Parallel, Asynchronous Ray-Tracing for Scalable, 3D, Full-Core Method of Characteristics Neutron Transport on Unstructured Mesh

by

Derek Ray Gaston

M.S., Computational Applied Mathematics, 2006
University of Texas in Austin

B.S., Computer Science, 2004
University of Missouri in Rolla

Submitted to the Department of Nuclear Science and Engineering
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Computational Nuclear Science and Engineering

at the
Massachusetts Institute of Technology
February 2020

©2020 Massachusetts Institute of Technology. All rights reserved.

Author____
Department of Nuclear Science and Engineering
September 12, 2019

Certified by____________________________________
Kord S. Smith, Ph.D.
KEPCO Professor of the Practice of Nuclear Science and Engineering
Thesis Supervisor

Certified by____________________________________
Benoit Forget, Ph.D.
Professor of Nuclear Science and Engineering
Thesis Supervisor

Accepted by____________________________________
Nicolas G. Hadjiconstantinou, Ph.D.
Professor of Mechanical Engineering
Co-Director, Computational Science and Engineering

Accepted by____________________________________
Ju Li, Ph.D.
Battelle Energy Alliance Professor of Nuclear Science and Engineering
Professor of Materials Science and Engineering
Chairman, Department Committee on Graduate Theses

Parallel, Asynchronous Ray-Tracing for Scalable, 3D, Full-Core Method of Characteristics Neutron Transport on Unstructured Mesh

by

Derek Ray Gaston

Submitted to the Department of Nuclear Science and Engineering
on September 12, 2019, in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy in Computational Nuclear Science and Engineering

Abstract

One important goal in nuclear reactor core simulations is the computation of detailed 3D power distributions that will enable higher confidence in licensing of next-generation reactors and lifetime extensions/power up-rates for current-generation reactors. To date, there have been only a few demonstrations of such high-fidelity deterministic neutron transport calculations. However, as computational power continues to grow, such capabilities continue to move closer to being practically realized.

Predictive reactor physics needs both neutronics calculations and full-core, 3D coupled multiphysics simulations (e.g., neutronics, fuel performance, fluid mechanics, structural mechanics). Therefore, new reactor physics tools should harness supercomputers to enable full-core reactor simulations and be capable of coupling for multiphysics feedback.

One candidate for full-core nuclear reactor neutronics is the method of characteristics (MOC). Recent advancements have seen a pellet-resolved 3D MOC solution for the BEAVRS benchmark. However, MOC is traditionally implemented using constructive solid geometry (CSG) that makes it difficult (if not impossible) to accurately deform material to capture physical feedback effects such as fuel pin thermal expansions, assembly bowings, or core flowering.

An alternative to CSG is to use unstructured, finite-element mesh for spatial discretization of MOC. Such mesh-based geometries permit directly linking to unstructured mesh-based multiphysics tools, such as fuels performance. Utilizing unstructured mesh has been attempted in the past, but those attempts have fallen short of producing usable 3D reactor simulators. Several key issues have hindered these attempts: lack of fuel volume preservation, approximations of boundary conditions, inefficient spatial domain decompositions, excessive memory requirements, ineffective parallel load balancing, and lack of scalability on massively parallel modern computer clusters.

This thesis resolves these issues by developing a massively parallel, 3D, full-core MOC code, called `MOCkingbird`, using unstructured meshes. Underpinning `MOCkingbird` is a new algorithm for parallel ray tracing: the Scalable Massively Asynchronous Ray Tracing (SMART) algorithm. This algorithm enables efficient parallel ray-tracing across the full reactor domain, alleviating issues of reduced convergence associated with standard parallel MOC algorithms.

In addition, to enable full-core simulation using unstructured mesh MOC, several new algorithms are developed, including reactor mesh generation, sparse parallel communication, parallel cyclic track generation, and weighted partitioning. Within this

work `MOCkingbird` and SMART are tested for scalability from 10 to 20,000 cores on the Lemhi supercomputer at Idaho National Laboratory. Accuracy is tested using a suite of benchmarks that ultimately culminate in a first-of-a-kind, 3D, full-core, simulation of the BEAVRS benchmark using unstructured mesh MOC.

Thesis Supervisor: Kord S. Smith
Title: Professor of the Practice of Nuclear Science and Engineering

Thesis Supervisor: Benoit Forget
Title: Professor of Nuclear Science and Engineering

ACKNOWLEDGMENTS

First and foremost, I need to thank my fiancé, Andrea, for your incredible patience and support over these last two years. We met at a crazy time, and I'll always be thankful for your help in finishing this. Also, I have to thank my little boy, Eli, you haven't yet known a world where your father is free to play anytime.

Thank you, Prof. Smith, and Prof. Forget for everything you've done for me over the years. I value our collaborations and our friendship.

To my parents, Edie and Michael: thank you for always believing in me and always being there when I need you the most. Your support has never wavered, and I'm forever grateful for that. I also want to thank my sister and brother, Dayna and Thomas, for all of your support.

None of this would have been possible, not MOOSE, not MOCkingbird, not MIT, without my good friend, Dr. Richard Martineau. Thank you for everything.

Sometimes in life, we run into people, and you know that you'll be friends with them forever. Thank you, John Tramm, for always being there, always listening and for being a constant sounding board for all of my crazy ideas. I wouldn't have made it through without you.

A huge thank you to the MOOSE team, for being tolerant of my absences and for always being friends when I return.

Thank you, Logan, for putting up with beta software and my distracted self. This is only the beginning!

I need to thank a massive list of colleagues and collaborators who have helped in this project: Carl, Mike, Roy, Matt, Geoff, Sterling, Guillaume, Johnny, Sam, Will, the CRPG, Yaqi, Sebastian, Javi, Jieun, Mark and so many more. Thank you!

I need to thank Jacopo, Peter, Brandy, Heather, and Lisa from the MIT NSE department. I know I was a pain - I appreciate everything!

This wouldn't have been possible without the fellowships I received. Thank you, Dr. David C. Aldrich and Mrs. Hamila Atefi for the Driscoll Fellowship, Mrs. Carol Papay for the Thompson Fellowship and Ms. Kearney for the Bechtel/Kearney Fellowship.

I would also like to thank Idaho National Laboratory for all of the support over the years.

I have to thank the high-performance computing team at Idaho National Lab and, in particular, Eric Whiting and Tami Grimmett. All of the results in this document were only possible because of you.

CONTENTS

1 INTRODUCTION 18
1.1 Motivation 18
1.2 Objective 19
1.3 Thesis Outline 19

2 METHOD OF CHARACTERISTICS 21
2.1 Characteristic Equation 21
2.2 Multigroup Approximation and Cross Section Condensation 23
2.3 Angular Quadrature 25
2.4 Spatial Discretization 26
2.5 Tracks, Segmentation, and Ray Tracing 27
2.6 Cyclic Tracking 29
2.7 Source Iteration and Transport Sweeps 30
2.8 Transport Correction and Stabilization 32
2.9 Parallelism and Domain Decomposition 33
2.10 Memory 35

3 COMPUTING BACKGROUND 38
3.1 Cluster Computing 38
3.1.1 Lemhi Supercomputer 39
3.2 Message Passing Interface 40
3.3 Meshes 42
3.4 Domain Decomposition 44
3.4.1 Partitioning 44
3.4.2 Parallel Storage 44
3.5 libMesh 45
3.6 MOOSE 46
3.6.1 Scalability 47
3.7 OpenMOC 47

4 LITERATURE REVIEW AND `mockingbird` OVERVIEW 48
4.1 Literature Review 48
4.1.1 MOCFE / Proteus-MOCEX 48
4.1.2 MOCUM 49
4.1.3 Linear Source On Meshes 49
4.1.4 MoCha-Foam 50
4.2 MOCkingbird 50
4.2.1 Libraries Used 51
4.2.2 Solution Methodology 53
4.2.3 Executing `MOCkingbird` 56

5 NUCLEAR REACTOR MESHING 58
5.1 Cubit Pin-Cells 59

5.2 Graph-based Mesh Generation . . . 68
5.2.1 Three-dimensional Reactor Meshing . . . 71
5.2.2 Meshing Conclusion . . . 76

6 SPARSE, SCALABLE, ASYNCHRONOUS COMMUNICATION ALGORITHMS FOR PROBLEM SETUP 77
6.1 Sparse Data Exchange Algorithms . . . 77
6.1.1 Scalable Sparse Data Exchange . . . 78
6.1.2 Testing Scalable Sparse Data Exchange . . . 84
6.1.3 An Interface For Sparse Data Exchange . . . 89

7 SCALABLE MASSIVELY ASYNCHRONOUS RAY TRACING (smart) 91
7.1 Tracks and The Ray Object . . . 92
7.2 Element Traversal Algorithm . . . 95
7.2.1 Corner Cases . . . 97
7.2.2 Corner Case Rarity . . . 100
7.3 Parallel Ray Tracing: SMART . . . 102
7.3.1 Overview . . . 102
7.3.2 Algorithms . . . 104
7.3.3 Data Structures . . . 109
7.3.3.1 Object Pool . . . 109
7.3.3.2 Communication Buffers . . . 110
7.3.4 Object Oriented Design . . . 111

8 MOCKINGBIRD MOC 115
8.1 Track Generation and Parallel Ray Claiming . . . 115
8.1.1 Claiming Example . . . 123
8.2 Source Iteration . . . 125
8.2.1 Storage . . . 126
8.2.2 Object Oriented Design . . . 126
8.2.3 Transport Sweep . . . 128
8.3 Scalability . . . 130
8.3.1 Weak Scalability . . . 132
8.3.1.1 2D Weak Scalability . . . 133
8.3.1.2 3D Weak Scalability . . . 139
8.3.2 Strong Scalability . . . 141
8.3.3 Mesh Partitioning . . . 144
8.3.3.1 Perfect Lattice . . . 145
8.3.3.2 Weighted Partitioning . . . 147
8.3.4 Comparison To Other Algorithms . . . 156
8.3.4.1 Weak Scaling . . . 159
8.3.4.2 Strong Scaling . . . 159
8.3.4.3 Conclusion . . . 162

9 BENCHMARK PROBLEMS 163
9.1 Introduction . . . 163
9.2 2D C5G7 . . . 163
9.3 3D C5G7 . . . 167

9.4 2D BEAVRS 170
9.4.1 Meshing 171
9.4.2 Cross Sections and Reference Values 173
9.4.3 Results 174
9.4.4 Scalability 176
9.5 3D BEAVRS 178
9.5.1 Geometry and Meshing 179
9.5.2 3D Quarter-Core 182
9.5.3 3D Full-Core 187

10 CONCLUSION 190
10.1 Summary of Work 191
10.2 Future Work 193
10.2.1 Acceleration 193
10.2.2 Saving Segments 194
10.2.3 Vectorization 195
10.2.4 Hybrid Parallel 195
10.2.5 Linear Source 196
10.2.6 TRRM 196
10.2.7 Multiphysics 196

I Appendix 197

11 mockingbird INPUT FILE FORMAT 198

12 C5G7 INPUT 199

13 ENERGY GROUP STRUCTURE FOR BEAVRS CROSS SECTIONS 203

14 GEOMETRICAL INTERSECTION ALGORITHMS 206

15 OBJECTPOOL 211

BIBLIOGRAPHY 213

LIST OF FIGURES

Figure 2.1 U238 capture cross section (blue), example neutron flux (green) and 16 energy group representation of the cross section. From [3] 24
Figure 2.2 An example of a traditional flat source region treatment of a 4x4, pin-cell lattice. From [2] 26
Figure 2.3 Example equally spaced tracks and the segments that would lie along one such track. 27
Figure 2.4 2D and 3D cyclic tracking. The 3D tracks are defined as "stacks" above the 2D tracks. From [12]. 29
Figure 2.5 Modular ray tracing combined with spatial domain decomposition. The domain has been decomposed into 9 partitions. From [5] 34
Figure 2.6 Growth of angular flux storage vs scalar flux storage when using modular ray tracing. Computed using partitioning into cubes of MPI ranks. From [5] 36
Figure 3.1 Pictorial of a modern cluster computer. 39
Figure 3.2 Examples of two- and three-dimensional elements 43
Figure 3.3 Examples of structured and unstructured meshes. 43
Figure 3.4 The partitioning process. First, a dual-graph is made from the mesh. Next, the dual-graph is partitioned using partitioning software that tries to balance the work and minimize the number of edge cuts. Finally, the elements are assigned MPI ranks based on the partitioning. 45
Figure 4.1 The relationship between `MOCkingbird` and libraries it uses. The orange libraries represent the set of libraries a `MOOSE`-based code would normally use. `OpenMOC` is set apart in green to show that it's added for `MOCkingbird`. 51
Figure 4.2 The mesh is split for 3 MPI ranks, and one track is considered. With a block-Jacobi-like algorithm, the boundary angular flux would only propagate to the other side of the domain after 11 source iterations. In `MOCkingbird` the entire track is traced in one iteration, without any intermediate global synchronization. 55
Figure 5.1 Example LWR pin-cell. 59
Figure 5.2 The constructive solid geometry process. [83] 59
Figure 5.3 Constructive solid geometry and simple meshing of a C5G7 pin-cell. 60
Figure 5.4 Meshing by quarters gives much better control and symmetry. Generated by Listing 5.1. 61
Figure 5.5 By utilizing (5.2) the exact fuel area can be meshed. Generated by Listing 5.2. 63
Figure 5.6 Meshing using moderator zoning. 65
Figure 5.7 Mesh modifications are straightforward using the zoned moderator algorithm. 66
Figure 5.8 Meshed pin-cells from the BEAVRS benchmark [87]. 68

Figure 5.9 Representations of the mesh generation process for LWR reactors. 70
Figure 5.10 3D assemblies can be generated by extruding 2D assemblies. . . 73
Figure 5.11 Components of a nuclear fuel assembly. From [92] 73
Figure 5.12 The same mesh for a fuel pin-cell with different material assignments on the outer edge to represent either moderator or spacer grid. 74
Figure 5.13 Input file syntax and resulting 3D assembly with spacer grids using the pin-cells in Figure 5.12. The zircaloy elements have been highlighted. 75
Figure 6.1 One-sided, single-hop vs. multi-hop messaging. Single-hop algorithms must globally sync between each communication. Multi-hop allows a message to move through multiple MPI ranks without global synchronization. 78
Figure 6.2 Pictorial showing the action of the `MPI_Alltoall` within the PEX algorithm to notify receivers of who will be sending to them. 80
Figure 6.3 Sparse data exchange scalability in message size. The test was run using 4096 MPI processes spread across 128 nodes. Message sizes are in number of double precision floating point numbers (64bit) and Time is based on 10 iterations of the algorithm. . . . 85
Figure 6.4 Weak scalability of each sparse data exchange algorithm from 32 MPI processes to 8192. This is "weak" scaling: the number of neighbors is held constant at 26 and the message size is held at 100 double precision floating point numbers. The raw data for this plot can be found in Table 6.2 86
Figure 6.5 Testing each sparse data exchange algorithm's ability to deal with decreasing sparsity. The test was run using 4096 MPI processes spread across 128 nodes. Message size was held constant at 100 double precision floating point numbers. The time represents 10 iterations of the algorithm. The raw data for these plots can be found in 6.3 87
Figure 7.1 Pictorial representation of an internal corner strike. The blue ray is hitting an internal shared corner and the orange intersection will be the next segment chosen. 98
Figure 7.2 Pictorial representation of an intersection very near an external domain boundary. 99
Figure 7.3 Spatial distribution of intersection cases. The totals shown are the amount that happened within each MPI rank that owned that portion of the geometry. 101
Figure 7.4 High-level overview of how a `Ray` traverses a domain decomposed unstructured mesh. 103
Figure 8.1 Cyclic tracks from `OpenMOC` need to be mapped onto the partitioned, unstructured mesh (in this case, partitioned for 3 MPI ranks). The starting position of each track must be uniquely located within a single MPI rank. 116
Figure 8.2 Local partition bounding boxes for three partitions. 118

Figure 8.3 The track starting positions for three tracks: orange, pink and blue . . . 123
Figure 8.4 Steps of scalable track generation and claiming. . . . 124
Figure 8.5 Example of a NumericVector. The mesh is partitioned into three pieces, therefore the scalar flux vector is also split into three contiguous pieces which are held on the ranks whose elements are associated with those entries in the vector. . . . 127
Figure 8.6 The evolution of a sweep. Four rays (yellow, pink, green, orange) and their angular fluxes (ψ_g) are traced through the mesh. They start in the RayBank of an MPI rank. During the sweep they move through the mesh, modifying ψ_g and accumulating into $\phi_{i,g}$ in each element. When they reach the domain boundary they are put into the RayBank on that rank and given the new direction (reflected) they travel during the next sweep. . . 129
Figure 8.7 Unit cell for each partition within the 2D weak scalability study. 11520 quadrilateral elements total. . . . 134
Figure 8.8 Mesh partitioning (each block represents the partition for a different MPI process) for 144 MPI processes spread over 4 computational nodes (36 processes per node). . . . 135
Figure 8.9 2D weak scaling performance. . . . 137
Figure 8.10 Average memory used by each MPI process. . . . 138
Figure 8.11 Number of Ray objects held in the ObjectPool by each MPI process overlaid on the domain. . . . 139
Figure 8.12 The mesh for the 32 MPI process case for 3D weak scaling. . . . 140
Figure 8.13 3D weak scaling performance. . . . 141
Figure 8.14 Pin-cell used for strong scaling. 352 elements total. . . . 142
Figure 8.15 2D strong scaling performance. . . . 143
Figure 8.16 Comparison of 2D strong scaling performance with three different partitioning schemes. . . . 146
Figure 8.17 Total size (in cm) of the off-node communication surface area for each partitioning scheme. . . . 147
Figure 8.18 Memory scalability with the hierarchical partitioner. . . . 148
Figure 8.19 C5G7 geometry and mesh detail used for testing weighted partitioning. . . . 152
Figure 8.20 Results of a strong scaling study conducted using the mesh in Figure 8.19. ParMETIS and the hierarchical partitioner were tested both with and without element and side weights. . . . 153
Figure 8.21 Comparison of hierarch. partitioning without (left) and with (right) weighting. . . . 155
Figure 8.22 The number of global synchronizations the BS algorithm will incur as it tries to trace a ray across a jagged domain boundary. 157
Figure 8.23 Comparison of 2D weak scaling performance with three communication algorithms. . . . 160
Figure 8.24 Comparison of 2D strong scaling performance with three communication algorithms. . . . 161

Figure 9.1 Quarter-core C5G7 geometry and mesh detail for the bottom right corner of the core. Colors represent sets of fuel mixtures and the moderator. 163
Figure 9.2 Group flux results using 128 azimuthal angles. 165
Figure 9.3 Relative error in normalized pin powers. Left: 128 azimuthal angles, Right: 4 azimuthal angles. 165
Figure 9.4 Convergence in eigenvalue and pin-power for 2D C5G7. 166
Figure 9.5 Thermal flux for the 3D mesh with 50 axial layers. 167
Figure 9.6 Visualizations of the performance of `MOCkingbird` for solving 3D C5G7. 169
Figure 9.7 BEAVRS benchmark assembly layout for cycle 1. From [87] . . . 170
Figure 9.8 Pin-cell meshes and inter-assembly gap mesh. 171
Figure 9.9 Example of the baffle and water mesh surrounding the core. . . 172
Figure 9.10 As-meshed geometry for the 2D BEAVRS core. 172
Figure 9.11 Normalized assembly powers for the BEAVRS benchmark as computed by `OpenMC`. `OpenMC` computed eigenvalues: 2D k_{eff} = 1.00491, 3D k_{eff} = 1.00024 174
Figure 9.12 Normalized assembly power differences for the 2D BEAVRS solution computed by `MOCkingbird` compared to the OpenMC Monte Carlo result. 175
Figure 9.13 Thermal flux (0 ev to 9.87 eV) for the 2D BEAVRS benchmark. . 176
Figure 9.14 Strong scaling results for 2D BEAVRS. 177
Figure 9.15 BEAVRS axial elevation specification from [87]. 178
Figure 9.16 Slices showing the detail in the top and midplane of the, as meshed, full-core geometry. 179
Figure 9.17 The original BEAVRS elevations in cm (left) and the condensed elevations used here (right). 180
Figure 9.18 3D quarter-core assembly layout. Reflective boundary conditions will be applied on the north and west sides of the domain, with vacuum on the south and east. 182
Figure 9.19 Slices through the 3D quarter-core solution showing the thermal flux. Also, the assembly power error. 184
Figure 9.20 Time for track claiming algorithms for different numbers of MPI processes. The two algorithms taking the most time are shown in Figure 9.20a while the rest are in Figure 9.20b 185
Figure 9.21 Strong scaling and iteration time percent for the 3D quarter-core BEAVRS problem. A constant LNSI equates to 100% parallel efficiency. 186
Figure 9.22 Thermal flux for the full-core shown on two slices through the core. 188
Figure 9.23 A view of the partitioning and the amount of work (intersections) per partition. Figure 9.23b shows a three-dimensional view of the partitions for ranks 1000 and 1001. 189
Figure 10.1 Average memory usage per-core for `MOCkingbird`, compared against the theoretical limit for `SMART`, a pure MPI block-Jacobi implementation and a hybrid parallel block-Jacobi implementation. 194

Figure 14.1 Visual representation of Algorithm 21 206
Figure 14.2 Ray tracing across a hexahedral element from the front face to the back face. The back face will be made into two triangles that are each tested for intersections using Algorithm 23. The vertex numbering is also shown. 208

LIST OF TABLES

Table 6.1 Data for the message size testing of sparse data exchange algorithms. Each entry represents the time in milliseconds for 10 iterations of the algorithm. This data is also plotted within Figure 6.3. . . . 84
Table 6.2 Weak scaling of each sparse data exchange algorithm. The number of neighbors is held constant at 26 while the message size is 100 double precision numbers. Each entry represents the time in milliseconds for 10 iterations of the algorithm. This data is also plotted within Figure 6.4. . . . 86
Table 6.3 Scalability in the number of neighbors. Ran using 4096 MPI processes, messages size is held constant at 100 double precision numbers. Each entry represents the time in milliseconds for 10 iterations of the algorithm. This data is also plotted within Figure 6.5. . . . 87
Table 7.1 Ray tracing statistics for one transport sweep. . . . 100
Table 8.1 The core and mesh sequences used for the 2D weak scaling study. 137
Table 8.2 The core and mesh sequences used for the 3D weak scaling study. 139
Table 9.1 `MOCkingbird` converged eigenvalues for increasing azimuthal angles with an azimuthal spacing of 0.01cm. . . . 164
Table 9.2 Performance characteristics for each run in the angular refinement study. . . . 166
Table 9.3 Eigenvalues and fission source errors for each mesh used for the 3D C5G7 *Rodded B* configuration. The reference k_{eff} is 1.07777. 168
Table 9.4 "Polar angle study for 3D C5G7." . . . 168
Table 9.5 Performance characteristics for solving 3D C5G7 *Rodded B*. . . . 168
Table 9.6 Problem settings and computational requirements for 2D BEAVRS. 175
Table 9.7 Problem settings and computational requirements for the quarter-core 3D BEAVRS solution. . . . 183
Table 9.8 Timing for track generation and claiming. . . . 185
Table 9.9 Percentage of source iteration time in different code segments. . 186
Table 9.10 Problem settings and computational requirements for the full-core 3D BEAVRS solution. . . . 187

ACRONYMS

BEAVRS Benchmark for Evaluation and Validation of Reactor Simulations

BP Burnable Poison

C5G7 Neutronics Benchmark

CMFD Coarse Mesh Finite Difference

JSON JavaScript Object Notation

LNSI Levelized Nanoseconds Per Intersection

LWR Light-Water Reactor

MC Monte Carlo

MOC Method of Characteristics

MOOSE Multiphysics Object Oriented Simulation Environment

NSI Nanoseconds Per Intersection

NDA Nonlinear Diffusion Acceleration

PWR Pressurized Water Reactor

SMART Scalable Massively Asynchronous Ray-Tracing

RMS Root Mean Square

1 INTRODUCTION

1.1 MOTIVATION

Nuclear energy plays an essential role in our nation's energy infrastructure. The near-constant base-load generation from reactors is currently powering 20% of the electrical grid. In addition, that power generation is CO_2 free, an increasingly important fact as the world's climate changes. Keeping the existing reactors operating and bringing online new reactor designs is critical to ensuring the energy security of our nation.

Modeling and simulation of nuclear reactors play a vital role in their life cycle. The existing fleet has relied heavily on modeling and simulation, based on informed approximation from many experiments and many years of operating experience. However, next-generation designs and increased experimental costs increase the requirements for higher fidelity simulation tools. Many of the new designs, such as micro-reactors, rely on multiphysics effects for safety and operation that cannot be captured reliably with current approximation methods. Further, it would take significant experimental campaigns to demonstrate safe operation. Emerging reactor designs are employing sophisticated geometrical features such as twisted fuel and heat pipes that go beyond current modeling capabilities. Therefore, predictive modeling and simulation tools are needed to support the economical development of new nuclear reactor technologies.

These new tools need to be capable of high-fidelity, full-core neutron transport calculations. Also, they must handle heterogeneous, non-extruded geometries, and material deformation. An operating nuclear reactor is a balance of many different physics (e.g., neutron transport, heat conduction, fluid flow, chemistry) interacting to produce the behavior of the core. Therefore, a high-fidelity neutron physics simulator must also be able to interact with other physics solvers to achieve a multiphysics reactor simulation. Finally, due to the rise of cluster computing, a high-fidelity solver should scale well in parallel to reduce time to solution.

Therefore, this work focuses on finding a neutron transport method that is accurate, scalable, geometrically flexible, capable of full-core, 3D calculation, and able to respond to complex multiphysics interactions such as geometric deformation.

1.2 OBJECTIVE

This thesis develops a scalable, massively parallel, unstructured mesh-based, 3D, full-core MOC neutron transport tool: `MOCkingbird`. The software has the following capabilities:

- Both two- and three-dimensional
- Scalable on clusters, including:
  - Scalable setup and track generation phase
  - Weighted domain-decomposition for work balance
  - Scalable ray-tracing during the transport sweep
  - Scalable source update and convergence checking
  - Scalable memory usage
- Parallel agnostic (same solution behavior in serial and parallel)
- Cyclic tracking for accurate representation of reflective boundary conditions
- LWR mesh generation for symmetric, volume-preserving pin-cells
- Straightforward integration path with multiphysics solvers
- Serial execution speed similar to contemporary, traditional MOC codes

`MOCkingbird` is capable of completing efficient, parallel, full-core, 3D, steady-state, k-eigenvalue neutron transport solves utilizing unstructured, finite-element mesh for the domain description.

1.3 THESIS OUTLINE

This thesis proceeds by first providing some background on the method of characteristics (MOC). The Boltzmann neutron transport equation is introduced, and it is discretized it in energy, angle, and space. In addition, an introduction is given to traditional parallelization methods for MOC. Chapter 3 is meant to establish a basic vocabulary for parallel computing terms that are used within the thesis. It also introduces several of the code libraries which are relied upon by `MOCkingbird`

Chapter 4 explores the existing literature for unstructured mesh MOC, highlighting many of the current barriers to 3D full-core simulation. Next, an introduction is given for `MOCkingbird`, the 3D full-core, unstructured mesh MOC code developed in this thesis. In particular, the idea of using "real" long-characteristics in parallel is introduced.

Chapter 5 develops the tools required by MOCkingbird for reactor mesh generation. It begins by introducing a pin-cell generation technique. Next, a novel mesh-generation capability based around directed-acyclic-graphs (DAGs) is detailed. This mesh-generation capability is massively parallel, enabling the generation of 3D full-core meshes necessary for this thesis.

Chapter 6 studies several asynchronous communication algorithms suitable for use within MOCkingbird. Two novel algorithms are discussed which can provide scalable non-blocking communication for setting up the MOC calculation. This chapter will also fully explore the scalability of these algorithms.

Chapter 7 develops the Scalable Massively Asynchronous Ray-Tracing (SMART) algorithm that forms the core of the scalable transport sweep capability within MOCkingbird. An asynchronous ray-tracing algorithm performs the transport sweep. The implementation of this algorithm is an object-oriented, pluggable system for defining both on-segment calculations and boundary-conditions.

Chapter 8 details the MOCkingbird code itself. A new algorithm for parallel track generation and claiming is developed. Data structures and object-oriented design is explored, leading to an explanation of how SMART is utilized to perform massively parallel transport sweeps. The latter part of Chapter 8 is devoted to thoroughly testing the performance of the MOCkingbird code and SMART. Several different scaling studies are run, and comparisons to other ray-tracing algorithms are made. In addition, a new surface weighted partitioning scheme is developed and tested.

Finally, Chapter 9 tests the ability of MOCkingbird to solve a series of neutron transport benchmark problems. Both 2D and 3D versions of the C5G7 benchmark are first solved. These are used as the basis for several parameter studies. Next, the BEAVRS benchmark is solved in 2D then in a 3D quarter core configuration. This chapter ends with a first-of-a-kind solution of the 3D full-core BEAVRS benchmark using unstructured mesh MOC. Chapter 10 concludes the thesis and provides directions for future work.

2 METHOD OF CHARACTERISTICS

Neutron transport simulations can model shielding, neutron detectors, critical experiments, and the distribution of neutron flux within a reactor core. The Method of Characteristics (MOC) is a solution method that is used in many fields and has seen broad adoption in the nuclear engineering community for solution of the neutron transport equation[1]. The `MOCkingbird` code developed as part of this thesis utilizes a traditional form of MOC which makes use of flat spatial source regions and transport-corrected-P0 scattering cross sections. Many detailed treatments of this formulation are found in the literature [2, 3, 4, 5, 6].

This chapter develops the characteristic form of the steady-state Boltzmann neutron transport equation. The following sections simplify the equation through the use of the multigroup approximation and the introduction of angular quadrature and spatial discretization. Ultimately, a source iteration scheme is developed for the iterative solution of the k-eigenvalue problem.

2.1 CHARACTERISTIC EQUATION

A simplified form of the Boltzmann neutron transport equation is considered. In particular, steady state is assumed and an eigenvalue problem is formed:

$$\begin{aligned} \boldsymbol{\Omega} \cdot \nabla \psi(\boldsymbol{r}, \boldsymbol{\Omega}, E) + \Sigma_t(\boldsymbol{r}, E)\psi(\boldsymbol{r}, \boldsymbol{\Omega}, E) = \\ \int_0^\infty \int_{4\pi} \Sigma_s(\boldsymbol{r}, \Omega' \to \boldsymbol{\Omega}, E' \to E)\psi(\boldsymbol{r}, \Omega', E')d\Omega' dE' \\ + \frac{\chi(\boldsymbol{r}, E)}{4\pi k_{\text{eff}}} \int_0^\infty \int_{4\pi} \nu\Sigma_f(\boldsymbol{r}, E')\psi(\boldsymbol{r}, \Omega', E')d\Omega' dE'. \end{aligned} \tag{2.1}$$

Equation 2.1 represents a balance equation between production and loss of neutrons, with the fundamental eigenvalue, k_{eff}, balancing the system. The angular flux, ψ, is the dependent variable in Equation 2.1. Solving for ψ allows for the computation of reaction rates throughout the core. Σ_t, Σ_s and $\nu\Sigma_f$ are the continuous-in-energy total, scatter and fission macroscopic cross sections, respectively. A position in three-dimensional space is denoted by $\boldsymbol{r}$. The direction of neutron travel is represented by $\boldsymbol{\Omega}$, with E being the energy of the neutrons. The fission emission spectrum is represented by χ in an equilibrium of prompt and delayed emissions.

Equation (2.1) needs to be further simplified and discretized in order to efficiently find the angular flux and eigenvalue. The source terms from fission and scattering on the right-hand side of Equation (2.1) is "lagged" and an iterative method, called "source

iteration," is utilized to find a stationary point that satisfies the discretized form of the equation. What follows is the construction of the simplified and discretized form of the equation and development of the iteration scheme.

To begin, the right hand side of (2.1) is defined to be the total neutron source:

$$\begin{aligned} Q(\boldsymbol{r},\boldsymbol{\Omega},E) &= \int_0^\infty \int_{4\pi} \Sigma_s(\boldsymbol{r},\Omega' \to \boldsymbol{\Omega}, E' \to E)\psi(\boldsymbol{r},\Omega',E')d\Omega' dE' \\ &+ \frac{\chi(\boldsymbol{r},E)}{4\pi k_{\text{eff}}} \int_0^\infty \int_{4\pi} \nu\Sigma_f(\boldsymbol{r},E')\psi(\boldsymbol{r},\Omega',E')d\Omega' dE'. \end{aligned} \tag{2.2}$$

The total source, Q, can be simplified by assuming isotropic scattering (scattering is uniform in all angle) isotropic fission emission (neutrons are born uniformly in angle). While an isotropic fission source is a good assumption, isotropic scattering may not be (especially for collisions with light isotopes such as hydrogen). To account for this, a transport correction is applied to the cross sections, as explained in §2.8. With these simplifications, the scalar flux is defined as,

$$\phi(\boldsymbol{r},E) = \int_{4\pi} \psi(\boldsymbol{r},\Omega,E)d\Omega, \tag{2.3}$$

giving,

$$\begin{aligned} Q(\boldsymbol{r},E) &= \frac{1}{4\pi} \int_0^\infty \Sigma_s(\boldsymbol{r},E' \to E)\phi(\boldsymbol{r},E')dE' \\ &+ \frac{\chi(\boldsymbol{r},E)}{4\pi k_{\text{eff}}} \int_0^\infty \nu\Sigma_f(\boldsymbol{r},E')\phi(\boldsymbol{r},E')dE' \end{aligned} \tag{2.4}$$

With this substitution the transport equation becomes:

$$\boldsymbol{\Omega} \cdot \nabla\psi(\boldsymbol{r},\boldsymbol{\Omega},E) + \Sigma_t(\boldsymbol{r},E)\psi(\boldsymbol{r},\boldsymbol{\Omega},E) = Q(\boldsymbol{r},E). \tag{2.5}$$

This equation can be considered along any "characteristic": a straight path through the domain in the direction $\boldsymbol{\Omega}$. This path, or track, can be parameterized by specifying $\boldsymbol{r} = \boldsymbol{r_0} + s\boldsymbol{\Omega}$ where $\boldsymbol{r_0}$ is a reference location and s is a scalar. Therefore, $s\boldsymbol{\Omega}$ is a distance s along the track. Making this substitution:

$$\Omega \cdot \nabla\psi(\boldsymbol{r_0}+s\boldsymbol{\Omega},\boldsymbol{\Omega},E) + \Sigma_t(\boldsymbol{r_0}+s\boldsymbol{\Omega},E)\psi(\boldsymbol{r_0}+s\boldsymbol{\Omega},\boldsymbol{\Omega},E) = Q(\boldsymbol{r_0}+s\boldsymbol{\Omega},\boldsymbol{\Omega},E). \tag{2.6}$$

After applying the differential along the direction of $\boldsymbol{\Omega}$, and letting s have implied dependence on the reference location, the characteristic form of the neutron transport equation can be written as:

$$\frac{d}{ds}\psi(s,\boldsymbol{\Omega},E) + \Sigma_t(s,E)\psi(s,\boldsymbol{\Omega},E) = Q(s,\boldsymbol{\Omega},E). \tag{2.7}$$

Equation 2.7 is now an ordinary differential equation with continuously varying cross sections in space and energy. The differential equation can be solved using an integrating factor,

$$e^{-\int_0^s \Sigma_t(s',E)ds'}, \tag{2.8}$$

which gives:

$$\psi(s, \mathbf{\Omega}, E) = \psi(\boldsymbol{r_0}, \mathbf{\Omega}, E)e^{-\int_0^s \Sigma_t(s',E)ds'} + \int_0^s Q(s'', \mathbf{\Omega}, E)e^{-\int_0^{s''} \Sigma_t(s',E)ds'} ds''. \tag{2.9}$$

Equation 2.9 is the analytic solution to the characteristic form of the Boltzmann equation given the stated assumptions. It provides the angular flux at a point located a distance s along a track in the direction $\mathbf{\Omega}$. The first of the two terms on the right-hand side includes a known "incoming" angular flux at the origin of the track. This flux is being attenuated along this direction as s moves along the track. The second term on the right-hand side represents the angular flux that results from a source of neutrons emitted into this direction and energy, either by being born from fission or scattering from some other direction and energy.

While Equation 2.9 can theoretically provide the angular flux at any point in space, it is not yet useful for numerical solutions. In particular, the cross sections are continuously varying in space and energy, complicating numerical evaluation of the integrals required in 2.9. In addition, the spatially varying neutron flux needs to be computed throughout the entire reactor, not just along one line. Therefore, a discretized form of 2.9 is paired with quadratures in both space and angle.

The following sections further simplify and discretize Equation 2.9 and form an iterative method for finding the neutron flux throughout the reactor.

2.2 MULTIGROUP APPROXIMATION AND CROSS SECTION CONDENSATION

The first simplification is to discretize continuous energy into G discrete energy groups: $g \in \{1, 2, ..., G\}$ where energy group g spans the energy range E_g to E_{g-1}. It is customary to order the energy group boundaries from highest energy to lowest energy, i.e. $E_g < E_{g-1}$. To be able to utilize this multigroup simplification, group-wise cross sections need to be found.

As shown in Figure 2.1, a continuous energy cross section can be rapidly varying in energy. Attempting to average this cross section in an energy range would result in a poor representation of the true reaction probability and lead to errors in the calculated reaction rates. Instead, a weighted average in energy is needed. To preserve reaction

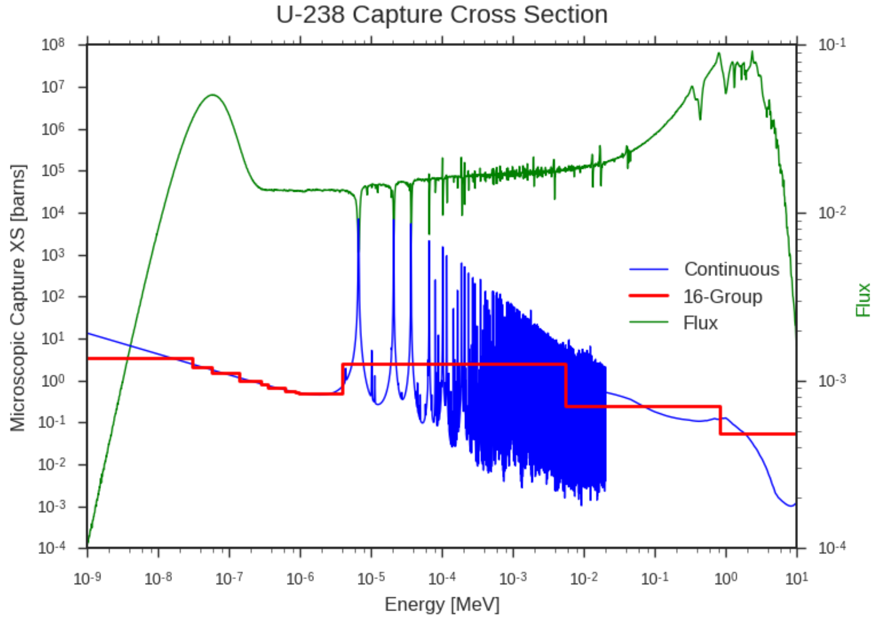


Figure 2.1. U238 capture cross section (blue), example neutron flux (green) and 16 energy group representation of the cross section. From [3]

.

rates, the cross section weighted with an assumed neutron flux spectrum is used to condense continuous energy cross sections into energy groups,

$$\Sigma_{x,g}(\boldsymbol{r}) = \frac{\int_{E_g}^{E_g-1} \Sigma_x(\boldsymbol{r}, E)\phi(\boldsymbol{r}, E)dE}{\int_{E_g}^{E_g-1} \phi(E)dE}, \tag{2.10}$$

where Σ_x represents the cross section of interest. It should be noted that the correct weighting factor for a quantity may be something other than ϕ. For an in-depth exploration see [4].

With Equation 2.10, it is possible to produce group-wise cross sections for all of the cross sections present in Equation 2.9. However, a major difficulty with Equation 2.10 is that it requires the neutron flux, ϕ, which is a primary solution of the transport equation we seek. Traditionally, resonance self-shielding calculations resolve this interdependence [7] on simpler geometries than the full reactor. However, modern techniques can now make use of full reactor Monte Carlo (MC) for multi-group cross section calculation [4]. This thesis uses MC for computation of group cross sections.

Using this multi-group approximation for cross sections, the multi-group form of Equation 2.9 can be written as,

$$\psi_g(s, \boldsymbol{\Omega}) = \psi_g(\boldsymbol{r_0}, \boldsymbol{\Omega})e^{-\int_0^s \Sigma_{t,g}(s')ds'} + \int_0^s Q_g(s'', \boldsymbol{\Omega})e^{-\int_0^{s''} \Sigma_{t,g}(s')ds'} ds'', \tag{2.11}$$

where Q_g is defined as:

$$Q_g(\boldsymbol{r}) = \frac{1}{4\pi}\left(\sum_{g'}^{G}\Sigma_{s,g'\to g}(\boldsymbol{r})\phi_{g'}(\boldsymbol{r}) + \frac{\chi_g(\boldsymbol{r})}{k_{\text{eff}}}\sum_{g'}^{G}\nu\Sigma_{f,g'}(\boldsymbol{r})\phi_{g'}(\boldsymbol{r})\right). \tag{2.12}$$

Here, $\Sigma_{s,g'\to g}$ is the scattering from energy group g' to group g.

2.3 ANGULAR QUADRATURE

It is important to note that there has been a recent development in the use of stochastic angular quadrature. The Tramm Random Ray Method (TRRM) [8] integrates angular flux across tracks chosen randomly within the domain. One significant advantage of TRRM is reduced memory usage when compared to deterministic track laydowns. `MOCkingbird` can perform solves using TRRM, but this thesis focuses on using traditional deterministic tracks.

To enable computation of Equation 2.12, the continuous integral used in the definition of the scalar flux in Equation 2.3 can be discretized using numerical quadrature. Numerical quadratures approximate integrals by summing the product of weights and evaluations of the integrand at points within the integration bounds,

$$\int_a^b f(x)dx \approx \sum_{\text{qp}} \omega_{\text{qp}} f(x_{\text{qp}}), \tag{2.13}$$

where qp is an index over the quadrature points, ω_{qp} are the weights associated with each quadrature point and x_{qp} is the position of each quadrature point. This idea is very general, allowing computers to efficiently and accurately calculate numerical integrals.

Many different families of quadrature rules exist that are tailored to specific solution methodologies. This study utilizes equal spacing azimuthal quadrature with 3-angle TY polar quadrature [9] in 2D and equal weight polar quadrature in 3D. The scalar flux in Equation 2.12 can be discretized using both an azimuthal (2D plane) and polar (out of plane) quadrature as,

$$\phi_g(\boldsymbol{r}) = \sum_m \sum_p \omega_m \omega_p \psi_g(\boldsymbol{r}, \Omega_{m,p}), \tag{2.14}$$

where m and p are the indices for the azimuthal and polar quadratures, respectively.

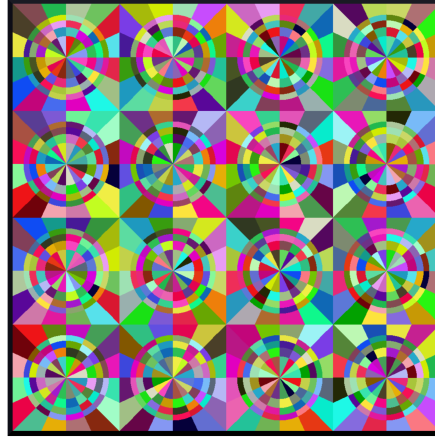


Figure 2.2. An example of a traditional flat source region treatment of a 4x4, pin-cell lattice. From [2]

2.4 SPATIAL DISCRETIZATION

Many terms within Equations 2.11, 2.12, 2.14 are still defined in terms of continuously varying points in space. These include the cross sections (Σ_x), fission spectrum (χ) and scalar flux (ϕ), that all need to be discretized in space.

Traditionally, spatial discretization for MOC has been accomplished by creating regions, called flat source regions (FSRs), within the domain, as depicted by the various colored regions in Figure 2.2. Within these regions, the cross sections, fission spectrum and average scalar flux ($\overline{\phi}_g$) are approximated as spatially constant. With this approximation, the total source, Q_g, is constant within each region (i),

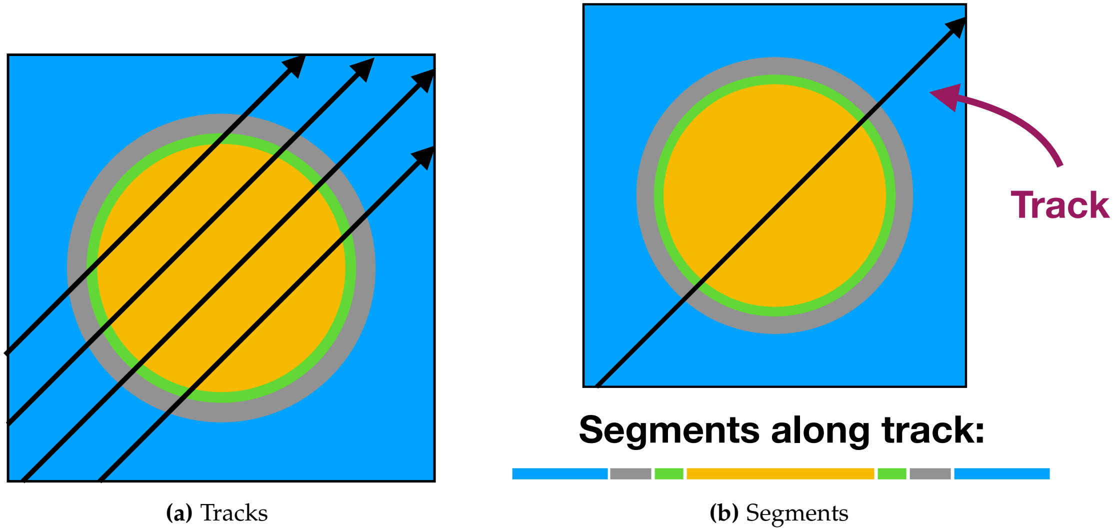
$$Q_{i,g} = \frac{1}{4\pi}\left(\sum_{g'}\Sigma_{s,i,g'\rightarrow g}\overline{\phi}_{i,g'} + \frac{\chi_{i,g}}{k_{\text{eff}}}\sum_{g'}\nu\Sigma_{f,i,g'}\overline{\phi}_{i,g'}\right). \tag{2.15}$$

where i represents an individual FSR, and

$$\overline{\phi}_{i,g} = \frac{\int_{V_i}\phi_g(\boldsymbol{r})d\boldsymbol{r}}{V_i}, \tag{2.16}$$

is the volume-averaged scalar flux, found by integrating the scalar flux in Equation 2.3 over the volume of one FSR (V_i).

The choice of the size and distribution of FSRs has consequences for the accuracy of the simulation. If the FSRs are large where the source is highly variant, then increased error occurs in the calculations. In `MOCkingbird`, the FSRs are constructed from elements of a finite-element mesh representing the reactor geometry. A more accurate method, which approximates the source as linearly varying within an FSR [10], can allow for the use of a much coarser mesh. Future work will address this extension.



(a) Tracks

(b) Segments

Figure 2.3. Example equally spaced tracks and the segments that would lie along one such track.

With these values being constant in an FSR, the 1D integrals in Equation 2.11 can be exactly computed for a segment of the track which crosses a single FSR from s to s'. By defining the optical length to be $\tau_{i,g} = \Sigma_{t,i,g}(s' - s)$, Equation 2.11 becomes:

$$\psi_g(s', \mathbf{\Omega}) = \psi_g(s, \mathbf{\Omega})e^{-\tau_{i,g}} + \frac{Q_{i,g}}{\Sigma_{t,i,g}}(1 - e^{-\tau_{i,g}}). \tag{2.17}$$

It is convenient to recast this equation in terms of the change in the angular flux across a segment:

$$\Delta\psi_{i,g}(\mathbf{\Omega}) = \left(\psi_g(s, \mathbf{\Omega}) - \frac{Q_{i,g}}{\Sigma_{t,i,g}}\right)(1 - e^{-\tau_{i,g}}). \tag{2.18}$$

This equation prescribes how the angular flux will change after it crosses each of the FSRs within the mesh. With a known starting angular flux at the outer domain boundary, the angular flux can now be integrated/propagated completely across the domain along one track.

2.5 TRACKS, SEGMENTATION, AND RAY TRACING

The final computation left to determine is the scalar flux in each FSR. This is accomplished by numerically approximating the integral and volume (V_i) in Equation 2.16 using the tracks crossing each FSR_i.

It is necessary to have a sufficient number of tracks crossing each FSR to ensure accurate evaluation of the scalar flux. The set of tracks laid down across the entire geometry is referred to as a "lay down." Within each lay down, the tracks are indexed by k. Figure 2.3a, shows one such set of tracks, although a realistic track lay down has significantly more tracks crossing the geometry. The intersections of tracks with each successive FSR define segments, as shown in Figure 2.3b. These segments can then be used to integrate a portion of the angular flux into a tally of the scalar flux. Using the fact that each track has an assigned spacing/width (ω_k), essentially a quadrature weight in space, the average scalar flux (after integrating across all the tracks passing through an FSR) can be computed using:

$$\overline{\phi}_{i,g} = \frac{4\pi}{\Sigma_{t,i,g}} \left[Q_{i,g} + \frac{1}{V_i} \sum_k \sum_p \omega_m(k) \omega_p \omega_k \Delta\psi_{k,i,g,p} \right]. \tag{2.19}$$

Note that a particular track, k, is at a particular azimuthal angle, m, and thus defines which weight will be used.

It is also important to note that the volume of each FSR (V_i) is also computed using the spatial and angular quadrature from the tracks. That is, `MOCkingbird` utilizes the "as tracked" approximation of the FSR volumes instead of the true volume.

These equations are applied to all segments of track. This can be viewed as a ray-tracing step: rays are the tracks, they are "traced" from their starting point, on the domain boundary, across the FSRs.

Finding all the segments is termed "segmentation." This is traditionally carried out as a pre-processing step by MOC-based codes [2, 11], where the tracks are traced and segments are stored in memory for later use during the iterative solution scheme. However, the number of segments required in 3D calculations can overwhelm available computer resources; therefore, recent advancements have been made in "on-the-fly" segmentation [3]. In this mode, ray-tracing is performed each time the track is used to integrate the angular flux across the domain. This provides savings in memory [3], but also requires additional computational work.

With this last piece in place, it is possible then to see the development of an algorithm to solve for the angular flux, scalar flux, and eigenvalue. It is an iterative method: computing the angular flux by integrating over all tracks (called a "transport sweep"), accumulating the scalar flux, updating the eigenvalue, then repeating. That algorithm, termed "source iteration", will be explored in §2.7.

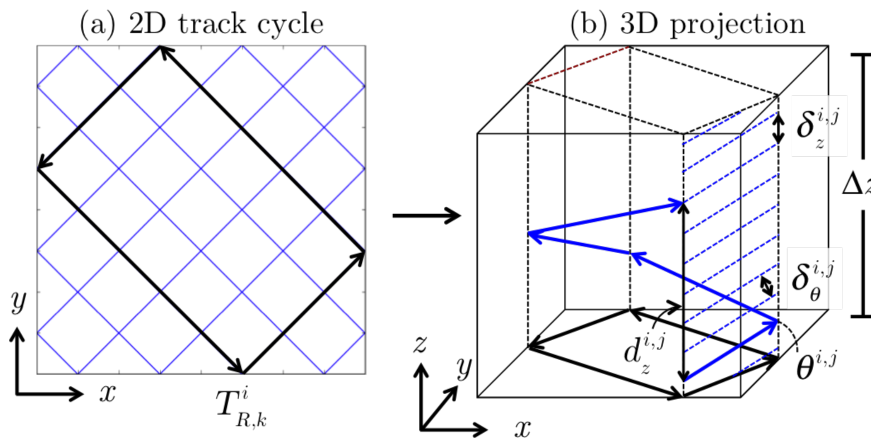


Figure 2.4. 2D and 3D cyclic tracking. The 3D tracks are defined as "stacks" above the 2D tracks. From [12].

2.6 CYCLIC TRACKING

The previous section explained how it is possible to compute the scalar flux by accumulating the integrations of the angular flux on segments across each FSR. Many different options exist for determining the tracks, including cyclic tracking [12], modular ray tracing [5], once-through [13] and back-projection [14]. The track laydown algorithm used by `MOCkingbird` is cyclic, global tracking as developed within `OpenMOC` [12].

Cyclic tracking creates track laydowns which form cycles through the domain. That is, starting at the origin of one track it is possible to move along all connected tracks in the cycle and arrive back at the starting position. Cyclic tracking is desirable for its ability to accurately represent reflected, periodic, or rotational boundary conditions. As can be seen in Figure 2.4, at the edges of the domain, each track meets the next track in the cycle. This allows for the angular flux to pass from one track to the next track. Without this feature, the incoming angular flux at the beginning of the tracks would have to be approximated or known. In full-core calculations, the incoming angular flux is often zero (vacuum), but having cyclic tracking capability makes the code much more flexible for the calculation of symmetric problems (e.g., pin-cells and assembly-level lattice calculations).

The cyclic tracking code utilized in `MOCkingbird` was initially developed for `OpenMOC` as detailed in [12]. As shown in Figure 2.4, the algorithms produce 2D tracks by specifying the number of azimuthal angles and the azimuthal spacing (space between parallel tracks). For 3D tracks, the 2D tracks are generated first. Then, "stacks" are made above each 2D track to create the 3D tracks. However, just as in segmentation, the storage of all of the 3D track information can be prohibitive for a full-core 3D calculation. The index of the 2D track uniquely identifies each 3D track it projects to,

the index of its polar angle and its index in the z-stack of tracks. This allows `MOCking-bird` to create a 3D track on-the-fly, enabling scalable, distributed track generation which is explored in §8.1.

The track generation capability within `OpenMOC` also computes the angular quadrature and track spacing weighting factors. Cyclic track generation is a complex undertaking due to the small adjustments needed in these factors [12].

2.7 SOURCE ITERATION AND TRANSPORT SWEEPS

With the equations defined and track generation specified, a solution algorithm can now be developed. The scheme solves for the angular fluxes along each track, scalar fluxes in each FSR and the eigenvalue which balances the system. The algorithm is called "source iteration" due to iterating between sweeping the angular flux across the tracks and updating the flat source by recomputing the scalar fluxes.

Algorithm 1: Source Iteration Algorithm

1. Estimate Scalar and Domain Boundary Angular Flux
2. Compute Initial Source
3. **while** *not converged* **do**
4. Normalize Scalar and Boundary Angular Fluxes
5. Compute New Source
6. Zero The FSR Scalar Flux
7. Zero The FSR Volumes
8. Transport Sweep - Accumulates scalar fluxes and volumes
9. Compute New k_{eff}
10. Check Convergence Criteria
11. **end**

An outline of the algorithm is in Algorithm 1. First, the fluxes are initialized, then the scalar fluxes are utilized to compute the initial source Q using Equation 2.15. The scalar and domain boundary angular fluxes are then normalized by dividing by the sum of the fission source:

$$F = \sum_i \sum_g \nu\Sigma_{i,f,g}\phi_{i,g}. \tag{2.20}$$

Normalization is necessary to keep the source values from continually increasing/decreasing (depending on the value of k_{eff}). Next, the source (Q) is computed using Equation 2.15.

The scalar flux and FSR volumes are then zeroed for accumulation during the transport sweep. The transport sweep itself is shown in Algorithm 2. Each segment of each

track is iterated over, Equations 2.18 and 2.19 are applied to update the angular flux and accumulate the scalar flux, respectively.

Algorithm 2: Transport sweep. The k tracks are iterated over, intersecting them with the geometry. Equation 2.18 is used to update the angular flux and Equation 2.19 to accumulate the new scalar flux.

```
1  foreach track in tracks do
2      foreach segment in track do
3          foreach polar angle do
4              foreach group do
5                  Update Angular Flux
6                  Accumulate Scalar Flux
7              end
8              Accumulate Volume
9          end
10     end
11 end
```

After the transport sweep, the eigenvalue is updated using the ratio of the old and new fission sources (as computed using Equation 2.20),

$$k_{\text{eff, new}} = k_{\text{eff, old}} * \frac{F_{\texttt{new}}}{F_{\texttt{old}}}, \tag{2.21}$$

and then convergence criteria are checked. For this thesis, the convergence criteria used is based on the root mean squared (RMS) change of the element fission source:

$$F_{RMS} = \sqrt{\frac{\sum_{\texttt{Fissionable } i} \left(\frac{\sum_g \Sigma_{f,i,g}\phi_{i,g,\texttt{new}} - \sum_g \Sigma_{f,i,g}\phi_{i,g,\texttt{old}}}{\sum_g \Sigma_{f,i,g}\phi_{i,g,\texttt{old}}} \right)^2}{N_{\texttt{Fissionable}}}}. \tag{2.22}$$

Where "`Fissionable`" refers to FSRs which contain fissionable material and N is the number of fissionable elements. The change in k_{eff} is also monitored:

$$\Delta k = \left| k_{\text{eff, new}} - k_{\text{eff, old}} \right|. \tag{2.23}$$

Once F_{RMS} or Δk is small enough (typically 10^{-4} and 10^{-6}), the iteration is terminated. It should be noted that these conditions are sometimes referred to as "stopping criteria" rather than "convergence criteria".

The scalar fluxes in each FSR are solved for using Algorithm1. In addition, the k_{eff} which balances the system is obtained.

2.8 transport correction and stabilization

The source iteration outlined above has been used for many years without issue. However, transport-corrected cross sections have recently been shown sometimes to cause this iteration scheme to diverge [15]. The transport correction is a modification to the cross sections to take first-order anisotropic effects into account. The development of this correction is outside the scope of this thesis, for an in-depth treatment see [16]. Using $\Delta\Sigma_{tr,g}$ as the correction, it is applied through modification of the total and scattering cross sections:

$$\Sigma_{tr,g} = \Sigma_{t,g} - \Delta\Sigma_{tr,g} \tag{2.24}$$

$$\tilde{\Sigma}_{s,g'\to g} = \Sigma_{s,g'\to g} - \Delta\Sigma_{tr,g}\delta_{g',g}. \tag{2.25}$$

It is important to note the Kronecker delta function in Equation 2.25. The action of that delta function is to only apply the transport correction to the in-group scatter cross section (diagonal of the scattering matrix). The transport correction is straightforward to apply through the use of these modified cross sections in place of the total and scattering cross sections in an MOC code.

However, as mentioned at the beginning of this section, making these cross section modifications can cause the source iteration scheme to become unstable. For an in-depth analysis of why this is the case see [3, 15]. A simplified explanation is that subtracting from the diagonal of the scattering matrix can lead to a system that is not diagonally dominant, and therefore, the iterative solution scheme is not guaranteed to converge.

A solution for stabilizing transport-corrected source iteration solvers was developed in [15]. The amelioration takes the form of "damping" the iterative scheme by selectively combining the newly obtained scalar flux with the previous (old) scalar flux. This is achieved by defining a damping factor,

$$D_{i,g} = \begin{cases} \frac{-\rho\tilde{\Sigma}_{s,i,g\to g}}{\Sigma_{tr,i,g}}, & \text{for } \Sigma_{s,i,g\to g} < 0 \\ 0, & \text{otherwise,} \end{cases} \tag{2.26}$$

where ρ is a positive scalar which controls the amount of damping to apply for FSRs that have negative in-group scattering cross sections. Larger values of ρ lead to more damping. While the damping is sometimes necessary for convergence, too much damping will also slow down the convergence rate [3].

The application of this damping factor occurs immediately after the transport sweep. The new scalar fluxes ($\phi_{i,g,\texttt{new}}$) are combined with the scalar fluxes from the previous iteration ($\phi_{i,g,\texttt{old}}$) via a weighted average:

$$\phi_{i,g} = \frac{\phi_{i,g,\texttt{new}} + D_{i,g}\phi_{i,g,\texttt{old}}}{1 + D_{i,g}}. \tag{2.27}$$

The new k_{eff} is computed using this damped flux.

To stabilize Algorithm 1, Equation 2.27 is applied directly after the transport sweep. However, the damping is only applied if ρ is specified to be a positive number, thus only being used when problems require it, such as the BEAVRS benchmark in §9.4.

2.9 PARALLELISM AND DOMAIN DECOMPOSITION

MOC is a highly parallelizable algorithm. The majority of the computational effort is located within the transport sweep, and each independent track can be swept simultaneously. Many different MOC-based codes have implemented some level of parallelism [3, 4, 5, 17].

Two types of parallelism that are ubiquitous within high-performance computing today are shared memory and distributed memory. Shared memory parallelism is implemented within a single node/computer where all tasks can access the same memory. The programming models often used for shared memory parallelism are threading [18] and OpenMP [19]. OpenMP, in particular, has seen use in many MOC codes [2, 3, 4, 5, 17, 20].

Distributed memory parallelism is required when more than one computer/node is used, typically in a cluster. In this case, network communication is needed to transfer data from one computer to the next. The programming model typically employed is the Message Passing Interface (MPI) [21], which is covered in detail in §3.2. In MPI, many separate processes (called "ranks") communicate via sending and receiving messages.

A natural way to use a distributed memory cluster is to distribute the reactor geometry among the MPI ranks. During a transport sweep, each rank is responsible for segment integration across tracks which intersect the assigned portion of the domain. Separating the domain is known as "partitioning," and the individual sub-domains assigned to each MPI rank are called "partitions" (Note: in many MOC studies partitions are also called "domains"). In this thesis, "domain" refers to the full geometry.

Spatial domain decomposition (SDD) [22] is a common method used for partitioning reactor geometries [3, 5]. As shown in Figure 2.5, SDD, for LWRs, is a Cartesian splitting of the geometry. For LWR geometries, this is often optimal as the partitions can be chosen to be entire assemblies. If the assumption is made that each partition

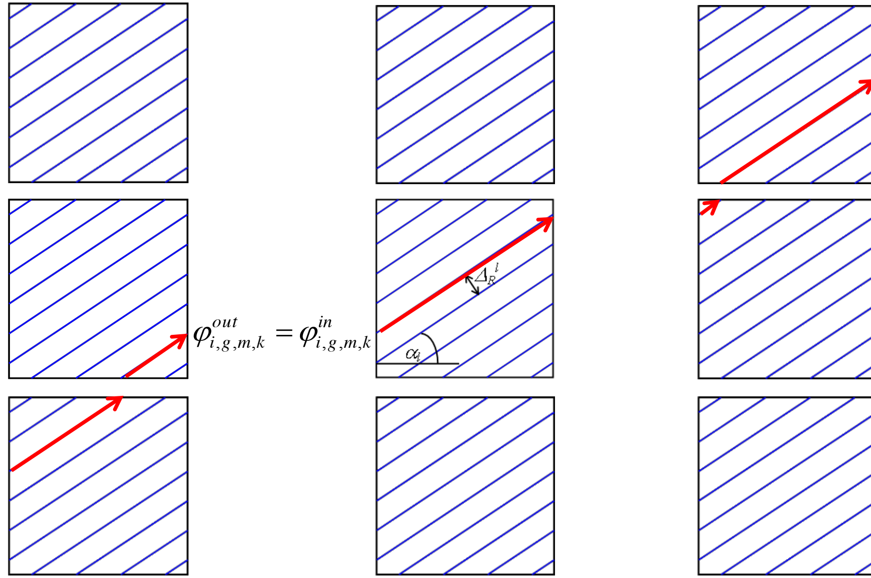


Figure 2.5. Modular ray tracing combined with spatial domain decomposition. The domain has been decomposed into 9 partitions. From [5]

has the same (or similar) geometry within it, then it is optimal to have square/cube partitions. This is due to trying to minimize communication (which scales with the surface area of the partitions) while maximizing work [5]. If all partitions are the same, then the amount of work scales with the volume of the partitions.

There are two restrictions imposed by the use of SDD. The first is that the number of MPI ranks used to solve the problem must be some multiple of the number of Cartesian partitions. In 3D, the requirement is to have a multiple of the number of partitions in the azimuthal plane. This restriction creates issues when trying to fit full-core problems into available compute resources.

Another restriction is that modular ray tracing (MRT) [5] is often employed with SDD. As shown in Figure 2.5, the rays in each partition are the same and meet each other at boundaries. This allows for a simplified communication pattern with only nearest neighbors. Each partition is treated as coupled, individual reactor physics problems when using modular ray tracing. Within each source iteration, the incoming angular flux is set, and all rays are traced from one edge of the partition boundary to the next partition boundary. At the end of each sweep, the boundary fluxes are communicated to neighboring partitions.

The outcome of this process is that angular flux data only moves across one partition during each source iteration. In Figure 2.5, the track in red takes five source iterations for the origin domain boundary angular flux to reach the opposite boundary. This can degrade the convergence rate of the algorithm [3]. In [3], on a test problem involving a single assembly, the effect was found to be small until many axial domains were

used. However, in that test problem, only 1 radial domain was utilized. In a full-core calculation, there are both many radial partitions and axial partitions, which may further degrade the convergence rate.

`MOCkingbird` does not utilize either SDD nor MRT and hence, is not subject to these limitations. Instead, finite-element partitioning processes are used to allow decomposition into any number of partitions. Ray-tracing in `MOCkingbird` will use "true long characteristics": the angular flux will be swept from domain boundary to domain boundary in each transport sweep. In parallel, the integration across a track will be swept across many partitions to reach from one side of the domain to another within one iteration. Therefore, `MOCkingbird` will achieve the exact same convergence behavior in parallel as in serial.

2.10 MEMORY

Full-core reactor 3D simulations generate large problem sizes. For the 3D BEAVRS benchmark calculation which will be presented in §9.5, there are $1.4e9$ flat source regions, $300e6$ individual tracks and 70 energy groups. The storage of the scalar fluxes and the domain boundary angular fluxes is approximately $140e9$ double-precision floating-point numbers or roughly 1TB of RAM. Therefore, it is important to consider how parallelization affects memory consumption. Ideally, as the problem is decomposed into smaller partitions, the total memory used is roughly constant (i.e., it would be perfectly split among the MPI ranks). However, with MRT, this is not the case.

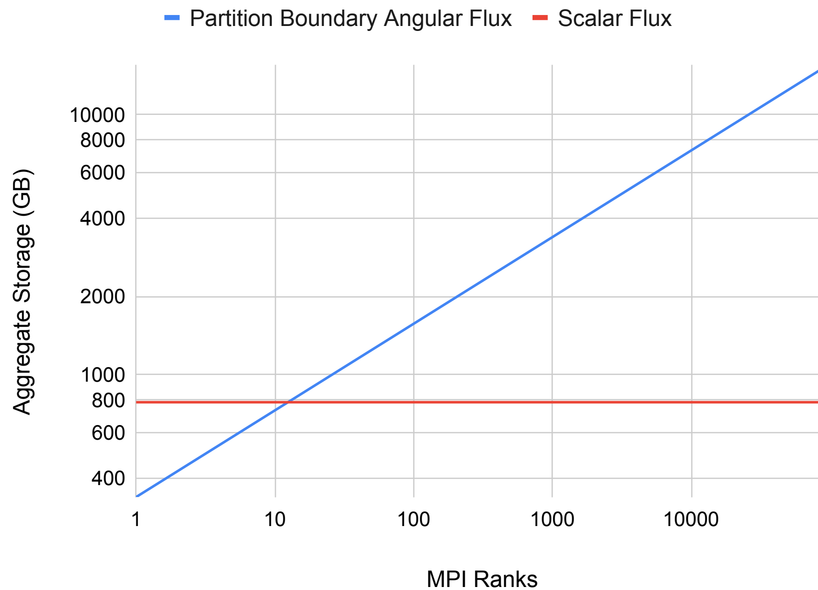


Figure 2.6. Growth of angular flux storage vs scalar flux storage when using modular ray tracing. Computed using partitioning into cubes of MPI ranks. From [5]

When using MRT, the angular flux is often stored at both the beginning and the end of each track on the partition. Therefore, as the number of partitions grows, so does the memory usage. To compute an estimate of the total memory that would be used by MRT for the 3D BEAVRS problem, a few assumptions are made:

- Domain size is 360cm x 360cm x 460cm
- 32 azimuthal angles
- 4 polar angles
- Tracks are isotropic (a reasonable approximation)
- Domain is partitioned using cubic numbers of MPI ranks

Using these assumptions, the graph in Figure 2.6 is produced, showing the amount of memory used to store the partition boundary angular fluxes. In that data, angular flux storage goes from 340GB in serial to 9.1TB when using 19683 MPI ranks. With that many MPI ranks, the memory usage is almost exactly 10x as much as in serial (once 784GB for scalar fluxes is added). This represents an issue for usability on clusters with small amounts of memory per CPU (an increasingly common trend). For instance, on the Mira supercomputer at Argonne National Laboratory [23], each processor has access to 1GB of RAM. At nearly 20k MPI ranks that would mean the angular flux storage would be taking up half of the available RAM per CPU. Depending on other

storage needs (cross sections, neutron source, geometry), the problem may not fit on that machine.

It should be noted that the use of on-node, shared memory parallelism (such as using `openmp`) can somewhat ameliorate this effect. If using only one MPI rank per node, the number of partitions is divided by the number of on-node cores.

In contrast, `MOCkingbird` only stores angular flux on the domain boundaries. Therefore, there is no increase in angular flux storage as the problem is decomposed into smaller partitions. This will be explored later in §8.3.1.1.

3 COMPUTING BACKGROUND

The computing landscape is in a constant state of change. New processors, network interconnect, memory, and GPUs combine to make high-performance computing an ever-moving target. What follows is a basic introduction to the computing terms and capability used throughout this thesis. These are meant to be high-level descriptions with many of the details available in citations.

3.1 CLUSTER COMPUTING

Early supercomputer designs [24] utilized many processors, often with wide vector units, operating within a single system image. With these designs, all processors worked within shared memory. While some remnants of this type of supercomputer persisted into the late '90s [25] and even early 2000s [26], by the turn of the millennium the idea of using "clusters" of computers to carry out parallel workloads was taking over [27, 28]. Cluster computing, where multiple separate computers linked over a network work together, is the dominant form of high-performance computing today.

Modern cluster computing utilizes designs similar to the one shown in Figure 3.1. Many (hundreds or even thousands) individual computers, called "nodes," are linked together by a high-speed network. Each of these nodes contains multiple central processing units (CPUs, "processors") with multiple "cores", which can all address the "shared" memory on the node. These configurations are often referred to as "distributed" memory due to the fact that the CPUs within each node can only directly address the memory within that node. This detail presents challenges for parallel algorithms. In modern designs, there may be more organization than this within the node such as multiple memory zones [29], multiple processor packages which each contain multiple CPUs, and multiple levels of cache memory. However, for this thesis, nodes are viewed as a collection of processors and one shared memory pool.

The network, or "interconnect," can have a significant impact on the efficacy of a cluster. Most notably, the "speed" of the network (typically in gigabits/second) is important with modern interconnects having speeds of 50 Gb/s [30]. In addition to raw speed, latency (i.e., how long a signal takes to reach its destination) is also an important consideration. Impacting both speed and latency is network "topology": the network shape as determined by the links and switches. There are many choices for network topology with "fat tree" [31] and "hypercube" [32] being two of the most common. The differences between these two topologies is not critical to the current

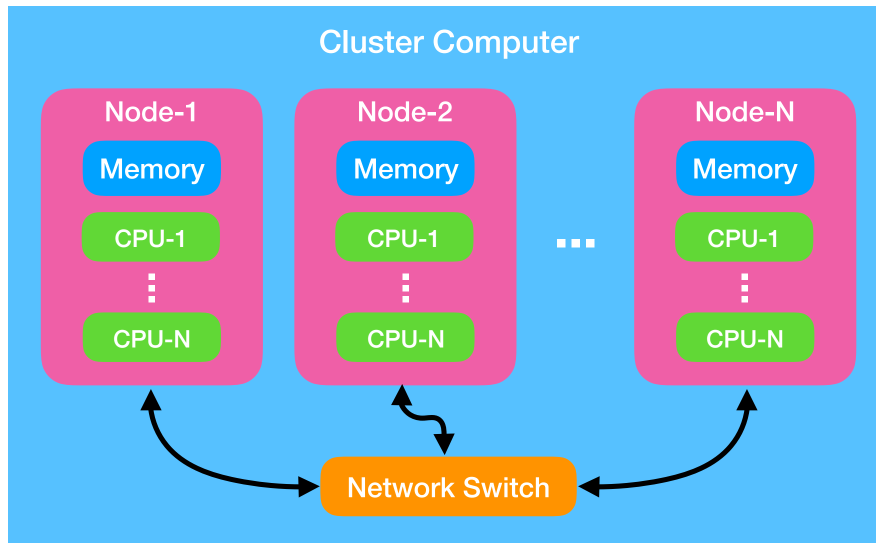


Figure 3.1. Pictorial of a modern cluster computer.

study. However, the results within this study utilized a cluster with a fat-tree network architecture as detailed in §3.1.1.

One increasingly important aspect of cluster networks is the capability for hardware "offload" of network communication. Offloading network communication to a hardware unit allows CPUs to continue to do tasks while network communication is handled for them. For instance, the BlueGene/Q supercomputer design contained an extra processor dedicated to accelerating network communication [33]. The Infiniband [30] interconnect provides an offloading capability for sends, receives, and collective operations [34]. The Infiniband Host Channel Adapter (HCA) can directly read and write memory both locally and on remote nodes through Remote Direct Memory Access (RDMA). Messaging systems such as MPI [21] can utilize these capabilities to allow the network to complete messages while CPUs continue to process data, allowing overlapping of communication and computation. This thesis makes extensive use of network communication offload to overlap communication and computation, achieving sustained scalability.

3.1.1 *Lemhi Supercomputer*

All of the results presented in this work were developed using the Lemhi Supercomputer at Idaho National Laboratory. Lemhi has 504 nodes, each with two processors containing 20 cores each giving a total of 20,160 cores. Each node is outfitted with 186GB of RAM and connected to the network using Intel OmniPath interconnect. Intel OmniPath is a high-speed interconnect providing low latency, 40 Gb/s link speed in a fat-tree configuration. Lemhi has a LINPACK [35] rating of 1 Petaflop/s.

3.2 MESSAGE PASSING INTERFACE

Over the last 40 years, there have been many approaches to distributed-memory parallel computing. One approach is that of "message passing," where the network is utilized to send data from one node to another in discrete "messages." Some of the earliest attempts at defining message passing interfaces [36] still bare many resemblances to the modern protocols defined within the newest Message Passing Interface (MPI) specifications [37]. The MPI specification has continued to evolve, with multiple projects dedicated to creating implementations of MPI. Three open-source MPI implementations are OpenMPI [38], MPICH [39] and MVAPICH [40]. MVAPICH, based on MPICH, is specialized to target computers using Infiniband-like interconnects and is used exclusively within this thesis.

MPI works by starting multiple instances of an application binary simultaneously. Each of these instances can be called an MPI "process" or an MPI "rank." All of the MPI processes started together can send and receive messages with each other by calling functions from within MPI. This is true regardless of whether the MPI processes are started on the same computer/node or separate nodes linked within a cluster.

It is important to distinguish between "process" and "processor." A modern operating system has many processes running simultaneously: word processors, spreadsheets, terminal applications, web browsers, etc. Any number of processes can be active at the same time - no matter how many processors (CPUs) the machine contains. However, if multiple processes are attempting to utilize the same cores within a CPU simultaneously, it causes a slowdown as the processes have to wait and take turns.

MPI processes are no different from these other processes. Any number of them can be running simultaneously on a node in the cluster, regardless of the number of CPUs present within that node. However, because scientific applications heavily utilize CPUs, it rarely makes sense to "oversubscribe" (create more MPI processes than the number of CPUs present within a node) MPI processes on nodes.

The communication methods within MPI can be loosely grouped into two major sets: point-to-point and collective operations. Point-to-point operations involve individual ranks sending and receiving information while collective operations typically involve all ranks (or large amounts of ranks) sending and receiving. These are further broken down into routines which are "blocking" (execution does not continue until the routine finishes) and those which are "non-blocking" (execution can continue while the routine executes in the background, also referred to as "asynchronous"). A few of the message passing routines available are:

- `MPI_Receive`: Blocking receive
- `MPI_Irecv`: Non-blocking receive

- `MPI_Send`: Blocking send
- `MPI_Isend`: Non-blocking send
- `MPI_Issend`: Non-blocking synchronous send (sender when receive started)
- `MPI_Test`: Test if non-blocking routine has finished
- `MPI_Probe`: Wait for incoming message
- `MPI_Iprobe`: Non-blocking check for incoming message
- `MPI_Alltoall`: Collective where every process sends to every other
- `MPI_Ibarrier`: Non-blocking collective routine to check if all processes have reached a point in the program
- `MPI_Iallreduce`: Non-blocking collective where an operation is performed on the data and every process receives the outcome of that operation. For instance: summation.

Until recently, most message passing was accomplished using blocking methods: that is, when a process began a messaging operation, the process would wait until the operation was complete before allowing the program to continue. In the case of point-to-point communications, this would mean both the sending and receiving ranks would need to "post" (start) the requisite send/receive routines and wait for them to complete before either of them could continue processing. In the case of collective operations (such as an `MPI_Alltoall`), this would cause all processes to hold until the collective operation was complete.

The major side-effect of this type of message passing is that a lot of computational potential can be "wasted" waiting for communication routines to start and finish. While communication of data across the network has its own intrinsic time, it adds to the calculation; this "lag" in waiting for communication to start and finish can often be even more dominant - leading to poor parallel efficiency.

Utilizing blocking message passing routines can cause inefficiency. MPI ranks, which reach communication points earlier than other ranks, need to wait. Time spent waiting is lost time that could be used for computing. In addition, blocking routines take time to set up the communication, adding to the amount of time lost. In particular, if one process has a lot more work to do than others, it can often be the case that all other ranks are waiting on one to finish before all of the communication can complete. This "load imbalance" is particularly prevalent when working with unstructured mesh and is explored in §8.3.3.

Newer message passing specifications such as MPI-3 allow for both point-to-point and collective operations to be "asynchronous" (or "non-blocking"). In this mode, processes are allowed to start a messaging operation and then continue without waiting for the operation to complete. Later, the process can then check the status of the operation (using `MPI_Test`) to see whether or not it is complete. This asynchronous communication can allow for significant speedups, especially in very unstructured parallel algorithms. By allowing communication to complete in the "background" while a process continues doing useful work, much of the latency and lag associated with the communication of data within the parallel execution can be hidden.

Hardware support is critical for asynchronous communication to be effective. If a process begins an asynchronous "send" operation, it is paramount that the computer/network can take control over the successful movement of the data to the destination process. Within a shared-memory node of a cluster, it is straightforward for one process to give another a message by merely advertising a memory address from which the process can pull information. Over the network of a distributed memory cluster, it is far less obvious how to achieve asynchronous communication. Some clusters have specialized hardware or even have dedicated processors just for asynchronous communication [33]. However, many modern clusters offload asynchronous communication to the network infrastructure itself, including the network cards. As mentioned earlier, this is the approach taken in Infiniband-based systems. In these systems, the network card within each node can take control over an asynchronous communication and see it through to completion while the process that started it continues doing other work.

This work heavily relies on MPI for all parallel communication. In particular, asynchronous communication routines are utilized, and hardware offload is capitalized on to achieve parallel scalability during the transport sweep for the method of characteristics. All simulations for this study were executed using MVAPICH 2.3 [40].

3.3 MESHES

A "mesh" is a data structure many numerical algorithms use as the description of the geometry. It is composed of "elements" which make up the volume/surface of the domain and "nodes" which link the elements together, typically at the vertices. By combining many elements, linked through the nodes, a mesh can span an entire computational domain. Numerical methods such as the finite-element, finite-volume, and finite-difference all utilize a mesh. In addition to describing the geometry, the mesh also often serves as the discretization of the volume.

Figure 3.2 shows three options for two-dimensional (2D) and three-dimensional (3D) elements. A node is located at the vertices of each of these elements. There are

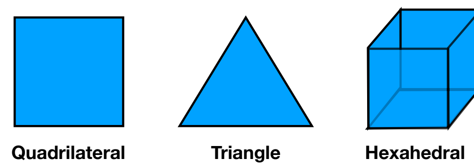
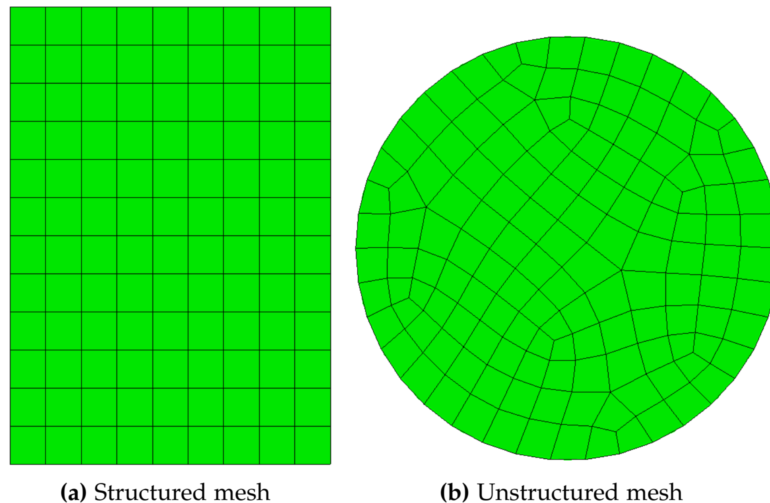


Figure 3.2. Examples of two- and three-dimensional elements



(a) Structured mesh **(b)** Unstructured mesh

Figure 3.3. Examples of structured and unstructured meshes.

many more options for elements, including lines, prisms, tetrahedrals, and pyramids. By using many of these elements (possibly skewed and stretched) linked together, it's possible to make nearly any shaped domain. This effort focuses on using quadrilateral elements for 2D and hexahedral elements for 3D.

There are two primary types of meshes: "structured" and "unstructured." Both types are visible in Figure 3.3. Structured meshes are also often referred to as "grids." Structured meshes excel at being compact in memory.

An unstructured mesh, as seen in Figure 3.3b, does not contain regularity. Instead of having indices that can directly refer to the elements, the elements are instead uniquely numbered. A data structure must be created, linking elements to nodes and elements to neighboring elements. An unstructured mesh is far more flexible than structured mesh at representing complex geometries. By allowing elements with arbitrary rotations and connectivity, nearly any shaped domain can be approximated. Also, it should be noted that structured meshes are a subset of unstructured meshes. That is, an unstructured mesh data structure can perfectly replicate the domain of a structured mesh (although typically with a memory overhead compared to utilizing a structured mesh directly).

In this thesis, unstructured mesh is utilized to provide the flexibility needed to discretize complicated geometries such as those found in nuclear reactors. However,

perfect circles/cylinders are not representable with Lagrange based linear and second-order elements used in finite-element meshes. Chapter 5 describes a method for generating pin-cell meshes which preserve the material volumes.

3.4 DOMAIN DECOMPOSITION

3.4.1 *Partitioning*

In parallel, it is natural to split the work by assigning portions of the mesh/domain to each MPI process. Ideally, this splitting should result in equal amounts of work assigned to each MPI process. Any imbalance in work results in a loss in parallel efficiency as some MPI processes finish their work early and CPU time is wasted. The mesh splitting procedure is known as "partitioning" and has been an important field of study since researchers began using distributed memory clusters [41]. Many different ways to partition a mesh have been studied including orthogonal decomposition, spectral/eigenvalue decomposition, multilevel (k-way), space-filling curves, and more advanced ideas such as hierarchical partitioning [42].

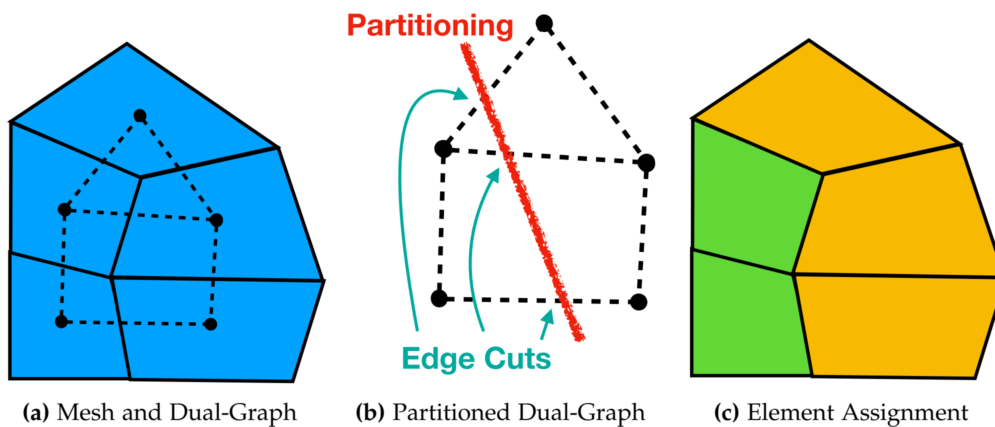
Of all of these types of partitioning strategies, the multilevel "k-way" based decomposition methods (possibly with a spectral option) have been the most popular [43, 44, 45, 46, 47]. To date, the most successful partitioning software is `METIS` [43] and its parallel offshoot: `ParMETIS` [44]. These two pieces of software have dominated finite-element mesh decomposition for over 20 years and are a good fit for the current effort.

Both versions of `METIS` take, as input, the connectivity or "dual-graph" describing the connectivity of the element. This graph is then used to solve for an optimal partitioning: one where work is balanced (over nodes) and communication is minimized (over edges). This process is shown in Figure 3.4. In MOC, the nodes of the dual-graph represent the work to be done on each element (ray-tracing and angular flux integration) and the memory utilization for each element (scalar fluxes and cross sections). The edges of the dual-graph represent the communication pathways between elements for angular flux/tracks. To best balance the mesh for MOC, a novel weighted partitioning scheme is developed in §8.3.3.

3.4.2 *Parallel Storage*

Within `MOCkingbird`, there are two possibilities for how the mesh is stored in parallel:

1. Replicated: each MPI process holds a full copy of the mesh
2. Distributed: each MPI process holds only the portion of the mesh assigned to it



(a) Mesh and Dual-Graph **(b)** Partitioned Dual-Graph **(c)** Element Assignment

Figure 3.4. The partitioning process. First, a dual-graph is made from the mesh. Next, the dual-graph is partitioned using partitioning software that tries to balance the work and minimize the number of edge cuts. Finally, the elements are assigned MPI ranks based on the partitioning.

The first option is the simplest; each MPI process has a full view of the geometry. However, this comes with a significant memory overhead. With a large mesh, keeping a full copy of the mesh within each MPI process may not be possible due to limited memory on the computational nodes.

To alleviate this issue, a "distributed" mesh only stores the assigned mesh partition within each MPI process. Assuming the partitioning process is working well (relatively equal number of elements on each process), distributed storage should allow for "memory scalability" (memory for each process decreases as the mesh is spread out over more processes).

`MOCkingbird` can use either "replicated" or "distributed" meshes. However, due to the massive memory inefficiency of "replicated," this thesis uses distributed mesh.

3.5 LIBMESH

The `libMesh` [48] open-source library provides the finite-element fundamentals necessary for this thesis. It was originally developed at the University of Texas in Austin as a tool for researching finite-element methods for fluid-flow applications [49]. The core of `libMesh` is the finite-element capability it contains. It provides a complete set of data structures for reading and writing meshes. In particular, it provides a `DistributedMesh` object which provides a domain-decomposed parallel mesh object that is utilized by `MOCkingbird`.

`libMesh` also provides interfaces to other open-source numerical software libraries such as `PETSc` [50] and `Hypre` [51]. Through these interfaces, `libMesh` simplifies the task of making a non-linear parallel finite-element solver. Developers using the library

can read in meshes, create fields on them to solve for and invoke parallel non-linear solvers to compute the solution. `libMesh` can then write the solution files out in several different formats, with ExodusII [52] used in this these.

In this work, `libMesh` provides the domain-decomposed mesh data structure necessary to hold the geometry. In addition to the mesh itself, it also provides many utilities for working with the mesh, including an oct-tree based point location method[53] which is used for locating the beginning point of tracks in §8.1. `libMesh` also manages parallel vectors created by `PETSc` for storing the various fields needed by the method of characteristics including total source, group scalar fluxes, total cross section. These parallel vectors are distributed across the MPI ranks, and `libMesh` manages the parallel indices.

3.6 MOOSE

Built using `libMesh`, the open-source Multiphysics Object-Oriented Simulation Environment (`MOOSE`) computational science framework [54] has been under development by Idaho National Laboratory and their partners since 2008. `MOOSE` provides a package for development of new multiphysics simulation tools. It is primarily focused on utilizing finite-element analysis (FEA) to solve fully-coupled systems of partial differential equations (PDEs).

`MOOSE` has been successfully utilized in many areas of science and engineering including reactor physics [55], nuclear fuel performance [56], geothermal [57], plasma-liquid interface [58] and computational fluid dynamics [59]. It has also been successfully applied to multiscale, multiphysics analysis [60]. The `MOOSE` framework can model any geometry that can be meshed and has a large array of options for coupling physics such as loose coupling or Picard. All of this capability makes `MOOSE` a good fit for a new reactor physics capability that combines the method of characteristics (MOC) with FEA to achieve a multiphysics simulation of a nuclear reactor.

`MOOSE` has previously been applied to the area of reactor multiphysics [61]. The `Rattlesnake` [55] neutron transport code developed by Idaho National Laboratory is built using `MOOSE` and has been successfully coupled to heat conduction, fluid flow, and solid mechanics [62]. `Rattlesnake` utilizes second-order S_n and P_n methods to solve neutron transport on unstructured mesh. Scalability of these methods has been a major research topic [63]. However, these methods suffer from increased memory usage both for storing angular flux degrees of freedom (DoFs) and creating matrices during the solve. `MOOSE` has proven to be a good platform for reactor multiphysics. Therefore, a neutron transport capability, such as MOC, that is not burdened with angular flux storage, or matrix creation, is a significant contribution to the `MOOSE` community.

3.6.1 *Scalability*

"Scalability" or "parallel efficiency" is a measure of how well a code can make use of more processing capability. `MOOSE` is scalable to thousands of cores when using FEA to solve systems of PDEs [64]. As with any parallel code, scalability deteriorates as the problem is spread over more MPI processes.

`MOCkingbird` has different scalability limits. However, this recommendation should be kept in mind due to the aforementioned goal of allowing multiphysics coupling.

3.7 OPENMOC

`OpenMOC` [2] is an open source, MOC-based neutron transport code primarily developed by the Computational Reactor Physics Group (CRPG) at MIT. The basis of `OpenMOC` is a cyclic-track-based, steady state, eigenvalue solver using constructive solid geometry (CSG) and accelerated by CMFD [65]. Many different areas of neutron transport have been studied using `OpenMOC` including: parallelization [66], acceleration [67], transients [68], cross section computation [4] and three-dimensional, full-core MOC [3].

`MOCkingbird` utilizes the cyclic-track generation capability from `OpenMOC` that can generate both two- and three-dimensional cyclic track "laydowns" (a set of cyclic tracks) [12]. Of particular importance is the fact that the track generation capability in `OpenMOC` can generate 3D tracks "on-the-fly," where the track information for any 3D track is programmatically generated without needing to hold all tracks in memory. This feature is critical for the full-core, 3D calculations `MOCkingbird` performs, and it plays a role in scalable track generation discussed in §8.1.

For this work, the track generation routines from `OpenMOC` were compiled directly into `MOCkingbird`.

4 LITERATURE REVIEW AND MOCKINGBIRD OVERVIEW

This chapter contrasts the current state-of-the-art capability for MOC on unstructured mesh with the capabilities developed in this thesis for `MOCkingbird`. The first section explores each of the existing implementations of MOC on unstructured mesh for reactor physics problems. Following the literature review, a high-level overview of `MOCkingbird` is presented.

4.1 LITERATURE REVIEW

4.1.1 *MOCFE / Proteus-MOCEX*

`MOCFE` is a MOC solver based on finite-element geometry developed at Argonne National Laboratory [69]. It uses MPI for parallelism and includes domain-decomposition. The original development [69] was capable of both 2D and 3D solutions.

`MOCFE` did not use cyclical tracking, instead, relying on approximations of boundary fluxes for reflective boundary conditions. Track generation utilized a back-projection method with trajectories started at domain boundaries, local partition boundaries, or from every element. The stated reason for this type of track generation was that locating track starting and ending positions within the mesh from cyclic tracking would be difficult. Track trajectories were generated in serial by the first MPI process and then handed out to the rest. This was noted to be a scalability issue going forward.

For ray-tracing, it utilized the Moller-Troumbore algorithm [70] for finding intersections with 3D faces. The segments would be traced up-front and stored, leading to significant memory use. During the eigenvalue solve it would use the true analytical volume of the finite-elements as a correction factor.

The solver casts MOC into a matrix form, solving the resulting matrix using GMRES [71]. The choice to use a matrix solver was made to sidestep the issue of multiply reentrant tracks on partition boundaries. However, one of the significant benefits of a traditional MOC solution technique is that it doesn't require a matrix, leading to less memory usage [69]. By casting the MOC problem into a matrix `MOCFE` incurs a prohibitive memory footprint [72] and scalability issues including load imbalance, difficulty in preconditioning and a substantial increase in degrees of freedom as the mesh is decomposed [71].

In addition to domain-decomposition, `MOCFE` also allows for decomposition in angle and energy. With multiple MPI processes working on the same spatial partitions (with different energies or angles), `MOCFE` requires global reductions to compute the scalar

and boundary fluxes. In [71] scalability is shown up to 65,536 MPI processes. However, that is only for the parallel GMRES solver and does not include global reductions. Where the entire MOC solver is benchmarked, scaling up to 4 or 16 cores is shown.

The developers of `MOCFE` also cited large load imbalance issues [71]. This is due to the domain decomposition placing an equal number of elements on each MPI rank, but this would not mean an equal number of intersections would be dispersed. Correct load balance was said to require additional research [71].

Due to "enormous memory and computational effort" [73] for 3D MOC it was ultimately abandoned as a separate project [72]. It was folded into `Proteus`; another neutron transport package developed at Argonne National Laboratory where it was used in 2D and 2D-1D schemes.

Within `Proteus` it has been further developed to allow for calculation on axially extruded geometries [73]. It was used in simulation of the TREAT reactor [73], although parallelism and scalability are unknown to this author.

4.1.2 *MOCUM*

MOC on Unstructured Mesh (`MOCUM`)[17] was developed at Purdue University as a 2D only MOC solver. It focused heavily on a CSG meshing capability to convert CSG geometry into triangles. This triangulation capability used a Delaunay triangulation algorithm which led to non-symmetric meshes that didn't preserve material volumes. Parallelism within `MOCUM` was based solely around OpenMP. Cyclic tracking wasn't used; instead, approximations were made for reflective boundary conditions.

4.1.3 *Linear Source On Meshes*

A research code developed by the Bhabha Atomic Research Centre in Mumbai [74] utilized a linear source approximation together with unstructured mesh to create a MOC solver. Similarly to `MOCUM` a triangulation was used to generate the geometry. Although they were able to show that using a linear source approximation allowed for fewer elements, it also led to a representation of the geometry with more approximation error. To better approximate the material volumes, the edges of triangles on the boundary of materials were turned into curves. It's unclear how ray-tracing was handled with curved edges, and no parallel implementation was discussed.

4.1.4 *MoCha-Foam*

`MoCha-Foam`[75] was developed using the `OpenFoam` open-source, PDE solver framework. The reason for using unstructured mesh and `OpenFoam` was to allow for simplified multiphysics solutions. The code has several limitations, including not being parallel, not using cyclic ray-tracing and having very slow ray-tracing. However, it did include a capability for anisotropic scattering.

4.2 MOCKINGBIRD

Creating a scalable, unstructured mesh, MOC reactor physics tool capable of arbitrary geometry, 3D, full-core simulation is to be problematic. This thesis seeks to overcome many of the obstacles encountered by the codes in the previous section. In particular, the following issues are addressed, with the relevant sections of this document shown in parenthesis:

- Lack of volume preserving meshing (§5)
- Non-symmetric meshing (§5)
- Serial track generation (§8.1)
- Lack of track starting point location (§8.1)
- Serial track distribution (§8.1)
- Lack of cyclic tracking (§2.6)
- Approximations for reflective boundary conditions (§8)
- Lack of domain decomposed parallelization (§7)
- Struggling with partition re-entrant ray-tracing (§7)
- Excessive memory usage from a matrix representations (§8)
- Load imbalances (§8.3.3)
- Lack of scalability (8.3)

Each of these issues is treated, in detail in the following chapters. This section provides an overview of `MOCkingbird` and serves as a guide for the rest of the document.

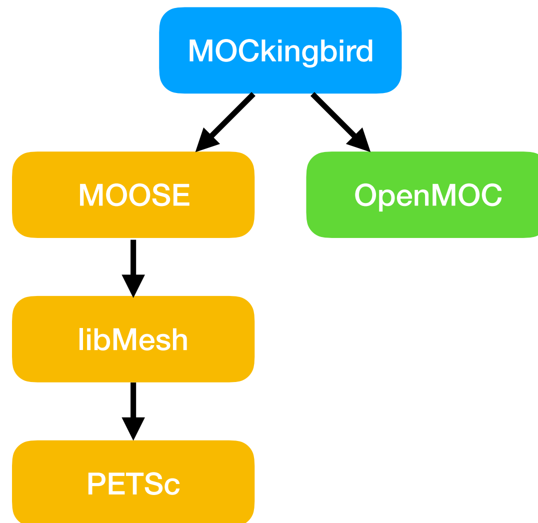


Figure 4.1. The relationship between MOCkingbird and libraries it uses. The orange libraries represent the set of libraries a MOOSE-based code would normally use. OpenMOC is set apart in green to show that it's added for MOCkingbird.

4.2.1 *Libraries Used*

MOCkingbird has been developed using several software libraries: MOOSE, libMesh, PETSc, and OpenMOC. Each of these libraries plays an important roll in providing functionality to MOCkingbird. Figure 4.1 shows the relationship of MOCkingbird and the libraries. What follows is a brief description of the functionality used by MOCkingbird from each of these codes.

The most-used feature of PETSc within MOCkingbird are parallel vectors for storage of scalar flux, neutron source, and total cross section. These parallel vectors are managed by libMesh so that they are decomposed across the MPI ranks with the mesh. Therefore, the storage of these values is scalable (the total memory used stays constant as the problem is decomposed).

libMesh, which was described in detail in §3.5, plays many roles. As mentioned, it manages the PETSc parallel vectors to distribute them with the mesh. libMesh also contains the mesh data structure itself and the objects that make up the mesh: elements and nodes. MOCkingbird can read and write many varied mesh formats due to capability in libMesh. libMesh also contains support for a wide array of finite-elements geometric types allowing MOCkingbird to be used with meshes containing 1D, 2D, and 3D elements. While currently MOCkingbird has only been used with line, quadrilateral and hexahedral elements, only a few lines of code would enable triangles, tetrahedrals, prisms, and pyramids to be used. These options were not developed due to the lack of need for this thesis. libMesh provides a uniform interface to elements, allowing the

ray-tracing code detailed in §7 to be general and re-usable regardless of the type of elements which exist in the mesh.

Other essential features of `libMesh` are the utilities for spatial search within a mesh. As mentioned previously, `libMesh` provides a quad/octree spatial search capability for locating the element containing a physical point in space. It also provides routines for testing whether a point is within a given element or near it. These routines are critical to solving the scalable, parallel track generation, and distribution problems, as detailed in §8.1.

Finally, `libMesh` also provides an expansive C++ interface to MPI. This allows for simplified parallel semantics within `MOCkingbird` for sending and receiving C++ objects and data.

`MOOSE` is a library for building multiphysics tools and, therefore, is a natural fit for a code that will ultimately perform multiphysics analysis of reactors. `MOOSE` also provides important infrastructure to `MOCkingbird` including:

- Input file reading: Flexible, human readable input
- Command-line parameter specification: Allows for parameter studies
- Mesh-file splitting: Off-line, parallel mesh decomposition
- Online postprocessing: Error calculations and more
- Auxiliary field calculations: Such as power or pin-power error
- Interface to partitioners: Allows for weighted partitioning
- Load balance metrics and visualization
- Parallel debugging assistance
- `GridPartitioner`: Perfect partitioning for lattices
- `PerfGraph`: Code timing
- `MultiApp` capability: For visualizing pin-power error
- `CircularBuffer`: Efficient work queue
- `DependencyResolver`: For directed acyclic graph mesh generation
- `SharedPool`: Pool for re-usable objects in memory
- `StaticallyAllocatedSet`: Efficient set container
- `RankMap`: Determines which MPI processes are on which nodes (and allows visualization of that)

MOOSE also provides the infrastructure for building and testing codes which use it. The testing system allows regression tests to be created for MOCkingbird. As development progresses, the tests are run to ensure existing functionality is not broken.

The only functionality MOCkingbird relies on from OpenMOC is also one of the most critical: track generation [12]. OpenMOC contains routines for efficient generation of cyclic 2D and 3D tracks. The ability to selectively generate 3D tracks based on a unique ID is critical to efficient track generation and claiming as described in §8.1. In addition, the track generation in OpenMOC also computes the necessary quadrature weights to use with the tracks for angular and spatial integration.

4.2.2 *Solution Methodology*

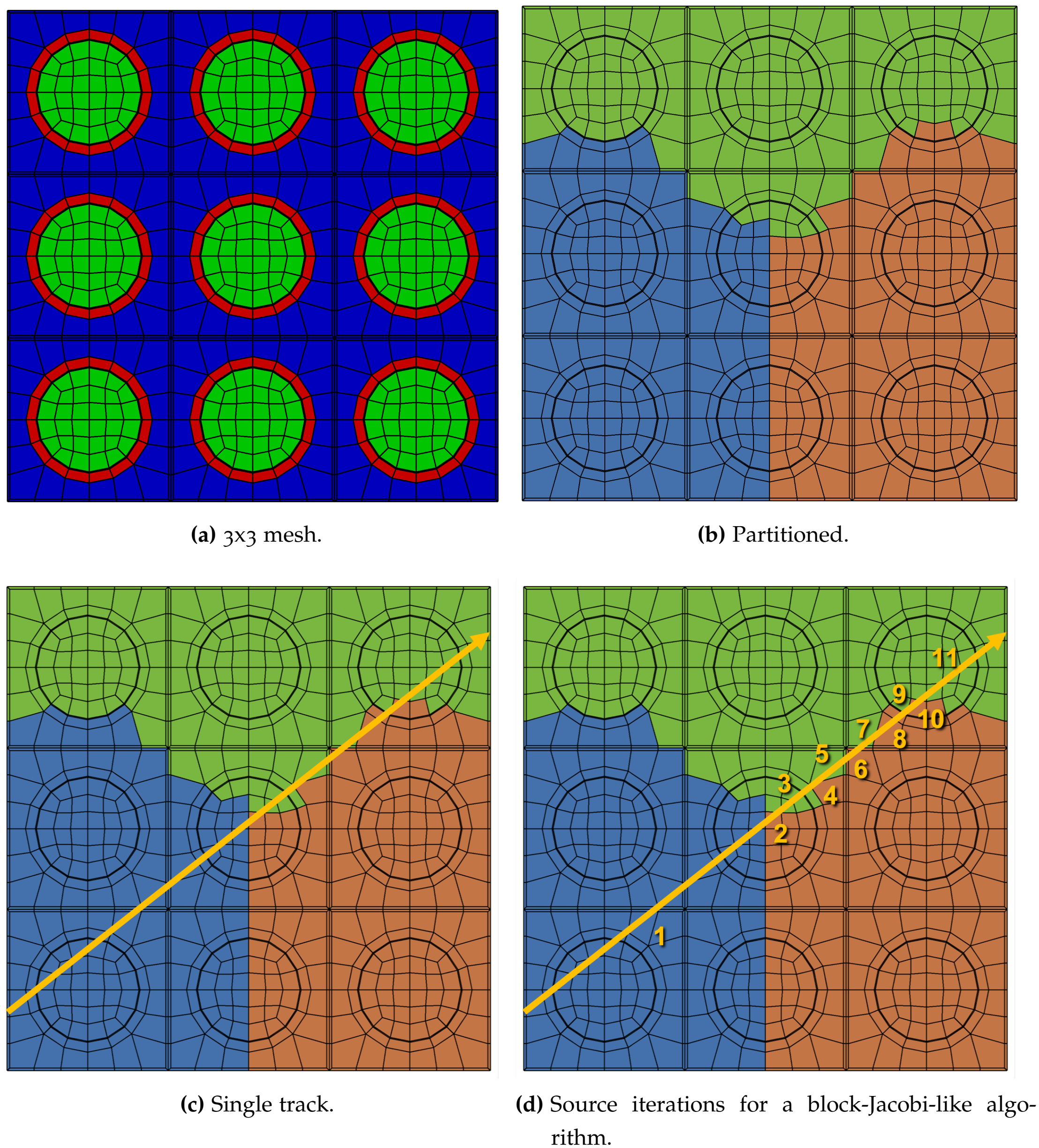
MOCkingbird uses a conventional MOC formulation. Long characteristic tracks (domain boundary to domain boundary) are used in both serial and parallel: providing the same solver behavior regardless of the number of MPI ranks used. MOCkingbird uses flat source, isotropic scattering, isotropic fission source, and multigroup cross section approximations just as many other codes. MOCkingbird does not contain cross section generation capability; all cross sections must be developed externally. However, a flexible interface exists to allow reading many different types of cross section databases.

Some interesting/unique aspects of MOCkingbird are:

- Unstructured mesh for geometry definition and spatial discretization
  - Completely general, no geometric assumptions made in the solver
  - Capable of working with moving mesh, e.g., from thermal expansion
- Parallel agnostic: same solution behavior in serial and parallel
- Parallel, scalable cyclic track generation, starting point location and track distribution: described as "Track Claiming" in §8.1
- Use of asynchronous sparse data exchange algorithms during problem setup
- Flexible boundary condition specification, including vacuum, and reflective
- Scalable communication routines during problem setup
- Efficient, robust element traversal
- Scalable, domain decomposed ray-tracing
- Smart buffering for messages

- Scalable memory usage
- Object pool utilization to reduce memory allocation and deallocation
- Weighted partitioning for load balance
- High parallel scalability (tested to over 18k cores)
- Developed using `MOOSE` for simplified linking to other physics codes

One aspect of `MOCkingbird`, which is critical to its efficacy, is asynchronous parallel ray-tracing which allows the integration of a complete track across the entire domain decomposed geometry (domain boundary to domain boundary) without any intermediate global synchronization or iterations. The algorithm that underpins this capability is the Scalable Massively Asynchronous Ray-tracing (SMART) algorithm and is explored in detail in Chapter 7.



(a) 3x3 mesh.

(b) Partitioned.

(c) Single track.

(d) Source iterations for a block-Jacobi-like algorithm.

Figure 4.2. The mesh is split for 3 MPI ranks, and one track is considered. With a block-Jacobi-like algorithm, the boundary angular flux would only propagate to the other side of the domain after 11 source iterations. In `MOCkingbird` the entire track is traced in one iteration, without any intermediate global synchronization.

As mentioned in 2.9, many MOC codes employ modular ray tracing (MRT) and modular spatial domain decomposition (SDD) [3, 5] in parallel. When using a modular decomposition, tracks are only integrated from one partition boundary to another within one source iteration, creating a block-Jacobi-like method. As noted in [71], this idea is untenable for arbitrarily decomposed unstructured mesh which is partitioned with a mesh partitioner. Figure 4.2 shows why this is the case. With unstructured mesh

partitioning, there are many jagged/re-entrant corners along partition boundaries. If a track were to pass through the domain in such a way that it follows a partition boundary, then a block-Jacobi-like tracing of that track would take many source iterations before boundary information is propagated across the domain [3]. In a large 3D mesh for a full reactor core with millions of tracks, this situation frequently occurs.

The unstructured MOC solver detailed in [71] attempted to solve this problem by building a matrix for MOC and adding all of the partition boundary degrees of freedom to it. Then a Krylov solver was used to solve the resulting system. This proved not to be viable, causing massive memory usage [72].

However, with the asynchronous, parallel, multi-hop algorithms developed in Chapters 6, 7 `MOCkingbird` can efficiently trace the track shown in Figure 4.2c in one source iteration without any intermediate global synchronization. This provides multiple advantages:

- Same behavior in parallel and serial
- No storage of partition angular fluxes required (therefore scalable in memory)
- No reduction in convergence rate as observed in [3]

As shown in [5], it is critical for MOC work to be distributed evenly. Therefore, `MOCkingbird` utilizes a weighted partitioning scheme developed in §8.3.3. The goal of weighted partitioning is to reduce communication and balance work by ensuring the same number of intersections occur within each partition.

Using this asynchronous communication algorithm together with weighted partitioning, `MOCkingbird` achieves excellent parallel scalability. This is explored in §8.3 for a set of simplified problems and then revisited for each of the benchmarks presented in Chapter 9.

4.2.3 *Executing* `MOCkingbird`

The first step to using `MOCkingbird` is discretizing the reactor geometry using unstructured mesh. As mentioned above, generating volume-preserving, symmetric mesh for reactor geometries is essential and is covered in Chapter 5. Within the mesh, each material region is denoted using a block ID which can be set within a meshing utility or by using the `MeshGenerator` system detailed in 5.2.

Next, `MOCkingbird` requires a set of cross sections. `MOCkingbird` contains a flexible, plug-in system for defining new cross section data formats. The current studies use a simplified JSON [76] format for specifying the needed cross sections for each material region existing within the mesh. Small python conversion utilities were developed to

convert the cross sections for both the C5G7 and BEAVRS benchmarks explored in Chapter 9.

`MOCkingbird` utilizes the `MOOSE` input file syntax for specification of all problem parameters. The possible parameters include: mesh to be read or generated, track generation parameters, boundary conditions, cross sections, number of energy groups, number of iterations, convergence tolerances, and output formats. An example of this input file syntax for a 2D, fully-reflective pin-cell problem can be found in Appendix 11.

5 NUCLEAR REACTOR MESHING

While §3.3 provided an overview of the concept of a mesh, this chapter develops methods for generating meshes of nuclear reactor geometries. This process, termed "meshing," is critical to the viability of MOCkingbird as a reactor physics tool. The finite-element world has been working on general meshing capability for over 40 years [77], but has never been popular in the nuclear reactor physics world due to the lack of symmetry and conservation of volume or surface area. Today, there are several meshing tools to choose from, including both free [78] and commercial [79] packages. Many commercial finite-element tools also contain meshing capabilities [80, 81, 82]. libMesh (and therefore MOCkingbird) can read many mesh formats while also containing interfaces for creation and modification of meshes which are useful for reactor physics.

Reactor meshing, especially for light-water reactor (LWR) geometries, generally involves a few major pieces: the 2D pin-cells (Figure 5.1), assemblies, reflector region, and the pressure vessel. Each of these pieces of the reactor requires care in how it is meshed. There are three main features of nuclear reactor meshes which must be carefully considered:

- Volume/mass preservation
- Symmetry
- Mesh density/fineness

Volume preservation is critical for accurate eigenvalue solution. Symmetry is important both for obtaining accurate solutions in inherently symmetrical problems and allowing for reduced computational effort. Mesh density, within areas of rapidly changing source (such as in the moderator), plays a role in MOC solution fidelity.

This chapter describes the hybrid approach that has been utilized to mesh LWR geometries for MOCkingbird. Firstly, Cubit [79] is used to create symmetric, volume-preserving pin-cell meshes. These are then used within a graph-based mesh generation capability called the MeshGenerator system, which was added to MOOSE. Through this system, the pin-cells can be manipulated, combined, extruded, and added to, in order to get the final meshes needed for calculation.

This chapter proceeds by first describing the pin-cell generation process. Next, the MeshGenerator system is described, detailing how the process of making an LWR mesh can be cast into a directed acyclic graph (DAG) for flexible, memory-efficient mesh generation.

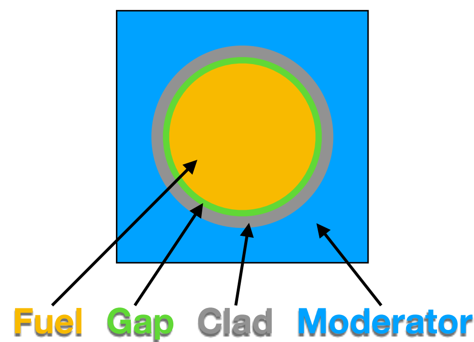


Figure 5.1. Example LWR pin-cell.

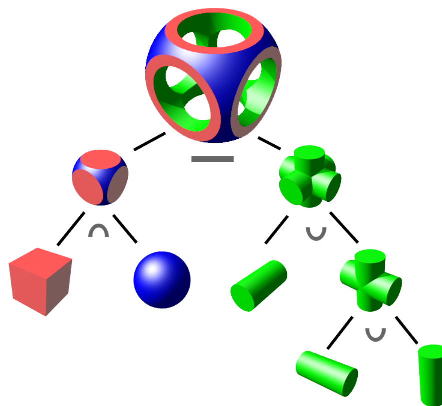
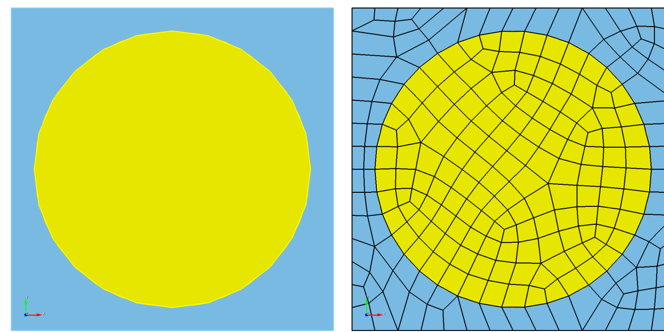


Figure 5.2. The constructive solid geometry process. [83]

5.1 CUBIT PIN-CELLS

The pin-cell is the traditional starting point of any LWR mesh. As shown in Figure 5.1, it is a two-dimensional construct containing rings for fuel, gap, cladding, burnable poisons, and moderator surrounding those. Generating high-quality pin-cell meshes is critical to both accuracy and efficiency.

Volume preservation is desirable for an accurate solution. If the volume of fuel is incorrect, then obtaining the correct eigenvalue is difficult. LWR nuclear fuel is cylindrical: presenting a unique challenge to mesh with straight-sided finite-elements such as quadrilaterals, triangles, hexahedrals, tetrahedrals. Meshing tools, such as `Cubit`, generally rely on constructive solid geometry (CSG) to describe the geometry. As shown in Figure 5.2, CSG geometries are built through union, intersection, and subtraction operations on primitives. For simplified LWR geometries this often means the geometry can be perfectly represented by CSG. However, when the geometry is meshed, volume can be "lost"/"gained" in the process. As mentioned in Chapter 4.1, some unstructured mesh MOC projects have attempted to ameliorate this issue in various ways, including corrections based on the true volume, [84] and using curved



(a) CSG for a C5G7 pin-cell Fuel area: 0.91609 cm^2 **(b)** Simplest mesh, Fuel area: 0.91192 cm^2

Figure 5.3. Constructive solid geometry and simple meshing of a C5G7 pin-cell.

edges on elements [74], while others didn't have a solution [17]. Within this section, a scheme is developed to directly create volume-preserving, for a broad class of, pin-cell meshes.

Reactor geometries typically have a high degree of symmetry. While real-world reactors may have non-symmetric features, such as non-symmetric coolant flow and imperfections in assembly position, reactor models are often perfectly symmetric. This symmetry should be reflected in the mesh. Without mesh symmetry, the solution obtained cannot be perfectly symmetric. In addition, symmetry can be capitalized on to reduce the computational domain to just a symmetric portion, greatly reducing the computational effort needed to obtain a solution.

Figure 5.3a shows a 2D, fuel/moderator pin-cell from the C5G7 [85] benchmark as developed within `Cubit`. The circle, square and their intersection are perfectly represented in CSG with the fuel having an area of 0.916088 cm^2. Utilizing the simplest meshing technique in `Cubit` generates the quadrilateral mesh in Figure 5.3b. While this mesh might look reasonable, it has several undesirable qualities:

1. Volume (area) is not preserved. The fuel area is 0.91192 cm^2representing a loss of 0.45%.

2. Little to no control over the amount of mesh in the moderator.

3. The mesh is not symmetric, ruling out symmetric solutions.

4. Boundary nodes are not at regular or symmetric intervals, making it impossible to create an assembly mesh by tiling these pin-cells.

5. The mesh is not easily split along any axis, making it difficult to construct half/quarter pin-cells.

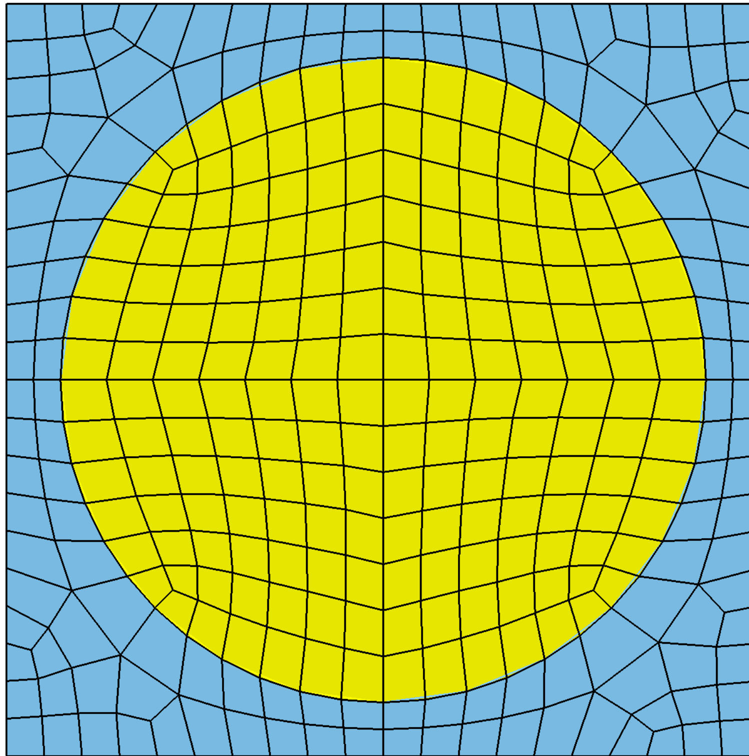


Figure 5.4. Meshing by quarters gives much better control and symmetry. Generated by Listing 5.1.

Creating a better mesh involves taking more control over the `Cubit` meshing process. In particular, solving 5 will allow for simpler fixes to 3. Figure 5.4 shows a large improvement utilizing quarter pin-cells. This guarantees symmetry but also provides many surfaces for controlling the mesh generation. In `Cubit`, any surface allows for setting the number of "intervals" along that surface. Intervals are the number of equally spaced mesh segments which are along that surface. Interval control is critical to controlling mesh density both within the fuel and in the moderator. Also, controlling intervals on the exterior of the pin-cell, together with symmetry, allows pin-cells to be "tiled" thus facilitating the creation of assembly meshes.

Even though using quarter pin-cells solves several of the issues with the previous mesh, it still fails to address 1. The "volume" (surface area) of the fuel in this mesh is .913475 $\mathtt{cm}^2$, which represents an error of 0.285%.

One idea, similar to the arbitrary polynomial fuel discretization scheme in NEWT [86], is to mesh the fuel pin as a regular, n-sided polygon. In this manner, the node positions can be corrected using the exact formula for the area of the cell, which is,

$$A = \frac{1}{2} n R^2 \mathtt{sin} \frac{2\pi}{n}, \tag{5.1}$$

Listing 5.1. Cubit script which generates the mesh in Figure 5.4.

```
reset

create surface rectangle width 1.26 height 1.26 zplane
create surface circle radius .54 zplane
subtract volume 2  from volume 1   keep
delete Body 1

merge all

webcut body  all plane xplane
webcut body all yplane

delete surface 8 10 14 12 13 15

compress all

color surface 1 yellow
color Surface 2 lightskyblue
graphics linewidth 3

mesh surface all

Volume all copy rotate 90 about z
Volume all copy rotate 180 about z

merge all
compress all
```

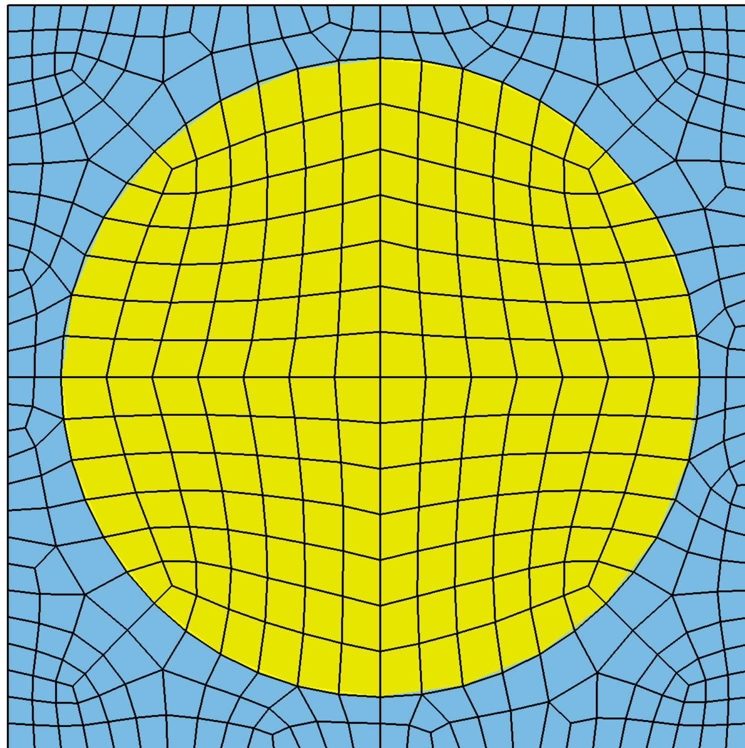


Figure 5.5. By utilizing (5.2) the exact fuel area can be meshed. Generated by Listing 5.2.

where n is the number of sides, and R is the distance from the center of the regular polygon to one of its vertices. For the purposes of meshing the fuel pin while preserving area, this formula can be reorganized to solve for the radial distance of the vertices:

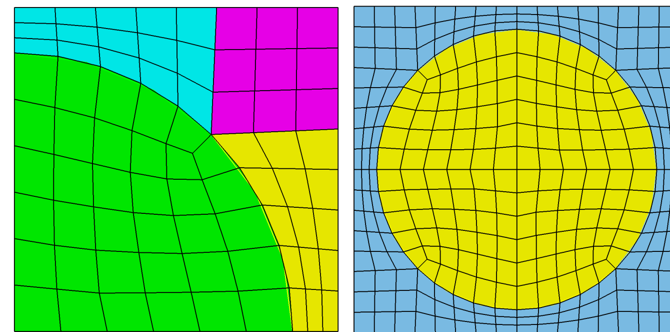
$$R = \sqrt{\frac{2A}{n\texttt{sin}\frac{2\pi}{n}}}. \tag{5.2}$$

Knowing the actual area (A) and the number of sides of the mesh surface (settable within `Cubit` as an interval), R is obtained.

Figure 5.5 shows a mesh with the correct area for the fuel pin: 0.916088$\texttt{cm}^2$. The Python script found in Listing 5.2 was used to generate this mesh. It is similar to Listing 5.1 with a few added pieces (besides the `cubit.cmd()` wrapping everything). The most notable addition is the `discretizedRadius()` function at the top. It takes in the real radius of the fuel pin and the number of regular sides of the polygon that swill represent the fuel. It returns the radial distance at which all of the nodes should be placed so that when the circle is meshed, the meshed volume is exactly the correct volume of the fuel. Further, the number of regular mesh intervals around the perimeter of the quarter circle is set to ensure that a regular polygon is produced with the correct number of sides.

Listing 5.2. Cubit script which generates the mesh in Figure 5.5.

```
#!python
import math

def discretizedRadius(real_radius, num_sides):
  area = math.pi * real_radius * real_radius
  return math.sqrt( (2.0*area) / ( float(num_sides) * math.sin( (2.0*math.pi) /
      float(num_sides) ) ) )

total_num_sides = 48

cubit.cmd('reset')

cubit.cmd('create surface rectangle width 1.26 height 1.26 zplane ')
cubit.cmd('create surface circle radius ' + str(discretizedRadius(0.54,
    total_num_sides)) + ' zplane ')
cubit.cmd('subtract volume 2  from volume 1   keep')
cubit.cmd('delete Body 1')

cubit.cmd('merge all')

cubit.cmd('webcut body  all plane xplane')
cubit.cmd('webcut body all yplane')

cubit.cmd('delete surface 8 10 14 12 13 15')

cubit.cmd('compress all')
cubit.cmd('merge all')

cubit.cmd('# Radial "rings" in fuel')
cubit.cmd('curve 3 5  interval 7')

cubit.cmd('# Number of intervals along pin-cell boundary')
cubit.cmd('curve 4  interval ' + str(total_num_sides/4))

cubit.cmd('mesh surface all')

cubit.cmd('Volume all copy rotate 90 about z')
cubit.cmd('Volume all copy rotate 180 about z')

cubit.cmd('merge all')
cubit.cmd('compress all')

cubit.cmd('color surface 7 1 3 5 yellow')
cubit.cmd('color Surface 8 2 4 6 lightskyblue')
cubit.cmd('graphics linewidth 3')
```



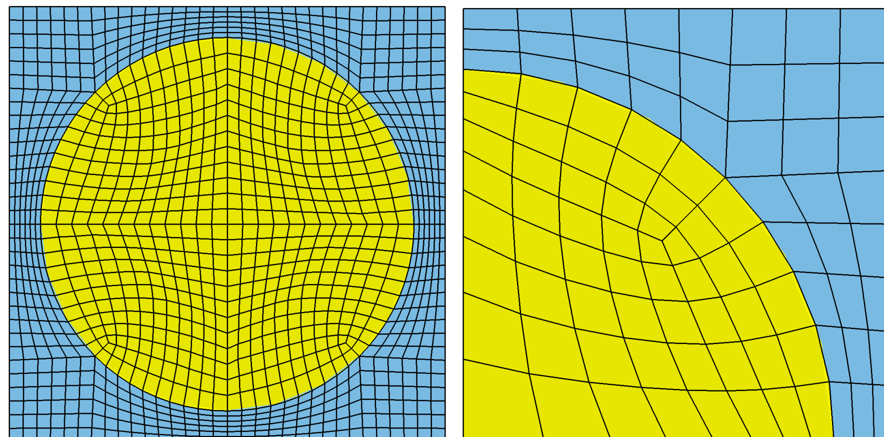
(a) Quarter pin-cell with moderator zones identified.
(b) Final mesh using moderator zoning.

Figure 5.6. Meshing using moderator zoning.

Another improvement that can be made is to clean up the moderator mesh. In 5.5, the moderator mesh is simply using a "pave" meshing scheme within `Cubit`. Pave leads to a loss of symmetry, and worse, a loss in control of the element size. The mesh density in the moderator impacts solution fidelity through better or worse representation of the scattering source with flat-source regions. To control the moderator mesh, three zones are identified within the moderator quarter geometry, each having four bounding curves. These three zones can be seen in Figure 5.6a. It should be noted that in Figure 5.6a, there are a total of eight "azimuthal" intervals around the fuel. These eight intervals are then mirrored on the outside (right and top) of the quarter pin-cell. By distinguishing these three zones with four sides each and matching intervals on each side, the meshing can be applied through a straightforward "submap" scheme within `Cubit`. This provides a better mechanism to control mesh density in this region.

Of particular importance in Figure 5.6a is the "pink" zone which looks like a "box" or "square" in the upper right corner. This zone must connect from the outer part of the cladding to the outer surface of the pin-cell. As previously noted, the number of intervals on the outer surface of the pin-cell is set by the number of azimuthal sectors chosen. In Figure 5.6, the number of sectors is chosen to be 32 (so 8 in the quarter pin-cell). Therefore there are 8 equally spaced mesh intervals on the outside of the quarter pin-cell. Creating the pink "box" is accomplished by drawing lines from the middle of the outer surface of the cladding to the nearest nodes on the outer surface of the pin-cell. In Listing 5.3, the nearest nodes on the outside of the pin-cell have been found manually. However, this could be accomplished with a simple loop over the intervals on the outside of the pin-cell. This is the process used the `Cubit` scripts created for the BEAVRS mesh discussed in §9.4.

The corner zone is as square as possible. The number of outer pin-cell intervals contained within the pink box then sets the number of radial intervals in the modera-



(a) Utilizing 64 azimuthal intervals for a finer mesh.

(b) Biasing the mesh toward the outside of the pin-cell.

Figure 5.7. Mesh modifications are straightforward using the zoned moderator algorithm.

tor. This scheme allows for adjusting to fuel/burnable poisons/guide tubes which are closer or further away from the edge of the pin-cell.

The completed pin-cell can be seen in Figure 5.6b. The mesh is symmetric in each cardinal direction and contains uniform mesh sizing/spacing. The `Cubit` script which generates the mesh in Figure 5.6b can be found in Listing 5.3.

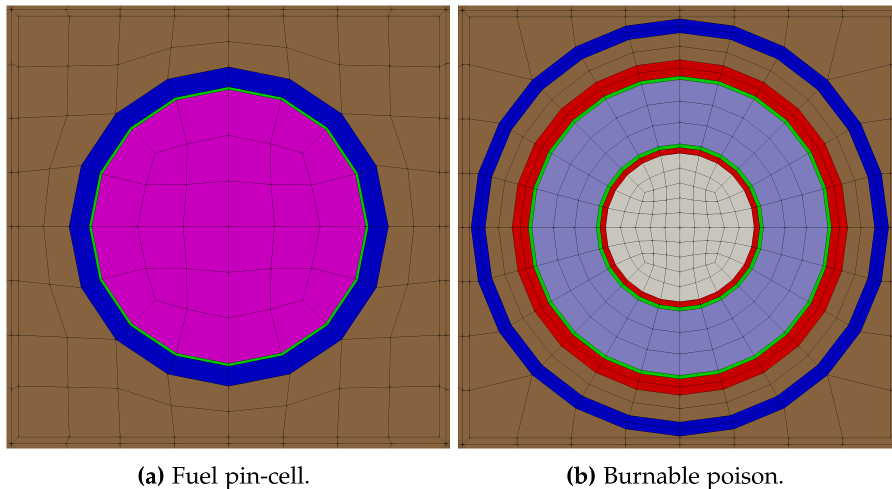
Also, using this scheme, it is straightforward to vary the mesh density. As seen in Figure 5.7a, by changing the number of azimuthal intervals, everything else changes smoothly. Also, as shown in Figure 5.7b, interval biasing, and extra intervals in the fuel can provide rings toward the outside of the fuel, which would be useful for capturing burnup/depletion effects.

Listing 5.3. Cubit script which generates the mesh in Figure 5.5.

```
#!python

cubit.cmd('reset')

import math

def discretizedRadius(real_radius, num_sides):
  area = math.pi * real_radius * real_radius
  return math.sqrt( ( (2.0*area) / float(num_sides) ) * (1.0 / math.sin( (2.0*math.pi) / float(num_sides) ) ) )

total_number_of_azimuthal_sectors = 32

quarter_sectors = total_number_of_azimuthal_sectors / 4
quarter_sectors_in_box = int(quarter_sectors / 2.45)
quarter_sectors_outside_box = quarter_sectors - quarter_sectors_in_box

outer_clad_radius = discretizedRadius(0.54, total_number_of_azimuthal_sectors)

cubit.cmd('create surface circle radius ' + str(outer_clad_radius) + ' zplane ')

# Compute the outermost node on the cladding location
outermost_xy = outer_clad_radius / math.sqrt(2.)

# The limits of the pin_cell
```

```
half_pitch = 1.26 / 2.

cubit.cmd('webcut body all xplane')
cubit.cmd('webcut body all yplane')

cubit.cmd('delete surface 4 6 7')

cubit.cmd('merge all')
cubit.cmd('compress')

# Create vertices on outside of cladding for moderator
cubit.cmd('create vertex ' + str(outermost_xy) + ' ' + str(outermost_xy) + ' 0 on curve 2')
cubit.cmd('imprint body 1 with vertex 4')

cubit.cmd('merge all')
cubit.cmd('compress')

one_interval = half_pitch / quarter_sectors

# Put down vertices for outer part of pin-cell
cubit.cmd('create vertex ' + str(half_pitch) + ' ' + str(0.) + ' 0')
cubit.cmd('create vertex ' + str(half_pitch) + ' ' + str(quarter_sectors_outside_box*one_interval) + ' 0')
cubit.cmd('create vertex ' + str(half_pitch) + ' ' + str(half_pitch) + ' 0')
cubit.cmd('create vertex ' + str(quarter_sectors_outside_box*one_interval) + ' ' + str(half_pitch) + ' 0')
cubit.cmd('create vertex ' + str(0.) + ' ' + str(half_pitch) + ' 0')

cubit.cmd('merge all')
cubit.cmd('compress')

# Create all of the bounding curves
cubit.cmd('create curve vertex 2 5')
cubit.cmd('create curve vertex 5 6')
cubit.cmd('create curve vertex 6 4')

cubit.cmd('merge all')
cubit.cmd('compress')

cubit.cmd('create curve vertex 6 7')
cubit.cmd('create curve vertex 7 8')
cubit.cmd('create curve vertex 8 4')

cubit.cmd('merge all')
cubit.cmd('compress')

cubit.cmd('create curve vertex 8 9')
cubit.cmd('create curve vertex 9 1')

cubit.cmd('merge all')
cubit.cmd('compress')

# Create the moderator surfaces
cubit.cmd('create surface curve 3 5 6 7')
cubit.cmd('create surface curve 7 8 9 10')
cubit.cmd('create surface curve 4 10 11 12')

cubit.cmd('merge all')
cubit.cmd('compress')

# Set Intervals for meshing

cubit.cmd('curve 7 10 8 9 interval ' + str(quarter_sectors_in_box))
cubit.cmd('curve 3 6 4 11 interval ' + str(quarter_sectors_outside_box))

# Pellet interior
cubit.cmd('curve 1 2 interval ' + str((quarter_sectors/2)+2))

# Mesh
cubit.cmd('surface 4 2 3  scheme SubMap')
cubit.cmd('mesh surface 1 2 4 3')

#Create the rest of the pin-cell
#cubit.cmd('Volume all copy rotate 90 about z')
#cubit.cmd('Volume all copy rotate 180 about z')

cubit.cmd('merge all')
cubit.cmd('compress')
```

```
cubit.cmd('color surface 13 1 5 9 yellow')
cubit.cmd('color Surface 15 16 2 3 4 6 7 8 10 11 12 14 lightskyblue')
cubit.cmd('graphics linewidth 3')
```



(a) Fuel pin-cell. **(b)** Burnable poison.

Figure 5.8. Meshed pin-cells from the BEAVRS benchmark [87].

This scheme is extendable to pin-cells containing multiple concentric rings such as the fuel, gap, and clad represented in Figure 5.8a. The annular sections surrounding the interior circle are given the same number of azimuthal intervals. Even complicated structures, such as the burnable poison in Figure 5.8b, can be meshed. In addition, the pin-cells in Figure 5.8 display how simplified spacer grids can be added to the outside of the pin-cell.

While all of the meshing shown here is accomplished using `Cubit`, the algorithm is straightforward enough to code up analytically. That is, the nodes and elements can be laid down in a deterministic way that does not need the unstructured mesh capability within `Cubit` and instead can be directly programmed.

5.2 Graph-based mesh generation

The preceding section developed a strategy for generating a single volume-preserving fuel pin-cell. However, reactors are built of combinations of more complicated structures including guide tubes, instrument tubes, burnable poisons, baffles, spacer grids, and other structural pieces. For light water reactors (LWRs), there are many repeating pieces of geometry. Using an unstructured meshing tool such as `Cubit` for these repetitive meshing tasks is both cumbersome and slow [88]. Instead, what is needed is an efficient way to take the individual repeating pieces (such as pin-cells) and place them together to form 2D/3D assemblies or cores. This section describes a new capability

which was added to `MOOSE` to do graph-based mesh generation for generating reactor meshes: the `MeshGenerator` system.

It's important to recognize that the greater reactor simulation community has long recognized and capitalized on this repetition of geometry. Many CSG-based reactor simulation tools [2, 11, 89] utilize nested geometry where pin-cells, assemblies, etc. are uniquely described and then placed into arrangements to complete the core. However, the unstructured mesh reactor simulation community has not commonly made this same connection. This is due to the added constraints of conforming unstructured mesh: element sides must meet perfectly with their neighbors and nodes are shared between neighbors. Therefore, it is not as simple to refer to a geometric pattern within a nested hierarchy. Instead, the elements and nodes must actually be created and linked (stitched) to create the complete mesh.

The `MeshKit` [90] project was able to make this connection. `MeshKit` allows for a reactor mesh generation process to be described as a directed acyclic graph (DAG) of steps to be taken to generate a mesh. Repeated cells of the reactor can be read in, repeated, and stitched to create reactor cores. However, this process happened as a pre-processing step to running the calculation, ruling out the ability to generate a mesh in memory and then utilize it in a calculation. By developing a similar capability within `MOOSE`, the mesh generation and modification capabilities can be run, in parallel, during the execution of `MOCkingbird`.

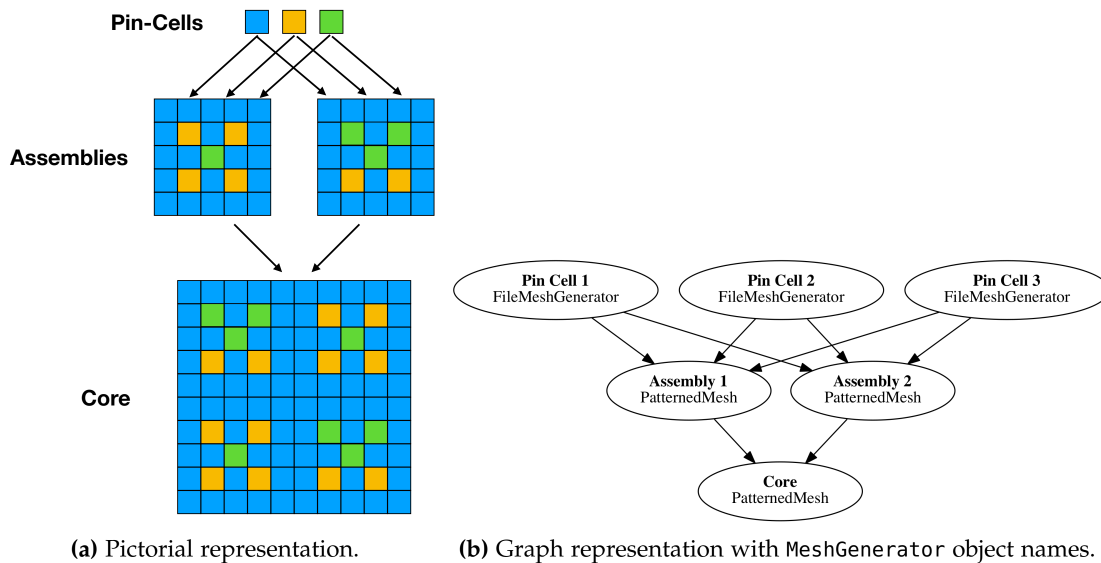
Meshing a three-dimensional LWR core typically follows these steps:

1. Generate 2D pin-cells for fuel, burnable poisons, guide tubes, etc.
2. Combine pin-cells into 2D assemblies
3. Combine the assemblies into a 2D core
4. Add baffle and water mesh
5. Extrude the two-dimensional reactor mesh capturing material changes for each unique elevation

A new capability within `MOOSE` called the `MeshGenerator` system is a way to specify this "flow" of mesh from 2D pin-cell → 2D assembly → 2D core → 3D core in a natural way that is also memory and time-efficient. This flow can be visualized as a directed-acyclic-graph (DAG), as shown in Figure 5.9. The `MeshGenerator` system is composed of objects (in the object-oriented programming sense), where each object represents a discrete action to be taken for building a mesh. These objects populate the mesh generation graph.

As examples, `MeshGenerator` objects include (but are not limited to):

- `FileMeshGenerator`: Read a mesh from a file



(a) Pictorial representation.

(b) Graph representation with `MeshGenerator` object names.

Figure 5.9. Representations of the mesh generation process for LWR reactors.

- `TransformGenerator`: Rotate and translate meshes
- `StitchedMeshGenerator`: Take two meshes as input and make them one by stitching together their common nodes
- `PatternedMeshGenerator`: Take as input multiple meshes and create a tiling of them from a specified pattern
- `SubdomainIDGenerator`: Set the "Block ID" of the elements in the mesh (used to assign material properties)
- `StackedMeshGenerator`: Take 3D meshes as input and stack them and stitch them together.

In Figure 5.9b, the pictorial representation of building a core from pin-cells and assemblies has been translated into a DAG. Nodes of the DAG are labeled with a unique name and the name of the `MeshGenerator`. Some of the `MeshGenerator` objects such as `FileMeshGenerator` don't require any input, but produce a mesh as output while others both take input and produce output meshes.

It is important to distinguish this graph of processes and the hierarchy of geometry within a typical, CSG-based reactor physics code. Here, the relationship is not necessarily geometric. Each of the nodes of the DAG represents a discrete step in the mesh generation process. Those steps might modify the geometry (stretch, skew, extrude, change properties, delete elements, and add boundary IDs) or combine and repeat it. The input and output of each node of the graph is an unstructured mesh with all elements and nodes represented. In contrast, a CSG-based geometric hierarchy describes

the relationship between discrete pieces of geometry. The geometry is not modified; it is combined by nesting the original instances of the individual pieces.

Once the `MeshGenerator` objects and DAG are created, the next step is to execute all of the objects in the correct order. To achieve this, the DAG is first topologically sorted. A topological sort [91] creates a linear ordering from a DAG such that visiting the nodes in that ordering always satisfies the dependencies of each node before visiting it. Note that, in general, a topological sorting is not unique: nodes which don't have either a direct or implied dependency can be listed in any order. As an example, a valid topological sorting of the graph represented in Figure 5.9b would be:

1. Pin Cell 1
2. Pin Cell 2
3. Pin Cell 3
4. Assembly 1
5. Assembly 2
6. Core

A sorting that would be equally valid would be to transpose `Pin Cell 1` and `Pin Cell 3` for instance, or to swap the two `Assembly` nodes.

To create the DAG of `MeshGenerator` objects, a new set of custom input file syntax was added to `MOOSE`. The input file syntax allows for the creation of each `MeshGenerator` (a node in the graph) and its dependencies (the inputs it needs - the edges in the graph). An example of this syntax to build a mesh like the one in Figure 5.9 is shown in Listing 5.4. From this syntax, `MOOSE` can build the requisite objects, topologically sort them and execute each one to create the final mesh.

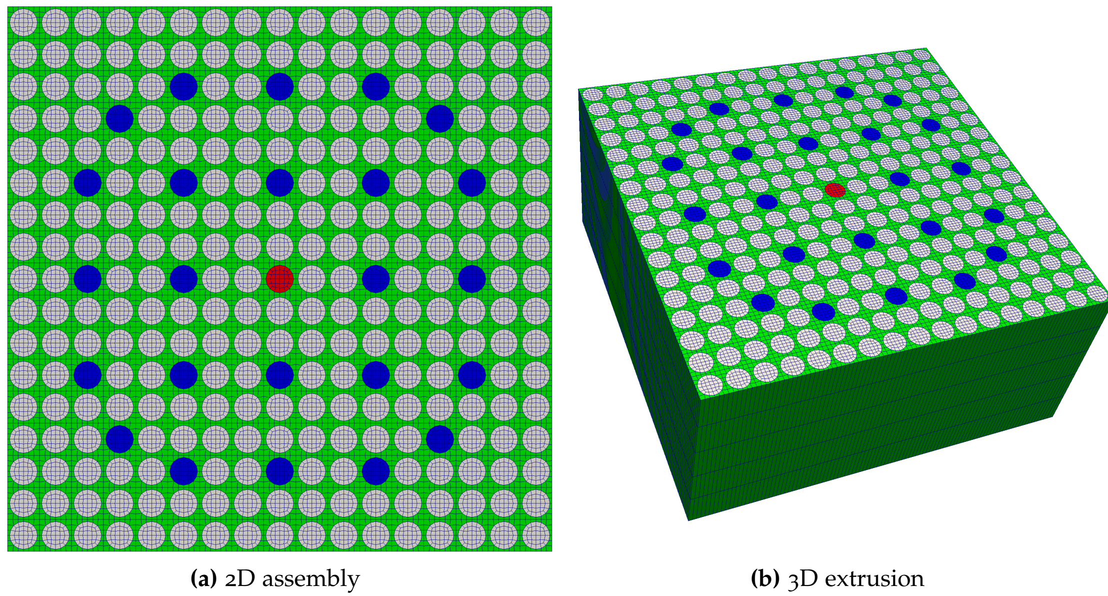
5.2.1 *Three-dimensional Reactor Meshing*

The `MeshGenerator` system can be readily utilized to generate 3D meshes for reactors consisting of extruded geometry. A 2D fuel assembly can be created using `PatternedMeshGenerator` and then extruded to 3D using `MeshExtruderGenerator`. The `MeshExtruderGenerator` takes a 2D mesh as input along with the extrusion directional vector and the number of layers to create. An example of this can be seen in Figure 5.10. In Figure 5.10, a 2D assembly mesh has been extruded into 3D with 5 total layers.

While the `MeshExtruderGenerator` can generate 3D geometries from 2D meshes, real reactor geometries are not so simple. A real reactor assembly, such as the one shown in Figure 5.11, contains many heterogeneities in the axial direction such as

```
...
[MeshGenerators]
  [pin_cell_1]
    type = FileMeshGenerator
    file = moderator.e
  []
  [pin_cell_2]
    type = FileMeshGenerator
    file = fuel.e
  []
  [pin_cell_3]
    type = FileMeshGenerator
    file = burnable_poison.e
  []

  [assembly_1]
    type = PatternedMeshGenerator
    inputs = 'pin_cell_1 pin_cell_2 pin_cell_3'
    pattern = '0 0 0 0 0;
               0 1 0 1 0;
               0 0 2 0 0;
               0 1 0 1 0;
               0 0 0 0 0'
  []
  [assembly_2]
    type = PatternedMeshGenerator
    inputs = 'pin_cell_1 pin_cell_2 pin_cell_3'
    pattern = '0 0 0 0 0;
               0 2 0 2 0;
               0 0 2 0 0;
               0 1 0 1 0;
               0 0 0 0 0'
  []

  [core]
    type = PatternedMeshGenerator
    inputs = 'assembly_1 assembly_2'
    pattern = '1 0;
               0 1'
  []
[]
...
```

Listing 5.4. Example input file syntax for a mesh similar to that found in Figure 5.9



(a) 2D assembly (b) 3D extrusion

Figure 5.10. 3D assemblies can be generated by extruding 2D assemblies.

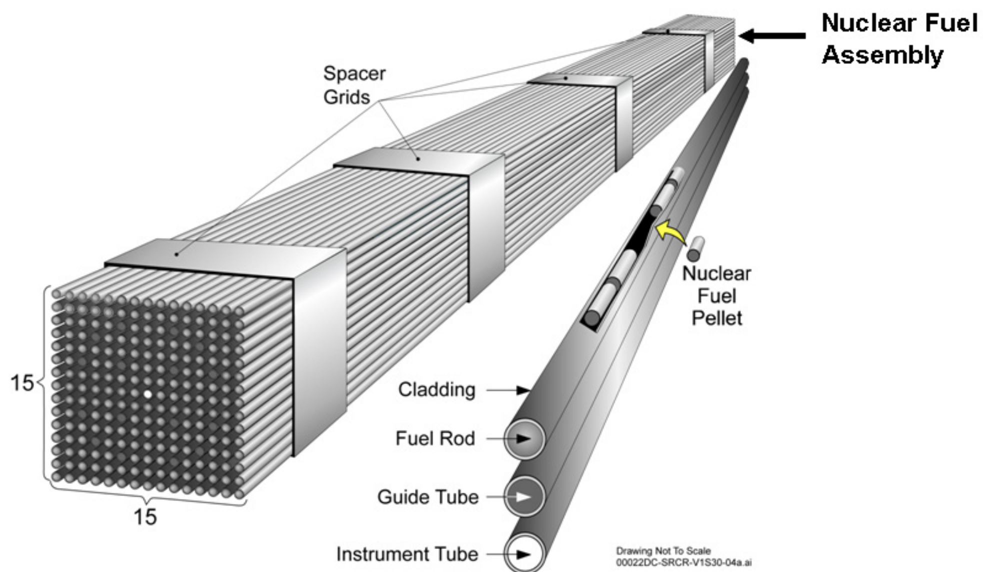
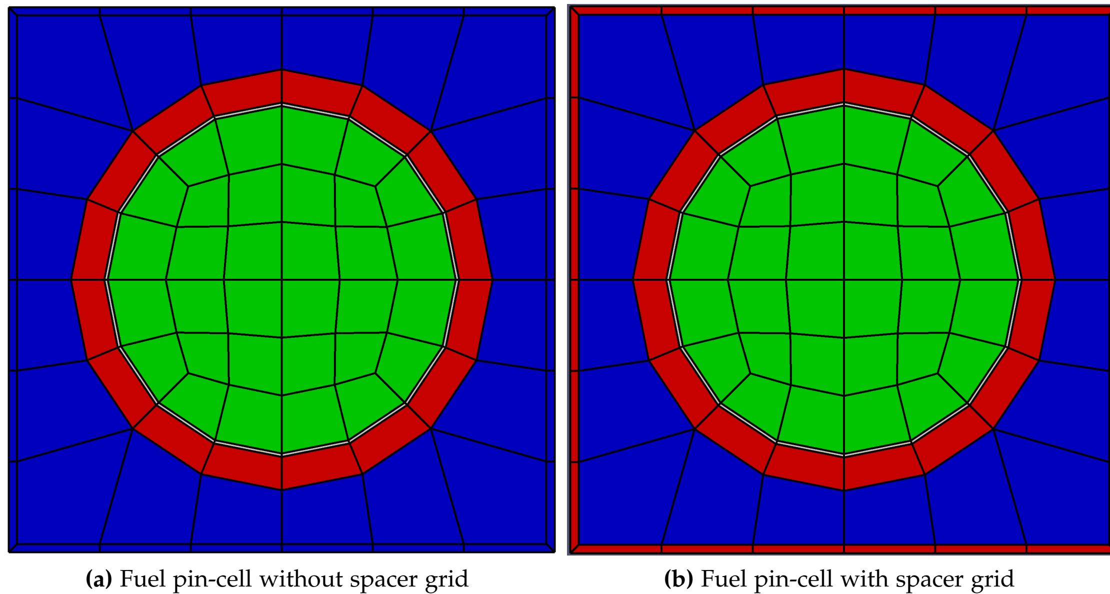


Figure 5.11. Components of a nuclear fuel assembly. From [92]



(a) Fuel pin-cell without spacer grid

(b) Fuel pin-cell with spacer grid

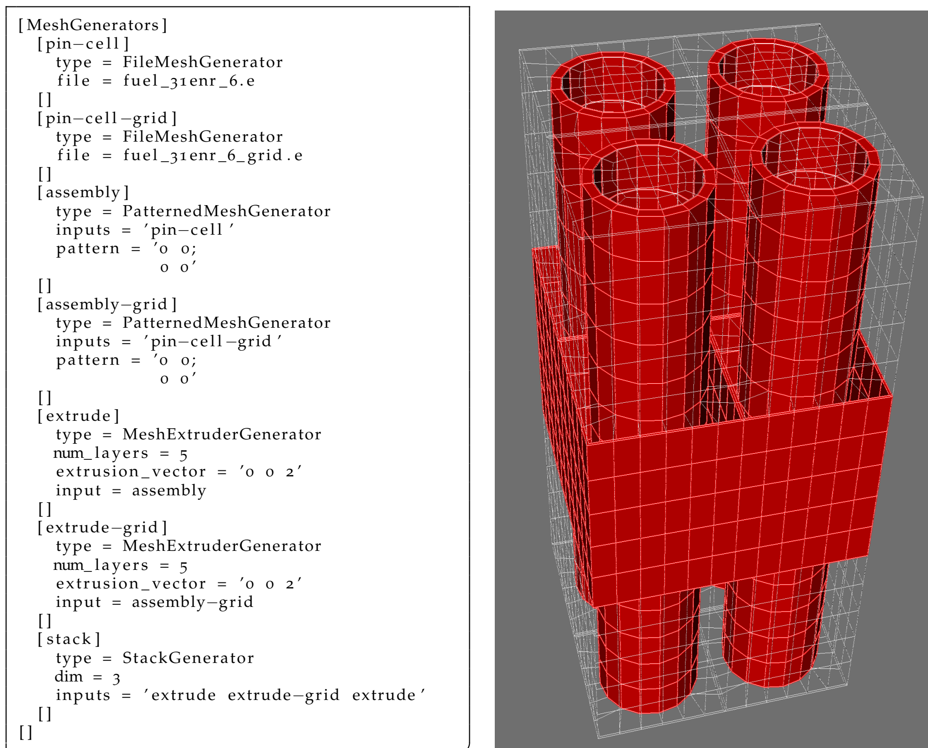
Figure 5.12. The same mesh for a fuel pin-cell with different material assignments on the outer edge to represent either moderator or spacer grid.

spacer grids that separate extruded sections. Each of the homogeneous axial sections is referred to as an "elevation." Additionally, the top and bottom of each assembly differ significantly from the pin structure.

This heterogeneity adds significant complexity. The mesh must be conforming (all neighboring element's nodes must meet each other); therefore, from one elevation to the next, the mesh must match on the interface between the elevations. Within an elevation, it is possible to have fully-unstructured mesh fill the volume, but that presents all the difficulties and drawbacks already mentioned (plus adding significantly to the degree of difficulty in 3D mesh generation). Therefore, keeping the mesh as an extrusion bottom to top is key to efficient meshing. It ensures that every level matches perfectly on the interface.

What is needed is to use the same mesh "template" in the radial direction for every elevation. Each elevation can assign different material properties to portions of that template and then extrude it to create the heterogeneity needed. Each extrusion can then be stacked atop one another to build the full 3D assembly. One added complication is that any radial feature to be tracked must be "in the mesh" from top to bottom.

The simplest example of axial heterogeneity is the addition of spacer grids. As seen in Figure 5.11, spacer grids are metal (typically zircaloy) grids that hold the fuel rods in place. Spacer grids are distributed axially with wide spacing between them. Representing the spacer grids is important for accurate full-core simulation.



```
[MeshGenerators]
  [pin-cell]
    type = FileMeshGenerator
    file = fuel_31enr_6.e
  []
  [pin-cell-grid]
    type = FileMeshGenerator
    file = fuel_31enr_6_grid.e
  []
  [assembly]
    type = PatternedMeshGenerator
    inputs = 'pin-cell'
    pattern = '0 0;
               0 0'
  []
  [assembly-grid]
    type = PatternedMeshGenerator
    inputs = 'pin-cell-grid'
    pattern = '0 0;
               0 0'
  []
  [extrude]
    type = MeshExtruderGenerator
    num_layers = 5
    extrusion_vector = '0 0 2'
    input = assembly
  []
  [extrude-grid]
    type = MeshExtruderGenerator
    num_layers = 5
    extrusion_vector = '0 0 2'
    input = assembly-grid
  []
  [stack]
    type = StackGenerator
    dim = 3
    inputs = 'extrude extrude-grid extrude'
  []
[]
```

Figure 5.13. Input file syntax and resulting 3D assembly with spacer grids using the pin-cells in Figure 5.12. The zircaloy elements have been highlighted.

The zircaloy absorbs some neutrons, but it also reduces the amount of moderator in those sections of the reactor, leading to a dip in local thermal flux.

One possible way of modeling the spacer grids used in the BEAVRS benchmark [87] (discussed in more detail in §9.4) is to add a thin zircaloy layer to the outside of pin-cells that maintains the total mass and volume of the grid within the elevations containing spacer grids. For simplified meshing, this requires that a thin layer of mesh run from the top of the mesh to the bottom that surrounds every pin cell. As shown in Figure 5.12, the very same mesh is used for all axial elevations of the fuel pins. To produce the desired effect of only having the spacer grid during certain elevations, the material in the spacer grid mesh is changed between being moderator or zircaloy depending on the elevation.

An example of using this strategy can be found in Figure 5.13. The input file syntax showing the `MeshGenerator` objects used is on the left with the resulting mesh on the right. Two pin-cells are read from files; they are then be used to build 2x2 2D assemblies, one with and one without zircaloy around the outside of the pin-cells. Next, each assembly is extruded to create the 3D elevations. Finally, the 3D elevations are "stacked" to create the full 3D assembly. Note the flexibility of the `MeshGenerator`

system to mix, match, extrude, and stack meshes. An analogous procedure is utilized to generate the 3D core in §9.5.

5.2.2 *Meshing Conclusion*

Meshing is a critical piece for performing MOC on unstructured mesh. It's needed to define the geometry and impacts solution fidelity. A meshing scheme was developed for creating pin-cells that can be tiled. These pin-cells are made of quadrilateral elements, are symmetric and preserve volume. Also, it is possible to adjust the mesh density within each zone of the pin-cell and even bias the mesh within the fuel. An explanation was given for how the `MeshGenerator` system can be utilized to efficiently generate 3D heterogeneous geometry critical to modeling real-world reactors. A DAG of `MeshGenerator` objects can be used to read in, pattern, extrude, and stack meshes to achieve the desired geometrical and mesh density needs.

6 SPARSE, SCALABLE, ASYNCHRONOUS COMMUNICATION ALGORITHMS FOR PROBLEM SETUP

`MOCkingbird` requires several different methods for communication of data. Most-obvious is the parallel communication of angular flux values during the transport sweep, which is explored in detail within §7. Another sparse parallel algorithm is required within the "setup phase," the period when the solver is started in parallel. "Sparse" here means that each process communicates with only a few other processes, generally just with "neighboring" processes (processes which own adjoining portions of the domain-decomposed geometry). While problem startup time may not seem significant when power iteration time should dwarf it, full 3D reactor simulations can stress the startup algorithms.

In this chapter, several sparse, communication algorithms are tested for viability within `MOCkingbird`. They are tested for scalability and robustness. Ultimately, one is selected, and a novel interface is developed, enabling its use throughout `libMesh`, `MOOSE`, and `MOCkingbird`. An example usage is the "track-claiming" algorithm in §8.1.

6.1 SPARSE DATA EXCHANGE ALGORITHMS

The geometric representation and accuracy needed in reactor physics requires a massive amount of computational capability. Therefore, parallelization is essential for any neutron transport tool. As mentioned in §3.1, modern clusters are typically built utilizing many compute nodes connected using high-speed network infrastructure such as Infiniband [30] or Intel OPA [93]. To make use of such a machine, a program must run simultaneously on the nodes, with each instance of the program communicating with the others across the network using MPI. These MPI processes need to coordinate and collaborate to solve the problem at hand.

MPI contains two main types of communication: "point-to-point" and "collective." Point-to-point operations have single-senders and single-receivers. As an example, MPI rank 5 might send an array of data to MPI rank 8. Collective operations, on the other hand, require all MPI processes to contribute. An example collective operation is global summation (which, in MPI, is a "reduction"): all ranks contribute to part of the final value.

This goal of this section is to achieve scalable point-to-point communication with only a few (thus, "sparse"), neighboring, processes. Traditional algorithms for sparse communication make use of a collective operation for coordination. However, new algorithms have been introduced which remove this step [94]. The following section

Figure 6.1. One-sided, single-hop vs. multi-hop messaging. Single-hop algorithms must globally sync between each communication. Multi-hop allows a message to move through multiple MPI ranks without global synchronization.

tests traditional sparse exchange algorithms against more modern variants and introduces two new sparse exchange algorithms.

6.1.1 *Scalable Sparse Data Exchange*

A key algorithm within `MOCkingbird` is when each MPI process sends data to a few neighboring processes, sometimes with the need for a return message. This type of "sparse" communication can be delineated into a few categories depending on the need for a return message (one-sided vs. two-sided) and whether or not the message travels between two ranks (one-hop) or multiple (multi-hop):

1. One-sided, one-hop
2. Two-sided, one-hop
3. One-sided, multi-hop.

Figure 6.1 shows the differences between a single-hop algorithm and a multi-hop algorithm. Both of the algorithms depicted in Figure 6.1 are one-sided (there is no return message), but in the case of the multi-hop algorithm, the message is passed to multiple other MPI processes. Note: this is not just for "routing" of the message (such as in [95]), a multi-hop algorithm is needed when each of the receivers needs to receive that message, do some work with it, then pass it on. While it is possible to achieve the same effect with multiple rounds of a single-hop algorithm (communicate, stop, communicate again, repeat), a multi-hop algorithm allows messages to move through as many ranks as necessary without stopping for synchronization.

As is discussed in §8.1, `MOCkingbird` employs single-hop algorithms during the setup phase of the calculation to communicate track starting information. However, the core of track integration engine for MOC is entirely multi-hop (3). This section explores options for one- and two-sided single-hop communication; leaving the multi-hop communication algorithm for Chapter 7 to detail domain-decomposed ray-tracing.

Single-hop, sparse data exchange can be implemented in multiple ways using MPI. In [96], Hoefler et al. outlined four methods for sparse data exchange:

1. PEX: Personalized Exchange
2. PCX: Personalized Census
3. RSX: Remote Summation
4. NBX: Nonblocking Consensus

Algorithm 3: One possibility for implementation of the Personalized Exchange (PEX) algorithm from [96].

1 Fill number of MPI process length vector with zeros
2 Insert 1 into each position where this process will send
3 Call `MPI_Alltoall`
4 Inspect resultant vector to see who will send to this process
5 Begin `MPI_Isend`
6 Begin `MPI_Receive`
7 Wait for sends and receives to complete

Of these algorithms, PEX and NBX represent the "classic" and the "new" way, respectively, for single-hop data exchange and are worth benchmarking for use within `MOCkingbird`. As shown in Algorithm 3, PEX is straightforward to implement using an `MPI_Alltoall` to notify all receivers of whom will be sending to them (and optionally how much data or how many messages). First, a vector is created by each MPI process that is the number of MPI processes in length. Next, each sender places a number into the "rank" position of each receiver it will send to. That number can either, simply, be "1" (stating an intention to send to that rank) or can represent the amount of data or number of messages to be sent. That vector is then "transposed" across the MPI processes using `MPI_Alltoall`, as shown in Figure 6.2. In this way, each MPI process receives a vector that is the length of the total number of MPI processes long, with each spot in the vector containing the intention of that rank to send (or not send) to this MPI process. That is, each process receives a "personalized" vector containing who will send to them (and possibly how much data).

However, although the PEX algorithm always begins with an `MPI_Alltoall`, to notify receivers of the intention of the senders, there are several options for implementing the actual communication. While [96] only specifies one option (posting nonblocking sends/receives), several variations are viable and should be explored. These vary from utilizing blocking sends and/or receives to whether or not to specify the rank to `MPI_Receive` from. Here are just some of the options:

Figure 6.2. Pictorial showing the action of the `MPI_Alltoall` within the PEX algorithm to notify receivers of who will be sending to them.

1. Start non-blocking sends; do in-order blocking receives with sender specification (`isend_recv_in_order`)
2. Start non-blocking sends; do any-order blocking receives without sender specification (`isend_irecv_any_order`)
3. Start non-blocking sends; start all non-blocking receives with sender specification (`isend_irecv`)
4. Start all non-blocking receives with sender specification; start all non-blocking sends (`irecv_isend`)

Algorithm 4: Original NBX algorithm published in [96]

```
1  Start all MPI_Issend
2  while true do
3      if MPI_Iprobe then
4          Do MPI_Receive
5          Act on data
6      end
7      if MPI_Issend are complete and not started MPI_Ibarrier then
8          Start MPI_Ibarrier
9      end
10     if Started MPI_Ibarrier and MPI_Ibarrier Complete then
11         break out of loop
12     end
13 end
```

Of these options, 4 contains a theoretical advantage in the case of large messages due to the pre-allocation of receive buffers. When the message from the sender arrives

the memory has already been set aside for the incoming data, and MPI can very readily fill it. However, one downside to option 4 is that no sending operations begin for a moment, which could hinder its performance with small messages which MPI may have been able to buffer internally. This could theoretically lead option 3 to have an advantage with smaller messages. The two options that rely on blocking receives should always be at a disadvantage due to lag in setting up the communication and possibly waiting for specific senders.

Algorithm 5: Modified Nonblocking Exchange (NBX) algorithm utilizing non-blocking receives.

```
1  Start all MPI_Issend
2  while true do
3  |  if MPI_Iprobe then
4  |  |  Start MPI_Irecv
5  |  |  Add to receive list
6  |  end
7  |  Clean up receive list using MPI_Test
8  |  Possibly act on data that has been completely received
9  |  if MPI_Issend are complete and not started MPI_Ibarrier then
10 |  |  Start MPI_Ibarrier
11 |  end
12 |  if Started MPI_Ibarrier and MPI_Ibarrier Complete and receive list empty then
13 |  |  break out of loop
14 |  end
15 end
```

The Nonblocking Consensus (NBX) algorithm shown in Algorithm 4 represents a modern MPI-3 approach to sparse data exchange. Nonblocking, point-to-point MPI_-Issend operations are immediately started. MPI_Issend only shows as complete using MPI_Test if the message is acknowledged as starting to be received (it may be completely received, but possibly not - a positive return from MPI_Test implies that the buffer being sent using the MPI_Issend can be reused). Once all of the messages from a sender are beginning to be received, the sender can then start the MPI_Ibarrier. Once all senders begin the MPI_Ibarrier, the algorithm is complete.

The Modified-Nonblocking-Exchange algorithm (MNBX) shown in Algorithm 5 was developed for this work to overcome shortcomings in NBX. The improvement over the NBX algorithm found in [96] is in the way receives are handled. In [96], receives are handled using MPI_Receive while in Algorithm 5, nonblocking MPI_Irecv operations are used for incoming messages. This change might appear to be minor, but it has large implications. First, using MPI_Irecv can speed up receiving when useful work can be done while messages are still being received (overlapping communication and computation). It also could allow for overlapping larger communications coming from

multiple sources. However, by utilizing MPI_Irecv, it is not possible to know when every MPI process has completed the routine and exited it.

The asynchronous nature of Algorithm 5 allows some MPI processes to move through to the next phase of the computation before others. If processes, which are finished, begin sending other communications, the processes that have not yet finished the algorithm may interpret those incoming messages as more messages that should be received and begin receiving them into the wrong part of the calculation.

However, MPI contains a method for keeping this from happening: tags. MPI tags are integers which are attached to each message that a developer can use to help discern which message is meant for which part of the code. In this case, all that is needed is every time the MNBX Algorithm 5 is invoked a new MPI tag is used and no other portion of the code reuse that tag. In this way, every process working on Algorithm 5 can know exactly which messages it needs to process for this particular invocation of the algorithm. To ensure this, a unique, rolling MPI tag system was created within libMesh. Each time Algorithm 5 is invoked, a new unique tag is given to it. To avoid running out of tags, the algorithm wraps around to unused tags. The PETSc library [50] uses a similar rolling MPI tag technique to avoid these issues.

In addition to the four sparse data options outlined in [96], another option exists known as "crystal router." Originally developed in the 1980s and published in [95], this method utilizes a hypercube (or virtual hypercube) topology to deliver point-to-point messages efficiently. A message is routed through multiple compute nodes along paths through a hypercube to end up at their final destination. This is an old technique that is still in use today by some extremely high-performance codes such as the Nek5000 CFD application [97]. The implementation from Nek5000 has been open-sourced within the GSLIB library [98] making it possible to test a high-performance implementation of this algorithm. It should be noted that the crystal router algorithm here is not used to perform the data transmission; instead, it is used in place of the MPI_Alltoall within a PEX-like algorithm to notify the receivers of the senders.

Finally, an experimental algorithm has also been developed that is the first step towards the scalable multi-hop algorithm needed for ray-tracing. This algorithm is referred to as "Fully Asynchronous" due to it being valid for anything to happen in any order. One possibility can be found in Algorithm 6. This algorithm is similar to the MNBX algorithm 5 in that nonblocking sends and receives are utilized, and a nonblocking collective (MPI_Iallreduce vs. MPI_Ibarrier) is used for global consensus. However, this algorithm is more flexible. It allows for an unknown number of senders to send an unknown number of messages of unknown size to an unknown number of receivers in an unknown order. Further, the sending can be mixed with the receiving (although that's not how it's stated in Algorithm 6, for simplicity). As can be seen later, it allows messages to start on one MPI process and navigate through

Algorithm 6: Fully asynchronous one-sided, single-hop communication scheme

```
1  Start MPI_Isend operations
2  Count number of messages sent
3  Start MPI_Iallreduce to find the global number of messages sent
4  while true do
5      Check MPI_Iallreduce to find the global number of messages sent
6      while MPI_Test incoming messages do
7          Start MPI_Irecv for incoming message
8          Add it to a list of current receive operations
9      end
10     Clean up MPI_Irecv operations and act on data
11     Count the number of messages received
12     if Global number of messages is known then
13         if Not yet communicating the number of messages received then
14             Start MPI_Iallreduce for the global number of messages received
15         end
16         else
17             if MPI_Iallreduce of number of messages received is finished then
18                 if global number of messages received == global number sent then
19                     break loop
20                 end
21             end
22         end
23     end
24 end
```

Msg. Size	in_order	any_order	irecv	irecv_isend	FA	NBX	MNBX	CR
1	9.28	9.24	9.27	9.71	1.56	1.50	1.60	3.89
10	9.33	9.29	9.38	9.71	1.62	1.55	1.63	4.25
100	9.68	9.72	9.73	9.91	2.30	2.48	2.31	5.50
1000	23.91	23.85	23.66	25.52	18.84	19.08	18.59	20.67
10000	180.41	166.80	164.59	168.17	153.09	162.43	152.32	165.78

Table 6.1. Data for the message size testing of sparse data exchange algorithms. Each entry represents the time in milliseconds for 10 iterations of the algorithm. This data is also plotted within Figure 6.3.

several before stopping, something that isn't achievable with the `MPI_Issend` required by NBX/MNBX. However, that particular aspect of this algorithm won't be utilized until §8.3.1

6.1.2 *Testing Scalable Sparse Data Exchange*

The previous section outlined five algorithms that should be benchmarked for use within `MOCkingbird`:

1. Personalized Exchange (PEX) (all four variations)
2. Nonbocking Consensus (NBX)
3. Modified Nonblocking Exchange (MNBX)
4. Crystal Router (CR)
5. Fully Asynchronous (FA).

To test these algorithms, a sample implementation of each was developed within a targeted testing code. The test application times how long N communication steps take with each algorithm. With each algorithm, M random receivers are chosen. They are each sent a message of L double-precision floating-point numbers. The tests are carried out on the Lemhi Supercomputer described in §3.1.1. Each test is run 3 times, during each run, the algorithm is executed 10 times. The run with the fastest time for each algorithm is used. The results of this study can be found in Figures 6.3, 6.4, 6.5.

A study was completed looking at how each algorithm behaves as message size is increased. The study utilized 4096 MPI processes spread across 128 nodes. The number of neighbors was chosen to be 26 to represent all of the neighbors surrounding a cube in a perfect lattice. The results of this study can be found in Figure 6.3 and Table 6.1.2. Immediately obvious when looking at 6.3 is that the PEX algorithms that rely on `MPI_Alltoall` are all at least an order of magnitude slower at delivering small messages.

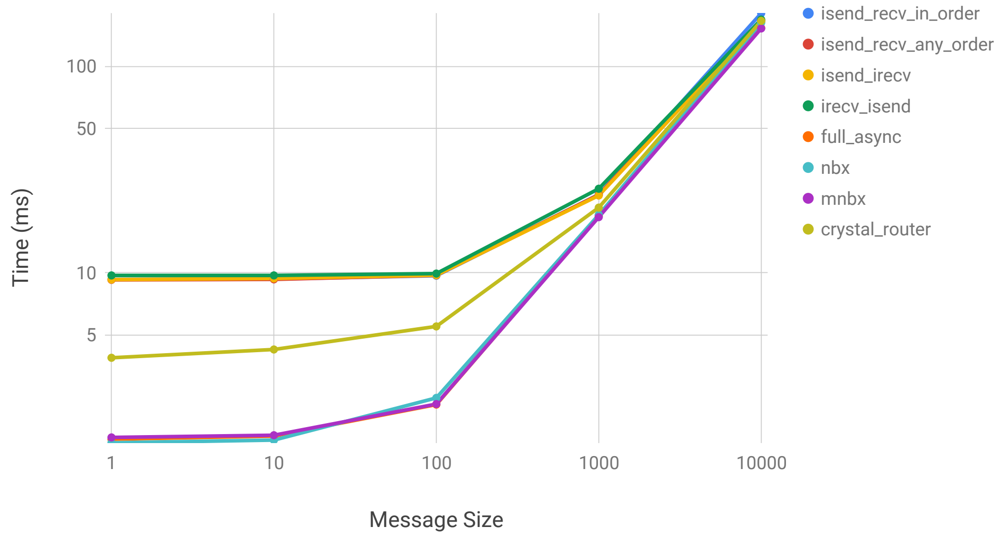


Figure 6.3. Sparse data exchange scalability in message size. The test was run using 4096 MPI processes spread across 128 nodes. Message sizes are in number of double precision floating point numbers (64bit) and Time is based on 10 iterations of the algorithm.

NBX, MNBX, and FA all achieve very similar times. The CR algorithm falls somewhere in-between the other two sets.

As the message size is increased, all of the algorithms achieve similar timings. This is due to the dominant amount of time shifting to the actual transit time of the bytes themselves across the network as message size grows. Even so, it is still interesting to note that for a message size of 10,000, both MNBX (6.6%) and FA (6%) are faster than NBX.

Next, weak scaling of the algorithms is tested. The message size is fixed at 100, and the number of neighbors remained 26 while the number of MPI processes ranged from 32 to 8192. The results of this study can be found in Figure 6.4 and Table 6.2. While all of the algorithms (other than CR) are fairly competitive up to 500 MPI processes, at that point the `MPI_Alltoall` based algorithms start to show their increased algorithmic complexity, in terms of the number of processes: p, of O(p) whereas FA, NBX and MNBX all have a worst-case complexity of O(log(p)). Those three algorithms are giving almost constant time over this large range in number of cores. There is a small growth in those algorithms due to the O(log(p)) complexity of the reductions (`MPI_Ibarrier` and `MPI_Iallreduce`). This provides these algorithms excellent scalability.

The final test performed on these algorithms looks at increasing the number of neighbors. Earlier, the number of neighbors was set to 26 to be representative of the number of neighbors in a 3D simulation, in this case, the number of neighbors is varied

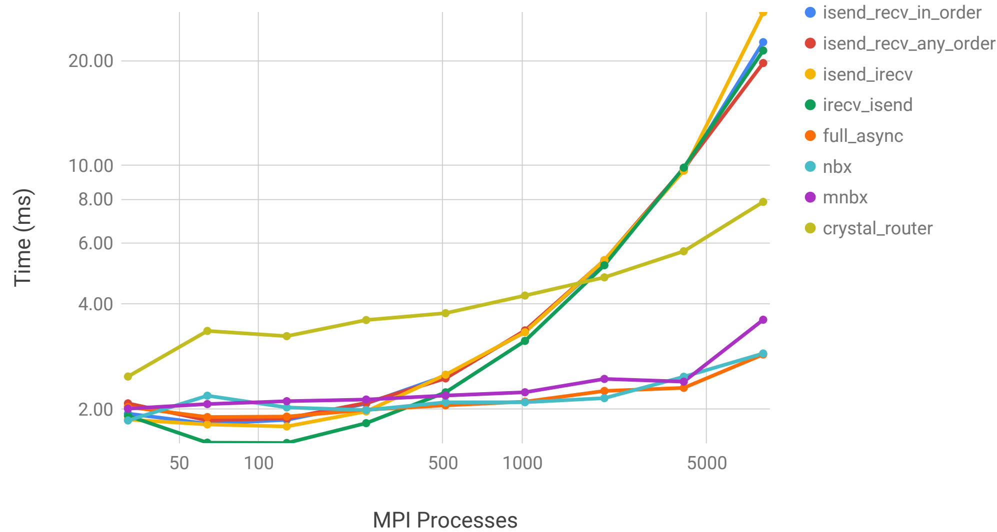


Figure 6.4. Weak scalability of each sparse data exchange algorithm from 32 MPI processes to 8192. This is "weak" scaling: the number of neighbors is held constant at 26 and the message size is held at 100 double precision floating point numbers. The raw data for this plot can be found in Table 6.2

MPI	in_order	any_order	irecv	irecv_isend	FA	NBX	MNBX	CR
32	1.94	2.08	1.87	1.91	2.03	1.85	2.00	2.48
64	1.81	1.86	1.81	1.60	1.90	2.18	2.07	3.35
128	1.86	1.88	1.78	1.60	1.90	2.02	2.11	3.24
256	2.07	2.08	1.97	1.82	1.99	1.99	2.13	3.60
512	2.49	2.45	2.51	2.23	2.05	2.09	2.19	3.77
1024	3.33	3.36	3.32	3.14	2.10	2.09	2.23	4.23
2048	5.31	5.34	5.35	5.17	2.25	2.15	2.44	4.77
4096	9.68	9.80	9.65	9.85	2.30	2.48	2.40	5.67
8192	22.57	19.67	27.50	21.36	2.87	2.89	3.61	7.86

Table 6.2. Weak scaling of each sparse data exchange algorithm. The number of neighbors is held constant at 26 while the message size is 100 double precision numbers. Each entry represents the time in milliseconds for 10 iterations of the algorithm. This data is also plotted within Figure 6.4.

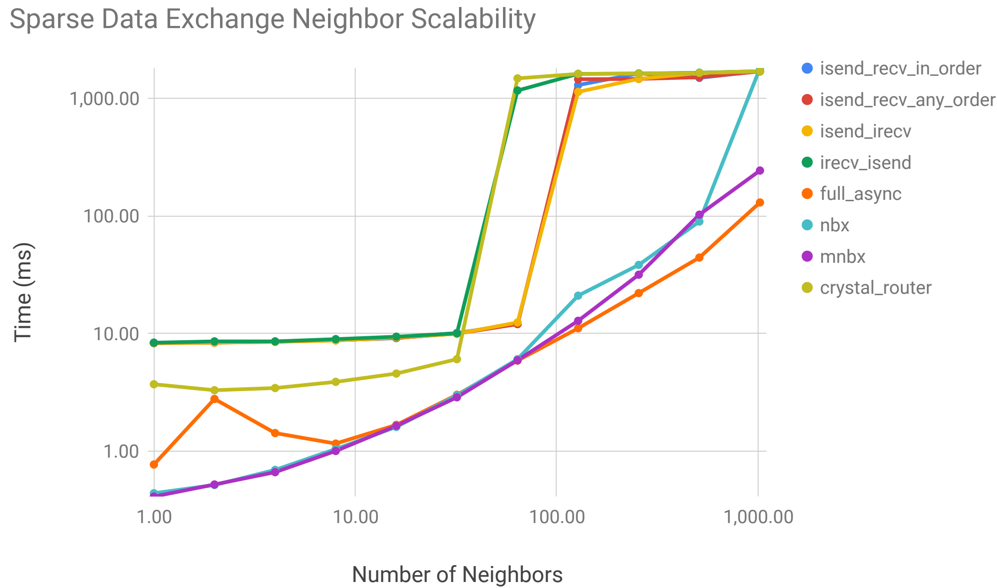


Figure 6.5. Testing each sparse data exchange algorithm's ability to deal with decreasing sparsity. The test was run using 4096 MPI processes spread across 128 nodes. Message size was held constant at 100 double precision floating point numbers. The time represents 10 iterations of the algorithm. The raw data for these plots can be found in 6.3

Neighbors	in_order	any_order	irecv	irecv_isend	FA	NBX	MNBX	CR
1	8.37	8.29	8.32	8.38	0.77	0.44	0.41	3.71
2	8.41	8.37	8.40	8.60	2.79	0.52	0.52	3.30
4	8.57	8.53	8.51	8.59	1.43	0.69	0.66	3.45
8	8.79	8.79	8.75	8.98	1.16	1.04	1.01	3.89
16	9.16	9.19	9.27	9.42	1.68	1.61	1.64	4.58
32	10.13	10.00	9.98	10.08	3.02	2.99	2.88	6.07
64	12.08	12.05	12.46	1,168.21	5.90	6.07	5.94	1,482.00
128	1,299.14	1,457.15	1,138.30	1,618.34	11.11	21.04	12.86	1,617.22
256	1,632.53	1,466.22	1,466.86	1,629.29	22.12	38.43	31.73	1,626.52
512	1,496.52	1,514.70	1,648.91	1,654.49	44.36	89.80	102.72	1,650.31
1024	1,708.59	1,697.63	1,693.57	1,704.62	130.44	1,808.43	243.77	1,706.50

Table 6.3. Scalability in the number of neighbors. Ran using 4096 MPI processes, messages size is held constant at 100 double precision numbers. Each entry represents the time in milliseconds for 10 iterations of the algorithm. This data is also plotted within Figure 6.5.

to see how the algorithms respond. The number of MPI processes was 4096 split over 128 nodes of the Lemhi cluster, and the message size stayed constant at 100 doubles. The results of this study can be found in Figure 6.5 and Table 6.3. The `MPI_Alltoall` based PEX algorithms all have a fixed upfront cost for the `MPI_Alltoall` itself. This leads to those algorithms being 10x slower than the point-to-point based FA, NBX, and MNBX. As the number of neighbors grows, the amount of communication done by all the algorithms increases, and so does time.

Unfortunately, at around 64 neighbors, something breaks down in the nonblocking send/receive routines of the PEX algorithms. It is suspected that some limit is reached within the MVAPICH MPI implementation being used. Several MVAPICH options were tried to attempt to overcome the issue, but nothing worked. OpenMPI was also tried, but it showed a similar trend (although the breakdown in the algorithm didn't occur until 256 neighbors).

The other algorithms (FA, NBX, and MNBX) show excellent resiliency and stable growth out to 500+ MPI processes. FA shows some interesting behavior with just a few neighbors where the time spikes. This is most likely because FA can require multiple `MPI_Iallreduce` operations depending on the order that messages come in. Even so, FA is still faster than any of the PEX based algorithms and quickly goes back to being competitive (and even faster than) NBX, and MNBX as the number of neighbors grows.

To summarize these results, the PEX based algorithms are all encumbered by the O(p) time the `MPI_Alltoall` takes to communicate sender's intentions while FA, NBX, and MNBX all communicate in constant time with a O(log(p)) overhead for sharing completion data. This gives them all excellent scalability. MNBX is a good improvement over NBX that allows for multiple messages to be received asynchronously and those messages to be acted on as they are completely received. It also shows good scalability in all the metrics tested here. Therefore, MNBX algorithm was chosen to be utilized within the setup steps of `MOCkingbird` anytime neighbor data needs to be exchanged (such as during track claiming, which is discussed in §8.1).

However, the FA algorithm is still impressive. It has nearly the same speed and scalability metrics of NBX and MNBX while offering greater flexibility. In particular, it allows for messages to be sent through multiple hosts (multi-hop) - which is not possible with any of the algorithms presented here. The distributed-geometry ray-tracing algorithm at the heart of `MOCkingbird` utilizes a (more complicated) version of the FA algorithm.

```
template <typename MapToVectors,
          typename ActionFunctor>
void push_parallel_vector_data(const Communicator & comm,
                               const MapToVectors & data,
                               const ActionFunctor & act_on_data);
```

Listing 6.1. Interface for a sparse send operation similar in ways to a sparse version of `MPI_Scatterv`.

6.1.3 *An Interface For Sparse Data Exchange*

The preceding section explored several algorithms for viability within `MOCkingbird`. It was found that the Modified Nonblocking Exchange (MNBX) algorithm would be a good fit for the types of neighbor communication needed during the setup phase of `MOCkingbird`. However, the MNBX algorithm, even at its simplest, is about 100 lines of code (not including the actual work to do with each incoming message). Further, that's for one-sided communication, and some of the setup tasks within `MOCkingbird` require two-sided communication which would necessitate careful implementation each time it is needed within the code.

A better idea is to attempt to encapsulate the MNBX algorithm (both one- and two-sided) into functions with a flexible interface that can be reused throughout the code. Working with Dr. Roy Stogner at University of Texas in Austin, a set of flexible routines encapsulating the idea of sparse data exchange were developed and added to the `libMesh` library: `push_parallel_vectors()` and `pull_parallel_vectors()`. These two functions act as proxies for sparse versions of `MPI_Scatterv` and `MPI_Gatherv` and are backed by a sparse data exchange algorithm.

The interface for `push_parallel_vectors()` can is shown in Listing 6.1. `MapToVectors` is any "maplike" object (`std::map`, `std::unordered_map`, etc.) that holds MPI ranks as keys and a "vectorlike" (i.e. `std::vector`) of data to send to that MPI rank. The `ActionFunctor` is a function pointer, function-like object, or lambda function that specifies what to do with each piece of incoming data. As each message is received and unpacked, the `ActionFunctor` is called to operate on the received data. Depending on the algorithm behind `push_parallel_vectors()`, this processing of data may happen asynchronously with the reception of the data.

The `pull_parallel_vectors()` interface can be seen in Listing 6.2. It is similar to `push_parallel_vectors()` except the `MapToVectors` is this time filled with a vector of "queries" for the remote MPI rank. The idea is to communicate a set of "questions" to remote processes and let them respond with "answers." For instance, this can be used to ask neighboring MPI ranks for information about variable values on elements they own which border this MPI rank. In that scenario, the "queries" could contain the

```
template <typename datum,
          typename MapToVectors,
          typename GatherFunctor,
          typename ActionFunctor>
void pull_parallel_vector_data(const Communicator & comm,
                               const MapToVectors & queries,
                               GatherFunctor & gather_data,
                               ActionFunctor & act_on_data,
                               const datum * example);
```

Listing 6.2. Interface for a sparse send and receive operation similar in ways to a sparse versions of `MPI_Scatterv` followed by `MPI_Gatherv`.

element IDs this process is wanting information from. The requests are sent utilizing a sparse data exchange algorithm. When the requests are received, each MPI process calls the `GatherFunctor` and pass each of the received queries. The `GatherFunctor` is responsible for fulfilling the request and returning the data. Another round of sparse data exchange is then used to send the gathered data back to the requestors. Finally, the requestors asynchronously receive the results of their queries, and the `ActionFunctor` is invoked to act on that data.

There are many options for the implementation of `push_parallel_vectors()` and `pull_parallel_vectors()`. After some discussion centering on the patterns of communication, Dr. Stogner originally developed the interfaces and backed them with something similar to a PEX algorithm. However, as shown in the preceding section, it does not perform well with many thousands of MPI ranks. Therefore, for this thesis, a new implementation of `push_parallel_vectors()` was developed, which utilizes a similar algorithm to MNBX. The new implementation of `pull_parallel_vectors()` then became two `push_parallel_vectors()` operations: `push_parallel_vectors()` is called to create the requests then, once all of the requests have been handled, `push_parallel_vectors()` is then used to return the results.

While implementing `pull_parallel_vectors()` with two `push_parallel_vectors()` operations is expedient and makes for maximum code reuse, it doesn't exploit all of the parallelism possible. During a `pull_parallel_vectors()`, many requests are being sent, worked on, coming back, and acted upon. All four phases of that communication pattern could be completed asynchronously. To achieve that, two MNBX algorithms could be blended, which is left to future work.

Within `MOCkingbird`, these `push_parallel_vectors()` and `pull_parallel_vectors()` routines are utilized during the setup phase of the calculation. In particular, they show up during track generation and "claiming" as discussed in §8.1.

7 SCALABLE MASSIVELY ASYNCHRONOUS RAY TRACING (SMART)

The heart of a MOC code is the transport sweep: integration along tracks laid down through the reactor. This can be viewed as a "ray tracing" problem: tracks are akin to "rays" that pass through the reactor interacting with each material encountered. Pixar software utilizes a very similar method for rendering computer-generated graphics [99]. While ray-tracing for MOC has been extensively studied [8, 100] using the more traditional constructive solid geometry, very little work has been done on unstructured mesh [17, 84].

Using unstructured mesh has both pros and cons. As mentioned earlier, working with unstructured mesh allows `MOCkingbird` to directly couple to other physics solved using `MOOSE`, allows for a large amount of geometrical flexibility (anything that can be meshed can be used as the domain) and makes it possible to deform the mesh (such as with thermal expansion). However, scalable distributed memory parallelization is not straightforward with unstructured mesh [84]. To tackle these difficulties, a new parallel algorithm for tracing rays through unstructured mesh has been developed, named Scalable Massively Asynchronous Ray Tracing (`SMART`).

`SMART`, itself, is agnostic of the physics being computed. Because it's agnostic of physics, an implementation of it has been added to `MOOSE` as a "physics module." Physics modules in `MOOSE` are open-source sets of common physics that any `MOOSE`-based application can utilize.

Several aspects of `SMART` are unique:

- Overlapping generation and propagation execution phases
- Completely asynchronous multi-hop messaging
- Asynchronous distributed stopping criteria
- Smart memory pools for rays, data, and messages
- Intelligent buffering
- Efficient execution on unstructured grids

Within this section, each of these capabilities is explored in detail.

This chapter develops an efficient ray-tracing algorithm for use on distributed memory clusters. The chapter proceeds by first exploring algorithms for how a `Ray` can move from element to element through an unstructured mesh. While the algorithms

for the intersection of rays with element-sides appears straightforward, many corner cases must be addressed for robustness.

The second significant development is the Scalable Massively Asynchronous Ray Tracing (SMART) algorithm itself. The utilization of asynchronous (non-blocking) message passing for efficient ray tracing in a distributed memory setting is the primary focus. The key idea being that if communication is overlapped with computation, then the simulation time taken by message passing is minimized.

7.1 TRACKS AND THE ray OBJECT

Before any discussion about tracing a path through distributed, unstructured mesh, the idea of the path, itself, needs to be explored. As explained in Chapter 2, the source iteration scheme used by MOC to solve the Boltzmann transport equation requires the ability to integrate along "lines" through the domain. Those lines are called "tracks" and are typically laid down in regular patterns through the domain. Each track represents a line from one domain boundary to another, as shown in Figure 2.3a.

Traditionally, in MOC codes, the tracks were created and intersected with the geometry to create segments as in 2.3. This was done as a pre-processing step, with the segments stored in memory. For each transport sweep, the MOC solver would then iterate through the segments and compute the outgoing angular flux according to Equation 2.18. This is time-efficient, with the intersection of each track with the geometry only occurring twice. However, this is intractable for detailed, full-core 3D analysis due to the memory requirements necessary to store each segment. This idea is revisited in §10.2.2.

Instead, `MOCkingbird` utilizes "on-the-fly" segmentation: during each transport sweep, each track is traced through the unstructured mesh, with each element crossing forming a segment to be integrated. Other MOC implementations have used the same idea to keep memory use to a minimum [3, 101]. A transport sweep in `MOCkingbird` traces each track completely across the domain: from boundary to boundary. Each time the tracing crosses an element, generating a segment, a routine within `MOCkingbird` is called to integrate that segment.

The job of `SMART` is then to perform the ray-tracing on a parallel, distributed, unstructured mesh. To do this, it "moves" a `Ray` object through the domain. A `Ray` is a C++ object containing all of the data needed to trace the track's path through the domain, but it also contains data arrays capable of carrying other information such as the angular flux and angular quadrature needed by MOC.

The `Ray` data-structure contains the members shown in Listing 7.1. The `data` member is a generic, re-sizeable array that can carry any floating-point data; `MOCkingbird` utilizes it for storing angular flux. The `id` is a unique ID assigned to the `Ray`, for `MOC`-

Listing 7.1. The data members within a Ray object.

```
/// The data that is carried with the Ray
std::vector<Real> data;

/// A unique ID for this Ray.
unsigned long int id;

/// Start of the Ray
Point start;

/// End of the Ray
Point end;

/// The element the Ray begins in
const Elem * starting_elem;

/// The side of the the _starting element the ray is incoming on.
/// -1 if the Ray is starting _in_ the element
unsigned long int incoming_side = -1;

/// The id of the element the Ray ends in (used to optimize when
    _ends_within_mesh)
dof_id_type ending_elem_id = DofObject::invalid_id;

/// Whether or not the Ray ends within the Mesh
bool ends_within_mesh = false;

/// The azimuthal spacing
Real azimuthal_spacing;

/// The azimuthal weight
Real azimuthal_weight;

/// The polar spacing
Real polar_spacing;

/// The sin of the polar angle for this Ray (just 1.0 for 3D)
std::vector<Real> polar_sins;

/// The weight for each polar angle (just 1.0 for 3D)
std::vector<Real> polar_weights;

/// True if this Ray was created as the reverse of another ray
bool is_reverse = false;

/// Wether or not the Ray should continue to be traced
bool should_continue = true;
```

kingbird this is set to the ID of the track coming from the track generator. The start and end points define the beginning and ending of the Ray as a Point object which contains x, y, z floating-point values. For the ray-tracing routines, explained in §7.2, it is critical that the starting location within the unstructured mesh be completely defined. This is handled by the starting_elem and incoming_side data members which will be set during the track claiming phase explained in §8.1.

The rest of the data members are unique to MOC and are set directly from the track data from the track generator discussed in §8.1. In traditional MOC codes, many of these parameters, such as spacing and weights, are not stored for each track. Instead, they are looked up in an array based on an azimuthal index, xy index and, in 3D, polar index and z index. In MOCkingbird there are no loops over track indices (because the tracks are domain decomposed); therefore the Ray would need to store these 4 indices and do a lookup into the OpenMOC track generator to retrieve this information every time the Ray is communicated. Therefore, it is much simpler for MOCkingbird to forego storing these 4 indices and doing the lookup to instead store up to 9 doubles in 2D (using TY polar quadrature) or 5 doubles in 3D. The storage and communication of these values pales in comparison to the rest of the data stored on a Ray, especially the angular flux. The angular flux alone adds 210 doubles to the Ray in 2D with 3 angles of TY polar quadrature and 70 energy groups. Therefore, the convenience and utility of having these MOC parameters instantly available makes up for the tiny increase in memory used and extra communication.

It is important to consider that the Ray objects will between MPI ranks in parallel. Therefore all of the data in Listing 7.1, including all of the values in the arrays, is to be packed up (serialized), communicated, then unpacked (de-serialized) on the receiving end. This is achieved using a libMesh mechanism where pack() and unpack() routines are created for the type. These routines fill/read raw data buffers to be communicated by MPI. For this thesis, libMesh was enhanced to asynchronously communicate arrays containing these packable types.

As a Ray is packed and communicated to another MPI rank, the current rank no longer needs to hold this object in memory. This is one of the advantages of fully tracing rays from domain boundary to domain boundary: no partition boundary angular fluxes need to be stored. However, it was found that the creation/destruction of Ray objects and re-allocation of the data array was slow. Therefore, a memory pool keeps a small number of reusable Ray objects on each rank. More detailed information about the memory pool can be found in §7.3.3.

7.2 ELEMENT TRAVERSAL ALGORITHM

Before moving into parallel ray-tracing, it's important to discuss how rays can move through the mesh on one partition. Ray-tracing within unstructured mesh heavily relies on the connectivity structure present within the mesh. As shown in Algorithm 7, each ray begins at the edge of a particular element. Each side of that element is tested for intersection with the ray. The connectivity of the elements is then utilized to find the next element the ray moves to, and then the algorithm repeats. By testing each side and moving to neighboring elements, the ray can be traced entirely across the domain from one domain boundary to another.

Algorithm 7 works on one ray at a time, moving it from the starting position to its ultimate end. The end can be the boundary of the domain, specified end-point within the domain or the edge of a process's partition. For `MOCkingbird` it will always be the domain boundary. Each iteration of the loop in Algorithm 7 moves the ray forward through one more element. This is accomplished by testing the intersection of the ray with each side of the current element.

Efficient infinite precision calculations would allow for ray-tracing routines to be exact. However, computers work with finite-precision, and therefore ray-tracing routines rely on tolerances for finding intersections. These tolerances, finite-precision arithmetic, and rays exactly hitting nodes can lead to situations where more than one or even no sides are found to intersect the ray. This leads to all of the "corner cases" found within Algorithm 7 which will be discussed in detail in §7.2.1 below.

At the point where an intersection is found, Algorithm 7 calls back to the application (such as `MOCkingbird`) to allow it to use the newly found "segment" (part of the ray from the current position to the new intersection point) for calculation. The segment calculations within `SMART` are completed using an object-oriented system named `RayKernel`. If the new intersection point is located on the domain boundary, then a `RayBC` object is invoked to apply any relevant boundary conditions. Both of these systems are discussed in §7.3.4.

One significant feature of Algorithm 7 is that it is agnostic of dimension. By abstracting the idea of traversing through elements using element connectivity, the core ray-tracing capability can operate in one-dimension (1D), two-dimensions (2D) or three dimensions (3D) with no restrictions on the domain shape (other than being meshable). The `SMART` algorithm currently works with geometry made up of line, quadrilateral, triangular, or hexahedral finite-elements. It would be straightforward to extend this to other element types such as wedges, pyramids or tetrahedral elements.

While the core algorithm remains unchanged regardless of dimension or element type, the side-intersection testing code is specialized depending on the type of element side being intersected. In 1D, on line-elements, the intersection testing is straightfor-

Algorithm 7: Element Traversal Algorithm

```
1  current_elem ← starting element;
2  current_point ← starting point;
3  while not finished with ray do
4      for each side of current_elem do
5          test intersection
6      end
7      if no intersection then
8          for each node on current_elem do
9              test for intersection
10         end
11         if still no intersection and in 3D then
12             for each edge on current_elem do
13                 test for intersection
14             end
15         end
16         if still no intersection then
17             for each side of current_elem that lies on the domain boundary do
18                 use floating-point test to see if the end of the ray lies on that side
19             end
20         end
21         if still no intersection and very near domain boundary then
22             apply boundary condition
23             if ray shouldn't continue then
24                 break loop
25             end
26         end
27         if still no intersection and near node or edge then
28             for all elements that touch this point do
29                 look for longest path out
30             end
31         end
32         if still no intersection then
33             raise ERROR
34         end
35     end
36     if intersection found then
37         execute RayKernel objects on segment
38         if ray shouldn't continue then
39             break loop
40         end
41         if neighbor exists across intersection then
42             current_point ← optimal intersection;
43             current_elem ← neighboring element;
44         else
45             apply RayBC
46             if at domain corner then
47                 apply all RayBC objects at corner
48             end
49         end
50         if ray shouldn't continue then
51             break loop
52         end
53     else
54         raise Error
55     end
56 end
```

ward (only the left/right direction determines the intersection with the end of each element). Two dimensional side-intersections are all completed using the line-line intersection algorithm shown in Algorithm 21. Three-dimensional intersections, with the quadrilateral sides of a hexahedral element, are handled by breaking the quadrilateral into two triangles for performing intersection tests. These intersection algorithms are detailed in Appendix 14.

7.2.1 *Corner Cases*

Unstructured mesh provides many opportunities for both figurative and literal corner cases that must be carefully considered. For instance, a ray directly striking a junction between four quadrilateral elements has difficulty traversing through the connectivity structure to emerge on the other side. This is due to floating point tolerances inherent to finite-precision arithmetic. Due to tolerances, the intersection algorithm can get stuck in an infinite loop, alternating between intersecting sides that all meet at the node. These cases are explicitly handled within the ray-tracing capability with specialized code that searches for pathways away from the current intersection point. Each of these cases is examined in this section.

Within a three-dimensional (3D) mesh there are seven main cases to handle, in order of most likely to least likely:

1. Intersecting a face of the current element
2. Intersecting an edge (the line between two nodes of a three-dimensional element)
3. Intersecting a node
4. End of ray lies on a face of the current element
5. Current point is already extremely near a boundary and near the end of the ray
6. Ray cannot find an intersection within the current element, need to look for a way out through a neighbor
7. Ray cannot continue (should never happen)

These seven cases are each explicitly handled within the element traversal algorithm, as shown in 7. For efficiency, the six cases are handled in the order given above with one exception: intersecting a node is checked before intersecting an edge. This is done because node-intersection is a much more severe case than edge intersection and would most-likely be misinterpreted as an edge intersection.

When a ray strikes an interior node or 3D edge, each element connected to the node/edge of the current element is inspected to see if it is a candidate for the ray

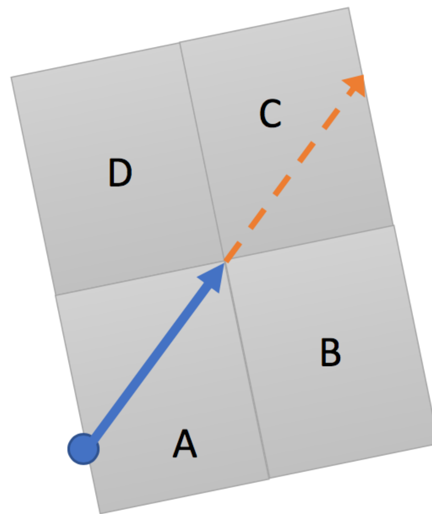


Figure 7.1. Pictorial representation of an internal corner strike. The blue ray is hitting an internal shared corner and the orange intersection will be the next segment chosen.

to leave through. Longer path lengths are preferred; otherwise, a ray may infinitely cycle through all of the elements meeting at the point due to floating-point tolerances. Therefore, each element surrounding the point is searched, with the longest path out of the corner chosen.

This same code path (choosing the neighbor with the longest path out) is also used in case 6 where the incoming point on an element is extremely close to a node (but did not hit it). In this case, all other intersection algorithms may fail due to floating-point arithmetic. The best that can be done is to find a neighbor, which also contains the incoming point, and has the longest path away from the current intersection point and move to that neighbor. This case is handled last, not only because it would be better for one of the other intersection algorithms to work, but also because this search can be expensive.

Figure 7.1 shows an example of the process where a ray needs to move through the neighbor with the longest distance out. The longest distance is chosen to move away from the current position, avoiding floating point issues that can lead to infinite looping. The (blue) ray is moving across element A and strikes a shared node between multiple quadrilateral elements. Elements B, C, and D are searched for an intersection from the shared node along the path of the ray. The intersection from the shared corner through element C is the longest and is chosen as the next segment (shown in orange). It should be noted that this type of movement "across" an internal corner can be problematic for MOC acceleration algorithms such as Coarse Mesh Finite Difference (CMFD) [65], due to a lack of balance of current through the element sides, and therefore may need to be rethought in the future.

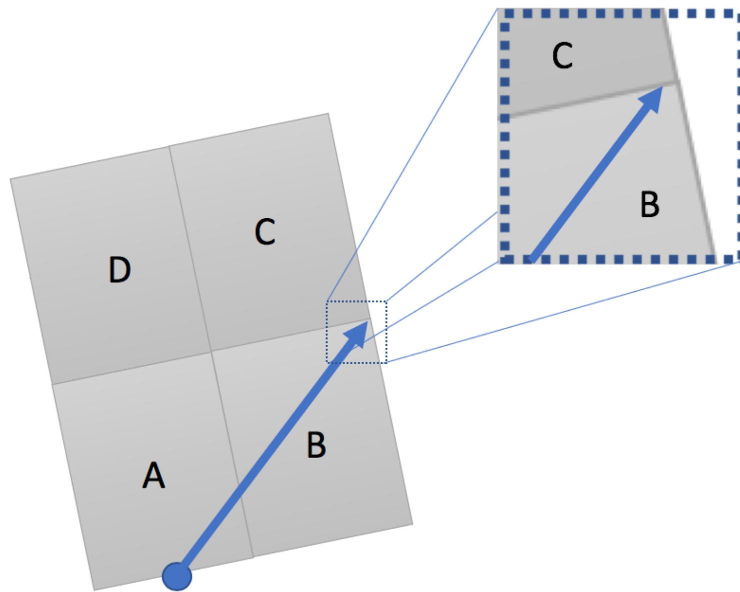


Figure 7.2. Pictorial representation of an intersection very near an external domain boundary.

In the case where it can be detected that the end of of the ray lies within the face of the current element, case 4 is used to short-circuit all other logic. Detecting the "end" of a ray touching a face is particularly troublesome with floating-point round-off. If through floating point precision loss, it's deemed that the end of the ray is just short of the face it's supposed to hit on the boundary; then no intersection is found. Therefore, this case is explicitly checked with floating-point tolerances. A positive check short-circuits all other logic to end the ray at the end point.

A ray intersecting an element at a point near a boundary (within floating point tolerance) but not an element side that lies on the boundary may not be able to continue to the boundary. In this case, as shown in Algorithm 7, the ray is considered to have met the domain boundary. Next, each side in the current element that is on a boundary and (within floating point tolerance) could contain the current intersection point is searched. The intersection point is then assigned to that side, ultimately causing the boundary condition to be applied to the ray and the ray-tracing to end for that ray.

Figure 7.2 demonstrates this case. The ray is passing through element B and strikes the side of element B neighboring element C. However, with floating-point tolerances, the intersection point on the side between elements B and C can be considered to be at the boundary. It should be noted that in Figure 7.2, the intersection shown is exaggerated away from the shared corner for ease of demonstration. Within `MOCkingbird`, the tolerance for this check is currently $5e-5cm$. This is detected within the ray tracing algorithm, and the sides of element B are searched to find one that both lies on the external boundary and (within floating point tolerance) "contains" the intersection

point. Any boundary conditions associated with that external boundary are applied (possibly reflecting the ray for instance).

7.2.2 *Corner Case Rarity*

While the above section detailed the way the code deals with corner cases, these events are very rare in practice. The vast majority of intersections are found using case 1: ray/face intersection. As a demonstration, a 2D lattice of 144x144 pin-cells containing 26,542,080 elements was used with 128 azimuthal angles and 0.001cm spacing to do one transport sweep. During the sweep, counts are kept for each of the cases from the previous section. Only three cases occurred: 1, 4 and 6.

Table 7.1. Ray tracing statistics for one transport sweep.

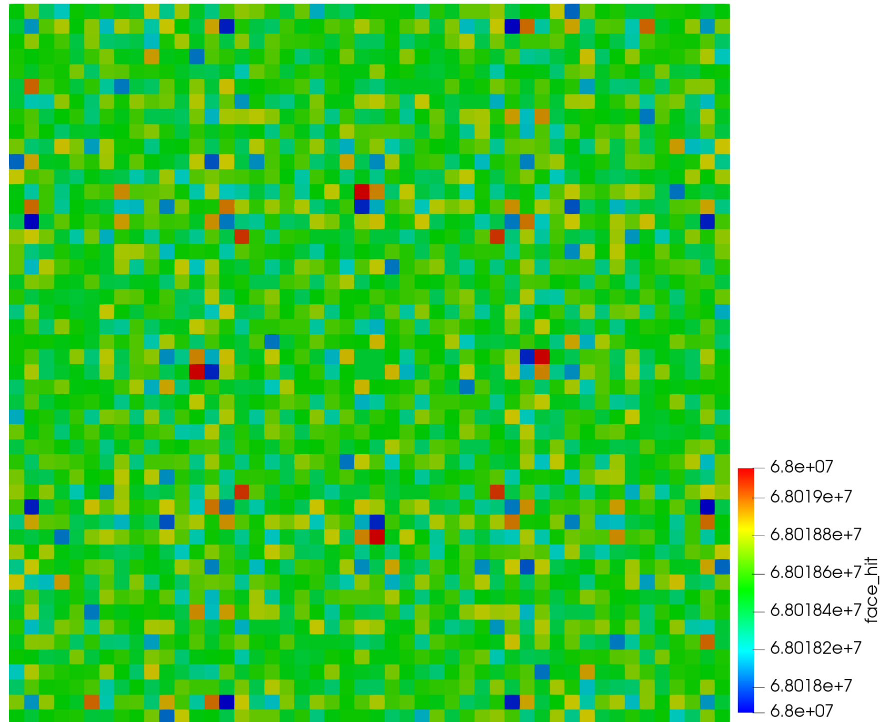
Case	Count	Percent
Total	156714657017	100
Face Hit	156714525672	99.99991619
Moved Through Point Neighbor	130584	4.85595932e-7
End On Boundary	761	8.33259648e-5

The data for how often each case occurred can be found in Table 7.1. Nearly all intersections were completed with the "regular" element traversal algorithm which finds intersections of the rays with the element faces. However, just over 130k intersections needed to do extra work to allow the ray to continue.

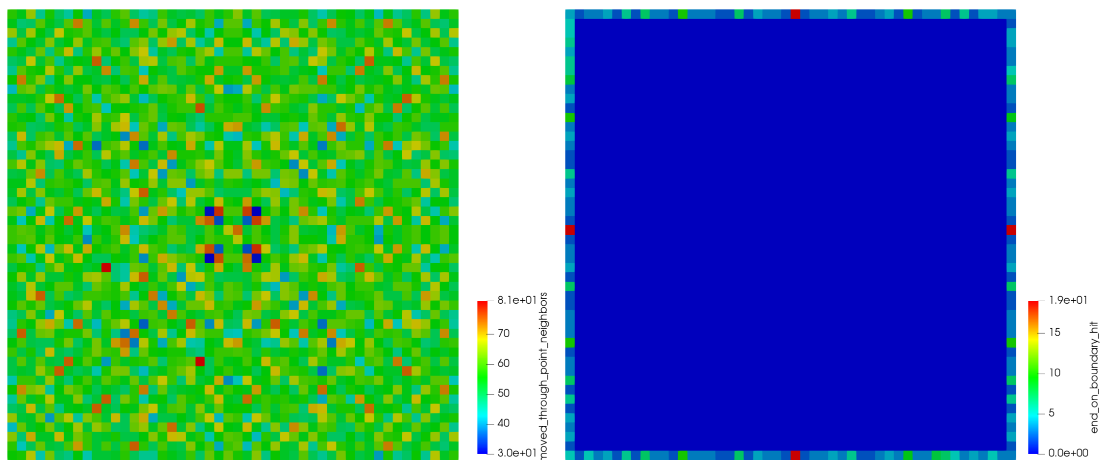
Figure 7.3 shows the spatial distribution within the domain for each of the three intersection cases that occurred for this run. The run was executed using 2304 MPI ranks and the `GridPartitioner` in a 48x48 grid. The totals shown in Figure 7.3 represent the total counts for each intersection case on the MPI ranks, which owned that portion of the geometry.

Figure 7.3a shows that each MPI rank performed nearly the same number of intersections: about 68 million. Some obvious symmetry is visible due to the symmetry in the track generation and the symmetry of the domain. Figure 7.3b details how the maximum number of case 6 (a ray moving out through a neighbor) on any MPI rank was just 81 out of the 68 million intersections performed on each rank. Finally, the MPI ranks where rays ended by following case 4 are shown in Figure 7.3c.

While the number of times a ray needs to use a more complicated algorithm to move through the mesh is vanishingly small, these cases cannot be ignored. Even just one single failure in billions is enough to kill a MOC solve. Therefore, even though these events are rare, handling them is critical.



(a) Face hit.



(b) Moved through point neighbor.

(c) End on boundary.

Figure 7.3. Spatial distribution of intersection cases. The totals shown are the amount that happened within each MPI rank that owned that portion of the geometry.

7.3 PARALLEL RAY TRACING: SMART

While the previous section detailed how to move rays through an unstructured mesh from one end of the domain to the other, there was no parallel capability. To make ray-tracing scale in parallel, the code needs to be able to communicate rays from one MPI rank to another. This is what `SMART` was built to do. `SMART` could be thought of as a more advanced version of Algorithm 6 that allows for sparse, one-sided, multi-hop communication of `Ray` objects through the domain.

7.3.1 *Overview*

The `SMART` algorithm is completely asynchronous: all parts of the algorithm are occurring simultaneously on every MPI rank. At every moment, every rank is doing the work that it can while simultaneously pulling in more work and pushing finished work to its neighbors. Here, "work" means using the algorithm in §7.2 to trace rays through the portion of the domain assigned to it by mesh partitioning. The communication is performed using non-blocking, asynchronous, point-to-point MPI communication with the primary objective being to overlap communication and computation in order to gain parallel efficiency.

The major "tasks" to be completed are:

1. Generate rays claimed in 8.1
2. Check for incoming rays and pull them into buffers
3. Use Algorithm 7 to trace a subset of rays called a "chunk"
4. Collect finished rays into buffers to be sent to other MPI ranks and send them
5. Check for completion

Again, it should be stressed that all of these tasks are happening simultaneously and overlapping with each other. Within `SMART` these tasks are grouped into two phases: "generation" and "propagation." During generation, MPI ranks create rays and begin tracing them in subsets called "chunks." A chunk is a set of rays that are given to the element traversal routine at one time. The idea is to start some parallel communication, then go do some work tracing a set of rays, then repeat. This is necessary so that incoming rays can periodically be checked for, in-between tracing (Step 2). Ranks without rays to generate, skip directly to propagation. A rank which starts by generating rays enters the propagation phase once it has generated all rays. During propagation, Tasks 2-5 are continuously executed until all rays have been completely traced.

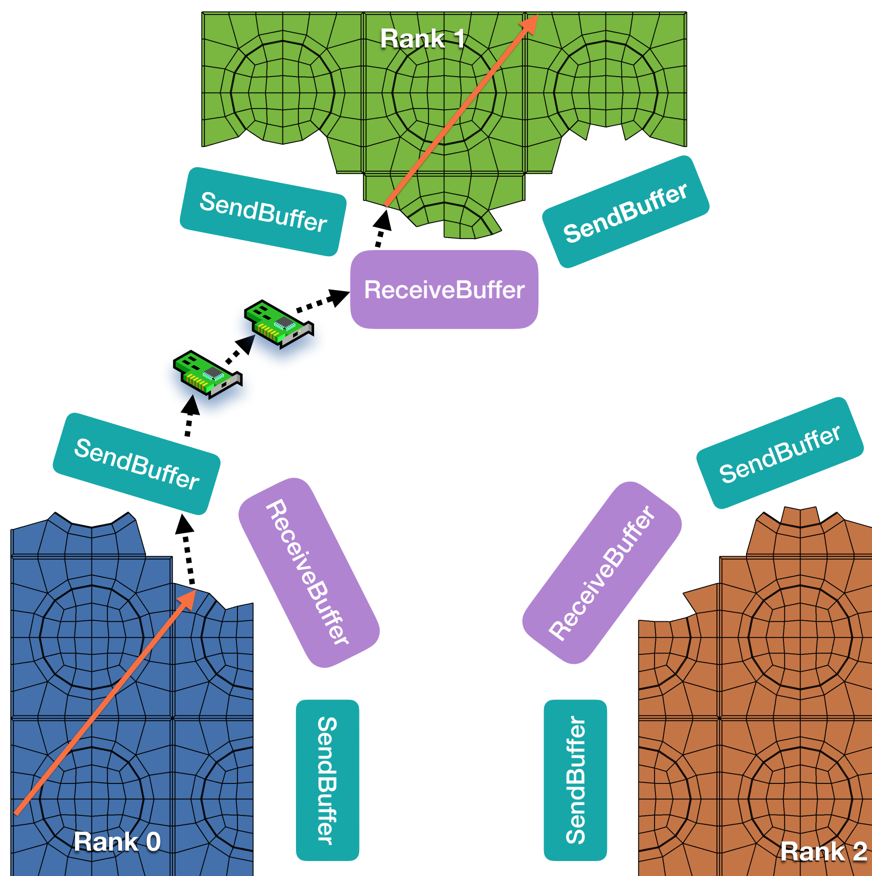


Figure 7.4. High-level overview of how a `Ray` traverses a domain decomposed unstructured mesh.

A high-level pictorialization of how one Ray traverses a domain decomposed unstructured mesh is shown in Figure 7.4. Each MPI rank owns a ReceiveBuffer and several SendBuffer objects, one for each MPI rank it sends to. With MOCkingbird a Ray starts at a domain boundary, in this case on MPI rank 0. The algorithms in §7.2 are then used to move it through the unstructured mesh elements until it reaches a partition boundary. At a partition boundary, the Ray is then added to the SendBuffer for the MPI rank it needs to be sent to. As explained momentarily, once that SendBuffer fills, it is then given to the network hardware to be asynchronously communicated to the other MPI rank.

On the other MPI rank (rank 1 in this case), an asynchronous receive is started to allow the network hardware to pull the Ray objects into the ReceiveBuffer. Once rank 1 is nearly out of work to do (which will be more concretely defined in Algorithm 10), it gets the rays from the ReceiveBuffer and begins tracing them through its partition until they reach the domain boundary (or another partition boundary).

This process is covered in great detail in the following sections.

7.3.2 *Algorithms*

All work during the execution of SMART flows into and out of the work_buffer. The work_buffer is a custom-coded circular buffer. The work_buffer holds smart pointers to Ray objects (std::shared_ptr<Ray>). It is designed to reduce memory allocation/deallocation and allow for a first-in-first-out (FIFO) workflow. In addition, it is possible to retrieve a contiguous range of Ray objects from the front of work_buffer (a chunk) and pass it to the element traversal routine.

FIFO is important for SMART. In general, rays placed into the work_buffer earlier have a longer distance to travel (and thus should be traced earlier so they can move to the next rank). This is especially true on the boundaries: rays created during the generation phase are filled into the work_buffer first. Then, as those rays are traced in chunks, more rays are added to the work_buffer from incoming messages. However, rays generated on the boundary have the longest distance left to travel, whereas rays incoming to ranks holding boundary elements are, by definition, nearly finished tracing. Therefore, it is vital to move generated rays off of boundary ranks as quickly as possible and wait until later to trace rays incoming to the boundary rank.

A similar issue exists to a lesser extent in the interior of the domain. Although, once the interior of the domain is saturated with rays being traced, the effect is diminished. Tests were completed which gave specific priority to tracing rays which still had the longest distance to travel without much improvement and therefore that mechanism is not used in any of the results presented in this work (though the capability still resides within the code).

Before the `Ray` can be added to the `work_buffer`, the starting element, position, and direction are set for the `Ray`. Without it, the tracing of the `Ray` might not be deterministic. Small differences in floating point round-off could send the `Ray` down a slightly different path in subsequent source iterations. If this were to happen then the algorithm is no longer deterministic and may not converge. The handling of track information to make this possible will be covered in §8.1.

Algorithm 8: `execute()`: SMART start.

1 Call Algorithm 9
2 Call Algorithm 12

As previously mentioned, `SMART` is composed of two phases: generation and propagation. As shown in Algorithm 8, this amounts to two function calls within the main `execute()` method on the `RayTracingStudy` object (which will be examined in §7.3.4). Also explained within §7.3.4, is the way applications can override the `generate()` function. As will be shown in §8.2.3, `MOCkingbird` overrides it to start rays associated with MOC tracks along the boundary. Here, in Algorithm 9 a generic algorithm is given for generation.

Algorithm 9: `generate()`: Ray Generation

1 `local_rays_started` $\leftarrow 0$
2 `work_buffer` $\leftarrow$ Initialize empty CircularBuffer
3 **for** *ray in rays_to_start* **do**
4   `work_buffer` $+=$ ray
5   `local_rays_generated++`
6 **end**
7 Call Algorithm 10 with `start_receives_only` = True

Keeping track of the number of rays that have been generated (as is done in Algorithm 9 is critical to `SMART`. Each MPI rank tracks this number separately. Later, this number is (asynchronously) summed and ultimately used to know when to stop.

Finally, Algorithm 9 ends by calling Algorithm 10. When it does, it specifies that although receive operations can be started by Algorithm 10, those receive operations should not add to the `work_buffer`. The reason to not add to the `work_buffer` at this time is that there is, presumably, already a lot of work in `work_buffer` and checking for finished receive operations can be expensive. Therefore, it's best to trace existing work, waiting until later to pull in new work.

Algorithm 10 is an important piece of `SMART`. In particular, this is the most direct place to observe the overlap of communication and computation. Algorithm 10 will remove a `chunk_size` amount of rays from the `work_buffer` and pass them to Algorithm 11 for processing. However, as it is doing that, Algorithm 10 also overlaps the

Algorithm 10: `chunkyTraceAndBuffer()`: traces chunks of rays from the `work_buffer` while also receiving incoming rays.

```
1 input: start_receives_only a boolean specifying whether incoming rays should
    be added to the work_buffer or not
2 while work_buffer not empty do
3   if start_receives_only or work_buffer size > 2 * chunk_size then
4     Use ReceiveBuffer to start asynchronously receiving rays without adding
        received rays to the work_buffer
5   else
6     Use ReceiveBuffer to start asynchronously receiving rays and add
        received rays to the work_buffer
7   end
8   Call Algorithm 11 to trace current_chunk_size amount of rays from the
      work_buffer
9 end
```

tracing of rays with the receiving of new rays to be traced. This is key to maintaining parallel efficiency. By doing this, as work is completed, new work already resides within memory for this MPI rank.

When explicitly specified (by passing `start_receives_only=true`) or when enough work already exists locally, Algorithm 10 will specify that the `ReceiveBuffer` should only *start* receive operations without adding to the `work_buffer`. As has already been mentioned, it is generally advantageous to trace rays that are already in the `work_buffer`. However, incoming messages containing rays should not be completely ignored. Ignoring them fills up MPI buffers with incoming message data and causes a slowdown in execution. Therefore it is important to start asynchronous receives so that data can be transmitted in the background.

The process of completing a receive can take time. In particular, using `MPI_Test` to check if a receive is finished takes time. Therefore, while there is still work left in the `work_buffer`, or if explicitly specified, it's better to only *start* asynchronous receives and not check if they're finished. Receives should be completed (and their contents added to the `work_buffer`) only once the current `work_buffer` is depleted (less than two chunks remain). This amount, of *two* chunks, is arbitrary but was found to work well across a wide selection of problems.

Algorithm 10 calls Algorithm 11 with a contiguous chunk of rays to be traced across the local mesh partition. Algorithm 11 loops over each ray in the chunk, calling 7. After each ray has been traced across the local partition, it is then placed in a `SendBuffer` to be sent to the next partition or, killed. Note that a `RayBC` may have added the ray to a `RayBank`. More information is given about the `RayBank` in §8.2.3. If the `Ray` reaches the edge of the domain, then the `local_rays_finished` counter is incremented. This local counter is used in the stopping criteria in Algorithm 12.

Algorithm 11: `traceAndBuffer()`: calls Algorithm 7 to trace a contiguous array of `Ray` objects across the local domain then adds `Ray` objects which hit partition boundaries to `SendBuffer` objects and `Ray` objects which intersect boundaries to `RayBank` objects.

```
1  input: A contiguous set of Ray objects to be traced through the local partition
2  foreach ray in rays do
3      Call Algorithm 7
4      if ray intersected partition boundary then
5          Add ray to the correct SendBuffer
6      else if ray intersected domain boundary then
7          local_rays_finished++
8      else
9          Ray ended within domain, kill it (does not happen in MOC)
            local_rays_finished++
10     end
11 end
```

The final major algorithm within `SMART` is the "propagation" loop which is responsible for checking for new work and testing for the finishing criteria. Algorithm 12 shows the propagation loop. This algorithm is called on each rank once generation is finished (possibly immediately if there is nothing to generate on the local rank).

The main feature of Algorithm 12 is the "while" loop. It loops indefinitely, checking for incoming rays and calling Algorithm 10 to trace them. Note that once Algorithm 10 is called, that algorithm itself continues to pull in new work until the rank is ultimately work starved (no work left to do). This is by design; it keeps the cores focused on tracing rays while receiving data in the background and not losing time checking for the finishing criteria, which is toward the end of Algorithm 12.

The next piece of Algorithm 12 deals with "forcing" the `SendBuffer` objects to send their current contents. As is explained in §7.3.3, the `SendBuffer` objects buffer rays to be sent to neighboring MPI ranks until the buffer is full and then it is sent. However, after propagating the majority of the rays through the domain, a point is reached where MPI ranks no longer have any incoming work. In this situation, the `SendBuffer` objects have rays in partially filled buffers waiting to be sent. The algorithm detects this state and tells the `SendBuffer` objects to send their partially filled buffers.

The final part of Algorithm 12, is to check the stopping criteria. Again, `SMART` can be thought of as an advanced version of Algorithm 6 and therefore the stopping criteria is similar. A non-blocking `MPI_Iallreduce` is used to sum the total number of rays generated by all ranks (this happens in the background while rays are traced). Later, when ranks run out of work, non-blocking `MPI_Iallreduce` is used again to sum the total number of rays finished. If the two match, then everything is done and the algorithm completes.

Algorithm 12: `propagate()`: Called after Algorithm 9 to continue to receive, trace and communicate `Ray` objects. Also responsible for checking for the finishing criteria.

```
1  Start nonblocking MPI_Iallreduce to sum total_rays_generated
2  while not finished do
3      Check for and possibly receive rays
4      if rays in work_buffer then
5          Call Algorithm 10
6      else
7          Force all SendBuffer objects to send
8      end
9      if Finished MPI_Iallreduce for total_rays_generated then
10         if Finished or not started MPI_Iallreduce for total_rays_finished then
11             Start nonblocking MPI_Iallreduce to sum total_rays_finished
12         end
13     end
14     if Finished MPI_Iallreduce for total_rays_generated and
         total_rays_finished and total_rays_generated == total_rays_finished
         then
15         finished ← true
16     end
17 end
```

It's important to note that it may be necessary to check the finishing criteria, using `MPI_Iallreduce`, multiple times. This happens as the number of rays left to trace is low. As an example, if only one ray is left in the domain, as it is handed off from one rank to the next, each rank it passes through believes that the simulation is finished and could, theoretically, complete the `MPI_Iallreduce`. Each time it is completed it is started again. In practice, anywhere between 3-10 `MPI_Iallreduce` calls were observed.

The `MPI_Iallreduce` to retrieve the number of rays that have finished is not complete until every rank has called it. Once the `MPI_Iallreduce` is complete, all ranks check the output value. If that number is not equal to the number of rays started, then the ranks keep doing work until they run out of work again. When a rank is out of work, it contributes to the next `MPI_Iallreduce`. The reason there are so few total completions of `MPI_Iallreduce` is that they only complete when the sweep is nearly finished, and all ranks are out of work.

All of these algorithms together comprise `SMART`. By overlapping generation, communication and ray-tracing, `SMART` is able to keep cores working and, as is shown in §8.3, is able to keep parallel efficiency high. The next section details some of the data structures used by these algorithms.

7.3.3 *Data Structures*

The previous section outlined the algorithms comprising SMART. Within those algorithms, several data structures are utilized: SendBuffer, ReceiveBuffer, and an ObjectPool. These data structures are critical to the working of SMART and warrant some detail.

7.3.3.1 *Object Pool*

During a typical invocation of SMART, many millions (possibly billions) of Ray objects are created, traced, communicated, and ultimately destroyed. However, as shown in Listing 7.1, a Ray object has many data members, and the data arrays can be large. For instance, when using a set of 70 energy groups for an MOC calculation, the _data member contains 70 double-precision floating-point values: one for each energy group making up the angular flux. Because of this, creating and destroying (allocating and deallocating) millions or billions of these objects would not be efficient.

To rectify this problem SMART utilizes an "object pool." It reduces the need to ask the operating system to allocate memory, which is a slow process. However, an object pool goes one step further since it contains completely constructed objects. This not only bypasses the need for retrieving memory from the operating system but it also eliminates construction time.

The ObjectPool utilized by SMART is shown in Listing 15.1. The design of this object started from [102] and has been enhanced to meet the needs of this thesis. The ObjectPool can hold any C++ object and is therefore utilized in multiple places within SMART. It is general enough that it was added to MOOSE as a utility.

To use the ObjectPool, code that would normally construct an object using new TheType(args) would instead ask for the object from the pool using: pool.acquire<TheType>(args). If an object of TheType already exists within the pool, then it is returned. If the pool is empty, then a new object is constructed and returned.

The way the pool works is in the custom deleter, called ExternalDeleter in Listing 15.1. The ExternalDeleter maintains a pointer to the ObjectPool. When the object is deleted, the custom deleter is called, returning the object to the pool instead of destroying it.

The exact C++ mechanisms behind the usage of unique and shared pointers is beyond the scope of this thesis. However, it is essential to point out that the custom deleter is called automatically anytime the object is no longer referenced. This allows for objects like the Ray object to be held in multiple buffers. A Ray might end up in multiple buffers due to the way boundary conditions might be implemented by the code using SMART. For instance, MOCkingbird boundary conditions add the ray to a RayBank, but that Ray will also temporarily remain in the work_buffer. Once the Ray

is no longer needed (such as if it is communicated to another MPI rank), the `Ray` is automatically returned to the pool.

Considering how the `ObjectPool` works with the `Ray` is important. The `Ray` object itself has some data members that are reset during `acquire()`. However, the `_data` member is not reset but it is zeroed. Therefore, when using many groups (many of the studies in Chapter 8 use 70) the `_data` member of a `Ray` object that is being reused from the pool is the correct size, zeroed, and ready to be traced.

Finally, the memory use of the pool is must be discussed. Each MPI rank has an `ObjectPool` for efficient retrieval of `Ray` objects. Although, it was previously said that `SMART` does not need to store any data (angular fluxes) along partition boundaries (which is true), utilization of the `ObjectPool` will mean that there is some persistent memory within each MPI process for `Ray` objects which contain allocated data arrays (for `MOCkingbird` that length is set by the angular flux). MPI ranks along the domain boundary request enough `Ray` objects from the pool to be able to start them all during the generation phase.

Interior MPI ranks (those with partitions assigned which don't touch the domain boundary) make as many `Ray` objects as are needed concurrently during the execution of `SMART`. For instance, a rank which ultimately has 1000 `Ray` objects trace through its domain does not need to create 1000 `Ray` objects. Instead, it creates as many `Ray` objects as the maximum which need to be held simultaneously. If `Ray` objects are received in bunches of 10, then that number might be just 10: with the same 10 `Ray` objects being reused hundreds of times. As shown in §8.3.1 this is less than if the boundary fluxes were explicitly stored.

The amount of memory used by the `ObjectPool` for storing `Ray` objects and other things within `SMART` (such as buffers for sending and receiving) is further explored in the benchmarks in Chapter 8.

7.3.3.2 *Communication Buffers*

The two buffer objects: `SendBuffer` and `ReceiveBuffer` are both used for asynchronous communication. A `SendBuffer` is created for each neighbor an MPI rank needs to send to. As rays are traced by the rank, they contact a partition boundary and are then be added to the correct `SendBuffer` for later communication. Only one `ReceiveBuffer` gets created for each MPI rank. The reception of all `Ray` objects comes through that one `ReceiveBuffer`.

Both types of buffers use asynchronous MPI: `MPI_Isend` for `SendBuffer` and `MPI_Irecv` in the `ReceiveBuffer`. In asynchronous MPI, two things are needed for communication a "buffer" (allocated memory space) to either send or receive from/to and a `MPI_Request`. The `MPI_Request` object keeps track of the state of the communication, whether it is in progress or complete. Therefore, the primary data member of both

of these buffers is a std::list<std::pair<bufftype, MPIrequest». When the SendBuffer sends a data buffer to a remote rank, it creates a MPI_Request and joins them together in a std::pair that is added to the list of in-progress communication.

The ReceiveBuffer works similarly, though it needs to detect the incoming message first. To do so, it utilizes MPI_Iprobe. MPI_Iprobe returns a boolean/integer which signals whether or not an incoming message exists. If there is an incoming message, it also returns a MPI_Status object with information about the incoming message including the length. In the case where the ReceiveBuffer detects an incoming message, it uses the information in the MPI_Status object to size a data buffer and creates a MPI_Request. The MPI_Request is then used to call MPI_Irecv to begin the nonblocking reception of the message. Finally, it creates a pair of the buffer and the MPI_Request and adds them to the list of in-progress communication.

Deciding when to send a buffer is a parameter within the SMART algorithm. For sending, the main parameter is the max_buffer_size: how many Ray objects to queue up for sending before starting communication. As shown in Chapter 6, MPI excels at sending small to medium length messages. Therefore, it is important to choose a good maximum buffer length. However, the nature of asynchronous communication means that the performance of the overall algorithm is not very sensitive to this parameter. All of the problems in the this thesis utilized a buffer size of either 100 or 200 Ray objects. This would equate to about 60-120 kB of memory with 70 energy groups in 3D.

As mentioned earlier, sometimes it is necessary to "force send" (tell the SendBuffer objects to send what they currently have buffered even if it's only partially full). This happens towards the end of a ray-tracing execution when only a few rays are left in the system, and they need to be expedited through the mesh.

7.3.4 *Object Oriented Design*

One design goal for the ray-tracing component of MOCkingbird was to make it general and separable so that it can be utilized by anyone using MOOSE and needing ray-tracing capability. Therefore, the SMART algorithm has been encompassed in a C++ object called RayTracingStudy. The RayTracingStudy is meant to be inherited from, with the intent of overriding the generate() function. In this way, MOCkingbird can perform ray tracing for deterministic MOC by creating an object called TrackListStudy which utilizes the list of tracks found in §8.1 to generate rays.

In addition to ray generation being customizable, the new ray_tracing module within MOOSE also provides a "plug-in" mechanism for specification of physics, boundary conditions, and material properties. The most important of these is the RayKernel:

a base class that can be inherited from to create calculations which occur at every intersection of a `Ray` with the mesh.

Listing 7.2. `RayKernel` base class.

```
class RayKernel
{
public:
  RayKernel(const InputParameters & params);

  /// Called at the beginning of a new Ray trace
  virtual void rayStart() {}

  /**
   * Called on each Segment
   * @param start The beginning of the segment
   * @param end The end of the segment
   */
  virtual void onSegment(const Elem * /*elem*/,
                         const Point & /*start*/,
                         const Point & /*end*/) = 0;

  /// Set the current Ray that's being worked on
  virtual void setRay(const std::shared_ptr<Ray> & ray) { _ray = ray.get(); }

protected:
  /// Number of groups
  unsigned long int _num_groups;

  /// The Ray that's being worked on
  Ray * _ray;

  /// Number of polar angles
  unsigned long int _num_polar;

  /// Offest into the vectors associated with the RaySystem
  dof_id_type & _current_offset;

  /// The current group values for this thread
  PetscScalar *& _group_solution_values;
};
```

An abbreviated declaration showing the `RayKernel` base class can be found in Listing 7.2. There is one virtual function an inheriting class must override: `onSegment()`. That function is each time the current `Ray` computes an intersection across an element. The `RayKernel` receives the current element and the beginning and endpoint of the segment (corresponding to the entry and exit point on the element's surfaces).

`onSegment()` is where computations such as calculating the new angular flux using Equation 2.18 can be accomplished.

The virtual function `setRay()` is called when a new ray is starting to be traced using a `RayKernel`. Within `setRay()`, the `RayKernel` will typically cache data from the `Ray` object such as quadrature weights and the position of the `_data` array. Any data from the `Ray` that need to be accessed often should be pulled out of the `Ray` and cached within the `RayKernel`.

In addition to these functions, a number of data members currently exist within the `RayKernel` base class. For now, the number of energy groups (`_num_groups`) and number of polar angles (`_num_polar`) are within the `RayKernel` base class. That will change in the future as the module is generalized further. The `_current_offset` is an index into the parallel solution vectors and changes based on the current element. This data member is utilized to access into `_group_solution_values` both for reading and writing.

The `_group_solution_values` data member is a parallel vector (a `libMesh PetscNumericVector`) which holds the main solution values being computed by the `RayKernel`. In the case of MOC, this corresponds to the new values of the group-wise scalar flux. More detail is given about how those values are arranged and accessed in Chapter 8.

Listing 7.3. `RayMaterial` base class.

```
class RayMaterial
{
public:
  RayMaterial(const InputParameters & params);

  /// Called on each segment so the material can recompute the sigma_t
  virtual void reinitSigmaT(const Elem * /*elem*/){};

  /// Called on each segment so the material can recompute all XS
  virtual void reinit(const Elem * /*elem*/){};
};
```

The base class for the `RayMaterial` is shown in Listing 7.3. It is simple, with only two virtual functions to possibly override. The purpose of a `RayMaterial` is to compute coefficients for use by `RayKernel` objects. In the most general case, that means overriding `reinit()` which gets called on each segment. For MOC, `reinit()` will be responsible for computing the cross sections. However, there is a special case just for MOC: `reinitSigmaT()` the purpose of which is a shortcut to compute Σ_t for use during the transport sweep. More detail is given about this in Chapter 8.

One important aspect of `RayKernel` and `RayMaterial` objects is that they can couple to any other variable in the `MOOSE` system. That includes the possibility of coupling

to variables like "temperature" and "density" to effect cross section calculation. This is one of the ways that `MOCkingbird` is well suited to multiphysics analysis.

Listing 7.4. `RayMaterial` base class.

```
class RayBC
{
public:
  RayBC(const InputParameters & params);

  /// Called on the boundary.
  virtual void apply(const Elem * elem,
                     const unsigned long int intersected_side,
                     const Point & intersection_point,
                     const std::shared_ptr<Ray> & ray) = 0;

protected:
  /// The Ray Problem
  RayProblemBase & _ray_problem;

  /// The Ray System
  RaySystem & _ray_sys;
};
```

The `RayBC` encompasses algorithms that are executed when a `Ray` strikes an external domain boundary. The `RayBC` is allowed to directly modify the `Ray` object that is passed into the `apply()` virtual function. Possible options include modifying the direction of the `Ray` (such as reflection), stopping the `Ray` or possibly storing the `Ray` for later retrieval (as is the case for MOC).

These three types of objects: `RayKernel`, `RayMaterial`, and `RayBC` are sufficiently general to allow many different types of physics to be modeled using the `SMART` algorithm. This thesis utilizes a `RayKernel` to update the angular flux and accumulate scalar flux for each segment, `RayMaterial` to compute cross sections and `RayBC` to create reflective and vacuum boundary conditions. However, many other options exist including particle transport, photon transport, laser modeling, line sources in finite-elements, and Monte Carlo neutron transport. All of these represent possible avenues of future study using `SMART`.

8 MOCKINGBIRD MOC

With mesh generation (§5), sparse communication algorithms (§6) and parallel ray-tracing (§7) developed, these capabilities can now be combined to create a scalable, parallel MOC application capable of 3D, full-core, neutron transport calculations called `MOCkingbird`.

This chapter proceeds by first discussing a critical capability for turning MOC tracks into the `Ray` objects needed by `SMART`, and uniquely determining their starting MPI process, position, and element. Next, a description of the parallel sweep is given. That is followed by descriptions of the `RayKernel` and `RayBC` objects `MOCkingbird` will plug-in to the `SMART` algorithm to perform a transport sweep. Finally, there is a detailed look at the scalability of `MOCkingbird` and the `SMART` algorithm, including comparison with other communication algorithms.

8.1 TRACK GENERATION AND PARALLEL `ray` CLAIMING

As described earlier, MOC tracks are created for integration of the angular flux during a transport sweep. Tracks have regular spacing, and their directions are determined by angular quadrature such that both space and angle are adequately covered. In addition, the track creation used here is "cyclic," meaning that all tracks "reflect" at the domain boundary to a track with a complimentary angle that is part of the quadrature. Tracks form complete cycles. Cyclic tracking is desirable for its ability to model reflective boundary conditions. The determination of the tracks, their starting, and ending positions is performed using track generation code from the `OpenMOC` project [2, 12].

As previously discussed, Parallel MOC solvers often employ "Spatial Domain Decomposition" (SDD) [22] where tracks begin and end at partition boundaries, and boundary angular fluxes are exchanged after each iteration. As was shown in 2.10, SDD requires the storage of angular fluxes at the intersection of every track with every partition boundary, necessitating the use of a large, fixed amount of memory within each process. While this may be a possibility for codes employing structured geometry that can split into equal pieces, unstructured mesh partitioning can create many partition boundaries which would create an excessive amount of memory utilization for `MOCkingbird`, this aspect of unstructured mesh MOC has previously been found to be an impediment to the creation of such a code [71].

To avoid this, `MOCkingbird` utilizes a "long characteristic" approach for MOC. For each source iteration, during the transport sweep, all tracks start at the domain boundary and are traced the full length of the domain until they intersect another domain

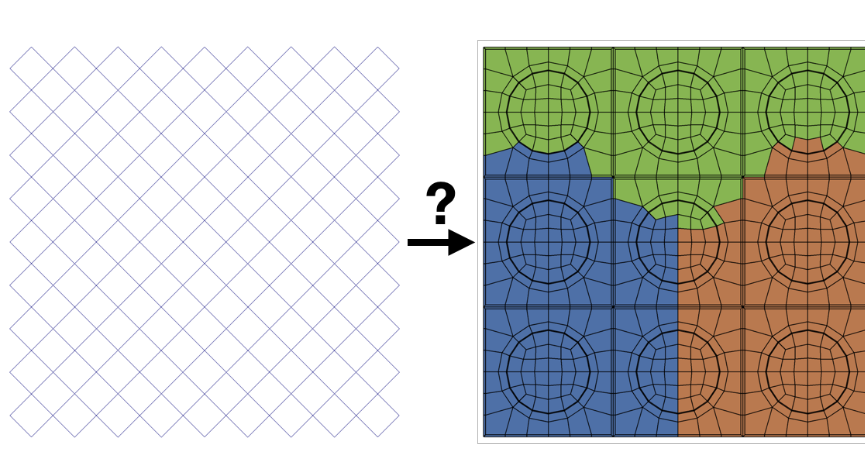


Figure 8.1. Cyclic tracks from OpenMOC need to be mapped onto the partitioned, unstructured mesh (in this case, partitioned for 3 MPI ranks). The starting position of each track must be uniquely located within a single MPI rank.

boundary. That behavior is consistent with a MOC code running in serial. Therefore, MOCkingbird has the same solver behavior (same iterations) in both serial and parallel. Also, there are no partition boundary angular fluxes which require storage, reducing the overall memory requirements of the application. However, as discussed in §7.3.3.1 MOCkingbird does use some memory for caching Ray objects, though this is less than if the partition boundary angular fluxes were stored as is shown in §2.10

Critical to this capability is the assignment of track starting positions. As shown in Figure 8.1, MOCkingbird must uniquely locate the MPI rank, and the element on that rank, which contains each track's starting position. Track generation routines from OpenMOC are utilized to generate the long characteristic tracks, in memory, during a startup phase. However, in 3D, it can be the case that merely generating all the start and end positions would overwhelm available memory on each MPI rank. For that reason, MOCkingbird employs a scalable multi-step process to create, claim, and communicate track starting information in parallel.

An overview of track claiming is shown in Algorithm 13. The algorithm begins by using OpenMOC functions to create tracks. From tracks, Ray objects are created and communicated to MPI ranks that might own the mesh element where the Ray would start. Next, a fine-grained search is done to attempt to locate the starting element for each Ray. Finally, a round of communication accomplishes the task of each ray being uniquely claimed by one MPI rank.

This set of algorithms is run once, at the beginning of the calculation. From that point on, the MPI ranks which claimed each Ray are then responsible for starting that Ray at the beginning of each transport sweep. After starting the Ray, SMART takes over and traces it to its final destination.

Algorithm 13: Overview of the track claiming algorithm.

1 Set up the track generator: Algorithm 14
2 Generate 2D tracks: Algorithm 14
3 Create `Ray` objects from the tracks: Algorithm 15
4 Find the MPI ranks holding nearby mesh: Algorithm 16
5 Communicate the `Ray` objects to MPI ranks which may hold their starting position: Algorithm 17
6 Search for a local element containing the start point of each ray: Algorithm 17
7 Uniquely claim each `Ray` by one rank: 18

What follows is a detailed explanation of each of the steps in the claiming algorithm. A central theme across these algorithms is a drive for parallelization and scalability. Ultimately, millions or even billions of tracks are used for calculation; therefore, this startup algorithm needs to be efficient.

Algorithm 14: Setting up the `OpenMOC` track generator.

1 Create a bounding box (BB) completely encompassing the entire mesh
2 Generate one `OpenMOC` `Cell` using that bounding box
3 Add "dummy" materials to `OpenMOC` for that `Cell`
4 Create an "inflated" local bounding box (BB) around this rank's mesh partition
5 Create an `OpenMOC` track generator using the geometry
6 Call `generate2DTracks()` on the `OpenMOC` track generator to generate the 2D tracks
7 Continue to Algorithm 15

The first step in this process is detailed in Algorithm 14, which sets up the `OpenMOC` track generator. A "bounding box" (an axis-aligned rectangle/rectangular prism in 2D/3D respectively) is created around the entire mesh. Bounding boxes, as shown in Figure 8.2, play a critical role throughout the setup process. The bounding box that encompasses the entire domain is known as a "mesh bounding box." For `OpenMOC`, this rectangular prism serves as a "`Cell`," which is the complete domain of the problem. The `OpenMOC` track generation routines also expect a certain amount of material data to be specified for each cell (in this case there is only one), so a dummy material object is created and provided to satisfy the interface. `MOCkingbird` only needs track information from `OpenMOC`; therefore, this material data is never used.

A "local bounding box" encompassing this MPI rank's partition is then created, as shown in Figure 8.2. This local bounding box is "inflated," all dimensions are increased by a small amount (currently $1e-8$cm), to account for floating-point round-off.

With the geometry, material data and bounding boxes created, the last step in Algorithm 14 is to call `generate2DTracks()`. This causes `OpenMOC` to create track information for all 2D tracks. As detailed in [12], 3D tracks can be generated on-the-fly. The 3D

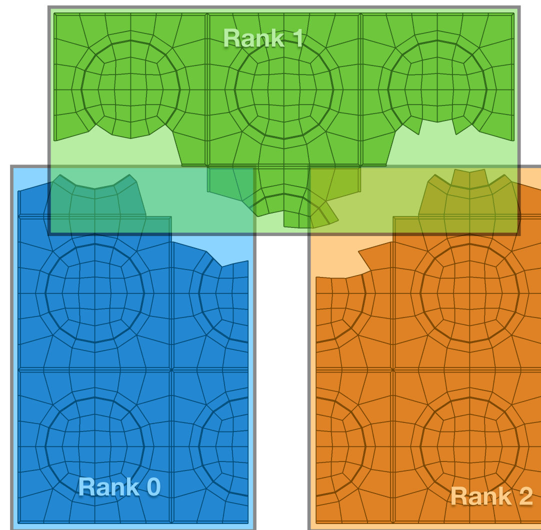


Figure 8.2. Local partition bounding boxes for three partitions.

tracks are based on the 2D tracks and have a closed-form expression for computing the starting and ending points.

The `OpenMOC` track generation routines are currently set up such that all MPI ranks must create all 2D tracks. This is the only serial portion of the track claiming algorithm. In practice, this isn't a burden. As is shown in §9.5, even for large simulations this step takes less than a minute. However, alleviating this requirement is the focus of future work.

The next phase of track claiming is shown in Algorithm 15. This is where `Ray` objects are created from each track. The 2D tracks in `OpenMOC` have IDs which are numbered from 0 to `num_2d_tracks`. Each track can be uniquely retrieved using its ID. Therefore, to parallelize `Ray` creation, the range of $[0, num_2d_tracks]$ is evenly split among all of the MPI ranks. Then, each MPI rank is responsible for creating `Ray` objects associated with the 2D tracks for the portion of the range it is assigned.

With the ranges of 2D track IDs assigned, Algorithm 15 then creates rays and stores them for later communication. If the domain is two-dimensional, the 2D track is directly turned into two `Ray` objects (one for the forward track and one for reverse). However, for three-dimensional domains, the 3D tracks associated with the 2D track need to be retrieved from `OpenMOC` then, similarly to 2D, turned into two `Ray` objects for each 3D track.

The range of 2D tracks assigned to the MPI ranks in Algorithm 15 is unrelated to the partitioning of the mesh. Each MPI rank is assigned a range based on its rank: rank 0 is assigned the first range, rank 1 the next, etc. Due to this, the start position of the `Ray` objects created by each MPI rank, most likely, won't fall within the mesh assigned to that rank. Therefore, in Algorithm 15, after the `Ray` objects are created,

Algorithm 15: Partition the range of 2D tracks, create Ray objects for each 2D track from the part of the range assigned to this MPI rank, check them against all bounding boxes, store them in a data structure for communication.

```
1  local_2d_track_range ← partitionRange(num_2d_tracks, num_ranks, this_rank)
2  bbs ← Communicate the local BB to all ranks using MPI_Allgather
3  rays_to_communicate ← Initialize a map of vectors of Ray objects
4  foreach track in local_2d_track_range do
5      Retrieve the 2D track from OpenMOC
6      if Domain is 2D then
7          Create a Ray object for forward and backward versions of the track
8          foreach bb in bbs do
9              if Ray start is within bb then
10                 rays_to_communicate ← Store the Ray with the rank for the bb
11             end
12         end
13     else
14         foreach 3D track associated with this 2D track do
15             Create a Ray object for forward and backward versions of the track
16             foreach bb in bbs do
17                 if Ray start is within bb then
18                     rays_to_communicate ← Store the Ray with the rank for the bb
19                 end
20             end
21         end
22     end
23 end
24 Continue to Algorithm 16
```

the bounding box of every MPI rank is tested to see if the `Ray` starting position lies within that MPI rank's bounding box. If it does, the `Ray` is then added to a map of `rays_to_communicate`. The communication of the rays is in a later step. Note that this is not unique. The `Ray` object's starting position may lie within multiple bounding boxes and will, therefore, be stored for communication to multiple ranks. Multiple copies of the `Ray` objects are avoided by having the `rays_to_communicate` data structure hold pointers.

Algorithm 16: Determine neighbor ranks

1 **foreach** *remote BB* **do**
2 | **if** *remote BB intersects local BB* **then**
3 | | Store remote BB as a neighbor
4 | **end**
5 **end**
6 Continue to Algorithm 17

Before continuing to the next phase of processing, each MPI rank needs to have an understanding of which other MPI ranks are "neighbors." Neighboring MPI ranks are found using Algorithm 16. The bounding boxes, which were exchanged at the beginning of Algorithm 15, are tested for intersection against the local bounding box. In this way, the set of neighboring MPI ranks are found. This "neighborhood" is utilized for further communication.

At this point, each rank holds a set of `Ray` objects which need to be communicated. In Algorithm 17, the asynchronous sparse data exchange method, `push_parallel_vector_data()` defined in Listing 6.1, is called to send `Ray` objects to remote ranks. Simultaneously, `Ray` objects are received. As they are received, they are further processed, overlapping communication and computation for efficiency.

For each incoming `Ray` in Algorithm 17, a fine-grained search is performed to attempt to locate an element which contains the starting point of the `Ray`. This search is done using `findElementContainingPoint()`. That routine utilizes a `libMesh` capability to do a quad or octree search [53], in 2D or 3D respectively, for an element that exists on the local rank which contains the given position in space. The return value of the function is either a `NULL` pointer (meaning no element on the local rank contains the starting position) or an `Elem` pointer.

In the case of a `NULL` pointer, the `Ray` is deleted. If the local rank owns the element, then the `Ray` is added to a list of `possibly_claimed_rays` which is then further operated on in Algorithm 18. However, as mentioned in §5 The element returned from this search may not belong to this rank. There are two possible cases: the starting point of the `Ray` is not near the current rank's elements and can be safely discarded, or the starting point lies just on the border between an element this rank owns and one a

Algorithm 17: Communicate Ray objects and find Ray objects which begin within an element on this rank.

```
1  possibly_claimed_rays ← Initialize a list of Ray objects this rank will might claim
2  begin
3      Call push_parallel_vector_data() as declared in Listing 6.1
4      // The following communication and processing happens simultaneously
5      Asynchronously send the data in rays_to_communicate to remote ranks
6      // Check each incoming Ray to see if the starting point
7      // actually lies within a mesh element owned by this rank
8      foreach Incoming Ray do
9          elem ← findElementContainingPoint(Ray.start)
10         if elem not owned by this rank then
11             // Check neighbor elements to see if this point might lie on
12             // the side of an element which is owned by the local rank
13             foreach neighbor of elem do
14                 if neighbor is owned by this rank and neighbor contains Ray.start then
15                     elem ← neighbor
16                 end
17             end
18         end
19         if elem owned by this rank then
20             Add Ray to possibly_claimed_rays Ray.starting_elem = elem
21         end
22     end
23 end
24 Continue to Algorithm 18
```

neighbor rank owns. In that case, the Ray must be saved locally for more processing using Algorithm 18.

Algorithm 18: Uniquely claim Ray objects by ranks.

```
1  ids_attempting_to_claim ← Initialize map of rank to vector of Ray IDs
2  // Build a data structure of claims to be communicated
3  foreach ray in possibly_claimed_rays from Algorithm 17 do
4  |  foreach neighbor do
5  |  |  if ray.id is even and neighbor's rank > local rank then
6  |  |  |  ids_attempting_to_claim[neighbor rank] ← ray.id
7  |  |  else if ray.id is odd and neighbor's rank < local rank then
8  |  |  |  ids_attempting_to_claim[neighbor rank] ← ray.id
9  |  |  end
10 |  end
11 end
12 ids_remotely_claimed ← initialize a set of Ray IDs
13 begin
14 |  Call push_parallel_vector_data() as declared in Listing 6.1
15 |  // The following communication and processing happen simultaneously
16 |  Asynchronously communicate the data in ids_attempting_to_claim to
   |   remote ranks
17 |  foreach id in every received vector do
18 |  |  ids_remotely_claimed ← id
19 |  end
20 end
21 foreach id in ids_attempting_to_claim do
22 |  if id not in ids_remotely_claimed then
23 |  |  Claim the Ray and store it
24 |  end
25 end
26 Clear memory for all non-claimed Ray objects
```

The final step for track claiming involves parallel communication between neighbors to have one rank uniquely claim each Ray. This is achieved with a single, one-sided, asynchronous sparse communication step. The idea is that each rank assumes that it has claimed the Ray objects it is holding until told otherwise by the communication step.

The beginning of Algorithm 18 prepares two lists of Ray objects: one to be communicated to neighbors whose rank is larger than the local rank and one to be sent to ranks lower than the local rank. The choice to send a message up or down is based on the Ray ID being even or odd. This splitting is done for balance.

Next, once the lists have been prepared, the push_parallel_vector_data() interface described in §6.1.3 is invoked to asynchronously send the lists to the neighbors. During this asynchronous process, ranks are sending, receiving, and processing simul-

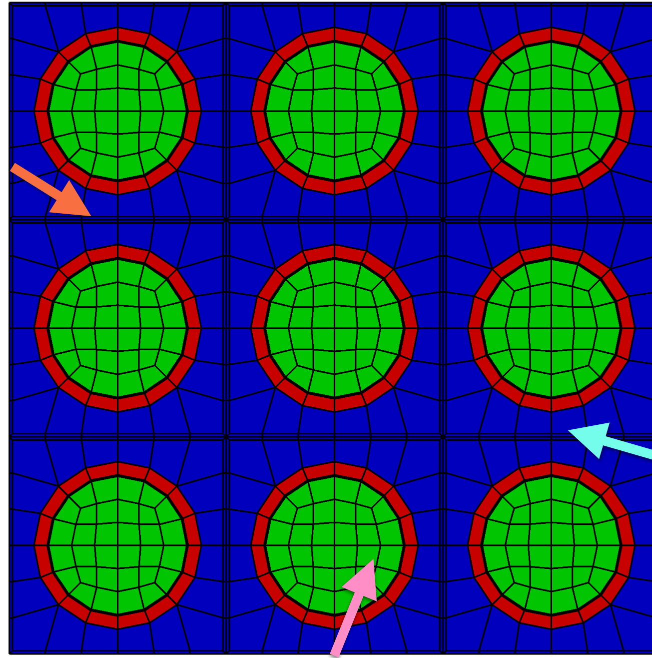


Figure 8.3. The track starting positions for three tracks: orange, pink and blue

taneously. The incoming messages specify that other ranks are claiming those `Ray` IDs, and hence, the local rank can safely discard it.

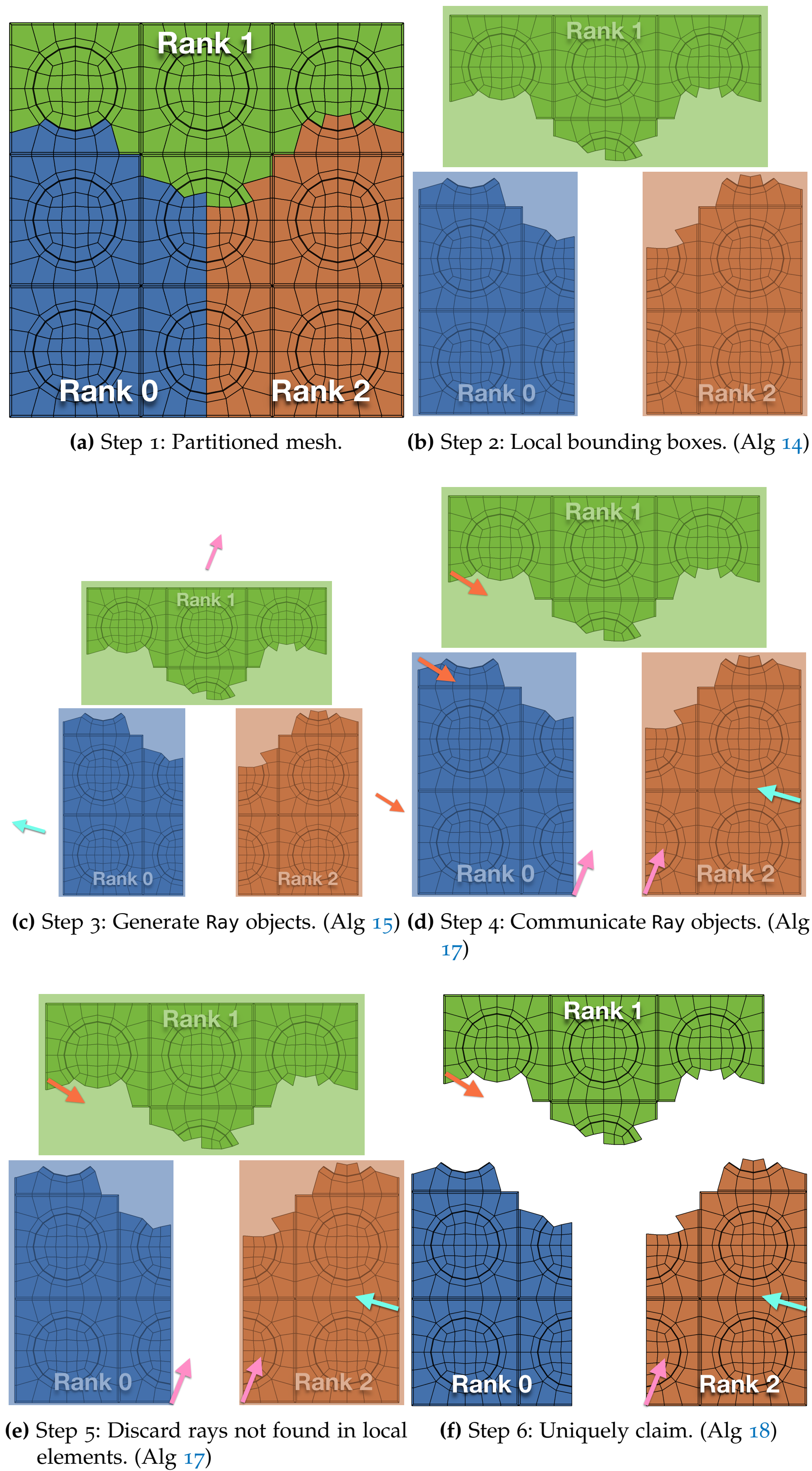
After the asynchronous communication and processing completes, any `Ray` object left on the local rank is then uniquely owned by that rank. During each transport sweep, the owning ranks are uniquely responsible for starting the `Ray` objects which they own.

8.1.1 *Claiming Example*

A concrete example is presented for clarification of the track claiming algorithm. As seen in Figure 8.3, there will be three tracks: orange, pink, and blue. The tracks are represented with their starting positions and directions. In the non-partitioned mesh shown in Figure 8.3, locating the starting position within the mesh is trivial.

The first step in the process is to partition the mesh, as shown in Figure 8.4a. In this case, the mesh was split into three pieces and distributed to three MPI ranks. The next step, as seen in Figure 8.4b is to generate the local bounding boxes and exchange them with all other ranks (Algorithm 14). Also, during this step, the `OpenMOC` track generation routines are initialized.

In step 3 each MPI rank is responsible for generating a range of tracks. In Figure 8.4c, each MPI rank generates one of the three tracks for this demonstration. The tracks are represented as being just outside the local mesh partition since they most likely won't



(a) Step 1: Partitioned mesh.

(b) Step 2: Local bounding boxes. (Alg 14)

(c) Step 3: Generate Ray objects. (Alg 15)

(d) Step 4: Communicate Ray objects. (Alg 17)

(e) Step 5: Discard rays not found in local elements. (Alg 17)

(f) Step 6: Uniquely claim. (Alg 18)

Figure 8.4. Steps of scalable track generation and claiming.

be found within that geometry. In reality, each rank will most likely generate many thousands of tracks.

The next step is to create lists of rays to send MPI ranks which may own their starting position. This is done, as shown in Algorithm 15, by checking the bounding box for each other partition to see if the starting position lies within that bounding box and creating lists of tracks to send to other ranks.

After the lists to be sent are complete, an asynchronous communication step, detailed in 17, is used to communicate the rays and simultaneously do a fine-grained search on the local geometry for the starting positions of any incoming rays. As can be seen in Figure 8.4d, the orange track is communicated twice: once to rank 1 and once to rank 0. This is due to the starting position for that track falling within both bounding boxes. This "duplication" of that track will be fixed in a later step. Similarly, the pink track is also "duplicated" to rank 0 and rank 2. During this process, any incoming rays with starting positions that are not found within the local elements are discarded, as is shown in Figure 8.4e.

The final step is to do one round of single-sided communication to claim the rays uniquely. As is explained in Algorithm 18, asynchronous messages are sent to "neighboring" MPI ranks (those with bounding boxes that intersect the local bounding box). The messages state an intent to claim a ray. For balance, these messages are either sent to higher or lower MPI ranks depending on the ID of the `Ray`. In this way, the pink ray is uniquely claimed by rank 2 in Figure 8.4f.

After completing all of these steps, the three rays have had their starting positions uniquely located within the domain decomposed geometry. If Figure 8.4f is compared to Figure 8.3, it can be seen that the three tracks ended up in the correct positions. Now, during each transport sweep, these ranks always start these tracks.

8.2 source iteration

Source iteration is used to solve the steady-state k-eigenvalue neutron transport problem, which was introduced in Chapter 2. All of the previous chapters and sections have developed the capability needed for efficient, parallel, source iteration for MOC; this section details how those capabilities are put together within `MOCkingbird`.

The source iteration algorithm within `MOCkingbird` is performed as stated in §2.7, including, initializing the flux, computing the source, updating the eigenvalue, and applying stabilization. Therefore, this section only provides detail into deviations `MOCkingbird` takes from standard MOC solvers. There are three main differences:

1. Parallel Storage: memory for values associated with elements is distributed across MPI ranks using `PETSc` vectors.

2. Object-Oriented: Segment integration, materials, and boundary conditions exist as objects that are plugged into `SMART`.

3. Transport Sweep: The transport sweep is performed by passing `Ray` objects representing the tracks to `SMART`.

These differences are explored in detail in the following sections.

8.2.1 *Storage*

With the mesh domain decomposed into partitions, the storage mechanism for spatially varying data becomes important. There are four main spatially varying values stored within `MOCkingbird`:

1. Scalar Flux (ϕ_g)

2. Total Cross Section ($\Sigma_{t,g}$)

3. Total Source (Q_g)

4. Element Volume (V)

The arrays holding the scalar flux, total cross section, and total source are each ordered such that they indexed by element (FSR) then by group. The volume vector is indexed only by the element.

These arrays use a storage mechanism from `libMesh`: the `NumericVector`. A `NumericVector` in `libMesh` is an interface to a parallel, distributed `PETSc` vector, whose layout matches the mesh partitioning. Therefore, the storage is scalable: as the mesh is decomposed across MPI ranks, so is the storage. This is shown in Figure 8.5.

During a transport sweep, a `RayKernel` can both read and write to these parallel vectors. For each segment, `MOCkingbird` computes the current index into the parallel vectors for the current element. The `RayKernel` can then utilize the index to look up values it needs for updating angular flux and accumulating scalar flux.

8.2.2 *Object Oriented Design*

As described in §7.3.4, the `SMART` implementation provides a pluggable set of interfaces for defining physics, boundary conditions, and material properties to be used during ray-tracing. `MOCkingbird` utilizes these interfaces to plug in C++ objects which compute the requisite values for performing a MOC calculation.

As first described in §4.2, `MOCkingbird` contains the ability to read cross sections from data files. This is accomplished within `RayMaterial` derived objects. In §7.3.4

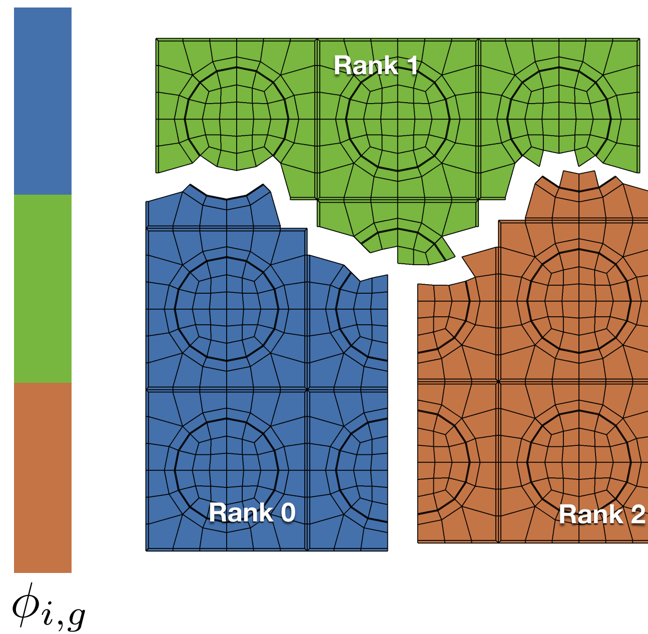


Figure 8.5. Example of a `NumericVector`. The mesh is partitioned into three pieces, therefore the scalar flux vector is also split into three contiguous pieces which are held on the ranks whose elements are associated with those entries in the vector.

a `RayMaterial` was defined as an object which has two methods on it: `reinit()` for computing all cross sections and `reinitSigmaT()` for only recomputing the total cross section. Within `MOCkingbird`, the total cross section is cached into a parallel vector for later retrieval by a `RayKernel` during the transport sweep.

`MOCkingbird` contains one `RayKernel` named `FlatSource`. When `SMART` begins tracing a `Ray`, it will call the `setRay()` method on the `FlatSource` `RayKernel`. Since the `Ray` was created from a track, as explained in §8.1, the `Ray` carries all of the quadrature data needed for spatial and angular integration and a data vector containing the current angular flux (ψ_g). During `setRay()`, `FlatSource` will cache these values internally for easy access during each segment computation.

As `SMART` traces the `Ray` across the domain, the `onSegment()` method will be called on the `FlatSource` `RayKernel` for each element crossing. Recall from §7.3.4 that the `onSegment()` method receives a pointer to the current element and the start and end point of the current segment. This, combined with the quadrature values and the current parallel vector index computed by `MOCkingbird`, allows `FlatSource` to update the angular flux using Equation 2.18 and accumulate ino the scalar flux within the current element using Equation 2.19.

Once the `Ray` reaches the domain boundary, `SMART` executes active `RayBC` objects for that boundary. In §7.3.4 the `RayBC` object was defined as having only one method on it: `apply()`, which receives the current element, intersected side, intersection point

and the Ray object itself. The RayBC performs operations to apply the boundary condition. The VacuumBC object zeros out the angular flux held within the Ray, while a ReflectingBC leaves the angular flux as it is. By leaving the angular flux untouched, the ReflectingBC allows the Ray to start the next transport sweep with the outgoing angular flux.

The RayBC objects within MOCkingbird also perform one other crucial activity: they add the Ray to the RayBank. A RayBank object exists on each MPI rank and is a storage area for Ray objects that have completed tracing during this transport sweep. At the beginning of the next transport sweep, the Ray objects within the RayBank are pulled out, given the direction and information of the next track in the cycle and passed to SMART to trace across the domain. This is further clarified in the next section.

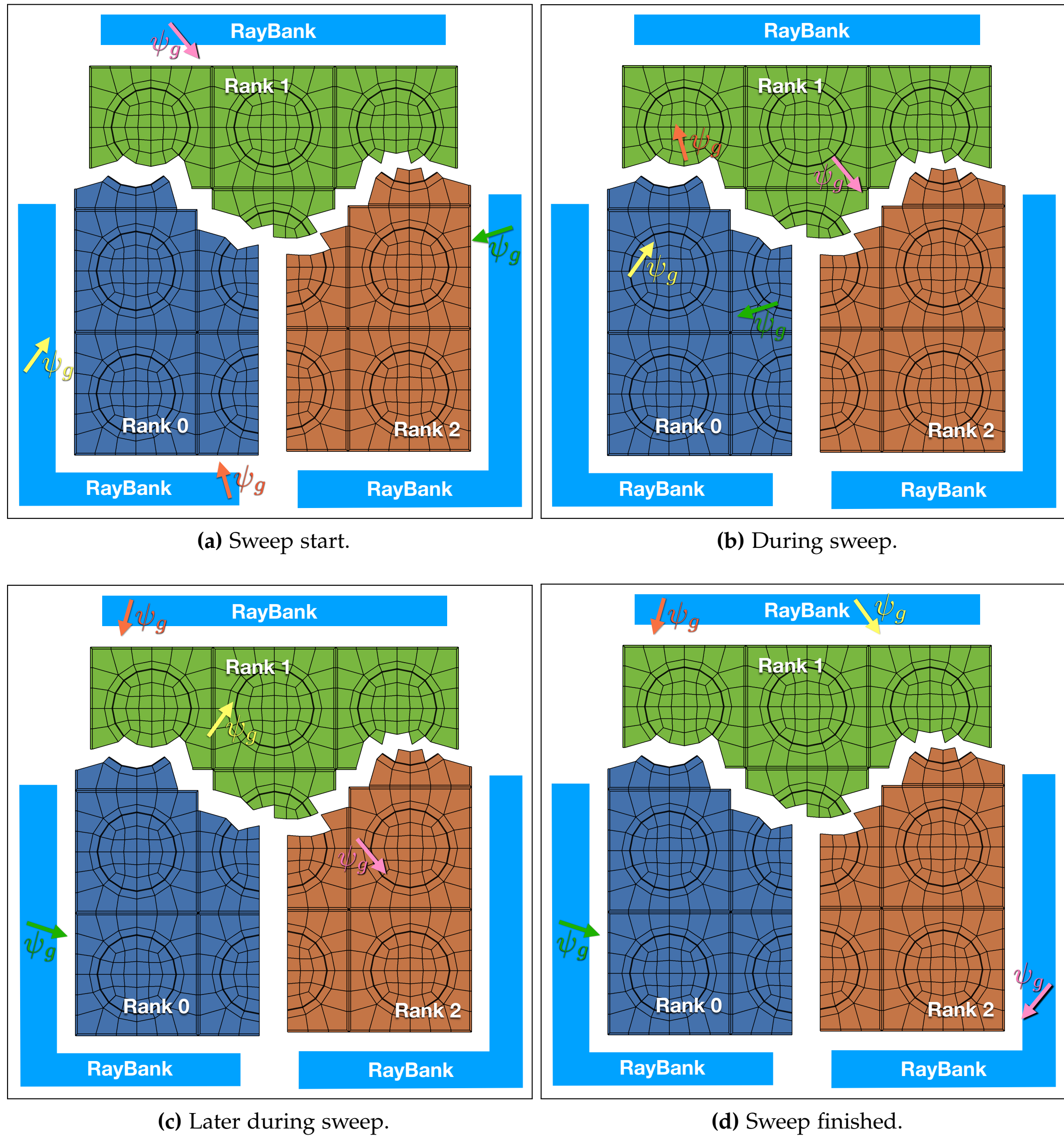
8.2.3 *Transport Sweep*

As mentioned in §7.3.4, MOCkingbird contains an object called TrackListStudy which inherits from the SMART object called RayTracingStudy. TrackListStudy overrides the generate() method to give SMART the Ray objects which represent MOC tracks. During the very first transport sweep, TrackListStudy will retrieve the newly created Ray objects from the track claiming algorithm described in §8.1. It will then add these rays to the work_buffer and let SMART do the rest.

When a Ray reaches a domain boundary SMART calls the RayBC object that has been specified for that boundary. The RayBC (whether it is vacuum or reflecting) adds the Ray to the RayBank on the local MPI rank. During each subsequent transport sweep, the TrackListStudy pulls Ray objects from the local RayBank, uses the track information stored during track claiming to point the Ray in the direction of the next track in the cycle, and then adds the Ray to the work_buffer. Once all Ray objects have been processed from the RayBank the work_buffer will, again, be handed to SMART for tracing.

This process is pictorialized in Figure 8.6. As a demonstration, only four Ray objects are shown, while in a real calculation there are many more. Figure 8.6a shows the starting state for every transport sweep after the first one: all of the Ray objects reside within RayBank objects for each MPI rank. Also, note that each Ray object contains its own group-wise angular flux (ψ_g) which moves with the Ray as it moves through the domain.

When a transport sweep starts, the TrackListStudy pulls the Ray objects from the RayBank and gives them to SMART to start tracing. The Ray objects instantly begin traversing the domain. This is shown in Figure 8.6b where all of the Ray objects are out in the middle of the domain. As the Ray moves through the domain SMART calls



(a) Sweep start.

(b) During sweep.

(c) Later during sweep.

(d) Sweep finished.

Figure 8.6. The evolution of a sweep. Four rays (yellow, pink, green, orange) and their angular fluxes (ψ_g) are traced through the mesh. They start in the `RayBank` of an MPI rank. During the sweep they move through the mesh, modifying ψ_g and accumulating into $\phi_{i,g}$ in each element. When they reach the domain boundary they are put into the `RayBank` on that rank and given the new direction (reflected) they travel during the next sweep.

on the `FlatSource` `RayKernel` to update the angular flux and add to the scalar flux for each element crossed.

Figure 8.6c shows the state of the `Ray` objects after more time has elapsed during the transport sweep. At this point, two `Ray` objects have reached the domain boundary. When that happened a `RayBC` was called (likely `ReflectingBC` for a set of pin-cells like this). The `RayBC` can then modify the angular flux (`ReflectingBC` does not) and place the `Ray` in the local `RayBank`. In Figure 8.6c, both the green and orange rays have completed and are now sitting in a `RayBank`. For illustrative purposes, these rays are shown with their directions modified for the next track in the cycle. In `MOCkingbird`, that step is not performed until the beginning of the next transport sweep, but it is useful to think about it happening this way.

Finally, Figure 8.6d shows the end-state of the transport sweep. Every ray has moved from one domain boundary to another, has had a `RayBC` applied to it, then has been placed in a `RayBank` to await the next transport sweep.

It's important to look at how memory is used during a transport sweep. Notice that `Ray` objects (and hence, angular flux) are not stored anywhere in the interior of the domain. The only apparent storage during the entire process is the angular flux attached to each `Ray` object. That angular flux storage "moves" with the `Ray` such that the total amount of memory used by the algorithm stays constant during the sweep. However, as mentioned in §7.3.3.1, `SMART` utilizes an `ObjectPool` to reduce memory allocation/deallocation as `Ray` objects move into and out of MPI ranks. Because of this, there is still some residual memory usage, even for MPI ranks holding partitions which don't contain any of the domain boundary. This effect will be further explored in §8.3.1.1.

8.3 SCALABILITY

Scalability is critical for a neutron transport tool. There is no end to the amount of computational complexity that simulating a reactor can bring. Geometrically, reactors have many intricate features to be modeled and attempting to do so in three-dimensions creates a significant computational burden. Also, for MOC, using more angles and energy groups is necessary for high-fidelity, predictive simulation. A full-core, three-dimensional neutron transport simulation could be made to utilize the entirety of even the largest supercomputers. Therefore, it is critical that as more computational resources are used, they are used efficiently. This section explores the scalability of the `SMART` algorithm and `MOCkingbird` in both two- and three-dimensional domains on up to 20,000 cores.

For a MOC code like `MOCkingbird`, one of the most important measures of performance is how fast it can perform segment integrations. That is, the time for each

source iteration divided by the number of intersections multiplied by the number of groups and, in 2D, by the number of polar angles,

$$NSI = \frac{T}{\sum_k \sum_i N_g * N_p}, \tag{8.1}$$

where T is the time for one source iteration (in nanoseconds), k are the tracks, i are the segments on each track, N_g is the number of groups and N_p is the number of polar angles in the 2D polar quadrature. NSI here stands for "nano-seconds-per-integration" which is a measure that has been utilized in other MOC papers [2, 3].

A crucial point about NSI is that 2D calculations utilizing multiple polar angles (such as 3 polar angles using TY quadrature [9]) have a floating-point operation (flop) density advantage over 3D calculations. This is because, in 3D, each intersection only integrates one polar angle (the one in the direction of the ray). Therefore, with the same number of groups, a 2D intersection has 3x as many flops to compute as an intersection in 3D. This means, all-other-things-being-equal (which they are not), that 3D NSI should be expected to be higher than 2D.

When considering scalability, if the number of cores is doubled, it would ideally be the case that the time (T) would be cut in half and, therefore, NSI would also fall by half. This provides one way to check the scalability of an algorithm: plot time versus the number of cores utilized and compare to the ideal line. This is one of the primary ways to present scaling results in the following sections.

Though plotting decreasing time as more computational resources are used can provide a visual depiction of how well a code is scaling, the numbers themselves do not mean much. Therefore, another measure is used: "Levelized" NSI, which is defined as:

$$LNSI = N_{procs} * NSI. \tag{8.2}$$

This essentially "re-scales" NSI back to a meaningful number that can be instantly compared with other codes (even if they are serial). If a code is perfectly scaling, the NSI stays constant.

Another measure of scalability is "parallel efficiency." As mentioned earlier, "ideal" speedup would see the time divided directly by the number of cores in use. Therefore, if something took 120s on one core, it should ideally take 30s on 4. However, in reality, there can be some efficiency loss when utilizing more cores by using more MPI ranks: communication costs and load imbalance, in particular, play a major role here. In the

previous example if the code spent 40s computing while using 4 cores that would be an efficiency of 75%, which is computed as,

$$E = \frac{T_{ideal}}{T_{actual}}, \tag{8.3}$$

where $T_{ideal} = T_1/N_{procs}$: the serial compute time divided by the number of cores currently in use and T_{actual} is simply the actual time the algorithm took.

It is often the case that it is not possible (or desirable) to get T_1. This can happen due to the problem simply being too large to run serially or, as is the case in most modern computers, running using one core of a multicore processor is not directly comparable to using many cores of that processor. There are several reasons for that, including automatic frequency scaling and when CPU resources are shared, such as memory bandwidth. In these cases, efficiency can be computed by comparing to the time utilized by the run using the least number of cores (the "reference" (R) case),

$$E_N = \frac{LNSI_R}{LNSI_N}, \tag{8.4}$$

where R is the reference case, $LNSI_R$ is the levelized-nanoseconds-per-integration as defined in Equation (8.2) and similarly for, N: the current number of cores used. As an example, if the code were to achieve 25 levelized nanoseconds-per-integration when using 40 cores and 50 levelized nanoseconds-per-integration when using 400 cores, the parallel efficiency would be 50%.

8.3.1 *Weak Scalability*

A "weak scaling" study can be used to assess the efficacy of the `SMART` algorithm. Weak scaling refers to a study where the amount of computation per core is held constant. As the number of cores grows, so does the problem size (in equal proportion). Weak scaling is very relevant to the engineering community. It represents the idea of running a coarse (simplified) simulation on a laptop/desktop/workstation until it is working correctly, then running the same simulation with 1000x more fidelity using 1000x more cores on a cluster. Ideally, the two simulations should run in the same amount of time.

It should be noted that weak scaling for `MOCkingbird` and `SMART` is much more challenging than for traditional MOC codes using MRT and SDD. This is because, in `MOCkingbird`, true long characteristic rays are traced completely across the domain. As the domain grows larger in weak scaling, the rays have to move farther and pass through many more partitions. In addition, since rays only start on the domain boundary, work has to propagate inward before interior partitions receive any work. This "lag" could, theoretically, cause scaling issues, and this effect worsens as the domain size and num-

ber of MPI ranks grows. Therefore, if `MOCkingbird` maintains high parallel efficiency while weak scaling, that shows the algorithms are working well.

This is not the case with MRT and SDD. With those methods, the tracks are only traced over one partition within a transport sweep. Therefore, for a perfectly repeating lattice, precisely the same work is done by every core (by definition). This means that MRT and SDD should be perfectly weak scalable (as is shown in [3]). However, that does not mean that the solution time for MRT with SDD is perfectly weak scalable. As the domain size grows, the percentage of the domain traced within a partition shrinks with MRT and SDD. This causes a degradation in solver performance, leading to more iterations being required [3]. That does not occur with `MOCkingbird`.

An alternative scalability measure called "strong scaling" also exists. In strong scaling, the problem sized is fixed, and more compute capability is utilized to try to solve it faster. This type of scalability is explored in §8.3.2.

Time is measured by the number of nanoseconds needed for each MOC integration. This is becoming a standard method of measuring the speed of MOC codes [3, 8, 101] and is convenient to measure. It should be noted that the times reported include everything necessary for a MOC iteration: the transport sweep itself (integrating across all of the tracks through the domain) and any pre/post calculations such as recomputation of source terms and convergence checking. In practice, this "non-sweep" time is negligible.

8.3.1.1 *2D Weak Scalability*

To begin, the scalability of the algorithm within a two-dimensional domain is explored. The chosen simulation is based on the BEAVRS [87] benchmark. The cross sections are a set of 70 group cross sections previously generated by [3]. In addition, TY quadrature [9], utilizing three polar angles, is used for polar angle integration. This gives 210 individual angular fluxes to integrate on each segment. Further, the angular quadrature has a fine spacing of 128 azimuthal angles with a 0.001 `cm` spacing. Each execution is run 3 times (doing only one power iteration each time) with the best time taken.

A uniform lattice of 3.1 enriched fuel pin-cells is generated for each number of partitions to be tested. The basic unit mesh for each partition can be seen in Figure 8.7. Each pin-cell contains 1280 quadrilateral elements to describe the fuel, gap, clad, and moderator, this brings the total element count per-partition to 11520.

Mesh partitioning is critical to achieving good scalability. In this case, with a perfect lattice, "optimal" partitioning can be achieved. This "optimal" partitioning is the same as described for SDD in §2.9 A new object was added to `MOOSE` called the `HierarchicalGridPartitioner` that takes into account the layout of the MPI processes across the nodes of the cluster. The idea is to attempt to keep partitions grouped to-

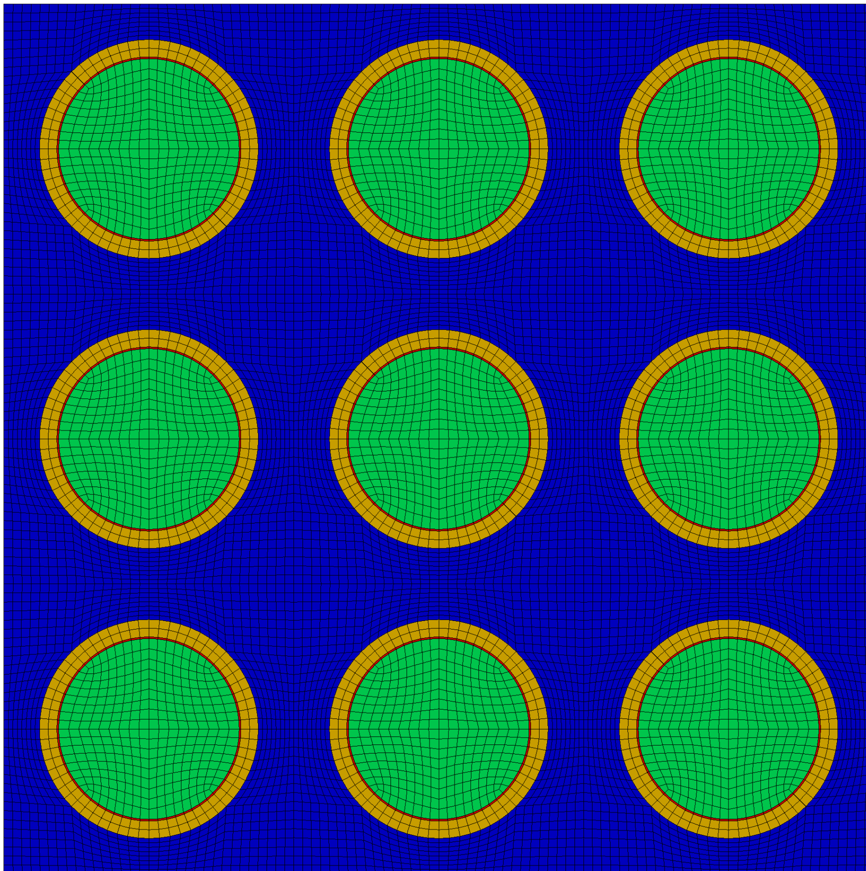


Figure 8.7. Unit cell for each partition within the 2D weak scalability study. 11520 quadrilateral elements total.

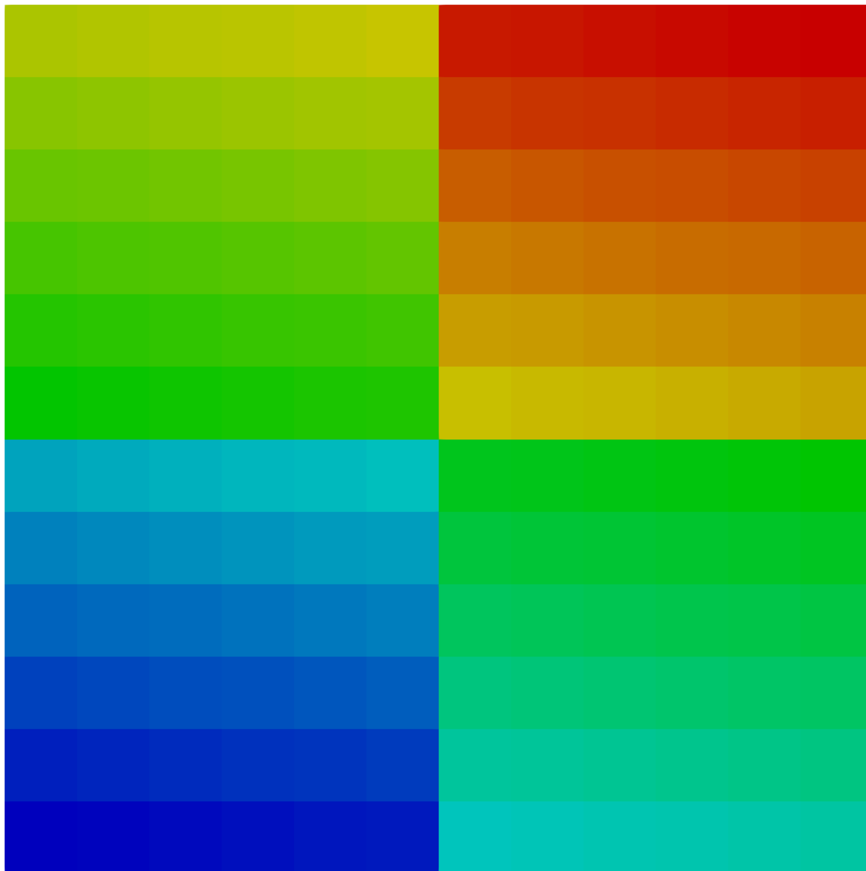


Figure 8.8. Mesh partitioning (each block represents the partition for a different MPI process) for 144 MPI processes spread over 4 computational nodes (36 processes per node).

gether on one compute node that are communicating with each other, thus reducing network communication.

An example partitioning is shown in Figure 8.8. In that figure, the mesh has been partitioned for nodes with 36 processes per node. The coloring is showing the rank each part of the domain is assigned to (from 0 to 143). Each of the four large "squares" (one in each quadrant) is a group of 36 partitions (6x6). The 36 partitions within each quadrant of the mesh are assigned to the same computational node. Therefore, only 44 of the 144 MPI processes do network communication. All of the processes owning partitions "interior" to each node's mesh can communicate using much faster on-node pathways within MPI [103].

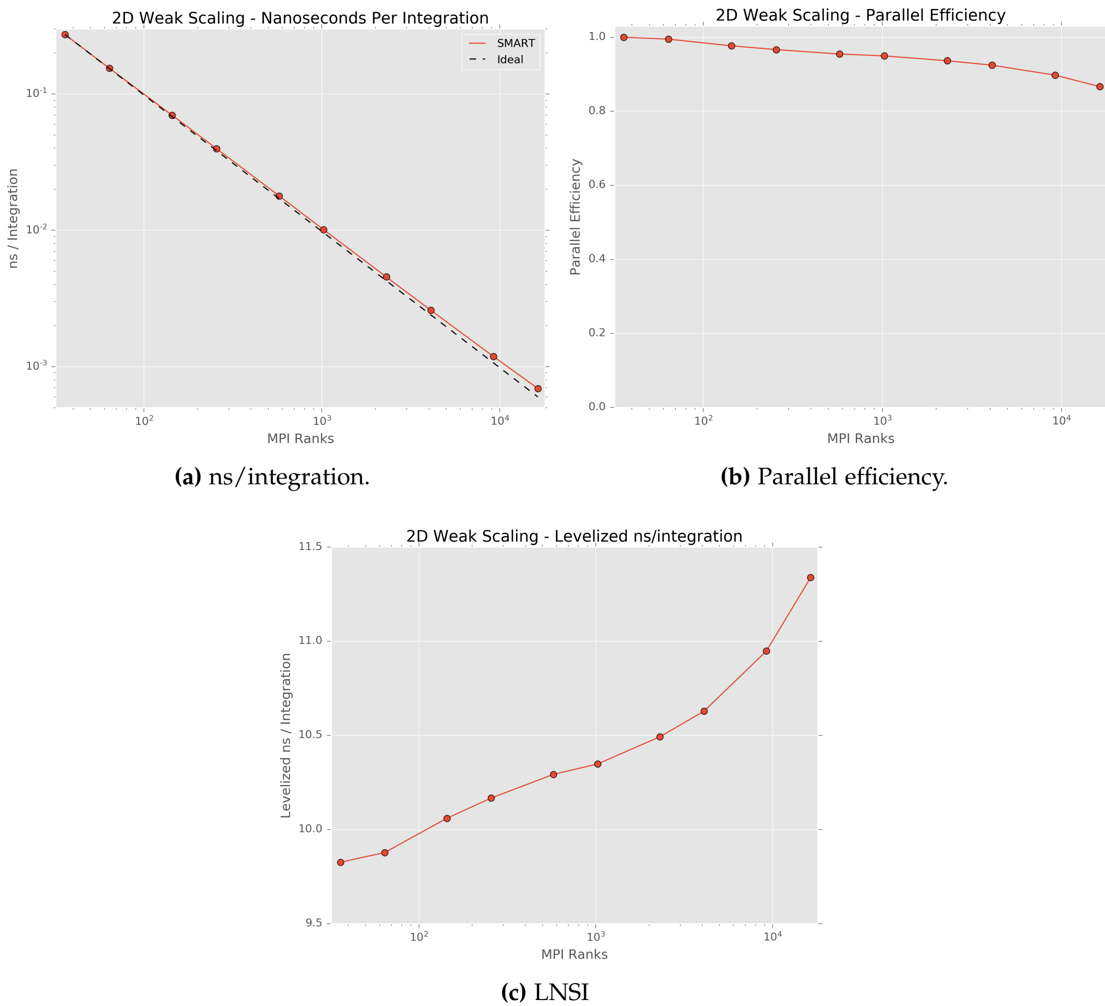
While this type of partitioning can be "optimal," it is also challenging to work with. Only particular numbers of MPI processes per node and numbers of nodes work with a given set of pin-cells. In addition, thought must be used when scaling the domain size and scaling up the number of cores. This is especially true for 3D. Therefore, while this is a useful way to test the algorithm in a way that minimizes the impact of partitioning, it is not a practical capability for production runs. These problems also demonstrate some of the constraints inherent in SDD for traditional MOC codes. A more involved treatment of partitioning options is given in §8.3.3.

As mentioned in §3.1.1, nodes in the Lemhi supercomputer contain 40 cores; however, 40 is not a very convenient number for Cartesian grid scaling. Therefore, for this scaling study, two different sequences of cores were used: one based on 32 MPI ranks per compute node and the other based on 36. The results show that that there is no speed difference in leaving 8 or 4 cores unused. The full set of information for each mesh utilized can be found in Table 8.1. These numbers were chosen due to the above discussion; 32 and 36 are natural numbers to work with to create partitions using the `HierarchcialGridPartitioner`. The results of the executions of both the 32- and 36-core series are graphed together as one result.

The timing results can be seen in Figure 8.9. In Figure 8.9a, the red line shows the nanoseconds-per-integration that was achieved with each successive simulation. Note that we should expect this line to go down linearly on a log-log plot, and it does, following very close to the ideal line that is plotted. A more detailed way to look at the efficacy of the algorithm is in Figure 8.9b. The 36 MPI process result is used as a reference, and the efficiency is plotted in Figure 8.9b. Scaling from 36 cores to 16384 an efficiency of 89% is maintained. This shows that `MOCkingbird` can make efficient use of large clusters for solving highly detailed problems. Finally, the levelized nanoseconds per integration (LNSI) are shown in 8.9c. Ideally, an LNSI plot should always be flat. Here we have a very modest increase going from 36 MPI ranks to over 16k. The absolute numbers are near 10 ns/integration, which is comparable to other published MOC calculation speeds [3, 101, 104], even when working with 16384 MPI ranks.

MPI Processes	X-Pincells	Y-Pincells	Total Pincells	Total Elems
64	24	24	576	737280
256	48	48	2304	2949120
1024	96	96	9216	11796480
4096	192	192	36864	47185920
16384	384	384	147456	188743680
36	18	18	324	414720
144	36	36	1296	1658880
576	72	72	5184	6635520
2304	144	144	20736	26542080
9216	288	288	82944	106168320

Table 8.1. The core and mesh sequences used for the 2D weak scaling study.



(a) ns/integration.

(b) Parallel efficiency.

(c) LNSI

Figure 8.9. 2D weak scaling performance.

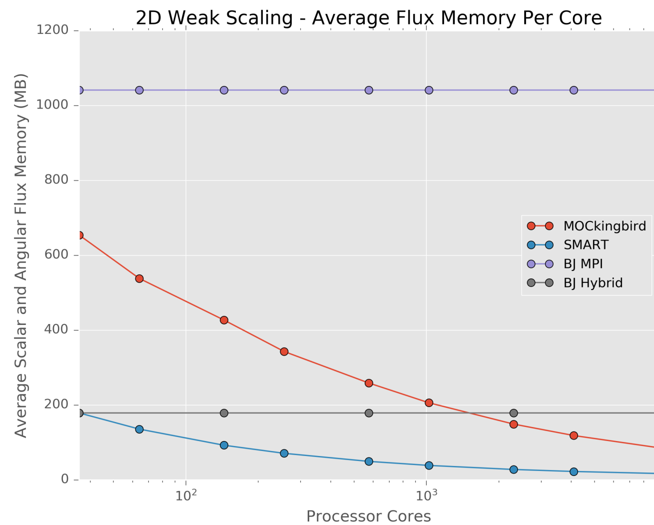


Figure 8.10. Average memory used by each MPI process.

Another interesting aspect of scalability can also be studied with this problem setup: that of memory scalability. An application which scales well in memory should use a constant amount of memory per-process during weak scaling. Figure 8.10 shows the average amount of memory `MOCkingbird` utilized for each MPI rank to store scalar and angular flux information during the scaling study. A good result here would be simply a flat line since each MPI rank has the same amount of information. However, memory actually decreases as the number of MPI ranks increases. This is due to the way the `SMART` algorithm opportunistically reuses `Ray` objects from the `ObjectPool` as they pass through the partition. Along with showing `MOCkingbird` memory performance, Figure 8.10 also shows calculated values for three other cases:

- `SMART`: The theoretical lower limit for memory usage when using SMART.
- BJ MPI: A "block-Jacobi-like" algorithm (as mentioned in §2.10) using only MPI for parallelism.
- BJ Hybrid: The same block-Jacobi-like algorithm, but using hybrid parallelism: OpenMP for shared memory and MPI for distributed.

Each of these other three lines is calculated based on the ray-tracing diagnostic output from `MOCkingbird`. In this case, due to the perfectly square partitioning, the BJ algorithms represent the same memory storage that would be used for Modular Ray Tracing (MRT) with Spatial Domain Decomposition (SDD) as described in §2.10. Figure 8.10 shows that `SMART` theoretically uses the least amount of memory. However, the implementation in `MOCkingbird` uses more, due to buffering and using the `ObjectPool` for efficiency. Even so, `MOCkingbird`, using only MPI, eventually uses less memory than even the BJ Hybrid algorithm, with the crossover point falling near 1000 MPI ranks.

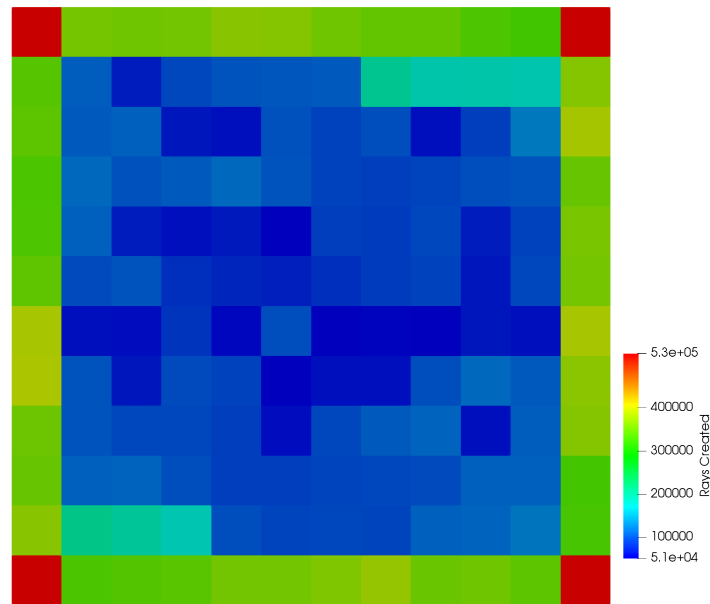


Figure 8.11. Number of Ray objects held in the ObjectPool by each MPI process overlaid on the domain.

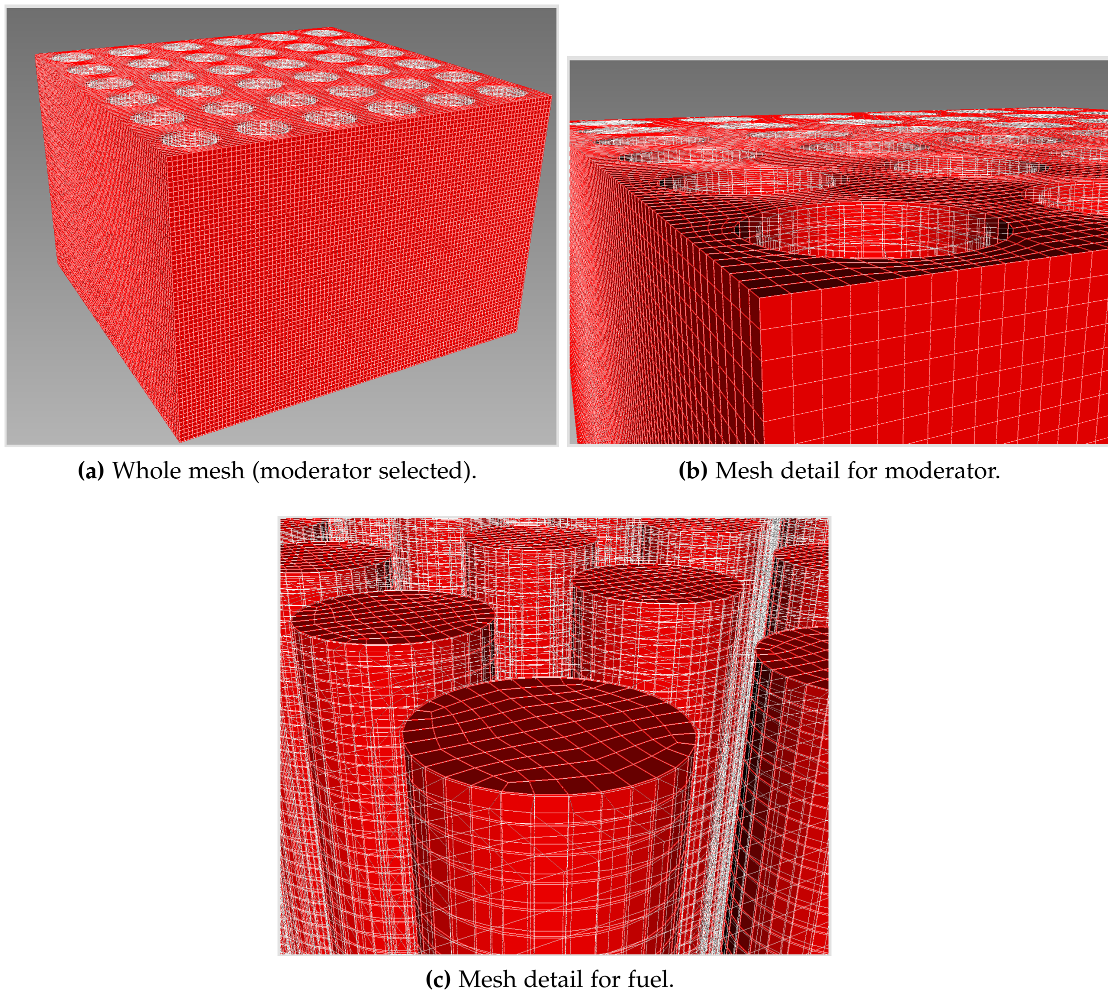
MPI Processes	X-Pincells	Y-Pincells	Z-Layers	Total Elems
36	6	6	64	811008
288	12	12	128	6488064
2304	24	24	256	51904512

Table 8.2. The core and mesh sequences used for the 3D weak scaling study.

Figure 8.11 allows us to visually understand the use of the ObjectPool by each MPI process. The figure shows the number of Ray objects created by each MPI process overlaid on the physical domain for the 144 MPI process case. The MPI ranks owning the boundary must generate all of the Ray objects they start. However, the interior processes only need to generate as many Rays as they ever have "in-flight" at one time. This leads to a dramatic decrease in the average amount of memory consumed per process. Ultimately, this memory scalability shows that SMART can be utilized to solve extremely large, detailed problems.

8.3.1.2 *3D Weak Scalability*

Weak scaling should also be checked with a three-dimensional (3D) problem as well. Three-dimensional weak scaling can be difficult to perform, since each successive grid is 8 times larger than the last, therefore requiring 8 times the number of MPI processes. Because of these limitations, only a small series of tests was completed in 3D, but the overall trend is very similar to 2D. More 3D testing is in Chapter 9



(a) Whole mesh (moderator selected).

(b) Mesh detail for moderator.

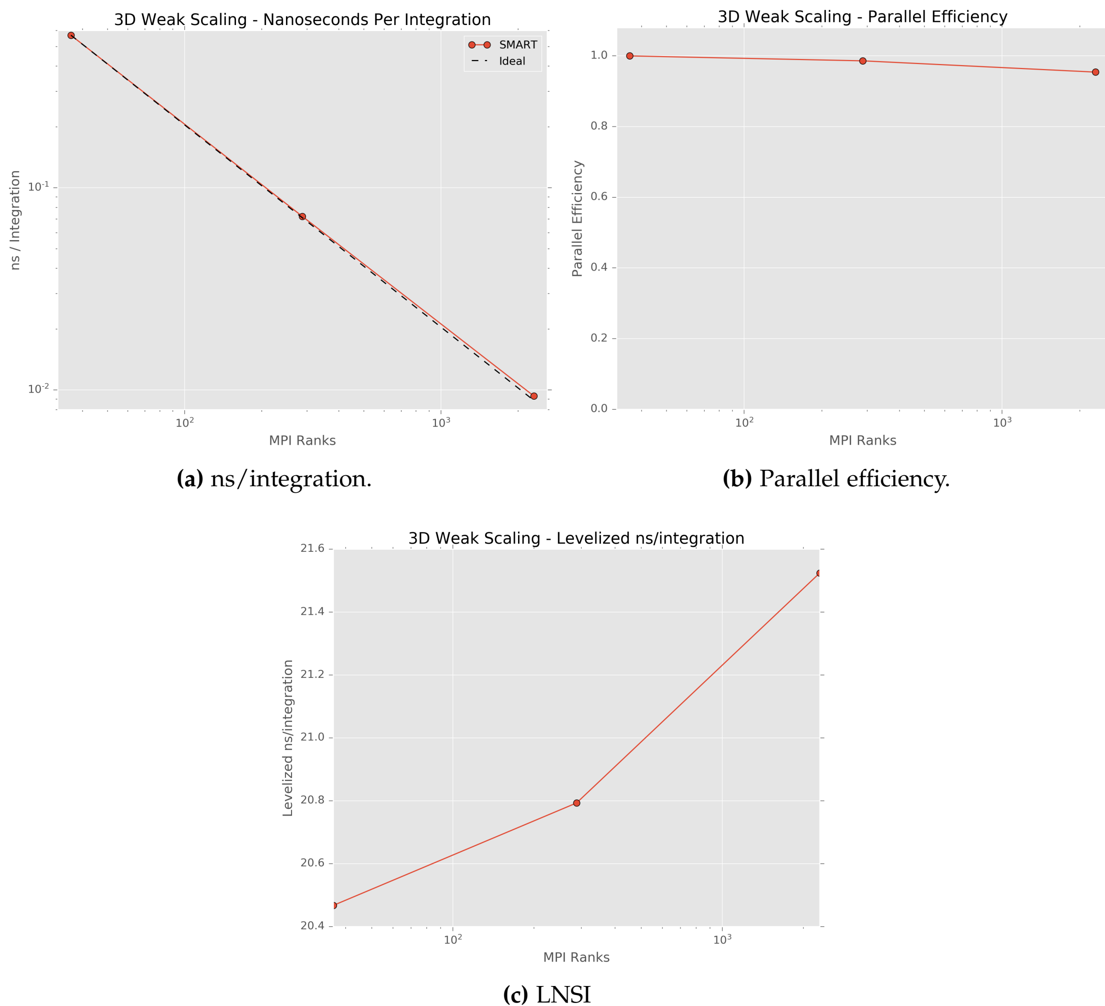
(c) Mesh detail for fuel.

Figure 8.12. The mesh for the 32 MPI process case for 3D weak scaling.

Table 8.2 shows the series of problems carried out to test 3D weak scaling. The number of elements per MPI process is held constant at 22,528. That number is still within the reasonable range of where a finite-element problem would normally be run with MOOSE. Each mesh was created by starting with the two-dimensional set of pin-cells and then using the `MeshGenerator` system within MOOSE to extrude the two-dimensional mesh the specified number of z-layers. The height of the z-layers was set to 1.25984 cm to generate relatively square elements in each pin-cell. Figure 8.12a shows the mesh used for the 36 MPI process case. Each of the successive meshes looks similar, just doubled in each direction.

The results for this scaling study are shown in Figure 8.13. Overall, a similar trend to the 2D results is obtained, showing excellent scalability and speed. Note that speed in three-dimensions is slower than in 2D for multiple reasons:

1. No polar quadrature, reducing the amount of work to do on each segment to one-third of the 2D case.



(a) ns/integration.

(b) Parallel efficiency.

(c) LNSI

Figure 8.13. 3D weak scaling performance.

2. More faces to check for intersections on each element.

3. The intersection tests themselves are also more complicated and costly.

Therefore, a 2x slowdown when going from 2D to 3D is acceptable. The 20 ns/integration that `MOCkingbird` achieves on thousands of cores is similar to the performance obtained for a small, 3D MOC calculation using 36 cores in [15].

8.3.2 *Strong Scalability*

While weak scaling shows the ability of the parallel algorithm to handle increased fidelity using increasing amounts of computational resources, strong scaling shows how the algorithm handles a fixed-size problem. Strong scaling is an essential test because a user of an algorithm/code wants to know if they can solve their problem

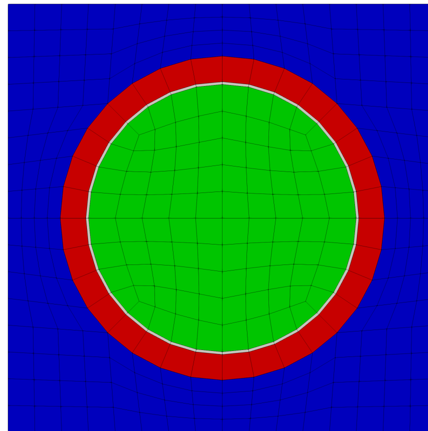


Figure 8.14. Pin-cell used for strong scaling. 352 elements total.

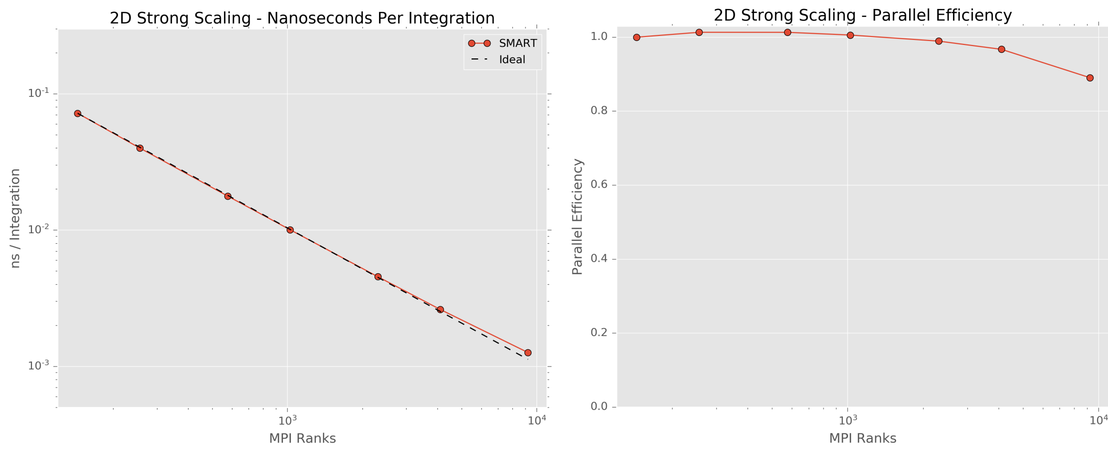
twice as fast if they use twice as many cores. For the 3D, full-core calculations targeted by `MOCkingbird`, it is going to be paramount to be able to use more compute resources to solve them faster.

To test strong scaling, a similar lattice to the one from §8.3.1.1 is used. The target number of cores for the problem to run well on is 1024. Therefore, the pin-cell shown in 8.14 is used in a 192x192 configuration. Note that 192 was chosen so that the `HierarchicalGridPartitioner` could evenly split the resulting lattice perfectly. In §8.3.3 more exotic combinations will be tried for strong-scaling. This pin-cell contains 352 elements giving the final mesh 12,976,128 total elements. This test, then, is an excellent challenge to see how `SMART` deals with a lack of work.

The results of this scaling study can be seen in 8.15. Excellent strong scaling was achieved, with the final result, at 9216 MPI ranks, achieving 89% parallel efficiency. With 9216 MPI ranks, each one is only working with 1408 elements. This shows excellent "algorithmic elasticity": the algorithm doesn't seem to break down even when work is spread very thin. This is a useful feature of any parallel code; it allows a user to utilize whatever resources are available to solve the problem.

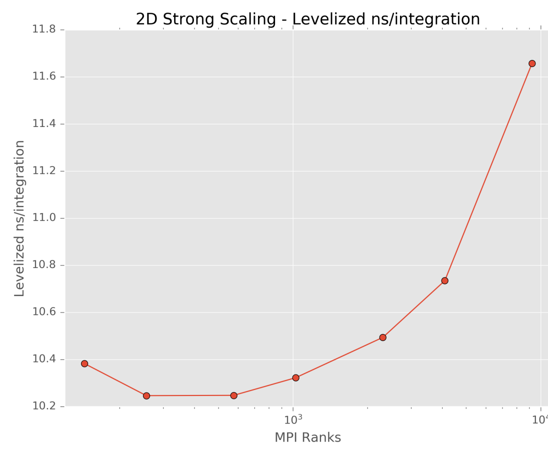
Looking at Figure 8.15b, an interesting phenomenon occurs with 256, 576, and 1024 MPI ranks: they are actually faster than with 36 MPI ranks. While these runs require more communication, it's also possible that the lowered amount of memory needed for each process meant gains were made in memory efficiency (including cache-coherency). The effect is slight though, and turns over at 2304 and continues a more conventional downward trend in parallel efficiency.

Looking at Figure 8.15c the same trend can be seen, `MOCkingbird` can achieve faster integration speeds with 256, 576, and 1024 MPI ranks than with 32. Also, while the end of the plot in Figure 8.15c looks like the code is slowing down by a lot, the vertical axis must be taken into account. In reality, LNSI only grows by a little over 1 ns/integration when scaling the same problem from 32 MPI ranks to 9216.



(a) ns/integration.

(b) Parallel efficiency.



(c) LNSI

Figure 8.15. 2D strong scaling performance.

Once again, these results were achieved using a "perfect" hierarchical partitioning with the `HierarchicalGridPartitioner`. From this point on, we'll investigate more realistic setups using standard finite-element partitioning technology to see how the algorithms fare.

8.3.3 *Mesh Partitioning*

All of the previous tests of scalability utilized a perfect lattice of pin-cells and capitalized on the ability to run using the ideal number of MPI processes to allow perfectly splitting the work. Unfortunately, while that is an excellent way to test the SMART algorithm without many side-effects, it does not translate well into real-world usage. In reality, domains are not always made of perfectly repeating lattices. Even within light-water reactors (LWRs), heterogeneity is provided by spacer grids, guide tubes, burnable poisons, large moderator regions just outside the core and even the pressure vessel. In addition, having to restrict the number of cores used based on the particular numbers of pin-cells and vertical spacing is overly restrictive. Further, unstructured mesh allows for more natural modeling of more exotic types of reactors that may be devoid of a perfectly repeating lattice or might have a complex structure in 3D. Parallel efficiency of real applications of SMART is directly dependent on its ability to balance the workload across a cluster.

This section explores the effect of different partitioning algorithms on the MOC solve within `MOCkingbird`. Novel weighting of the dual-graph (connectivity graph) is utilized to achieve better load balance and improved scalability. The first two schemes tested are `ParMETIS` [44] and a "hierarchical" partitioner which is a multi-level composition of `ParMETIS` partitionings[42].

The `ParMETIS` results are the outcome of directly calling the `ParMETIS` routines and passing the dual-graph of the finite-element mesh. The hierarchical partitioner is a recent novel development within PETSc [42]. The hierarchical partitioner was developed to address a particular design trend within supercomputers: the growth of the number of processor cores within a node of a cluster. In recent years, the number of cores within one node of a supercomputer has significantly increased to numbers such as 40, 48, and 64. Also, supercomputers that are deployed within the next year will contain nodes with 128 or more cores.

As the number of cores within a supercomputer node increases, it becomes crucial to take advantage of the fact that MPI processes running on the same node can communicate with each other much more efficiently than those located on disparate nodes [103]. MPI processes within the same node can utilize "shared memory" messaging schemes developed within MPI; bypassing the overhead of network communication typically required with distributed memory parallelism. This has the potential

to not only speed up those individual communications but also reduce contention for available networking resources, potentially also improving communication speed for messages that must be sent over the network.

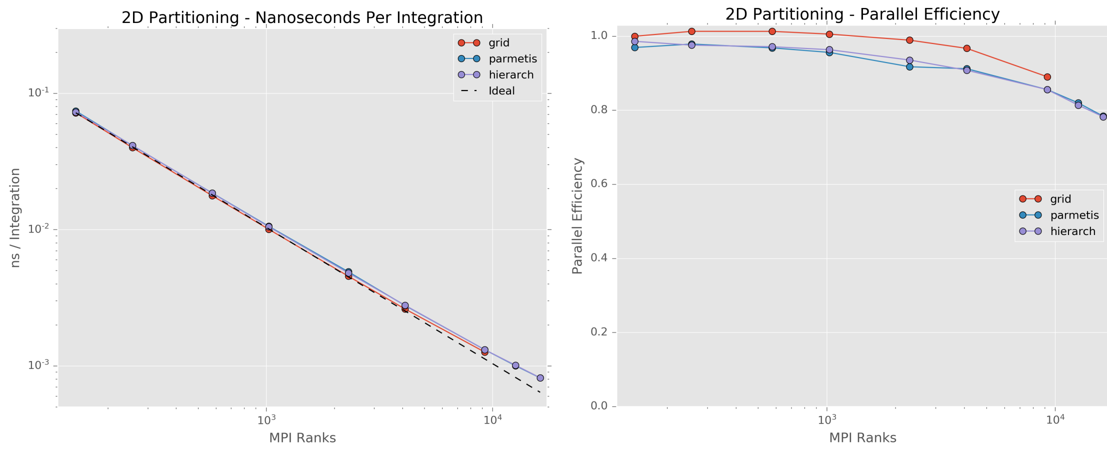
"Hierarchical" partitioning then, is recognizing this two-level communication hierarchy: the network and shared memory. Instead of treating the partitioning problem as merely trying to balance work distribution and reduce communication, hierarchical partitioning strives to cluster partitions which communicate with each other often, onto the same physical node within the cluster to reduce "off-node" network communication. The present work utilizes recent advances within PETSc [42] that allow for a two-level hierarchical partitioning with many available partitioning packages. In this case, ParMETIS is utilized at both levels: first to obtain a partitioning for the compute nodes and secondly to partition the elements assigned to each node for each MPI process on that node.

8.3.3.1 *Perfect Lattice*

To understand the impact of partitioning on the parallel performance of SMART, both ParMETIS and hierarchical partitioning are contrasted with the solution speed on the same perfect lattice problems from §8.3.2. In this case, the solver is pushed to 16200 MPI processes to understand how SMART works with less than perfect partitioning. At that number of processes, there are only 800 elements per process - a real test for the algorithm.

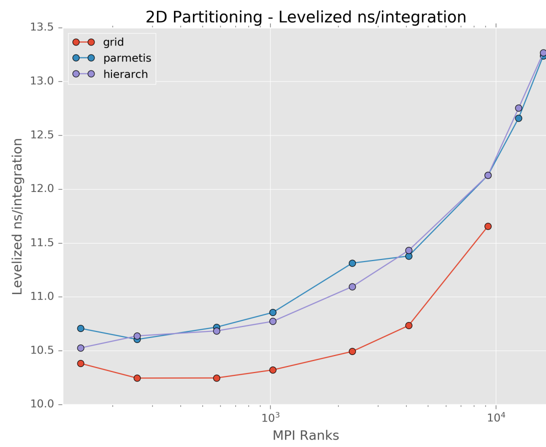
The results of this study can be found in 8.16. For this simple mesh, both ParMETIS and the hierarchical partitioner perform similarly. Both of these schemes compare well to the "perfect" partitioning of the grid partitioner. Further, unlike the grid partitioner, these two schemes do not rely on particular numbers of MPI processes to map the partitions to. This allows the study to push further and try 12600 and 16200 MPI processes. Both ParMETIS and the hierarchical partitioner perform very well when stretched to these limits: keeping nearly 80% efficiency even when only 800 elements are assigned to each MPI process. It appears that SMART can overcome the increased communication costs of these less than perfect partitions.

To look at the increased communication of these partitioning schemes compared to the perfect grid partitioner the size of the off-node communication surfaces can be compared. With tracks completely covering the domain, the amount of communication required is related to the surface area of the partitions. To look at any possible differences between ParMETIS and the hierarchical partitioner Figure 8.17 shows the surface area (in the case of this 2D study the "length") of the partition communication surface. In the case of this simple mesh, ParMETIS performs similarly to the hierarchical partitioner, both of which show an increase in network communication compared to the grid partitioner.



(a) ns/integration.

(b) Parallel efficiency.



(c) LNSI

Figure 8.16. Comparison of 2D strong scaling performance with three different partitioning schemes.

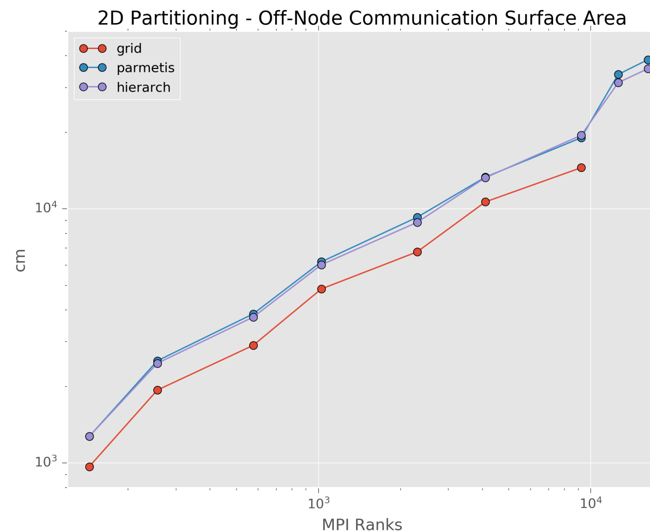


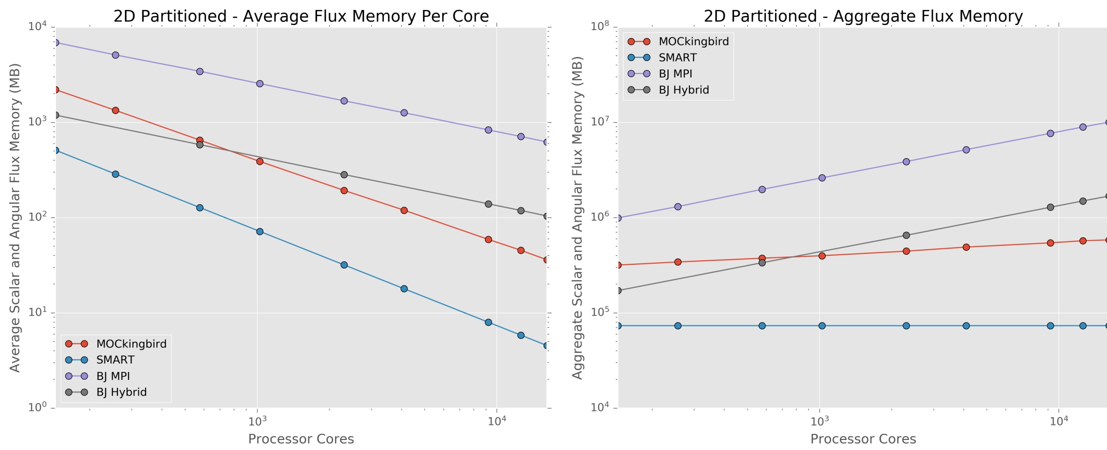
Figure 8.17. Total size (in cm) of the off-node communication surface area for each partitioning scheme.

Another interesting aspect is how much memory the algorithm uses with less than perfect partitioning. Figure 8.18 shows three different views of the memory performance of the same four algorithms studied earlier. Firstly, Figures 8.18a 8.18b show `MOCkingbird` scaling well in memory usage. The average amount of memory per MPI rank decreases at a faster rate than either of the BJ methods. In addition, the crossover point between `MOCkingbird` and the hybrid BJ scheme is now less than 1000 cores. This is due to the increased storage requirements the BJ method would need when the mesh decomposition is not perfect squares/cubes.

In Figure 8.18c, it can be seen that `MOCkingbird` uses the most memory for storing angular flux data. The next largest use of memory within `MOCkingbird` is buffers for receiving messages. This makes sense, the `SMART` algorithm is set up to start receive operations often, but only complete them when the MPI rank is running out of work to do. This means that a large number of messages get queued up in the `ReceiveBuffer`, waiting to be pulled out and added to the `work_buffer`. It is possible that future work could be done to optimize the memory usage further, cutting down on the need for such a large percentage of memory being used for communication buffers.

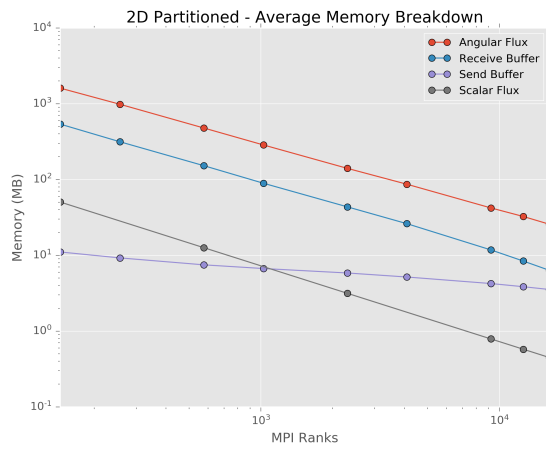
8.3.3.2 *Weighted Partitioning*

The preceding section showed that with a perfect lattice of pin-cells, the `SMART` algorithm could perform well, even when utilizing less-than-perfect partitioning and being pushed to the limits of scalability. However, the real world doesn't present a perfect array of the same repeating pin-cell. Within a reactor mesh, there are finer areas of



(a) Average memory use per core.

(b) Total memory used by all processes.



(c) How memory is used in `MOCkingbird`.

Figure 8.18. Memory scalability with the hierarchical partitioner.

mesh to capture, e.g., the intricacies of burnable poisons alongside coarser areas of mesh for featureless regions such as in the reflector.

The differences in element size drive an imbalance of work attributed to each element. MOC track generation aims to deliver uniform angular and spatial coverage of the physical domain. Therefore larger elements have more integrations to compute. The number of integrations for an element are proportional to the surface area of the element (the perimeter length in 2D). The larger the surface area, the more "work" there is to perform for that element.

This is a subtle point. In both [5] and [3] there are statements made that MOC work is proportional to the **volume** of a partition. Those two studies were using SDD to develop partitions containing a similar geometry arrangement. In the case of repeating geometry and same-sized partitions, their statements are true. However, if the partitions contain differing geometry, then a partition with more FSR surface area within it has more work to do, even if the partitions themselves are the same size.

The surface area of an FSR provides its probably for being intersected by a ray. This idea is very similar to the fundamental ideas used in nuclear cross sections and nuclear shielding. Something with a larger solid angle is more likely to be struck by tracks which are uniformly distributed in space and angle (which might not be entirely true in 3D but is in 2D). Therefore, two FSRs/elements having the same volume, but different surface areas have different probabilities of tracks cross them. The more surface area, the more likely to be crossed. Thus, the amount of work to be done on each element is proportional to its surface area.

The partitioning schemes utilized in §8.3.3.1 were blind to this effect: treating each element as having the same amount of work. Therefore those schemes would try to assign the same number of elements to each partition. However, given a mesh with unequally sized elements, this assignment would cause a load imbalance, leading to a loss in parallel efficiency. This loss in efficiency is due to MPI ranks sitting idle while others are still working. In an ideal partitioning, each MPI rank would have the same amount of work apportioned to it.

While the work for a partition is set by the amount of surface area contained within it, the communication burden for a partition is proportional to the surface area of the partition [3, 5]. A finer-grained statement is that the communication cost between any two elements is proportional to the surface area of the face between them. Therefore, a partitioner should prefer to "cut" surface/edges, which are smaller in order to minimize communication.

Thinking back to the dual-graph introduced in §3.4.1, the relative workload associated with the surface area of an element could be viewed as a "weight" associated with a node of the dual-graph. Similarly, the surface area of each side represents a "weight" associated with each edge in the dual-graph. Therefore, what's needed is a

partitioning algorithm which takes into account these node and edge weights. The partitioner would seek to balance the amount of work done by each MPI process (as expressed by the node weights) while minimizing the total communication cost of the computation (found from the edge weights).

Fortunately, `METIS` and `ParMETIS` both allow the specification of these weights. Working with Dr.Fande Kong at Idaho National Laboratory, the `Partitioner` interface within `MOOSE` was expanded to allow a simple way to specify these weights. The interface is defined by two virtual function callbacks, as shown in Listing 8.1. Within `MOCkingbird`, these functions were overridden to compute element and side weights as the total surface area of the sides of the element and the surface area of the particular side of the element, respectively.

```
// Retrieve the "weight" (work-load) for the given mesh element
virtual long unsigned int computeElementWeight(Elem & elm);

// Retrieve the "weight" (communication burden) for a given side of an element
virtual long unsigned int computeSideWeight(Elem & elem, unsigned int side);
```

Listing 8.1. `MOOSE` interface for specification of partitioning weights.

It is important to note that these functions return integers. This is due to the interface in `METIS` and `ParMETIS` requiring weights be integers. The idea that the weights should be integers is related to the idea that in finite-element problems the work-load on an element is typically specified by the number of degrees of freedom on that element (which is, in turn, due to the number of shape functions falling on that element). However, the weights computed by `MOCkingbird` are floating-point values computed from the surface areas of the elements.

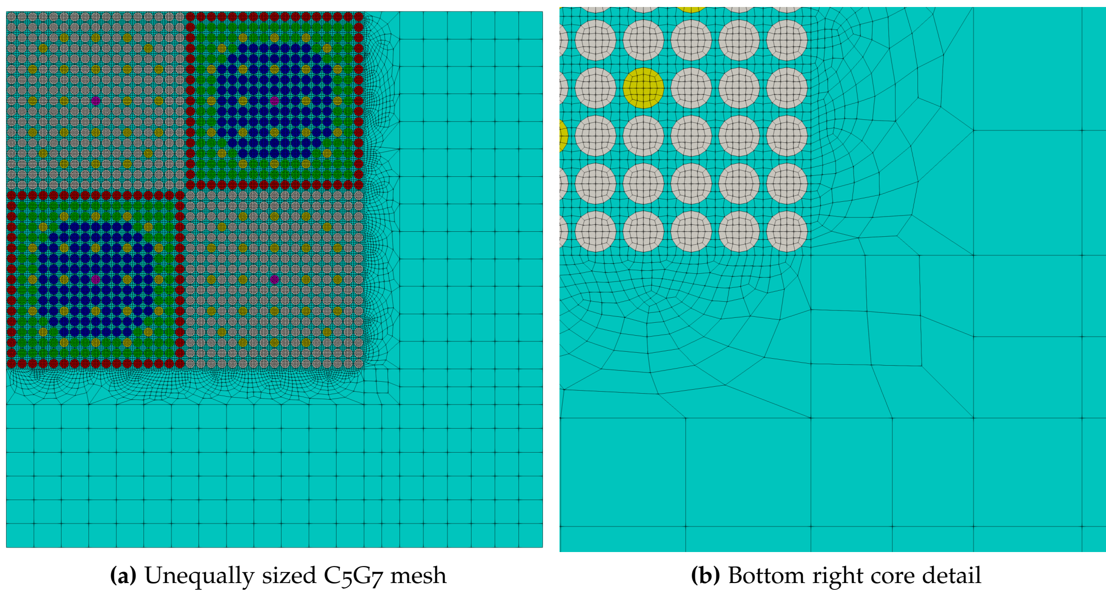
In C++ a floating-point number which is cast to an integer loses all information after the decimal point. As an example `3.4` would become simply `3`. While this is a small loss of information which would seem to be non-critical (the weights don't need to be perfect to work well), it can lead to a lack of quality in the output of the partitioners. In particular, if a weight is less than `1` then the weight is cast to zero.

With a reactor mesh in centimeters, the surface areas of sides of elements are routinely less than one, leading to a significant loss in precision of the weights. To combat this, `MOCkingbird` scales the weights by dividing by the smallest weight within the mesh (making all weights `>=1`) then multiplying by 100. The 100, while arbitrary, provides several decimal points of precision and was found experimentally to be effective across a wide range of reactor problems.

The exact procedure utilized by `MOCkingbird` for computing weights, can be found in Listing 8.2. That listing shows the implementation of the `SurfaceAreaWeighted-`

```
void
SurfaceAreaWeightedPartitioner::_do_partition(MeshBase & mesh, const unsigned int n)
{
  // Loop over all of the elements and find the _smallest_ side area
  _min_side_area = std::numeric_limits<Real>::max();
  _min_elem_surface_area = std::numeric_limits<Real>::max();

  for (auto & elem : mesh.active_local_element_ptr_range())
  {
    Real elem_surface_area = 0;

    for (auto s : elem->side_index_range())
    {
      auto side_area = elem->build_side(s)->volume();

      _min_side_area = std::min(side_area, _min_side_area);

      elem_surface_area += side_area;
    }

    _min_elem_surface_area = std::min(elem_surface_area, _min_elem_surface_area);
  }

  // Find the min over all procs
  _communicator.min(_min_side_area);
  _communicator.min(_min_elem_surface_area);

  // Call the base class
  PetscExternalPartitioner::_do_partition(mesh, n);
}

std::unique_ptr<Partitioner>
SurfaceAreaWeightedPartitioner::clone() const
{
  return libmesh_make_unique<SurfaceAreaWeightedPartitioner>(_pars);
}

dof_id_type
SurfaceAreaWeightedPartitioner::computeElementWeight(Elem & elem)
{
  Real elem_surface_area = 0;

  for (auto s : elem.side_index_range())
    elem_surface_area += elem.build_side(s)->volume();

  return 100*(elem_surface_area / _min_elem_surface_area);
}

dof_id_type
SurfaceAreaWeightedPartitioner::computeSideWeight(Elem & elem, unsigned int side)
{
  auto side_elem = elem.build_side(side);

  return 100*(side_elem->volume() / _min_side_area);
}
```

Listing 8.2. Implementation of `SurfaceAreaWeightedPartitioner` within `MOCkingbird`



(a) Unequally sized C5G7 mesh (b) Bottom right core detail

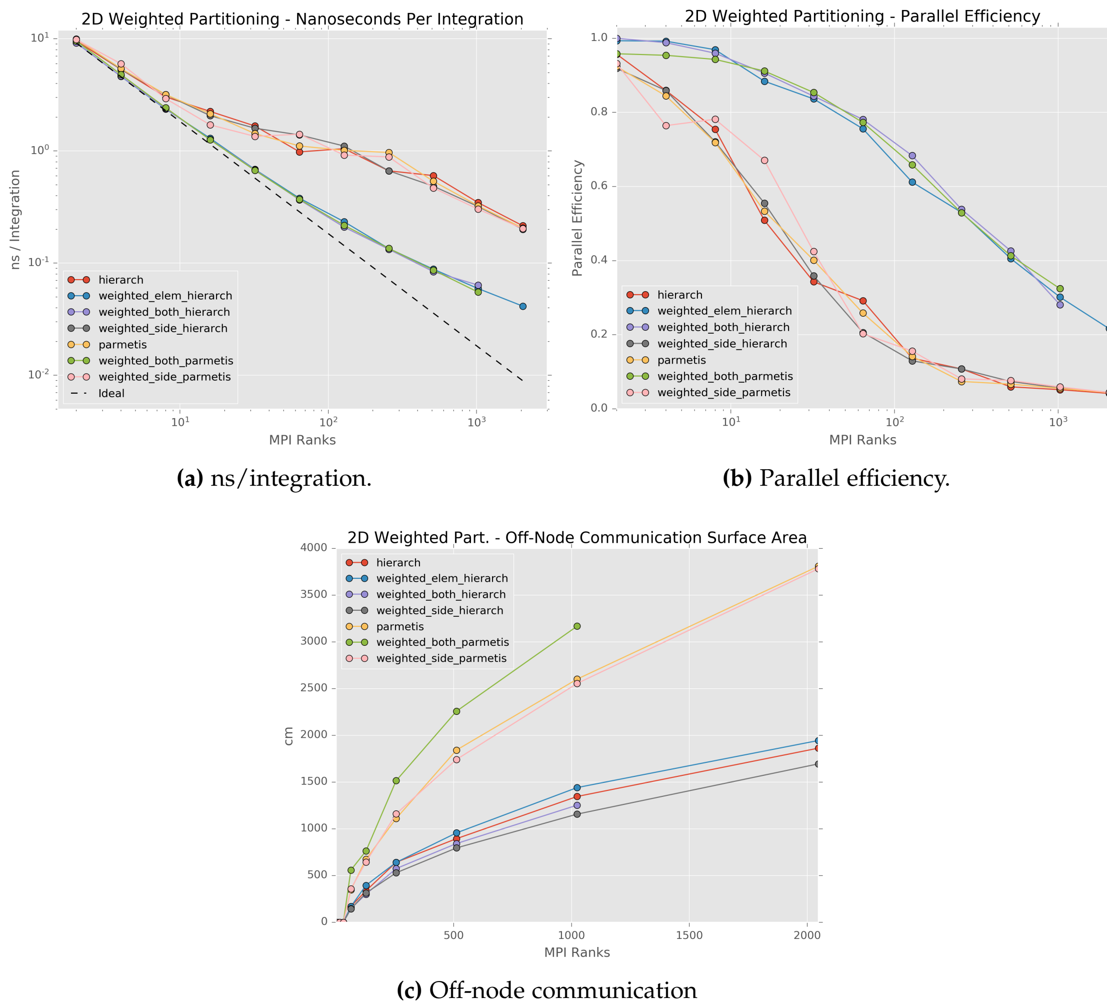
Figure 8.19. C5G7 geometry and mesh detail used for testing weighted partitioning.

`Partitioner`. The `_do_partition()` function is called first and is responsible for finding the global minimum of both the element and side weights. The virtual function overrides, responsible for returning the weights, are readily readable within Listing 8.2.

To test the utility of weighted partitioning, a mesh with very unequally sized elements representing the C5G7 benchmark problem [85] domain (which is further discussed in §9.2) was generated as shown in Figure 8.19. The moderator section of the mesh utilizes large elements, while tiny elements can be found within the core. The mesh contains only 62,258 elements; therefore, scalability is fairly limited. In addition, this problem utilizes cross sections from C5G7, giving only 7 energy groups. However, TY quadrature with three polar angles generates a total of 21 integrations that need to be performed on every segment. That leads to 10x less work per segment than the previous 2D tests. All of these factors combine for a very challenging test of scalability for both the partitioners and `SMART`.

A strong scaling study was completed using this mesh; the results can be seen in Figure 8.20. Both partitioners (`ParMETIS` and hierarchical) were tested in four configurations: non-weighted, element weighted, side weighted, and weighing on "both" elements and sides. In total this gave seven configurations which are represented on each of the three graphs (the ParMETIS element weighting was omitted to help with graph clarity).

The graph in Figure 8.20a instantly displays how important weighted partitioning is to the scalability of `MOCkingbird`. Only the schemes that include element weighting are able to scale appropriately, staying close to the ideal line through several hundred



(a) ns/integration.

(b) Parallel efficiency.

(c) Off-node communication

Figure 8.20. Results of a strong scaling study conducted using the mesh in Figure 8.19. `ParMETIS` and the hierarchical partitioner were tested both with and without element and side weights.

MPI processes and still showing speedup out to 2048 MPI processes. It's important to note that at 2048 processes, there are only 30 elements per MPI processes, and, as pointed out earlier, not much work to do for each intersection.

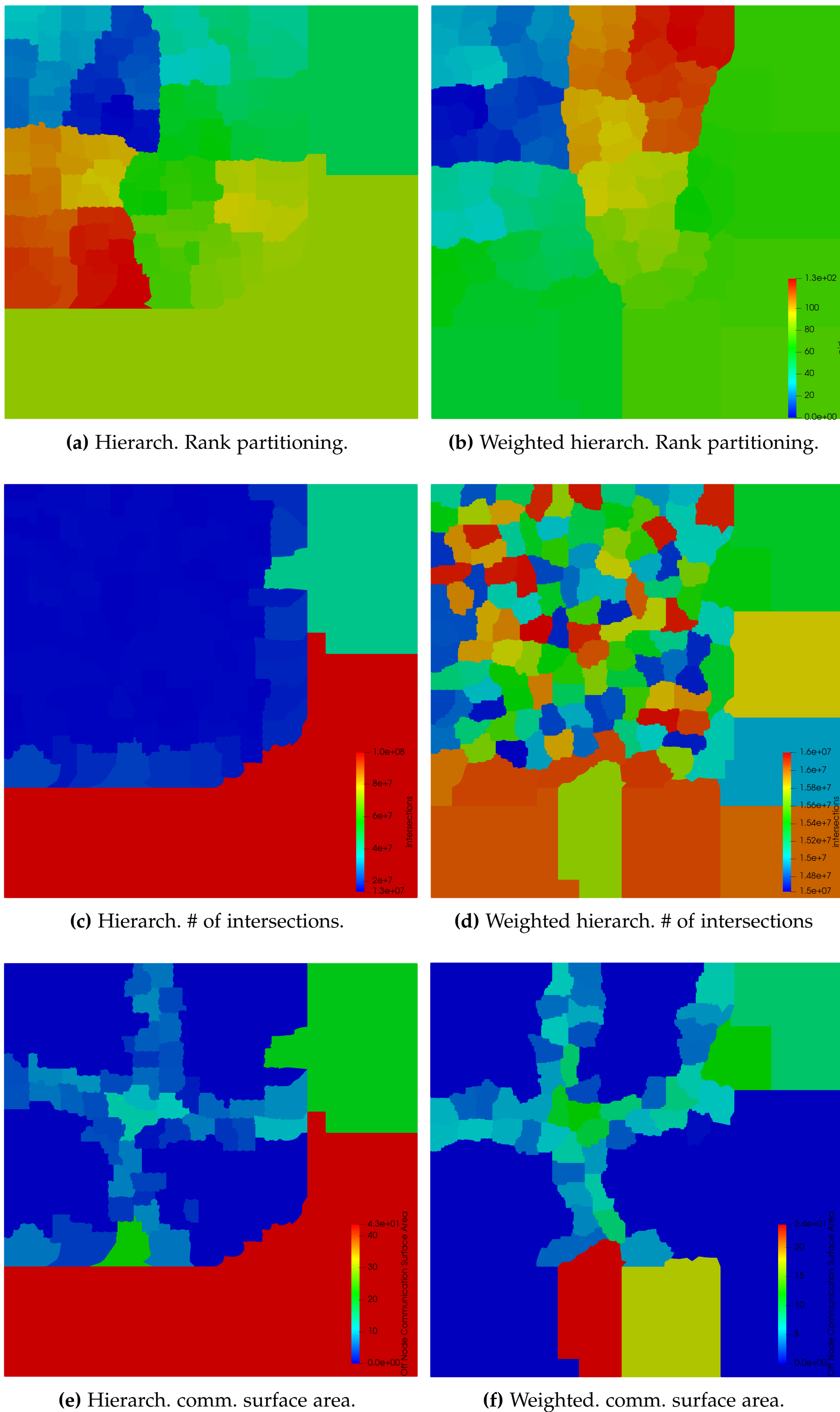
Looking at Figure 8.20b supports the same findings, with element weighting being critical to efficiency. At 2048 MPI processes, element weighted hierarchical partitioning is still able to achieve 28% efficiency with just 30 elements per process. It should be noted that several of the schemes failed to run with 2048 MPI ranks due to some ranks getting assigned zero elements. Therefore, 2048 processes is a difficult test, and the code is still able to gain some execution speed.

In contrast, all of the schemes without element weighting fared poorly. By 196 MPI ranks, they were already at 50% efficiency and going down fast. The workload imbalance cannot be overcome.

Figure 8.20c, displaying the total amount of off-node communication surface area, contains many interesting pieces of information. Firstly, all four of the schemes employing hierarchical partitioning are able to keep off-node communication under control, even with the slim number of elements per process. The hierarchical method employing only side weighting is the best at controlling the network communication surface size. This is to be expected since that scheme is employing multilevel partitioning to group nearby partitions together and simultaneously preferentially seeking to minimize the communication instead of work balance. However, even though communication is at a minimum, this scheme is still unable to perform well due to load imbalance. `SMART` plays a role here: the overlapping of communication and computation means that communication is less of a bottleneck than it would be with a traditional communication algorithm. Therefore, while hierarchical partitioning with side weighting can limit off-node communication effectively, it has little impact on overall run time.

Also in 8.20c it can be seen that `ParMETIS` is not helped much at reducing off-node communication by adding side weighting. Reduction of off-node communication is not the goal of `ParMETIS`. In addition, it's striking how much more off-node communication there is when `ParMETIS` utilizes both element and side weighting. By optimizing for better work-balance, instead of communication, off-node communication is significantly increased. However, in spite of this increased communication, the other graphs show that element weighting and weighting with both elements and sides leads to better run times. This, once again, shows that `SMART` is fairly insensitive to the total amount of communication, yet work-balance is very important.

Finally, Figure 8.21 visually displays the differences between unweighted hierarchical partitioning (left column) and weighted hierarchical partitioning (right column), of the mesh in Figure 8.19. The partitionings themselves are shown in Figures 8.21a 8.21b. Contrasting these two partitionings, the effect of weighting is clear. In Figure



(a) Hierarch. Rank partitioning.

(b) Weighted hierarch. Rank partitioning.

(c) Hierarch. # of intersections.

(d) Weighted hierarch. # of intersections

(e) Hierarch. comm. surface area.

(f) Weighted. comm. surface area.

Figure 8.21. Comparison of hierarch. partitioning without (left) and with (right) weighting.

8.21a the partitions are holding equal numbers of elements while in Figure 8.21b the partitions hold an equal amount of *surface area*.

These two partitionings were used in a `MOCkingbird` simulation, giving the number of intersections in each partition as shown in Figures 8.21c 8.21d. The effect of weighting is, once again clear. The unweighted result in Figure 8.21c has nearly an order of magnitude of difference in the amount of work between partitions. Meanwhile, the weighted result, in Figure 8.21d, only has 6.0% maximum difference in workload. As shown in Figure 8.20, without load balancing, scalability cannot be achieved.

In the final set of images in Figure 8.21, the off-node communication surface area is on display. In this case, the maximum communication surface area was cut in half when weights were applied. In massively parallel use-cases this could make a tremendous difference.

In summary, neither `ParMETIS` nor the hierarchical partitioner can achieve good scalability without weighting. While side weighting can reduce communication, only element weighting (based on the surface area of the element) is effective at providing parallel efficiency. Weighted partitioning is a critical capability for unstructured mesh MOC. Load imbalance was seen as one of the more significant issues in the literature [71], and it has been overcome through the use of surface area-weighted partitioning.

8.3.4 *Comparison To Other Algorithms*

In the previous sections, `SMART` proved to be fast and scalable when performing transport sweeps for `MOCkingbird`. However, alternative communication algorithms exist. This section tests two of them: Bulk Synchronous (BS) and the Hybrid Adaptive Ray-Moment Method or HARM^2[105].

The BS algorithm is the most straightforward way to do communication of angular fluxes between partitions in a MOC solver. Many MOC codes have been developed using it [3, 5, 101] (although [5] used a few advanced MPI capabilities the algorithm was still bulk synchronous). The basic BS algorithm can be found in Algorithm 19. It can be seen as a series of single-hop communications. Those communications can either be blocking (as they often are [106]) or non-blocking [5], but they are still single-hop due to the global synchronization between each successive communication.

While BS can be a viable scheme in CSG-based MOC codes employing SDD and MRT, it will most likely struggle with unstructured mesh MOC. The reason for this is readily apparent in Figure 8.22: jagged partition edges, which occur due to unstructured mesh partitioning algorithms, cause large numbers of iterations to be required to move information completely across the domain. This problem is one of the reasons cited [107] for why MOCFE ultimately chose to use a Krylov solver to solve the MOC system of equations.

Algorithm 19: Basic BS Algorithm

```
1 while Not finished do
2     Trace all tracks on local partition
3     Send all outgoing angular fluxes
4     Receive all incoming angular fluxes
5     Wait for all communication to end globally
6     Check if finished
7 end
```

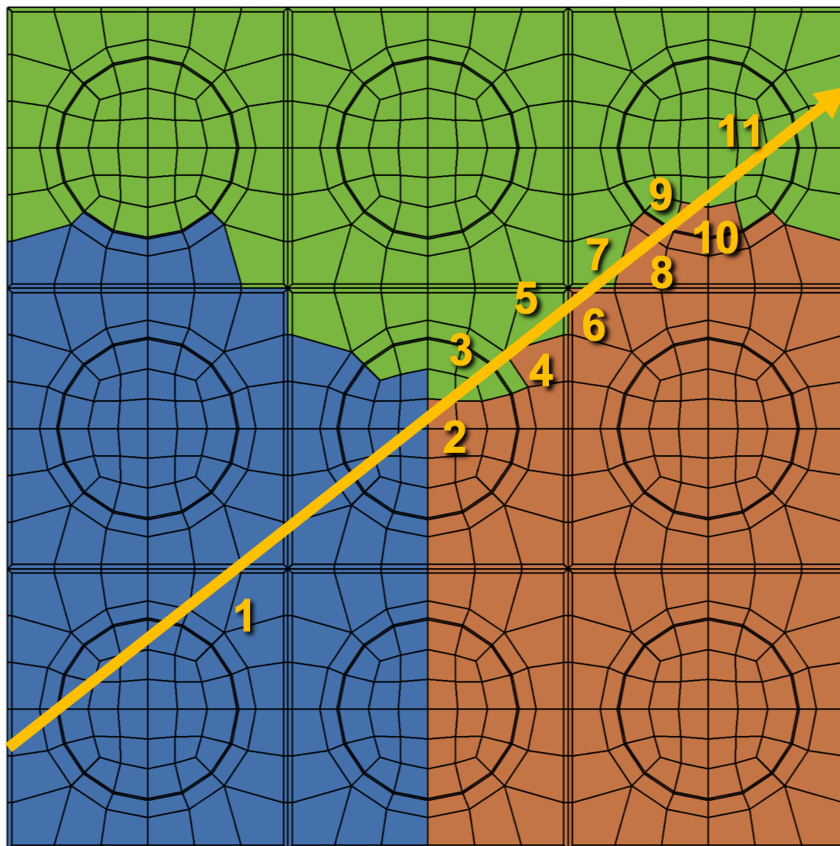


Figure 8.22. The number of global synchronizations the BS algorithm will incur as it tries to trace a ray across a jagged domain boundary.

A more modern, multi-hop communication algorithm was recently developed for use in radiation hydrodynamics simulations, primarily targeting astrophysics [105]. The Hybrid Adaptive Ray-Moment Method (HARM^2) shares many similarities with `SMART`. In particular, it makes use of non-blocking sends and an asynchronous stopping criteria to allow rays to continue to trace without global synchronization. The HARM^2 scheme can be found in Algorithm 20

Algorithm 20: HARM^2 Algorithm, slightly simplified for the use-case of this thesis. From [105].

```
1  compute max_rays
2  all_done = False
3  while not all_done do
4  |  while work remains do
5  |  |  advance all rays, return number destroyed and add to destroyed count
6  |  end
7  |  MPI_Isend rays to other MPI ranks
8  |  while MPI_Iprobe for rays returns true do
9  |  |  Non-blocking MPI MPI_Iprobe other MPI ranks for rays
10 |  |  if MPI MPI_Iprobe returns true then
11 |  |  |  Blocking MPI_Receive rays
12 |  |  end
13 |  end
14 |  Non-blocking MPI_Testsome rays MPI_Isend requests
15 |  Non-blocking MPI_Testsome destroyed counts MPI_Isend requests
16 |  while MPI_Iprobe for destroyed counts returns true do
17 |  |  MPI_Iprobe other MPI ranks for destroyed counts
18 |  |  if MPI_Iprobe returns true then
19 |  |  |  Blocking MPI_Receive MPI ranks destroyed counts
20 |  |  end
21 |  |  if destroyed count > previous destroyed count then
22 |  |  |  MPI_Isend destroyed count to all MPI ranks
23 |  |  end
24 |  |  if no work remains and Ray Send Requests == 0 and destroyed count send
   |  |   Requests == 0 and sum of destroyed count == max_rays then
25 |  |  |  all_done = True
26 |  |  end
27 |  end
28 end
```

Examining Algorithm 20 some immediate differences between HARM^2 and `SMART` can be seen. Firstly, blocking `MPI_Receive` operations are used to receive data, which limits the overlap of communication and computation. Secondly, it traces all rays that are currently available, before doing any communication, again limiting communication overlap. In addition, tracing all rays delays interior ranks from receiving any

work until it propagates inward. Thirdly, while the finishing criteria is asynchronous, it is not efficient. If any rank detects that some rays have finished, it broadcasts, using MPI_Isend, the new count of destroyed rays to all other MPI ranks. This is a scalability problem once the number of ranks reaches a certain level.

Both the BS and HARM^2 algorithms were implemented within the RayTracingStudy of SMART. To simplify development, the BS algorithm was allowed to use the asynchronous stopping criteria from SMART. The HARM^2 algorithm, however, is faithfully recreated. Having all three algorithms available within the same code simplifies their testing. The object-oriented design of SMART and MOCkingbird allowed these algorithms to be introduced without having to change MOCkingbird and with only minimal additions to SMART.

Two problems are run to test these algorithms. First, the weak scaling problem from §8.3.1 is tried. That problem has perfect partitioning, so this is a chance to see these algorithms in the best-case scenario. Next, the strong scaling study from §8.3.3 is tested.

8.3.4.1 *Weak Scaling*

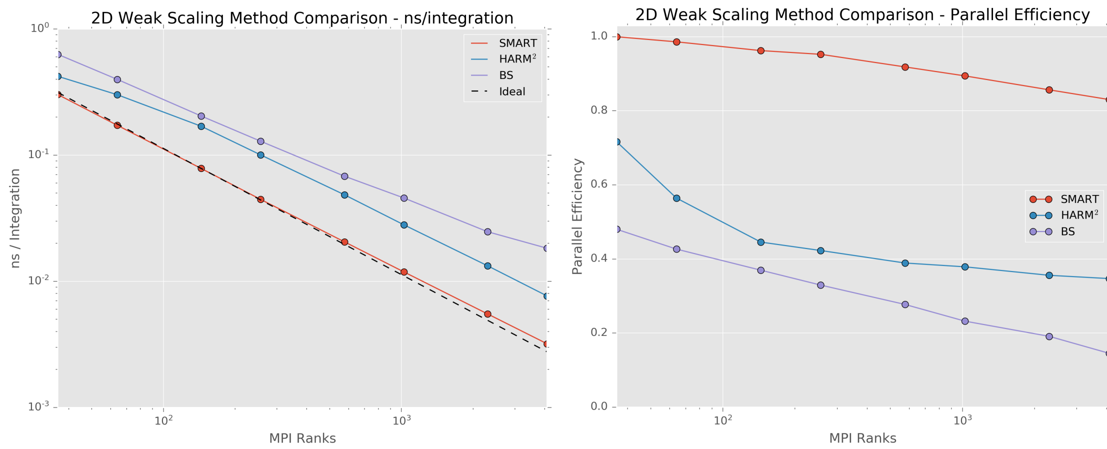
The weak scaling problem from §8.3.1 is tested with the three algorithms. Only a couple of changes were made: the number of azimuthal angles was set to 64, and the azimuthal spacing was set to 0.01 cm. This was done because it is anticipated that both BS and HARM^2 have poor enough performance that a large number of rays would take too long to test. Secondly, using fewer rays makes this a harder test, as there is less parallel work to do.

The results of this test can be viewed in Figure 8.23. It's immediately obvious from Figure 8.23b that neither BS nor HARM^2 perform well. Both algorithms start at a major disadvantage to SMART, and neither of them catches up. While SMART is still above 80% efficiency at 4096 MPI ranks HARM^2 and BS are below 40% and 20% respectively.

8.3.4.2 *Strong Scaling*

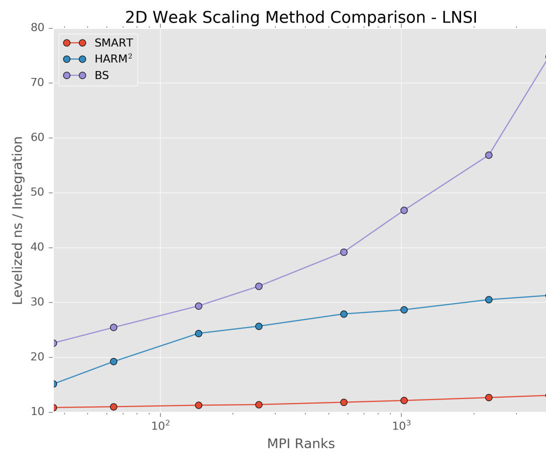
The next test is the strong scaling test from §8.3.3. Similarly to the weak scaling test, the number of azimuthal angles was lowered to 64, and the azimuthal spacing was set to 0.01 cm. This represents a significant challenge to both BS and HARM^2.

Figure 8.24 shows that there is an even wider gulf between the algorithms on this test. Compared to SMART, HARM^2 begins at 55% and ends at 35% parallel efficiency. BS has an even tougher time, ending at 20% parallel efficiency when compared to SMART. Looking at Figure 8.24c, BS is nearly four times slower than SMART when using 4096 MPI ranks. SMART fares well in this test, finishing with 85% parallel efficiency at 4096 MPI ranks.



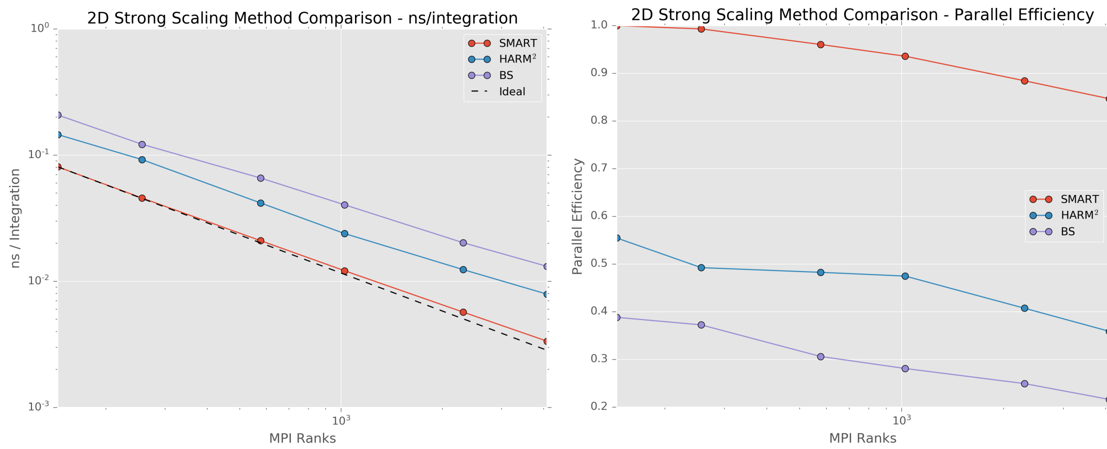
(a) ns/integration.

(b) Parallel efficiency.

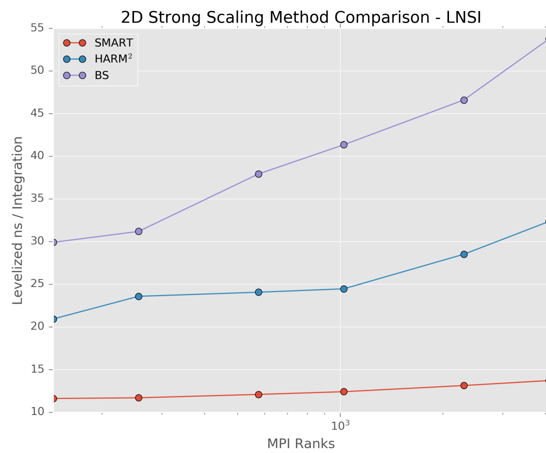


(c) LNSI

Figure 8.23. Comparison of 2D weak scaling performance with three communication algorithms.



(a) ns/integration. **(b)** Parallel efficiency.



(c) LNSI

Figure 8.24. Comparison of 2D strong scaling performance with three communication algorithms.

8.3.4.3 *Conclusion*

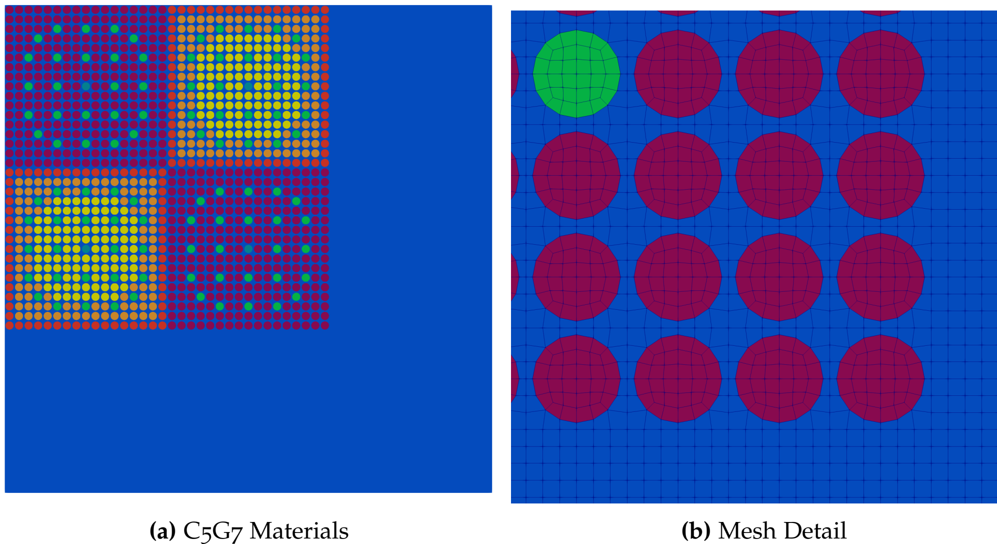
SMART was tested against two other algorithms: Bulk Synchronous (BS) and HARM^2. Two different tests were performed: weak and strong scaling in 2D. Both tests were conclusive: neither algorithm was competitive with SMART. Both algorithms were significantly slower at the low end of the scaling test and only fell further behind as more MPI ranks were used.

9 BENCHMARK PROBLEMS

9.1 INTRODUCTION

The objective of this thesis is to develop a scalable, unstructured mesh, method of characteristics neutron transport code: `MOCkingbird`. The previous chapters have described the algorithms `MOCkingbird` utilizes and their parallel performance. This chapter explores their efficacy for solving steady-state, k-eigenvalue, neutron transport benchmarks. Four benchmarks will be analyzed: 2D C5G7 [85], 3D C5G7 Rodded-B [108], 2D BEAVRS [87] and 3D BEAVRS. While solution accuracy is important, these benchmarks are also used to probe the performance characteristics of `MOCkingbird` on realistic problems. By the end of this chapter, it is demonstrated that `MOCkingbird` is capable of full-core, 3D, neutron transport simulation via MOC.

9.2 2D C5G7



(a) C5G7 Materials **(b)** Mesh Detail

Figure 9.1. Quarter-core C5G7 geometry and mesh detail for the bottom right corner of the core. Colors represent sets of fuel mixtures and the moderator.

The C5G7 benchmark [85] has often been used in the literature as a verification step for deterministic transport codes [2, 10, 109, 110]. As shown in Figure 9.1, the quarter-core 2D benchmark includes four assemblies with 17x17 fuel pins and a water reflector.

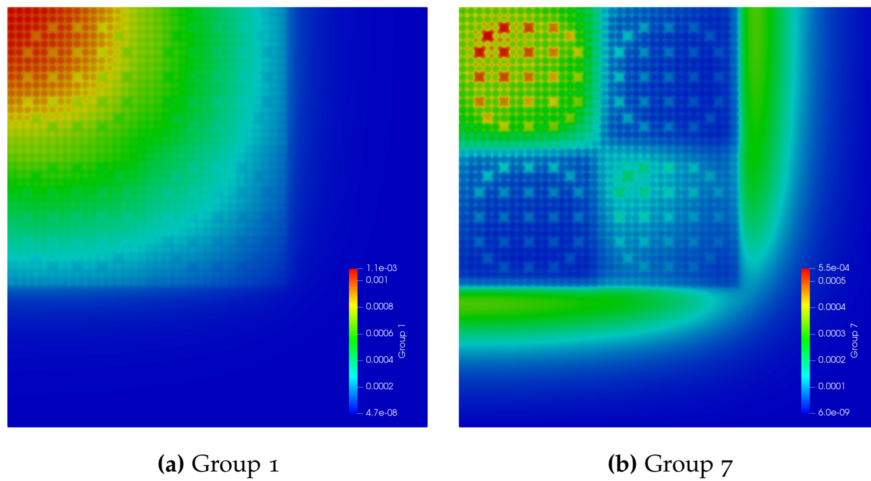
Table 9.1. `MOCkingbird` converged eigenvalues for increasing azimuthal angles with an azimuthal spacing of 0.01cm.

# Angles	# Sweeps	k_{eff}	$\Delta\rho$ (pcm)	AVG	MRE	RMS
4	648	1.18552	-102	1.95	1.78	2.42
8	656	1.18484	-169	0.43	0.38	0.53
16	656	1.18512	-139	0.27	0.20	0.38
32	655	1.18638	-14	0.28	0.22	0.40
64	654	1.18664	11	0.29	0.24	0.42
128	654	1.18676	23	0.30	0.24	0.43

One of the defining features of C5G7 is the heterogeneous representation of the pin cell with the exception of the clad and gap homogenization.

As noted in §5, the first step in producing solutions using `MOCkingbird` is to discretize the domain using a finite-element spatial mesh. The graph-based mesh generation system detailed in §5.2 together with the pin-cell creation capability in 5.1 can be utilized to generate the mesh needed for C5G7. Working with Leora Chapuis, the script in Listing 12.1 was developed for generating the mesh. These 150 lines of MOOSE input file syntax read in each of the pin-cells, create assemblies, stitch the assemblies together, generate mesh for the moderator, stitch everything together, and finally fix up boundary names. The mesh generation process takes 1.475s to build on a 2.7GHz Intel Xeon E5 processor. The process generates 112,132 elements for the mesh shown in Figure 9.1. The number of elements and their location are in with the spatial fidelity used in [2].

This mesh is used in an angular refinement study to explore the accuracy of `MOCkingbird`. Track spacing will be held constant at 0.01cm while the number of number of azimuthal angles is swept from 4 to 128. In the polar direction, TY [9] quadrature with three angles is used. The $\Delta\rho$ is computed against the MCNP benchmark reference k_{eff} of 1.18655. Convergence was judged using a fission source RMS change between successive fission source iterations of 1e-5.



(a) Group 1 **(b)** Group 7

Figure 9.2. Group flux results using 128 azimuthal angles.

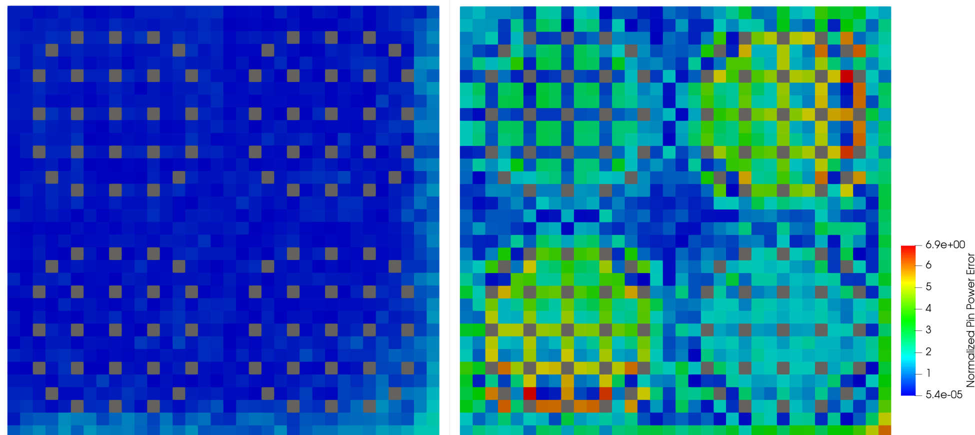


Figure 9.3. Relative error in normalized pin powers. Left: 128 azimuthal angles, Right: 4 azimuthal angles.

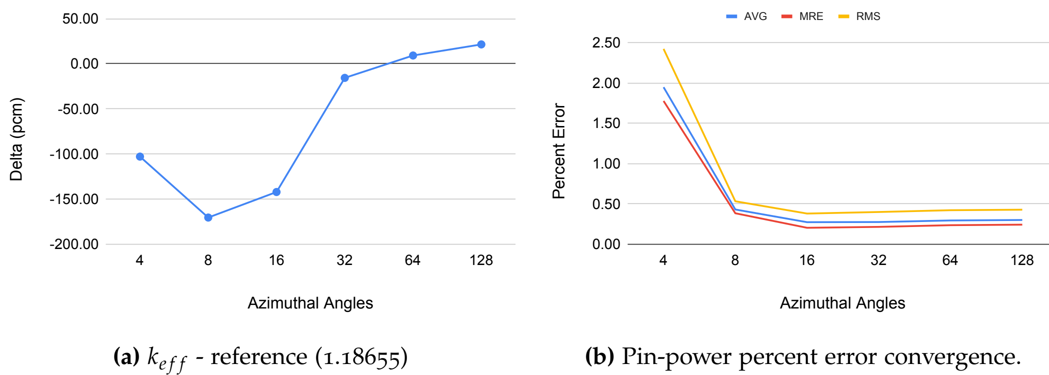
The results of this study can be found in Table 9.1. `MOCkingbird` converges as the number of azimuthal angles increases. These numerical results and the qualitative fluxes found in Figure 9.2 agree well with other published results [2]. In addition, the normalized relative pin power error, shown in Figure 9.3, is in line with the results found in [10], showing larger error in the fuel pins near the reflector. Finally, within Table 9.1 AVG, MRE and RMS are measures of the fission source error and are defined as:

$$AVG = \frac{\sum_{n} |e_n|}{N}, \tag{9.1}$$

$$MRE = \frac{\sum_n |e_n| \cdot p_n}{N \cdot p_{avg}}, \tag{9.2}$$

$$RMS = \sqrt{\frac{\sum_n e_n^2}{N}}, \tag{9.3}$$

where n is iterating over fuel pins, N is the total number of fuel pins, p is power and e_n is the percent error as compared to the benchmark. The values in 9.1 can also be graphically viewed in 9.4.



(a) k_{eff} - reference (1.18655) **(b)** Pin-power percent error convergence.

Figure 9.4. Convergence in eigenvalue and pin-power for 2D C5G7.

Table 9.2. Performance characteristics for each run in the angular refinement study.

# Angles	Solve Time(s)	Iterations	Intersections	LNSI
4	66.646	648	12014062	41
8	77.244	656	22442732	31
16	115.04	656	44105097	25
32	194.8	655	88073264	24
64	362.2	654	175991664	24
128	770.7	654	351892634	24

All of these simulations utilized 160 MPI ranks spread across 4 nodes of the Lemhi cluster. Using 160 MPI ranks is stretching the scalability of this problem with an average of only 700 elements per rank. That is nearly ten times more MPI ranks than would be recommended for running a finite-element simulation using this mesh. Table 9.2 details the performance of each run for the angular refinement study. The "Solve

Time" is the total time to convergence in seconds spent doing source iterations. The number of iterations and intersections per iteration are also shown in Table 9.2. Looking at the LNSI, as the amount of work per MPI rank increased, the LNSI decreased, plateauing near 24 ns/integration.

9.3 3D C5G7

The C5G7 benchmark has been extended to a full suite of 3D benchmarks [108]. There are three primary configurations in the 3D benchmark: *Unrodded*, *Rodded A* and *Rodded B*. The focus of the current study is the *Rodded B* case due to it offering the largest challenge by having multiple banks of control rods partially inserted.

The mesh for *Rodded B* was generated similarly to the 2D mesh using the graph-based mesh generation system and using the same pin-cells as the 2D mesh. In this way, the radial fidelity was set by the 2D mesh; however, the axial fidelity is left to be determined. The number of axially extruded elements needed to achieve a high-fidelity result is the focus of this verification effort.

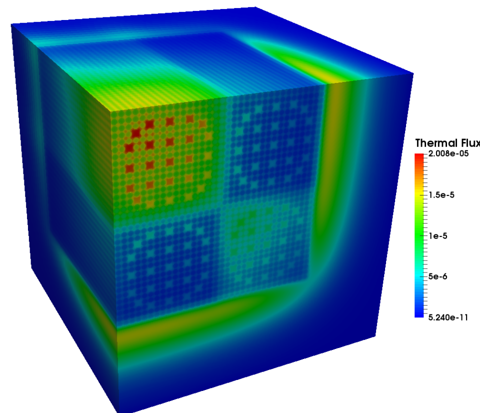


Figure 9.5. Thermal flux for the 3D mesh with 50 axial layers.

Four meshes were created with 50, 100, 200 and 400 axial layers. Correspondingly, the meshes contained 5.6M, 11.2M, 22.4M, and 44.8M elements, respectively. After examining the results in Table 9.1, the number of azimuthal angles for track generation was chosen to be 32. An azimuthal spacing of 0.1cm was selected to balance accuracy and problem size. In 3D, the number of polar angles and axial ray spacing must also be determined. The current study used 6 polar angles with 0.3cm axial spacing. Ultimately, these settings generated 53.9M tracks within the 3D domain.

The meshes were used in runs of `MOCkingbird` that utilized 400 cores on the Lemhi supercomputer. The convergence criteria was $1e-5$ relative change in the fission source. The results of these runs can be found in Table 9.3.

Table 9.3. Eigenvalues and fission source errors for each mesh used for the 3D C5G7 *Rodded B* configuration. The reference k_{eff} is 1.07777.

Axial Elev. (cm)	k_{eff}	$\Delta\rho$ (pcm)	AVG	RMS
50	1.073129	-464	8.92E-01	1.16334
100	1.076307	-146	5.71E-01	8.25E-01
200	1.077239	-53	4.92E-01	7.39E-01
400	1.077498	-27	4.70E-01	7.13E-01

The results show that `MOCkingbird` can achieve a k_{eff} within 27pcm of the benchmark solution (1.07777) on the finest grid. It is clear that axial fidelity plays a large roll in accuracy, as the coarsest mesh has an eigenvalue that is 464 pcm too low. The convergence of `MOCkingbird` toward the true eigenvalue and fission source as the mesh is refined is a good indicator that the 3D parallel unstructured mesh MOC solution algorithm is working correctly.

Table 9.4. "Polar angle study for 3D C5G7."

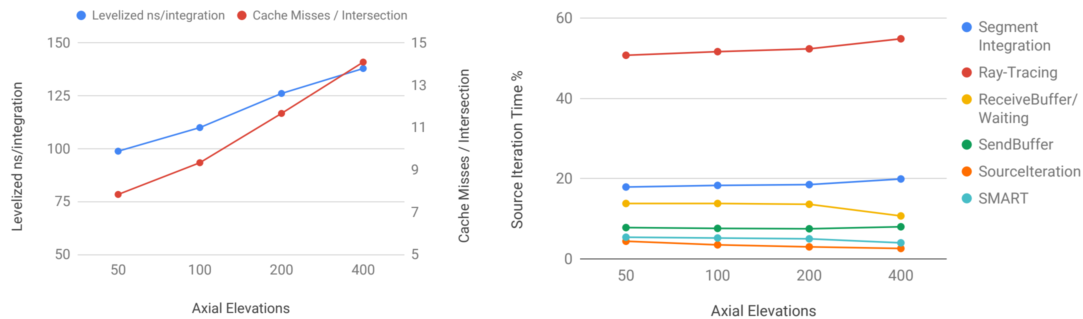
# Angles	# Sweeps	k_eff	$\delta\rho$(pcm)	AVG	MRE	RMS
2	708.0	1.082	465	1.519	1.250	2.259
4	706.0	1.078	29	0.602	0.482	0.960
6	709.0	1.077	-27	0.470	0.386	0.713
8	710.0	1.077	-43	0.428	0.358	0.629
10	711.0	1.077	-49	0.408	0.344	0.587
12	711.0	1.078	-50	0.396	0.336	0.563

A polar angle study was also completed. The 400 axial layer mesh was used, with the same 32 azimuthal angles, 0.1 cm azimuthal spacing, 0.3 polar spacing settings as the previous test. The detailed results can be viewed in Table 9.4. While the error is already low by 6 polar angles a stationary point is not met until 10 angles.

Table 9.5. Performance characteristics for solving 3D C5G7 *Rodded B*.

Axial Elev.	Solve Time(s)	Iter.	LNSI	Intersect. / It.	Cache Miss / Intersect.
50	15488.434	700	99	12386466041	7.84
100	18575.211	707	110	13380319921	9.33
200	24561.552	709	126	15370859231	11.7
400	33659.308	709	138	19351886010	14.11

Table 9.5 details the performance of `MOCkingbird` for the solution of C5G7 Rodded B. One interesting trend in Table 9.5 is that the LNSI increases as the amount of axial layers increases. All solves were performed using 400 MPI ranks; therefore, increasing the mesh density should, in theory, increase the amount of local work to be done and decrease LNSI. However, that is not the outcome.



(a) Comparison of LNSI and cache misses / intersection.

(b) Percentage of iteration time.

Figure 9.6. Visualizations of the performance of `MOCkingbird` for solving 3D C5G7.

After a detailed analysis, the increase in LNSI is due to an increase in cache misses. Traversing the unstructured mesh involves moving through memory in an unstructured pattern. With more mesh, it becomes less likely that the next element being traced is already in the processor's cache hierarchy. The Linux performance monitoring capability [111] was used to count the number of cache misses (memory accesses by the CPU which end up accessing main memory) during a transport sweep. That number, divided by the number of intersections, is reported in Table 9.5.

Figure 9.6 displays several performance metrics. Firstly, Figure 9.6a compares the increase in LNSI and cache misses as the number of axial layers is increased. Both show a similar trend, offering evidence that the increase in LNSI is due to cache misses.

Also, the Google Perf Tools (gperftools) [112] were used to examine the performance of this problem. Aggregate results across all MPI ranks from gperftools can be seen in 9.6b. The percentage of time, during a source iteration, spent in each of five different code segments is detailed. "Ray-Tracing" is the time spent in element traversal (Algorithm 7), "Segment Integration" is angular flux integration time, the send and receive buffer times are for communication and "SourceIteration" is everything else including source update and convergence checking. While the percentage of time spent in communication and "SourceIteration" stay relatively steady as the mesh density is increased, the time for tracing and, to a lesser extent, segment integration is growing.

This is in line with the idea that cache misses, due to moving through more unstructured geometry, are playing a roll in decreasing performance.

9.4 2D BEAVRS

The BEAVRS benchmark [87] was developed at MIT as a realistic benchmark with real-world measurements for comparison. It is a full-core benchmark containing 193 fuel assemblies, each with an array of 17x17 fuel rods, guide tubes, and burnable poisons. Two fuel cycles are contained within the benchmark; however, the current study focuses on the core configuration from cycle 1. This section solves a two-dimensional configuration of the BEAVRS benchmark while §9.5 explores a fully three-dimensional solution.

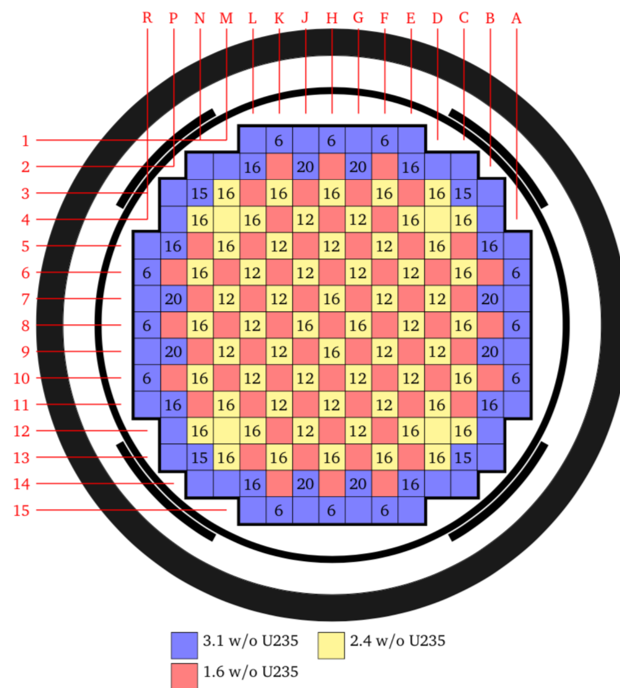
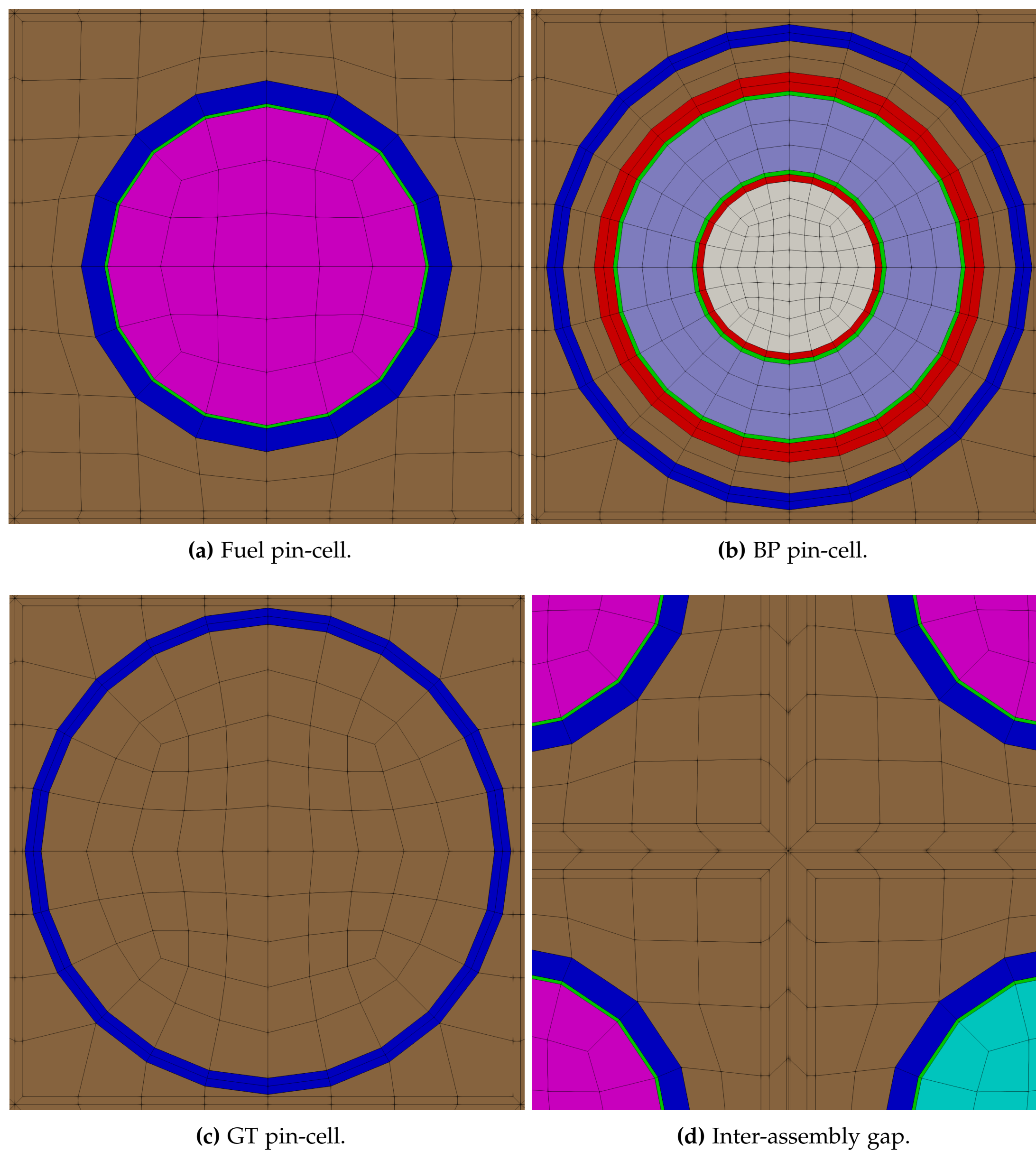


Figure 9.7. BEAVRS benchmark assembly layout for cycle 1. From [87]

The layout of the assemblies for cycle 1 can be seen in Figure 9.7. There are three different levels of enrichment: 3.1%, 2.4%, 1.6%, and multiple layouts of burnable poisons (6, 12, 16). Each pin-cell is fully specified by the benchmark, without any homogenization. This provides a challenging test for `MOCkingbird` due to needing to mesh each of the intricate features of the pin-cells, including gap, clad, and spacer grids.

9.4.1 *Meshing*

The 2D mesh is built using pin-cells generated from Cubit, as shown in Figure 9.8 which are then utilized in the `MeshGenerator` system to create assemblies and ultimately the core. In addition to the fuel assemblies, meshing the baffle and bypass water is accomplished by generating "water assemblies" in Cubit which are then rotated to fit around the core and added to the core pattern. Figure 9.9 shows the baffle and water meshes. A view of the full geometry can be found in Figure 9.10. Ultimately, the mesh used for this study contained 10,553,408 elements.



(a) Fuel pin-cell. **(b)** BP pin-cell.

(c) GT pin-cell. **(d)** Inter-assembly gap.

Figure 9.8. Pin-cell meshes and inter-assembly gap mesh.

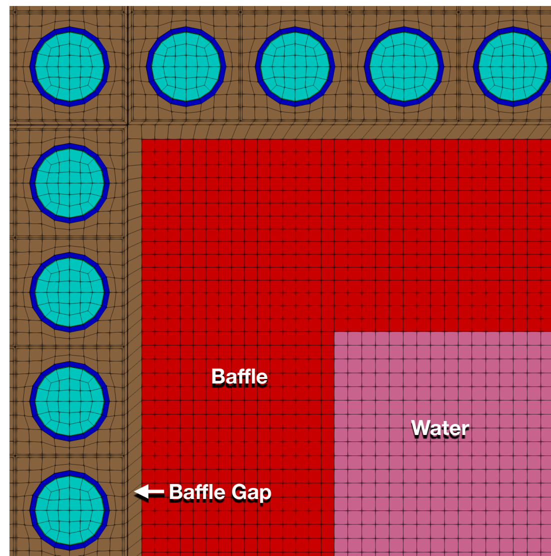


Figure 9.9. Example of the baffle and water mesh surrounding the core.

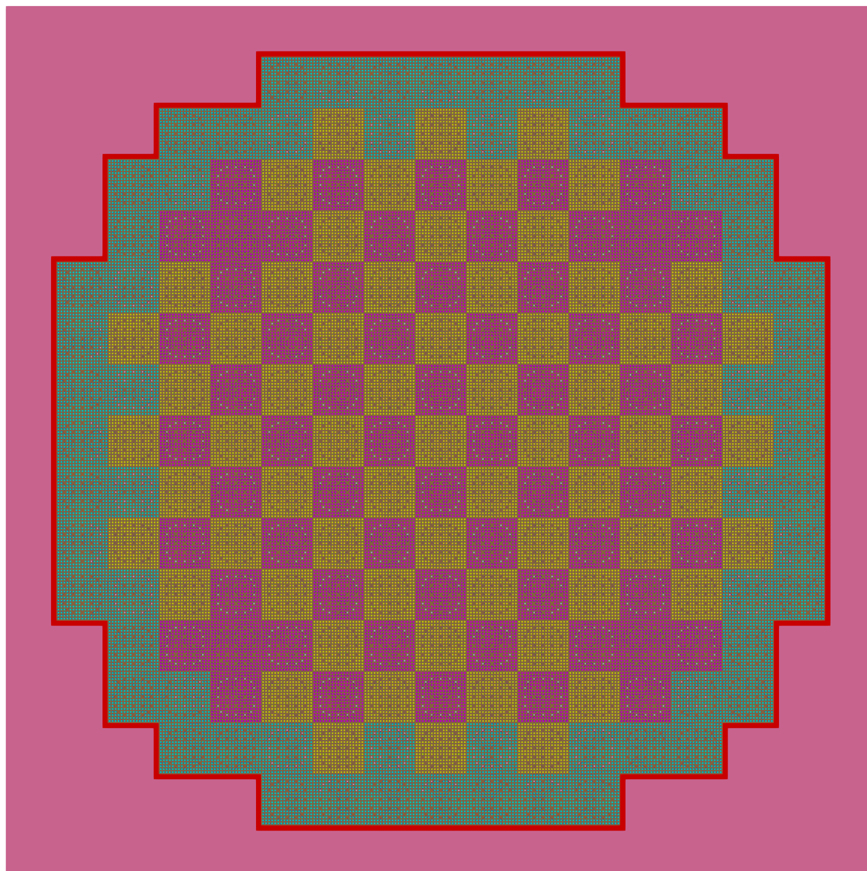


Figure 9.10. As-meshed geometry for the 2D BEAVRS core.

9.4.2 *Cross Sections and Reference Values*

As mentioned in §4.2, `MOCkingbird` relies on cross sections being provided as an input. The BEAVRS benchmark does not include a set of cross sections; rather, it specifies the materials in the reactor analysts need to generate the appropriate multi-group or continuous energy cross section to analyze the physical reactor. However, a detailed `OpenMC` model exists [113] which can be utilized to generate such cross sections for the Hot Zero Power, isothermal case at 975 ppm boron and 560K. For this study, Guillaume Giudicelli worked with Zhaoyuan Liu at MIT, using `OpenMC` and the Cumulative Migration Method (CMM) [16] to build transport-corrected 70 energy group cross sections. The group structure is detailed in Appendix 13. The cross sections were developed using material tallies (over all regions in the reactor model that contain the material) for these materials:

- Air
- SS304
- Helium Gap
- Zircaloy
- 1.6 w/o U235 Fuel Pellet
- 2.4 w/o U235 Fuel Pellet
- 3.1 w/o U235 Fuel Pellet
- Borosilicate Glass
- Coolant
- Support Plate Water
- Support Plate Stainless Steel
- Water Outside of Baffle

These 70-group cross sections represent the spatial averaging of reaction rates over distributed material regions, and they do not reflect small local effects - such as the Dancoff effects that arise from resonance self-shielding of strong absorbers. For a more thorough discussion of such effects, see Giudicelli [114] or Gibson [115].

0.726	0.819	0.821	0.986	0.882	0.970	0.937	0.988
	0.785	0.955	0.881	1.016	0.904	1.126	1.046
		0.880	1.035	0.921	1.012	0.937	0.922
			0.961	1.100	1.024	1.174	0.767
				1.435	1.187	1.250	
					1.233	0.921	

Figure 9.11. Normalized assembly powers for the BEAVRS benchmark as computed by `OpenMC`. `OpenMC` computed eigenvalues: 2D $k_{eff} = 1.00491$, 3D $k_{eff} = 1.00024$

Giudicelli used `OpenMC` to compute a normalized set of assembly powers. Those normalized powers collapsed to one of the symmetric octants of a core can be found in Figure 9.11, and is used as reference values for comparison with the calculations done with `MOCkingbird`. Also, using this set of cross sections with `OpenMOC`, he conducted a detailed 2D calculation, without axial leakage, which produced a k_{eff} of 1.00188 in 2D. That result utilized a flat source model with pin-cell moderator discretized with 8 rings and 8 sectors and 4 rings and 4 sectors in the fuel. The `OpenMOC` calculation utilized 64 azimuthal angles with a 0.05cm spacing and TY polar quadrature with 3 angles. The 300 pcm difference between `OpenMOC` and `OpenMC` is due to equivalence methods, which is the focus of Giudicelli's research [114].

9.4.3 *Results*

Utilizing this problem setup, `MOCkingbird` was run on the Lemhi cluster at the Idaho National Laboratory to solve the 2D full-core eigenvalue problem. The problem settings and computational requirements can be found in Table 9.6. With these settings, a k_{eff} of 1.00231 was computed, 43 pcm above the `OpenMOC` solution of Giudicelli. In addition, as shown in Figure 9.12, the normalized assembly power differences are all within 1%.

Table 9.6. Problem settings and computational requirements for 2D BEAVRS.

Number Of Elements	10,558,033
Fission Source Convergence	1e-5
Azimuthal Angles	64
Azimuthal Spacing	0.05 cm
TY Polar Quadrature Angles	3
MPI Processes	4000
Solve Time	4801.7 s
LNSI	19
Intersections	2,035,381,032
Number of Transport Sweeps	2200

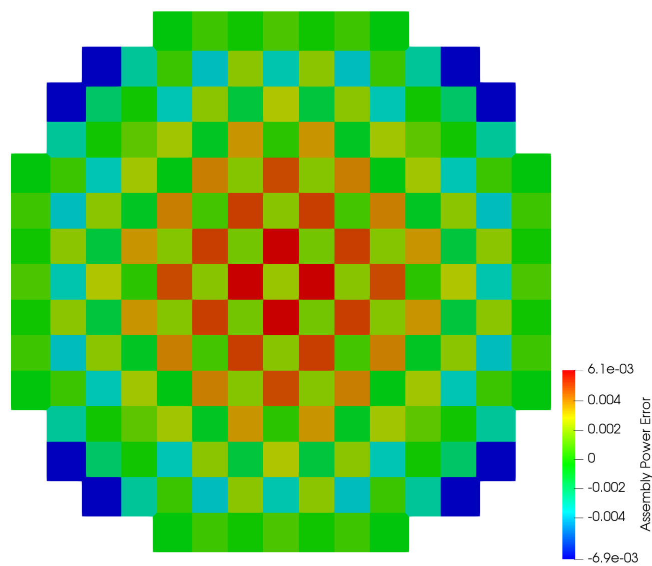


Figure 9.12. Normalized assembly power differences for the 2D BEAVRS solution computed by `MOCkingbird` compared to the OpenMC Monte Carlo result.

9.13.

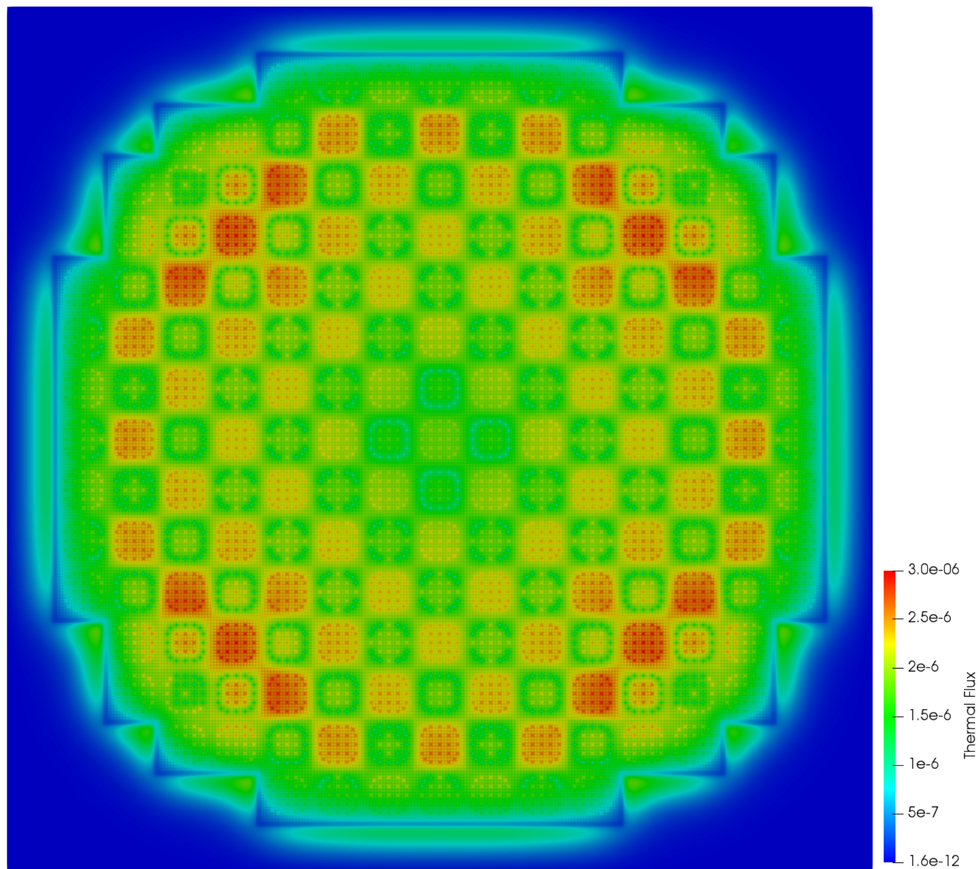
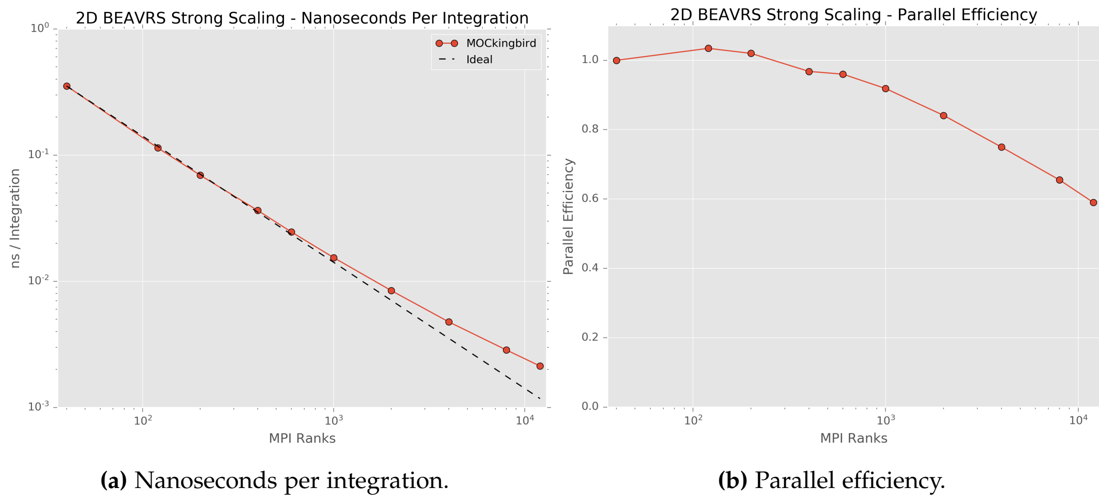


Figure 9.13. Thermal flux (0 ev to 9.87 eV) for the 2D BEAVRS benchmark.

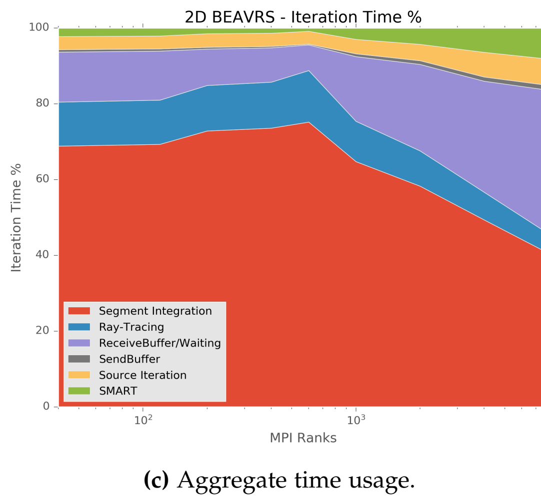
9.4.4 *Scalability*

The 2D BEAVRS benchmark problem represents an ideal case to test scalability with realistic geometry and angular/spatial quadratures. A scalability study was performed with a mesh that did not represent the inter-assembly gap and therefore had 10,343,424 total elements. The weighted ParMETIS partitioner was used for partitioning.



(a) Nanoseconds per integration.

(b) Parallel efficiency.



(c) Aggregate time usage.

Figure 9.14. Strong scaling results for 2D BEAVRS.

The results of the strong scaling study can be viewed in Figure 9.14. The nanoseconds per intersection shown in Figure 9.14a are close to ideal out to 1024 MPI processes, with performance gradually tapering off after that point. This behavior is shown in Figure 9.14b, where parallel efficiency goes from over 80% at 1024 MPI processes down to 40% using 18423 MPI processes. At 18432 processes, each MPI rank is only responsible for 550 elements. This broad range in scalability makes `MOCkingbird` a flexible tool for full-core analysis.

Finally, Figure 9.14c displays the percentage of time used by different parts of the code as the strong scaling is performed. From that figure, it is clear that up until 1k MPI ranks, the algorithm is working well to hide the extra communication. After that point, the time spent waiting for new messages starts to increase. This is indicative of work starvation beginning; there is not enough work to keep all of the cores busy

all of the time. There is a smooth degradation as more MPI ranks are added, which echoes the smooth drop-off in efficiency in 9.14b.

9.5 3D BEAVRS

Three-dimensional simulation of the BEAVRS core is challenging due to the size of the domain and the intricate axial detail which must be modeled. That axial detail is shown in Figure 9.15. In all, 34 separate axial zones must be accounted for. In addition, as shown in [3, 101] and in §9.3, a fine axial discretization is needed for accuracy: optimally < 2.0cm mesh layers. With the core being 460cm in height that is more than 230 layers in the axial mesh. Due to constraints imposed by the geometrical axial elevations, 241 axial layers were required. If the full-core mesh from the previous section is used as a template and extruded on 241 elevations, that is more than 2.5 billion elements.

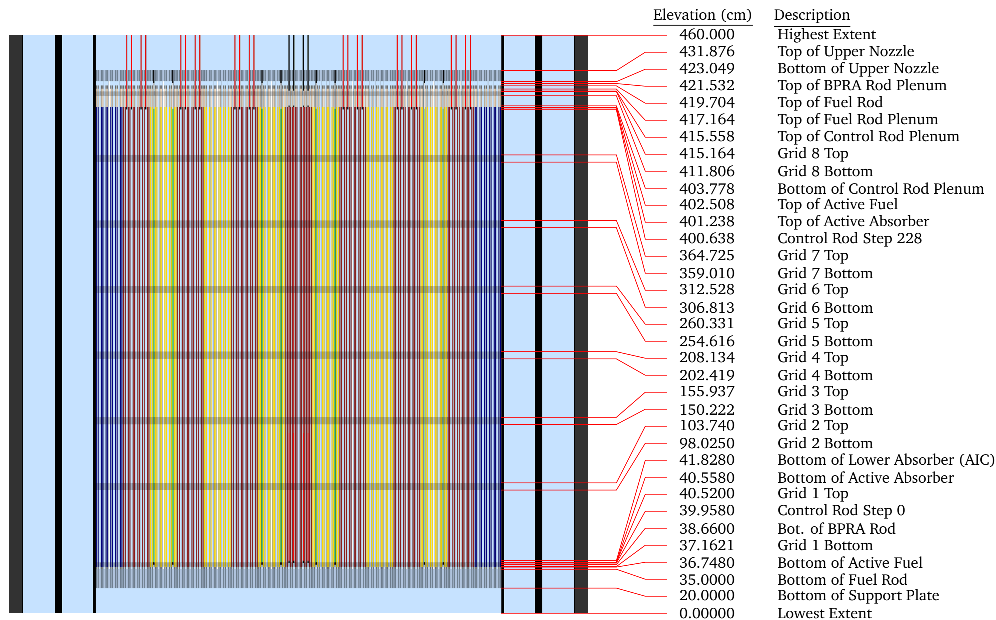


Figure 9.15. BEAVRS axial elevation specification from [87].

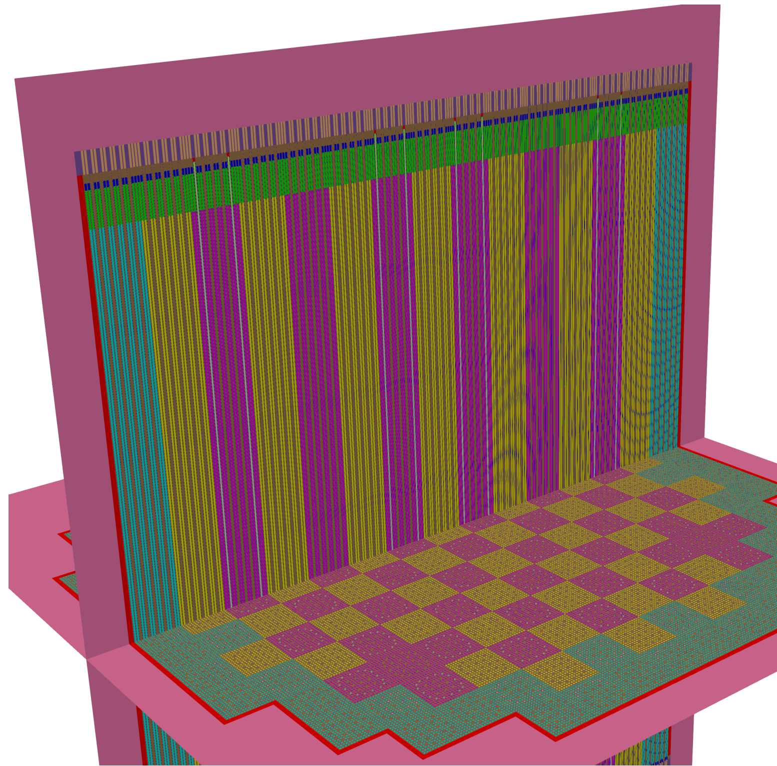


Figure 9.16. Slices showing the detail in the top and midplane of the, as meshed, full-core geometry.

9.5.1 *Geometry and Meshing*

Before running the 3D problem, the 3D mesh needs to be generated. This is done through an "extrusion" process as previously detailed. However, due to the size and complexity of the 3D mesh for BEAVRS, this is an intricate process involving several steps. The target number of cores used for solving the 3D BEAVRS problem was chosen to be 12,000 due to scheduling constraints on the Lemhi supercomputer. Therefore, the mesh generation process ultimately needs to create a mesh suitable for running with that number of MPI ranks.

Original Elevations	Original Materials
460	Highest Extent
431.876	Top of Upper Nozzle
423.049	Bottom of Upper Nozzle
421.532	Top of BPRA Rod Plenum
419.704	Top of Fuel Rod
417.164	Top of Fuel Rod Plenum
415.558	Top of Control Rod Plenum
415.164	Grid 8 Top
411.806	Grid 8 Bottom
403.778	Top of Active Fuel
402.508	Top of Active Absorber
401.238	Control Rod Step 228
400.638	Grid 7 Top
364.725	Grid 7 Bottom
359.01	Grid 6 Top
312.528	Grid 6 Bottom
306.813	Grid 5 Top
260.331	Grid 5 Bottom
254.616	Grid 4 Top
208.134	Grid 4 Bottom
202.419	Grid 3 Top
155.937	Grid 3 Bottom
150.222	Grid 2 Top
103.74	Grid 2 Bottom
98.025	Bottom of Lower Absorber (AIC)
41.828	Bottom of Active Absorber
40.52	Grid 1 Top
39.958	Control Rod Step 0
38.66	Bot. of BPRA Rod
37.1621	Grid 1 Bottom
36.748	Bottom of Active Fuel
35	Bottom of Fuel Rod
20	Bottom of Support Plate
0	Lowest Extent

Modified Elevations	Modified Materials
460	Highest Extent
431.876	Top of Upper Nozzle
423.049	Bottom of Upper Nozzle
421.532	Top of BPRA Rod Plenum
419.704	Top of Fuel Rod
417.164	Top of Fuel Rod Plenum
415.164	Grid 8 Top
411.806	Grid 8 Bottom
402.508	Top of Active Absorber and Fuel
401.238	Grid 7 Top
364.725	Grid 7 Bottom
359.01	Grid 6 Top
312.528	Grid 6 Bottom
306.813	Grid 5 Top
260.331	Grid 5 Bottom
254.616	Grid 4 Top
208.134	Grid 4 Bottom
202.419	Grid 3 Top
155.937	Grid 3 Bottom
150.222	Grid 2 Top
103.74	Grid 2 Bottom
98.025	Bottom of Lower Absorber (AIC)
40.558	Grid 1 Top - Bottom of BA
37.1621	Bottom of Active Fuel
35	Bottom of Fuel Rod
20	Bottom of Support Plate
0	Lowest Extent

Figure 9.17. The original BEAVRS elevations in cm (left) and the condensed elevations used here (right).

The first issue is that cyclical track laydown needs to be fine enough to go through every element. This means the smallest element in the mesh dictates how fine the spatial quadrature is. In 3D, a fine spatial and angular quadrature leads to an explosion in the number of tracks needed, making the problem intractable. To combat this, some modeling decisions were made to keep the problem at a reasonable size.

Within the axial elevations enumerated in Figure 9.15, there are some short layers. Several are below 1cm with the smallest being 0.394cm. If a mesh similar to the one from the previous section is extruded using those elevations, then tiny material regions, like the helium gap in a fuel pin-cell, end up as tiny 3D volumes requiring a fine track laydown. To combat this, several of the small axial layers in Figure 9.15 were merged.

To properly merge elevations like this, the material density in those areas would need to be modified when the cross sections are created. However, that was not done. Instead, the merging of these elevations creates a small modeling error. One helpful feature is that these tiny sections occur at the very top or very bottom of the reactor

where they have less impact on the solution. Also, their tiny size means that the error made in merging them is small. Figure 9.17 shows both the original elevations and the "condensed" elevations used to generate the 3D meshes for this study. In total, 8 elevations are removed, leaving 26 to be modeled.

Similarly, small features in the 2D radial mesh may be too small to hit with a reasonable track lay down in 3D. Because of this, the inter-assembly gap was left out of the 2D radial pattern used for generating the 3D mesh. That gap is only 0.00384cm wide, creating a tiny volume in 3D. Leaving out the inter-assembly gap causes the eigenvalue to be lower (less moderation) and could have a noticeable, but small, effect on assembly power distribution.

Generating the 3D mesh for BEAVRS used these steps:

1. Generate the 2D mesh.

2. Partition/split the 2D mesh for the number of MPI ranks to be used when running the 3D problem.

3. Using the number of MPI ranks the 3D problem will be run with, read in the split 2D mesh.

4. In parallel, extrude the mesh, changing the material definitions for each elevation.

5. Re-partition the 3D mesh using the weighted hierarchical partitioner.

6. Write out the individual partitions to separate files.

Step 2 is critical because there is not enough memory for every MPI rank to read the entire 2D mesh at the same time. Therefore, each MPI process reads a small piece of it. A new parallel extrusion capability was created for Step 4. Even for the 250 layers needed to create the 3D mesh, the parallel extrusion is nearly instantaneous (just seconds) due to using 12,000 MPI ranks. Without Step 5, the mesh partitioning would simply be a huge number of columns (corresponding to the partitioning of the 2D mesh). The weighted, hierarchical partitioner is used to find a partitioning with lower communication costs. Once those steps are complete, individual files, one for each MPI rank are created which contain a portion of the 3D mesh. When `MOCkingbird` is ultimately run to solve the k-eigenvalue problem, each MPI rank reads its particular part of the 3D mesh.

When this process is carried out for the quarter-core 3D mesh with 241 axial layers , it generates 12,000 files containing a total of 652,854,540 elements. Those files are 162GB in total size. To control the size of the full-core problem, and make it tractable to run within the available compute time, 128 axial layers were used. This brought the

12,000 mesh files to 320GB and 1,386,977,280 elements. That number of elements is, by far, the largest mesh ever used by a MOOSE-based application.

9.5.2 *3D Quarter-Core*

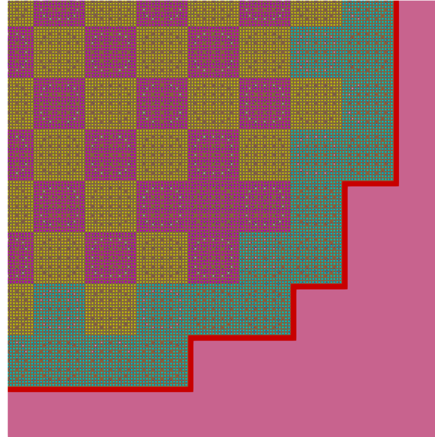


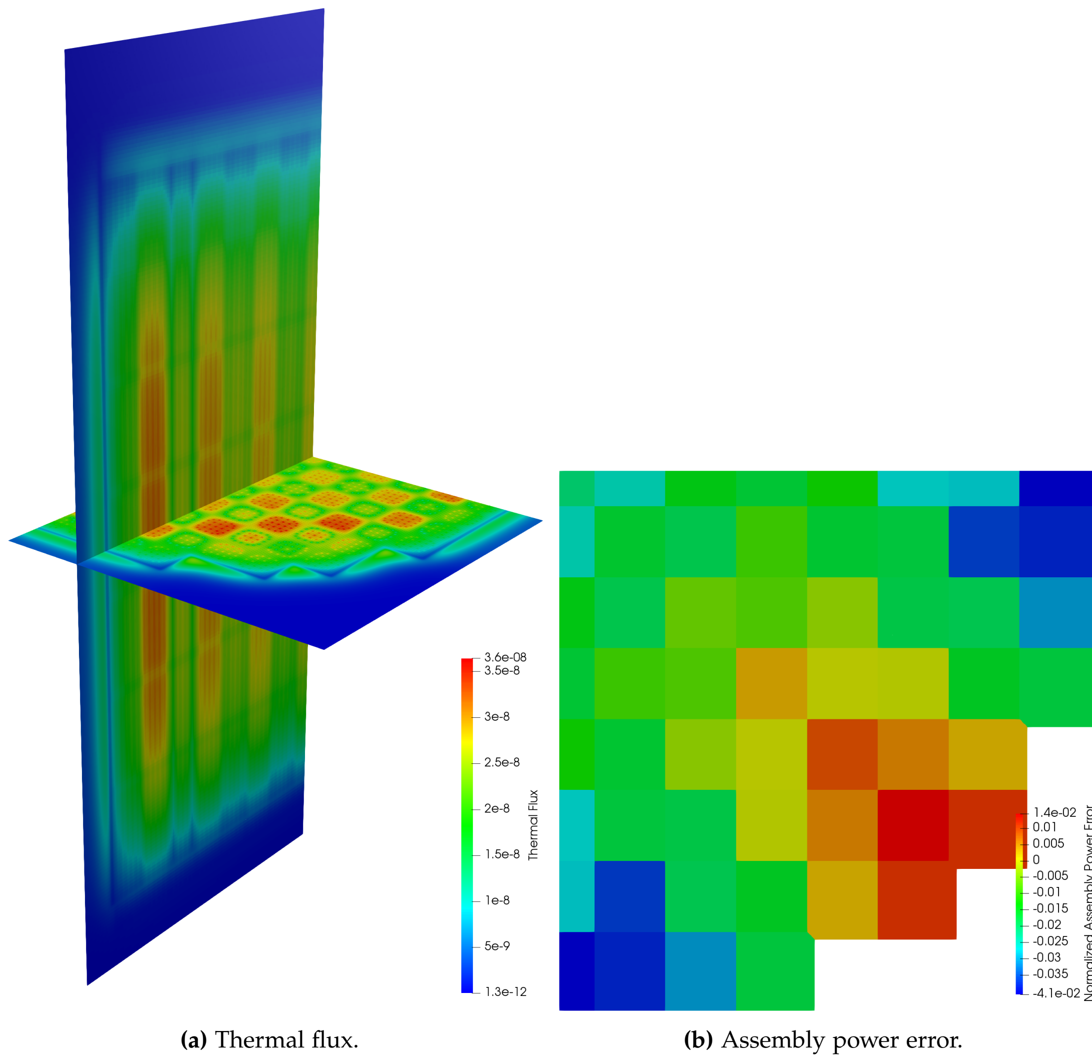
Figure 9.18. 3D quarter-core assembly layout. Reflective boundary conditions will be applied on the north and west sides of the domain, with vacuum on the south and east.

Before solving the full-core, symmetry is used to simplify the 3D BEAVRS problem to one-quarter of the domain. The quarter-core layout is shown in Figure 9.18, reflective boundary conditions are placed on the planes of symmetry. As mentioned in the previous section, the mesh for the 3D quarter-core contains 653M elements. The simulation was run using 12,000 cores on the Lemhi supercomputer. Problem settings and computational requirements for this run are in Table 9.7.

Table 9.7. Problem settings and computational requirements for the quarter-core 3D BEAVRS solution.

Number Of Elements	652,854,540
Axial Layers	241
Average Element Height	1.9 cm
Number of tracks	342,325,264
Fission Source Convergence	3.6e-4
k_{eff} Convergence	5.1e-6
Azimuthal Angles	32
Azimuthal Spacing	0.1cm
Polar Angles	8
Polar Spacing	1cm
MPI Processes	12,000
Solve Time	17.4h (12.26h)
LNSI	44 (31)
Intersections	489,712,983,737
Number of Transport Sweeps	500
k_{eff}	0.99381

As shown in Table 9.7 the problem didn't reach full convergence for the fission source ($1e-5$). The job ran out of time and quit. After the job completed, it was found that two nodes in-use by this job were running slower than they should. This was found by noticing aberrations in a scaling study and bisecting the node list until the slow nodes were isolated. After removing those nodes from service and restarting this job, it ran significantly faster. The solve time and LNSI in Table 9.7 reflect this. While the job was running, it was working with an LNSI of 44. After restarting the job without those nodes, the code achieved an LNSI of 31. Therefore, the solve time and LNSI in Table 9.7 both have a number in parentheses, which represents the performance without the two slow nodes.



(a) Thermal flux.

(b) Assembly power error.

Figure 9.19. Slices through the 3D quarter-core solution showing the thermal flux. Also, the assembly power error.

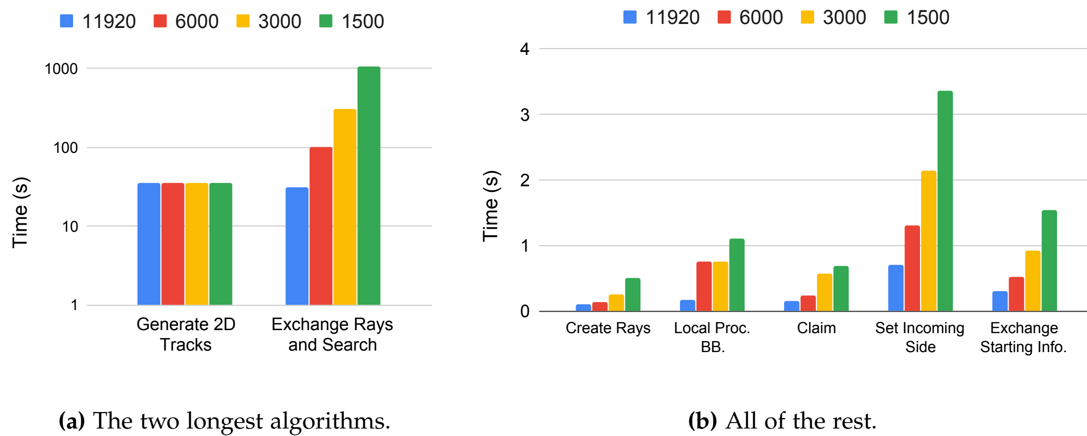
The thermal flux for the 3D quarter-core can be viewed in Figure 9.19a. It is visualized using two slices through the domain. The effect of the spacer grids is clearly seen as depressions in the thermal flux. As shown in Figure 9.19b without reaching convergence for the fission source, the axially integrated assembly power errors for this run are within +/- 4% when compared to the OpenMC computed assembly powers shown in Figure 9.11.

The eigenvalue achieved during this solve was 0.99381. This compares favorably to the result in [3] of 0.99677. It is 300 pcm low; however, both the spatial and angular discretization are coarser here, which would lead to a lower eigenvalue. In addition, this solution is not yet fully converged, and without acceleration, k_{eff} converges very slowly.

This large, three-dimensional problem is an ideal test-case for the track generation and claiming algorithms from §8.1. In addition to splitting the mesh for 12,000 MPI ranks, splits were also created for 6,000, 3,000 and 1,500. The problem was then started with each of these numbers of MPI processes, and timing was taken for the track generation and claiming algorithms.

Table 9.8. Timing for track generation and claiming.

MPI	Gen. 2D Tracks	Create Rays	Local BB.	Exchng. and Search	Claim	Incoming Side	Exchng. Start Info.
11920	35.072	0.102	0.178	31.256	0.161	0.704	0.3
6000	35.234	0.135	0.759	100.053	0.248	1.316	0.528
3000	35.159	0.266	0.76	311.039	0.57	2.153	0.921
1500	35.137	0.505	1.109	1056.766	0.696	3.356	1.541



(a) The two longest algorithms.

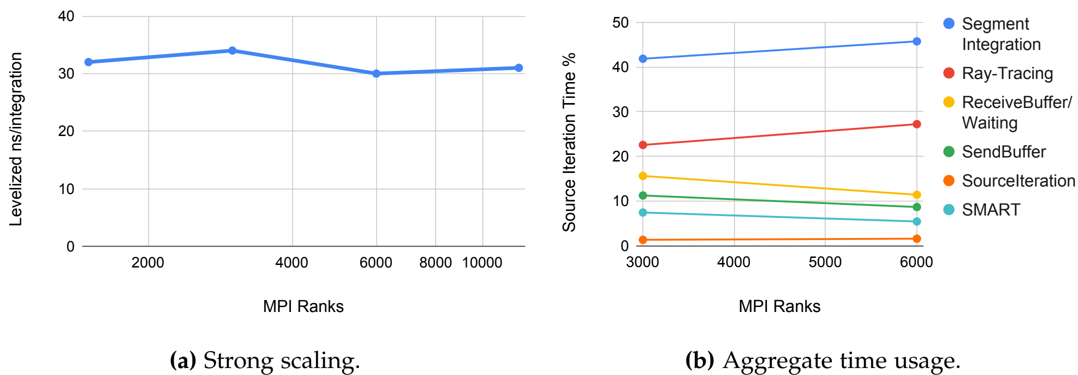
(b) All of the rest.

Figure 9.20. Time for track claiming algorithms for different numbers of MPI processes. The two algorithms taking the most time are shown in Figure 9.20a while the rest are in Figure 9.20b

The results of the track generation and claiming scaling study can be found in Table 9.8 with visualizations of the data in Figure 9.20. Examining Figure 9.20, it's clear that all of the algorithms scale well, with the exception of 2D track generation. In §8.1 it is explained that all 2D tracks are generated on all MPI ranks. However, tracks are split among the MPI ranks, so the rest of the algorithms are scalable. In particular, in Figure 9.20a, the asynchronous exchange of rays combined with searching for the starting element scales well. Since it involves a geometrical search when the number of MPI ranks is doubled, the time is cut to one-third. The "claim" time is the time to execute the one-sided, single-hop data push for uniquely claiming rays. It scales well, showing the efficacy of the MNBX sparse data exchange algorithm.

Table 9.9. Percentage of source iteration time in different code segments.

MPI Ranks	Segment Intg.	Ray-Tracing	Receive/Waiting	SendBuffer	SourceIteration
3000	41.9	22.6	15.6	11.2	1.3
6000	45.8	27.2	11.4	8.7	1.6



(a) Strong scaling.

(b) Aggregate time usage.

Figure 9.21. Strong scaling and iteration time percent for the 3D quarter-core BEAVRS problem. A constant LNSI equates to 100% parallel efficiency.

In addition, the scalability of the source iteration was also captured for the quarter-core problem. Figure 9.21 shows the results of this strong scaling study. `MOCkingbird` performs well, with the levelized ns/integration staying essentially constant over a large range of MPI processes. At 12000 MPI ranks, there are, on average, over 50,000 elements per MPI rank. This suggests, based on previous scaling studies, that this problem could still scale well to around 100,000 MPI ranks. `MOCkingbird` having such a large range of parallel efficiency makes it a flexible tool. If only 1500 cores on a cluster are available, `MOCkingbird` can run, but if 12000 are available, then `MOCkingbird` solves the same problem 8 times faster. This solve took (re-normalized) 12h. If 100,000 cores were used, there would still be 5,000 elements per MPI rank, which has been shown to have a high parallel efficiency for `MOCkingbird`. Assuming even 50% parallel efficiency, it would finish in under 3h. 100k cores is an important number because the next cluster at Idaho National Laboratory is likely to be in that range.

The percentage of source iteration time taken in each part of the code is shown in both Table 9.9 and Figure 9.21b. Data was only collected for the 3000 and 6000 MPI cases. The values stay relatively constant; however, the percentages for ray-tracing and segment integration slightly rise, as the number of core is doubled. This is showing that `SMART` is continuing to be effective at overlapping communication and computation.

9.5.3 *3D Full-Core*

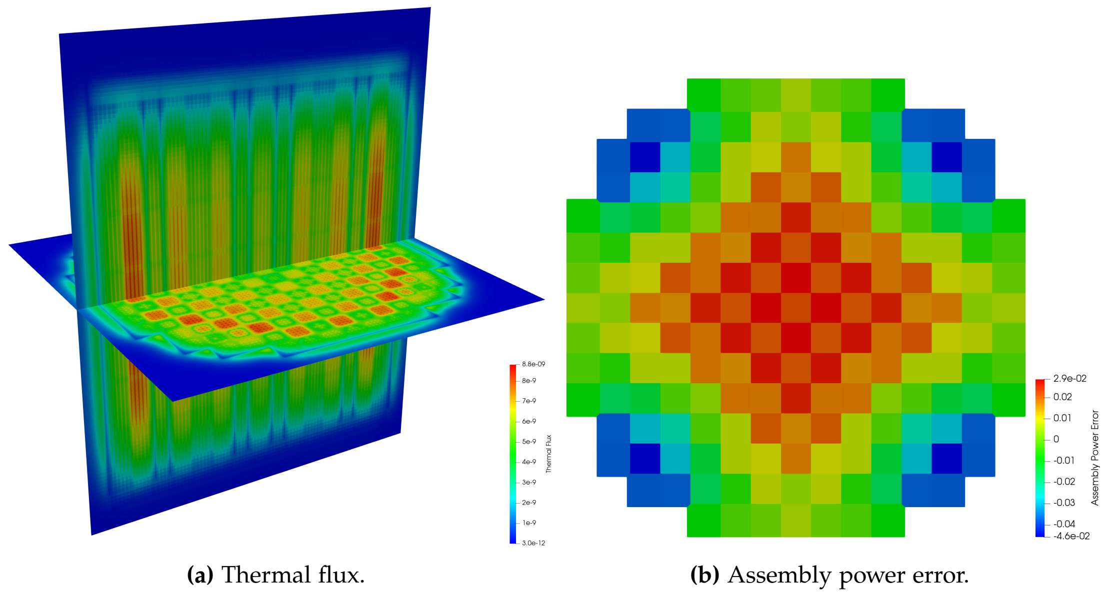
The full-core, 3D BEAVRS problem represents an enormous challenge for any neutron transport tool. Many groups have simulated it [116, 117, 118, 119, 120, 121, 122], some using homogenization [122] many others using Monte Carlo based codes [118, 119, 120, 121]. However, to date, only one code [3, 104] has utilized MOC for full-core 3D analysis of BEAVRS. However, [3] also, smartly, made use of the extruded nature of BEAVRS to gain efficiency and reduce memory use. Also, another effort [101] solved a full-core problem similar to BEAVRS using TRRM, which is a related method to MOC. There has yet to be a direct MOC calculation of full-core, 3D BEAVRS by a fully general, unstructured mesh, MOC code without any geometrical assumptions made within the solver.

Table 9.10. Problem settings and computational requirements for the full-core 3D BEAVRS solution.

Number Of Elements	1,386,977,280
Axial Layers	128
Average Element Height	3.6 cm
Number of Tracks	302,341,016
Fission Source Convergence	2e-4
k_{eff} Convergence	2.1e-06
Azimuthal Angles	32
Azimuthal Spacing	0.1 cm
Polar Angles	6
Polar Spacing	2 cm
MPI Processes	12,000
Solve Time	33.1 h
LNSI	33
Intersections	722,139,821,713
Number of Transport Sweeps	800
k_{eff}	0.99402

The full-core problem represents a significant increase in computational needs. If the quarter-core mesh from the previous section were simply replicated four times, the full-core mesh would include nearly 2.5 billion elements. To make the problem solvable by `MOCkingbird` with available computational resources, the axial mesh for the full-core simulation uses 128 layers. This brings the full-core mesh to 1,386,977,280

elements. In addition, as shown in Table 9.10 the polar quadrature is reduced to 6 polar angles with a polar spacing of 2cm. The overall effect of coarsening both spatially and angularly is that the number of tracks stays almost constant, and the number of intersections per iteration grows by less than two-times. However, solution accuracy could suffer [3].

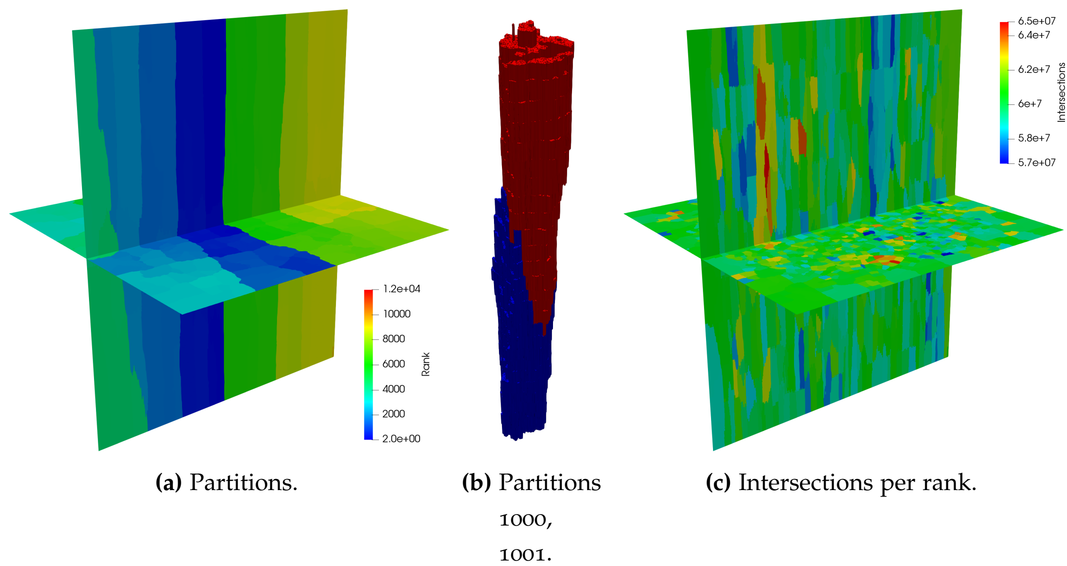


(a) Thermal flux.

(b) Assembly power error.

Figure 9.22. Thermal flux for the full-core shown on two slices through the core.

The thermal flux solution for the full-core problem is visible in Figure 9.22a. Just as in the quarter core result, the spacer grids are plain to see as thermal flux depressions. The assembly power error in Figure 9.22b shows a similar trend and magnitude to that in the quarter core. The k_{eff} is low at 0.99402, but that is to be expected with the coarse angular discretization and coarse axial layers.



(a) Partitions. **(b)** Partitions 1000, 1001. **(c)** Intersections per rank.

Figure 9.23. A view of the partitioning and the amount of work (intersections) per partition. Figure 9.23b shows a three-dimensional view of the partitions for ranks 1000 and 1001.

Figure 9.23 shows a view of the partitions in Figure 9.23a and the resulting amount of work (intersections) per partition in Figure 9.23c. While the partitions may look long and columnar, they are shorter and more block-like as can be seen in the image of the work balance in Figure 9.23c. The work is well balanced with the partition with the highest number of intersections being $6.5e7$, while the least intersections were $5.6e7$ a difference of about 14%. Figure 9.23b shows two of the partitions in three-dimensions. It is obvious that the partitioning is not optimal; the long, columnar nature of the partitions has more surface area than is necessary. With this partitioning, `MOCkingbird` achieves 31 ns/integration. This time can be reduced with better partitioning.

This solution, though not perfect, represents a large step forward for unstructured mesh MOC. The result, and the performance of 33 levelized ns/integration show that unstructured mesh MOC is a viable capability for full-core reactor simulation. However, it is obvious from the hundreds/thousands of iterations required to converge that some form of acceleration should be investigated quickly.

10 CONCLUSION

The objective of this thesis is to overcome the obstacles surrounding the method of characteristics (MOC) on unstructured mesh, to create a scalable, massively parallel, domain decomposed, unstructured mesh, 3D, full-core MOC neutron transport tool: `MOCkingbird`. Previous attempts have been severely limited by poor meshing strategies, excessive memory usage, load imbalances, and poor parallelization that this work overcame. Many unique capabilities were developed and cohesively bonded together to ultimately allow for solution of the full-core heterogeneous 3D BEAVRS benchmark. The primary developments that led to this new capability were:

- **Reactor Mesh Generation**: A capability was developed for generation of volume-preserving pin-cell finite-element meshes. In addition, a directed-acyclic-graph-based mesh generation system was added to `MOOSE`, allowing for efficient creation of heterogeneous 3D full-core reactor meshes.

- **Sparse Communication Algorithm**: A new scalable, MPI-based, sparse communication algorithm was developed: Modified Non-blocking Exchange. This algorithm is used by `MOCkingbird` to accelerate the communication of cyclic tracks and track linking metadata during the setup phase.

- **Scalable Track Generation and Spatial Claiming**: A scalable algorithm for efficiently generating tracks in parallel and having MPI ranks uniquely claim starting positions was developed. While track claiming is somewhat specific to MOC codes, the algorithm developed as part of this work provides a general claiming approach that can be applied to many other applications.

- **Scalable Ray Tracing**: The Scalable Massively Asynchronous Ray Tracing (`SMART`) algorithm was developed to trace MOC tracks across domain decomposed unstructured meshes efficiently. Element traversal algorithms were developed, allowing for ray-tracing through unstructured with extensive handling of corner cases. These were used within a completely asynchronous communication scheme to allow for overlap of communication and computation to achieve excellent scalability. The `SMART` system provides a pluggable, object-oriented system which allows for many different physics to be solved using this new capability.

- **Weighted Partitioning**: Load balance is critical for parallel scalability and was achieved using a new weighted partitioning scheme. The scheme uses the surface area of elements to guide a parallel mesh partitioning package to create de-

compositions that more evenly distribute work, favor on-node communication, and minimize global communication.

- **MOCkingbird** To bring these capabilities together, a new reactor physics tool was created. MOCkingbird enables parallel neutron transport in both 2D and 3D using MOC. This tool was shown to be highly scalable, with testing from 32 to over 18k cores.

- **Full-Core BEAVRS**: A heterogeneous full-core solution to the BEAVRS benchmark was computed using 12k cores and nearly 400k core-hours. This computation was coarse in both space and angle, yet it demonstrated that MOCkingbird is capable of 3D full-core simulations.

10.1 SUMMARY OF WORK

Many technical obstacles were overcome during the development of MOCkingbird to achieve 3D full-core computations, leading to several new capabilities and algorithms. This new tool is capable of integration into high-fidelity multiphysics solutions, leading to higher-fidelity modeling and simulation for both existing and next-generation reactors.

The development of MOCkingbird was possible by building on existing software frameworks and capabilities. These include PETSc, libMesh, MOOSE, OpenMOC, and MPI. The parallel vector storage needed for scalar flux values, neutron sources, and cross section data was provided by PETSc. libMesh supplied the foundational unstructured mesh capability, including the ability to read and write mesh files and perform geometric searches within them. The MOOSE multiphysics framework formed the backbone of MOCkingbird, providing the pluggable, object-oriented architecture and many utilities needed for creation of the new code. MOCkingbird also relies on the track generation capability of OpenMOC track generation capability. All parallelism within MOCkingbird is handled using the message passing interface (MPI) with all of the work for this thesis depending on the MVAPICH open-source implementation. These are all open-source libraries, developed by other scientists and engineers which played a critical role in the creation of MOCkingbird. In turn, many of the new capabilities developed by this thesis are now available, with the hope of enabling further contributions.

A neutron transport tool requires a geometrical description of the reactor. For MOCkingbird, the geometrical descriptions are based on unstructured mesh which has seen much broader use in other communities and can facilitate coupling with other physics packages. Therefore, the first technical achievements were advancements in unstructured mesh generation, specifically for nuclear reactor geometries. A scheme was developed, using the Cubit meshing tool, which created volume-preserving sym-

metric pin-cell meshes. In addition, a new capability for graph-based mesh generation was added to MOOSE. Ultimately, a 1.4 billion element mesh was generated and utilized in the solution of the BEAVRS benchmark, by far, the largest mesh ever used with the MOOSE framework.

A new sparse parallel communication method was developed to allow for the efficient transmission of quantities needed during the startup and setup phases of the solve. The MNBX algorithm proved to be reliable and efficient, allowing MOCkingbird to have scalable track generation capabilities for 3D full-core simulation.

Cyclic tracking had not been used with unstructured mesh MOC before, due to limitations in identifying starting positions of each of the tracks within the partitioned mesh. MOCkingbird utilizes the 2D and 3D cyclic track generation capability developed as part of OpenMOC, and a new parallel algorithm was developed to identify the unique MPI rank claiming the starting position of each track. This new algorithm was shown to be scalable by testing with a 3D quarter-core model of the BEAVRS benchmark using as many as 12k cores.

One of the primary new contributions from this thesis is a new capability for massively parallel ray-tracing: the Scalable Massively-Asynchronous Ray-Tracing (SMART) algorithm. SMART utilizes non-blocking MPI methods, together with a unique, distributed, asynchronous stopping criteria, to allow for efficient parallel ray-tracing through domain decomposed mesh. By starting asynchronous communication for both sending and receiving in-between tracing chunks of rays, SMART can achieve a substantial overlap in communication and computation, leading to greater parallel efficiency. Scalability tests were performed, showing excellent scalability to over 18k cores.

A new weighted partitioning capability was developed, providing scalability for problems with disparately sized elements. Identifying that the workload of an individual element within the mesh is proportional to the surface area of the element allowed for better load balance to be achieved. This scheme was tested, and shown to provide MOCkingbird with scalability for problems with widely varying element sizes.

Two other algorithms were tested for their relative performance to SMART: a Bulk Synchronous (BS) algorithm and the Hybrid Adaptive Ray-Moment Method (HARM^2). Results showed that the SMART algorithm is up to 7x faster than BS and 3x faster than HARM^2 and it is well suited to the task of domain-decomposed, unstructured mesh ray tracing.

All of these new capabilities were developed to overcome the many challenges identified in the literature as impediments to unstructured mesh-based MOC solvers. Ultimately, the combination of these capabilities culminated in the creation of a scalable, massively parallel, unstructured mesh, 3D full-core MOC solver: MOCkingbird. Testing showed that, for problems with many energy groups, MOCkingbird could achieve se-

rial speeds of 10 ns/integration in 2D and 20 ns/integration in 3D. These numbers are in line with results others have published [3, 101, 104]. A series of scalability tests were performed, which showed that the solver could scale from tens of cores to over 18k on supercomputers. This is a critical feature that enables the solution of 3D, full-core neutron transport.

Finally, a first-of-a-kind, unstructured mesh, 3D MOC simulation was performed of the 3D BEAVRS benchmark. The benchmark represents a formidable challenge for high-fidelity 3D neutron transport. A mesh containing 1.4 billion elements was used for over 33 hours on 12k cores to perform 800 iterations, converging the fission source to $2e-4$.

The case solved here is more coarse than the optimal parameters identified in [3]. To meet those parameters the number of axial layers would need to be tripled, azimuthal angles doubled, azimuthal spacing halved, polar angles doubled and polar spacing halved. However, the gains to be made in adding acceleration and other optimizations detailed in the future work section below should is expected to enable a fully-resolved calculation to use approximately 230k core hours on Lemhi. This would make a fully-resolved-calculation possible in 24 hours using 10k cores.

Solving such a massive problem with `MOCkingbird` was only possible through the developments in this thesis: mesh generation, sparse communication, massively parallel ray-tracing, scalable cyclic track generation and claiming, surface-area-weighted load balancing and ultimately the development of `MOCkingbird` itself. These developments have overcome all of the known issues from the literature to show that a massively parallel, unstructured mesh, 3D full-core reactor simulation tool based on MOC is not only possible but practical, on modern supercomputers.

10.2 FUTURE WORK

As shown in this thesis, `MOCkingbird` is already a useful tool and has demonstrated of unstructured mesh MOC is viable. However, there are still many improvements needed in order to become a useful reactor simulator tool.

10.2.1 *Acceleration*

Currently, the largest limitation of `MOCkingbird` is the lack of acceleration methods. Properly implemented acceleration schemes have the ability to bring the number of transport sweeps down from thousands to a few tens of iterations. This is typically accomplished by solving a lower-order system, such as neutron diffusion, on a coarser mesh, that is fed information from the transport sweep. Adding acceleration is critical before `MOCkingbird` can be useful for any production capability.

If an acceleration scheme were added, the 3D, full-core BEAVRS problem could go from needing 800 iterations to only 20 (or less). As mentioned previously, this, plus a few other optimizations, should allow a fully-resolved, 3D full-core calculation to use an estimated 230k core hours.

Many different types of acceleration have been used with MOC including: CMFD [65], DSA [123, 124] and NDA [125]. Research is needed to find which acceleration method fits best within the computational framework and mesh partitioning of MOCkingbird. One important facet of acceleration with MOCkingbird is that the full, parallel non-linear partial differential equation solver of MOOSE is readily available. Therefore, it might be wise to find an acceleration method that can be easily solved using MOOSE.

10.2.2 *Saving Segments*

Currently, MOCkingbird does all ray-tracing "on-the-fly." This means nothing is saved between iterations and the full set of ray-tracing routines must be rerun each time. However, it is technically possible to save the results of the ray-tracing during the first iteration and then, simply playback the sequence of segments for each subsequent iteration. Drawbacks to this approach are that it uses more memory and any benefit may be negated in multiphysics simulations with thermal expansion (which would require new ray-tracing).

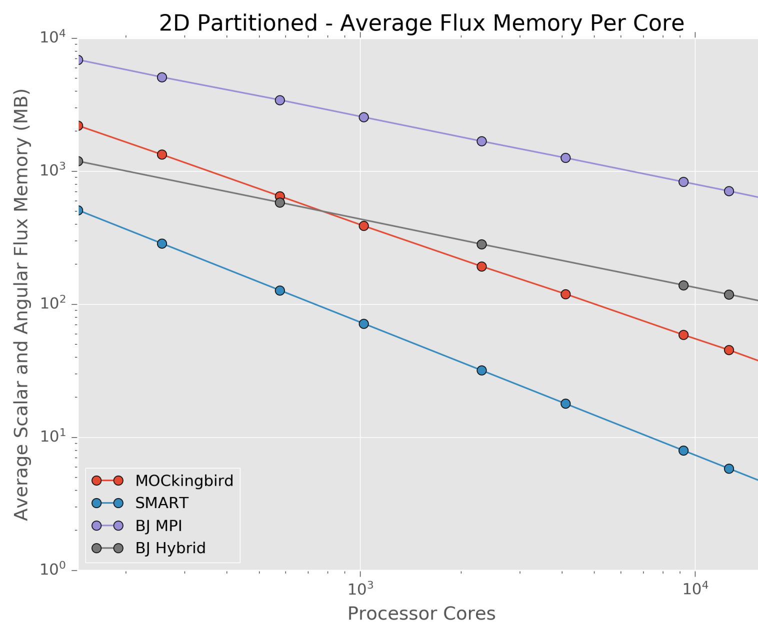


Figure 10.1. Average memory usage per-core for MOCkingbird, compared against the theoretical limit for SMART, a pure MPI block-Jacobi implementation and a hybrid parallel block-Jacobi implementation.

However, in spite of these drawbacks, it is worth pursuing. As shown in Figure 10.1, MOCkingbird does not use much memory and is very memory scalable. Therefore, if the memory is available, it might make sense to use it to speed up the transport sweeps. This is only possible due to the parallel efficiency of SMART. When the project began, it was assumed that saving segments would use too much memory. However, because problems can be extremely decomposed and still maintain performance, it frees up memory for other uses such as saving segments.

Figure 10.1 shows that as, even this large benchmark problem, is decomposed, the memory usage per process quickly drops under 1GB and continues down to under 100MB by the time it reaches 10k cores. Each core on the Lemhi cluster has access to 4GB of memory; therefore, a lot is available for storage of segments. In addition, the amount of geometry on each MPI rank is also going down, meaning that the number of segments that would need to be stored would also be decreasing. Figure 9.14c showed that 10% of solve time was used by ray-tracing in 2D with Figure 9.21b showing that increasing to 25% for 3D. Therefore, the time reduction from saving segments would be within a few tens of percent. All of these things combine to make saving segments possible and ripe for future research.

10.2.3 *Vectorization*

Modern processors include what are known as SIMD units: Single Instruction Multiple Data, also often referred to as "vector" units. These parts of the processor have the ability to perform the same computation on many entries in a vector simultaneously. Intel and AMD have been consistently increasing the amount of vector instruction capability within their CPUs. These instructions have gone through many iterations: MMX, SSE, SSE2, AVX, AVX2, and now AVX-512. The newest, AVX-512, allows for operating on 8 double-precision floating-point number or 16 single-precision floating-point numbers with one instruction in one clock cycle. This thesis did not use vectorization. This was due to some early trials using vectorization that showed only small improvements. However, recently published results for adding vectorization to a MOC code [104], show speedups between 2-5x. Therefore, this topic should be revisited.

10.2.4 *Hybrid Parallel*

All of the results and analysis in this thesis were performed purely using MPI. However, MOCkingbird already contains the ability to use a hybrid parallel model of threading + MPI. This allows for on-node parallelism using threading and MPI only for off-node communication. This could, theoretically, provide speedup. However, to date, the threading capability within MOCkingbird has always been shown to be slower than

running with pure MPI. The reasons for this are varied, but one of them is that MPI also uses shared memory semantics when communicating with MPI ranks on the same node. Therefore, some of the gains sought by using threading (to reduce message passing overhead) don't really exist, due to MPI already being extremely efficient at on-node communication. Other complications such as non-uniform memory access (NUMA), false sharing, and synchronized memory allocation can all play a roll in slowing down threading. More research needs to be done to either speed up threading within `MOCkingbird` or understand why pure MPI is faster for this application.

10.2.5 *Linear Source*

As described in §2, `MOCkingbird` currently utilizes a flat source representation. This means that the source, is constant over an element. This impacts accuracy and requires the use of small elements to capture the continuously varying source accurately. A linear source [10] formulation allows for the source to vary linearly within an element, raising accuracy and allowing for larger elements to be used. This has been shown to be particularly useful at reducing the number of axial layers needed for accurate LWR simulation [3]. Linear source requires more work per segment and therefore larger ns/integration, but this can be offset by using coarser spatial mesh.

10.2.6 *TRRM*

A new MOC-like algorithm has been developed called the Tramm Random Ray Method (TRRM) [8]. Instead of fixed tracks like those used in this thesis, TRRM utilizes new random tracks for each iteration and builds the solution out of an average of many iterations. This has been shown to provide a large speedup over MOC for some problems [101]. `MOCkingbird` already contains the ability to use TRRM; however, more work needs to be done to assess accuracy and scalability before this method can be more widely used.

10.2.7 *Multiphysics*

`MOCkingbird` represents the first 3D full-core, MOC tool capable of directly handling thermal expansion or general deformation of fuel. On-the-fly ray-tracing, coupled with unstructured mesh, allows thermal expansion to be naturally incorporated. `MOCkingbird` being built using `MOOSE` provides simplified coupling to other physics. Several studies need to be performed using `MOCkingbird` with other `MOOSE`-based physics solvers to assess the viability of this pairing for predictive simulation of reactors.

Part I

Appendix

Listing 11.1. `MOCkingbird` input file for a fully-reflective pin-cell calculation

```
[Mesh]
  type = FileMesh
  file = pin_cell_fine.e
  dim = 2
[]

[UserObjects]
  [track_reader]
    type = TrackGenerator2D
    execute_on = initial
    num_azim = 32
    azim_spacing = 0.01
  []
  [study]
    type = TrackListStudy
    send_buffer_size = 100
    track_reader = track_reader
  []
[]

[RayKernels]
  [flat_flux]
    type = FlatSource
  []
[]

[RayBCs]
  [bottom]
    boundary = 3
    type = MOCReflectAndBankBC
    normal = '0 -1 0'
  []
  [right]
    boundary = 4
    type = MOCReflectAndBankBC
    normal = '1 0 0'
  []
  [top]
    boundary = 5
    type = MOCReflectAndBankBC
    normal = '0 1 0'
  []
  [left]
    boundary = 6
    type = MOCReflectAndBankBC
    normal = '-1 0 0'
  []
[]

[RayMaterials]
  [fuel]
    type = Fuel
    block = fuel
  []
  [moderator]
    type = Moderator
    block = moderator
  []
[]

[Problem]
  type = MOCProblem
  deterministic_study = study
  solve = false
  num_groups = 8
  num_polar = 3
  power_iterations = 2000
  fission_source_tolerance = 1e-4
  k_tolerance = 1e-5
  use_tracked_volume = true
[]

[Executioner]
  type = Steady
[]

[Outputs]
  exodus = true
[]
```

12 C5G7 INPUT

Listing 12.1. MOOSE input file syntax utilizing the graph-based mesh-generation system to create the mesh for the 2D C5G7 benchmark.

```
[MeshGenerators]
   #inactive = 'top_part change_block_id3 translate stitch3'
  [./uo2]
    type = FileMeshGenerator
    file = ../pin_cells/uo2_coarse.e
  []
  [./43_mox]
    type = FileMeshGenerator
    file = ../pin_cells/43_mox_coarse.e
  []
  [./70_mox]
    type = FileMeshGenerator
    file = ../pin_cells/70_mox_coarse.e
  []
  [./87_mox]
    type = FileMeshGenerator
    file = ../pin_cells/87_mox_coarse.e
  []
  [./gt]
    type = FileMeshGenerator
    file = ../pin_cells/gt_coarse.e
  []
  [./fc]
    type = FileMeshGenerator
    file = ../pin_cells/fc_coarse.e
  []
  [./mod]
    type = FileMeshGenerator
    file = ../pin_cells/mod.e
  []
#---------------------------------------------------------
  [./mox_assembly]
    type = PatternedMeshGenerator
    inputs = 'uo2 43_mox 70_mox 87_mox gt fc mod'
    pattern = '1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ;
               1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 ;
```

```
              1 2 2 2 2 4 2 2 4 2 2 4 2 2 2 2 1 ;
              1 2 2 4 2 3 3 3 3 3 3 3 2 4 2 2 1 ;
              1 2 2 2 3 3 3 3 3 3 3 3 3 2 2 2 1 ;
              1 2 4 3 3 4 3 3 4 3 3 4 3 3 4 2 1 ;
              1 2 2 3 3 3 3 3 3 3 3 3 3 3 2 2 1 ;
              1 2 2 3 3 3 3 3 3 3 3 3 3 3 2 2 1 ;
              1 2 4 3 3 4 3 3 5 3 3 4 3 3 4 2 1 ;
              1 2 2 3 3 3 3 3 3 3 3 3 3 3 2 2 1 ;
              1 2 2 3 3 3 3 3 3 3 3 3 3 3 2 2 1 ;
              1 2 4 3 3 4 3 3 4 3 3 4 3 3 4 2 1 ;
              1 2 2 2 3 3 3 3 3 3 3 3 3 2 2 2 1 ;
              1 2 2 4 2 3 3 3 3 3 3 3 2 4 2 2 1 ;
              1 2 2 2 2 4 2 2 4 2 2 4 2 2 2 2 1 ;
              1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 ;
              1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1'
    y_width = 1.26
    x_width = 1.26
    bottom_boundary_id = 1
    right_boundary_id = 2
    top_boundary_id = 3
    left_boundary_id = 4
  []
#---------------------------------------------------------------
  [./uo2_assembly]
    type = PatternedMeshGenerator
    inputs = 'uo2 43_mox 70_mox 87_mox gt fc mod'
    pattern = '0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ;
               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ;
               0 0 0 0 0 4 0 0 4 0 0 4 0 0 0 0 0 ;
               0 0 0 4 0 0 0 0 0 0 0 0 0 4 0 0 0 ;
               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ;
               0 0 4 0 0 4 0 0 4 0 0 4 0 0 4 0 0 ;
               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ;
               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ;
               0 0 4 0 0 4 0 0 5 0 0 4 0 0 4 0 0 ;
               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ;
               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ;
               0 0 4 0 0 4 0 0 4 0 0 4 0 0 4 0 0 ;
               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ;
               0 0 0 4 0 0 0 0 0 0 0 0 0 4 0 0 0 ;
               0 0 0 0 0 4 0 0 4 0 0 4 0 0 0 0 0 ;
               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ;
               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0'
    y_width = 1.26
```

```
    x_width = 1.26
    bottom_boundary_id = 1
    right_boundary_id = 2
    top_boundary_id = 3
    left_boundary_id = 4
  []
#--------------------------------------------------------------
  [./assembly]
    type = PatternedMeshGenerator
    inputs = 'uo2_assembly mox_assembly'
    pattern = '0 1;
               1 0'
    y_width = 21.42
    x_width = 21.42
    bottom_boundary_id = 1
    right_boundary_id = 2
    top_boundary_id = 3
    left_boundary_id = 4
  []
  [./new_ids]
    type = RenameBoundaryGenerator
    input = assembly
    old_boundary_id = '1 2 3 4'
    new_boundary_id = '0 1 2 3'
  []
#-------------------------------------------------------------
  [./bottom]
    type = GeneratedMeshGenerator
    dim = 2
    nx = 204
    xmin = -0.63
    xmax = 42.21
    ny = 102
    ymin = -63.63
    ymax = -42.21
    elem_type = QUAD4
  []
  [./change_block_id]
    type = SubdomainIDGenerator
    input = bottom
    subdomain_id = 7
  []
#-----------------------------------------------------------
  [./right_part]
```

```
    type = GeneratedMeshGenerator
    dim = 2
    nx = 102
    xmin = 42.21
    xmax = 63.63
    ny = 306
    ymin = -63.63
    ymax = 0.63
    elem_type = QUAD4
  []
  [./change_block_id2]
    type = SubdomainIDGenerator
    input = right_part
    subdomain_id = 7
  []
#-----------------------------------------------------------
  [./stitch2]
    type = StitchedMeshGenerator
    inputs = 'new_ids change_block_id change_block_id2'
    stitch_boundaries_pairs = '0 2;
                               1 3'
  []
  [./boundary_names]
    type = RenameBoundaryGenerator
    input = stitch2
    old_boundary_id = '0 1 2 3'
    new_boundary_name = 'bottom right top left'
  []
[]
```

13 ENERGY GROUP STRUCTURE FOR BEAVRS CROSS SECTIONS

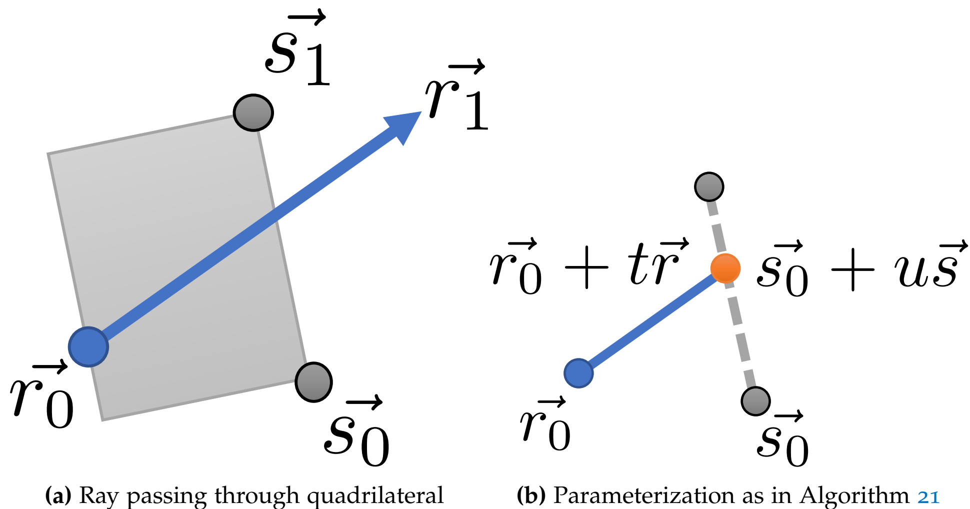
This 70 group energy structure comes from CASMO-4 [6] and was utilized for the BEAVRS benchmark.

Group No.	Lower Bound [MeV]	Upper Bound [MeV]
70	0.0000E+00	5.0000E-09
69	5.0000E-09	1.0000E-08
68	1.0000E-08	1.5000E-08
67	1.5000E-08	2.0000E-08
66	2.0000E-08	2.5000E-08
65	2.5000E-08	3.0000E-08
64	3.0000E-08	3.5000E-08
63	3.5000E-08	4.2000E-08
62	4.2000E-08	5.0000E-08
61	5.0000E-08	5.8000E-08
60	5.8000E-08	6.7000E-08
59	6.7000E-08	8.0000E-08
58	8.0000E-08	1.0000E-07
57	1.0000E-07	1.4000E-07
56	1.4000E-07	1.8000E-07
55	1.8000E-07	2.2000E-07
54	2.2000E-07	2.5000E-07
53	2.5000E-07	2.8000E-07
52	2.8000E-07	3.0000E-07
51	3.0000E-07	3.2000E-07
50	3.2000E-07	3.5000E-07
49	3.5000E-07	4.0000E-07
48	4.0000E-07	5.0000E-07
47	5.0000E-07	6.2500E-07
46	6.2500E-07	7.8000E-07
45	7.8000E-07	8.5000E-07

44	8.5000E-07	9.1000E-07
43	9.1000E-07	9.5000E-07
42	9.5000E-07	9.7200E-07
41	9.7200E-07	9.9600E-07
40	9.9600E-07	1.0200E-06
39	1.0200E-06	1.0450E-06
38	1.0450E-06	1.0710E-06
37	1.0710E-06	1.0970E-06
36	1.0970E-06	1.1230E-06
35	1.1230E-06	1.1500E-06
34	1.1500E-06	1.3000E-06
33	1.3000E-06	1.5000E-06
32	1.5000E-06	1.8550E-06
31	1.8550E-06	2.1000E-06
30	2.1000E-06	2.6000E-06
29	2.6000E-06	3.3000E-06
28	3.3000E-06	4.0000E-06
27	4.0000E-06	9.8770E-06
26	9.8770E-06	1.5968E-05
25	1.5968E-05	2.7700E-05
24	2.7700E-05	4.8052E-05
23	4.8052E-05	7.5501E-05
22	7.5501E-05	1.4873E-04
21	1.4873E-04	3.6726E-04
20	3.6726E-04	9.0690E-04
19	9.0690E-04	1.4251E-03
18	1.4251E-03	2.2395E-03
17	2.2395E-03	3.5191E-03
16	3.5191E-03	5.5300E-03
15	5.5300E-03	9.1180E-03
14	9.1180E-03	1.5030E-02
13	1.5030E-02	2.4780E-02
12	2.4780E-02	4.0850E-02

11	4.0850E-02	6.7340E-02
10	6.7340E-02	1.1100E-01
9	1.1100E-01	1.8300E-01
8	1.8300E-01	3.0250E-01
7	3.0250E-01	5.0000E-01
6	5.0000E-01	8.2100E-01
5	8.2100E-01	1.3530E+00
4	1.3530E+00	2.2310E+00
3	2.2310E+00	3.6790E+00
2	3.6790E+00	6.0655E+00
1	6.0655E+00	2.0000E+01

At the heart of the ray-traversal Algorithm 7 is the ability to find intersections between the rays and the unstructured mesh geometry. This part of the algorithm is the only part specialized for particular element types: quadrilaterals for 2D and hexahedrals in 3D. The rest of 7 is completely dimension agnostic with the code working the same way regardless of element type or dimension.



(a) Ray passing through quadrilateral **(b)** Parameterization as in Algorithm 21

Figure 14.1. Visual representation of Algorithm 21

To find intersections of faces (edges) in 2D the code employs Algorithm 21. As shown in Figure 14.1 the intersection point is found as the distance along the side starting from one node. Within `MOCkingbird` each side of the current element will be checked with the longest distance found being chosen as the correct intersection.

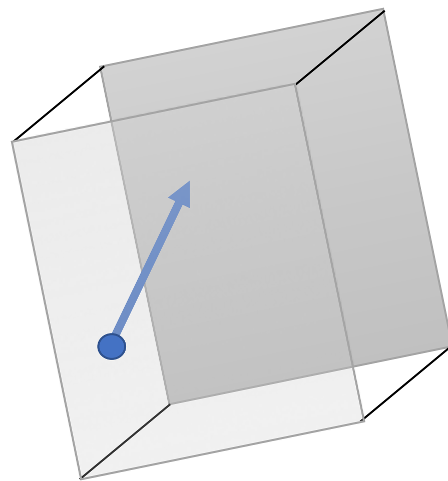
In three-dimensions, intersections with the quadrilateral sides of hexahedral elements must be performed. Several algorithms were tried for this work with the best (most efficient and most accurate) being to split the quadrilateral face into two triangles and use a back-face culling algorithm from [70] to test each one in turn (with a quick return if the first triangle yields an intersection).

Algorithm 21: Line-line intersection algorithm utilized for finding side intersections within a two-dimensional domain. Based on the algorithm found at [126]

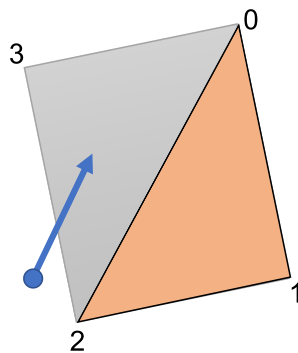
input : Beginning and ending of ray ($\boldsymbol{r_0}$, $\boldsymbol{r_1}$)
input : Beginning and ending of side to test ($\boldsymbol{s_0}$, $\boldsymbol{s_1}$)
output: Whether or not the lines intersect (*true*, *false*)
output: Intersection point ($\boldsymbol{r_0} + t\boldsymbol{r}$)

1 *Using* $\boldsymbol{a} \times \boldsymbol{b}$ *as* $a_x b_y - a_y b_x$

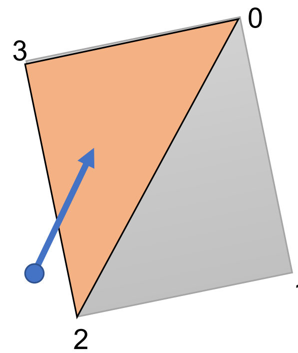
2 $\boldsymbol{r} \leftarrow \boldsymbol{r_1} - \boldsymbol{r_0}$
3 $\boldsymbol{s} \leftarrow \boldsymbol{s_1} - \boldsymbol{s_0}$
4 $w \leftarrow \boldsymbol{r} \times \boldsymbol{s}$
5 $\boldsymbol{h} = s_0 - r_0$
6 **if** $|w| \approx 0$ **then**
7 | return *false*
8 **end**
9 $t = h \times \boldsymbol{s}/w$
10 $u = h \times \boldsymbol{r}/w$
11 **if** $0 \lesssim t \lesssim 1$ *and* $0 \lesssim u \lesssim 1$ **then**
12 | return *true* and $\boldsymbol{r_0} + (t\boldsymbol{r})$
13 **end**
14 return *false*



(a) Element and ray.



(b) First triangle tested.



(c) Second triangle tested.

Figure 14.2. Ray tracing across a hexahedral element from the front face to the back face. The back face will be made into two triangles that are each tested for intersections using Algorithm 23. The vertex numbering is also shown.

Figure 14.2 pictorializes Algorithm 23. In Figure 14.2a the ray is shown as coming through the front face of the element, crossing the element and striking the back face. Figure 14.2b breaks out the back face, showing the vertex numbering and the first triangle that will be tested. The vertex numbering is such that using the right-hand-rule the normal vector of the face is facing outward from the element (in the same direction of the ray). Because of this, care must be taken to feed the vertices into the routines in Algorithm 23 to ensure that, with respect to the ray, the algorithm believes that the normal of the face is towards the ray. To create each of the triangles seen in

Figures 14.2b, 14.2c two sets of vertices are passed into the `rayIntersectsTriangle()` routine found in Algorithm 23.

Algorithm 22: Intersection algorithm used for quadrilateral faces of hexahedral elements.

input : Beginning of and unit normal direction ($\boldsymbol{r_0}$, $\boldsymbol{r}$)
input : Three vertices of the triangle ($\boldsymbol{v_0}$, $\boldsymbol{v_1}$, $\boldsymbol{v_2}$)
output: Whether or not the ray intersects the triangle (*true*, *false*)
output: Intersection point ($\boldsymbol{r_0} + t\boldsymbol{r}$)

1 EPSILON $\leftarrow$ 1e-10
2 $\boldsymbol{e_1} \leftarrow v_1$ - v_0
3 $\boldsymbol{e_2} \leftarrow v_2$ - v_0
4 $\boldsymbol{p} \leftarrow \boldsymbol{r} \times \boldsymbol{e_2}$
5 det $\leftarrow \boldsymbol{e_1} \cdot \boldsymbol{p}$
6 **if** *det* < -*EPSILON* **then**
7 │ return *false*
8 **end**
9 $\boldsymbol{t} \leftarrow r_0 - v_0$
10 u = $\boldsymbol{t} \cdot \boldsymbol{p}$
11 **if** *u* < -*EPSILON* | | *u* > *det* + *EPSILON* **then**
12 │ return *false*
13 **end**
14 $\boldsymbol{q} \leftarrow \boldsymbol{t} \times \boldsymbol{e_1}$
15 v $\leftarrow \boldsymbol{r} \cdot \boldsymbol{q}$
16 **if** *v* < -*EPSILON* | | *u* + *v* > *det* + *EPSILON* **then**
17 │ return *false*
18 **end**
19 t $l \leftarrow \boldsymbol{e_2} \cdot \boldsymbol{q}$
20 inv_det $\leftarrow \frac{1.}{det}$
21 t $*=$ inv_det
22 u $*=$ inv_det
23 v $*=$ inv_det
24 **if** *t* <= *0* **then**
25 │ return *false*
26 **end**
27 return *true*

Algorithm 23: Intersection algorithm used for quadrilateral faces of hexahedral elements.

input : Beginning of and unit normal direction ($\boldsymbol{r}_0$, $\boldsymbol{r}$)
input : Three vertices of the quadrilateral ($\boldsymbol{v}_0$, $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, $\boldsymbol{v}_3$)
output: Whether or not the ray intersects the triangle (*true*, *false*)
output: Intersection point ($\boldsymbol{r}_0 + t\boldsymbol{r}$)

1 **if** *Call Algorithm 22 with* ($\boldsymbol{r}_0$, $\boldsymbol{r}$, $\boldsymbol{v}_0$, $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, u, v, t) **then**
2 | return *true*
3 **end**
4 **if** *Call Algorithm 22 with* ($\boldsymbol{r}_0$, $\boldsymbol{r}$, $\boldsymbol{v}_3$, $\boldsymbol{v}_0$, $\boldsymbol{v}_2$, u, v, t) **then**
5 | return *true*
6 **end**
7 return *false*

15 OBJECTPOOL

An object pool is used by SMART to reduce the amount of memory allocation and deallocation during execution. The object pool code is shown in Listing 15.1. The acquire() method is called to retrieve an object from the pool. It returns a smart pointer, which, when the object is no longer needed, automatically returns the object to the pool. If the pool is empty when acquire is called, then a new object is constructed.

Listing 15.1. Object pool utilized by SMART

```
#ifndef OBJECTPOOL_H
#define OBJECTPOOL_H

// System Includes
#include <stack>
#include <memory>

template <class T, typename... Args>
auto
reset(int, T & obj, Args... args) -> decltype(obj.reset(args...), void())
{
  obj.reset(std::forward<Args>(args)...);
}

template <class T, typename... Args>
void
reset(double, T & /*obj*/, Args... /*args*/)
{
}

/**
 *
 * Originally From https://stackoverflow.com/a/27837534/2042320
 *
 * with added variadic templated perfect forwarding to acquire()
 *
 * For an object to be resetable it needs to define a reset() function
 * that takes the same arguments as its constructor.
 */
template <class T>
class ObjectPool
{
private:
  struct ExternalDeleter
  {
    explicit ExternalDeleter(std::weak_ptr<ObjectPool<T> *> pool) : _pool(pool) {}

    void operator()(T * ptr)
    {
      if (auto _poolptr = _pool.lock())
      {
        try
        {
          (*_poolptr.get())->add(std::unique_ptr<T>{ptr});
          return;
        }
        catch (...) {}
      }
      std::default_delete<T>{}(ptr);
    }

  private:
    std::weak_ptr<ObjectPool<T> *> _pool;
  };
```

```cpp
public:
  using ptr_type = std::unique_ptr<T, ExternalDeleter>;

  ObjectPool() : _this_ptr(new ObjectPool<T>*(this)) {}
  virtual ~ObjectPool() {}

  void add(std::unique_ptr<T> t) { _pool.push(std::move(t)); }

  template <typename... Args>
  ptr_type acquire(Args &&... args)
  {
    // if the pool is empty - create one
    if (_pool.empty())
    {
      _num_created++;
      return std::move(ptr_type(new T(std::forward<Args>(args)...),
                              ExternalDeleter{std::weak_ptr<ObjectPool<T>*>{_this_ptr}}));
    }
    else
    {
      ptr_type tmp(_pool.top().release(),
                   ExternalDeleter{std::weak_ptr<ObjectPool<T>*>{_this_ptr}});
      _pool.pop();

      reset(1, *tmp, std::forward<Args>(args)...);

      return std::move(tmp);
    }
  }

  bool empty() const { return _pool.empty(); }
  size_t size() const { return _pool.size(); }
  size_t num_created() const { return _num_created; }

private:
  std::shared_ptr<ObjectPool<T>*> _this_ptr;
  std::stack<std::unique_ptr<T>> _pool;

  size_t _num_created = 0;
};
#endif
```

BIBLIOGRAPHY

[1] JR Askew. A characteristics formulation of the neutron transport equation in complicated geometries. Technical report, United Kingdom Atomic Energy Authority, 1972.

[2] William Boyd, Samuel Shaner, Lulu Li, Benoit Forget, and Kord Smith. The OpenMOC method of characteristics neutral particle transport code. *Annals of Nuclear Energy*, 68:43–52, 2014.

[3] Geoffrey Gunow. *Full core 3D neutron transport simulation using the method of characteristics with linear sources*. PhD thesis, Massachusetts Institute of Technology, 2018.

[4] William Boyd. *Reactor agnostic multi-group cross section generation for fine-mesh deterministic neutron transport simulations*. PhD thesis, Massachusetts Institute of Technology, 2017.

[5] Brendan Matthew Kochunas. *A Hybrid Parallel Algorithm for the 3-D Method of Characteristics Solution of the Boltzmann Transport Equation on High Performance Compute Clusters.* PhD thesis, University of Michigan, 2013.

[6] KS Smith. Casmo-4 characteristics methods for two-dimensional pwr and bwr core calculations. *Trans. Am. Nucl. Soc.*, 83:322, 2000.

[7] Dan Gabriel Cacuci. *Handbook of Nuclear Engineering: Vol. 1: Nuclear Engineering Fundamentals; Vol. 2: Reactor Design; Vol. 3: Reactor Analysis; Vol. 4: Reactors of Generations III and IV; Vol. 5: Fuel Cycles, Decommissioning, Waste Disposal and Safeguards*, volume 1. Springer Science & Business Media, 2010.

[8] John R Tramm, Kord S Smith, Benoit Forget, and Andrew R Siegel. The random ray method for neutral particle transport. *Journal of Computational Physics*, 342: 229–252, 2017.

[9] Akio Yamamoto, Masato Tabuchi, Naoki Sugimura, Tadashi Ushio, and Masaaki Mori. Derivation of optimum polar angle quadrature set for the method of characteristics based on approximation error for the bickley function. *Journal of Nuclear Science and Technology*, 44(2):129–136, 2007.

[10] Rodolfo M Ferrer and Joel D Rhodes III. A linear source approximation scheme for the method of characteristics. *Nuclear Science and Engineering*, 182(2):151–165, 2016.

[11] Joel Rhodes, Kord Smith, and Deokjung Lee. Casmo-5 development and applications. In *Proceedings of the international conference on physics of reactors (PHYSOR2006)*, 2006.

[12] Samuel Shaner et al. Theoretical analysis of track generation in 3D method of characteristics. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method, Nashville, Tennessee, 2015.

[13] Ben Lindley, Glynn Hosking, Peter Smith, David Powney, Brendan Tollit, Tim Fry, Ray Perry, Tim Ware, Christophe Murphy, Chris Grove, et al. Developments within the wims reactor physics code for whole core calculations. *Proceedings of M&C*, 2017.

[14] *Method of characteristics development targeting the high performance Blue Gene/P computer at Argonne National Laboratory*, 2011.

[15] Geoffrey Gunow, Benoit Forget, and Kord Smith. Stabilization of multi-group neutron transport with transport-corrected cross-sections. *Annals of Nuclear Energy*, 126:211–219, 2019.

[16] Zhaoyuan Liu, Kord Smith, Benoit Forget, and Javier Ortensi. Cumulative migration method for computing rigorous diffusion coefficients and transport cross sections from monte carlo. *Annals of Nuclear Energy*, 112:507–516, 2018.

[17] Xue Yang and Nader Satvat. MOCUM: A two-dimensional method of characteristics code based on constructive solid geometry and unstructured meshing for general geometries. *Annals of Nuclear Energy*, 46:20–28, 2012.

[18] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[19] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1): 46–55, 1998.

[20] S Santandrea, D Sciannandrone, R Sanchez, L Mao, and L Graziano. A neutron transport characteristics method for 3D axially extruded geometries coupled with a fine group self-shielding environment. *Nuclear Science and Engineering*, 186(3):239–276, 2017.

[21] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[22] B. Kelley, et al. CMFD Acceleration of Spatial Domain-decomposition Neutron Transport Problems. In *Proceedings of the international conference on physics of reactors (PHYSOR2012)*, Knoxville, TN, USA, April 2012.

[23] Kalyan Kumaran. Introduction to mira. In *Code for Q Workshop*, 2016.

[24] Steve S Chen, Christopher C Hsiung, John L Larson, and Eugene R Somdahl. Cray xmp: A multiprocessor supercomputer. Technical report, SAE Technical Paper, 1988.

[25] James Laudon and Daniel Lenoski. The sgi origin: a ccNUMA highly scalable server. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 241–251. ACM, 1997.

[26] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. The sgi® altixtm 3000 global shared-memory architecture. *Silicon Graphics, Inc*, 2005.

[27] Thomas L Sterling, John Salmon, Donald J Becker, and Daniel F Savarese. *How to build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT press, 1999.

[28] Thomas Lawrence Sterling, William Gropp, and Ewing Lusk. *Beowulf cluster computing with Linux*. MIT press, 2002.

[29] Christoph Lameter et al. NUMA (non-uniform memory access): An overview. *Acm Queue*, 11(7):40, 2013.

[30] Susan K Coulter and Jesse Edward Martinez. Introduction to infiniband. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2015.

[31] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[32] Youcef Saad and Martin H Schultz. Topological properties of hypercubes. *IEEE Transactions on computers*, 37(7):867–872, 1988.

[33] Dong Chen, Noel A Eisley, Philip Heidelberger, Robert M Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J Parker. The ibm blue gene/q interconnection

network and message unit. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2011.

[34] Richard L Graham, Steve Poole, Pavel Shamis, Gil Bloch, Noam Bloch, Hillel Chapman, Michael Kagan, Ariel Shahar, Ishai Rabinovitz, and Gilad Shainer. Connectx-2 infiniband management queues: First investigation of the new support for network offloaded collective operations. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 53–62. IEEE, 2010.

[35] Jack J Dongarra, Cleve Barry Moler, James R Bunch, and Gilbert W Stewart. *LINPACK users' guide*. SIAM, 1979.

[36] Geoffrey C Fox. *Solving problems on concurrent processors*. Old Tappan, NJ; Prentice Hall Inc., 1988.

[37] MPI Forum. MPI: A message-passing interface standard. version 3.0. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, 2015. Accessed 8/23/2019.

[38] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 97–104. Springer, 2004.

[39] William Gropp. MPICH2: A new start for MPI implementations. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 7–7. Springer, 2002.

[40] Wei Huang, Gopalakrishnan Santhanaraman, H-W Jin, Qi Gao, and Dhabaleswar K Panda. Design of high performance MVAPICH2: MPI2 over infiniband. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, volume 1, pages 43–48. IEEE, 2006.

[41] G Fox, M Johnson, G Lyzenga, S Otto, J Salmon, D Walker, and Richard L White. Solving problems on concurrent processors vol. 1: General techniques and regular problems. *Computers in Physics*, 3(1):83–84, 1989.

[42] Fande Kong, Roy H Stogner, Derek R Gaston, John W Peterson, Cody J Permann, Andrew E Slaughter, and Richard C Martineau. A general-purpose hierarchical mesh partitioning method with node balancing strategies for large-scale numerical simulations. In *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*, pages 65–72. IEEE, 2018.

[43] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. *SC Conference*, 0:35, 1996. doi: 10.1109/SC.1996.32.

[44] George Karypis. METIS and ParMETIS. *Encyclopedia of parallel computing*, pages 1117–1124, 2011.

[45] Cédric Chevalier and François Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel computing*, 34(6-8):318–331, 2008.

[46] Bruce Hendrickson and Robert Leland. The chaco users guide. version 1.0. Technical report, Sandia National Labs., Albuquerque, NM (United States), 1993.

[47] Peter Sanders and Christian Schulz. Kahip v2. 0–karlsruhe high quality partitioning–user guide. *arXiv preprint arXiv:1311.1714*, 2013.

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

[49] Benjamin S Kirk, Roy H Stogner, Paul T Bauman, and Todd A Oliver. Modeling hypersonic entry with the fully-implicit navier–stokes (fin-s) stabilized finite element flow solver. *Computers & Fluids*, 92:281–292, 2014.

[50] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, W Gropp, et al. Petsc users manual. Technical report, Argonne National Laboratory, 2019.

[51] Robert D Falgout, Jim E Jones, and Ulrike Meier Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In *Numerical solution of partial differential equations on parallel computers*, pages 267–294. Springer, 2006.

[52] Larry A Schoof and Victor R Yarberry. Exodus ii: a finite element data model. Technical report, Sandia National Labs., Albuquerque, NM (United States), 1994.

[53] Mohammad Reza Abbasifard, Bijan Ghahremani, and Hassan Naderi. A survey on nearest neighbor search methods. *International Journal of Computer Applications*, 95(25), 2014.

[54] Derek Gaston, Chris Newman, Glen Hansen, and Damien Lebrun-Grandie. Moose: A parallel computational framework for coupled systems of nonlinear equations. *Nuclear Engineering and Design*, 239(10):1768–1778, 2009.

[55] Yaqi Wang. Nonlinear diffusion acceleration for the multigroup transport equation discretized with s n and continuous fem with rattlesnake. In *Proceedings of*

the 2013 International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering-M and C 2013, 2013.

[56] RL Williamson, JD Hales, SR Novascone, MR Tonks, DR Gaston, CJ Permann, D Andrs, and RC Martineau. Multidimensional multiphysics simulation of nuclear fuel behavior. *Journal of Nuclear Materials*, 423(1):149–163, 2012.

[57] Robert Podgorney, Hai Huang, and Derek Gaston. Massively parallel fully coupled implicit modeling of coupled thermal-hydrological-mechanical processes for enhanced geothermal system reservoirs. Technical report, Idaho National Laboratory (INL), 2010.

[58] Alexander D Lindsay, David B Graves, and Steven C Shannon. Fully coupled simulation of the plasma liquid interface and interfacial coefficient effects. *Journal of Physics D: Applied Physics*, 49(23):235204, 2016.

[59] John W Peterson, Alexander D Lindsay, and Fande Kong. Overview of the incompressible navier–stokes simulation capabilities in the moose framework. *Advances in Engineering Software*, 119:68–92, 2018.

[60] Derek R Gaston, Cody J Permann, John W Peterson, Andrew E Slaughter, David Andrš, Yaqi Wang, Michael P Short, Danielle M Perez, Michael R Tonks, Javier Ortensi, et al. Physics-based multiscale coupling for full core nuclear reactor simulation. *Annals of Nuclear Energy*, 84:45–54, 2015.

[61] Derek R Gaston, Cody J Permann, John W Peterson, Andrew E Slaughter, David Andrš, Yaqi Wang, Michael P Short, Danielle M Perez, Michael R Tonks, Javier Ortensi, et al. Physics-based multiscale coupling for full core nuclear reactor simulation. *Annals of Nuclear Energy*, 84:45–54, 2015.

[62] Frederick N Gleicher, Richard L Williamson, Javier Ortensi, Yaqi Wang, Benjamin W Spencer, Stephen R Novascone, Jason D Hales, and Richard C Martineau. The coupling of the neutron transport application rattlesnake to the nuclear fuels performance application bison under the moose framework. Technical report, Idaho National Lab.(INL), Idaho Falls, ID (United States), 2014.

[63] Fande Kong, Yaqi Wang, Derek R Gaston, Alexander D Lindsay, Cody J Permann, and Richard C Martineau. A scalable multilevel domain decomposition preconditioner with a subspace-based coarsening algorithm for the neutron transport calculations. *arXiv preprint arXiv:1906.07743*, 2019.

[64] D. R. Gaston, C. J. Permann, D. Andrs, and J. W. Peterson. Massive hybrid parallelism for fully implicit multiphysics. In *Proceedings of the International Conference*

on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013), Sun Valley, Idaho, May 5–9, 2013.

[65] Kord S Smith. Nodal method storage reduction by nonlinear iteration. *Trans. Am. Nucl. Soc.*, 44:265–266, 1983.

[66] William Robert Dawson Boyd III. *Massively parallel algorithms for method of characteristics neutral particle transport on shared memory computer architectures*. PhD thesis, Massachusetts Institute of Technology, 2014.

[67] Lulu Li, Kord Smith, and Benoit Forget. Techniques for stabilizing coarse-mesh finite difference (CMFD) in methods of characteristics (moc). 2015.

[68] Samuel Christopher Shaner. *Transient method of characteristics via the Adiabatic, Theta, and Multigrid Amplitude Function methods*. PhD thesis, Massachusetts Institute of Technology, 2014.

[69] Cristian Rabiti, Micheal A Smith, Yang Wong Sik, Dinesh Kaushik, and Giuseppe Palmiotti. Parallel method of characteristics on unstructured meshes for the UNIC code. In *Proc. International Conference on the Physics of Reactors, Interlaken, Switzerland*, 2008.

[70] Tomas Möller and Ben Trumbore. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, page 7. ACM, 2005.

[71] MA Smith, A Marin-Lafleche, WS Yang, Dinesh Kaushik, and Andrew Siegel. Method of characteristics development targeting the high performance blue Gene/P computer at argonne national laboratory. In *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2011)*, 2011.

[72] A Marin-Lafleche, MA Smith, and C Lee. Proteus-MOC: A 3D deterministic solver incorporating 2D method of characteristics. In *Proceedings of the 2013 International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering-M and C 2013*, 2013.

[73] CH Lee, YS Jung, HM Connaway, and TA Taiwo. Simulation of TREAT cores using high-fidelity neutronics code proteus. *Proc. of M&C 2017*, pages 16–20, 2017.

[74] Tanay Mazumdar and SB Degweker. Solution of the neutron transport equation by the method of characteristics using a linear representation of the source within a mesh. *Annals of Nuclear Energy*, 108:132–150, 2017.

[75] Paul Cosgrove and Eugene Shwageraus. Development of MoCha-Foam: a new method of characteristics neutron transport solver for OpenFOAM. In *International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering, M&C*, 2017.

[76] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[77] CO Frederick, YC Wong, and FW Edge. Two-dimensional automatic mesh generation for structural analysis. *International Journal for Numerical Methods in Engineering*, 2(1):133–144, 1970.

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

[79] Ted D Blacker, William J Bohnhoff, and Tony L Edwards. CUBIT mesh generation environment. volume 1: Users manual. Technical report, Sandia National Labs., Albuquerque, NM (United States), 1994.

[80] ABAQUS Documentation Abaqus. Dassault systemes. *Providence, RI, USA*, 2011.

[81] Ansys Inc. Fluent 12.0 theory guide. 2009.

[82] Swanson Analysis Systems. ANSYS user manual. *Inc., Volumes I, II, III, IV, Revision*, 5(0), 1994.

[83] Zottie. Illustration of csg, 2005. URL `https://commons.wikimedia.org/w/index.php?curid=263170`. Accessed: 11/11/2018.

[84] S Santandrea, D Sciannandrone, R Sanchez, L Mao, and L Graziano. A neutron transport characteristics method for 3D axially extruded geometries coupled with a fine group self-shielding environment. *Nuclear Science and Engineering*, 186(3):239–276, 2017.

[85] EE Lewis, MA Smith, N Tsoulfanidis, G Palmiotti, TA Taiwo, and RN Blomquist. Benchmark specification for deterministic 2-D/3-D MOX fuel assembly transport calculations without spatial homogenization (c5g7 mox). *NEA/NSC*, 2001.

[86] Mark D DeHart, Ian C Gauld, and Mark L Williams. High-fidelity lattice physics capabilities of the SCALE code system using TRITON. In *Joint International Topical Meeting on Mathematics & Computation and Supercomputing in Nuclear Applications, Monterey, California*, page 15, 2007.

[87] Nicholas Horelik, Bryan Herman, Benoit Forget, and Kord Smith. Benchmark for evaluation and validation of reactor simulations (beavrs), v1. 0.1. In *Proc. Int. Conf. Mathematics and Computational Methods Applied to Nuc. Sci. & Eng*, 2013.

[88] Timothy J Tautges and Rajeev Jain. Creating geometry and mesh models for nuclear reactor core geometries using a lattice hierarchy-based approach. *Engineering with Computers*, 28(4):319–329, 2012.

[89] Paul K. Romano and Ben Forget. The OpenMC Monte Carlo particle transport code. *Ann. Nucl. Energy*, 51:274–281, 2013.

[90] Rajeev Jain and Timothy J Tautges. NEAMS MeshKit: Nuclear reactor mesh generation solutions. In *Proceedings of International Congress on the Advances in Nuclear Power Plants (ICAPP)*, pages 6–9, 2014.

[91] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Section 22.4: topological sort. *Introduction to Algorithms (2nd ed.), MIT Press and McGraw-Hill*, pages 549–552, 2001.

[92] Joseph Carter. Containers for commercial spent nuclear fuel. https://www.nwtrb.gov/meetings/past-meetings/summer-2016-board-meeting---august-24-2016, August 2016. Accessed: 8/19/2019.

[93] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. Enabling scalable high-performance systems with the intel omni-path architecture. *IEEE Micro*, 36(4): 38–47, 2016.

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

[95] G Fox, M Johnson, G Lyzenga, S Otto, J Salmon, D Walker, and Richard L White. Solving problems on concurrent processors vol. 1: General techniques and regular problems. *Computers in Physics*, 3(1):83–84, 1989.

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

[97] Michael Schliephake and Erwin Laure. Performance analysis of irregular collective communication with the crystal router algorithm. In *International Conference on Exascale Applications and Software*, pages 130–140. Springer, 2014.

[98] Argonne University of Chicago. gslib. https://github.com/gslib/gslib, 2019.

[99] Per H Christensen, Julian Fong, David M Laur, and Dana Batali. Ray tracing for the movie cars. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 1–6. IEEE, 2006.

[100] Geoggrey Gunow, Samuel Shanner, Benoit Forget, and Kord Smith. Reducing 3D MOC storage requirements with axial on-the-fly ray tracing. 2016.

[101] John Tramm. *Development of the random ray method of neutral particle transport for high-fidelity nuclear reactor simulation*. PhD thesis, Massachusetts Institute of Technology, 2018.

[102] swalog. C++ object-pool that provides items as smart-pointers that are returned to pool upon deletion, 2015. URL https://stackoverflow.com/a/27837534/2042320. Accessed: 8/20/2018.

[103] Shigang Li, Torsten Hoefler, and Marc Snir. NUMA-aware shared-memory collective communication for MPI. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 85–96. ACM, 2013.

[104] Guillaume Giudicelli, Wenbin Wu, Colin Josey, Kord Smith, and Benoit Forget. Adding a third level of parallelism to OpenMOC, an open source deterministic neutron transport solver. In *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering, M&C 2019*, 2019.

[105] Anna L Rosen, Mark R Krumholz, Jeffrey S Oishi, Aaron T Lee, and Richard I Klein. Hybrid adaptive ray-moment method (harm2): A highly parallel method for radiation hydrodynamics on adaptive grids. *Journal of Computational Physics*, 330:924–942, 2017.

[106] Geoffrey Gunow, Benoit Forget, and Kord Smith. Stabilization of multi-group neutron transport with transport-corrected cross-sections. *Annals of Nuclear Energy*, 126:211–219, 2019.

[107] M Smith, A Mohamed, A Marin-Lafleche, E Lewis, K Derstine, CH Lee, A Wollaber, and WS Yang. FY2010 status report on advanced neutronics modeling and validation. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2011.

[108] OECD. Benchmark on deterministic transport calculations without spatial homogenisation, MOX fuel assembly 3D extension case. *NEA*, 2005.

[109] Brendan Kochunas, Benjamin Collins, Dan Jabaay, Thomas J Downar, and William R Martin. Overview of development and design of mpact: Michigan

parallel characteristics transport code. In *Proceedings of the 2013 International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering-M and C 2013*, 2013.

[110] Xue Yang, Rajan Borse, and Nader Satvat. {MOCUM} solutions and sensitivity study for {C5G7} benchmark. *Annals of Nuclear Energy*, 96:242 – 248, 2016. ISSN 0306-4549. doi: http://dx.doi.org/10.1016/j.anucene.2016.05.030. URL http://www.sciencedirect.com/science/article/pii/S0306454916303450.

[111] Arnaldo Carvalho De Melo. The new linux perf tools. In *Slides from Linux Kongress*, volume 18, 2010.

[112] Google. gperftools. https://github.com/gperftools/gperftools, 2019.

[113] MIT CRPG. PWR benchmarks. https://github.com/mit-crpg/PWR_benchmarks, 2019.

[114] Guillaume Giudicelli, Kord Smith, and Benoit Forget. Generalized equivalence methods for 3D multi-group neutron transport. *Annals of Nuclear Energy*, 112: 9–16, 2018.

[115] Nathan Andrew Gibson. *Novel Resonance Self-Shielding Methods for Nuclear Reactor Analysis*. PhD thesis, Massachusetts Institute of Technology, 2016.

[116] Jaakko Leppänen, Riku Mattila, and Maria Pusa. Validation of the serpent-ares code sequence using the mit beavrs benchmark–initial core at hzp conditions. *Annals of Nuclear Energy*, 69:212–225, 2014.

[117] Min Ryu, Yeon Sang Jung, Hyun Ho Cho, and Han Gyu Joo. Solution of the beavrs benchmark using the ntracer direct whole core calculation code. *Journal of Nuclear Science and Technology*, 52(7-8):961–969, 2015.

[118] Daniel J Kelly, Brian N Aviles, Paul K Romano, Bryan R Herman, Nicholas E Horelik, and Benoit Forget. Analysis of select beavrs pwr benchmark cycle 1 results using mc21 and openmc. In *Proceedings of the international conference on physics of reactors (PHYSOR2015)*, 2015.

[119] Ho Jin Park, Hyun Chul Lee, Hyung Jin Shim, and Jin Young Cho. Real variance analysis of monte carlo eigenvalue calculation by mccard for beavrs benchmark. *Annals of Nuclear Energy*, 90:205–211, 2016.

[120] Kan Wang, Shichang Liu, Zeguang Li, Gang Wang, Jingang Liang, Feng Yang, Zonghuan Chen, Xiaoyu Guo, Yishu Qiu, Qu Wu, et al. Analysis of beavrs two-cycle benchmark using rmc based on full core detailed model. *Progress in Nuclear Energy*, 98:301–312, 2017.

[121] Zhiyan Wang, Bin Wu, Lijuan Hao, Hongfei Liu, and Jing Song. Validation of supermc with beavrs benchmark at hot zero power condition. *Annals of Nuclear Energy*, 111:709–714, 2018.

[122] Matthew Ellis, J Ortensi, Y Wang, Kord Smith, and RC Martineau. Initial rattlesnake calculations of the hot zero power beavrs. Technical report, Idaho National Lab.(INL), Idaho Falls, ID (United States), 2014.

[123] Raymond E Alcouffe. Diffusion synthetic acceleration methods for the diamond-differenced discrete-ordinates equations. *Nuclear Science and Engineering*, 64(2): 344–355, 1977.

[124] Richard Sanchez and Abdelhuahed Chetaine. A synthetic acceleration for a two-dimensional characteristic method in unstructured meshes. *Nuclear science and engineering*, 136(1):122–139, 2000.

[125] H. Park, D. A. Knoll, and C. K. Newman. Nonlinear acceleration of transport criticality problems. *Nuclear Science and Engineering*, 172(1):52–65, 2012. doi: 10.13182/NSE11-81. URL https://doi.org/10.13182/NSE11-81.

[126] Gareth Rees. Line segment intersection, 2009. URL https://stackoverflow.com/a/565282. Accessed: 3/12/2016.