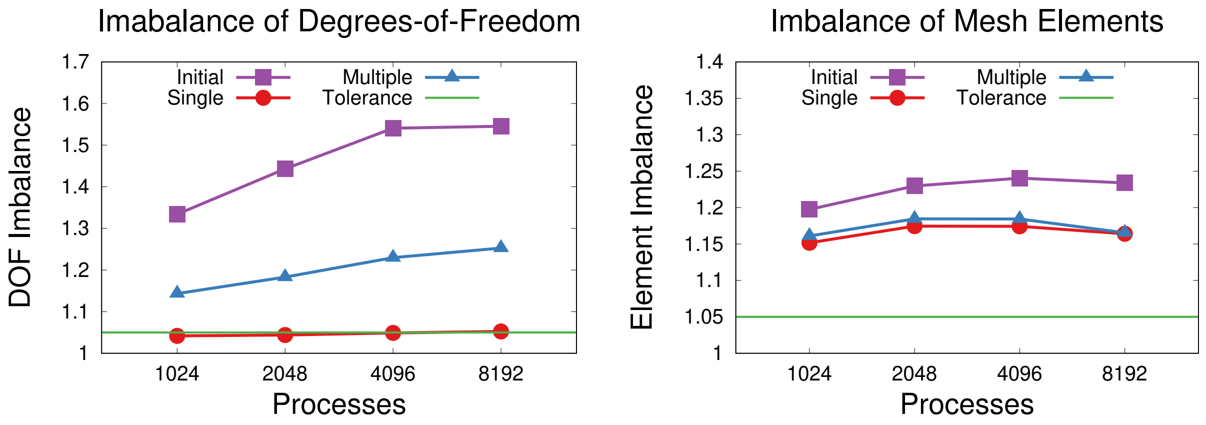

Figure 3.6: Imbalances of degrees of freedom (left) and mesh elements (right) before using EnGPar and after running EnGPar on the N-graphs for one hyperedge type per lower mesh dimension (Multiple) and one hyperedge type for all DOF holders (Single). Lower is better.

While the single case results in better quality partitions, the time cost is significantly greater than for the multiple hyperedge types case. Each single hyperedge case takes about four to five times longer than the same node multiple hyperedge case. Part of this increase in time cost is because the single case is performing more iterations than the multiple case as the single case does not stagnate and instead reaches the target imbalance of DOFs. The bigger contribution to the increase in time to partition is the size of the hyperedges that are operated on in each iteration. For the single case, each iteration operates on $|M^0| + |M^1| + |M^2|$ hyperedges while the multiple case performs three different sets of iterations each with $|M^0|$, $|M^1|$, and $|M^2|$ hyperedges respectively. Thus there is a tradeoff between using the single case for the best partition of DOFs or a faster method using the multiple case with a higher imbalance. The correct choice for this depends on how much the extra improvement to imbalance decreases the time within the simulation.

3.5 Summary

EnGPar's fast diffusive load balancing method has been applied to improve the partitions of parallel unstructured meshes with over a billion elements and efficiently scales to over half a million processes. The graph distance algorithm for ordering candidate boundary entities for migration has been sped up compounding with a faster migration operation resulting in EnGPar being 54% faster than ParMA at 512Ki processes. EnGPar is applied to other unstructure mesh cases that can be more accurately represented with the N-graph than in ParMA, namely vertex-partitioned meshes and higher-order finite element simulations. Control mechanisms for mitigating the increase in edge cut for high-degree vertex graphs was incorporated into the methods to achieve better partitions for vertex-partitioned meshes.

CHAPTER 4
FRAMEWORK FOR DESIGNING MESH-BASED UNSTRUCTURED-MESH PARTICLE-IN-CELL SIMULATIONS ON GPUS

An important class of multiscale simulation is the Particle-in-Cell (PIC) method in which particle tracking to capture fine-scale behaviors is coupled to fields defined in terms of PDEs represented at the scale of the overall domain. The continued advancement of massively parallel computing technologies along with the ability to effectively scale PIC calculations on those systems is supporting the effective application of PIC codes to the modeling of plasmas in fusion reactors [21], [42], [43], [96], [97], linear accelerators [98] and other systems.

PIC methods are implemented as a time-advancing procedure in which the position of particles is tracked as they move through a domain, driven by a field that is typically a function of the position of the particles, and thus that field evolves as the particles move. In the coupled case there are four steps carried out in each time advance [21], [43], [96], [99]–[102]. Those steps are:

Field to Particle: The values of the current mesh-based fields that drive the particles are associated with each particle through an appropriate interpolation procedure.

Particle Push: The particles are moved, or if you will, pushed, to a new location as a function of the field and time step size.

Charge Deposition: The “charge” information associated with the particles is then related to the domain definition such that the forcing function driving the field evolution is updated.

Field Solve: The partial differential equations (PDEs) governing the field is then solved using this updated forcing function and other potential system updates.

Portions of this chapter are to appear in: G. Diamond, C. W. Smith, C. Zhang, E. Yoon, and M. S. Shephard, “PUMIPic: A mesh-based approach to unstructured mesh particle-in-cell on GPUs,” *J. of Parallel and Distrib. Comput.*, vol. 157, pp. 1–12, Nov. 2021. DOI: `10.1016/j.jpdc.2021.06.004`

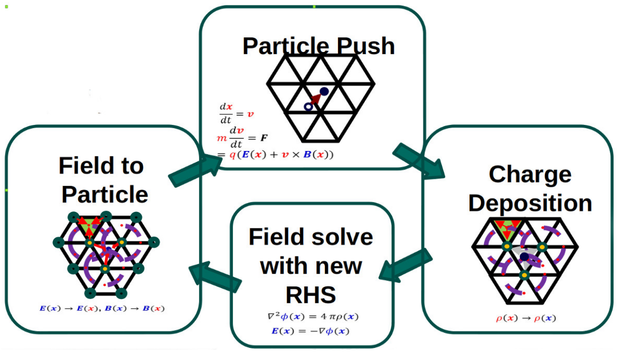
Particle Push
$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{v}$
$m\frac{d\boldsymbol{v}}{dt} = \boldsymbol{F} = q(\boldsymbol{E}(\boldsymbol{x}) + \boldsymbol{v} \times \boldsymbol{B}(\boldsymbol{x}))$
Charge Deposition
$\rho(\boldsymbol{x}) \rightarrow \rho(\boldsymbol{x})$
Field solve with new RHS
$\nabla^2\phi(\boldsymbol{x}) = 4\pi\rho(\boldsymbol{x})$
$\boldsymbol{E}(\boldsymbol{x}) = -\nabla\phi(\boldsymbol{x})$
Field to Particle
$\boldsymbol{E}(\boldsymbol{x}) \rightarrow \boldsymbol{E}(\boldsymbol{x}), \boldsymbol{B}(\boldsymbol{x}) \rightarrow \boldsymbol{B}(\boldsymbol{x})$

Figure 4.1: Depiction of the 4 main steps involved in an iteration of the PIC loop.

Since the PDEs governing the domain fields can rarely be solved in closed form over the domains of interest, the needed fields are solved numerically over a spatial discretization that can range from a uniform grid to a graded unstructured mesh. Particles are represented as distinct objects with properties that are tracked as they move throughout the domain of the mesh. Figure 4.1 gives a basic graphical description of the four PIC steps on a mesh. Structured grids are commonly used in PIC codes to represent the domain due to the simplicity of storing and maintaining the mesh representation as well as interactions between the mesh and particles [103]–[107]. For simulations with complex geometries or varying scales of behavior, uniform structured meshes are ill-suited as a highly refined mesh is required to achieve the desired accuracy. Adaptive mesh refinement is one method to reduce the issue by refining the mesh where more accuracy is needed while coarsening the less important regions [108], [109]. Alternatively, unstructured meshes can be used to more accurately represent the domain and can provide the required levels of field accuracy over general domains using the fewest number of unknowns [110]–[113]. The advantages of unstructured meshes come with a cost of larger, more complex, data structures and more complex algorithms to achieve parallel scalability.

A key piece of information that needs to be known for each particle throughout the simulation, is the grid cell or mesh element, each particle is within, that we call the particle's parent element. This information is required by the Field to Particle and Charge Deposition steps. In the case where the PIC method is tracking particles through a uniform grid, this operation is trivially defined by the particle coordinates.

When unstructured meshes are used, the determination of the parent element after a push is not a one-step algebraic evaluation. Instead, it requires a numerical evaluation process that includes explicit consideration of the geometric definition of the elements. To avoid the need to evaluate the required geometric calculations involved with determining particle containment for the entire list of elements until the element the particle is within is found, some form of search mechanism that limits the number of elements that must be considered is employed. The complexity of maintaining the knowledge of each particle's parent element introduces a coupling between the mesh data structure and the storage of particles.

In large-scale computations executed on massively parallel computers, the PIC algorithm introduces partitioning challenges due to the coupling between the mesh data structure and the particles. A common approach for parallel PIC is to partition the particle data structure where each particle maintains knowledge of the mesh element the particle is in at the current time while storing an independent mesh data structure that is copied on every process. When using independent particle and mesh data structures it is common to also introduce a spatial data structure such as a uniform grid or spatial tree to support an efficient search process to find the element the particle is contained in after a push. This approach can achieve good scaling and is reasonably performant, particularly for meshes with little or no gradation. However, having a copy of the entire mesh on each process has the obvious drawback of not being scalable with growing the mesh size past a point dictated by the available memory on a process for the mesh. Since PIC implementations employ methods in which the number of particles is two or even three orders of magnitude larger than the number of mesh elements, this approach has been satisfactory. However, as more complex, both in terms of physics and geometric complexity, systems are considered, there is a desire to employ distributed meshes with millions of elements that are strongly graded. The use of independent particle and mesh data structures which are both distributed complicates the implementation of efficient memory access patterns and, in the case of graded meshes, complicates the search process of efficiently determining what element a moved particle is within.

The central goal of the approach presented in this chapter is to be able to support having both the particles and mesh distributed over the memory spaces of the processes used in the parallel execution of the PIC simulation. Any effective implementation of such an approach must carefully consider appropriate mechanisms to control interprocess communications that arise through the interactions of the particles and the mesh. A possibility for doing this in the case of independent particle and mesh data structures is to employ mechanisms that distribute closely related mesh elements and particles. For example, the same background grid used to support the element search process could be used. This chapter presents an alternative approach that we will refer to as the mesh-based approach in which the mesh data structure is the core data structure and the particles are stored based on their parent elements for access based on the mesh elements. Since this approach maintains an explicit relationship of the particles to the mesh, there is no need for a secondary structure and search process to support maintaining the relationship between them.

With the majority of top supercomputers including some form of accelerators, in most current US systems being GPUs, it is vital to employ data structures and algorithms that effectively take advantage of the accelerators. The PIC algorithm have been shown to have good performance improvements when porting from CPU to GPU [114], [115]. Parallel PIC codes have been employed for structured meshes such as PIConGPU that utilize linked lists for storing particles [38]. The EMPIRE-PIC [116] code performs parallel PIC on an unstructured mesh using the Trilinos suite[117] including Kokkos [35] for performance on accelerators. The Cabana library [118] has shown good performance using an Array-of-Structs-of-Arrays for storing particles on GPUs for various PIC codes [119]. The porting of plasma-physics PIC codes such as XGC [42], [120], GITR [44] show there is great potential for improved performance using GPUs for PIC applications.

This chapter presents the PUMIPic library that takes the aformentioned mesh-based approach for PIC simulations to provide the necessary data structures and key algorithms for implementing PIC codes on GPUs. The library includes a partitioned unstructured mesh data structure that uses extensive buffering to improve the control of inter-process communications and optimization of particle migrations between processes. Different data structures for the storage of particles in PUMIPic are presented that all maintain the storage of particles based on the mesh element. Key algorithms for performing the four main stages of the PIC algorithm are detailed. Notably, this chapter includes the algorithms for supporting particle and mesh interactions, synchronizing fields across the partitioned mesh, determining the new parent elements after particles move, and updating the particle data structure based on new parent elements.

As the simulation evolves and particles continously move between processes, the balance of work associated with particles may become uneven across the GPUs. To maintain the balance of particles, a method for dynamic load balancing is presented using the EnGPar library that exploits the specifics of PUMIPic's mesh partition to reassign particles efficiently.

The remaining sections of this chapter are as follows. The core data structures for storage and operation on GPUs for the unstructured mesh and particles are detailed in Section 4.1. Section 4.2 details the core operations in PUMIPic to support the development of PIC simulations using the library. Load balancing for particles on a fixed mesh using EnGPar is discussed in Section 4.3. Results for using the PUMIPic library in PIC simulations is provided in Chapter 5.

4.1 Data Structures for Mesh-Based PIC

For designing unstructured mesh-based PIC simulations, two core data structures are needed: a mesh data structure and a particle data structure. An unstructured mesh data structure on GPUs has to account for the complexity of storing and efficient access to the necessary mesh adjacencies and field information. The particle data structure must focus on efficient memory access of particle information and the interactions with the mesh entities and fields.

4.1.1 Performance Portability

With the three upcoming DOE leadership-class supercomputers, Aurora, Frontier, and Perlmutter, each using a different vendor GPU with specific languages and hardware functionality, it is highly desirable to implement codes for GPUs to be portable across different hardware. Writing GPU specific code in each low-level language for each new hardware device would be ineffient and difficult to manage as a code base grows. To avoid managing different versions of code there exists several programming models that supply abstractions for operating on various architectures. Compiler directive based approaches such as OpenMP [121] and OpenACC [122] use pragmas to annotate the code where parallel operations are to be carried out. Library approaches like Kokkos [35], Raja [36] and OCCA [123] provide an abstraction layer that can be used to write one implementation of the code and use different backends for different targeted hardware.

To address the ability to port to different hardware, all data structures in PUMIPic use the Kokkos library [35] for data management and parallel execution on GPUs. The structures in PUMIPic take a device-default approach where data is initialized on the GPU and remains in GPU memory unless the host needs to perform operations on the data such as performing communications between processes. The remainder of this section covers the design choices for storage of the unstructured mesh and particles for on process parallel execution and multi-process parallelism.

4.1.2 Unstructured Mesh Structure

For unstructured mesh PIC, the unstructured mesh data structure is the key structure that discretizes the domain for storing field information and performing the field solve of the governing equations. An unstructured mesh data structure is composed of a geometric model, mesh entities, mesh adjacencies, and field information attributed to the mesh entities [78]. The geometric model is the high level representation of the domain with information tied to the model entities such as material properties, boundary conditions, etc.. The mesh entities include vertices M^0, edges M^1, faces M^2, and regions M^3 in 3D. Mesh entities are classified on geometric model entities to define the entities on the boundary and specific physical quantities related to different portions of the mesh. Mesh adjacencies describe the connection between mesh entities. Most notably are the first order adjacencies $\{M_k^{d_i}\{M^{d_j}\}\}$ that is the set of mesh entities of dimension d_j that are adjacent to the kth mesh entity of dimension d_i. The field information is attached to mesh entities in order to store physical quantities of interest in the simulation.

There are different choices for the storage of mesh entities and their adjacencies with trade-offs between memory consumption and access time to query adjacencies [78]. A full mesh representation explicitly stores all mesh entities [124]. This is useful in the case where fields are defined on all dimensions of mesh entities or when explicit iteration of each dimension is needed such as for mesh adaptation. When some dimensions of mesh entities are not stored it is referred to as a reduced representation. Storing more mesh adjacencies has the benefit of improved runtime at the cost of storing more memory especially for upward adjacencies, $\{M_k^{d_i}\{M^{d_j}\}\}$ where $d_j > d_i$. For example, the average number of mesh regions adjacent to a mesh vertex in a tetrahedral mesh is 23. A complete mesh representation stores a sufficient set of the mesh adjacencies that allow $O(1)$ lookup for any individual mesh adjacency [124]. For individual mesh adjacencies one can build missing adjacencies by using set intersections for downward adjacencies and set unions for upward adjacencies [124].

A distributed unstructured mesh has additional complications for storage around the partition model boundary. Given a unique partition of the mesh elements into N parts, $P(M^d) = P_0, ..., P_{N-1}$ where $P_i \cap P_j = \emptyset$, the lower dimensional entities on the boundary of each part will be shared with the neighboring parts. These entities classified on the partition model boundary will exist on each part that share them and a method must be used to be able to communicate between the copies such as using remote copies [124], [125] where each shared entity maintains a list of the copies on each other part that share it.

PUMIPic's mesh data structure is built on top of the Omega_h library [46], [47]. Omega_h is a performance portable unstructured mesh library designed for GPUs supporting two and three dimensional meshes. The structure takes a device default approach where all mesh entities, mesh adjacencies and field data are allocated and remains on the device unless explicitly transferred to the host. Omega_h stores the one-level downward adjacencies,$\{M_k^{d_i}\{M^{d_j}\}\}$ where $d_j = d_i - 1$, as compact arrays. Other mesh adjacencies are constructed and maintained when the application requests them either as compact arrays for downward adjacencies or as compressed sparse row formats for upward adjacencies. At a start of an application the Omega_h mesh is input. This operation is of $O(n)$ where n is the number of mesh entities. Omega_h includes procedures that can be executed during the input process to construct any of the additional adjacencies that the application would like stored. The execution of these procedures maintains the $O(n)$ input cost. In our applications of Omega_h we define a sufficient set of adjacencies to yield a complete representation where any adjacency can be obtained in $O(1)$ time. This is done since the applications developed in this thesis employ many of the 12 possible first order adjacencies. If an application only needs very specific additional adjacencies past those already stored, then defining only those additional adjacencies is fine even if the resulting set is not a complete representation since there will be no request for an adjacency that requires $O(n)$ operations.

Omega_h is used to store and operate on the mesh elements and fields on a single process and GPU. In this regard, each process has its own instance of an Omega_h mesh that does not know of the mesh on other processes or the shared entities between the processes. As discussed in the next section, an expanded partitioning of the mesh is desireable that is not supported using Omega_h's distributed mesh capabilities. As such the distribution of mesh entities and fields between processes are handled through PUMIPic to achieve the level of parallel operations desired for a PIC simulation.

4.1.3 Mesh Partitioning for PIC

Partitioning an unstructured mesh is both a challenging and important problem for distributed mesh applications. Recall from Chapter 2 that the partition must balance the computational costs across all processes while also minimizing the communication costs that result from partitioning the mesh. The most common methods for partitioning meshes are multi-level graph methods [7], [9], geometric methods [63], [65], [67], and diffusive methods [11], [13], [22]. Details of these methods were covered in Section 2.1.

For PIC simulations, the difficulties of partitioning increase due to having to balance mesh computation and communication along with particle computation and migrations across processes. Typically, the mesh is partitioned to optimize either the field solve step or the interactions between the mesh and particles. Partitioning for the mesh field solve considers evenly distributing the number of degrees of freedom across processes and minimizing the number of degrees of freedom on the partition model boundary. Partitioning for particles targets allowing particles to be distributed evenly across processes and minimizing the migration of particles as the simulation evolves. Constructing the partition along the principal direction of motion of particles is ideal to maintain particle balance and minimize communications. Figure 4.2 shows two partitions of a two-dimensional mesh. The left mesh is partitioned using a multilevel graph method for the field solve. The right mesh is partitioned along flux faces where particles predominantly move within.

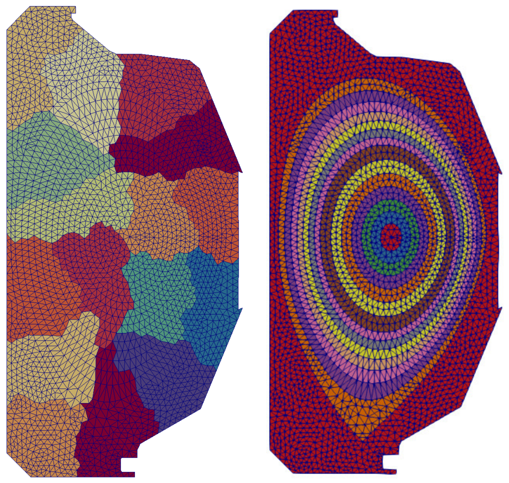

Figure 4.2: Two-dimensional unstructured mesh partitioned using multi-level graph method for field solve (left) and partitioned along principal direction of particle motion (right).

Regardless of the approach to partitioning, in PUMIPic, the partition of the mesh is expanded with surrounding mesh entities to create what we call PICparts in order to minimize the communication of particle information by ensuring that no communication is required during the execution of a single push step. Given a unique partition, $P(M^d)$, of the d-dimension mesh, M, into N parts $P_0, P_1, ...P_{N-1}$, we define a buffer, B_i, for each part that consists of layers of mesh elements surrounding the part P_i. The ith PICpart is the closure of the mesh elements in the set $\overline{P_i} = P_i \cup B_i$. We denote a subset of $\overline{P_i}$ as the safe zone, S_i which represents the set of mesh elements that can have particles on the PICpart. S_i is defined such that any particles in these mesh elements are guarenteed to have all field information required to perform operations and the particle will not be pushed outside the PICpart during the push operation. Note that particles that at the end of a push have moved into an element not in S_i will be migrated to a PICpart for which that element is within its safe zone before the next push operation. To ensure a particle can be safe in any mesh element, we require that for every element $M_j^d \in P_i$, $M_j^d \in S_i$.

The first step to creating PICparts is to define B_i for each part P_i. There are multiple options for selecting the set of buffer elements to be used for a PICpart. An obvious choice is to select a number of neighboring elements in the same way remote copies are defined in many PDE discretization methods [125]. For the PIC applications developed to date, it was found that this would require four layers of elements and it was determined that the data storage and update requirements if done as standard remote copies were problematic.

The alternative approach used herein defines B_i as the union of a set of the nearby parts P_j for $j \neq i$ to ensure that there is sufficient buffer with respect to P_i. Although this approach yields PICparts with more elements than a layerwise buffering would produce, it requires substantially less total data since it does not require maintaining remote copy data associated with each element. This approach also yields large safe zones that reduce particle migration and provides greater load balancing options.

In cases where the memory requirement to store the entire mesh and required fields on each PICpart is low, it is convenient for each PICpart to store the entire mesh. In this case, $B_i = \bigcup_{j \neq i} P_j$.

When the amount of memory required to store the entire mesh and required fields is such that it is desirable to distribute the mesh, the B_i is a strict subset of the the initial set of parts defined by executing a breadth-first traversal (BFT) out some number of layers from the boundary of the P_i. B_i is constructed as the union of every part reached during the BFT. An example PICpart constructed using five layers of BFT buffering is shown in Figure 4.3. The PICpart is for the part labeled A which then fully buffers each part within five iterations of BFT including the non-adjacent part labeled B since elements in part B are included in the third layer determined by the BFT.

After the buffer is constructured, the safe zone, S_i is determined using one of two approaches. The first is an overly conservative method that uses fewer layers of the BFT performed for determining the buffer. Any element found within the BFT is in S_i as well as every element in P_i. The second method performs a BFT for a set of iterations starting from the PICpart boundary defined as the lower dimension mesh entities on the boundary between the PICpart and parts that are neither P_i nor in B_i. In the second method, every element not reached in the BFT is in S_i. The second approach creates a much larger set of safe elements leading to fewer migrations and higher potential for particle load balancing to occur. Figure 4.4 shows example usage of both safe zone methods for the PICpart in Figure 4.3. The simulation can also provide a different definition for S_i if a more physics-dependent definition of safe elements is required.

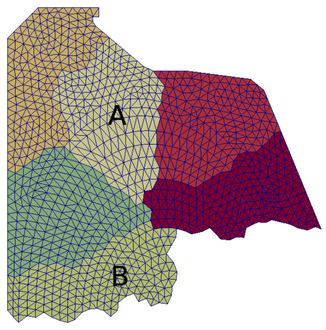

Figure 4.3: PICpart generated for the part A using 5 layers of breadth-first traversal for the left partition from Figure 4.2. Note that the part labeled B is also fully buffered.

The BFT algorithm is implemented using a bottom-up approach [126]. In the bottom-up approach, each iteration of the BFT processes has every mesh element check its neighbors to see if it is visited this iteration. This is performed through a selected lower dimension of entities called the bridge dimension. Each entity of the bridge dimension first checks its upward adjacencies to elements if any have been visited in a previous iteration. If at least

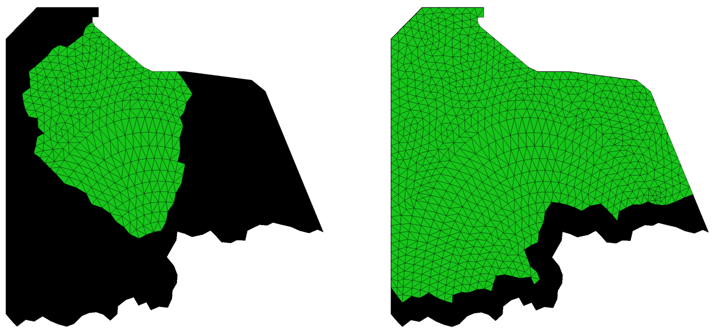

Figure 4.4: Safe zone for PICpart A using three iterations of BFT from the part (left) and excluding three layers from the edge of the PICpart (right).

one element has been visited then all of the upward adjacent elements are visited in the current iteration of the BFT.

The bottom-up approach is better than the more traditional top-down approach for many-core and GPU architectures when the frontier of the BFT is sufficiently large. Since the starting point for the BFT is the entire part, the iterations start with a sufficiently large frontier such that using the bottom-up method is more performant than the top-down.

Algorithm 5 provides pseudocode for constructing PICparts using bottom-up BFT iterations. Lines 2-6 setup the initial state for the BFT. The visited array is filled with true values for each element in the part, P_i. The BFT algorithm is iteratively run on lines 7-13. Each iteration executes the BFTIteration procedure on line 8 to perform one layer of the bottom-up BFT on lines 22-36. The bottom-up BFT iterates over all mesh entities $M^b_j \in M^{bridge\_dim}$ to operate on the mesh elements in $\{M^b_j\{M^d\}\}$. The first loop over the adjacent mesh elements on lines 25-29 checks if any of the mesh elements have been visited in a previous iteration. If at least one element was previously visited then $\{M^b_j\{M^d\}\}$ is iterated over a second time setting all of the elements as visited in this iteration by setting *visited_next* to be true on lines 30-34. After each call to BFTIteration, the *visited_next* array is copied to the *visited* array to set up the next iteration on line 9. When using the conservative method to construct the safe zone, lines 10-12 constructs the safe zone from the visited elements after the specified number of iterations for the safe zone is performed. After the BFT is completed, the mesh parts that include any visited elements are designated to make up the buffer on lines 14-19.

Algorithm 5 pseudocode for pull-based BFT to construct buffer region and safe zone.

```
1: function BFT($M$, $safe\_layers$, $buffer\_layers$, $bridge\_dim$)
2:     $visited$ = DeviceArray($|M^d|$)
3:     $visited\_next$ = DeviceArray($|M^d|$)
4:     parallel_for Element $M_i^d$ in $P_i$ do
5:         $visited[M_i^d]$ = true
6:     end parallel_for
7:     for $iter = 0$ to $buffer\_layers$ do
8:         BFTIteration($M$, $visited$, $visited\_next$)
9:         $visited \leftarrow visited\_next$
10:        if $iter = safe\_layers$ then
11:            $safe\_zone \leftarrow visited\_next$
12:        end if
13:    end for
14:    $buffer\_parts$ = DeviceArray(number of processes)
15:    parallel_for Element $M_i^d \in M$ do
16:        if $visited[M_i^d]$ then
17:            $buffer\_parts[owner(M_i^d)] = true$
18:        end if
19:    end parallel_for
20:    return $buffer\_parts, safe\_zone$
21: end function
22: function BFTIteration($M$, $visited$, $visited\_next$)
23:    parallel_for Entity $M_i^b \in M^{bridge\_dim}$ do
24:        $visit = false$
25:        for all Element $M_j^d \in \{M_i^b\{M^d\}\}$ do
26:            if $visited[M_j^d]$ then
27:                $visit = true$
28:            end if
29:        end for
30:        if $visit$ then
31:            for all Element $M_j^d \in \{M_i^b\{M^d\}\}$ do
32:                $visited\_next[M_j^d] = true$
33:            end for
34:        end if
35:    end parallel_for
36: end function
```

When constructing the safe zone with the second method, a separate BFT is performed after Algorithm 5. Algorithm 6 lists pseudocode for this construction of the safe zone. The ConstructSafe procedure on lines 1-21 takes in the PICpart, M, and a number of *layers* for the BFT. Lines 2-10 perform setup for the BFT by creating the *visited* and *visited_next* arrays. For every mesh entitiy $M_i^b \in M^{bridge\_dimension}$ on the PICpart boundary, the mesh elements $M_j^d \in \{M_i^b\{M^d\}\}$ are set to be visited during this setup. The bottom-up BFT is then executed for *layers* iterations on lines 11-14. This loop calls the BFTIteration procedure from Algorithm 5 on line 12 followed by copying the *visited_next* array into *visited* to prepare for the following iteration. After the BFT is completed, the safe zone is constructed on lines 15-19 by adding every element in the PICpart that was not visited in the BFT to the safe zone.

Algorithm 6 Algorithm for constructing the safe zone from the boundary of the PICpart

```
function ConstructSafe($M$, $layers$)
    $visited$ = DeviceArray($|M^d|$)
    $visited\_next$ = DeviceArray($|M^d|$)
    parallel_for Entity $M_i^b \in M^{bridge\_dimension}$ do
        if $M_i^b$ on PICpart boundary then
            for all Element $M_j^d \in \{M_i^b\{M^d\}\}$ do
                $visited[M_j^d]$ = true
            end for
        end if
    end parallel_for
    for $iter = 0$ to $layers$ do
        BFTIteration($M$, $visited$, $visited\_next$)
        $visited \leftarrow visited\_next$
    end for
    parallel_for Element $M_i^d \in M$ do
        if $!visited[M_i^d]$ then
            $safe\_zone \leftarrow M_i^d$
        end if
    end parallel_for
    return $safe\_zone$
end function
```

4.1.4 Particle Data Structure

Particle information such as position and velocity must be stored on the GPU in a structure that allows efficient access. For mesh-based PIC simulations, the main requirement for the particle data structure is to group particles by the parent mesh element. For performance on accelerators, it is important that the structure can be evenly distributed to threads and mapped to the hardware memory layout and the access pattern. As a library, it is also important that the particle data structure is tunable to support different application characteristics and be performant across different hardware. The characteristics present in the target applications of PUMIPic include: up to 10,000 particles per element, uniform and non-uniform particle distributions, particles will only traverse through a small number of neighboring elements during a single iteration, and simulation-defined representations of particle data including multiple definitions of particles for a single simulation.

To begin thinking about data structures to store particles based on mesh elements it is important to first note that this relationship of particles in elements can be described as an NxM matrix where N is the number of elements and M is the number of particles. The values of entry i, j is 1 if particle j is in element i and 0 otherwise. This matrix is extremely sparse as each parrticle will only have one nonzero entry. Many different data structures have been proposed to maximize the locality and alignment of memory accesses on GPUs for matrix applications. Several alternatives to the commonly used Compressed Sparse Row structure have been analyzed [127]. The ELLPACK [128] structure adds zero entries to ensure all rows have an equal length to align memory accesses. The structure also stores memory vertically which allows threads of a warp to work on a row while accessing a contiguous block of memory. The ELLPACK structure is best used on structured matrices due to the extra memory and computation added by the non-zero entries. For matrices without a uniform structure, a variant ELLPACK-R is suggested [129]. ELLPACK-R groups the rows into chunks based on the SIMD width of the target hardware. The zero entries added to ELLPACK-R are such that all rows within a chunk have the same length. This improves from the original ELLPACK by reducing the memory footprint of the zero entries as well as reducing computations on zero entries. Further improvements to reduce the padding are suggested by Kreutzer et al. [130], [131] with the Sell-C-σ structure. The Sell-C-σ sorts the rows by the number of filled entries which when performing full sorting minimizes the number of zero entries added to the structure.

Directly for PIC, different implementations for GPUs have explored structures for storing particles. Burau et al. [38] use a linked particle list to implement a PIC simulation with structured meshes on GPUs. Each mesh element points to the list of particles within the element. This structure allows particles to move quickly between the mesh elements, however as Burau et al. explain when particles move between elements the order of memory accesses becomes fragmented and leads to a reduction in the performance of particle operations. A three-stage memory hierarchy [37] has been used to alleviate the fragmentation problem by tiling particle data within a set of mesh elements.

Cabana [118] is a library for the storage of particle data for the traditional approach to PIC simulations. Cabana uses an array of structures of arrays (AoSoA) to store particle information aligned with and sized by the target hardware. Each struct of arrays (SoA) maintains particle information for the SIMD width number of particles. Then an array of the SoAs is created to store all particles. This results in the memory accesses being aligned with the hardware and execution of the GPU.

4.1.4.1 Storing Particle Data

To support simulation-defined particles including multiple different particles during a single simulation, the structures in PUMIPic must be designed generally for arbitrary data. The use of variadic templates in C++ allows the simulation to define all the data types that make up a particle. For example, if a particle is defined by its three-dimensional coordinates, velocity, and an id, the data types would be defined with six doubles and one integer type. PUMIPic provides abstractions and helper functions to create collections of particle types and operate on them as needed to work with PUMIPic's API. The particle structure is templated on the collection of data types representing the particle. Thus different types of particles can be maintained for one simulation in separate particle structures that are templated on different sets of data types. The following sections review the different particle structures implemented in PUMIPic. For each structure when referring to particles or the values associated with them this refers to all of the data types that define a particle.

4.1.4.2 Compressed Sparse Row

One of the most commonly used structures in high-performance computing applications to store a sparse matrix is the Compressed Sparse Row or CSR. The CSR consists of two one-dimensional arrays, the first an $offsets$ array length $N+1$ and the second a $values$ array sized the number of nonzeroes in the matrix. In the case of our particle data structure, the $values$ array is length M. The $values$ array stores each nonzero of the original matrix grouped by the nonzeroes in the same row. The entries of the offset array index into the $values$ array such that $offsets[i]$ is the first index for row i in the $values$ array. Additionally, $offsets[i+1]-1$ is the last index for row i in the $values$ array. This structure allows the NxM particle matrix to be reduced in memory to $N+M$. Accounting for D particle data types the memory usage is $N + D * M$ as only one offsets array is needed for each of the D *values* arrays. Figure 4.5 shows an example CSR structure with *offsets* on the left and *values* on the right. The arrows on the *values* array show the order of continuous memory entries.

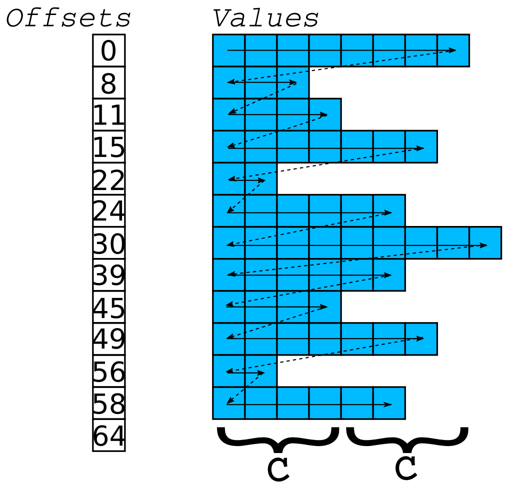

Figure 4.5: Example CSR offsets and values arrays. Each row represents one mesh element. Each box in the values array is one particle.

The CSR is a simply designed structure that is both easy to implement and use which can capture the mesh to particle relationship with minimal memory usage. However, this structure has problems when performing on accelerators. One important characteristic of accelerators is the Single-Instruction-Multiple-Data (SIMD) width that is the amount of data that can be processed by a thread for one instruction simultaneously. Ideally, the data should be split into $SIMD\_Width$ continuous array entries to perform operations simultaneously. For the CSR the work is first broken into each row which has the number of nonzero entries in that row. In general, the length of the row cannot be broken into $SIMD\_Width$ chunks and thus there will be wasted resources when operating on the accelerator. Figure 4.5 has an example $SIMD\_Width$ labeled C at the bottom of the *values* array showing that each row is unable to be split evenly. Another concern with the CSR is when the distribution of nonzeroes is non-uniform across the rows. In this case, each thread assigned to a row will have adifferent amount of work which will result in inefficient execution on accelerators.

4.1.4.3 Sell-C-Sigma

A structure that is more designed for accelerators based on the CSR is a structure from Kreutzer et al. called the Sell-C-σ (SCS) [131]. The structure was designed for efficient use of sparse matrices for tunable performance on different architectures including many-core and GPUs. The SCS targets solving the CSR's problem by aligning the data to GPU memory and access patterns. The SCS groups rows into chunks of size C and orders the memory vertically through chunks and horizontally across rows. The parameter C is set based on the SIMD width of the hardware being run on allowing the structure to map to the memory layout and align memory accesses with the hardware. Additional padding of empty cells is used to fill in the rows such that each row in a chunk has the same length and perfectly aligns with the hardware memory layout. A second parameter σ controls sorting the rows where sigma ranges from 1, no sorting, to the number of rows, full sorting. For non-uniform distributions, a higher σ will group longer rows which reduces the amount of padding and excess computations performed at the cost of running a sorting routine.

Figure 4.6 shows the storage of particles on a set of mesh elements in a CSR and two different SCS structures with $C = 4$ where the first uses no sorting ($\sigma = 1$) and the second has full sorting ($\sigma = 12$). Each row represents one mesh element with an entry in the row per particle within the mesh element. Arrows on the CSR and SCS structures show the continuous layout in memory. Empty cells in either SCS structures are padded cells with no particle data stored.

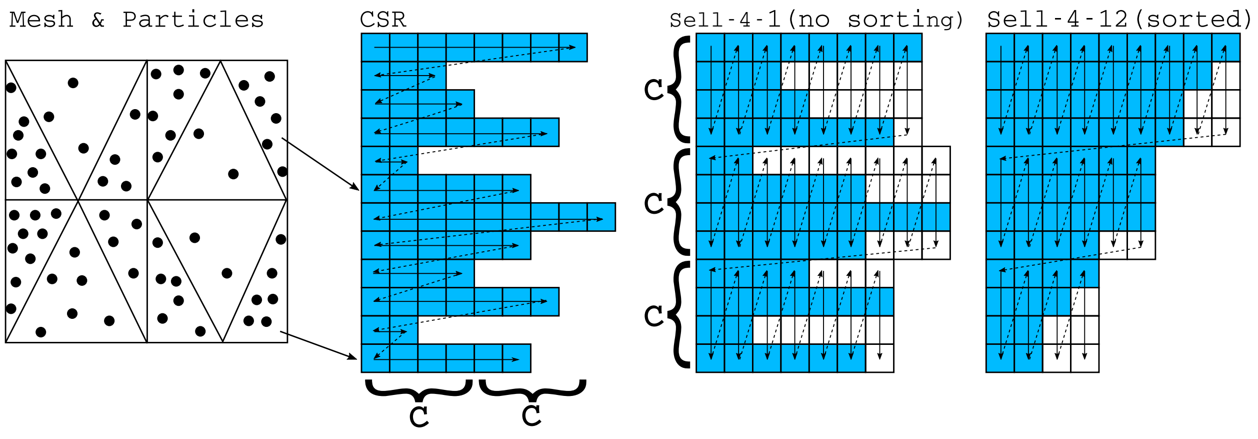

Figure 4.6: The storage of particles in a set of mesh elements (left) in a CSR (middle) and two SCS (right) with no sorting and with sorting. Arrows on each structure show the continous layout of memory.

Further improvements to the SCS structure are suggested by Besta et al. in their structure SlimSell [132]. As mentioned for the CSR in the previous section, when the distribution of particles to elements is non-uniform there will be workload imbalances between large chunks and small chunks. Besta et al. add vertical slicing to the chunks so that approximately equal slices are distributed to blocks of threads as opposed to the uneven chunks. An additional parameter V controls the horizontal length of each slice such that each chunk is split into slices of length V. Figure 4.7 shows the SCS with vertical slices boxed as done in PUMIPic. Only the last slices of each chunk will have a different workload which is bounded by the parameters C and V.

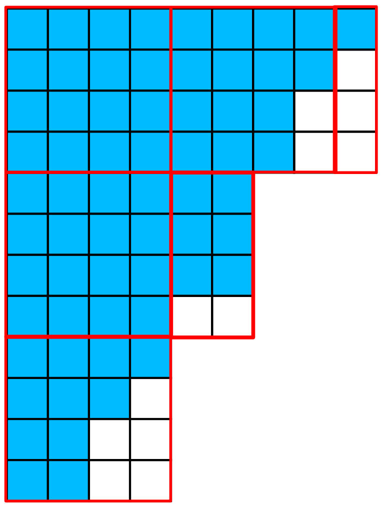

Figure 4.7: Sell-C-Sigma with full sorting and vertical slicing. Each box of entries represents a block of work given to a block of threads.

The addition of more breaks into the structure comes with more memory usage than the CSR. Given we have N elements and M particles with D data types per particle, let P be the number of padded entries and S be the total number of slices across all chunks. The SCS with vertical slicing has three one-dimensional arrays. The first is an $offsets$ array similar to the CSR. The size of the $offsets$ array is equal to the S and each entry denotes the starting index of the ith slice. The second is the $values$ arrays sized $M + P$. The third array is a $mask$ per entry in the $values$ array which states whether the entry is a particle or a padded cell also sized $M + P$. This results in a total of $(D+1)(M+P)+S$. P and S are dependent on the choices of the parameters C, σ, and V as well as the distribution of the particles across elements. The number of slices can be upper bounded by $S < \lceil \frac{M+P}{C*V} \rceil + \lceil \frac{N}{C} \rceil$ since each slice stores $C * V$ particles except the last slice in each chunk. There are $\lceil \frac{N}{C} \rceil$ chunks which each could have one extra slice at the end. For P, Besta et al. [132] note the effects for padding are negligible for larger values of σ such as $\sigma > \sqrt{N}$ because the added sorting leads to minimal padding in the structure.

The SCS satisfies the requirement for a mesh-based particle data structure as the particles in an element are accessible from the equivalent row in the SCS. The tuning of σ and vertical slicing allow near-even distribution of workload to threads while the parameter C and padding allow tunable performance on different accelerator hardware in terms of memory layout and access pattern. These parameters also allow the structure to be adapted to the different simulation characteristics including particle distribution and density using V to account for the imbalance of particles across elements.

4.1.4.4 Array of Structs of Arrays

A structure designed for accelerators that is used in traditional PIC simulations is the Array of Structs of Arrays (AoSoA) [133]–[135]. The AoSoA is as its name states an array where each entry is a Struct of Arrays (SoA) where each inner array has fixed length C. C is generally based on the $SIMD\_Width$ of the accelerator while the outer array is sized with enough structs to fit all the data needed. For M particles, the size of the outer array would be $\lceil \frac{M}{C} \rceil$. The last struct in the AoSoA may not be filled with particles. To maintain the fixed-length inner arrays, the last struct is padded similar to the SCS. The number of arrays in the struct is dependent on the different types of data that need to be associated with each particle. Figure 4.8 shows an example AoSoA with one struct shown containing three arrays x, y, and z and $C = 5$.

On its own, the AoSoA does not satisfy the requirement for us to relate particles to the mesh elements they are associated with. In traditional PIC simulations, there is an array in each struct that stores the element each particle is within. To use the AoSoA in a mesh-based PIC simulation an additional indexing layer is added to attribute SoAs to mesh elements. In this scheme, mesh element i with M_i particles will be assigned $\lceil \frac{M_i}{C} \rceil$ SoAs that are continuous in the outer array. An additional $offsets$ array is used to provide the starting index in the AoSoA for each element. Figure 4.9 shows an example AoSoA next to the previous CSR and SCS examples. Each row of the AoSoA represents an element. Each box of $C = 4$ entries is an SoA for the element of that row. Note in this version of the

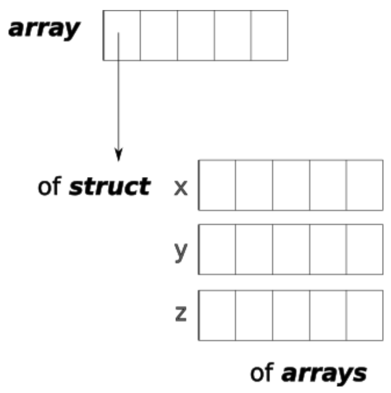

Figure 4.8: An AoSoA with five SoAs with $C = 5$ each storing three types arrays for x, y, and z.

AoSoA the last SoA of each element will have padded entries if the number of particles is not divisible by C.

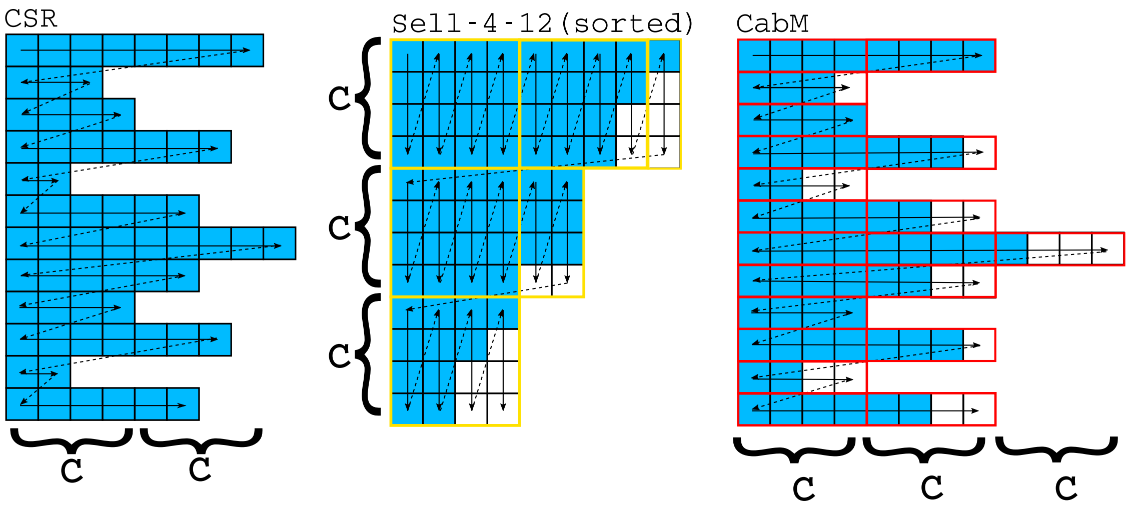

Figure 4.9: Particles stored in the CabM structure adjacent to the CSR and SCS for the same particles. Each group of C entries represents one SoA assigned to the element for that row.

In PUMIPic, the implementation of the mesh-based AoSoA is built off Cabana [118]. Cabana is a library for using AoSoA in traditional PIC simulations. Within PUMIPic, the $offsets$ array is added on top of the Cabana library resulting in the structure we call CabanaM (CabM). With the addition of the $offsets$ array, CabM satisfies the requirements of grouping particles in the same elements together in memory. Each thread performs work on an SoA leading to an equal distribution of work regardless of the distribution of particles to elements. Since C is set based on the $SIMD\_Width$ of the accelerator the layout of data matches the access pattern of the accelerator. So CabM also satisfies the requirements laid out for our particle structure.

The memory usage of CabM is the memory usage of an AoSoA plus the $offsets$ array. As with the SCS, the exact memory usage is dependent on the parameter C. For N mesh elements and M particles and D particle data types, let P be the number of padded entries. The size of the outer array is $\lceil \frac{M+P}{C} \rceil$ while summing all of the SoAs together results in a total of $D*(M+P)$ entries. The $offsets$ array is sized N so the total memory usage of CabM is $D*(M+P)+\lceil \frac{M+P}{C} \rceil + N$. We can upper bound P since only the last SoA of each element will contain padded cells by $P < (N-1)*C$.

4.2 Algorithms for Supporting PIC Operations

4.2.1 Supporting Particle-Mesh Interactions

For the Field to Particle, Charge Deposition, and Particle Push steps, it is vital to have efficient ability to interact between the fields associated to mesh entities and the particles. In PUMIPic, it is important to abstract these operations such that the different options of particle structures can be switched out seamlessly in a PIC code. As such, PUMIPic provides its own ParallelFor API that facilitates operating on the particles of a given mesh element. The ParallelFor API executes a user defined operation on the GPU for each particle individually. The indexing details of each particle structure is hidden from the user that results in both easy development for the user and allows the application to easily change which particle structure the simulation uses.

Algorithm 7 shows the general format of using the ParallelFor API. On lines 1-5 the user defines a lambda that would include the code to operate on a single particle. The three arguments provided here are the parent element's index, *elm_index*, the particle index, *ptcl_index*, and the *mask* that identifies if the index refers to a real particle or a padded cell like found in the Sell-C-Sigma structure. In many cases the *mask* is used as shown on line 2 to only operate on indices that represent real particles. However, when possible the mask can be ignored to avoid branch divergence and improve performance on the GPU. Within the lambda, operation specific code is written that can utilize the mesh fields and mesh adjacencies using *elm_index* or the particle's data using *ptcl_index*. The execution of the lambda on the GPU for all particles is done by the call to ParallelFor on line 6.

Algorithm 7 Format for performing iteractions between the mesh and particles using PUMIPic

```
1: $lambda$ =PS_LAMBDA($elm\_index$, $ptcl\_index$, $mask$) {
2:     if $mask == true$ then
3:         Perform operations using $elm\_index$ and $ptcl\_index$.
4:     end if
5: }
6: ParallelFor($ptcls$, $lambda$)
```

4.2.2 Unstructured Mesh Field Synchronization

To maintain dynamic mesh field information across processes, it is necessary to perform field synchronizations each iteration. This synchronization is performed after the charge deposition operation. Each process has contributions associated with the mesh entities stored in a field that must be accumulated before the solve operation. Due to the additional buffering included in the PICparts, there is a significant amount of data that must be communicated across processes. When the PICparts are constructed with full buffering of the mesh, field synchronization can be performed using a single reduction across all processes. The general approach for distributed mesh buffering is to use a fan-in fan-out communication protocol [136] leveraging the full part buffer regions. An important characteristic of the PICpart definition used in this algorithm is that buffering is communitive in other words if $\overline{P_i}$ has P_j buffered then $\overline{P_j}$ has P_i buffered.

The fan-in fan-out algorithm is a three-stage approach shown in Algorithm 8. The algorithm begins with a $field$ defined on the mesh entities of the PICpart $\overline{P_i}$ containing the contributions on process i. The FieldSync procedure has three steps on lines 1-5: FanIn, Reduction, and FanOut. The FanIn procedure on lines 6-11 iterates over each part, P_j, in the buffer B_i to perform non-blocking communication with each process j. On line 8 the field values associated to the mesh entities in P_j are sent to process j. Similarly, the field values associated to the mesh entities in P_i are recieved into local copies from process j on line 9. Since the mesh field is ordered by part and each part is fully buffered the communications are done by bulk communications of contiguous blocks of memory for each part in the same order on the sending and receiving process.

After FanIn, the contributions of each copy of the field for the mesh entities in P_i are reduced using the *reduction_op* such as sum, max, or min in the Reduction procedure on lines 12-17. For each part in the buffer, a wait on the nonblocking receive in FanIn is performed on line 14. Once the copy of the field contributions associated to P_i is receieved from process j it is reduced using the *reduction_op* on line 15. While the algorithm lists this as a loop through each process sequentially, using the MPI_WaitAny function allows this loop to be processed in the order the communications are received.

The FanOut procedure follows the Reduction operation in order to send the reduced values to each buffered process on lines 18-24 . This operation similar to the FanIn procedure loops over each part in the buffer B_i to perform nonblocking communications. For each buffered part, P_j, the reduced field values associated to P_i are sent to process j on line 20 while the reduced field values associated to P_j are received on line 21. This time the receive communication is copied directly into the field instead of a temporary copy. The final step is to wait for all of the nonblocking communications to complete on line 23.

Algorithm 8 pseudocode for fan-in fan-out algorithm for mesh field synchronization.

```
 1: function FieldSync($\overline{P_i}, field, reduction\_op$)
 2:     FanIn($\overline{P_i}, field$)
 3:     Reduction($\overline{P_i}, field, reduction\_op$)
 4:     FanOut($\overline{P_i}, field$)
 5: end function
 6: function FanIn($\overline{P_i}, field$)
 7:     for all $P_j \in B_i$ do
 8:         Nonblocking Send $field[P_j]$ to process $j$
 9:         Nonblocking Recv temporary copy of $field[P_i]$ from process $j$
10:     end for
11: end function
12: function Reduction($\overline{P_i}, field, reduction\_op$)
13:     for $P_j \in B_i$ do
14:         Wait for field $f$ from any process $j$
15:         Reduce $f$ into $field[P_i]$ using $reduction\_op$
16:     end for
17: end function
18: function FanOut($\overline{P_i}, field$)
19:     for all $P_j \in B_i$ do
20:         Nonblocking Send $field[P_i]$ to process $j$
21:         Nonblocking Recv $field[P_j]$ from process $j$
22:     end for
23:     Wait for all communications to finish
24: end function
```

4.2.3 Adjacency Search

The push operation moves the particles to new positions. In addition to recording the new position, the parent element the particle is within after moving must be recorded before the next PIC operation. There are two approaches to locating the new parent elements of particles. The first is to construct a spatial data structure, such as a uniform grid or spatially-based tree decomposition, over the domain to quickly find the potential elements a particle may be within [137]. An alternative approach that we call adjacency search is to directly use mesh topology to iteratively step toward the new parent element starting at the old element. Adjacency search is more performant than using a spatial data structure when particles move a short distance each iteration, where short distance means that the number of elements a particle traverses through a single push operation is at most a few. This aligns with the plasma physics simulations that PUMIPic targets wherein a given iteration most particles either do not leave the element they start in or move no more than two or three elements in any iteration. Additionally, no additional data structure needs to be allocated for adjacency search so there is both a memory and runtime improvement.

Adjacency search is an iterative algorithm where each iteration an exit face from the current element towards the new particle position is determined using a set of testing criteria [138]–[142]. Figure 4.10 shows an example particle and its path through elements using adjacency search. The open circle is the particle's original position and the closed circle is the particle's new position. Each arrow represents one iteration of the search moving across an edge to the next iteration's parent element. If a correct testing criteria is used then the search will converge to the mesh element the new particle position is in.

The general form of determining the exit face can be written as: given a particle's original position p_o, its final position p_f, and the current iteration's mesh element M_i^d that is bounded by N vertices $M_1^0, ...M_N^0$, find an exit face $M_j^{d-1} \in \{M_i^d\{M^{d-1}\}\}$.We will assume without loss of generality that the vertices $M_1^0, ..., M_N^0$ are ordered counter-clockwise around the element. There are different choices for criteria to determine the exit face that have tradeoffs in terms of runtime cost, code complexity, and proper care of edge cases. Zhou and Lechziner [143] for a 2D mesh suggest using the cross product between the vector $M_k^0 M_{k+1}^0$ and the vector $M_k^0 p_f$. The sign of the z-component of the cross product that we will denote as L_k reveals the direction the particle's new position from the mesh edge bounded by M_k^0 and M_{k+1}^0. Namely, $L_k > 0$ means the particle is to the left of the edge and $L_k < 0$ means

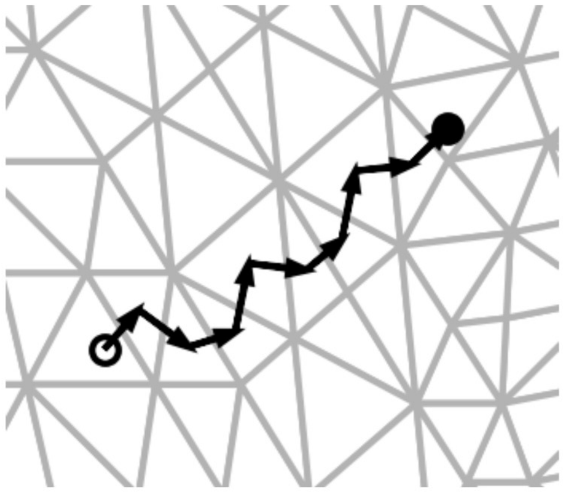

Figure 4.10: Path of an adjacency search for a particle on a 2D triangular mesh using edge adjacencies.

to the right. $L_k = 0$ is a special case meaning the particle's final position is along the edge. This test called the particle-to-the-left or P2L test by Chordá et. al. [139] is performed for each edge until an edge with $L_k < 0$ is found where that edge is determined as the exit face. If $L_k >= 0$ for all edges then p_f is inside the current element. Chordá et. al. note that this method can fail in special cases where the search will circle around the final element indefinitely. A robust approach by Chen and Pereira [144] combines the P2L test with the particle's path to move directly towards p_f. First, the P2L test is performed on each edge of the mesh element to determine all possible exit faces with $L_k < 0$. The possible exit face that intersects the line from p_o to p_f is selected as the exit face for the iteration. This approach is more robust and requires fewer iterations, at the cost of increased computation by performing the line intersections. Chordá et. al. [139] suggests a method which determines if the particle path intersects the edge first. For this method a second test is defined named trajectory-to-the-left or T2L that calculates T_k as the z-component of the cross product between $p_oM_k^0$ and p_op_f. The T2L test determines the direction the particle trajectory is from the given vertex M_k^0. Again, $T_k > 0$ means the particle trajectory is to the left of the vertex while $T_k < 0$ means the path is to the right of the vertex. For each mesh edge bounded by the vertices M_k^0 and M_{k+1}^0 the T2L test is performed for both vertices. If the signs of T_k and T_{k+1} differ then the particle path crosses the edge. For the edges that pass this test the P2L test is calculated and if $L_k < 0$ then this edge is the exit face for the iteration. This method was shown to be faster than the previous methods.

Chordá et. al. [139] also discuss extensions of these methods to 3D. For the Zhou and Lechziner [143] method, the P2L test is redefined in 3D space as the particle-towards-the-inside or P2I test that is checked for each face of the mesh region. The P2I for the face, M_j^2 tests all adjacent triplets of vertices M_i^0, M_{i+1}^0, and M_{i+2}^0 in $\{M_j^2\{M^0\}\}$ ordered counterclockwise to the vector $M_{i+1}^0 p_f$ by the calculation in Equation 4.1. The first face to fail the P2I test is the exit face and if the test passes for all faces of the region then the parent element is found. Chordá et. al. note that in 3D the chance of this algorithm circling the final element is significantly larger than the 2D case.

$$Sign[(M_{i+2}^0 M_{i+1}^0 \times M_{i+1}^0 M_i^0) \cdot M_{i+1}^0 p_f] > 0 \tag{4.1}$$

For the Chen and Pereira method, Chordá et. al. suggest the method could be extended to calculate the particle path's intersection with the cell face. They note that in 3D this intersection calculation is significantly more expensive to compute. Also noted is that for the case of nonplanar faces the calculation is more complex. Macpherson [138] et. al. detail a method that supports the case of nonplanar faces for convex mesh regions by computing two values λ_a and λ_c for each face of the region using the region's center C_c, the face's center, C_f, and the normal of the face S. Equation 4.2 details the computation of λ_a and λ_c.

$$\lambda_a = \frac{(C_f - p_o) \cdot S}{(p_f - p_o) \cdot S} \qquad \lambda_c = \frac{(C_f - C_c) \cdot S}{(p_f - C_c) \cdot S} \tag{4.2}$$

λ_a tests for faces that the particle path crosses, which is determined when $0 \leq \lambda_a \leq 1$. When the particle's final position is close to a mesh vertex, the nonplanar faces can cause the $lambda_a$ test to result in the wrong element being found. So, the $lambda_c$ tests the faces if the particle starting position is the center of the region as in general this will determine the same exit face, but not have issues related to being near the nonplanar face. For all faces where $0 \leq \lambda_c \leq 1$, the face with the smallest $lambda_a$ is determined as the exit face. The process terminates when $lambda_c < 0$ or $lamdba_c > 0$ for all faces of the region.

In PUMIPic, since the mesh only uses simplex elements, adjacency search is implemented using barycentric coordinates and line-face intersections in 3D to determine the exit face. To determine if a particle is in its new parent element barycentric coordinates of p_f in the current element are calculated. If all three, in 2D, or four, in 3D, coordinates are positive then the particle is in its parent element. This check is performed at the beginning of the search procedure to determine all of the particles that have not left their original element to reduce the number of particles that need to be searched.

To determine the exit face, the barycentric coordinates for p_f in M_i^d are used. The face opposite the vertex M_k^0 with the largest negative B_k is selected as the exit face. When an application requires knowledge of intersections with the model boundary, line-face intersections are calculated to determine the point of intersection, X_j between the particle path $p_o p_f$ and the plane the mesh face is on. The line-face intersections may also be used to more accurately determine the exit face along the path of the particle that will be included in the discussion of it here. First, the unit normal, N, in the direction out of the tetrahedron is computed as shown in Equation 4.3.

$$N = \overrightarrow{M_0^0 M_1^0} \times \overrightarrow{M_0^0 M_2^0} \tag{4.3}$$

The two inner products I_o and I_f are computed by Equation 4.4. If $I_o >= 0$ and $I_f >= 0$ then the particle path intersects the face and is moving out of the tetrahedron.

$$I_o = \langle \overrightarrow{p_o M_0^0}, N \rangle, I_f = \langle \overrightarrow{p_o M_0^0}, N \rangle \tag{4.4}$$

The intersection point, X_j, is then determined using Equation 4.5. To determine if X_j is within the bounds of the triangle, barycentric coordinates, B_0, B_1, B_2, for X_j in the triangle M_j^{d-1} are calculated. If $B_k > 0$ for all k then M_j^{d-1} is the exit face.

$$X_j = p_o + \frac{I_o}{\langle \overrightarrow{p_o p_f}, N \rangle} (\overrightarrow{p_o p_f}) \tag{4.5}$$

The barycentric coordinate method is an efficient choice for determining the exit face as the calculations are already performed in order to determine if the element has reached its new parent element. However, the problem is that this method moves towards the particle's final position in the direction of greatest motion instead of along the particle path. In geometries with small holes in the mesh that particles may be moving around, the barycentric coordinate method may incorrectly calculate or miss wall collisions that the line-face intersection method would capture more accurately. Optimizing the search routine between the usage of the fast barycentric coordinate calculation vs the more expensive line-face intersections is left for future work. In applications using the PUMIPic library, the adjacency search is set to use line-face intersections if either wall collisions need to be kept track or if the geometry has holes, otherwise the barycentric coordinates are used.

4.2.4 Particle Structure Rebuild

After the particles are pushed and the new parent elements are determined using adjacency search, the particle structure must regroup the particles by the new parent elements. This regrouping, referred to from here on as rebuild, in general, is performed by fully reconstructing the particle data structure. The choice to fully reconstruct the SCS as opposed to supporting a growth mechanism is largely due to the parallelism supported on a GPU. Supporting a per element growing structure would result in a large amount of contention between the concurrent threads of a GPU. When only performing a small number of updates it is efficient to support these incremental updates on the GPU [145], however when a majority of the structure needs changes then full reconstruction is the fastest solution [145], [146]. Reconstructing the data structure is done through data-parallel operations per particle along with efficient reductions that are better designed for running on GPUs. The general approach presented in this section will be shown in terms of the SCS particle structure, but the same method is applied for all particle data structures.

The rebuild algorithm constructs a new particle data structure with the new distribution of particles. Then, the particle information is copied from the old structure to the new structure and finishes by destroying the old structure. Figure 4.11 shows an example of rebuild being performed. On the left is the SCS with filled cells and padded cells with element rows labeled on the side. Numbers inside the cells represent the element the particle is moving into, while no label means the particle did not leave its previous parent element. The right side of the figure shows the new SCS after rebuild. Note the element labels are in a different order now to maintain the sorting

When there are a small number particles moving to new elements it may be beneficial to perform changes to the structure directly instead of a full rebuild [145]. To support incremental changes an in-place rebuild is implemented that utilizes padding in the structure. Instead of copying the data from an old structure to a new structure, rebuild can be performed in-place by copying the particles moving to a new element into the padded cells of the element. The in-place rebuild can only be performed when each element receiving particles has at least as many padded cells as incoming particles. If this is not the case, then the full reconstruction rebuild is used. The occurrence of in-place rebuild can be increased

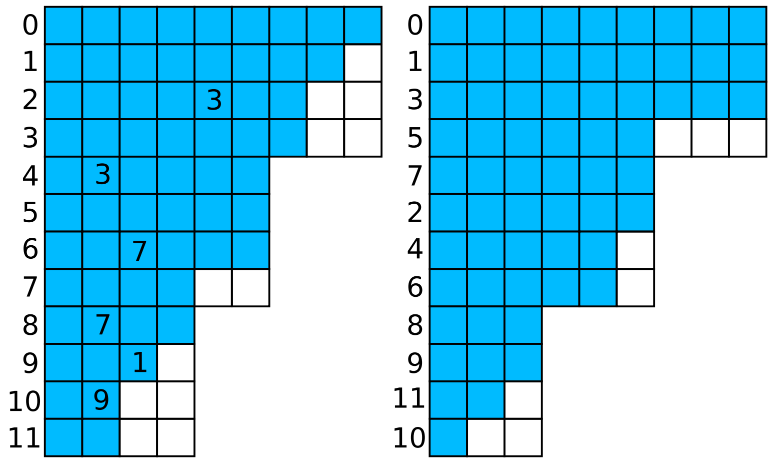

Figure 4.11: Example SCS before rebuild (left) with element numbering to the left. New parent elements of particles moving are listed within cells. The SCS after rebuild (right) with a new memory allocation along with resorting of the rows.

by adding additional padding to each element when the particle structure is constructed at the cost of more memory usage.

Figure 4.12 performs in-place rebuild on the same example from Figure 4.11. In this case, the right SCS uses the same memory as the original SCS after performing rebuild. Darker cells depict the new location of particles that have moved to a new element. Note that in-place rebuild does not perform sorting of the rows based on length. Along with the empty cells left throughout the structure, it is still ideal to perform a full rebuild after several iterations to improve the data layout.

An additional capability of the rebuild step is to introduce new particles or remove particles from the system. For the full rebuild algorithm, new particles are added when creating the new particle structure and removed particles are ignored as if they were padded cells resulting in their omission from the new data structure. The in-place rebuild has to account for the new particles when checking if the in-place operation is possible. Particles leaving the system can act as padded cells allowing particles to replace those entries in the current rebuild.

4.2.5 Particle Migration

For particles that were pushed to an element outside the safe zone of the process, the particles must be migrated to a new process such that they are well within the safe zone of the

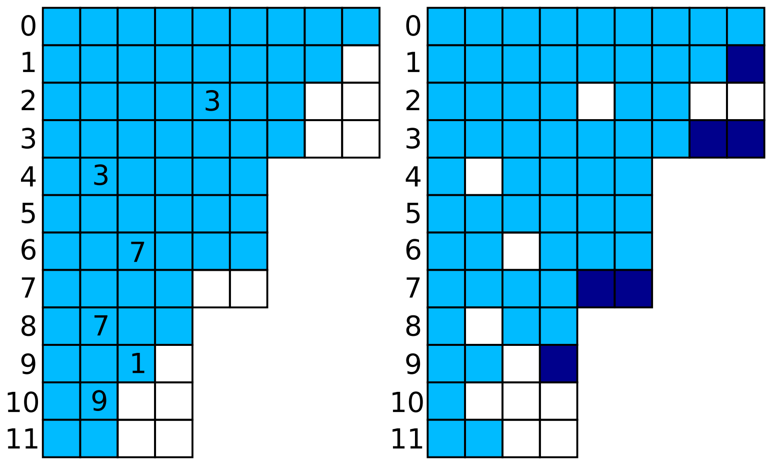

Figure 4.12: Example SCS before rebuild (left) with element numbering to the left. New parent elements of particles moving are listed within cells. The SCS after in-place rebuild (right) using the same memory as the initial SCS. Darker cells show the particles that moved to a new cell.

new process. Due to the overlapping of PICparts, there are multiple processes that a particle can be migrated to. A straightforward strategy is to send migrating particles to the process which has the new parent element within its part. This guarantees the particles are always sent to a process that the element is within the safe zone. However, continuous execution of this strategy over several iterations will likely result in an imbalance of the particles across the processes. In this case, load balancing the particles is required to reassign particles to have a better distribution across processes. The method used in PUMIPic for load balancing is described in Section 4.3

The particle migration routine is listed as pseudocode in Algorithm 9. The Migrate procedure on lines 1-19 takes in the particle structure in *ptcls*, the new process for each particle in *new_procs*, the new parent element from adjacency search in *new_elms* and an optional argument for new particles to add in *new_ptcls*. The first step is to count the number of particles that are being sent to each other process and store them in the *send_counts* array on lines 2-5. The counts are then used to perform a nonblocking all-to-all communication that receives the number of particles each process will send to the current process on all processes on line 6. While the nonblocking communication is being performed, the GPU is used to gather the particle data that will be sent to each process in a call to the GatherParticleData procedure on line 7. This procedure listed on lines 20-29 creates the *send_ptcls* loop that stores the particle information for every particle that is being migrated to a new process. The *send_ptcls* array is sectioned such that the particles being sent to a given process, p, are stored in a contiguous block of memory represented by *send_ptcls*[p]. The parallel loop on lines 22-27 is executed on the GPU to copy the memory from the particle structure to the *send_ptcls* array. Any particles that are being sent to a new process are marked for removal on line 25 that shall be handled in the call to Rebuild at the end of Migrate. After the particle information is gathered and the nonblocking communication has finished on line 11, a new round of nonblocking communications is performed to communicate the particle information to the correct processes on lines 8-15. These communication fill a new array *recv_ptcls* that stores the newly received particles from each sending process. One all particles have been received on a process on line 16, the received particles in *recv_ptcls* is combined with the new particles in *new_ptcls* on line 17. At the end, Rebuild as described in Section 4.2.4 is performed that rebuilds the particle structure moving all particles to their new element as noted in *new_elms*, adds the new particles and received particles in *new_ptcls*, and removes any particles that were marked for removal on line 25.

By default, the particle migration routine does not know which processes it will be communicating particles between. As such, the routine must perform the all-to-all communication on line 6 to receive the counts of particles being sent and received on all processes. When scaling to larger process counts, all-to-all communications can be a bottleneck [136]. Alternatively, the number of processes communicating can be vastly reduced by using the partition of the mesh and PICparts. Particles can only be migrated between two processes that overlap PICparts. Since PICparts are a collection of parts, $P_i \cup B_i$, each process only needs to communicate with the PICparts in B_i. The all-to-all communication can be replaced with nonblocking sends and receives when this information is available to the particle structure. The all-to-all approach is still important when the number of processes is low or the buffers are include a large percentage of the parts such as when the mesh is fully buffered. As such, both approaches are supported with the all-to-all method being default unless the partition information is provided to the particle structure.

4.3 Particle Load Balancing

To maintain parallel efficiency during a PIC simulation it is important to ensure the balance of particles is maintained as they move through the domain and migrate to different processes. Fast dynamic load balancing methods must be applied throughout the simulation

Algorithm 9 pseudocode for particle migration.

```
1: function MIGRATE($ptcls$, $new\_procs$, $new\_elms$, $new\_ptcls$)
2:     $send\_counts$ = DeviceArray($num\_procs$)
3:     parallel_for Particle $p \in ptcls$ do
4:         ATOMICADD($send\_counts[new_procs[p]]$, 1)
5:     end parallel_for
6:     $recv\_counts \leftarrow$ Nonblocking AllToAll communication of $send\_counts$
7:     $send\_ptcls \leftarrow$ GATHERPARTICLEDATA($ptcls$, $new\_procs$)
8:     for all Process $p$ such that $send\_counts[p] > 0$ do
9:         Nonblocking send particle data in $send\_ptcls[p]$ to process $p$
10:    end for
11:    Wait for AllToAll communication to complete
12:    Initialize $recv\_ptcls$ with enough space to receive incoming particles
13:    for all Process $p$ such that $recv\_counts[p] > 0$ do
14:        $recv\_ptcls \leftarrow$ Nonblocking recieve particle data from process $p$
15:    end for
16:    Wait for communications to complete
17:    $new\_ptcls \leftarrow recv\_ptcls$
18:    REBUILD($ptcls$, $new\_elms$, $new\_ptcls$)
19: end function
20: function GATHERPARTICLEDATA($ptcls$, $new\_procs$)
21:    Initialize $send\_ptcls$ with enough space to store each particle being sent
22:    parallel_for Particle $p \in ptcls$ do
23:        if $new\_procs[p]! = self$ then
24:            $send\_ptcls[new\_procs[p]] \leftarrow ptcls.data[p]$
25:            Mark particle $p$ for removal
26:        end if
27:    end parallel_for
28:    return $send\_ptcls$
29: end function
```

to manage the distribution of particles across the PICparts that can scale to trillions of particles. Directly operating on particles in simulations with over a trillion particles would be infeasible for providing fast repartitioning of particles so it is vital to create an abstraction of the problem with a significant decrease in the size. We leverage the properties of PICparts and the power of the N-graph in EnGPar to create a graph that represents the potential particle migrations. First, we define the overlap of safe zones in Section 4.3.1. These regions of overlapping safe zones is used to construct the N-graph on each process. A modified version of EnGPar's diffusive load balancer is applied to this graph that creates a plan for migrating the particles. The construction of the N-graph and the modifications to the load balancer is descibed in Section 4.3.2

4.3.1 Overlapping Safe Zones

Recall that PICparts are composed of a part, P_i, and a buffer, $B_i = \bigcup P_j$ of nearby parts. Furthermore, a subset of each PICpart is denoted as the safe zone S_i that represents the elements that can store particles on process. For balancing particles it is required that particles are migrated to a process for which its parent element is in the safe zone for the target process. For each element M_i^d, let $\{\overline{P}(M_i^d)\}$ be the set of PICparts such that $M_i^d \in S_j$ for all $\overline{P_j} \in \{\overline{P}(M_i^d)\}$. The set $\{\overline{P}(M_i^d)\}$ represents the list of PICparts that a particle in element M_i^d can be migrated to and remain in a safe element on its new process. We can further this notion by considering overlapping safe zones. We define an overlapping safe zone, $\overline{S}_P$, for a set of PICparts P, as the set of mesh elements, M_i^d, such that $\{\overline{P}(M_i^d)\} = P$. $\overline{S_P}$ gathers the list of mesh elements that the PICparts in P can share particles freely without breaking the requirement dictated by the safe zones. Figure 4.13 shows a portion of four parts on the left and three $\overline{S}_P$ in the middle for the PICparts around the boundary of parts A and B. Every element in the mesh belongs to exactly one $\overline{S}_P$ which allows the migrating of particles to be restricted to only the overlapping safe zones instead of each element individually.

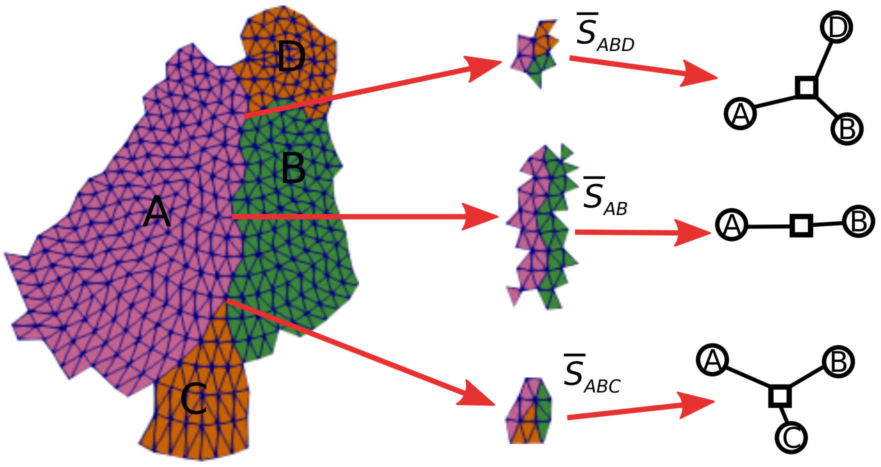

Figure 4.13: Left: A section of a triangular mesh with four parts A, B, C, and D. Middle: The three overlapping safe zones around the boundary between parts A and B. Right: The subhypergraphs for each of the three overlapping safe zones.

4.3.2 Applying EnGPar to Balance Particles

Using the notion of the overlapping safe zones we construct a hypergraph using the N-graph format in EnGPar introduced in Chapter 2. The hypergraph consists of a subhypergraph for each $\overline{S}_P$ where each subhypergraph has one graph vertex for each PICpart in P that are all connected by a single hyperedge. The graph vertex for a certain PICpart is owned by the process for that PICpart. The right side of figure 4.13 depicts the subhypergraphs for the three $\overline{S}_P$ around the boundary of cores A and B. The circles represent the graph vertices for each PICpart in P that are connected by the box representing the hyperedge for the subhypergraph The union of all subhypergraphs makes up the N-graph used in EnGPar. The resulting graph will have $\sum |\overline{S}_P|$ graph vertices and one hyperedge for each $\overline{S}_P$. The N-graph vertex and hyperedge counts for the 11.4 million element mesh are provided in Table 4.1. Note the orders of magnitude difference between the number of mesh elements in each PICpart compared to the number of graph vertices in the N-graph. For example, the 48 PICpart case has an average of 2.3 million mesh elements per PICpart where as the corresponding N-graph of overlapping safe zones has less than two thousand vertices. This substantial decrease in problem size results in the balancing of particles being much faster and less memory consuming.

Table 4.1: PICpart and N-graph entity counts for an 11.4 million element mesh excluding three layers from PICpart boundaries as the safe zone.

Number of PICparts	6	12	24	48
Average Elements per PICpart	8.8M	5.3M	3.7M	2.3M
Total Graph Vertices	20	102	450	1931
Total Graph Hyperedges	3	19	69	266

To use this hypergraph for balancing particles, first, the particles are associated with each graph vertex as weights. Every PICpart counts the number of particles in each $\overline{S}_P$ and applies the total as the weight for the owned graph vertex corresponding to the $\overline{S}_P$ for the PICpart. The sum of the weights on all owned graph vertices for a process will equal the number of particles on the PICpart. The sum of the weights on a subhypergraph is the total number of particles across all PICparts in one $\overline{S}_P$. Transferring weight between the connected vertices of the subhypergraph would be equivalent to migrating particles from one PICpart to another PICpart where the particles' parent element will remain safe.

EnGPar is then tasked to balance the total weights on each process using diffusive load balancing. The method applied in this case is slightly different from the method described in Section 2.3 as the intention is to migrate weight between connected vertices instead of migrating vertices as EnGPar is designed to do. To properly perform this the selection stage is modified such that when processing a cavity, weight is transferred across the hyperedges instead of migrating the vertices to the neighboring process. The other steps of EnGPar's diffusive balancer remain the same for determining which parts send and receive weight and how much weight to send each iteration. Algorithm 10 lists pseudocode for the new weight selection routine. The WEIGHTSELECTION procedure, defined on line 1 takes in the N-graph, G, that is made up of the set of vertices V and hyperedges H^0 and the *targets* array that stores the amount of weight to send to each process this iteration. A temporary array, *sending* is allocated on line 2 sized number of processes that keeps track of the amount of weight planned to be sent to each process. The procedure then iterates over each subhypergraph via the set of hyperedges on line 3. The owned vertex, V_{self} of the subhypergraph is found on line 4. For each hyperedge, H_i^0, the vertices, V_j, in $\{H_i^0\{V\}\}$ are iterated over on line 5. The owner of V_j is retrieved on line 6 followed by determining how much weight, if any, to send from V_{self} to V_j on line 7. This weight is calculated as the minimum of $\frac{\alpha * w(V_{self})}{|\{H_i^0\{V\}\}|}$ and $targets[own] - sending[own]$. The former is a small portion of the weight of a vertex calculated as the weight of the vertex averaged over the number of potential targets multiplied by the step factor α. The latter is the remaining weight to be sent to the target process. The latter enforces the amount of weight not to exceed the target weight for the iteration. If the weight calculated is greater than 0, the migration plan is updated on line 9 with the migration of *send_weight* from V_{self} to V_j. This migration represents sending *send_weight* particles in the mesh elements of $\overline{S_i}$ from $\overline{P_{self}}$ to $\overline{P_j}$. Lines 10-11 update the *sending* array and the weight of the vertex V_{self} based on the weight sent. Just as EnGPar's original load balancer, this selection process is repeated across iterations until the target imbalance of particles is reached or the process stagnates.

The result of EnGPar's weight diffusion is a plan detailing the amount of weight to send from each graph vertex to other graph vertices to reach the target imbalance of particles. This plan equates to the number of particles that need to be migrated in each $\overline{S}_P$. The final stage to finish balancing particles is to select particles on each PICpart for each $\overline{S}_P$ to satisfy the plan.

Algorithm 10 Pseudocode for the modified weight selection in EnGPar

```
1: function WeightSelection($G = (V, H^0), targets$)
2:     $sending \leftarrow \{0\}$
3:     for all $H_i^0 \in H^0$ do
4:         $V_{self} \leftarrow$ the vertex owned by this process in $\{H_i^0\{V\}\}$
5:         for all $V_j \in \{H_i^0\{V\}\}, V_j \neq V_{self}$ do
6:             $own \leftarrow owner(V_j)$
7:             $send\_weight \leftarrow \min\{\alpha * w(V_{self}), targets[own] - sending[own]\}$
8:             if $send\_weight > 0$ then
9:                 $plan \leftarrow \{V_{self}, V_j, send\_weight\}$
10:                $sending[own] += send\_weight$
11:                $w(V_{self}) -= send\_weight$
12:            end if
13:        end for
14:    end for
15: end function
```

4.4 Summary

In this chapter we presented the PUMIPic library for implementing PIC codes with a mesh-based approach where particles are always grouped in memory by the parent element to facilitate easy distribution of the unstructured mesh data structure and particle structure. An expanded definition of a partition is detailed where using large buffering of mesh entities allows for increased control of interprocess communications and the migration of particles. Three particle data structures were introduced in the PUMIPic library, namely the Compressed Row Storage, the Sell-C-Sigma, and a mesh-based version of Cabana's Array-of-Structs-of-Arrays. Key algorithms of PIC simulations provided by the PUMIPic library for operations involving the mesh data structure and particle data structure are detailed. Finally, a method for performing dynamic load balancing of particles within the PICparts was introduced.

CHAPTER 5
SUPPORTING MESH-BASED IMPLEMENTATIONS OF FUSION PLASMA PARTICLE-IN-CELL SIMULATION

In the field of simulating plasma physics in magnetic confinement devices, the Particle-In-Cell (PIC) method is a commonly used approach [110], [113], [147]–[151]. For our studies in this chapter we will focus on using a model of the ITER tokamak device [152]. Figure 5.1 shows a mesh of the tokamak on the left and a cross section on the right labeling the high level regions. The inner region is the core of the plasma while the region outside the core to the wall is referred to as the edge. The boundary of the model is the material wall. The curve inside the edge region is the separatrix that denotes the end of the closed magnetic field lines. The point where the separatrix crosses itself is called an X-point. The point near the center of the plama core is the magnetic axis. Due to the complexity of the geometric model of ITER and the high fidelity of the fields, unstructured meshes are commonly employed for discretizing the domain [110], [113], [150], [151].

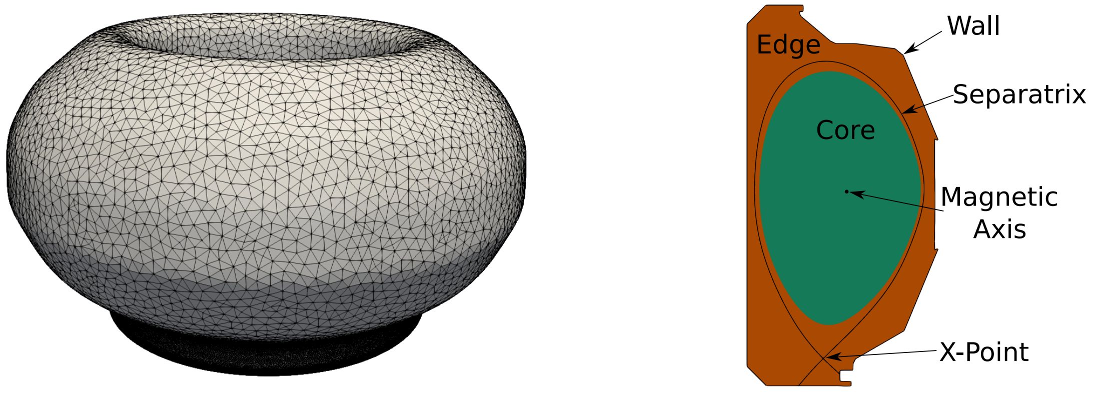

Figure 5.1: Left: A mesh of the plasma region of ITER tokamak up to the material wall. Right: A cross section of the region depicting the different regions and major points of the ITER model.

To understand and simulate the physics for these devices, various PIC codes have been designed to study specific portions of the domain, different physics involved with the plasma and the complexity of the systems. GENE [148], [149] and GEM [147] specialize

Portions of this chapter are to appear in: G. Diamond, C. W. Smith, C. Zhang, E. Yoon, and M. S. Shephard, "PUMIPic: A mesh-based approach to unstructured mesh particle-in-cell on GPUs," *J. of Parallel and Distrib. Comput.*, vol. 157, pp. 1–12, Nov. 2021. DOI: `10.1016/j.jpdc.2021.06.004` in the physics of the core of the plasma while XGC [110], [150] models the whole plasma specializing in the edge region up to the tokamak wall. Other codes like hPIC [96] and GITR [44], [153] specialize in interactions between the material wall and the plasma. To utilize multiple codes' specialities the Whole-Device-Modeling (WDM) project [152] seeks to couple various plasma-physics codes together to better model the physics of the device.

In this chapter we will discuss the ongoing work towards supporting the development of two plasma-physics codes using the mesh-based approach outlined in Chapter 4. The two simulations of interest are the X-Point Included Gyrokinetic Code (XGC) and the Global Impurity Transport Code (GITR). XGC [110], [150] is a 5D gyro-kinetic code specifically designed for the modeling of tokamak edge plasma physics while also being able to model in the core of the plasma. XGC employs a specialized discretization of the tokamak geometry based on the magnetic field to track ion and electron particles through the domain. The XGC simulation models the entire plasma from the magnetic axis out to the wall. GITR [44], [153] is a Monte Carlo PIC code that simulates impurities in the plasma and the physical iteractions caused by them. GITR uses background fields to perform operations on particles and a surface mesh of the domain to track wall collisions. These wall collisions are of greater importance in GITR as particles can reflect or cause sputtering when interacting with the model surfaces.

This chapter presents ongoing implementations towards mesh-based PIC simulations for XGC and GITR using the PUMIPic library named XGCm and GITRm respectively. Both XGC and GITR have a copy of the entire mesh on each process. XGCm and GITRm address the added complexity involved with supporting the partitioning of the mesh. While most of these details are handled using Omega_h and PUMIPic, specific algorithmic extensions and implementations were needed. In XGCm, the most notable of the affected operations are the Charge Deposition and Field Solve operations. The key portion of the Charge Deposition step for XGC called gyroaveraging involves scattering particle contributions to mesh vertices based on the magnetic field lines. This requires careful algorithmic design that is presented in detail to ensure performance on GPUs. The Field Solve step is complicated due to the extensive buffering involved in PICparts. Steps that have been taken to utilize the PETSc library are described as well as future plans for improvement. While the XGCm simulation is still in development and verification of physical results, a miniapp has been designed that uses pseudo physics along with the general memory access patterns of the XGC simulation.

Scaling results for this miniapp are presented up to most of the Summit supercomputer at Oak Ridge National Laboratories.

For GITRm, a most extensive change from GITR is performed. Instead of meshing the surface of the tokamak domain, we use a 3D graded unstructured mesh. The usage of a 3D unstructured mesh allows GITRm to accurately follow particles in the domain and use mesh adjacencies to track wall collisions. For this chapter, we will focus on two aspects of the GITRm simulation. First we will look at one of the most computationally expensive operations that is a distance-to-boundary calculation where particles that are close to the model surfaces have additional physics terms influencing the Particle Push step. The second aspect that we focus on is the balance of particles in the domain. In GITR particles are intialized on specific model surfaces and tracked as they move further into the domain. This leads to the intial distribution of particles being highly skewed and will drastically change as the simulation proceeds. PUMIPic's load balancing method is applied using the EnGPar library to improve the initial partition of particles and maintain good balance of particles throughout the simulation. Results for the particle load balancer are presented using the GITRm simulation.

The remainder of this chapter is structured as follows. Section 5.1 presents details on the XGC operations and the current progress towards mesh-based implementations. Section 5.2 details a miniapp developed to carry out core XGC PIC operations using the overall methods for data transfer and mesh-based operations used in XGC, but with simplified pseudo physics calculations. Using simplified pseudo physics allows us to focus on evaluating the performance of core PUMIPic operations. Results from this miniapp are presented using up to 4096 nodes of the Summit supercomputer at Oak Ridge National Laboratories is presented in Section 5.2.3. The mesh-based implementation of GITR is described in Section 5.3. The GITRm simulation is used to analyze the performance of PUMIPic's particle load balancing method with results presented in Section 5.4.

5.1 Supporting XGCm Implementation

In an XGC simulation, the particles exist in a 3D tokamak with cylindrical coordinates ρ, z, and ϕ. The tokamak domain is discretized into a set of cross sections perpendicular to the toroidal direction referred to as poloidal planes . The particle coordinates ρ and z define the particle's position within the poloidal plane and ϕ represents the angle in the toroidal direction. A poloidal plane is represented by a two-dimensional unstructured mesh of triangles. The same unstructured mesh is used for each poloidal plane. The interior domain between adjacent poloidal planes can be considered as an extrusion of the triangles around the tokamak through which the particles traverse. Figure 5.2 shows an example model for a poloidal plane. Although the particles representing the ions and electrons are moving at a very high velocity, they are generally field following meaning that they will mostly stay near the same flux surface. XGC takes advantage of this by defining a nearly field-following mesh using a set of curves of constant magnetic flux in the definition of the domain and its mesh.

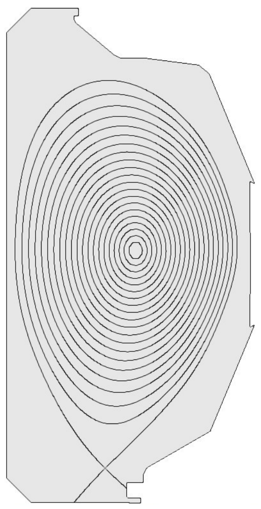

Figure 5.2: Geometric model of a poloidal plane. Curves on the model represent flux curves of constant magnetic flux.

5.1.1 Partitioning the XGC Domain

In XGCm, the simulation domain is made up of some number of poloidal planes each discretized by a 2D unstructured mesh. This introduces two dimensions of partitioning, the mesh and the poloidal planes. For the mesh partition, in order to minimize the number of particles migrated, it is ideal to partition the mesh along with the motion of particles. Therefore, the core of each PICpart is defined by a number of adjacent flux faces. The buffer of each PICpart will be some number of flux faces around the core. Figure 5.3 shows such a partition with one flux face per core. The region from the sepatrix to the material wall makes up an additional part. Since the mesh on each poloidal plane is the same, we use the same PICparts to represent each poloidal plane. Note that in actual XGCm simulations there are substantially more flux curves represented and the typical mesh on a single poloidal plane has on the order of a million triangles.

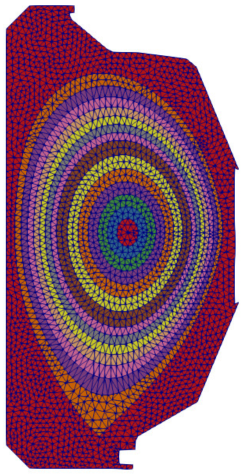

Figure 5.3: Partition of the poloidal plane with one part per flux face. The portion of the edge region outside the separatrix makes up an additional part.

With simulations using 96 poloidal planes [154] and upwards of 128 or more poloidal planes [155], storing all necessary mesh fields for a given PICpart for all poloidal planes would be impossible in the GPU memory. Thus, partitioning the planes is required. For XGCm, with M poloidal planes, the toroidal direction is partitioned into M sections defined by two adjacent poloidal planes. The poloidal planes are referred to as the forward plane and backward plane based on the direction of increasing angle around the tokamak. For N PICparts, the simulation will have $M * N$ processes.

Because the number of poloidal planes is fixed by problem design and increasing the number of mesh partitions eventually leads to degradation in performance, we introduce a third dimension of partitioning that allows more particles to be simulated given the same mesh partition and number of poloidal planes. We will refer to this third dimension as groups of size G where each pair of toroidal section and PICpart will be repeated on G processes. The notion of groups can be used in two different ways. For a given mesh partition and number of poloidal planes, if the simulation requires more particles, but doesn't have the memory available, then groups can be used to increase the available amount of memory by nearly a factor of G. If the simulation has enough particles that could be split across more GPUs for better performance, then the number of particles on any given process could be cut by a factor of G.

5.1.2 Charge Deposition: Gyroaverage

After the particle push, the charge deposition phase occurs where particles add contributions of charge onto the mesh fields. In a poloidal plane-based simulation, this involves every particle depositing onto the mesh vertices of the two poloidal planes surrounding the particle. As a 5D gyro-kinetic code, XGC is only following the mean path of the particles. Since the fast gyro motion that is not tracked is not small with respect to the mesh size, using simple mean path location charge deposition on the mesh via simple interpolation is not sufficient since this would only deposit charge to three mesh vertices. Two methods can be used to approximate the gyro-motion, using gyroaveraging matrices [156] or defining a set of gyro rings for each vertex [157], [158]. In XGCm, we use the gyro ring method. The general idea is that each mesh vertex is surrounded by R gyro rings. G points along the rings are projected to the poloidal planes along field lines to distribute contributions onto the mesh vertices. Figure 5.4 depicts the steps of the approximation. First, a particle's position is projected to a virtual plane, V, that resides halfway between the two forward plane, P1, and backward plane, P0. The mesh element on V containing the projected particle is found using adjacency search. Each vertex of the element distributes contributions on two gyro rings, g_0 and g_1, based on particle properties. Each point along the gyro rings are projected to planes P0 and P1. The mesh elements of these projected points are again found using adjacency search and contributions are divided between the vertices of these elements weighted by the barycentric coordinates of the point within the element.

Due to the number of projections and indirection of this approximation method, two mappings are constructed at the beginning of the simulation from each mesh vertex to the mesh vertices bounding the element containing the projected points on both the forward and backward planes. The mappings consists of $R * G * 3$ entries per mesh vertex. Additionally the barycentric coordinate of the gyro point projected to the poloidal planes is stored to be used for weighting the contributions. This reduces the gyro averaging operation to a projection of the particle's position to V followed by iterating over the corresponding map entries based on particle properties.

The gyroaverage operation in XGCm is performed in two steps after the particles have been projected to the virtual plane V. The first step accumulates the contributions of every

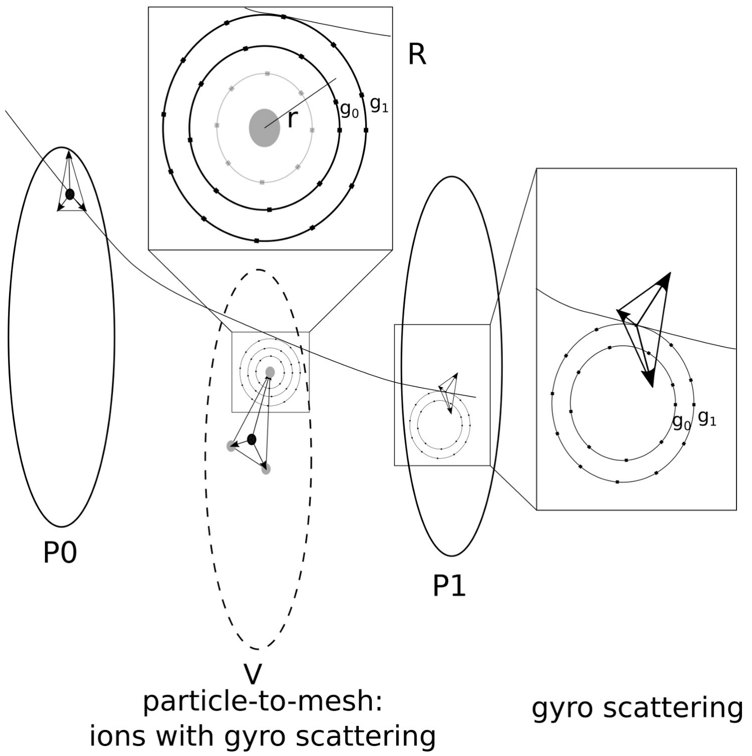

Figure 5.4: Depiction of charge deposition approximation for one particle. Particle on virtual plane V distributes contributions to gyro rings, g_0 and g_1, surrounding mesh vertices of element particle is within. Points along the gyro rings project contributions to poloidal planes P0 and P1 and deposit on mesh vertices.

particle to the rings around each mesh vertex. Each particle contributes to the three vertices that bound its parent element weighted by the barycentric coordinates of its position within the element. The gyro radius of the particle determines the two gyro rings for each vertex that the gyro radius lies between. The second step is to distribute the accumulated values of each gyro ring to the vertices on the forward and backward planes. The value accumulated on the gyro ring is split evenly to each gyro point along the ring. The mappings described above are used to find the three vertices on the poloidal planes for each gyro point. The value distributed from the gyro point to each vertex is weighted by the barycentric coordinates from the mapping. The second step is performed for both the forward and backward planes using the two different maps. Since the poloidal planes are shared across processes, after performing the gyroaverage operation, the fields on each plane must be synchronized across all processes that share the same poloidal plane. The details for this field syncrhonization are described in the next section.

5.1.3 Field Synchronization Between Planes

Given the three dimensions of partitioning, toroidal, mesh, and group, in the XGCm simulation, synchronizing fields across the domain is a more challenging problem. The field synchronization routine in PUMIPic can handle the operation in the mesh direction, but has no concept of the toroidal or group partitions. As such further operations are required to ensure the fields are updated correctly.

Before describing the steps to perform the field synchronization, we must first properly define the requirements. Given M poloidal planes, the PICparts, $\overline{P_0}, ..., \overline{P_{N-1}}$, and G processes per group, the contributions for each field on a poloidal plane attributed to mesh entities of dimension d , $F_0^d, ..., F_{M-1}^d$, must be accumulated and synchronized such that the processes that share the fields associated to a poloidal plane have equal values for each mesh entity in its own PICpart. The key note here is in the toroidal direction, only the processes that share a poloidal plane need to communicate contributions. Towards this we will add an ownership definition to the groups and planes of each process. The forward plane will be denoted as the major plane, while the backward plane will be called the minor plane. One process in each group will be denoted as the group leader, namely the processes with group rank of 0.

The full field synchronization for XGCm is performed in two stages, gather and scatter. The gather stage accumulates the field contributions while the scatter communicates the accumulated field values to the other processes. The gather stage first accumulates the field contributions of each process in a group to its group leader. Then, the each group leader sends its contributions on the minor poloidal plane to the neighboring process's major plane in the toroidal direction. Finally, the group leaders on each major plane use PUMIPic's field synchronization to sum the contributions across the PICparts. At the end of the gather stage the field contributons are accumulated for each PICpart's major plane for group leaders only. The scatter stage sends these accumulated values back to the other processes. First, values on the major planes are sent to the neighboring processes' minor plane for each group leader. Then, each group leader sends its field values to the other processes of the group. Now every process has the accumulated field values for every mesh entitiy in its PICpart.

5.1.4 Field Solve on PICparts

The field solve phase involves the solution of a Poisson equation [150] on each polidal plane. In XGC, this operation is performed by using the PETSc library. In order to use the PETSc solvers for XGCm, the complexity of the PICparts must be properly described to PETSc as it does not support the buffering that exists in PICparts. The simpliest approach to describing the mesh is to only use the core part for each PICpart and ignore the buffering. This approach effectively reduces the mesh back to a traditional partitioning that PETSc can properly understand.

One issue with this approach is as the number of mesh parts increase, the solver will become more and more of a bottleneck as the number of unknowns for each process will be too low to saturate the GPU. Ongoing research is looking at methods to reduce the number of processes that participate in the field solve by using the PICpart buffers. We define this by the set cover problem: Given PICparts, $\overline{P_0}, ..., \overline{P_{N-1}}$, for the mesh partition $P(M) = P_0, ..., P_{N-1}$ choose a subset of the PICparts, $\overline{P}$, such that $\bigcup_{\overline{P_i} \in \overline{P}} \overline{P_i} = M$. It is not necessary to find the minimum or minimal subset, but rather a good number of processes that performs best in the PETSc solver. Determining what defines a good number of processes is left for future work.

5.2 Pseudo Physics Simulation for XGCm

While developing the XGCm simulation, a miniapp was designed to test the scalability of PUMIPic using the existing implementation of specific XGC operations supplemented with pseudo physics for the remaining operations. For this miniapp, the core operations of PUMIPic including adjacency search, particle rebuild, particle migration, and field synchronization are used along with the gyroaverage operation detailed in Section 5.1.2 and a non-physical elliptical push described in Section 5.2.1. The simulation iteratively runs the PIC operations for a given number of iterations. The order of operations for each iteration is described in Algorithm 11. First, the elliptical push moves particles on line 1. Then, the new parent elements for each particle is determined using adjacency search on line 2. Line 3 migrates particles that have either left the safe zone or the toroidal region defined by the current process and performs the rebuild of the particle structure based on the new parent elements and migrated particles. The gyroaverage operation is executed on line 4 to deposit the charge from each particle to the corresponding mesh vertices. The final step synchonizes the field from gyroaveraging across all three partitioning dimensions as detailed in Section 5.1.3.

Algorithm 11 The operations executed in one iteration of the PIC loop

```
EllipticalPush($mesh$, $ptcls$)
$new\_elements$ ← AdjacencySearch($mesh$, $ptcls$)
MigrateAndRebuild($ptcls$, $new\_elements$)
$field$ ← Gyroaverage($mesh$, $ptcls$)
FieldSync($mesh$, $field$)
```

5.2.1 Elliptical Particle Push

The goal of defining a push routine for the pseudo simulation is to approximate the general behavior of particles, without needing to define all of the physical routines required to accurately push particles. The desired behavior is that particles move few elements per iteration and generally move within flux faces. To achieve this, particles are pushed in an elliptical pattern around the magnetic axis. The ellipse that each particle is pushed along is designed such that particles are roughly moving within a flux face. The distance each particle moves per iteration is controlled based on the distance from the center to ensure every particle moves only a short distance. In the toroidal direction, particles are pushed at a constant rate each iteration. This push routine accurately triggers the particle migration and rebuild of the particle data structure, but requires little computation per iteration and does not represent the computational cost of a real simulation's push.

5.2.2 Solve and Field to Particle

The pseudo simulation forgoes both the mesh field solve and field to particle phases. The field to particle phase follows similar calculations and data access patterns as the particle to mesh phase and as such were not implemented for the pseudo simulation. The field solve described in Section 5.1.4 is a newer development than the miniapp and as such was not included in the design of the miniapp. Performance analysis and improvements to the field solve is left for future work as part of the XGCm simulation in development.

5.2.3 Performance Study

Scaling experiments are performed on the Summit supercomputer using a two-million triangle mesh executing 100 iterations of the particle loop. Scaling studies are performed using a constant number of particles per GPU as the number of nodes increases. Each node is configured to use one core per GPU for a total of six MPI ranks and six GPUs per node. For these studies, the SCS structure is used to store the particles. Two sets of scaling studies are performed. First, the number of PICparts that the mesh is partitioned into is increased. This scaling has both weak scaling in terms of particles and strong scaling in terms of mesh entities per process. Then for a constant mesh partition, the number of poloidal planes is increased representing pure weak scaling in terms of particles. For both studies the processes per group is set to one as this feature is not the primary focus of these scaling studies, but potentially useful when simulating a large number of particles for a low number of poloidal planes and mesh parts.

5.2.3.1 Mesh Partition Scaling

The mesh partition scaling study is performed using one poloidal plane with six to 192 PICparts using BFT buffering. This study requires one Summit node for the lower case ranging up to 32 nodes for the 192 PICpart case. The pseudo simulation is executed with up to 48 million particles per GPU (mppg). Figure 5.5 presents the results from the simulation. The left plot shows the total time which is normalized by the single node experiment. Some increase in performance is gained by the partitioning of the mesh which is greater seen for lower particle counts. For four mppg there is up to a 55% decrease in time at 192 PICparts and a 7% decrease at 48 mppg. For larger particle counts, the particle operations dominate the computation and as a result, the scaling of the mesh has less of an effect on the total runtime. The scaling of the mesh also exhibits the expected diminishing performance gains. Around 90% of the reduction in runtime is achieved by 48 PICparts across the particle counts. For the given mesh entity count, partitioning the mesh further does not significantly decrease the PICpart size resulting in a majority of the part being comprised of the buffer. This results in no additional performance gains to the mesh-based operations.

The right plot of Figure 5.5 shows the timing of the operations involved in each time step for the 48-mppg case. The six operations included in the figure are the particle push represented by the push line, the reconstruction of the particle structure labeled rebuild, the adjacency search labeled search, the particle to mesh or charge deposition phase labeled deposition, particle migration labeled migration, and the field synchronization step labeled sync. The predominately mesh-based operations are search and sync. As the number of

PICparts increases, the cost for these operations decreases until flattening out. The rebuild operation is the dominant operation which is expected as it is the most data movement intensive operation. The remaining operations see no significant changes from scaling the mesh and the number of nodes used including particle migration.

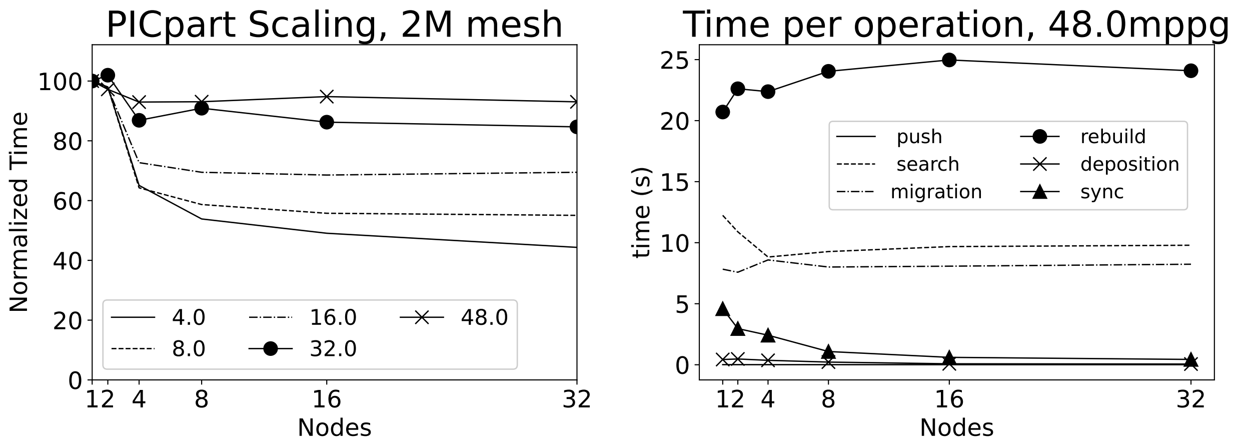

Figure 5.5: Simulation plots scaling PICparts from 6 to 192 with 2 to 48 mppg for 100 iterations of the particle loop. Normalized time (left) and breakdown on the time of the major operations (right).

5.2.3.2 Plane Scaling

The pure weak scaling study is performed by using the 192-PICpart partition with BFT buffering for the two-million element mesh and scaling the number of poloidal planes from one to 128. At 128 poloidal planes, the simulation uses 4096 of the 4608 nodes available on Summit. Results are presented with up to 48 mppg. Results are shown in Figure 5.6. Weak scaling efficiency is presented in the left plot. Scaling up to 256 nodes shows increased time by up to 30% at 4 mppg and 5% at 48 mppg. Scaling from 256 nodes up to 4096 nodes shows only fluctuations in total time with no major increase. The timing of the same major operations shown in the mesh scaling figures for 48 mppg is shown in the right plot. The increase in time scaling up to 256 nodes is caused by the migration and rebuild operations. All other operations show no major increase in time cost.

5.3 Supporting GITRm Implementation

GITR is a PIC code that tracks the motion of impurities in a plasma and the interactions those impurities have with the fields and the wall. The impurities are materials that

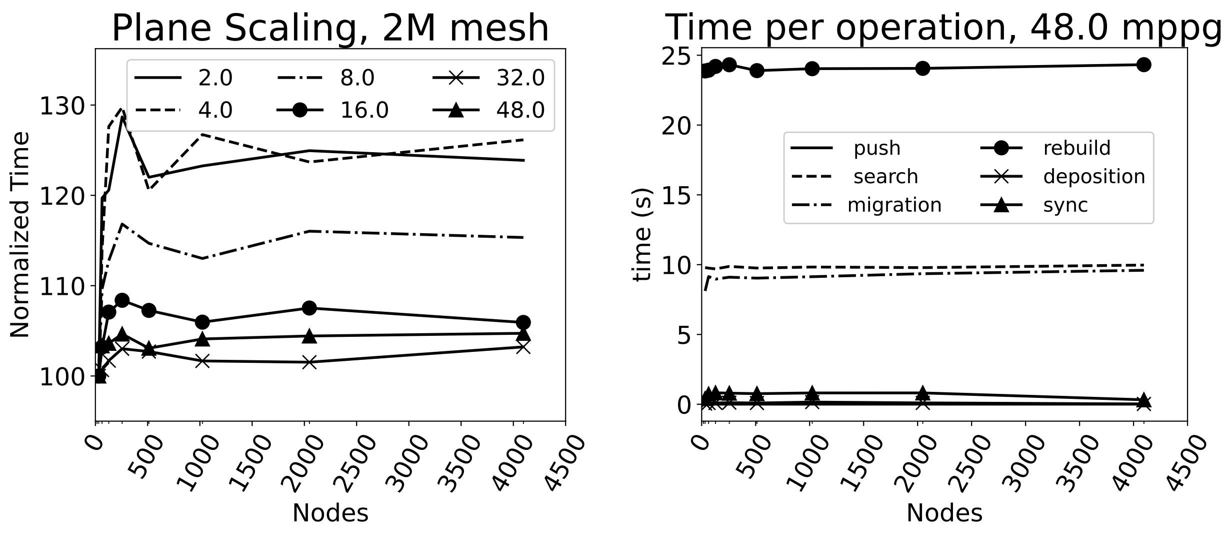

Figure 5.6: Simulation plots scaling planes from 1 to 128 with 2 to 48 mppg for 100 iterations of the particle loop. Normalized time (left) and breakdown on the time of the major operations (right).

are eroded off the plasma-facing components that then move through the domain. GITR's simulation is composed of the particles that represent the impurities, background grids that store the slowly varying quantities such as the magnetic field, temperature, etc., and a surface mesh of the geometry for wall iteractions with the particles. The steps involved in the particle loop are a particle push, mesh-particle interaction checks, and Monte Carlo operations for atomic physics processes. The mesh-particle interactions include the need to check for particle paths intersecting the mesh faces and the computation of fields near the surface boundary.

For GITRm, a major change from the GITR simulation is a full 3D unstructured mesh for representing the tokamak using Omega_h and PUMIPic. The usage of the unstructured mesh results in many operations requiring new implementations to explicitly use the mesh instead of background grids and specialized geometry algorithms. One algorithm change is no longer needing to have an explicit method for computing particle path intersections with the tokamak wall as the adjacency search routine in PUMIPic determines this. In the following sections, we overview some implementation details of the GITRm code and then present results using PUMIPic in a physically accurate simulation with specific focus on the particle load balancing routine.

5.3.1 Particle Initialization

In GITRm, particles are initialized on a model face and then disperse throughout the domain. Figure 5.7 shows the set of faces on the bottom of the tokamak that initialize particles on the left and the paths of particles moving upwards on the right. This initialization of particles is a challenge in the case of a distributed mesh as only the parts that have the mesh faces classified on the set of mesh faces begin with particles. This problem is a greater issue in GITRm because particles are created in PICparts where the mesh elements are part of the core part. This restricts the number of particles that can be created at the beginning to the amount of memory available on the PICparts with the mesh faces in the core part.

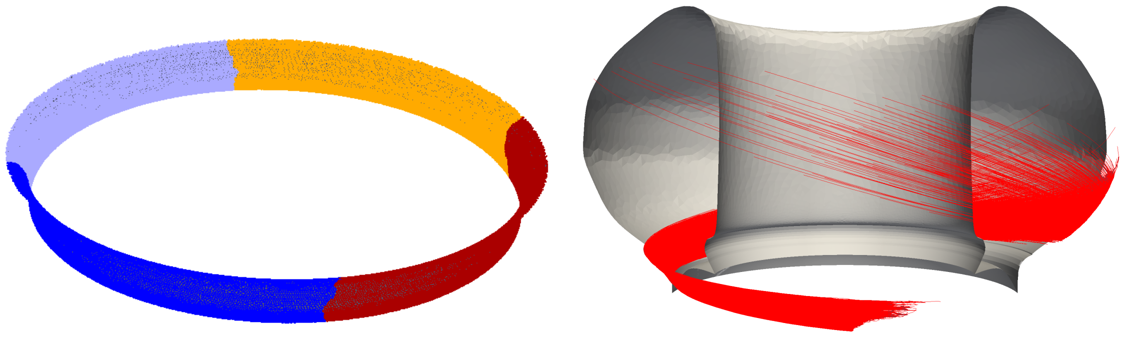

Figure 5.7: Left: The mesh entities where particles are initialized in the GITRm simulation. Right: The path for some particles in GITRm.

There are a few methods to alleviate this issue to allow for larger number of particles. The first step is to utilize PUMIPic's load balancing procedure described in Section 4.3 to the initial partition of particles prior to the allocation of particle memory. In this case, an array of particles per element per PICpart is taken as input to the load balancer that then redistributes the counts in the array to lighter PICparts. This allows GITRm to initialize more particles then the PICparts on the model boundary can store that will be migrated to other PICparts before the particle data is allocated to memory.

There are two other planned methods that have not been implemented to further increase the amount of particles that GITRm can simulate. The first is only applicable when the background fields do not change as a function of time during the simulation. In this case, particles can be initialized at any time throughtout the simulation. So, it is possible to start with a subset of the total number of particles, and then add the remaining particles once the initial set has moved further into the domain. The other approach, which can be applied in all cases including time dependent background fields, is to utilize the group mechanism from the XGCm simulation described in Section 5.1.1. For GITRm, the idea of groups will allow multiple processes to store the PICparts with owned faces classified on the model boundary. Each of the processes in the group can allocate particles that effectively increases the number of particles that can be initialized by the number of processes in the groups. These two approaches are left for future work as the usage of PUMIPic's load balancer is sufficient for the current scale of simulations.

After the particles are initialized, the distribution of particles is expected to change greatly as the particles move into the rest of the domain. To maintain a good balance of particles, PUMIPic's load balancing routine is used throughout the simulation. Results in Section 5.4 compare executing GITRm with and without the load balancing routine for increasing number of PICparts and particle counts.

5.3.2 Distance to Boundary

In GITR, particles that are close to the walls have additional physical terms that affect the forces applied to the particle [44]. This operation requires computing the closest point on the surface mesh. Computing this for every mesh face would be extremely inefficient so approaches to limit the search space must be employed. GITR precomputes a background grid where each grid cell stores the set of mesh faces to check. This precomputation step is expensive when a large 3D geometry is used as each cell in the structured background grid needs to determine the set of mesh faces that are closest to any point within the grid cell. The key behind using the background grid is that it only needs to be computed once for a given mesh.

For GITRm, since the full 3D domain is discretized with an unstructured mesh, a similar approach can be used to perform the distance to boundary calculation without building an additional background grid. Similar to the GITR approach, a pre-computing step is performed once per mesh, but in the GITRm case the set of closest faces on the model boundary is computed for each mesh element. When determining the faces to check for the distance to boundary calculation, each particle queries the set of faces from its parent element.

5.4 GITRm: Studying Large Particle Imbalance

In this section, we present performance analysis of using PUMIPic's particle load balancing method in GITRm. Experiments are performed on an 11.4 million element mesh partitioned from six to 48 PICparts. Each partition is executed with an average of two million particles per part up to 32 million particles per PICpart. The largest case of 48 PICparts with 450 million particles. Each setup performs 10,000 iterations of the GITRm particle loop. We compare running with load balancing targeting a 1.05 imbalance of particles performed every 100 iterations to a equivalent run without load balancing. The experiments are run on the AiMOS supercomputer with six NVIDIA Tesla V100 GPUs per node using one MPI process per GPU.

Figure 5.8 shows an array of plots of the particle imbalance as a function of iteration for each partition with 50 million particles and then a case with 150-400 million particles based on the number of GPUs used. At the beginning of the simulation the large imbalances are significantly reduced. For example at 48 GPUs with 400 miliion particles, the initial imbalance of 3.04 is reduced down to 1.34. For the smaller partitions, the particle imbalance is reduced to the tolerance of 1.05 during the initial repartitioning of particles. For the 24 and 48 GPU cases, once particles have moved far enough into the domain, the imbalance of particles is also reduced to the 1.05 tolerance. This occurs within the first few calls to the load balancing method. In contrast, when the load balancing method is disabled, the particle imbalance initally gets worse then gets slightly better when the forced migrations begin to occur. In all cases without load balancing the imbalance never goes below the imbalance at the beginning. For these cases, the application of load balancing achieved up to a 20% reduction in simulation time compared to the runs without load balancing. Studying how to gain further improvements to the simulation time as a result of the reduction in particle imbalance is ongoing research. We expect the improvements to simulation time are currently limited by the particle structure not properly reconfiguring as the particle distribution changes so drastically across iterations.

5.5 Summary

In this chapter we detailed the on-going implementations of plasma-physics particle-in-cell simulations using the PUMIPic library for the XGC and GITR codes. For XGC, a brief overview of important operations were discussed and the research towards solving

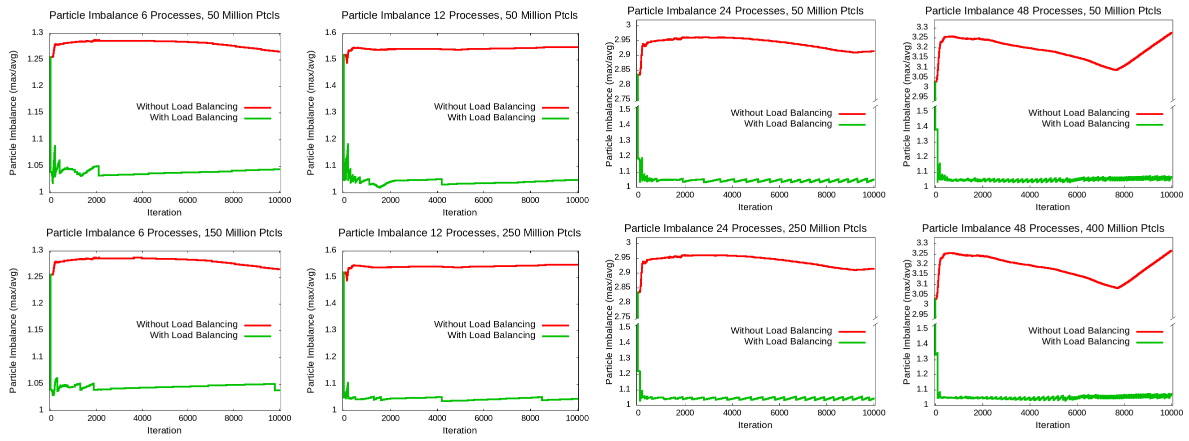

Figure 5.8: Particle imbalance across ten thousand iterations running GITRm with and without load balancing for 50 million particles up to 400 million particles on 6 to 48 GPUs. Lower is better.

the partitioning challenges involved with XGC's discretization of the domain were detailed. Scaling results using a pseudo-physics miniapp showed good scaling in terms of PICparts and poloidal planes up to 4096 nodes on the Summit supercomputer. For GITR, the challenge of the non-uniform particle distribution was introduced that features large particle imbalances in the distributed mesh case. Results for applying PUMIPic's load balancing were presented for up to 96 GPUs with over 3 billion particles.

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

6.1 Conclusion

This thesis presented research towards improving science and engineering simulations running on massively parallel heterogenous supercomputers in the scope of interprocess parallelism and intraprocess parallelism. A partition improvement method previously applied on element partitioned meshes was generalized to a multihypergraph structure. This provided the ability to address a broader range of applications with additional improvements to the runtime of the load balancing algorithm. New techniques for managing the edge cut for graphs with high degree vertices was introduced. The new generalized partition improvement library was applied to a range of unstructured mesh setups scaling up to half a million parts of a one billion element mesh. The developed multihypergraph methods was also applied to maintain particle load balance in PIC calculations.

A new mesh-based approach to particle-in-cell simulations was presented that uses the unstructured mesh as the primary data structure and stores particles based on the unstructured mesh elements. This approach supports the distribution of both the unstructured mesh and particles allowing particle-in-cell codes to scale to larger graded unstructured meshes. The library designed based on this new approach included specialized data structures for the unstructured mesh and particles to provide performance portablility across the current and future generations of heterogenous supercomputers. Scaling for this library is presented up to 4096 nodes of the Summit supercomputer simulating over one trillion particles. Included in the library is a fast dynamic particle load balancing method that exploits the distributed unstructured mesh data structure and the previously mentioned partition improvement method. This method is shown to improve the partition of particles on a non-uniform distribution and maintain good balance throughout the simulation. Finally, ongoing research towards the implementations of two plasma physics particle-in-cell codes using the mesh-based library was presented. In the case of the newly developed GITRm code, its development has advanced to the point that DOE researchers are starting to apply it to address physics simulations the original GITR code was not able to effectively address.

6.2 Future Work

6.2.1 Partition Improvement on GPUs

To provide faster methods for partition improvement it is important that EnGPar's algorithms be ported to work on GPU systems. Some developments have been made to port the more complex algorithms to the GPU [159]. The asynchronous nature of the GPU acceleration introduces a major challenge for EnGPar as selection decisions must be made without full knowledge of other concurrent decisions. This introduces a trade-off between acceleration of the method and quality of the partition achieved by EnGPar. Properly balancing user control of this trade-off and automatic decisions are key interests in developing optimal methods for different applications.

6.2.2 Optimized Use of Particle Structures

Determining the best particle data structure to apply is dependent on the amount of particles, the distribution across elements, and the extra memory available to use for improved performance. There are two directions for optimizing the particle structure for a given application. First is determining the optimal set of parameters to achieve the best performance for a structure given the particle distribution. Second is automatically making the decision of which particle structure to use and what parameters to set as the particle simulation evolves including switching to a different data layout when necessary.

6.2.3 Evolving Mesh in PICparts

One limitation of the current definition of the PUMIPic mesh partition is that it operates on a static mesh. As the simulations evolve, it is desirable to use mesh adaptation and repartitioning of the mesh entities to track regions that require increased granulation. Supporting both mesh adaptation and mesh repartitioning will require improvements to the storage of the mesh to allow for these changes across PICparts as efficiently as possible.

REFERENCES

[1] T. J. Hughes, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Mineola, NY, USA: Courier Dover Publications, 2012.

[2] T. Sonar, "Chapter 3 - classical finite volume methods," in *Handbook of Numerical Methods for Hyperbolic Problems - Basic and Fundamental Issues*. Braunschweig, Germany: Elsevier, 2016, pp. 55–76. DOI: `10.1016/bs.hna.2016.09.005`.

[3] Y. Grigoryev, V. Vshivkov, and M. Fedoruk, "Chapter 2 - particle-in-cell methods on unstructured meshes," in *Numerical Paritcle-in-Cell Methods Theory and Applications*. Utrecht, Netherlands: VSP, 2002, pp. 56–84.

[4] K. Schloegel, G. Karypis, and V. Kumar, "Graph partitioning for high-performance scientific simulations," in *Sourcebook of Parallel Computing*, J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, Eds. San Francisco, CA, USA: Morgan Kaufmann Inc., 2003, pp. 491–541.

[5] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage and Anal. (SC)*, Nov. 1998, pp. 1–13.

[6] K. Schloegel, G. Karypis, and V. Kumar, "Parallel static and dynamic multi-constraint graph partitioning," *Concurr. Comput.*, vol. 14, no. 3, pp. 219–240, Mar. 2002.

[7] U. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar, "UMPa: A multiobjective, multi-level partitioner for communication minimization," *Contemporary Math.*, vol. 588, no. 1, pp. 53–64, Feb. 2013.

[8] D. LaSalle and G. Karypis, "Multi-threaded graph partitioning," in *Proc. 27th IEEE Int. Parallel and Distrib. Process. Symp. (IPDPS)*, May 2013, pp. 225–236.

[9] G. Karypis and V. Kumar, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *Siam Rev.*, vol. 41, no. 2, pp. 278–300, Jun. 1999.

[10] D. F. Harlacher, H. Klimach, S. Roller, C. Siebert, and F. Wolf, "Dynamic load balancing for unstructured meshes on space-filling curves," in *26th Int. Parallel and Distrib. Process. Symp. Workshops Ph.D. Forum*, May 2012, pp. 1661–1669. DOI: `10.1109/IPDPSW.2012.207`.

[11] M. Zhou, O. Sahni, K. Devine, M. Shephard, and K. Jansen, "Controlling unstructured mesh partitions for massively parallel simulations," *SIAM J. Scientific Comput.*, vol. 32, no. 6, pp. 3201–3227, Nov. 2010.

[12] M. Zhou, O. Sahni, T. Xie, M. S. Shephard, and K. E. Jansen, "Unstructured mesh partition improvement for implicit finite element at extreme scale," *J. Supercomputing*, vol. 59, no. 3, pp. 1218–1228, Dec. 2012.

[13] C. W. Smith, M. Rasquin, D. Ibanez, K. E. Jansen, and M. S. Shephard, "Improving unstructured mesh partitions for multiple criteria using mesh adjacencies," *SIAM J. Scientific Comput.*, vol. 40, no. 1, pp. C47–C75, Feb. 2018. DOI: `10.1137/15M1027814`.

[14] X. Li, M. S. Shephard, and M. W. Beall, "3D anisotropic mesh adaptation by mesh modification," *Comput. Methods in Appl. Mech. and Eng.*, vol. 194, no. 48-49, pp. 4915–4950, Nov. 2005. DOI: `http://dx.doi.org/10.1016/j.cma.2004.11.019`.

[15] F. Alauzet, X. Li, E. S. Seol, and M. S. Shephard, "Parallel anisotropic 3D mesh adaptation by mesh modification," *Eng. with Comput.*, vol. 21, no. 3, pp. 247–258, May 2006. DOI: `10.1007/s00366-005-0009-3`.

[16] K. E. Jansen, C. H. Whiting, and G. M. Hulbert, "A generalized-α method for integrating the filtered Navier-Stokes equations with a stabilized finite element method," *Comput. Methods in Appl. Mech. and Eng.*, vol. 190, no. 3-4, pp. 305–319, Oct. 2000. DOI: `10.1016/S0045-7825(00)00203-6`.

[17] J. Fingberg, A. Basermann, G. Lonsdale, J. Clinckemaillie, J.-M. Gratien, and R. Ducloux, "Dynamic load balancing for parallel structural mechanics simulations with DRAMA," in *Developments in Engineering Computational Technology.* Edinburgh, UK: Civil-Comp Press, 2000, pp. 199–205.

[18] P. H. Worley, E. D'Azevedo, R. Hager, S.-H. Ku, E. Yoon, and C. Chang, "Balancing particle and mesh computation in a particle-in-cell code," in *Proc. Cray Users Group Meeting*, May 2016, pp. 1–10.

[19] S. J. Plimpton, D. B. Seidel, M. F. Pasik, R. S. Coats, and G. R. Montry, "A load-balancing algorithm for a parallel electromagnetic particle-in-cell code," *Comput. Phys. Commun.*, vol. 152, no. 3, pp. 227–241, May 2003. DOI: `10.1016/S0010-4655(02)00795-6`.

[20] E. A. Carmona and L. J. Chandler, "On parallel PIC versatility and the structure of parallel PIC approaches," *Concurr.: Practice and Experience*, vol. 9, no. 12, pp. 1377–1405, Dec. 1997.

[21] K. Madduri, K. Ibrahim, S. Williams, E.-J. Im, S. Ethier, J. Shalf, and L. Oliker, "Gyrokinetic toroidal simulations on leading multi-and manycore HPC systems," in *Proc. 2011 Int. Conf. for High Perfom. Comput., Netw., Storage and Anal.*, 2011, p. 23.

[22] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel and Distrib. Comput.*, vol. 7, no. 2, pp. 279–301, Oct. 1989.

[23] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 4, no. 9, pp. 979–993, Sep. 1993.

[24] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present, and future," *Concurr. and Comput.*, vol. 15, no. 9, pp. 803–820, Jul. 2003.

[25] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. (). "November 2020." top500.org. `https://www.top500.org/lists/top500/2020/11/` (accessed Jun. 17, 2021).

[26] Oak Ridge National Laboratory. (). "Summit - Oak Ridge National Laboratory." olcf.ornl.gov. `https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/` (accessed Jun. 18, 2021).

[27] J. Hines, "Stepping up to summit," *Comput. Sci. Eng.*, vol. 20, no. 2, pp. 78–82, Mar. 2018. DOI: `10.1109/MCSE.2018.021651341`.

[28] V. G. Vergara Larrea, W. Joubert, M. J. Brim, R. D. Budiardja, D. Maxwell, M. Ezell, C. Zimmer, S. Boehm, W. Elwasif, S. Oral, C. Fuson, D. Pelfrey, O. Hernandez, D. Leverman, J. Hanley, M. Berrill, and A. Tharrington, "Scaling the summit: Deploying the world's fastest supercomputer," in *Proc. Int. Conf. High Perfom. Comput.*, vol. 11887, Dec. 2019, pp. 330–351.

[29] Lawrence Livermore National Laboratory. (). "Sierra — High Performance Computing." hpc.llnl.gov. `https://hpc.llnl.gov/hardware/platforms/sierra` (accessed Jun. 18, 2021).

[30] J. P. Dahm, D. F. Richards, A. Black, A. D. Bertsch, L. Grinberg, I. Karlin, S. Kokkila-Schumacher, E. A. León, J. R. Neely, R. Pankajakshan, and O. Pearce, "Sierra center of excellence: Lessons learned," *IBM J. Res. and Develop.*, vol. 64, no. 3/4, pp. 2:1–2:14, May 2020. DOI: 10.1147/JRD.2019.2961069.

[31] C. W. Smith, G. Diamond, and M. S. Shephard, "Load Balancing on Many-Core and Accelerated Systems," presented at Salishan Conf. High Speed Comput., Gleneden Beach, OR, USA, Apr. 25, 2019.

[32] National Energy Research Scientific Computing Center. (). "Perlmutter." nersc.gov. https://www.nersc.gov/systems/perlmutter/ (accessed Jun. 18, 2021).

[33] Argonne Leadership Computing Facility. (). "Aurora — Argonne Leadership Computing Facility." alcf.anl.gov. https://www.alcf.anl.gov/aurora/ (accessed Jun. 18, 2021).

[34] Oak Ridge Leadership Computing Facility. (). "Frontier supercomputer - Oak Ridge Leadership Computing Facility." olcf.ornl.gov. https://www.olcf.ornl.gov/frontier/ (accessed Jun. 18, 2021).

[35] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. of Parallel and Distrib. Comput.*, vol. 74, no. 12, pp. 3202–3216, Aug. 2014. DOI: 10.1016/j.jpdc.2014.07.003.

[36] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM Int. Workshop Perfom., Portability and Productiv. in HPC (P3HPC)*, Mar. 2019, pp. 71–81. DOI: 10.1109/P3HPC49587.2019.00012.

[37] W. Hoenig, F. Schmitt, R. Widera, H. Burau, G. Juckeland, M. Müller, M. Bussmann, and M. Muller, "A generic approach for developing highly scalable particle-mesh codes for GPUs," in *2010 Symp. on App. Accelerators High Perf. Comput.*, Jul. 2010, pp. 1–3.

[38] H. Burau, R. Widera, W. Hönig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T. E. Cowan, R. Sauerbrey, and M. Bussmann, "PIConGPU: A fully relativistic particle-in-cell code for a GPU cluster," *IEEE Trans. Plasma Sci.*, vol. 38, no. 10, pp. 2831–2839, Aug. 2010.

[39] V. K. Decyk and T. V. Singh, "Particle-in-cell algorithms for emerging computer architectures," *Comput. Phys. Commun.*, vol. 185, no. 3, pp. 708–719, Mar. 2014. DOI: `https://doi.org/10.1016/j.cpc.2013.10.013`.

[40] A. Spirkin and N. A. Gatsonis, "Unstructured 3D PIC simulations of the flow in a retarding potential analyzer," *Comput. Phys. Commun.*, vol. 164, no. 1, pp. 383–389, Dec. 2004. DOI: `10.1016/j.cpc.2004.06.051`.

[41] N. A. Gatsonis and A. Spirkin, "A three-dimensional electrostatic particle-in-cell methodology on unstructured delaunay–voronoi grids," *J. of Comput. Phys.*, vol. 228, no. 10, pp. 3742–3761, Aug. 2009. DOI: `10.1016/j.jcp.2009.02.003`.

[42] E. DÁzevedo, S. Abbott, T. Koskela, P. Worley, S.-H. Ku, S. Ethier, E. Yoon, M. Shephard, R. Hager, J. Lang, J. Choi, N. Podhorszki, S. Klasky, M. Parashar, and C.-S. Chang, "The fusion code XGC: Enabling kinetic study of multi-scale edge turbulent transport in ITER," in *Exascale Scientific Applications: Scalability and Performance Portability*, T. Straatsma, K. Antypas, and T. Williams, Eds., London, England: CRC Press, Taylor & Francis Group, 2017, pp. 529–551.

[43] B. Wang, S. Ethier, W. Tang, K. Ibrahim, K. Madduri, S. Williams, and L. Oliker, "Modern gyrokinetic particle-in-cell simulation of fusion plasmas on top supercomputers," *Int. J. of High Perfom. Comput. Appl.*, vol. 33, no. 1, pp. 169–188, Jun. 2019. DOI: `10.1177/1094342017712059`.

[44] T. Younkin, D. Green, A. Simpson, and B. Wirth, "GITR: An accelerated global scale particle tracking code for wall material erosion and redistribution in fusion relevant plasma–material interactions," *Comput. Phys. Commun.*, vol. 264, pp. 1–11, Jul. 2021. DOI: `10.1016/j.cpc.2021.107885`.

[45] C. W. Smith, "Improving scalability of parallel unstructured mesh-based adaptive workflows," Ph.D. dissertation, Dept. Comput. Sci., Rensselaer Polytech. Inst., Troy, NY, USA, 2017.

[46] D. A. Ibanez, “Conformal mesh adaptation on heterogeneous supercomputers,” Ph.D. dissertation, Dept. Comput. Sci., Rensselaer Polytech. Inst., Troy, NY, USA, 2016.

[47] *Omega_h GitHub repository.* (2016). Sandia National Laboratories. Accessed: Aug. 10, 2021. `https://github.com/SNLComputation/omega_h`.

[48] G. Diamond, C. W. Smith, and M. S. Shephard, “Dynamic load balancing of massively parallel unstructured meshes,” in *Proc. 8th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst.*, Denver, CO, USA, Nov. 2017, pp. 9:1–9:7. DOI: `10.1145/3148226.3148236`.

[49] G. Diamond, C. W. Smith, E. Yoon, and M. S. Shephard, “Dynamic load balancing of plasma and flow simulations.,” in *Proc. 8th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst.*, Dallas, TX, USA, Nov. 2018, pp. 73–80. DOI: `10.1109/ScalA.2018.00013`.

[50] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM J. Scientific Comput.*, vol. 20, no. 1, pp. 359–392, Jul. 1998. DOI: `10.1137/S1064827595287997`.

[51] U. V. Çatalyürek and C. Aykanat, “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication,” *IEEE Trans. Parallel and Distrib. Syst.*, vol. 10, no. 7, pp. 673–693, Jul. 1999.

[52] U. V. Çatalyürek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen, “A repartitioning hypergraph model for dynamic load balancing,” *J. Parallel and Distrib. Comput.*, vol. 69, no. 8, pp. 711–724, Aug. 2009.

[53] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Çatalyürek, “Parallel hypergraph partitioning for scientific computing,” in *Proc. Int. Parallel and Distrib. Process. Symp.*, Apr. 2006, pp. 1–10.

[54] F. Boesch and R. Tindell, “Robbins's theorem for mixed multigraphs,” *The Amer. Math. Monthly*, vol. 87, no. 9, pp. 716–719, Nov. 1980.

[55] C. Chevalier and F. Pellegrini, “PT-Scotch: A tool for efficient parallel graph ordering,” *Parallel Comput.*, vol. 34, no. 6, pp. 318–331, Jul. 2008. DOI: `10.1016/j.parco.2007.12.001`.

[56] B. Hendrickson and R. Leland, *The Chaco User's Guide*, 2nd ed., (1995). [Online]. Available: `https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/guide.pdf`.

[57] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan data management services for parallel dynamic applications," *Comput. in Sci. Eng.*, vol. 4, no. 2, pp. 90–96, Mar. 2002. DOI: `10.1109/5992.988653`.

[58] D. A. Ibanez, E. S. Seol, C. W. Smith, and M. S. Shephard, "PUMI: Parallel unstructured mesh infrastructure," *ACM Trans. Math. Softw.*, vol. 42, no. 3, pp. 17:1–17:28, May 2016. DOI: `10.1145/2814935`.

[59] R. S. Pienta and R. M. Fujimoto, "On the parallel simulation of scale-free networks," in *Proc. 2013 ACM SIGSIM Conf. Princ. Adv. Discrete Simul.*, May 2013, pp. 179–188.

[60] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *10th USENIX Symp. Operating Syst. Des. and Implementation*, Oct. 2012, pp. 12–30.

[61] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Achieving high sustained performance in an unstructured mesh CFD application," in *Proc. Int. Conf. High Perfom. Comput., Netw., Storage and Anal.*, Nov. 1999, pp. 1–13. DOI: `10.1145/331532.331600`.

[62] B. Hendrickson and K. Devine, "Dynamic load balancing in computational mechanics," *Comput. Methods in Appl. Mech. and Eng.*, vol. 184, no. 2, pp. 485–500, Apr. 2000.

[63] M. J. Berger and S. H. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *Comput., IEEE Trans.*, vol. 100, no. 5, pp. 570–580, May 1987.

[64] V. Taylor and B. Nour-omid, "A study of the factorization fill-in for a parallel implementation of the finite element method," *Int. J. Numer. Meth. Eng.*, vol. 37, no. 22, pp. 3809–3823, Nov. 1994.

[65] R. D. Williams, "Performance of dynamic load balancing algorithms for unstructured mesh calculations," *Concurr.: Pract. Exper.*, vol. 3, no. 5, pp. 457–481, Oct. 1991. DOI: `10.1002/cpe.4330030502`.

[66] S. Schamberger and J.-M. Wierum, "Partitioning finite element meshes using space-filling curves," *Future Gener. Comput. Syst.*, vol. 21, no. 5, pp. 759–766, May 2005. DOI: 10.1016/j.future.2004.05.018.

[67] J. Skilling, "Programming the hilbert curve," in *23rd Int. Workshop Bayesian Inference and Maximum Entropy Methods Sci. and Eng.*, vol. 707, Aug. 2004, pp. 381–387. DOI: 10.1063/1.1751381.

[68] Y. F. Hu, R. J. Blake, and D. R. Emerson, "An optimal migration algorithm for dynamic load balancing," *Concurr.: Pract. and Exper.*, vol. 10, no. 6, pp. 467–483, May 1998.

[69] Y. Hu and R. Blake, "An improved diffusion algorithm for dynamic load balancing," *Parallel Comput.*, vol. 25, no. 4, pp. 417–444, Apr. 1999.

[70] C.-W. Ou and S. Ranka, "Parallel incremental graph partitioning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 8, pp. 884–896, Aug. 1997. DOI: 10.1109/71.605773.

[71] H. Meyerhenke, B. Monien, and S. Schamberger, "Graph partitioning and disturbed diffusion," *Parallel Comput.*, vol. 35, no. 10, pp. 544–569, Oct. 2009.

[72] R. Subramanian and I. D. Scherson, "An analysis of diffusive load-balancing," in *Proc. 6th Annu. ACM Symp. Parallel Algorithms and Architectures*, Jun. 1994, pp. 220–225.

[73] C. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in *19th Des. Automat. Conf.*, Jun. 1982, pp. 175–181.

[74] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, Apr. 1970.

[75] G. M. Slota, K. Madduri, and S. Rajamanickam, "Complex network partitioning using label propagation," *SIAM J. on Sci. Comput.*, vol. 38, no. 5, pp. S620–S645, May 2016. DOI: 10.1137/15M1026183.

[76] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *Proc. Int. Parallel and Distrib. Process. Symp.*, May 2017, pp. 646–655.

[77] J. Bang-Jensen and G. Gutin, “Alternating cycles and paths in edge-coloured multigraphs: A survey,” *Discrete Math.*, vol. 165, no. 1, pp. 39–60, Mar. 1997. DOI: 10.1016/S0012-365X(96)00160-4.

[78] M. W. Beall and M. S. Shephard, “A general topology-based mesh data structure,” *Int. J. Numer. Methods in Eng.*, vol. 40, no. 9, pp. 1573–1596, May 1997. DOI: 10.1002/(SICI)1097-0207(19970515)40:9<1573::AID-NME128>3.0.CO;2-9.

[79] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout.* Chirchester, U.K.: Willey–Teubner, 1990. DOI: 10.1007/978-3-322-92106-2.

[80] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw, “Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM,” *Parallel Comput.*, vol. 26, no. 12, pp. 1555–1581, Nov. 2000.

[81] H. Meyerhenke and S. Schamberger, “Balancing parallel adaptive FEM computations by solving systems of linear equations,” in *Proc. Euro-Par Parallel Process.*, Aug. 2005, pp. 209–219. DOI: 10.1007/11549468_26.

[82] B. A. Galler and M. J. Fisher, “An improved equivalence algorithm,” *Commun. ACM*, vol. 7, no. 5, pp. 301–303, May 1964. DOI: 10.1145/364099.364331.

[83] Z. Galil and G. F. Italiano, “Data structures and algorithms for disjoint set union problems,” *ACM Comput. Surv.*, vol. 23, no. 3, pp. 319–344, Sep. 1991. DOI: 10.1145/116873.116878.

[84] K. C. Chitale, M. Rasquin, O. Sahni, M. S. Shephard, and K. E. Jansen, “Anisotropic boundary layer adaptivity of multi-element wings,” in *Proc. 52nd Aerosp. Sci. Meeting*, vol. 117, Jan. 2014, pp. 1–14. DOI: 10.2514/6.2014-0117.

[85] O. Sahni, J. Müller, K. E. Jansen, M. S. Shephard, and C. A. Taylor, “Efficient anisotropic adaptive discretization of cardiovascular system,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 195, no. 41-43, pp. 5634–5655, Aug. 2006. DOI: 10.1016/j.cma.2005.10.018.

[86] E. Ramm, E. Rank, R. Rannacher, K. Schweizerhof, E. Stein, W. Wendland, G. Wittum, P. Wriggers, and W. Wunderlich, *Error-controlled Adaptive Finite Elements in Solid Mechanics.* West Sussex, England: John Wiley & Sons, 2003.

[87] K. C. Chitale, O. Sahni, M. S. Shephard, S. Tendulkar, and K. E. Jansen, "Anisotropic adaptation for transonic flows with turbulent boundary layers," *AIAA J.*, vol. 53, no. 2, pp. 367–378, Feb. 2014. DOI: 10.2514/1.J053159.

[88] O. Sahni, K. E. Jansen, M. S. Shephard, C. A. Taylor, and M. W. Beall, "Adaptive boundary layer meshing for viscous flow simulations," *Eng. Comput.*, vol. 24, no. 3, pp. 267–285, Sep. 2008. DOI: 10.1007/s00366-008-0095-0.

[89] A. Loseille and R. Löhner, "On 3D anisotropic local remeshing for surface, volume and boundary layers," in *Proc. 18th Int. Meshing Roundtable*, Oct. 2009, pp. 611–630.

[90] V. A. Dobrev, T. V. Kolev, and R. N. Rieben, "High-order curvilinear finite element methods for lagrangian hydrodynamics," *SIAM J. on Sci. Comput.*, vol. 34, no. 5, pp. B606–B641, Jul. 2012. DOI: 10.1137/120864672.

[91] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini, "MFEM: A modular finite element methods library," *Comput. and Math. Appl.*, vol. 81, pp. 42–74, Jan. 2021. DOI: 10.1016/j.camwa.2020.06.009.

[92] G. Legrain, N. Chevaugeon, and K. Dréau, "High order X-FEM and levelsets for complex microstructures: Uncoupling geometry and approximation," *Comput. Methods Appl. Mech. and Eng.*, vol. 241-244, pp. 172–189, Oct. 2012. DOI: 10.1016/j.cma.2012.06.001.

[93] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, Mar. 2012. DOI: 10.1109/MM.2011.108.

[94] P. Wesseling and C. Oosterlee, "Geometric multigrid with applications to computational fluid dynamics," *J. of Comput. and Appl. Math.*, vol. 128, no. 1, pp. 311–334, Mar. 2001. DOI: 10.1016/S0377-0427(00)00517-3.

[95] G. Diamond, C. W. Smith, C. Zhang, E. Yoon, and M. S. Shephard, "PUMIPic: A mesh-based approach to unstructured mesh particle-in-cell on GPUs," *J. of Parallel*

and Distrib. Comput., vol. 157, pp. 1–12, Nov. 2021. DOI: 10.1016/j.jpdc.2021.06.004.

[96] R. Khaziev and D. Curreli, "hPIC: A scalable electrostatic particle-in-cell for plasma–material interactions," *Comput. Phys. Commun.*, vol. 229, pp. 87–98, Aug. 2018. DOI: 10.1016/j.cpc.2018.03.028.

[97] W. Tang and Z. Lin, "Global gyrokinetic particle-in-cell simulation," in *Exascale Scientific Applications: Scalability and Performance Portability*, T. Straatsma, K. Antypas, and T. Williams, Eds., London, England: CRC Press, Taylor & Francis Group, 2017, ch. 23, pp. 507–528.

[98] K. Ko, A. Candel, L. Ge, A. Kabel, R. Lee, Z. Li, C. Ng, V. Rawat, G. Schussman, and L. Xiao, "Advances in parallel electromagnetic codes for accelerator science and development," SLAC National Accelerator Laboratory, Menlo Park, CA, USA, Tech. Rep., 2011.

[99] C. Birdsall and A. Langdon, *Plasma Physics via Computer Simulation*, 1st ed. Boca Raton, FL, USA: CRC Press, 2004.

[100] J. Neudorfer, A. Stock, R. Schneider, S. Roller, and C. Munz, "Efficient parallelization of a three-dimensional high-order particle-in-cell method for the simulation of a 170 GHz gyrotron resonator," *IEEE Trans. Plasma Sci.*, vol. 41, no. 1, pp. 87–98, Jan. 2013.

[101] J. Qiang and X. Li, "Particle-field decomposition and domain decomposition in parallel particle-in-cell beam dynamics simulation," *Comput. Phys. Commun.*, vol. 181, no. 12, pp. 2024–2034, Dec. 2010.

[102] H. Vincenti, M. Lobet, R. Lehe, L.-L. Vay, and J. Deslippe, "PIC codes on the road to exascale architectures," in *Exascale Scientific Applications: Scalability and Performance Portability*, T. Straatsma, K. Antypas, and T. Williams, Eds., London, England: CRC Press, Taylor & Francis Group, 2017, pp. 375–407.

[103] T. D. Arber, K. Bennett, C. S. Brady, A. Lawrence-Douglas, M. G. Ramsay, N. J. Sircombe, P. Gillies, R. G. Evans, H. Schmitz, A. R. Bell, and C. P. Ridgers, "Contemporary particle-in-cell approach to laser-plasma modelling," *Plasma Phys. and*

Controlled Fusion, vol. 57, no. 11, pp. 1–26, Sep. 2015. DOI: 10.1088/0741-3335/57/11/113001.

[104] J. Blahovec, L. Bowers, J. Luginsland, G. Sasser, and J. Watrous, "3-D ICEPIC simulations of the relativistic klystron oscillator," *IEEE Trans. Plasma Sci.*, vol. 28, no. 3, pp. 821–829, Jun. 2000. DOI: 10.1109/27.887733.

[105] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner," in *Proc. 2008 ACM/IEEE Conf. Supercomputing*, Nov. 2008, pp. 1–11. DOI: 10.1109/SC.2008.5222734.

[106] R. A. Fonseca, L. O. Silva, F. S. Tsung, V. K. Decyk, W. Lu, C. Ren, W. B. Mori, S. Deng, S. Lee, T. Katsouleas, and J. C. Adam, "OSIRIS: A three-dimensional, fully relativistic particle in cell code for modeling plasma based accelerators," in *Computational Science - ICCS*, P. Sloot, A. Hoekstra, C. Tan, and J. Dongarra, Eds. Berlin, Heidelberg: Springer, 2002. DOI: 10.1007/3-540-47789-6_36.

[107] K. Germaschewski, W. Fox, S. Abbott, N. Ahmadi, K. Maynard, L. Wang, H. Ruhl, and A. Bhattacharjee, "The plasma simulation code: A modern particle-in-cell code with load-balancing and GPU support," 2015, *arXiv*: 1310.7866.

[108] J.-L. Vay, P. Colella, J. W. Kwan, P. McCorquodale, D. B. Serafini, A. Friedman, D. P. Grote, G. Westenskow, J.-C. Adam, A. Héron, and I. Haber, "Application of adaptive mesh refinement to particle-in-cell simulations of plasmas and beams," *Phys. of Plasmas*, vol. 11, no. 5, pp. 2928–2934, Feb. 2004. DOI: 10.1063/1.1689669.

[109] J.-L. Vay, A. Almgren, J. Bell, L. Ge, D. Grote, M. Hogan, O. Kononenko, R. Lehe, A. Myers, C. Ng, J. Park, R. Ryne, O. Shapoval, M. Thévenet, and W. Zhang, "Warp-X: A new exascale computing platform for beam–plasma simulations," *Nucl. Instrum. and Methods Phys. Res. Sect. A: Accel., Spectrometers, Detect. and Associated Equip.*, vol. 909, pp. 476–479, Jan. 2018. DOI: 10.1016/j.nima.2018.01.035.

[110] S. Ku, R. Hager, C. Chang, J. Kwon, and S. Parker, "A new hybrid-lagrangian numerical scheme for gyrokinetic simulation of tokamak edge plasma," *J. Comput. Phys.*, vol. 315, pp. 467–475, Jun. 2016. DOI: 10.1016/j.jcp.2016.03.062.

[111] J.-F. Roussel, F. Rogier, G. Dufour, J.-C. Mateo-Velez, J. Forest, A. Hilgers, D. Rodgers, L. Girard, and D. Payan, "SPIS open-source code: Methods, capabilities, achievements, and prospects," *IEEE Trans. Plasma Sci.*, vol. 36, no. 5, pp. 2360–2368, Nov. 2008. DOI: `10.1109/TPS.2008.2002327`.

[112] R. Marchand, "PTetra, a tool to simulate low orbit satellite–plasma interaction," *IEEE Trans. Plasma Sci.*, vol. 40, no. 2, pp. 217–229, Nov. 2012. DOI: `10.1109/TPS.2011.2172638`.

[113] B. Wang, S. Ethier, W. Tang, T. Williams, K. Z. Ibrahim, K. Madduri, S. Williams, and L. Oliker, "Kinetic turbulence simulations at extreme scale on leadership-class systems," in *Proc. Int. Conf. High Perfom. Comput., Netw., Storage and Anal.*, Denver, CO, USA, Nov. 2013. DOI: `10.1145/2503210.2503258`.

[114] V. K. Decyk and T. V. Singh, "Adaptable particle-in-cell algorithms for graphical processing units," *Comput. Phys. Commun.*, vol. 182, no. 3, pp. 641–648, Mar. 2011. DOI: `10.1016/j.cpc.2010.11.009`.

[115] G. Chen, L. Chacón, and D. Barnes, "An efficient mixed-precision, hybrid CPU–GPU implementation of a nonlinearly implicit one-dimensional particle-in-cell algorithm," *J. Comput. Phys.*, vol. 231, no. 16, pp. 5374–5388, Jun. 2012. DOI: `10.1016/j.jcp.2012.04.040`.

[116] M. T. Bettencourt, D. A. S. Brown, K. L. Cartwright, E. C. Cyr, C. A. Glusa, P. T. Lin, S. G. Moore, D. A. O. McGregor, R. P. Pawlowski, E. G. Phillips, N. V. Roberts, S. A. Wright, S. Maheswaran, J. P. Jones, and S. A. Jarvis, "EMPIRE-PIC: A performance portable unstructured particle-in-cell code," *Commun. Comput. Phys.*, no. 4, pp. 1–37, Mar. 2021. DOI: `10.4208/cicp.OA-2020-0261`.

[117] The Trilinos Project Team. (). "The Trilinos Project Website" trilinos.github.io. `https://trilinos.github.io` (acccessed July 20, 2021).

[118] (). *Cabana*. (2016). Co-design Center for Particle Applications. Accessed: Aug. 10, 2021. `https://github.com/ECP-copa/Cabana`.

[119] S. M. Mniszewski, J. Belak, J.-L. Fattebert, C. F. Negre, S. R. Slattery, A. A. Adedoyin, R. F. Bird, C. Chang, G. Chen, S. Ethier, S. Fogerty, S. Habib, C. Junghans, D. Lebrun-Grandié, J. Mohd-Yusof, S. G. Moore, D. Osei-Kuffuor, S. J. Plimpton, A.

Pope, S. T. Reeve, L. Ricketson, A. Scheinberg, A. Y. Sharma, and M. E. Wall, "Enabling particle applications for exascale computing platforms," *Int. J. High Perfom. Comput. Appl.*, pp. 1–26, Jul. 2021. DOI: `10.1177/10943420211022829`.

[120] K. Germaschewski, C. S. Chang, J. Dominski, R. Hager, S.-H. Ku, and A. Scheinberg, "Quantifying and improving performance of the XGC code to prepare for the exascale," in *APS Division Plasma Phys. Meeting Abstr.*, Jan. 2019, pp. BP10.081.

[121] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 5.1*, (2020). Accessed: Jul. 19, 2021 `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf`, 2020.

[122] OpenACC-Standard.org, *The OpenACC Application Programming Interface Version 2.1*, (2020). Accessed: Jul. 19, 2021. `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf`, 2020.

[123] D. S. Medina, A. St-Cyr, and T. Warburton, "OCCA: A unified approach to multi-threading languages," 2014, *arXiv*: 1403.0968.

[124] E. S. Seol and M. S. Shephard, "Efficient distributed mesh data structure for parallel automated adaptive analysis," *Eng. Comput.*, vol. 22, no. 3-4, pp. 197–213, Dec. 2006.

[125] M. Mubarak, S. Seol, Q. Lu, and M. S. Shephard, "A parallel ghosting algorithm for the Flexible Distributed Mesh Database.," *Sci. Program.*, vol. 21, no. 1, pp. 17–42, Jan. 2013.

[126] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *Proc. Int. Conf. High Perfom. Comput., Netw., Storage and Anal.*, Nov. 2012, pp. 1–10.

[127] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. Conf. High Perfom. Comput. Netw., Storage and Anal.*, Nov. 2009, pp. 1–11.

[128] *ITPACK 2.0 user's guide*, CNA-150, Center for Numer. Anal., Univ. of Texas, Austin, TX, USA, 1979.

[129] F. Vázquez, J. J. Fernández, and E. M. Garzón, "A new approach for sparse matrix vector product on NVIDIA GPUs," *Concurr. and Comput.: Pract. and Exper.*, vol. 23, no. 8, pp. 815–826, Jun. 2011. DOI: `10.1002/cpe.1658`.

[130] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop, "Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation," in *Proc. 2012 IEEE 26th Int. Parallel and Distrib. Process. Symp. Workshops Ph.D. Forum*, May 2012, pp. 1696–1702.

[131] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units," *SIAM J. on Sci. Comput.*, vol. 36, no. 5, pp. C401–C423, Apr. 2014. DOI: 10.1137/130930352.

[132] M. Besta, F. Marending, E. Solomonik, and T. Hoefler, "SlimSell: A vectorized graph representation for breadth-first search," in *Proc. 31st IEEE Int. Parallel and Distrib. Process. Symp.*, Orlando, FL, USA, May 2017, pp. 32–41.

[133] R. Bird, P. Gillies, M. Bareford, J. Herdman, and S. Jarvis, "Mini-app driven optimisation of inertial confinement fusion codes," in *Proc. 2015 IEEE Int. Conf. Cluster Comput.*, Sep. 2015, pp. 768–776. DOI: 10.1109/CLUSTER.2015.132.

[134] R. F. Bird, P. Gillies, M. R. Bareford, A. Herdman, and S. Jarvis, "Performance optimisation of inertial confinement fusion codes using mini-applications," *Int. J. High Perfom. Comput. Appl.*, vol. 32, no. 4, pp. 570–581, Jul. 2018. DOI: 10.1177/1094342016670225.

[135] S. Desai, "Enhancing the predictive power of molecular dynamics simulations to further the materials genome initiative," Ph.D. dissertation, Dept. Materials Eng., Purdue Univ. Graduate School, West Lafayette, IN, USA, 2020.

[136] R. L. Graham and G. Shipman, "MPI support for multi-core architectures: Optimized shared memory collectives," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds. Berlin, Germany: Springer, 2008. DOI: 10.1007/978-3-540-87475-1_21.

[137] M. F. Adams, S.-H. Ku, P. Worley, E. D'Azevedo, J. C. Cummings, and C.-S. Chang, "Scaling to 150K cores: Recent algorithm and performance engineering developments enabling XGC1 to run at scale," *J. Phys.: Conf. Ser.*, vol. 180, no. 012036, pp. 1–10, Jul. 2009. DOI: 10.1088/1742-6596/180/1/012036.

[138] G. B. Macpherson, N. Nordin, and H. G. Weller, “Particle tracking in unstructured, arbitrary polyhedral meshes for use in CFD and molecular dynamics,” *Commun. Numer. Methods Eng.*, vol. 25, no. 3, pp. 263–273, Jun. 2009. DOI: `10.1002/cnm.1128`.

[139] R. Chordá, J. Blasco, and N. Fueyo, “An efficient particle-locating algorithm for application in arbitrary 2D and 3D grids,” *Int. J. Multiphase Flow*, vol. 28, no. 9, pp. 1565–1580, Sep. 2002. DOI: `10.1016/S0301-9322(02)00045-9`.

[140] T. Frank and I. Schulze, “Numerical simulation of gas-droplet flow around a nozzle in a cylindrical chamber using a lagrangian model based on a multigrid navier-stokes solver,” in *Proc. Int. Symp. Numer. Methods Multiphase Flows*, vol. 185, Jun. 1994, pp. 93–107.

[141] P. Oliveira, A. Gosman, and R. Issa, “A method for particle location and field interpolation on complex, three-dimensional computational meshes,” *Adv. Eng. Softw.*, vol. 28, no. 9, pp. 607–614, Dec. 1997. DOI: `10.1016/S0965-9978(97)00052-5`.

[142] J. Valentine and R. Decker, “A lagrangian-eulerian scheme for flow around an airfoil in rain,” *Int. J. Multiphase Flow*, vol. 21, no. 4, pp. 639–648, Aug. 1995. DOI: `10.1016/0301-9322(95)00007-K`.

[143] Q. Zhou and M. Leschziner, “An improved particle-locating algorithm for eulerian-lagrangian computations of two-phase flows in general coordinates,” *Int. J. Multiphase Flow*, vol. 25, no. 5, pp. 813–825, Aug. 1999. DOI: `10.1016/S0301-9322(98)00045-7`.

[144] X.-Q. Chen and J. Pereira, “A new particle-locating method accounting for source distribution and particle-field interpolation for hybrid modeling of strongly coupled two-phase flows in arbitrary coordinates,” *Numer. Heat Transfer, Part B: Fundam.*, vol. 35, no. 1, pp. 41–63, Oct. 1999. DOI: `10.1080/104077999276009`.

[145] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, “OptiX: A general purpose ray tracing engine,” *ACM Trans. Graph.*, vol. 29, no. 4, pp. 66:1–66:13, Jul. 2010.

[146] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, “Real-time parallel hashing on the GPU,” *ACM Trans. Graph.*, vol. 28, no. 5, pp. 1–9, Dec. 2009. DOI: `10.1145/1661412.1618500`.

[147] Y. Chen and S. E. Parker, "Electromagnetic gyrokinetic δf particle-in-cell turbulence simulation with realistic equilibrium profiles and geometry," *J. Comput. Phys.*, vol. 220, no. 2, pp. 839–855, Jan. 2007. DOI: `10.1016/j.jcp.2006.05.028`.

[148] F. Jenko, W. Dorland, M. Kotschenreuther, and B. N. Rogers, "Electron temperature gradient driven turbulence," *Phys. Plasmas*, vol. 7, no. 5, pp. 1904–1910, Feb. 2000. DOI: `10.1063/1.874014`.

[149] T. Görler, X. Lapillonne, S. Brunner, T. Dannert, F. Jenko, F. Merz, and D. Told, "The global version of the gyrokinetic turbulence code gene," *J. Comput. Phys.*, vol. 230, no. 18, pp. 7053–7071, May 2011. DOI: `10.1016/j.jcp.2011.05.034`.

[150] S. Ku, C. Chang, R. Hager, R. Churchill, G. Tynan, I. Cziegler, M. Greenwald, J. Hughes, S. E. Parker, M. Adams, E. D'Azevedo, and P. Worley, "A fast low-to-high confinement mode bifurcation dynamics in the boundary-plasma gyrokinetic code XGC1," *Phys. Plasmas*, vol. 25, no. 5, pp. 1–44, Apr. 2018.

[151] E. Lanti, N. Ohana, N. Tronko, T. Hayward-Schneider, A. Bottino, B. McMillan, A. Mishchenko, A. Scheinberg, A. Biancalani, P. Angelino, S. Brunner, J. Dominski, P. Donnel, C. Gheller, R. Hatzky, A. Jocksch, S. Jolliet, Z. Lu, J. Martin Collar, I. Novikau, E. Sonnendrücker, T. Vernay, and L. Villard, "ORB5: A global electromagnetic gyrokinetic code using the PIC approach in toroidal geometry," *Comput. Phys. Commun.*, vol. 251, no. 107072, pp. 1–22, Jun. 2020. DOI: `10.1016/j.cpc.2019.107072`.

[152] J. Dominski, J. Cheng, G. Merlo, V. Carey, R. Hager, L. Ricketson, J. Choi, S. Ethier, K. Germaschewski, S. Ku, A. Mollen, N. Podhorszki, D. Pugmire, E. Suchyta, P. Trivedi, R. Wang, C. S. Chang, J. Hittinger, F. Jenko, S. Klasky, S. E. Parker, and A. Bhattacharjee, "Spatial coupling of gyrokinetic simulations, a generalized scheme based on first-principles," *Physics of Plasmas*, vol. 28, no. 2, pp. 1–15, Feb. 2021. DOI: `10.1063/5.0027160`.

[153] T. Younkin, D. Green, R. Doerner, D. Nishijima, J. Drobny, J. Canik, and B. Wirth, "GITR simulation of helium exposed tungsten erosion and redistribution in PISCES-A," in *59th Annu. Meeting APS Division Plasma Phys.*, Oct. 2017, UO4.002.

[154] K. Kim, J.-M. Kwon, C. S. Chang, J. Seo, S. Ku, and W. Choe, “Full-f XGC1 gyrokinetic study of improved ion energy confinement from impurity stabilization of ITG turbulence,” *Phys. Plasmas*, vol. 24, no. 6, pp. 1–13, May 2017. DOI: 10.1063/1.4984991.

[155] G. Merlo, J. Dominski, A. Bhattacharjee, C. S. Chang, F. Jenko, S. Ku, E. Lanti, and S. Parker, “Cross-verification of the global gyrokinetic codes GENE and XGC,” *Phys. Plasmas*, vol. 25, no. 6, pp. 1–23, Jun. 2018. DOI: 10.1063/1.5036563.

[156] J. Dominski, S.-H. Ku, and C. Chang, “Gyroaveraging operations using adaptive matrix operators,” *Phys. Plasmas*, vol. 25, no. 5, pp. 1–13, Apr. 2018. DOI: 10.1063/1.5026767.

[157] W. Lee, “Gyrokinetic particle simulation model,” *J. Comput. Phys.*, vol. 72, no. 1, pp. 243–269, Sep. 1987. DOI: 10.1016/0021-9991(87)90080-5.

[158] Z. Lin and W. W. Lee, “Method for solving the gyrokinetic poisson equation in general geometry,” *Phys. Rev. E*, vol. 52, no. 5, pp. 5646–5652, Nov. 1995. DOI: 10.1103/PhysRevE.52.5646.

[159] G. Diamond, L. Davis, and C. W. Smith, “Accelerated Load Balancing of Unstructured Meshes,” presented at the 27th Int. Meshing Roundtable, Albuquerque, NM, USA, Oct. 2-3, 2018.

APPENDIX A. ENGPAR LOAD BALANCING INPUTS

Access to different inputs to EnGPar's diffusive load balancer are provided through the DiffusiveInput class. The most important inputs are the priorities and tolerances that can be easily setup using the addPriorities function. This function takes in the entity type to be balanced, either the graph vertices or a hyperedge type, and the target imbalance for those entities. The function is called for each criteria that the application desires to balance. Listing A.1 shows an example of setting up the priorities such that the hyperedge type 0 has first priority followed by the graph vertices. The hyperedges are set to be balanced to a goal of 1.05 and the goal for vertices is set to 1.1.

Listing A.1: Example setting up priorities in EnGPar

```
Ngraph* graph = //Create the N-graph
DiffusiveInput* input(graph);
input->addPriority(0, 1.05); //0 refers to hyperedge type 0
input->addPriority(-1, 1.1); //-1 equates to graph vertices
engpar::balance(input);
```

The DiffusiveInput class also contains other parameters such as the step factor, α, used to control the amount of weight sent each iteration, the edge mitigation value β discussed in Section 2.3.4, and lower level controls for the balancer ranging from iteration counts to the edge type used to determine different types of connectivity. For most applications, the default values for the lower level parameters are sufficient, but access is available in the event an application needs finer control.