Memory Efficient Indexing Algorithm for Physical Properties in OpenMC

by

Derek Michael Lax

B.S., Nuclear Engineering
University of Michigan (2012)

Submitted to the Department of Nuclear Science and Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Nuclear Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2015

Author ..... Signature redacted

Derek Michael Lax
Department of Nuclear Science and Engineering
January 16, 2015

Certified by ..... Signature redacted

Benoit Forget
Associate Professor of Nuclear Science and Engineering
Thesis Supervisor

Certified by ..... Signature redacted

Kord Smith
KEPCO Professor of the Practice of Nuclear Science and Engineering
Thesis Supervisor

Accepted by ..... Signature redacted

Mujid Kazimi
TEPCO Professor of Nuclear Engineering
Chairman, Department Committee on Graduate Theses

Memory Efficient Indexing Algorithm for Physical Properties in OpenMC

by

Derek Michael Lax

Submitted to the Department of Nuclear Science and Engineering
on January 16, 2015, in partial fulfillment of the
requirements for the degree of
Master of Science in Nuclear Science and Engineering

Abstract

OpenMC is an open source Monte Carlo code designed at MIT with a focus on parallel scalability for large nuclear reactor simulations. The target problem for OpenMC is a full core high-fidelity multi-physics coupled simulation. This encompasses not only nuclear physics, but also material science and thermohydraulics. One of the challenges associated with this problem is efficient data management, as the memory required for tallies alone can easily enter the Terabyte range. This thesis presents an efficient system for data storage which allows for physical properties of materials to be indexed without any constraints on the geometry. To demonstrate its functionality, a sample depletion calculation with 4 isotopes is completed on the BEAVRS benchmark geometry. Additionally, a temperature distribution assembly layout is presented.

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

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

Acknowledgments

This thesis was completed with significant contribution from William Boyd and Nicholas Horelik. Their work generating images and finalizing the algorithm was invaluable. Additionally, I would like to thank Matthew Ellis, the author of one of the main examples based off of this framework. A special thank you for Professors Benoit Forget and Kord Smith for their assistance in this work. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, US Department of Energy, under Contract DE-AC02-06CH11357.

Contents

1 INTRODUCTION **13**

1.1 Full Core Heterogenous Transport Calculations 16

1.1.1 Full 3D Reactor Problem 16

1.1.2 High Performance Computing 17

1.2 Nodal Methods 17

1.3 Other Monte Carlo Codes 19

1.4 Objectives 20

1.5 Constructive Solid Geometry 20

2 Distribution Algorithm **23**

2.1 Cell Offset Construction 23

2.2 Unique Region Indexing 27

2.3 Example Offset Assignment 30

2.4 Analysis 33

2.4.1 Computational Performance - Region Indexing 33

2.4.2 Memory Requirements 34

2.4.3 Scalability - BEAVRS Benchmark 36

2.4.4 Use with Tallies and Physical Properties 38

2.5 Ease of Input 39

3 Applications **43**

3.1 Depletion 43

3.1.1 Algorithmic Interaction 44

3.1.2 Results 44
3.2 Temperature Distributions 47

4 Conclusion **51**

List of Figures

1-1 The geometric structure of the fuel in a nuclear reactor. Fuel pellets are stacked into each fuel rod and surrounded by a thin cladding. The fuel rods are then placed in a regular lattice to form an assembly, where the fuel pins are held in place with spacer grids. Finally, a regular lattice of assemblies form the core. 14

1-2 Spatial variation of fuel pin composition within a nuclear reactor core from the BEAVRS benchmark. Each color represents a different fuel enrichment and each number represents burnable absorbers. 15

1-3 Core simulation, shown in a step-by-step process. 18

1-4 Construction of a complex shape using CSG. 21

1-5 Local coordinates within nested lattices (a) and a 4×4 lattice of three unique universes, each represented by a unique color (b). 22

2-1 Indexing with example geometry with 3 universe levels. 24

2-2 Conceptual flow for computing a unique ID number for a repeated cell. 28

2-3 Example: Lowest level universes. 30

2-4 Example: Placement of pincell universes. 30

2-5 Example: Counting pincell 1. 31

2-6 Example: Offsets for the top level universe. 31

2-7 Example: A particle is located. 32

2-8 Example: Begin summing offsets from the top down. 32

2-9 Example: Finish summing offsets from the top down. 33

2-10 A pin-cell discretization with 8 azimuthal sectors and 10 equal volume radial rings. 36
2-11 Memory scaling for increasing number of pin-cells up to that required for the PWR problem of interest. 37
2-12 Geometry of the BEAVRS benchmark. *Left:* Radial view, showing 193 fuel assemblies, colored by enrichment. *Right:* Axial view, showing grid spacers and axial pin-cell features. 38
2-13 Sample OpenMC materials input file. 39
2-14 Sample OpenMC materials input file showing unique enrichment per pin. 39
2-15 Sample OpenMC input file without the use of the mapping. 40
2-16 Sample OpenMC input file with the use of the mapping. 41

3-1 Comparison of U-235 in Depletion Experiment for 4 Fuel Pins 45
3-2 Comparison of FP235 in Depletion Experiment for 4 Fuel Pins . . . 45
3-3 Comparison of U-238 in Depletion Experiment for 4 Fuel Pins 46
3-4 Comparison of FP238 in Depletion Experiment for 4 Fuel Pins . . . 46
3-5 Assembly used from BEAVRS benchmark. 15 burnable poisons with 3.1% enriched fuel, colored by material. 48
3-6 Temperature distribution for a simulation containing a unique heat conduction solution for each pin. 49

4-1 Indexing with example geometry with 3 universe levels. 52

List of Tables

1.1 Constraints listed for the 3D Monte Carlo challenge. 16

2.1 Various benchmarks and their respective total memory usage for storing cell information and mappings. 37

Chapter 1

INTRODUCTION

Good data handling methodologies are advantageous for reducing the memory footprint created by large simulations. The largest simulations used in the nuclear industry today are the modeling of heterogeneous neutron transport calculations. A nuclear reactor is comprised of many different repeating geometric structures, some of which occur tens of thousands of times. Ideally, unique geometric structures only need to be defined and stored in memory once. To eliminate wasteful repetition, we desire to define and store a single geometric structure in memory once and refer to it for multiple locations. Similarly, a material need only be defined once as well. However, unlike the geometric structures which currently remain constant throughout the simulation, materials can change uniquely in transport simulations, which must be accounted for without carrying along duplicate information. Achieving this presents several algorithmic hurdles. First, the code must differentiate multiple instances of the same geometric or material construct. Second, for data fetching purposes, whichever algorithm is used must be bi-directional. Finally, the computational cost should be minimal once any initialization is complete.

The utility of this is made obvious through an example of a Pressurized Water Reactor (PWR). A PWR contains a grid of fuel assemblies, such as the one shown in Figure 1-1[2][13][6][18], each containing hundreds of fuel pins. These fuel pins and other parts of the fuel assembly are consistent in geometry throughout the reactor. However, the contents of the fuel pins will vary by location throughout the entirety

of the core, as shown in Figure 1-2[11], where each color represents a different fuel enrichment and the number in each box represents burnable absorbers. By using an algorithm with the properties described above, unique geometric structures, such as the fuel pin and other parts of the core, need only be defined once. Physical properties, such as temperature, material composition, etc, can then be distributed across all instances of the object with a single definition.

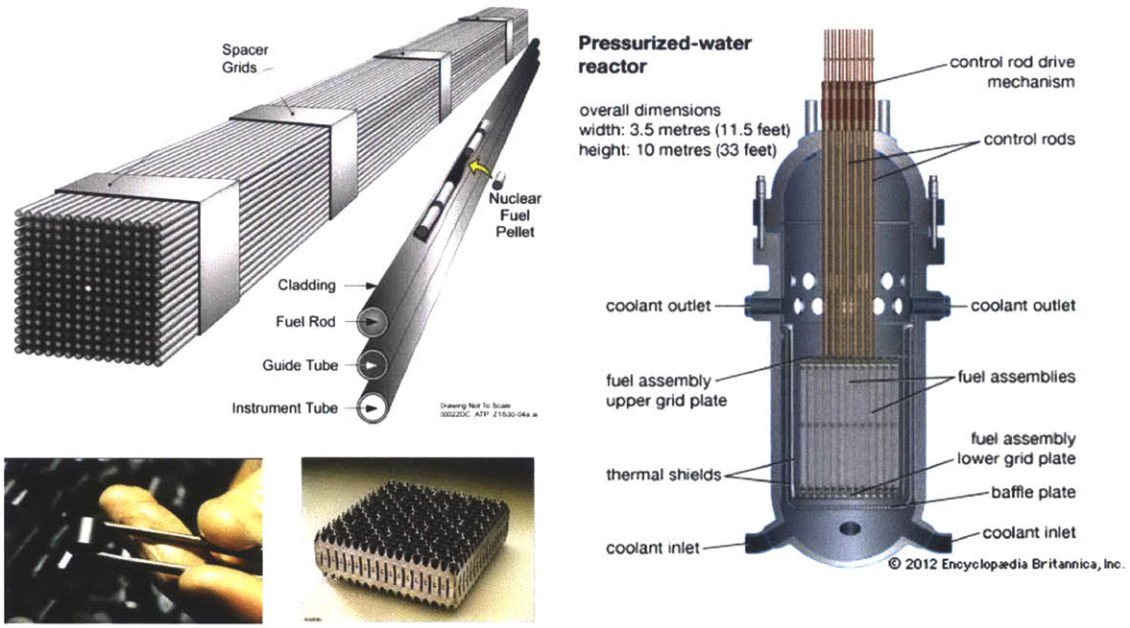


Figure 1-1: The geometric structure of the fuel in a nuclear reactor. Fuel pellets are stacked into each fuel rod and surrounded by a thin cladding. The fuel rods are then placed in a regular lattice to form an assembly, where the fuel pins are held in place with spacer grids. Finally, a regular lattice of assemblies form the core.

The scale of the full-core, fully detailed reactor problem highlights the necessity of indexing the materials and tallies of these regions efficiently if the data structures for each region are not repeated in memory. Such efficient indexing is required in order to implement several recently developed methods that attempt to solve the larger memory hurdles via decomposition - *e.g.* the tally server model discussed in [15] and [16], and the domain decomposition routines presented in [12]. Furthermore, many other applications, such as temperature dependent feedback, shown in Section 3.2, require the use of these algorithms. This thesis discusses the implementation of such

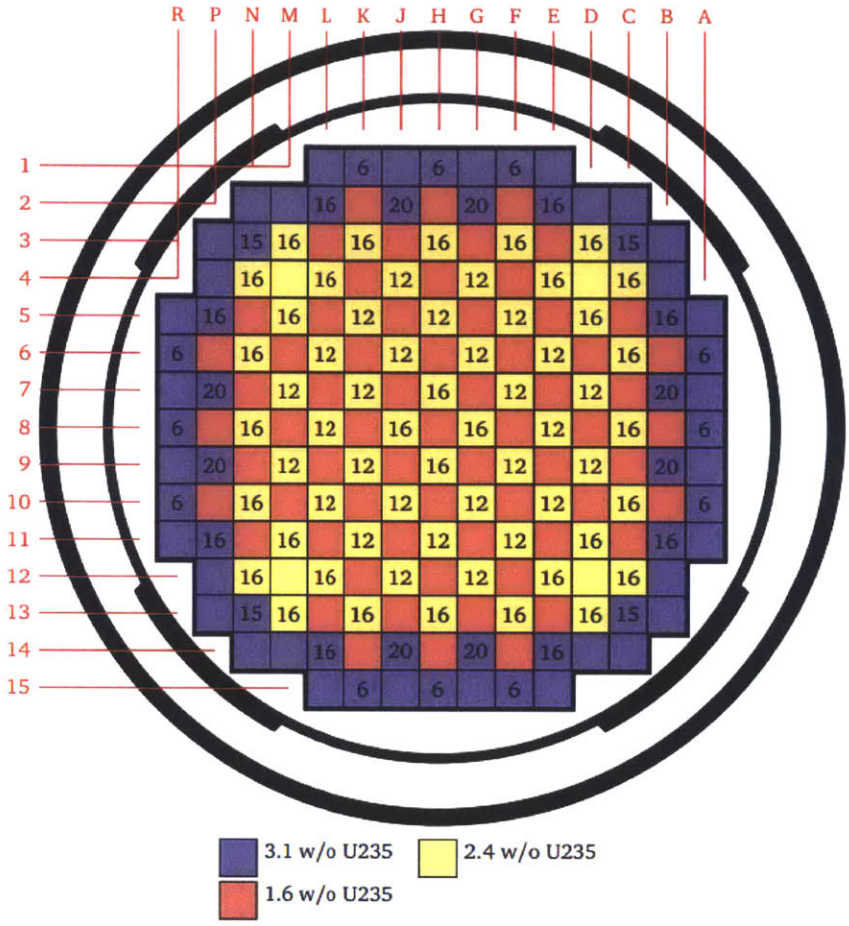


Figure 1-2: Spatial variation of fuel pin composition within a nuclear reactor core from the BEAVRS benchmark. Each color represents a different fuel enrichment and each number represents burnable absorbers.

algorithms in OpenMC. OpenMC[14] is an open source Monte Carlo neutron transport code written in compliance with the FORTRAN 2003 standard. Notable and relevant changes in the 2003 standard include support for object-oriented programming and allocatable memory storage, both of which are used extensively in the algorithms described in this thesis. The geometry model used is the Constructive Solid Geometry (CSG) model, which is described in Section 1.5.

1.1 Full Core Heterogenous Transport Calculations

Full core Monte Carlo reactor simulations are the gold standard for reactor physics calculations. Recent advances in algorithms and in high performance computing platforms have made such simulations potentially realizable for routine reactor analysis. Subsection 1.1.1 outlines a challenge problem for Monte Carlo simulations, with a focus on the parameters that impact the geometric treatment for such models. Subsection 1.1.2 reviews trends for memory storage capacity in high performance computing machines which necessitate scalable algorithms for tally systems and geometric models.

1.1.1 Full 3D Reactor Problem

In 2003, Kord Smith[17] proposed a challenge problem for Monte Carlo reactor physics calculations. The principle details for this problem are enumerated in Table 1.1. The completion of this challenge requires 1% statistical accuracy on all pin powers in a standard PWR with 100 axial planes. For simplicity, we approximate the problem to include 200 assemblies each containing 300 fuel pins. Additionally, this challenge also specifies 10 depletion regions per fuel pin and 100 tracked isotopes. Meeting all of these criteria requires 6 billion tallies which must converge to 1% statistical accuracy, as previously mentioned. Accordingly, this challenge serves as a motivation for the creation of scalable tally systems.

Table 1.1: Constraints listed for the 3D Monte Carlo challenge.

Constraint	Value
# Fuel Assemblies	200
# Axial Planes	100
# Pins per Assembly	300
# Depletion Regions per Pin	10
# Tracked Isotopes	100
Statistical Resolution	$\leq 1\%$

1.1.2 High Performance Computing

The high performance computing community is currently pushing to build an exaflop (10^{18} flops) supercomputer in the early 2020s. Exascale computing may enable heterogeneous transport simulations for nuclear reactors to become possible for the first time. GPU-like and many-core processors will have hundreds to thousands of cores with only 2-5 times the memory of today's multi-core systems [7]. In addition, the memory hierarchy (*e.g.*, multi-level cache) will likely grow more complex with different types of memory for different types of algorithmic characteristics. As a result, intelligent algorithms are needed to enable neutron transport codes to minimize their memory footprint while maximizing cache reuse.

A key area for improving memory performance is the formulation used to construct and categorize geometric models. The algorithm presented in this paper is well suited to minimize the memory footprint needed to index large numbers of geometric regions for transport codes which use the Constructive Solid Geometry formulation described in Section 1.5. Furthermore, the cache performance may be improved by reducing the number of unique regions which must be instantiated, stored and accessed to simulate a particular reactor core geometry.

1.2 Nodal Methods

Prior to the ideal of full core transport simulations by either deterministic or stochastic methods, nodal methods have been used and are still used today to perform full core analysis. Nodal methods solve the diffusion equation, a second-order approximation of the neutron transport equation with low angular resolution.

The solution technique is a multi-step process which involves building the full core simulation up from the roots. Beginning from the Evaluated Nuclear Data Files (ENDF), multi-group cross sections are constructed. Initially this includes a large number of groups, which get reduced as the simulation builds up. The simulation begins on the pin level, with a one dimensional transport calculation performed. This is done once for each unique pin type to gain information about self-shielding.

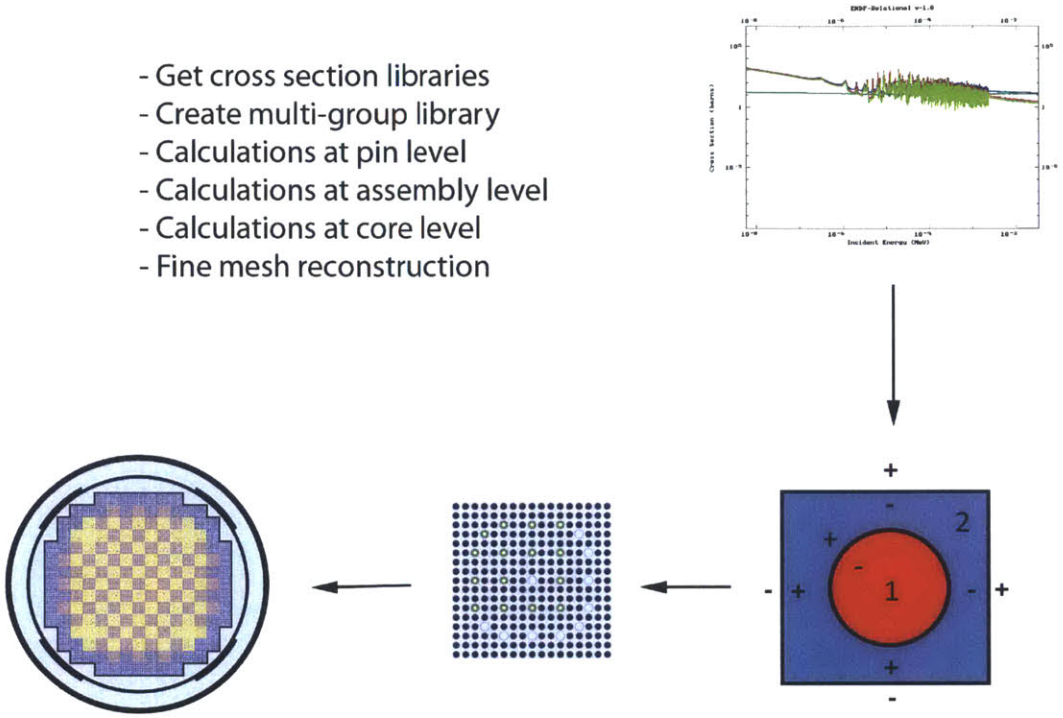


Figure 1-3: Core simulation, shown in a step-by-step process.

Next, the results from the pin simulations are used to create a two dimensional assembly simulation with reflective boundary conditions. Here the number of groups is reduced to the order of hundreds of energy groups. The lattice is simulated using diffusion theory or possibly transport theory. This is done once for each unique lattice to gain information about spatial effects.

Finally, the full core is simulated using the results of the lattices. Diffusion theory is used here, which as mentioned earlier, has limitations on its precision. This limits the spatial resolution for a full core simulation to nodes of roughly 10cm in any dimension. For a full core, the mesh will likely not exceed roughly 20,000 nodes. The three dimensional calculation is completed with few energy groups, oftentimes just two. Fine mesh information, such as pin powers, is reconstructed at the end of the simulation from the 2D lattice calculation's pin to box ratios. A summary of this process is shown in Figure 1-3

To handle data, tables of cross sections are generated as a function of temperature, burnup, and other parameters. Then, during a cross section lookup, interpolations

are used to approximate the value. Data tables are calculated during the assembly stage which store material compositions as a function of burnup, fuel temperature, moderator temperature, and several other factors. These tables are created for each material in the assembly, for each unique assembly, using a simulation where the assembly is in an infinitely repeating configuration.

At the end of the full core run, the flux profile is used to reconstruct the flux in each pin. This allows for the burnup in each pin to be determined, from which the previously created tables are referenced for the material composition. This limits the required data storage and keeps the memory free during the simulation as only homogenized material data is used during runtime.

The nodal method is extremely fast and has low memory requirements, but its precision is burdened with the second-order approximation. With a modern machine, such a method does not face memory storage issues due to the need for a maximum of 20,000 nodes.

1.3 Other Monte Carlo Codes

Obviously, OpenMC is not the first Monte Carlo transport code to face the problem of data efficiency for large simulations. MC21 is another Monte Carlo code for particle simulation that took a slightly different approach[4]. For tallies over repeated structures, MC21 tallies over meshes, which relies on a non-uniform Cartesian grid. MC21 creates a single memory instances of unique geometric structures and allows for them to be placed arbitrarily and assigned properties individually, similar to the methodology presented in this thesis. It is not clear exactly how this is done as the source code is not available. Additionally, for improved tracking speed, MC21 utilizes templates for optimized locating routines[1]. This addresses locating particles within the geometry, which is a lookup mechanism beyond the focus of this thesis.

It is important to note that the challenge discussed in Subsection 1.1.1 has been effectively completed [8]. In Kelly, Sutton, and Wilson's report, the benchmark problem they solve contains 6,362,400 regions. While they do not discuss memory usage

for their current efforts, they note that the memory usage of their previous analysis required 6.5GB of memory per processor. Their tallies only included fission density, but the full challenge problem also asks for reaction rates for 100 isotopes. In Subsection 2.4.3, the BEAVRS benchmark is presented with 100 divisions per pin cell in addition to 100 axial zones. With 100 times more regions and 100 isotopes worth of reaction rates per region, it is certain that memory will be a concern, emphasizing the need for efficient data handling.

1.4 Objectives

The ultimate objective of this thesis is to provide the framework in OpenMC for efficient data handling of physical properties. This thesis will present isotopic depletion and temperature profiles as two potential uses of this framework. The difficulty of these tasks lies primarily in data management, with large amounts of data stores and lookups at each depletion step and at each thermal-hydraulic iteration. To accomplish this, a system to manage tally data and a system for distributing physical properties (specifically temperature and atomic density) are introduced. The calculations involved in depletion are completely independent with respect to location, making parallelism trivial, and so are not discussed.

1.5 Constructive Solid Geometry

Constructive Solid Geometry (CSG) is a methodology used to represent complex geometric models. The Constructive Solid Geometry formulation is the method of choice for many advanced modeling software packages, including some Computer-aided Design (CAD) implementations. CSG allows complex spatial models to be built using Boolean operations - such as intersections and unions - of simple surfaces and building blocks termed *primitives*.

There are a number of benefits to using the CSG formulation. First, CSG enables simulation codes to significantly reduce the memory footprint required to model a

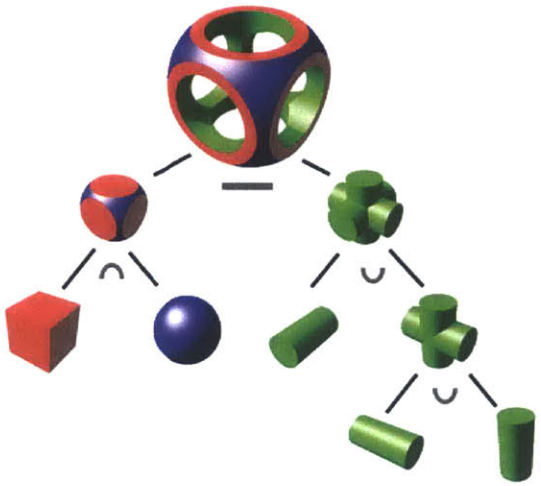
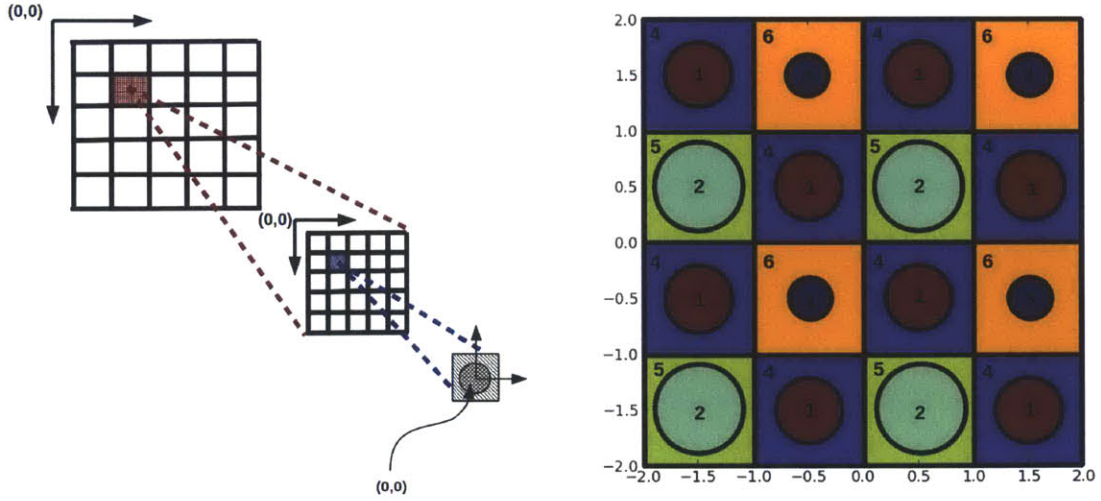


Figure 1-4: Construction of a complex shape using CSG.

geometry. For example, rather than representing each fuel pin explicitly in memory, a fuel pin primitive is represented by a single data structure. A second major benefit of the CSG formulation is that it is a general and extensible approach. This allows for ray tracing routines to be written in an abstract fashion that is independent of the primitives themselves. Furthermore, it allows for exact geometric representation, which eliminates errors introduced by inexact models. Finally, the CSG approach is well suited for reactor models, which typically are highly structured and contain repeating patterns. This is the case for commercial PWRs and BWRs whose cores are built out of a rectangular lattice of fuel assemblies, each of which is a rectangular lattice of fuel pins. The internal geometry of a PWR is shown in Figure 1-1 for reference.

OpenMC uses a system of *surfaces*, *cells*, *universes* and *lattices* in its CSG implementation. Once defined, a surface, cell, universe or lattice may be reused in multiple locations with local coordinate transformations throughout a geometry. A surface divides 3D space into two distinct sets of points termed *halfspaces*. A cell is a bounded region of 3D space specified by Boolean operations (*e.g.*, intersections) between surface halfspaces. A universe is effectively a container of cells which together span the entirety of 3D space. Each cell may be filled with a material or a universe. An example construction is shown in Figure 1-4[19].

Lattices are used to model regular, repeating structures of universes. The lattice specification represents a coordinate transformation such that the center of each lattice cell maps to the origin of the universe within it as illustrated in Figure 1-5a. In addition, the universe within each lattice cell is truncated such that only the portion of the universe encapsulated by the lattice cell is visible to the geometry. Lattices allow for a single universe to be replicated in some or all lattice cells without redundantly storing the universe many times in memory. Figure 1-5b illustrates a simple 4×4 lattice, with each lattice cell filled by one of three different universes. Each universe contains two cells representing the moderator and a fuel pin of some diameter.



(a) (b)

Figure 1-5: Local coordinates within nested lattices (a) and a 4×4 lattice of three unique universes, each represented by a unique color (b).

Chapter 2

Distribution Algorithm

In this thesis, distribution is defined as taking a single instance of an object in memory and using it to uniquely represent every occurrence of said object in the complete geometry. The distribution algorithm indexes repeated geometric structures to track any nonuniform features across all instances in the geometry. To accomplish this, *cell offsets* will need to be calculated for each fill cell and lattice, as described in this chapter. A cell offset is an integer detailing the quantity of a specified cell which occurs below the current depth in the geometry. Cell offsets are described further in Section 2.1.

2.1 Cell Offset Construction

A pre-processing step is necessary to build a mapping that helps identify and index unique regions based on the cells, universes and lattices used to construct each region. This is done by storing offset numbers in the data structures for each fill cell. A *fill cell* is any cell that is filled with a nested universe of cells. Lattice structures should be conceptualized as a collection of fill cells. Offsets correspond to the number of times normal cells (*i.e.* lowest nested level cells filled with a material) from an arbitrary universe appear in the preceding cells of the same nested universe level. The *level* refers to how deep in the CSG tree the current structure is, specifically how many universes and fill cells have been entered. For this purpose, the ordering of cells in a

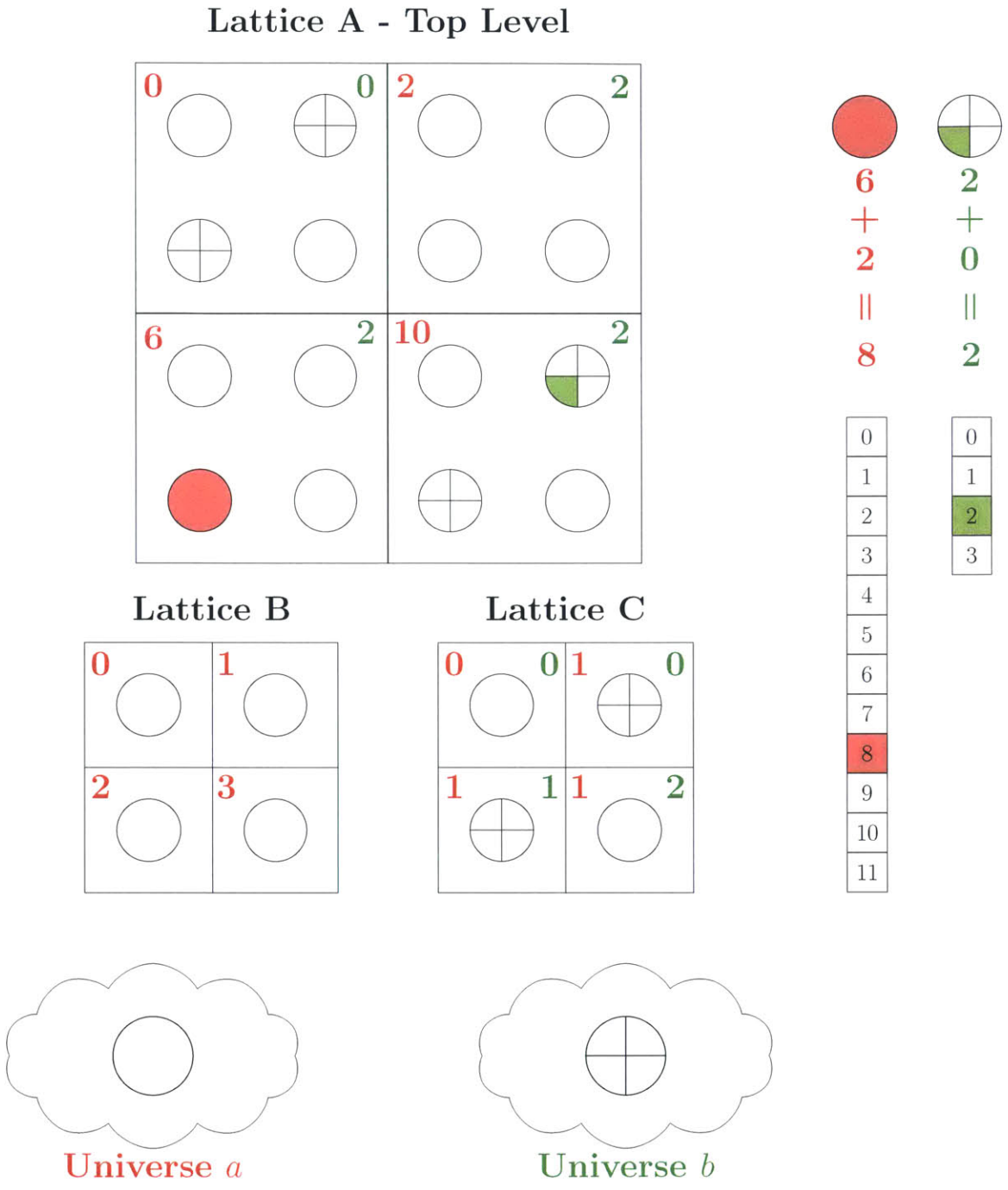


Figure 2-1: Indexing with example geometry with 3 universe levels.

universe can be arbitrary.

Figure 2-1 shows the offsets for a simple example, as well as how they can be used to quickly compute unique ID numbers for material cells. Material cells are defined in pin-cell universes a and b, which are filled into the cells of lattice universes B and C, which are filled into the cells of lattice universe A. The colored numbers in each fill cell are the offsets for each base universe, which can be used to quickly compute a unique ID for each instance of a material cell. From this we can readily make a few key observations:

Observation 1: Cells filled by a material do not need to store an offset. Only cells filled by another universe need to store offsets.

Observation 2: Only one offset needs to be stored in a cell per universe, regardless of the number of cells contained within that universe.

Observation 3: Not all fill cells need to store offsets for all universes. They only need to store offsets for universes filled into other cells on the same level. For example, Lattice B in Figure 2-1 does not need to store offsets for Universe b.

In this manner, Algorithm 1 is a wrapper for Algorithm 2, which builds the offset map starting from the top-level universe and recursing down the universes filled into fill cells. The algorithm is called for each universe that contains cells of interest (*e.g.* universes a and b in the example of Figure 2-1), specified via the *target* argument.

The reader should note that the offsets of each universe containing fill cells need only be computed once. Thus, Algorithm 2 stores the result of computed offsets for a given universe/target combination in a global variable. Also note that on line 8 of Algorithm 2, the target universe could be contained within the current fill cell at an arbitrary depth of recursion (*e.g.* Universe a is two levels down from position (0,1) in Lattice A in the example of Figure 2-1). This condition should be handled with an additional pre-processed mapping so trees of recursion aren't followed more than once. Finally, the result of the top-level call of Algorithm 2 on Line 10 of Algorithm 1 is the total number of unique appearances of the target universe in the global geometry. This is useful for knowing how large to allocate tally and materials arrays.

Algorithm 1 Offset Construction For Target Universes

```
1: procedure COMPUTEALLOFFSETS(target)
2:     targets = empty set                          ▷ Build list of universe targets
3:     for all universes in geometry do
4:         if universe contains any cells of interest then      ▷ Not all cells require
   indexing
5:             targets = targets + universe
6:         end if
7:     end for
8:     universe = GETBASEUNIVERSE( )                ▷ Start from the base universe
9:     for all targets do
10:        n_instances[target] = COMPUTEOFFSET(universe, target)  ▷ Algorithm 2
11:    end for
12: end procedure
```

Algorithm 2 Recursive Offset Construction Algorithm

```
1: procedure COMPUTEOFFSETS(universe u, universe target)
2:     n = 0
3:     for all cells in u do
4:         cell.offset[target] = n
5:         if cell is FILLCELL then
6:             if cell.fill is target then
7:                 return 1
8:             else if cell.fill contains target then    ▷ Could be on any lower level
9:                 if [cell.fill, target] not in computed_offsets then
10:                    computed_offsets[cell.fill, target] = computeOffsets(cell.fill, tar-
   get)
11:                end if
12:                n = n+ computed_offsets[cell.fill, target]
13:            else
14:                return 0
15:            end if
16:        end if
17:    end for
18:    return n
19: end procedure
```

2.2 Unique Region Indexing

Unique ID numbers for repeated cells are efficiently computed using the offset maps and the algorithm conceptualized in Figure 2-2. Algorithms 3-6 detail how this computation is implemented, using recursion to follow the chains of fill cells down to the normal cells that correspond to particular (x, y, z) positions. As the chain is traversed, the unique ID is updated using the offset from the fill cell at each nested level in the CSG hierarchy. Here, positions in lattice universes are treated separately from normal fill cells (Algorithm 6) since lattice positions are not typically implemented using cell data structures.

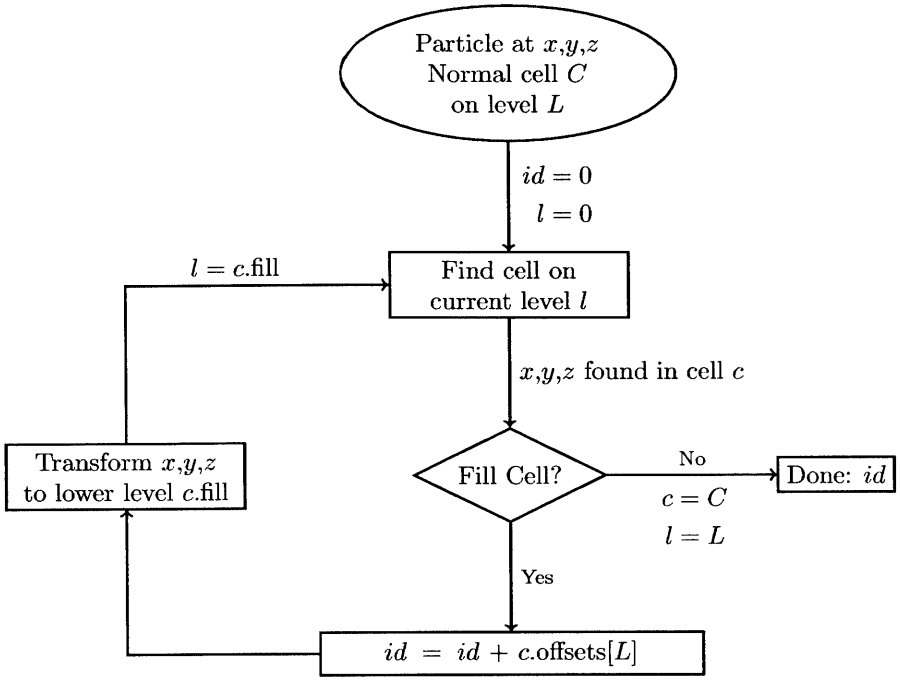


Figure 2-2: Conceptual flow for computing a unique ID number for a repeated cell.

Algorithm 3 Indexing Algorithm

```
1: procedure GETUNIQUEINDEX(x, y, z)
2:     cell = FINDCELL(x, y, z)                    ▷ Existing Monte Carlo Routine
3:     universe = GETBASEUNIVERSE( )               ▷ Initialize to the base universe
4:     target = cell.universe
5:     return GETUNIVERSEINDEX(0, x, y, z, universe, target)    ▷ Algorithm 4
6: end procedure
```

Algorithm 4 Recursive Universe Index Calculation

```
1: procedure GETUNIVERSEINDEX(id, x, y, z, universe, target)
2:     if universe is target then
3:         return id
4:     else
5:         for all cell ∈ universe do                ▷ Loop over all cells in universe
6:             if (x,y,z) ∈ cell then                ▷ If the location is in the cell
7:                 id = GETCELLINDEX(id, x, y, z, cell, target)   ▷ Algorithm 5
8:                 return id
9:             end if
10:        end for
11:    end if
12: end procedure
```

Algorithm 5 Recursive Cell Region Index Calculation

```
1: procedure GETCELLINDEX(id, x, y, z, cell, target)
2:     if cell.type is FILLCELL then                 ▷ Cell filled by a universe
3:         id = id + cell.offsets[target]
4:         id = GETUNIVERSEINDEX(id, x, y, z, cell.universe, target)   ▷ Algorithm 4
5:     else if cell.type is LATTICE then             ▷ Cell filled by a lattice
6:         id = GETLATTICEINDEX(id, x, y, z, cell.lattice, target)     ▷ Algorithm 6
7:     end if
8:     return id
9: end procedure
```

Algorithm 6 Recursive Lattice Region Index Calculation

```
1: procedure GETLATTICEINDEX(id, x, y, z, lattice, target)
2:     i = ⌊(y×lattice.maxrows)/lattice.height⌋          ▷ Compute row index
3:     j = ⌊(x×lattice.maxcols)/lattice.width⌋           ▷ Compute column index
4:     y = y − (i×lattice.height)/lattice.maxrows        ▷ Adjust to lattice cell coordinate system
5:     x = x − (j×lattice.width)/lattice.maxcols         ▷ Adjust to lattice cell coordinate system
6:     universe = lattice.universes[i][j]                ▷ Get universe filling lattice cell
7:     id = id + lattice.offsets[i][j][target]
8:     id = GETUNIVERSEINDEX(id, x, y, z, universe, target)        ▷ Algorithm 6
9:     return id
10: end procedure
```

Algorithm 6 is given for the case of a regular, rectangle lattice configuration. This could be applied to an arbitrary configuration by locating which lattice location the particle is at, applying the coordinate transformation, and then making the recursive call.

2.3 Example Offset Assignment

This section presents a more in-depth walk-through of Figure 2-1. At the lowest level, there exist two universes, each containing a fuel pin surrounded by a moderator. As shown in Figure 2-3, pincell universe 1 contains two materials, the fuel pin material and the surrounding material, and pincell universe 2 contains five.

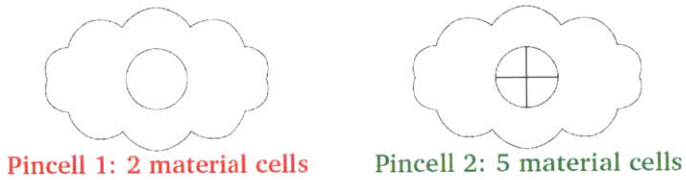


Figure 2-3: Example: Lowest level universes.

Presenting again the full geometry, pincell universes 1 and 2 are cropped to fit into the regular lattice in sub universes 1 and 2 in various locations. Pincell universe 1 appears six times and pincell universe 2 appears two times.

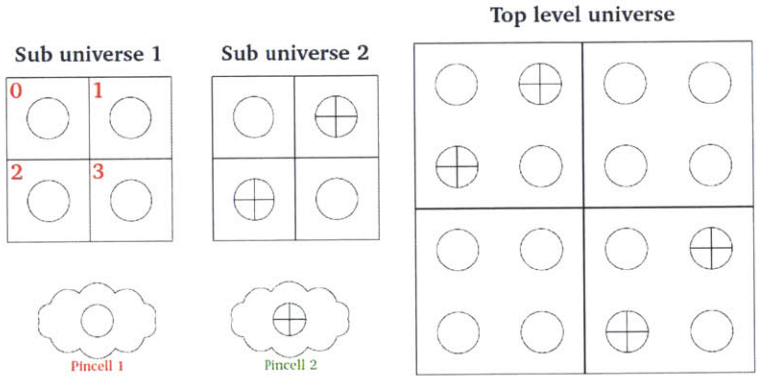


Figure 2-4: Example: Placement of pincell universes.

Beginning from the top level universe with a target of pincell 1, we assign a starting offset of 0 to the first universe, as no instances of pincell 1 have been counted yet, and proceed to count the number of instances of pincell 1 in all sub universes, beginning at 0. For this example, the counting moves left to right and top to bottom, but the direction does not matter so long as it is consistent. The first location in Sub universe 1 contains a single instance of pincell 1, so the offset of the next location becomes

$0 + 1 = 1$. Each subsequent location also contains, a single instance, leading to the single increment between locations. When counting Sub universe 2, it is clear that two of the locations do not contain any instances of pincell 1, and so we add 0 to the running total when counting these locations.

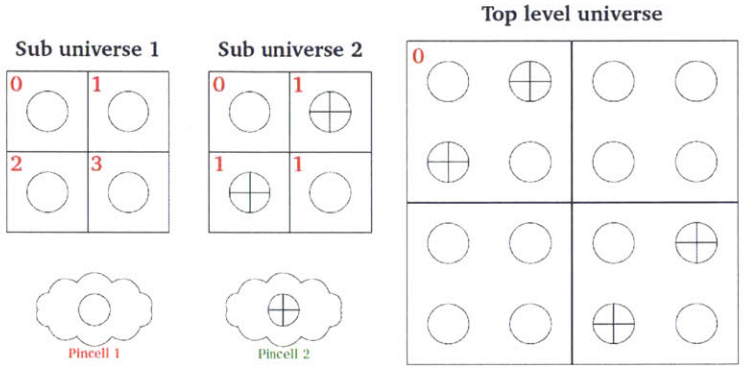


Figure 2-5: Example: Counting pincell 1.

Once the sub universes have been indexed, higher up universes can be indexed. As explained earlier, the first lattice location of the top level universe is 0. Subsequent universe offsets are defined as the offset of the previous universe plus the number of target instances found in the previous universe. There are 2 instances of pincell 1 within the first lattice location of the top level universe, so the next lattice location is assigned value $0 + 2 = 2$. The second and third lattice locations contain 4 instances of pincell 1, so the third and fourth lattice locations are assigned offsets $2 + 4 = 6$, and $6 + 4 = 10$, respectively.

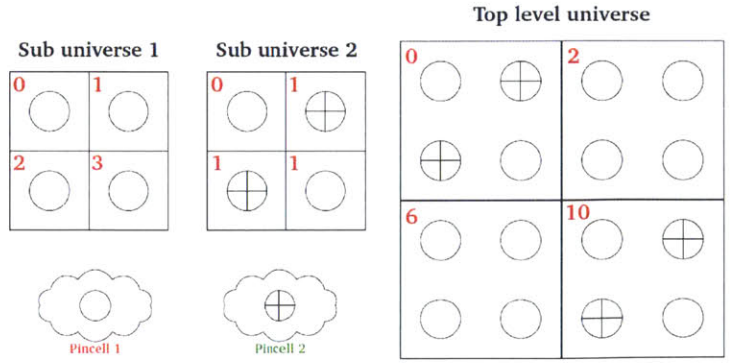


Figure 2-6: Example: Offsets for the top level universe.

Now that the indexing is complete, we would like to be able to determine the unique index of a cell from a location. For example, suppose a particle was located in the region with the filled-in red circle as shown in Figure 2-7.

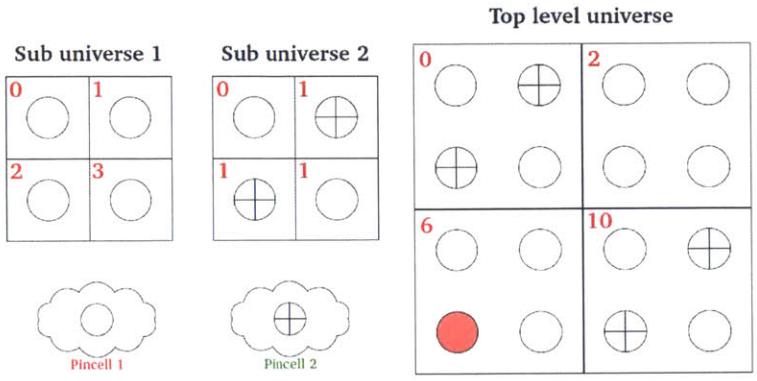


Figure 2-7: Example: A particle is located.

Beginning again from the top level universe, we sum the offsets as we move down to the lowest level. The first offset encountered is 6.

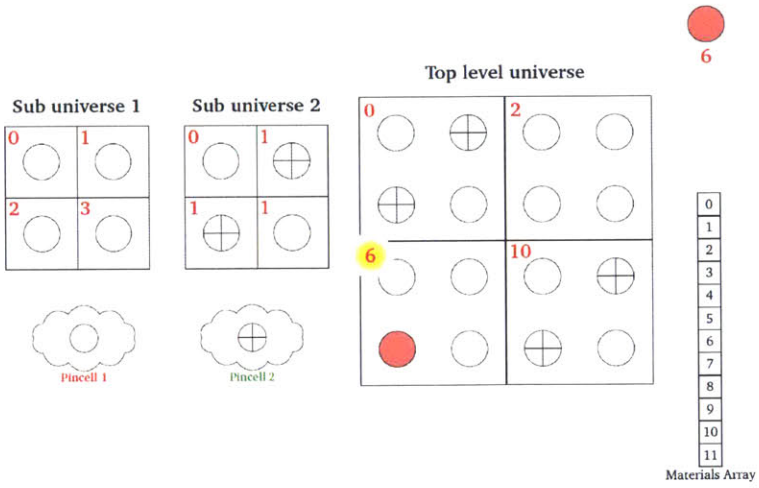


Figure 2-8: Example: Begin summing offsets from the top down.

Next the offset 2 is encountered within sub universe 1. We add the value 2 to our current sum of 6 for a total of 8. Within pincell universe 1, there is a single occurrence of our target, so we add 0 to our sum. We added an offset of 0 for pincell universe 1 because the offset counts instances prior to the current location within the current

level of the geometry, of which there are none. Now that our traversal through the geometry has finished, we know that our unique location is equal to the sum of the offsets during our search. Therefore, this location has index 8 in our physical property array, which in this case turned out to be a material.

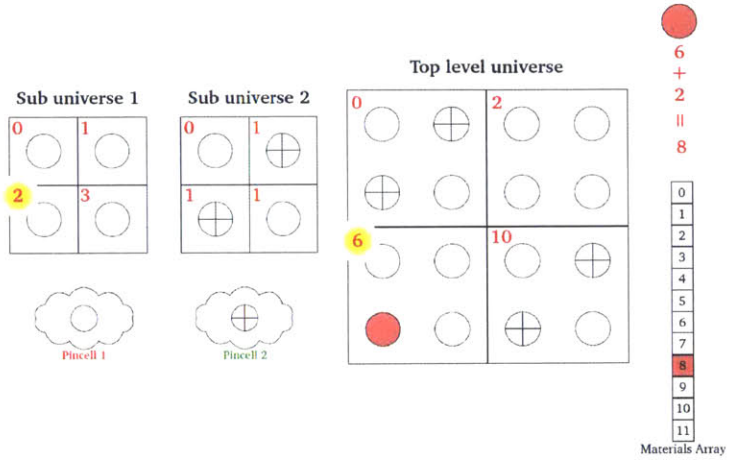


Figure 2-9: Example: Finish summing offsets from the top down.

2.4 Analysis

There are three aspects of this algorithm which are important from a computational standpoint and which are detailed in this section: the cost to initialize offsets, the cost to compute a region index and the memory footprint required to store cell offsets. The initialization cost, as well as the best and worst case scenarios for determining a unique region for a location using Algorithms 3-6, are described in Subsection 2.4.1. The memory requirements for storing cell offsets are detailed in Subsection 2.4.2, and Subsection 2.4.3 presents a case study using the BEAVRS benchmark problem to demonstrate the scalability of this algorithm.

2.4.1 Computational Performance - Region Indexing

The performance of the region indexing algorithms (Algorithms 3-6) are necessarily geometry dependent, namely on the number of cells, universes, and lattices as well as

the number of nested universe levels. The computation will primarily scale according to the search complexity across cells in a universe in Algorithm 4.

The initialization routine will visit every occurrence of every cell exactly once. The cost of this operation therefore scales as the total number of instances of all cells. The geometry can be accurately represented as a k-ary tree, a tree in which each node has at most k nodes. The maximum number of nodes in a k-ary tree is given as:

$$N = k^h \tag{2.1}$$

where k is the largest number of children for any node on the tree, and h is the maximum depth of the tree.

The best case search scenario will occur when the location (x, y, z) is found in the first cell searched for each nested universe level. In this case, the computational complexity is proportional to the number of nested universe levels l in the geometry. Therefore, the required cost for the best case scenario scales as $O(l)$.

The worst case scenario will occur when Algorithm 4 must iterate to the final cell in each universe at each nested universe level. Accordingly, the computational complexity for this case is proportional to the number of nested universe levels multiplied by the average number of cells in each universe. As a result, the required cost scales as $O(l * \bar{c})$, where $\bar{c}$ is the average number of cells per universe.

2.4.2 Memory Requirements

A simple performance model was developed for the memory footprint for the naive implementation (storing unique data structures for each unique instance of a cell) as well as for the approach described in this paper using cell offsets. This model only considers geometry storage and does not account for the memory requirements to actually store the tallies. The memory footprint M in bytes for the naive formulation with replicated cells/universes is given by the following:

$$M = \sum_p N_p \left[N_{p,c} m_c + \sum_c N_{p,c,s} m_s \right] \tag{2.2}$$

where p is a pin-cell universe, N_p is the number of times pin-cell universe p is repeated, $N_{p,c}$ is the number of cells in pin-cell universe p, and $N_{p,c,s}$ is the number of surfaces in cell c of pin-cell p. Likewise, m_c and m_s are the memory footprints for a single instance of the corresponding cell or surface data structure, which will depend on the implementation of the relevant data structures for a particular code. In OpenMC, these data structures consume $m_c \approx 120$ bytes and $m_s \approx 56$ bytes, respectively.

The algorithm presented in this paper only requires a single instance in memory to store each unique cell and universe. Hence, the memory footprint model for the naive case in Equation 2.2 may simply be modified by assigning $N_p = 1$. In addition, the model must include a term to account for the memory footprint consumed by the cell offsets. The cell offset memory footprint depends on which material cells have been specified by the user for tallying, as well as the division of fill cells across nested universe levels. Accordingly, the memory requirements for the offset mapping scale with the number of unique universes containing cells of interest, and the number of fill cells and lattice locations. An upper bound for the case where all material cells are targeted for tallies is given by the following:

$$M = \sum_p \left[N_{p,c} m_c + \sum_c N_{p,c,s} m_s \right] + N_u (N_{fc} + N_l) m_o \tag{2.3}$$

where N_u, N_{fc}, and N_l are the number of unique universes, the number of fill cells, and the number of lattice locations, respectively. m_o is the memory to store single offset value (in OpenMC, $m_o \approx 8$) as a 64-bit integer). The notable between Equation 2.2 and Equation 2.3 is the removal of the N_p term, which will greatly reduce the memory footprint.

To generate a few simple estimates, the average number of surfaces per cell is assumed to be $\overline{N_{p,c,s}} = 4$ (*e.g.* one azimuthal section of a radial ring in the pin-cell depicted in Figure 2-10). In addition, the average number of cells per universe can be approximated to that required for the challenge problem presented in Subsection 1.1.1: $\overline{N_{p,c}} = 10$ depletion zones $\times 100$ axial regions $= 1000$. Furthermore, for a PWR, it is

typical to use two nested universe levels with the first representing a core of roughly 200 fuel assemblies and the second representing an array of 17 $\times$ 17 fuel pins. Given these assumptions, Equation 2.2 and Equation 2.3 can be used to determine the effect of increasing the number of pin-cells on the memory footprint as shown in Figure 2-11. In Figure 2-11, the pin-cell was assumed to be replicated for all fuel positions of a simplified PWR (*i.e.* N_p ranges from 1 to $193 \times (17 \times 17 - 25) = 50,952$). The indexing case assumes 3 universes and that the number of lattice locations is equal to $17 \times 17 \times 2 = 578$ (maximum inner and outer lattice locations). Additionally, Figure 2-11 only considers fuel pins in the memory cost analysis. As illustrated in the next section, the cell indexing algorithm presented in this paper results in a memory savings of two orders of magnitude for the full 3D reactor challenge problem.

Figure 2-10: A pin-cell discretization with 8 azimuthal sectors and 10 equal volume radial rings.

2.4.3 Scalability - BEAVRS Benchmark

The completion of the full 3D reactor challenge will require highly scalable model. Table 2.1 provides a tabular comparison of the memory demands for a simplified BEAVRS model with increasing demand. The displayed total memory is the sum of the data within the cell structure and the mapping algorithm. It is assumed that all

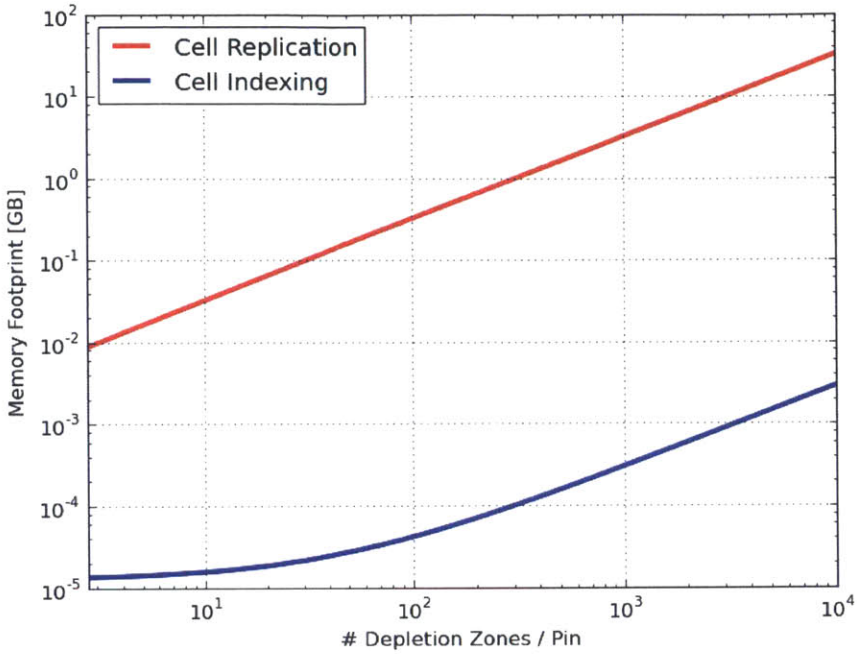


Figure 2-11: Memory scaling for increasing number of pin-cells up to that required for the PWR problem of interest.

cells have constant memory usage that varies based on the computer code used. The Benchmark for Evaluation and Validation of Reactor Simulations (BEAVRS) [11] is illustrated in Figure 2-12 and is used to benchmark performance of the algorithm for a realistic reactor model. Of the 4 cases given in the table, the first two depict a single assembly, and the final two depict the entire core. Again, the full 3D reactor challenge serves as motivation for determining the desired complexity. 3 pin types will be used, each with 10 radial rings, 10 azimuthal sectors, and 100 axial subdivisions, for a total of 2,890,000 unique regions per assembly. With 193 assemblies in the core, the total number of regions for the entire BEAVRS core is roughly 560 million.

Table 2.1: Various benchmarks and their respective total memory usage for storing cell information and mappings.

Mapping	# unique pins	# lattice cells	# unique regions	Mem. Footprint [MB]
Yes	3	1 x 17 x 17	2,890,000	48.6
No	3	1 x 17 x 17	2,890,000	3470
Yes	3	193 x 17 x 17	557,770,000	60.1
No	3	193 x 17 x 17	557,770,000	66900

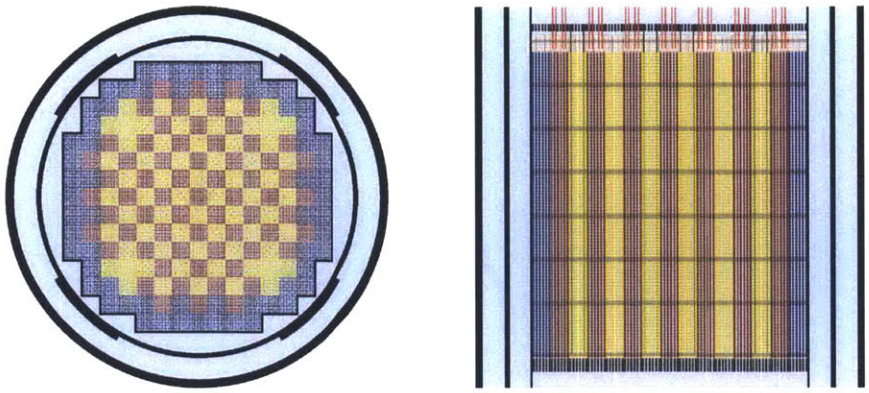


Figure 2-12: Geometry of the BEAVRS benchmark. *Left:* Radial view, showing 193 fuel assemblies, colored by enrichment. *Right:* Axial view, showing grid spacers and axial pin-cell features.

Given the scale and resolution of the benchmark shown, the memory demands are quite low. In fact, the complete cell data and mapping memory is so small that it could reasonably be loaded onto each processor in a highly parallel machine, minimizing the volume of network requests and further improving code scalability. The map's efficiency is further supported by the fact that the number of unique cells could not be reduced as far without having an indexing method to tally their unique instances. Additionally, a mesh based tally must have a mapping of some kind to allow support for depletion material indexing as well. Both of these features are available with this algorithm, and at minimal cost, even for problems of great size.

2.4.4 Use with Tallies and Physical Properties

Scoring tallies with distributed cells in easy. Each time the scoring routine is called, a lookup is made to determine which instance of the cell the particle is currently in, and the corresponding bin is then altered. This alteration is cheap from a development perspective and will work with any type of tally.

One of the objectives mentioned for this thesis was to introduce the ability to distribute physical properties across materials. A distributed material is defined by a single object in memory which is capable of storing nonuniform physical properties of a material. This can be achieved using the same algorithms described previously. By associating a distributed material with a distributed cell, offsets can be shared, and so

no additional initialization cost is incurred. Temperature, atomic densities, and any other physical property, can be distributed on a material and looked up in the same fashion. Two example input files for material specification are shown in Figure 2-13 and Figure 2-14. In Figure 2-13, only a single set of values is given for each property, which signals to the code to start every instance with the same initial state. Support also exists for individual specification of all instances, as shown in Figure 2-14

```
<?xml version="1.0"?>
<material id="1">
  <density value="10.0" units="g/cc" />
  <compositions>
    <units>ao</units>
    <nuclide name="U-235" xs="71c"/>
    <nuclide name="U-238" xs="71c"/>
    <values> 3.0 97.0 </values>
  </compositions>
  <temperature>
    <values> 600 </values>
  </temperature>
</material>
```

Figure 2-13: Sample OpenMC materials input file.

```
  <material id="1">
    <density value="10.0" units="g/cc" />
    <compositions>
      <units>ao</units>
      <nuclide name="U-235" xs="71c"/>
      <nuclide name="U-238" xs="71c"/>
      <values>
       1.0 99.0
       2.0 98.0
       3.0 97.0
       4.0 96.0
      </values>
    </compositions>
  </material>
```

Figure 2-14: Sample OpenMC materials input file showing unique enrichment per pin.

2.5 Ease of Input

Without the functionality to distribute a unique tally over all instances of a cell, each tally must be defined uniquely. For large full-core criticality calculations, this may require defining the same geometry multiple times for each tally region. This

requires the user to spend additional, error-prone time writing large input files by hand or writing scripts to generate the input files. In addition, it is difficult to read and modify such large input files.

An example input for OpenMC is given below for the cases with and without the indexing algorithm. This example consists of a 2 $\times$ 2 square lattice with cylindrical fuel pins at the center of each lattice location. Vacuum boundary conditions are applied to the external boundaries of the system.

```
1
2  <?xml version="1.0"?>
3  <geometry>
4
5    <cell id="1" fill="5" surfaces="1 -2 3 -4" />
6    <cell id="201" universe="21" material="1" surfaces="-5" />
7    <cell id="202" universe="21" material="2" surfaces="5" />
8    <cell id="203" universe="22" material="1" surfaces="-5" />
9    <cell id="202" universe="22" material="2" surfaces="5" />
10   <cell id="205" universe="23" material="1" surfaces="-5" />
11   <cell id="202" universe="23" material="2" surfaces="5" />
12   <cell id="207" universe="20" material="1" surfaces="-5" />
13   <cell id="202" universe="20" material="2" surfaces="5" />
14
15   <lattice id="5">
16     <type>rectangular</type>
17     <dimension>2 2</dimension>
18     <lower_left>-2.0 -2.0</lower_left>
19     <width>1.0 1.0</width>
20     <universes>
21       20 21
22       22 23
23     </universes>
24   </lattice>
25
26   <surface id="1" type="x-plane" coeffs="-2.0" boundary="vacuum" />
27   <surface id="2" type="x-plane" coeffs="2.0" boundary="vacuum" />
28   <surface id="3" type="y-plane" coeffs="-2.0" boundary="vacuum" />
29   <surface id="4" type="y-plane" coeffs="2.0" boundary="vacuum" />
30   <surface id="5" type="z-cylinder" coeffs="0.0 0.0 0.3" />
31
32 </geometry>
33
34 <?xml version="1.0"?>
35 <tallies>
36
37   <tally id="1">
38    <filter type="cell">
39      <bins>
40       201 203 205 207
41       </bins>
42    </filter>
43    <scores>total</scores>
44   </tally>
45
46 </tallies>
```

Figure 2-15: Sample OpenMC input file without the use of the mapping.

```
<?xml version="1.0"?>
<geometry>

  <cell id="1" fill="5" surfaces="1 -2 3 -4" />
  <cell id="201" universe="2" material="1" surfaces="-5" />
  <cell id="202" universe="2" material="2" surfaces="5" />

  <lattice id="5">
    <type>rectangular</type>
    <dimension>2 2</dimension>
    <lower_left>-2.0 -2.0</lower_left>
    <width>1.0 1.0</width>
    <universes>
      2 2
      2 2
    </universes>
  </lattice>

  <surface id="1" type="x-plane" coeffs="-2.0" boundary="vacuum" />
  <surface id="2" type="x-plane" coeffs="2.0" boundary="vacuum" />
  <surface id="3" type="y-plane" coeffs="-2.0" boundary="vacuum" />
  <surface id="4" type="y-plane" coeffs="2.0" boundary="vacuum" />
  <surface id="5" type="z-cylinder" coeffs="0.0 0.0 0.3" />

</geometry>

<?xml version="1.0"?>
<tallies>

  <tally id="1">
   <filter type="distribcell">
     <bins>
      201
      </bins>
   </filter>
   <scores>total</scores>
  </tally>

</tallies>
```

Figure 2-16: Sample OpenMC input file with the use of the mapping.

With the algorithm presented in this paper, a total of 6 lines are saved in the input specification. This may appear to be a small savings, but the implications for larger simulations scale well. For a 2×2 lattice, with only 2 divisions within the cell (fuel and moderator), we reach a total of 8 cells. The 6 lines saved are a result of the repetition of 6 of those 8 cells, as we need only to define the fuel and moderator geometries once. Once the system is expanded to a larger simulation, the savings are large. For a standard PWR with 192 assemblies, each with 17×17 pins, these savings extrapolate to 55,000 lines eliminated from the input file. The previous section gave memory requirements for the map proportional to n, where n is the number of cell definitions. In this case, $n = 2$, which results in the algorithm requiring only a very

small memory overhead. The total memory overhead for the map will be equal to roughly 8 * (2 + 55000 Bytes) = 440 KB for a full core. Reduction in XML input file size is also important due to XML parsers. For the full core BEAVRS benchmark with over 550 million unique regions in the fuel alone, an XML input file which must describe each pin's geometry uniquely would be prohibitive in size.

Savings to the length of the input file will vary depending on the code. The Serpent [9] code, for example, will only achieve savings of 1 line per assembly. This is a consequence of the lattice based input structure which best facilitates tally usage in that code. Input file savings of any size may not seem of great value to the user, as oftentimes the input file is generated by a customized script, but ultimately, performance increases take place within the code through improved geometry data management.

Chapter 3

Applications

This chapter presents two applications built on this framework. Section 3.1 uses distributed tallies and atomic densities to complete a simple depletion example. Section 3.2 uses distributed tallies and temperatures to complete a temperature feedback experiment.

3.1 Depletion

Using the framework created in the previous sections, a simplified example of isotopic depletion is completed using the BEAVRS geometry. This simplified case will only deplete Uranium 235 and Uranium 238. Each of these isotopes will deplete only through fission in a lumped fission product which will have no further interactions. Reaction rates will be integrated over the entire energy spectrum, and all radioactive decay is assumed to be negligible. Energy per fission will be constant with a constant core power of 10MW. Depletion calculations were completed using analytical solutions of differential equations, as the system was sufficiently simple to justify this method. The tallies of fission rates are tracked using this framework, as are the changing number densities.

3.1.1 Algorithmic Interaction

In this simulation, multiple examples of the distribution algorithm tie together in order to successfully model depletion. Beginning with the simulation environment, the fuel pin is defined as a distributed material, which tracks a unique value of its nuclide densities for each of the 4 isotopes in this simulation: U-235, U-238, 235FP, and 238FP. For the sake of simplicity, only a single tally (with 2 bins) is added to the cell containing the fuel pins. This tally tracks the fission reaction rate in each pin cell for U-235 and U-238 independently. Because the indexing algorithm is consistent between tallies and material properties, and because it is deterministic in nature, this can take place completely agnostic of the geometry. Once each steady state calculation is completed, the reaction rate data from the tallies is collected, along with their corresponding number densities. The tally information and number densities are collected using algorithms 3-6. After each depletion step is completed, the tallies are cleared for the next time step, and the cycle continues for all time steps. No predictor-corrector or other correction methodology was used.

3.1.2 Results

A brief simulation of 500,000 particles per time step for 18 depletion time steps is completed, with the number densities shown in the plots below. Each time step consisted of 5 active cycles of 100,000 particles. We expect to see a decline in both fuels, with a significantly larger decline in the U-235 due to the larger fission cross section. The four pins selected include the pins with the highest and lowest fluxes, and two pins with fluxes in between. All pins began with the same amount of U-235 and U-238, at 10.29 g/cc at 1.55% enrichment. The fission rates of U-235 and U-238 were tallied separately, with a fission cross section of 585 and 1.68 * 10^5 barns used for U-235 and U-238, respectively.

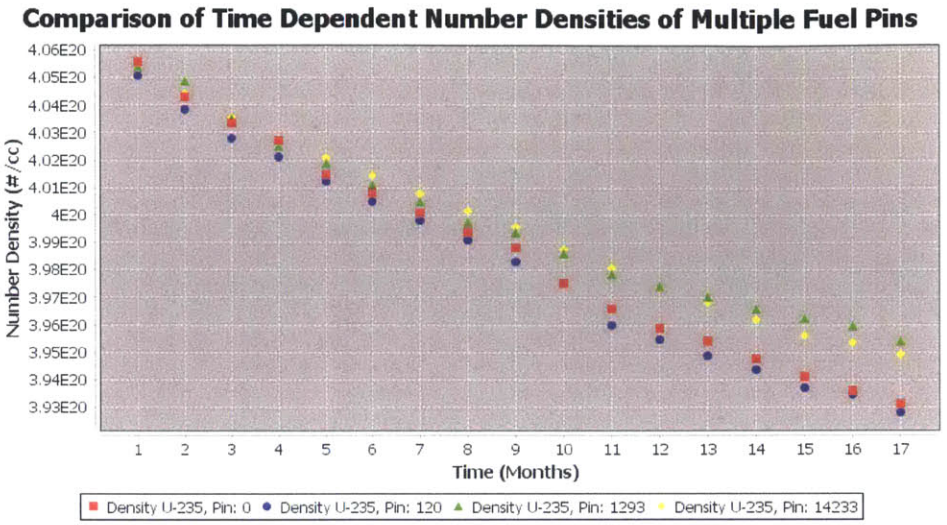


Figure 3-1: Comparison of U-235 in Depletion Experiment for 4 Fuel Pins

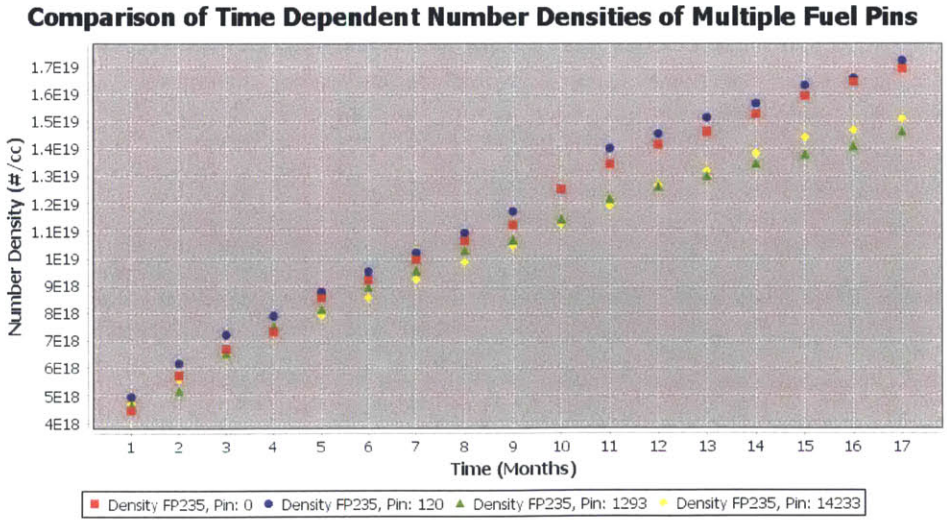


Figure 3-2: Comparison of FP235 in Depletion Experiment for 4 Fuel Pins

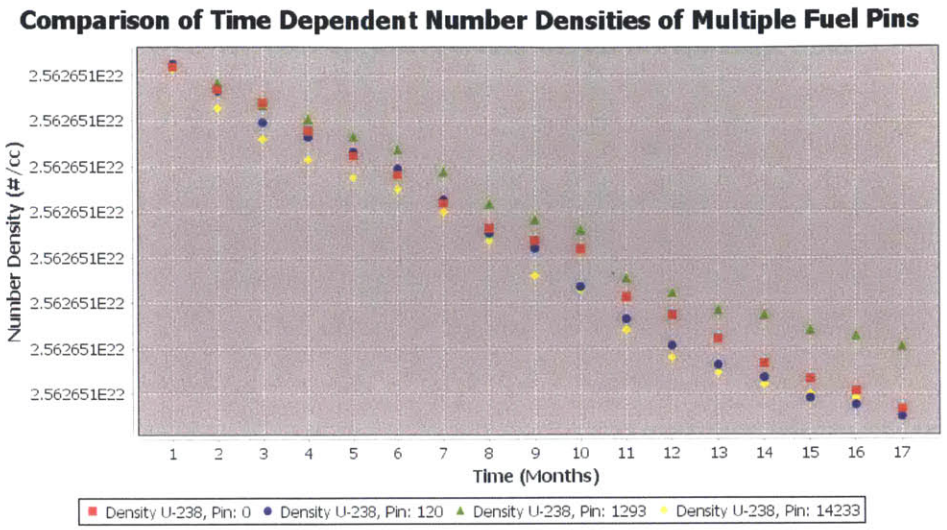


Figure 3-3: Comparison of U-238 in Depletion Experiment for 4 Fuel Pins

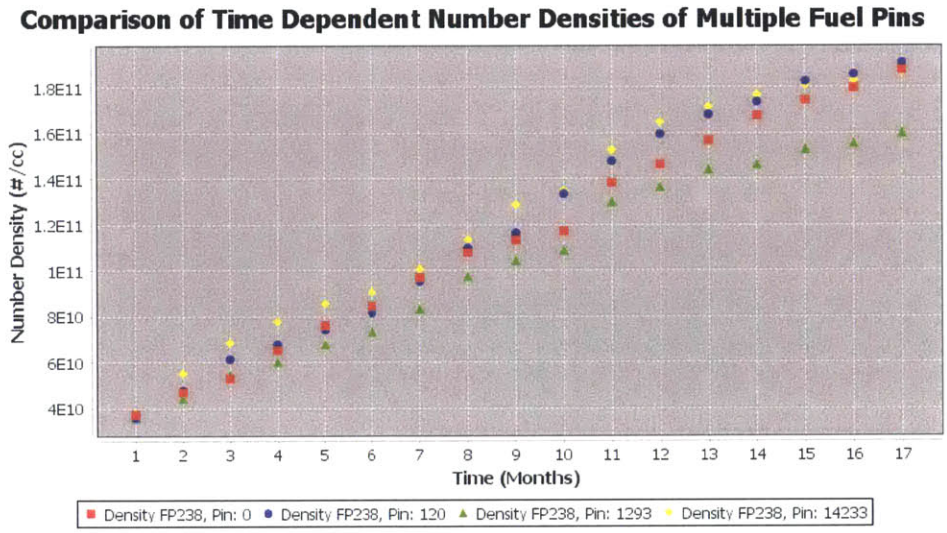


Figure 3-4: Comparison of FP238 in Depletion Experiment for 4 Fuel Pins

In all figures shown, the initial time step has been removed for illustrative purposes, as the jump up from 0 causes excessive stretching of the scale. In each of the four figures shown above, the same 4 pins are shown, with each figure showing the change in a different isotope. The figures show a clear decline in the amount of U-235 and U-238 over time, following the expected exponential curve. Additionally, the U-238 is destroyed extremely slowly; as shown in Figure 3-3, the change is so minute that even the seven significant figures shown on the axis are insufficient to

show the change. Also, as expected, both fission products only increase, as they are inert in this model for demonstrative purposes. The exponential curve for the growth of both fission product is clear, as is expected, due to the relatively rapid change. Additionally, the losses from the fuel equal the increases in the fission products.

Ultimately, while not expected to show all the complex behaviors of depletion, this proof of concept does demonstrate that the introduced framework did, in fact, provide a correct lookup mechanism useful for a real problem. Performing an identical simulation without using the physical property distribution framework produced identical results, thus serving as verification of the implementation. Furthermore, timing results are not meaningful in their current state, given both the hardware used for the simulation, a single core on a virtual machine, and the use of differential equations for the depletion calculation. Additionally, another effort using this framework has demonstrated scalability over distributed computing systems using domain decomposition[5].

3.2 Temperature Distributions

This section presents a brief synopsis of some of the work of Matthew Ellis, which is built off of this framework[10]. His work combines OpenMC with the Multiphysics Object-Oriented Simulation Environment (MOOSE)[3] framework developed at the Idaho National Laboratory. The two codes are coupled together for multiphysics feedback, with OpenMC performing on-the-fly windowed multipole based Doppler broadening for microscopic cross sections, and MOOSE performing heat conduction calculations. The simulation is done on a single assembly from the BEAVRS benchmark, as shown in Figure 3-5 In his work, temperature assignments and lookups were completed using the distribution framework presented in this paper, which was critical to its success.

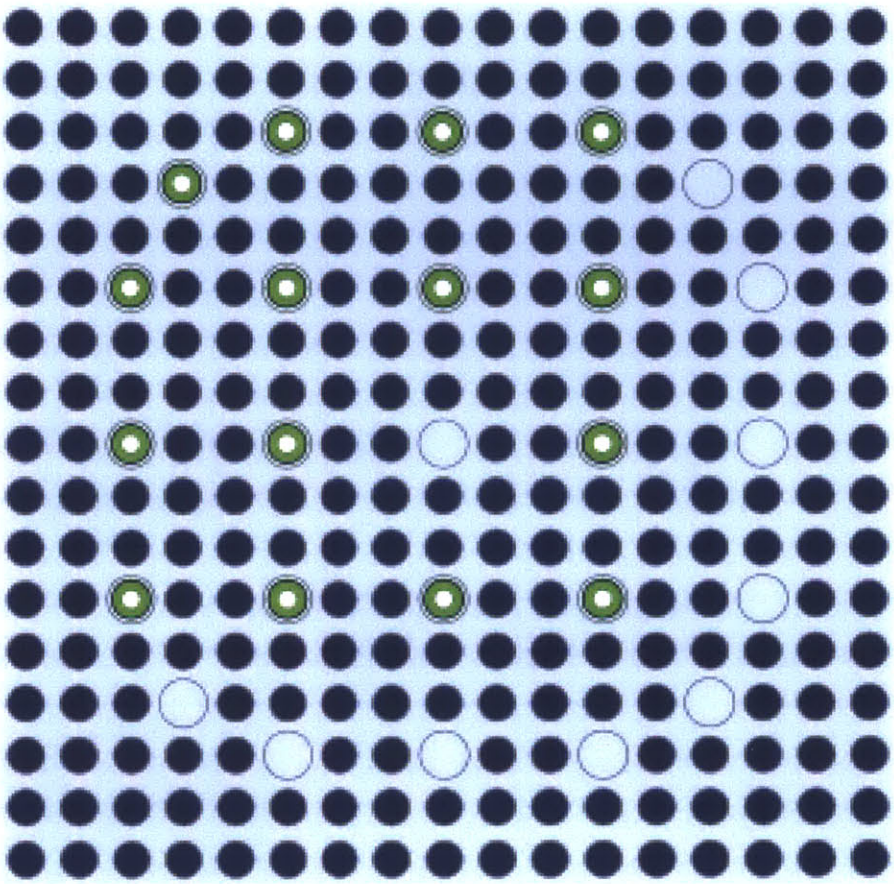


Figure 3-5: Assembly used from BEAVRS benchmark. 15 burnable poisons with 3.1% enriched fuel, colored by material.

In the OpenMC simulation, each pin is allowed to evolve in temperature independently using the distributed framework developed in this thesis. Due to the heterogeneous nature of the assembly, a nonuniform temperature is expected. A plot showing an example of the results is shown in Figure 3-6. This figure shows the temperature distribution in the assembly where the geometry and material compositions of all fuel pins are identical. The distribution algorithm allows the definition of a single material card, thus greatly simplifying the problem definition and facilitating coupling with thermal-hydraulic codes.

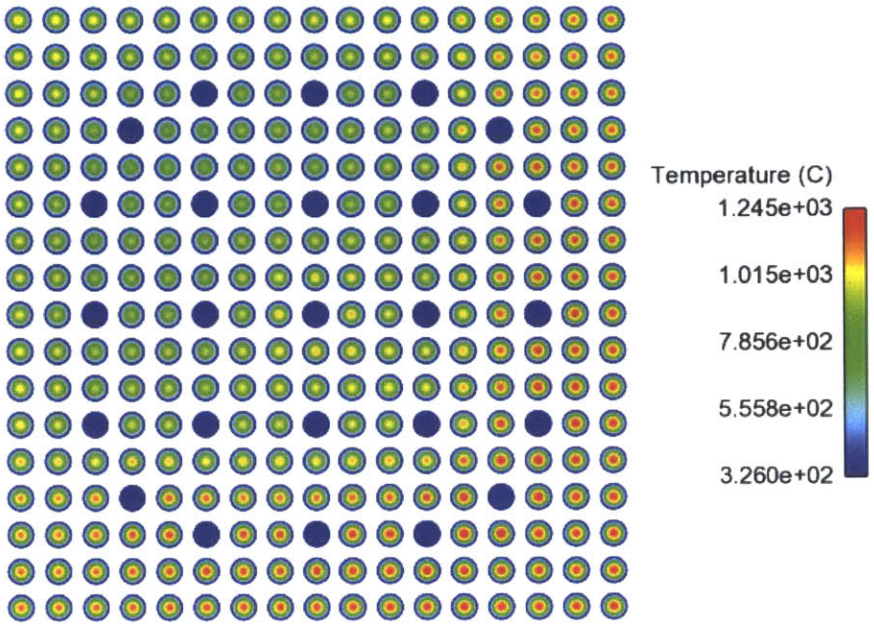


Figure 3-6: Temperature distribution for a simulation containing a unique heat conduction solution for each pin.

Chapter 4

Conclusion

Based on the successful completion of the example depletion simulation in Section 3.1 and temperature distribution in Section 3.2, the algorithms added to the OpenMC source code provide a method to efficiently index unique regions while not repeating data structures in memory. The results show a noticeable reduction in the memory used to handle geometry data, removing the need to declare the same geometric structure multiple times entirely. Additionally, this code reduces input file complexity. This thesis provides a detailed walk-through and analysis of all relevant algorithms, and the theoretical example shown in Figure 4-1 is explained in depth to show how the algorithms works on a simple case. The memory footprint is detailed, as well as the best and worse case lookup costs. For the full core BEAVRS benchmark with 10 radial rings, 10 azimuthal sectors, and 100 axial subdivisions, the memory costs to model the geometry were 60MB for the case using the distribution framework and 66900MB for the case without it.

The algorithms allow for a data structure to be mapped to a repeated structure with minimal overhead, and no additional overhead for multiple properties. The BEAVRS benchmark showcases the scalability of this method, both in the reduced memory footprint and the sample depletion case. The mapping's memory footprint for the target problem is shown to be entirely manageable for each processor on a large distributed system to store. This further improves scalability, as this reduces cross-CPU memory lookups for parallel executions.

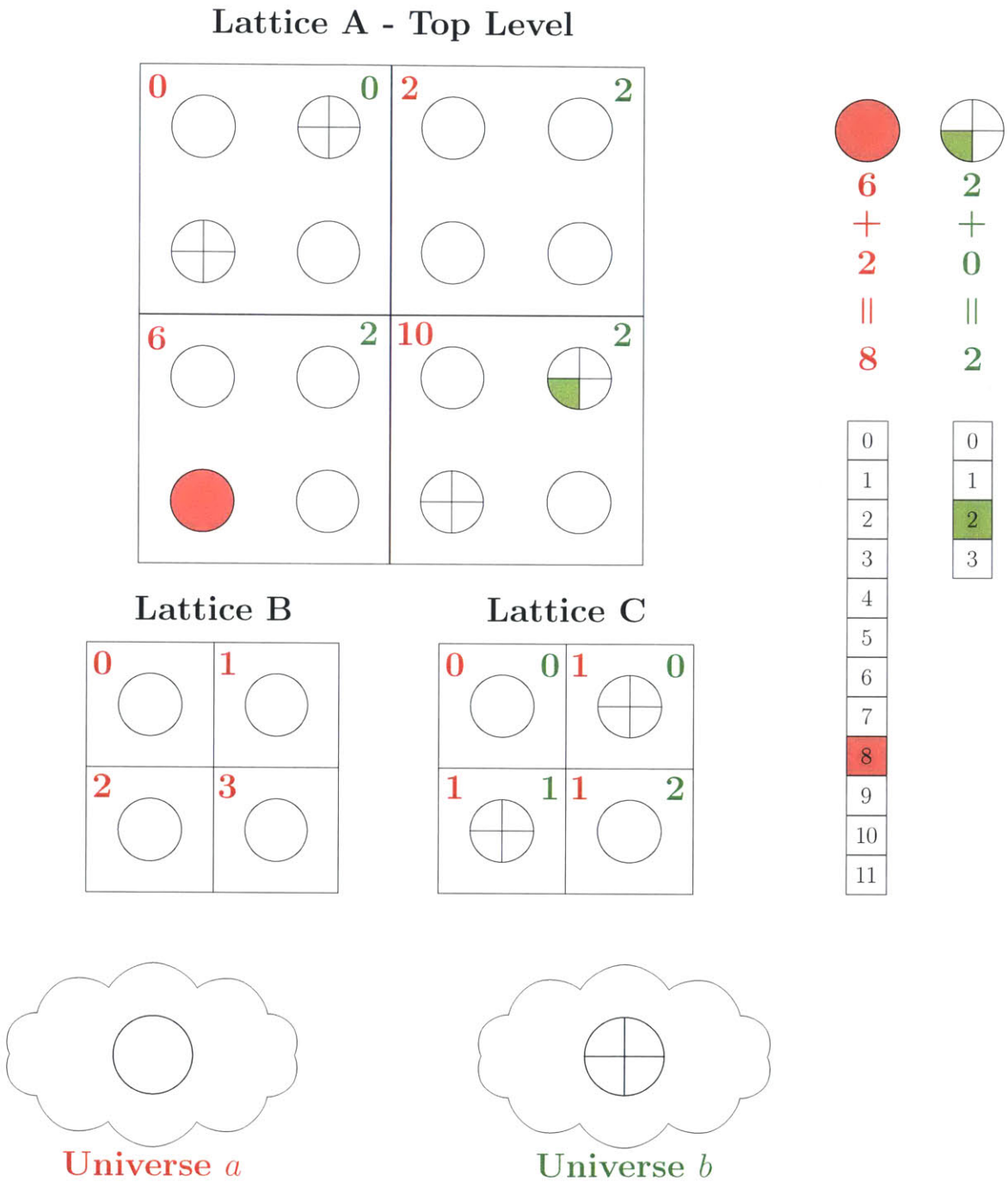


Figure 4-1: Indexing with example geometry with 3 universe levels.

The implementation in OpenMC shows large decreases in input file size and memory requirements for high fidelity simulations involving irregular geometries and geometries with non-rectangular grids. Additionally, it allows for efficient lookups of multiple properties for distributed structures. The two specific examples of physical property distribution shown are isotopic number densities and temperatures, but these are just examples. This thesis provides a general framework for distributing physical properties, which can be adapted as needed for any problem.

Currently, all demonstrated properties were stored in hard-coded variables. Future work on this framework might allow for a user-defined property to be distributed along with a geometric structure without the need for hard-coded properties. In its current state, only temperature and material composition may change uniquely, and although it is trivial for a developer to add additional properties, it is not the best approach to add all potentially useful properties to the code. Instead, the user should be able to define any property, possibly for use in conjunction with other codes or software modules. Another potential enhancement would be to allow the geometry to change uniquely like any other physical property. This would have benefits for simulations involving thermal expansion or fuel integrity.

Bibliography

[1] B. Nease, D. Millman, D. Griesheimer, D. Gill. Geometric Templates for Improved Tracking Performance in Monte Carlo Codes. *EDP Sciences*, 2014.

[2] CURIE. Public Gallery. `http://curie.ornl.gov/content/public-gallery`, 2015.

[3] D. Gaston, C. Newman, G. Hansen, D. Lebrun-GrandiÃľ. MOOSE: A parallel computational framework for coupled systems of nonlinear equations. *Nuclear Engineering and Design*, 10:1768–1778, 2009.

[4] D.P. Griesheimer, D.F. Gill, B.R. Nease, T.M. Sutton, M.H. Stedry, P.S. Dobreff, D.C. Carpenter, T.H. Trumbull, E. Caro, H. Joo, D.L. Millman. MC21 v.6.0 - A continuous-energy Monte Carlo particle transport code with integrated reactor feedback capabilities. *Annals of Nuclear Energy*, 2014.

[5] Nicholas Horelik. *Domain Decomposition for Monte Carlo Particle Transport Simulations of Nuclear Reactors.* 2015.

[6] Encyclopedia Britannica Inc. Pressurized Water Reactor. `http://www.britannica.com/EBchecked/media/160852/Section-of-a-pressurized-water-reactor-showing-inlets-and-outlets`, 2012.

[7] J. Shalf, S. Dosanjh and J. Morrison. Exascale Computing Technology Challenges. In *9th International Conference on High Performance Computing for Computational Science*, pages 1–25, Berkeley, CA, USA, 2011.

[8] DJ Kelly, TM Sutton, and SC Wilson. Mc21 analysis of the nuclear energy agency monte carlo performance benchmark problem. Technical report, American Nuclear Society, Inc., 555 N. Kensington Avenue, La Grange Park, Illinois 60526 (United States), 2012.

[9] Jaakko Leppänen. Serpent — a continuous-energy Monte Carlo reactor physics burnup calculation code, user's manual. Technical report, VTT Technical Research Centre of Finland, August 2012.

[10] M. Ellis, D. Gaston, B. Forget, and K. Smith. Preliminary Coupling of the Monte Carlo code OpenMC and the Multiphysics Object-Oriented Simulation

Environment (MOOSE) for Analyzing Doppler Feedback in Monte Carlo Simulations. In *ANS MC2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, Nashville, Tennessee, 2015.

[11] N. Horelik, B. Herman, B. Forget, and K. 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.*, Sun Valley, Idaho, 2013.

[12] Nicholas Horelik and Benoit Forget and Andrew Siegel and Kord Smith. Domain Decomposition and Terabyte tallies with the OpenMC Monte Carlo Neutron Transport Code. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, Japan, September 28 - October 3 2014.

[13] NRC. NRC Libary. `http://www.nrc.gov/images/reading-rm/photo-gallery/20071114-022.jpg`, 2007.

[14] P. K. Romano and B. Forget. The OpenMC Monte Carlo Particle Transport Code. *Annals of Nuclear Energy*, 51:274–281, 2013.

[15] Paul K. Romano and Benoit Forget and Kord Smith and Andrew Siegel. On the user of tally servers in monte carlo simulations of light-water reactors. In *Proc. Joint Int. Conf. on Supercomputing in Nuc. App. and Monte Carlo*, Paris, France, 2013.

[16] Paul K. Romano, Andrew R. Siegel, Benoit Forget, and Kord Smith. Data decomposition of monte carlo particle transport simulations via tally servers. *Journal of Computational Physics*, 252(0):20 – 36, 2013.

[17] Kord Smith. Reactor core methods. In *M&C 2003*, Gatlinburg, Tennessee, April 6–10 2003. Plenary Presentation.

[18] J. Vetter. Aspen: a performance modeling language for codesigning exascale applications and architectures. Presented to CCDSC 2012 La Maison des Contes, Lyon, France, 2012.

[19] Wikipedia. Demo of Constructive Solid Geometry. `https://upload.wikimedia.org/wikipedia/commons/8/8b/Csg_tree.png`, 2005.