LLNL-TH-647500

LAWRENCE LIVERMORE NATIONAL LABORATORY

Scalable Domain Decomposed Monte Carlo Particle Transport

M. J. O'Brien, P. S. Brantley, K. I. Joy, F. Gygi

December 10, 2013

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Scalable Domain Decomposed Monte Carlo Particle Transport

By

MATTHEW JOSEPH O'BRIEN
B.S. Computer Science (Carnegie Mellon University) 2001
M.S. Mathematics (Carnegie Mellon University) 2001

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In

APPLIED MATHEMATICS

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Prof. Kenneth I. Joy (Chair)

Dr. Patrick S. Brantley

Prof. Francois Gygi

Committee in charge

2014

Acknowledgements

I would like to thank Lawrence Livermore National Laboratory for supporting me and providing me the opportunity to complete a PhD at UC Davis, while remaining a full time employee at the lab. There was significant overlap between my job at the lab, and my dissertation subject, which is what made it possible. I would like to thank Spike Procassini who was my Livermore Lab advisor until 2011, and I would also like to thank Patrick Brantley who has been my Livermore advisor since then, when he took over leadership of the Monte Carlo code project. It has been great to work with Ken Joy, my advisor, who has always helped me out and done whatever he could to help me finish.

I want to thank my Mother and Father for always encouraging me and placing a high value on education. Finally, to my wife Jana and my daughter Nathalia, thanks for putting up with me, I love you!

Contents

Matthew Joseph O'Brien
March 2014
Applied Mathematics

Scalable Domain Decomposed Monte Carlo Particle Transport

Abstract

In this dissertation, we present the parallel algorithms necessary to run domain decomposed Monte Carlo particle transport on large numbers of processors (millions of processors). Previous algorithms were not scalable, and the parallel overhead became more computationally costly than the numerical simulation. The main algorithms we consider are:

- *Domain decomposition of constructive solid geometry:* enables extremely large calculations in which the background geometry is too large to fit in the memory of a single computational node.
- *Load Balancing:* keeps the workload per processor as even as possible so the calculation runs efficiently.
- *Global Particle Find:* if particles are on the wrong processor, globally resolve their locations to the correct processor based on particle coordinate and background domain.
- *Visualizing constructive solid geometry, sourcing particles, deciding that particle streaming communication is completed and spatial redecomposition.*

These algorithms are some of the most important parallel algorithms required for domain decomposed Monte Carlo particle transport. We demonstrate that our previous algorithms were not scalable, prove that our new algorithms are scalable, and run some of the algorithms up to 2 million MPI processes on the Sequoia supercomputer.

1. Introduction

This dissertation investigates how to design scientific computer modeling software to run efficiently on a very large supercomputer. Let us first consider the case of a single, *serial* computer that has just one processor. Given a scientific modeling program, the user typically desires simulations to run faster. As new computers get faster, the simulation automatically runs faster when run on the new computers. In this case, there is no need to rewrite any of the program, it just runs faster because the computer hardware gets faster.

Now let us consider a *parallel* computer that has more than one processor. The computer simulation has to be specifically written to divide the work among the processors, do some independent work, and combine the results at the end to obtain the final answer. As the number of processors on a new parallel computer increases, the simulation runs faster since the work is divided among more processors. Again, we do not have to rewrite any of the computer program to take advantage of the bigger supercomputers that have more processors.

But this mode of parallel scalability eventually brakes down. Eventually the *overhead* of dividing up the work among processors and communicating results back and forth to get the total answer, takes longer than the numerical computation portion of the calculation. When the number of processors gets too large, the management process becomes more expensive than the actual work. An analogy might be a supervisor who has two direct reports. In this case, the supervisor instructs each worker who in turn reports back to him. The supervisor then combines their results. This model may work for tens of direct reports. Now imagine the supervisor having one million direct reports! Clearly, there is not enough time in the day to talk to each worker individually and to combine their results. One solution is to impose a type of tree hierarchy, so that each supervisor only deals with a handful of direct reports. This same strategy works with computer algorithms. Imposing a tree structure on top of how the work is divided up and combining the results can significantly speed up the simulation.

The total “management overhead” time now becomes proportional to the depth of the tree. The depth of the tree is how many steps it takes to send a message to every employee if it comes from the top level CEO and each person delivers the message only to their direct reports. The depth of the tree is proportional to the logarithm of the total number of employees. The logarithm function grows *very* slowly, so the management overhead grows very slowly. If the number of employees is doubled, then only one extra layer is added at the bottom of the tree. This tree-based approach is much more efficient than one CEO having to deal with all of the employees. In that case, doubling the number of employees doubles the amount of management overhead, which is too costly to scale to large numbers of employees.

In order to run physics simulations on new supercomputers with hundreds of thousands or millions of processors, care must be taken to implement scalable algorithms. This means the algorithms must continue to perform well as the processor count significantly increases. This requirement rules out “global algorithms” that have a view of some information from every processor. Global algorithms need to be replaced with local algorithms that only communicate with nearest neighbors or some other small set of processors. In this dissertation, we examine some of the scalable algorithms necessary for Monte Carlo particle transport. Monte Carlo particle transport is a type of physics simulation that statistically models “in flight” particles interacting with a background material. The algorithm is called “Monte Carlo” in reference to the Monte Carlo Casino in Monaco and was first invented in the late 1940s at Los Alamos National Laboratory [1]. Monte Carlo uses random numbers to sample from statistical distributions to model the random nature of particle interactions [2], [3].

On a supercomputer that has up to 10,000 processors, it is feasible to both store data that is proportional to the total number of processors and write an algorithm whose run time is proportional to the number of processors. These algorithms consume a small enough fraction of the overall compute time to be in the "noise" and work quite well. Now that supercomputers may have up to millions of processors, these algorithms can become even more expensive (time-consuming) than the numerical computation of an application. So we must revisit the algorithms that have a strong dependence on the total number of processors. For related information, see [4].

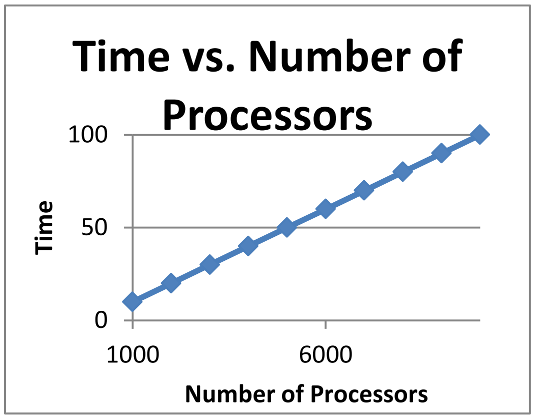
Table 1 shows some of the modern supercomputers from Lawrence Livermore National Laboratory (LLNL). Efficient use of the supercomputers with lower processor counts such as Blue, White and Purple could be achieved without implementing scalable algorithms. But for BlueGene/L [5], Dawn [6] and Sequoia [7], the processor counts are so large that they require scalable algorithms to use efficiently.

Table 1: Modern supercomputing history at LLNL.

Date	Computer	Number of Processors
1998	Blue	5.856
2001	White	8,192
2005	Purple	12,544
2004	BlueGene/L	65,636
2009	Dawn	147,456
2012	Sequoia	1,572,864

We define an algorithm to be *scalable* if its runtime is $O(N^{\varepsilon})\ \forall\, \varepsilon > 0$, where N is the number of processors. This definition rules out algorithms whose runtime is proportional to the number of processors, and even rules out runtimes proportional to $\sqrt{N}$ or $\sqrt[3]{N}$. Examples of algorithms that meet this definition have runtimes proportional to *log(N)* or powers of *log(N)*. However, a practical problem exists with this definition. This definition is valid for large *N*, as *N*→∞, but we always have a finite number of processors. At a particular processor count, an algorithm with run time proportional to *log(N)* may in fact take longer than an algorithm whose runtime is proportional to $\sqrt{N}$. So the question of whether an algorithm is scalable is subjective to some extent and depends on how much slowdown is acceptable, as the processor count increases.

When considering *weak scaling* problems for which there is constant work per processor, a scalable algorithm's runtime will be relatively flat as a function of processor count. See Figure 1 for sample graphs of non-scalable (on the left) and scalable (on the right) results. The non-scalable graph on the left shows an algorithm with a run time that is proportional to the number of processors and increases significantly with the number of processors. The scalable example on the right shows an algorithm with a run time that is proportional to the log of the number of processors and is relatively flat as the number of processors increases.



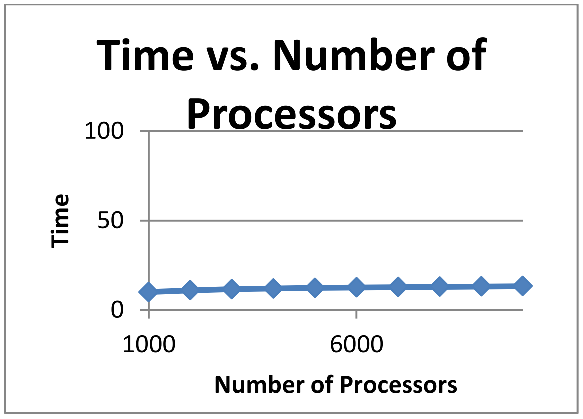


Figure 1: Runtime of non-scalable (left) and scalable (right) algorithms.

A common approach to making an algorithm scalable is to introduce some type of tree-based communication structure. If in the original algorithm all of the processors communicated with the 0th ranked processor, this algorithm will not scale. To make the algorithm scalable, it would be converted to send messages up and down a tree. Each processor then has to communicate only with a *fixed* number of other processors, for example one parent and two children in a binary tree. As the number of processors increases, the *depth* of the tree increases, but very slowly, proportional to *log(N).* So the cost of sending messages up and down the tree is *O(log(N)),* which is scalable.

1.1. Mercury Monte Carlo Particle Transport Code

Mercury [8], [9], [10], [11], [12], [13], [14] is LLNL's modern Monte Carlo particle transport code. Mercury solves static and dynamic neutron transport problems and eigenvalue criticality problems. It also has gamma transport and charged particle transport physics. Mercury is written in C++, has a python [15], [16], [17], [18] user interface, is massively parallel, uses MPI [19], [20] for distributed memory parallelism and OpenMP [21] for shared memory parallelism. Mercury uses the Silo [22] and HDF5 [23] I/O libraries for graphics and restart files. Mercury is parallelized via domain decomposition and domain replication and allows for a hybrid parallelization combining MPI, OpenMP, domain decomposition, and domain replication.

Mercury solves the linear Boltzmann particle transport equation for neutrons:

$$\begin{aligned}
&\frac{1}{v(E)}\frac{\partial}{\partial t}\psi\left(\vec{r},\hat{\Omega},E,t\right)+\hat{\Omega}\cdot\vec{\nabla}\psi\left(\vec{r},\hat{\Omega},E,t\right)+\Sigma_t\left(\vec{r},E,t\right)\psi\left(\vec{r},\hat{\Omega},E,t\right)=Q\left(\vec{r},\hat{\Omega},E,t\right)\\
&+\int_0^\infty dE'\int_{4\pi}d\hat{\Omega}'\Sigma_s\left(\vec{r},\hat{\Omega}'\cdot\hat{\Omega},E'\to E,t\right)\psi\left(\vec{r},\hat{\Omega}',E',t\right)\\
&+\frac{1}{4\pi}\int_0^\infty dE'\chi\left(\vec{r},E'\to E,t\right)\nu\Sigma_f\left(\vec{r},E',t\right)\int_{4\pi}d\hat{\Omega}'\psi\left(\vec{r},\hat{\Omega}',E',t\right)
\end{aligned}$$

This equation has a seven-dimensional phase space: three spatial: $\vec{r} = (x, y, z)$, two directional: $\Omega = (\theta, \phi)$, one energy: E, and one time: t. Each simulation particle carries these seven attributes along with a *weight* that represents the number of physical particles. Simulation particles carry several other attributes to facilitate the computation. The Monte Carlo method does not discretize this equation using finite differences or finite elements; instead it uses random numbers to sample from distributions to statistically simulate the particle interactions. The background geometry of the problem is specified using Constructive Solid Geometry (CSG) which will be described in detail in Chapter 2.

We have run Mercury up to 2^{21} = 2,097,152 MPI processes on the IBM BG/Q *Sequoia* supercomputer and observed scalable results that agree with our theoretical predictions. In this dissertation, we will explain why previous Mercury algorithms were not scalable and prove why the new algorithms are scalable.

1.2.Related Work

The Radiation Safety Information Computational Center (RSICC) is one of the world's leading resources for a broad range of the best available nuclear computational tools and services [24]. For an overview of Monte Carlo codes managed by RSICC, see the paper [25]; Mercury has a similar set of features but is not currently released under RSICC. MCNP (Monte Carlo N-Particle) [26] is the most well known related Monte Carlo particle transport code. MCNP has been developed for decades at Los Alamos National Laboratory. MCNP has a very large user base and has been thoroughly verified and validated. MC21 [27] is another modern Monte Carlo transport code being developed jointly at the Knolls Atomic Power Laboratory and the Bettis Laboratory. GEANT (for GEometry ANd Tracking) is a modern Monte Carlo code developed at

CERN [28]. TART [29] and COG [30] are legacy Monte Carlo particle transport codes developed at Lawrence Livermore National Laboratory, both written in Fortran. OpenMC [31] is a modern Monte Carlo transport code originally developed at the Massachusetts Institute of Technology. The code also addresses parallel challenges faced with large processors counts, but has a different set of constraints, see [32]. For reproducibility, each run of OpenMC processes particles in exactly the same order. Mercury, on the other hand, does not obey that constraint which enables a more efficient load balancing algorithm as discussed in Chapter 4. Serpent [33] is a continuous-energy Monte Carlo reactor physics burn up calculation code developed at VTT Technical Research Centre of Finland . Table 2 below summarizes some of the related Monte Carlo particle transport codes. Many other codes exist, so this list is not exhaustive.

Table 2: Table of related Monte Carlo particle transport code.

Code	Developed At	Comments
MCNP	Los Alamos National Laboratory	Large user base, lots of verification and validation. Legacy architecture.
TART	Lawrence Livermore National Laboratory	Legacy MC code at LLNL.
COG	Lawrence Livermore National Laboratory	Legacy MC code at LLNL.
MC21	Knolls Atomic Power Laboratory and the Bettis Laboratory	Modern MC code.
Serpent	VTT Technical Research Centre of Finland	MC reactor physics burn up code.
GEANT	CERN	High energy MC code.
OpenMC	Massachusetts Institute of Technology	Modern MC code. Written in FORTRAN.
Mercury	Lawrence Livermore National Laboratory	Modern MC code, written in C++ with a python user interface, scalable.

1.3. Overview of this Dissertation

In Chapter 2 we describe how to domain decompose Constructive Solid Geometry (CSG) for Monte Carlo particle transport. Typically CSG is *replicated* on every processor and the particles are distributed over the processors. Replication is generally preferred over domain decomposition since it is easier to implement (particle streaming communication does not have to be implemented) and can be faster. But replication limits the total geometry size to be small enough to fit in memory of a single compute node. Domain decomposition solves this problem by distributing the geometry over processors. This decomposition enables truly large geometry models, limited only by the amount of memory in the entire supercomputer. Domain decomposition has been used to run a problem with 5.6 million cells, and visualize a model of the city of Boston with 89 million cells.

Chapter 3 describes a domain decomposed load balancing algorithm. This novel algorithm has significantly improved the performance of almost all of our calculations. Some aspects of this algorithm are not scalable and remain as future areas of research. The algorithm has been run successfully up to 65,536 processors.

Chapter 4 describes a novel load balancing algorithm that we prove has $O(log(N))$ communication steps, where N is the number of processors. We present a scaling study of this algorithm up to 2^{21} = 2,097,152 MPI processes on the IBM BG/Q *Sequoia* supercomputer. The observed performance is in excellent agreement with our theoretical predictions.

Chapter 5 examines the problem of globally resolving particle locations to the correct processor. Particles may be *sourced* (created) on a processor that does not own the background geometry for the particle's coordinate. The particles must be communicated to the correct processor before they can track through the background geometry. This chapter shows how we create a hypercube network to efficiently communicate particles to the correct processor.

Chapter 6 considers the problem of visualizing constructive solid geometry. We show how Mercury is coupled to VisIt through both files and an inline interface for visualizing a calculation as it is running. We implement several methods for converting CSG to a mesh, where the mesh is then used to either visualize the CSG or used directly for the particle transport calculation. Mercury is also used directly as a ray-caster to visualize CSG directly.

Finally Chapter 7 examines other parallel algorithms required for Monte Carlo particle transport. First we consider sourcing particles scalably. We implement scalable particle sourcing by *uniformly* distributing source particles over *all* of the processors. This approach may mean that some particles are sourced on the *wrong* processor. We then simply rely on the global particle find algorithm from Chapter 5 to communicate particles to the correct processor. We also run a domain decomposed test problem on 2^{21} = 2,097,152 MPI processes on Sequoia demonstrating the scalability of the particle streaming communication and the "test for done" algorithm. Then we demonstrate the flexibility of the geometry available in Mercury by tracking to arbitrary surfaces defined by C-functions written in the user's input. Finally we investigate *spatial redecomposition* and *domain-to-processor assignment.* We show that both of these algorithms can have a large impact on the efficiency of a calculation, and more research should be performed to further understand and optimize these algorithms.

2 Domain Decomposed Constructive Solid Geometry

Constructive Solid Geometry (CSG) is a way of combing surfaces to form cells, and then using mathematical set operations such as *union*, *intersection* and *negation* to aggregate cells to form more complex cells. The CSG forms the background geometry that particles interact with. The user specifies the material properties of each cell, such as which isotopes is it made of, the density of the material and the temperature of the material. CSG is also known as Combinatorial Geometry (CG).

Domain decomposition has been implemented in Mercury, a CSG Monte Carlo neutron transport code [34], [35]. Previous methods to parallelize a CSG code relied entirely on *particle parallelism* [36]. In our approach, we distribute the geometry as well as the particles across processors. Domain decomposition enables calculations whose geometric description is larger than what could fit in the memory of a single processor. In addition to enabling very large calculations, we show that domain decomposition can speed up calculations compared to particle parallelism alone. We also show results of a calculation of the proposed Laser Inertial-Confinement Fusion-Fission Energy (LIFE) facility, which has 5.6 million CSG cells.

2.1 Introduction

Previous methods of parallelizing a CSG Monte Carlo neutron transport code have implemented a method know as *particle parallelism*, meaning that the geometry information is redundantly stored on *all* of the processors, while the particle workload is divided among the processors. This method is “embarrassingly parallel” in the sense that the processors can run independently of each other, until the end of the calculation, when a total answer is calculated that is the sum of all of the processors’ results.

Particle parallelism is in contrast to *domain decomposition*, where the geometry is partitioned into *domains* which are assigned to processors. As a particle streams from one domain to another, it must be communicated from one processor to another. The technique of domain decomposition is commonly used in parallel finite-difference or finite-element physics simulations running on a computational mesh. Well known techniques exist for partitioning a mesh into domains. The contrast here is that we do not have an underlying "mesh", we have a CSG (surfaces and cells) representation of the geometry.

We calculate a bounding box for every CSG surface and cell, and use the bounding box to decide if a given CSG surface or cell should exist on a given domain. The user specifies a Cartesian domain decomposition of their problem by defining the positions of decomposition planes normal to the three coordinate axes. Thus only *local* geometry information is stored on each domain producing a *scalable* algorithm.

Alme, Rodrigue, and Zimmerman [37] also investigated domain decomposition and load balancing for Monte Carlo applications. Their work addressed domain decomposition on a *mesh* as opposed to *constructive solid geometry* that we consider here. They also considered *static* load balancing in which a work estimate is used to determine the number of times a domain is replicated relative to the others. In Chapter 3, we will consider *dynamic* load balancing in which the number of times a domain is replicated may change each cycle in response to the particle workload.

The remainder of this chapter is organized as follows. Section 2.2 describes constructive solid geometry and how Mercury uses CSG to model background geometry. Section 2.3 delineates some distinctions between *mesh* domain decomposition vs. *CSG* domain decomposition. Section 2.4 describes the algorithms necessary to implement domain decomposition of CSG. Section 2.5 shows the results of running a *criticality of the world* test problem on various numbers of domains and getting statistically equivalent results, demonstrating the correctness of the domain decomposition implementation. Section 2.6 describes a LIFE test problem that is so large that domain decomposition is required. Section 2.7 describes an 89 million cell test problem of the city of Boston that we visualize using domain decomposition. Finally Section 2.8 discusses the conclusions of this chapter.

2.2 Constructive Solid Geometry

In our implementation of CSG, we implement *quadric surfaces*, which are at most 2nd order surfaces, such as planes, spheres, ellipsoids, cylinders, cones, etc. We also support general 4th order quartic surfaces, the most common being the torus. These surfaces are stored as a list of the coefficients in the implicit equation satisfied by the points on the surface:

$$f(x,y,z) = \sum_{\substack{0 \le i+j+k \le 4 \\ i,j,k \ge 0}} a_{ijk}(x-x_0)^i (y-y_0)^j (z-z_0)^k = 0$$

For example, a plane parallel to the x-axis is represented as

$$a_{100}x + a_{000} = 0$$

and a sphere is represented as

$$(x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2 + a_{000} = 0$$

where $a_{200} = a_{020} = a_{002} = 1, a_{000} < 0$, and all the other coefficients are 0.

The surfaces are used to define volumes by considering the points such that $\{(x, y, z) : f(x, y, z) < 0\}$ (for example, inside of a sphere) and $\{(x, y, z) : f(x, y, z) > 0\}$ (for example, outside of a sphere). The volumes are then combined using logical operations such as *AND, OR, NOT* to form more complex volumes. We call the volumes *CSG cells* or *cells*.

Example. Here we define two spherical surfaces, *sphere1* and *sphere2*. We then define *cell1* to be:

cell1 = insideOf(sphere1) AND outsideOf(sphere2)

See Figure 2 to see what cell1 looks like.

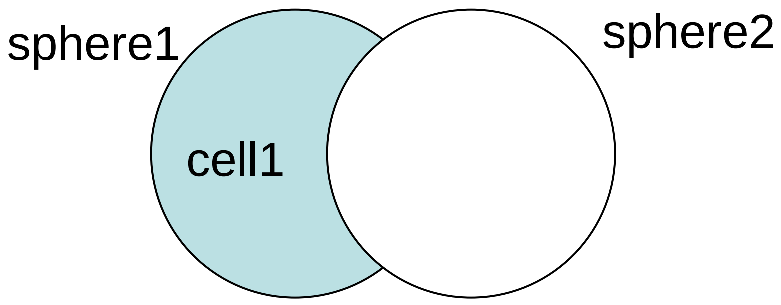


Figure 2: A simple example of creating a CSG cell that is inside the sphere1 surface and outside the sphere2 surface.

Using only these simple primitives, one can construct very complicated geometries. For example, in the Figure 3 below, the National Ignition Facility (NIF) (explained in Section 2.6) target chamber and support structures are modeled with CSG.

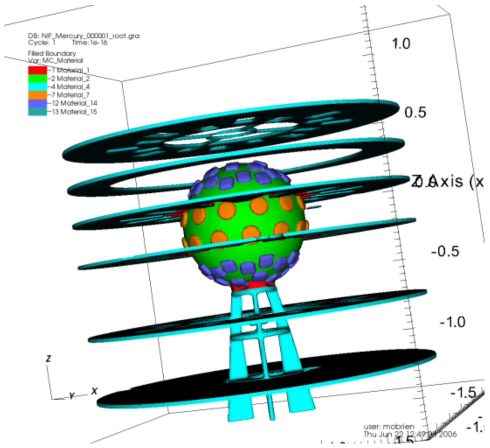
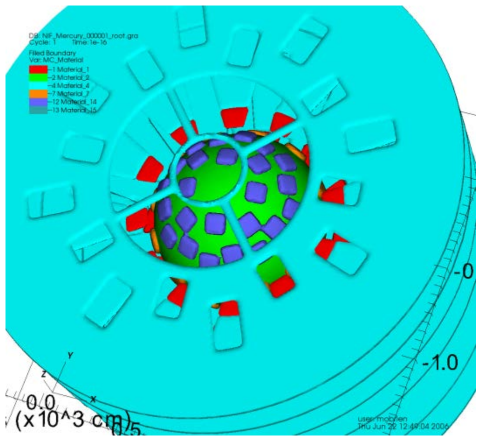



Figure 3: The NIF target chamber and support structures as modeled with CSG.

2.3 Domain Decomposition: Mesh vs. CSG

We would like to draw some distinctions between *mesh* domain decomposition and *CSG* domain decomposition. In the case when the underlying discretization of the geometry is *mesh-based*, part of the description of the mesh is the *connectivity* of the mesh cells. If the mesh is topologically Cartesian, then the connectivity of the mesh is implicitly known by using indexing and striding to move in the *i*, *j*, or *k* directions. If the mesh is unstructured, then a data structure is required that provides the face neighbors of every face of every zone. This data structure creates an underlying graph that is partitioned into domains.

Let G=(V,E) where

V = {the set of cells in the problem}, and

E={(c_1, c_2) : if cell c_1 is a face neighbor of cell c_2.}

In the case that the underlying discretization of the problem geometry is CSG-based, then no connectivity information about the adjacency of any of the CSG cells is known. As a particle exits a bounding surface of one cell and enters an adjacent cell, (a priori) the particle does not know what cell it will enter. The algorithm dynamically *learns* the connectivity of the cells as particles track through the cells. The first time a particle exits a cell by crossing a bounding surface, the algorithm loops over all other cells and asks the question "is this point in the given cell". Then the adjacent cell is saved in a connectivity table to be checked on subsequent particle surface crossings.

Thus at initialization time, when it is time to do the domain decomposition, we do not know any connectivity information about the CSG cells. No underlying cell-face-neighbor adjacency graph exists, so we cannot use graph partitioning to do the domain decomposition. Instead we use a technique that relies on the geometric position and extent of each cell by calculating a bounding box for each cell. The domains are themselves "boxes", since they are created from the Cartesian product of boundary planes normal to each of the three coordinate axes. As a result, the test for membership of a cell within a domain is a simple axis-aligned box-box intersection test. See Table 3 for a comparison of Mesh vs. CSG domain decomposition.

Table 3: This table compares the information at hand and underlying algorithms for mesh-based vs. CSG-based domain decomposed particle tracking.

	Mesh	**CSG**
Cell Boundary Crossing	Adjacent cells know, cell faces are 1-to-1.	Must check adjacent candidate cells, do not explicitly know adjacency.
Domain Boundary Crossing	Adjacent domains known.	Adjacent domains known. (new)
Input	Input description is already domain decomposed.	Must decide if each surface/cell should be assigned to each processor. (Need to domain decompose user input.) (new)
Output (graphics)	Each processor writes its domains. A master file describes how to assemble the pieces.	Each processor writes the portion of space it owns, explicitly introducing domain boundary surfaces for cells on domain boundaries. A master file describes how to assemble the pieces. (new)

Figure 4 shows an example of CSG domain decomposition:

Upper left: The user defines the global CSG problem, without regard to domain decomposition.

Upper right: The user defines the Cartesian domain decomposition by specifying the positions of axis aligned planes normal to the three coordinate axes. The code automatically calculates bounding boxes for all of the cells which are used to test for intersection with each domain. For example, one small orange sphere has a bounding box that intersects *both* Domain 0 and Domain 1, so that cell is assigned to *both* domains.

Lower left: The code automatically creates Domain 0, and assigns the correct cells to it.

Lower right: The code automatically creates Domain 1, and assigns the correct cells to it.

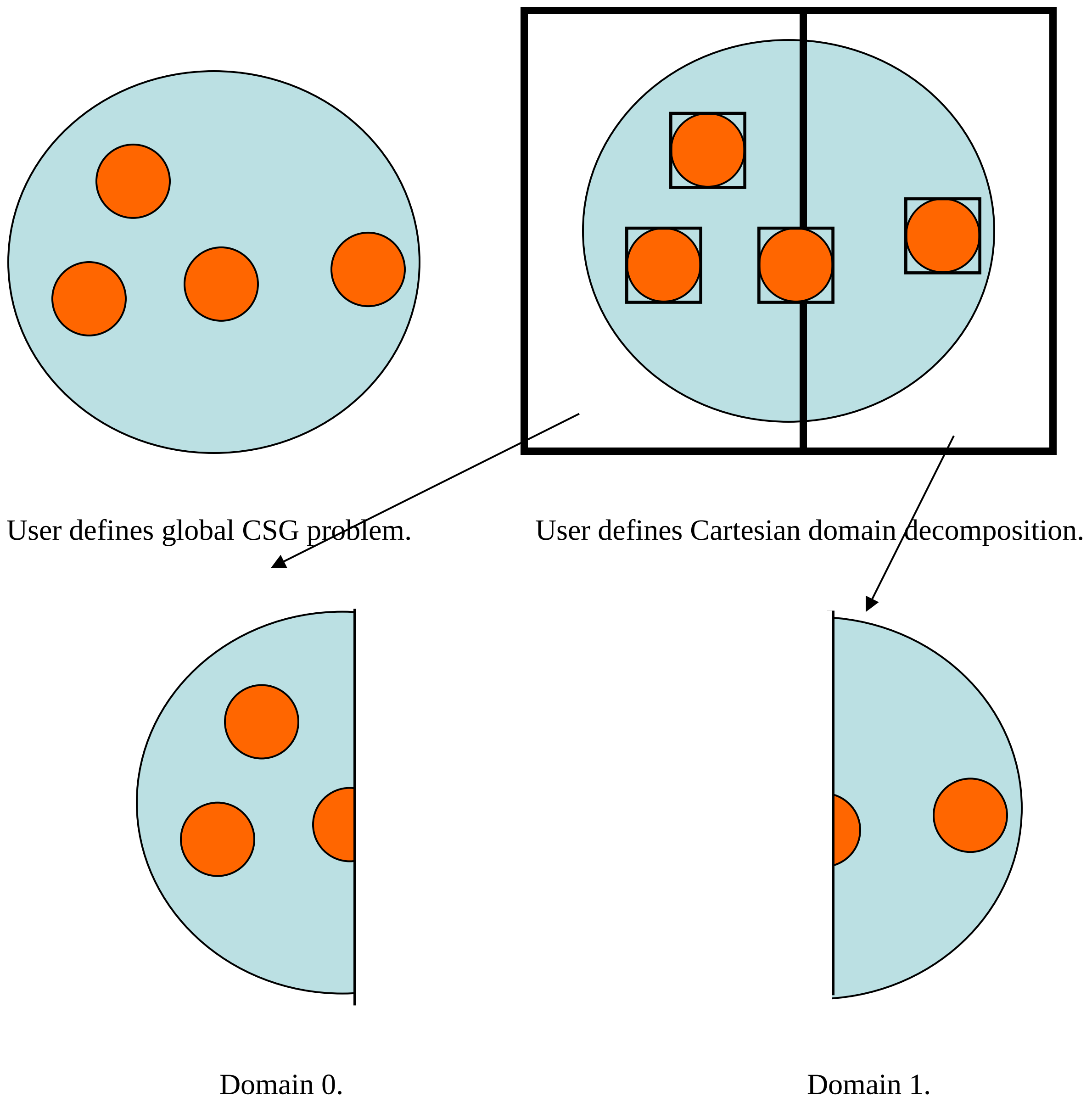

User defines global CSG problem.
User defines Cartesian domain decomposition.
Domain 0.
Domain 1.

Figure 4: Illustration of CSG domain decomposition.

Figure 5 shows CSG cells colored by domain. This problem is decomposed into 16 domains. We see examples of cells that are on 2 adjacent domains, part of the cell is one color (for one domain) and the other part of the cell is another color (for the other domain the cell is on).

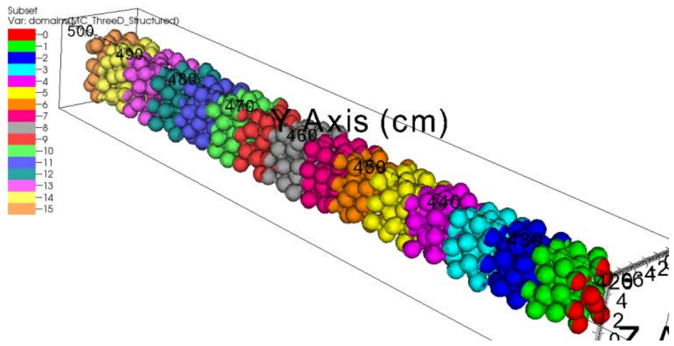


Figure 5: This problem has 16 domains, the CSG cells are colored by domain number.

2.4 Domain Decomposition of CSG

We started with an existing CSG Monte Carlo transport code that already had *mesh* domain decomposition. We leveraged the *particle streaming* communication already implemented in the mesh domain decomposition, to use with the new CSG domain decomposition. *Particle streaming* communication is the MPI communication that happens when particles cross a domain boundary and need to be sent to an adjacent domain on another processor to continue tracking on the other processor.

2.4.1 What is Distributed Across Domains

As the geometric description of a problem gets larger and larger, the following lists of data can grow arbitrarily long:

— List of surfaces.
— List of surfaces that define a cell.
— List of cells.
— List of *templated* (cloned) surfaces and cells.

As a result, we need a way to distribute this data across processors.

Every object in a CSG problem is defined by operations on surfaces, so the total number of surfaces can be very large. Rather than storing the entire list of surfaces redundantly on every processor, we must only store the *local* surfaces whose bounding box intersects the bounding box of a domain. The same is true for the CSG cells in the problem: each processor only stores local cells according to the portion of space that it owns.

Mercury has a user interface feature called ***templates*** which is a way of dealing with repeated structures. A user defines a template to be a list of surfaces and cells, and then instantiates the template as many times as they would like, each instantiation having a different translation and/or rotation. For example, a user could create a template of a "house" and then instantiate and translate a house template several times to create a neighborhood. This list of templates can also get very large, so we calculate bounding boxes for templates and only instantiate them on domains whose bounding box intersects the template's bounding box.

2.4.2 Scalability Issues

In the case of a mesh, the initial geometry conditions come from a mesh generator and are already domain decomposed into separate files. Domains are assigned to processors, and each processor only knows about its local domains. No single processor ever knows about the *global* description of the geometry. This situation is in contrast to the CSG, where a user must setup the problem geometry using input commands that define *all* of the surfaces and cells for the *entire* problem. Part of the domain decomposition algorithm takes the global CSG problem description, and each processor filters out parts of the geometry that it does not own.

The scalability issues occur only at initialization time and are:

- The entire CSG input text file must be read into memory at once.
- The entire list of surfaces/cells is read in, then a surface/cell is kept on a domain only if the surface's/cell's bounding box intersects the domain's bounding box.

- The entire list of surfaces that define a cell are read in, then only the surfaces that intersect the domain that the cell is on are kept.

2.4.3 Scalability Solutions

After initialization, each domain only stores *local* information; hence the algorithm is scalable. The only problem we have to solve is "How do we initialize the CSG geometry *locally*, so each processor only has to deal with local geometry and not *all* of the geometry?"

We could treat CSG input similar to how mesh geometry is treated: the geometry is decomposed into separate files and each processor only deals with the domains that are assigned to it. We have not yet implemented this solution. This has the disadvantage of requiring more work of the user. The user would have to split up their CSG input file into several files, each file containing geometry in some specified bounding box.

If the large cell count arises due to repeated hierarchical structures, we achieve scalability through the input "*template*" mechanism. For example, let's say we want to model a city made of 1,000 houses. We create a template of a house, which has let's say 2,500 cells. Each CSG cell requires about 7 Kilobytes of memory. So the total memory requirement is:

(1,000 houses/city) * (2,500 cells/house) * (7KB/cell) = 17.5GB/city.

17.5GB is more memory than could typically fit on any single processor. Using domain decomposition, we can distribute the geometry across processors and run the entire problem. Input templates are only instantiated on processors that contain domains that intersect the template's bounding box, so we have good input scalability using input templates.

2.4.4 Algorithms: Calculating a Cell's Bounding Box

We need to calculate a bounding box for both CSG surfaces and cells. Our surfaces are typically quadric surfaces (although we do support 4th order surfaces), specified by coefficients a_{ijk}*, such that* $i,j,k \geq 0$ *and* $0 \leq i+j+k \leq 4$*; and a translation* (x_0, y_0, z_0). These surfaces are created from user input, where the user specifies the *type* of surface:

Plane_X, Plane_Y, Plane_Z, Plane, Sphere, Ellipsoid, Cylinder_X, Cylinder_Y, Cylinder_Z, Cylinder, Cone_X, Cone_Y, Cone_Z, Cone, etc.

In addition to storing the surface coefficients and translation, we also store an enumerated type describing the type of the surface. Given the type of the surface, we can calculate its bounding box. For example, a plane normal to the X-axis, has a surface equation

$$x + a_{000} = 0$$

We store axis aligned bounding boxes which are specified by the minimum and maximum coordinates, in this case

$$Min = (-a_{000}, -\infty, -\infty) \quad Max = (-a_{000}, \infty, \infty)$$

Note that we allow for infinite extent in any or all of the coordinate directions. In particular, we could have an unbounded surface (for example, a plane that is not normal to any of the coordinate axes). When an unbounded surface is intersected with *any* domain, there will always be an intersection. As a result, unbounded surfaces will be assigned to *all* processors.

Another example bounding box calculation is that of a spherical surface that has the surface equation:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 + a_{000} = 0$$

The axis aligned bounding box is given by:

$$Min = (x_0 - \sqrt{-a_{000}}, y_0 - \sqrt{-a_{000}}, z_0 - \sqrt{-a_{000}})$$
$$Max = (x_0 + \sqrt{-a_{000}}, y_0 + \sqrt{-a_{000}}, z_0 + \sqrt{-a_{000}})$$

After every surface has an axis aligned bounding box, we use the bounding box to filter out non-local surfaces. Every domain also has a bounding box, so each domain only keeps the surfaces whose bounding boxes intersect the domain's bounding box.

CSG cells are built from surfaces, and we calculate the cell bounding boxes from the surface bounding boxes. A CSG cell is recursively defined as a tree of CSG cells, with an operator defined on the children of a parent cell. There are two binary operators, ***and, or,*** and one unary operator, ***not.*** We classify CSG cells as either *parent* or *leaf* cells. Parent cells have children, leaf cells do not. Below is pseudo-code for the recursive algorithm to calculate a CSG cell's bounding box:

```
CalculateBoundingBox(cell)
{
  if ( cell.isLeaf )
  {
    if ( cell.UnaryOperator == NOT )
    {
      return InifinteBoundingBox
    } else {
      // calculate cell’s bounding box
      return boundingBox
    }
  }
  else if ( cell.isParent  )
  {
    if ( cell.LeftUnaryOperator == NOT )
    {
      leftBBox  = InifinteBoundingBox
    } else {
      leftBBox  = CalculateBoundingBox(cell.leftChild)
    }
    if ( cell.RightUnaryOperator == NOT )
    {
      rightBBox = InifinteBoundingBox
    } else {
      rightBBox = CalculateBoundingBox(cell.rightChild)
    }

    if ( cell.operator == OR )
    {
      boundingBox.min = MIN(leftBBox.min, rightBBox.min)
      boundingBox.max = MAX(leftBBox.max, rightBBox.max)
      return boundingBox
    }
    else if ( cell.operator == AND )
    {
      boundingBox.min = MAX(leftBBox.min, rightBBox.min)
      boundingBox.max = MIN(leftBBox.max, rightBBox.max)
      return boundingBox
    }
  }
}
```

If a cell is bounded, then not(cell) is unbounded; thus we return an infinite bounding box for that case.

2.4.5 Algorithms: Cell Parsing

We implement simple filtering when parsing in the CSG cells from the user input file. We have a Cartesian domain decomposition, so every domain has an axis aligned bounding box. We use the above bounding box algorithm to calculate a bounding box for each cell. Each domain inserts a cell into its list of cells if the cell's bounding box intersects with the domain's bounding box. Cells that straddle domain boundaries will be inserted into multiple domains. Below is pseudocode for parsing in cells and assigning them to domains:

```
foreach (input file cell)
{
    temp_cell = inputFile.ParseCell(input file cell)
    CalculateBoundingBox(temp_cell)
    foreach (domain on this processor)
    {
        bool on_domain = domain.IsCellOnDomain(temp_cell)
        if ( on_domain )
        {
              domain.InsertCell(temp_cell)
        }
    }
}
```

2.4.6 Algorithms: Locate Coordinate

One of the most fundamental algorithms that a Monte Carlo transport code must implement is: "given a point in space, which cell is the point inside of?"

The modification to this algorithm for domain decomposition is trivial. We already have an existing algorithm that works for the case of no domain decomposition. Before using the existing algorithm, we implement a *domain filtering* step. If the point in question is *outside* the domain in question, that domain can immediately reject ownership of the particle. If the point in question is *inside* the domain in question, then proceed with the existing algorithm. To determine if a point is inside of a domain, the algorithm loops over all of the CG cells in the domain, using the cell's bounding box for a quick rejection test. If the point is within the cell's bounding box, then we must evaluate the point in the CSG tree and ultimately in the surface equations that define the cell.

The black dot in Figure 6 illustrates the position of a particle. The particle is outside Domain 0, so Domain 0 can immediately reject ownership of the particle. The particle is inside Domain 1, so Domain 1 must proceed as usual to test to see which cell the particle is in.

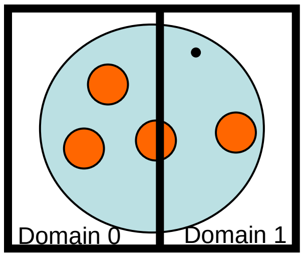


Figure 6: Illustration of determining the domain and cell of a particle. The bold black lines are domain boundaries.

Summary of the modified **Is-Point-In-Cell** algorithm for domain decomposition:

- During the ***Is-Point-In-Cell*** routine, then the algorithm ensures that the input particle is inside of the bounding box of the input domain.
- If that test passes, continue as before.
- Otherwise the particle is definitely not on the input domain.

2.4.7 Algorithms: Nearest Facet

One of the necessary algorithms to implement in a Monte Carlo particle tracking code is known as "Nearest Facet", and is isomorphic to ray tracing. As a particle is streaming through a CSG cell, it will eventually reach the current cell's boundary and cross into the next cell. Given the particle's position and direction of flight, the *Nearest Facet* algorithm calculates the distance to all of the bounding surfaces of the cell and selects the nearest boundary surface that the particle will cross.

In the case of domain decomposed CSG, we use the existing nearest facet algorithm with one modification. We must also check to find the distance to the next *nearest domain boundary interface*. If the nearest domain boundary interface is closer than the nearest cell boundary surface, then the particle must be communicated to the adjacent domain.

Mercury already had domain decomposition for *mesh* problems, so the infrastructure to buffer and communicate particles among adjacent domains already existed. After we determine that a particle is going to have a CSG domain boundary crossing, the existing infrastructure to communicate a particle from its current domain to the adjacent domain is used. Figure 7 shows an illustration of the modified Nearest Facet algorithm.

d_1 = distance to domain boundary

d_2 = distance to nearest facet.

$d_1 < d_2$ so we have a

Domain Boundary Crossing Event.

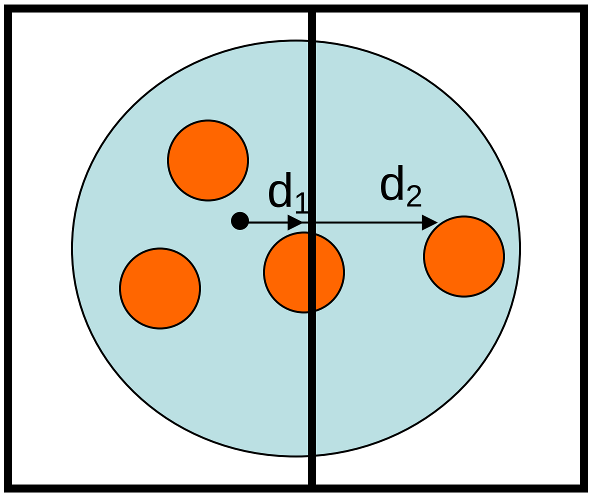


Figure 7: This example shows that the domain boundary crossing is closer than the nearest facet, so the particle will be communicated from one domain to the adjacent domain.

2.5 Criticality of the World Test Problem

This test problem is from a paper written by G. E. Whiteside in 1971: *A Difficulty in Computing k-effective of the World* [38]. The problem definition is a 9 x 9 x 9 lattice of Plutonium-239 spheres, each with a radius of 3.90cm, except for the center sphere which has a radius of 4.9320cm. All of the spheres are subcritical, except for the center sphere which is supposed to be very close to exactly critical. The centers of the spheres are 60cm apart, surrounded by low density air, which is surrounded by 30cm of water. See Figure 8 for an illustration of the problem setup.

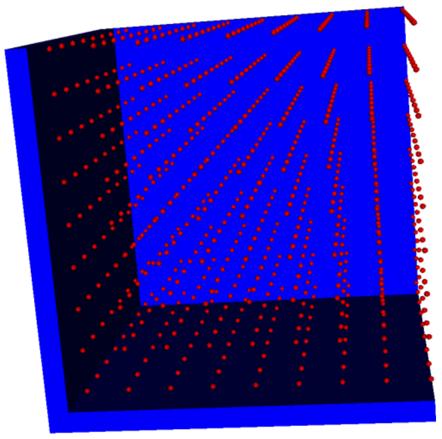


Figure 8: Problem setup of Plutonium Criticality of the World Test Problem.

This problem is a Static-K criticality problem with 150 inactive iterations, at which point the spatial distribution of the neutrons is stationary and the active portion of the calculation begins. The initial particle source is uniform throughout the spheres. This problem was run on 64 processors with 100,000 particles in all cases, and was run on 1, 2, 4, 8, 16, 32 and 64 domains. When there are less domains than processors, the domains are replicated over processors and the particle workload is distributed over the replicas. See Table 4 for the results of this study. Due to the statistical nature of the Monte Carlo algorithm and asynchronous communication, we get slightly different answers depending on the number of domains, but they are all statistically equivalent. These results demonstrate that the domain decomposition algorithms are functioning correctly.

Table 4: K_{eff} for various numbers of domains.

Number of Domains	K_{eff} +/- Std Dev
1	1.000160e+00 +/- 5.023e-3
2	1.000252e+00 +/- 4.917e-3
4	1.000313e+00 +/- 5.022e-3
8	1.000947e+00 +/- 4.848e-3
16	1.000106e+00 +/- 4.996e-3
32	1.000610e+00 +/- 5.036e-3
64	1.000872e+00 +/- 5.010e-3

2.6 LIFE Problem

Lawrence Livermore National Laboratory is the home of the National Ignition Facility (NIF) [39], the world's largest and most powerful laser. One possible application of a NIF-like laser is to use it for electricity production. That is the idea behind the Laser Inertial Fusion/Fission Energy (LIFE) [40] engine. The lasers fire on a tiny target in the center of the target chamber. This causes nuclear fusion, which release neutrons. The neutrons stream out through a fuel layer of fissionable material, which fission and release heat. The heat is used to generate electricity. LIFE is then a "fusion/fission" nuclear reactor.

To perform a detailed simulation of this facility requires a very large and complex geometric description. To model only a very small portion of the fuel layer (1^o by 1^o solid angle), requires 5.6 million CSG cells. To model the full 4π geometry would require billions of CSG cells.

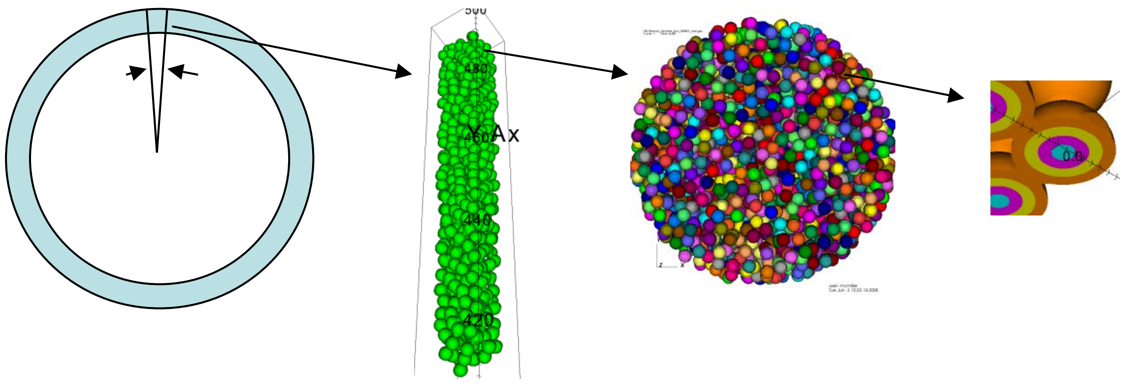
Figure 9 shows the hierarchical breakdown of the LIFE target chamber:

(a) The LIFE target chamber, the inner radius is 423 cm, the outer radius is 504 cm.

(b) A 1° by 1° wedge of pebbles, which contains 569 pebbles. Each pebble has a 1cm radius. The material between the pebbles is Flibe Coolant, made of Li, Be and F.

(c) 1 Pebble has a 1cm radius and contains 2445 triso pellets. The material between the triso pellets is Pebble Filler, Carbon.

(d) Each triso pellet has a radius of 497 μm and has 4 layers. Layer 1: U^{238}, O, C. Layer 2 and 3: C. Layer 4: C, Si.



(a) LIFE target chamber (b) 569 Pebbles (c) 1 Pebble (d) Each triso

1° by 1° wedge(2445 trisos) has 4 layers

Figure 9: This shows the LIFE target chamber and burnable fuel.

The total CSG cell count in the LIFE problem is

569 pebbles * 2445 trisos * 4 layers = 5.6 Million CSG cells.

We model the neutron scalar flux distribution as binned energy group data, with 175 energy groups. Therefore, each CSG cell requires at least 175 double precision floating point numbers, in addition to the data structure fields for describing cells and surfaces. The memory requirement for the above LIFE problem is 36GB for the geometry memory alone; additional memory is required for the Monte Carlo simulation particles. This is more memory than any one processor has, so we must distribute the problem across processors in order to solve it. We are in a unique position to solve extremely large scale, detailed problems like this.

2.6.1 Preliminary Results

In this test, we transport particles through only *one* pebble of the LIFE problem. One pebble is 2445 CSG cells (in this case, 2445 trisos, each triso is only 1 cell instead of 4). See Figure 10 for an illustration of the geometry of this test problem. The point of this test problem is to compare the run times for various numbers of domains.

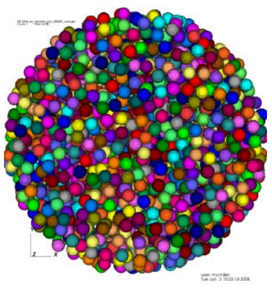


Figure 10: Pebble with 2445 CSG cells, homogenized trisos.

Table 5 shows the results of running an external source problem through a pebble with 2445 cells. The table has the nice property that if the number of domains is fixed (pick a specific row), then then particle transport time decreases as the number of processors increases. That is the “row-wise” analysis of the table. We want to concentrate on the “column-wise” analysis of the table. That is, for a fixed number of processors, what is the best number of domains to divide the problem into? One might expect that having only 1 domain would run the fastest since the calculation is embarrassingly parallel and there is no particle streaming communication. But running with only one domain is *not* the fastest way to run this problem.

Table 5: This shows the time in seconds spent doing particle transport under various domain and processor configurations.

Seconds Spent Doing Particle Transport.					
	1 proc	2 procs	4 procs	8 procs	16 procs
1 domain	848	427	226	131	74
2 domains	736	235	148	82	52
4 domains	668	190	65	34	20
8 domains	659	162	57	20	12
16 domains	686	214	113	32	12
64 domains	732	207	116	37	18

If we look at the first column of data, for 1 processor, we notice that as we increase the number of domains, the calculation actually runs faster. This is due to localization of geometry which avoids non-local intersection calculations that are impossible. Figure 11 compares the particle tracking with vs. without domain decomposition:

(a) Without domain decomposition, a particle in the filler must calculate the distance to 2445 surfaces, which is very expensive.

(b) With domain decomposition, a particle in the filler must calculate the distance to only ***local*** surfaces on this domain, which is significantly faster.

Figure 11: (a) *Without* domain decomposition (b) *With* domain decomposition

Adding more domains to a problem localizes the geometry, but a competing effect of calculating the distance to the new domain boundaries is introduced. For example, on one processor, when we go from 16 domains to 64 domains, the particle transport time goes from 686 seconds (16 domains) to 732 seconds (64 domains). In this case, the cost of tracking to more domain boundaries outweighed the cost savings of localizing the geometry.

Next we examine the column of data for 16 processors. We vary the number of domains used to simulate the problem. Using 1 domain is the traditional way of parallelizing Monte Carlo CSG transport calculations: all processors have *all* of the geometry, and the particle workload is divided evenly among the processors (i.e. geometry replication). The configuration of 16 processors and 1 domain takes 74 seconds. When the calculation is divided into 16 domains on 16 processors, it only takes 12 seconds, better than a factor of 6 speedup! This speedup again is due to localization of geometry. When a particle is inside the filler material of the pebble, it must calculate the distance to 2445 other surfaces without domain decomposition. But with 16 domains, it only has to calculate the distance to roughly 2445/16 = 153 surfaces. Competing with the speedup due to localization of geometry is the particle streaming communication introduced with domain decomposition (the calculation is slower on 64 domains). This example illustrates that domain decomposition can actually be faster than particle parallelism, as seen when comparing (16 processors, 1 domain, 74 seconds) to (16 processors, 16 domains, 12 seconds).

2.6.2 Dynamic Load Balancing

The code has an existing dynamic load balancing algorithm that is independent of the underlying geometry discretization, i.e. it is independent of the mesh type or CSG (described in detail in Chapter 3). When there are more processors than domains, the code will assign multiple processors to domains. What that means is that the particle workload will be shared evenly among the processors working on a particular domain. This is a hybrid domain decomposition/particle parallelism model. For example, in the problem below, we domain decompose the problem into 16 domains, but run it on 64 processors. Initially, each domain will have 64processors/16domains = 4 processors assigned to it. After each time step of the calculation, the code observes how much work each domain required and then redistributes the processors proportional to the workload of each domain.

Figure 12 shows the number of processors assigned to each domain, at various cycles. This test problem had the following properties:

- 64 processors, 16 domains.
- The number of processors assigned to each domain is proportional to the domain's workload.
- Pseudocolor plot of the number of processors working on each domain.
- Red = 17 processors, Blue = 1 processor.
- Cycle 0 (leftmost): uniform assignment of 4 processors to each domain.

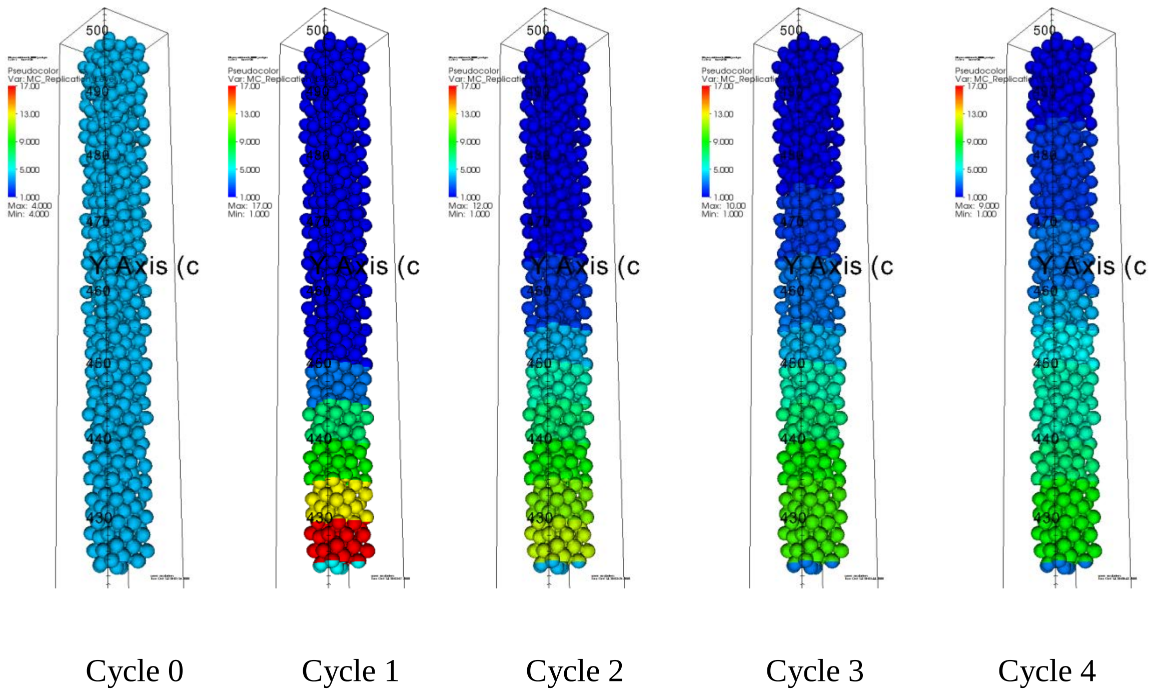


Figure 12: Dynamic load balancing example.

2.7 Boston Problem

In this test problem, all of the buildings in the city of Boston have been modeled with combinatorial geometry, including interior details of the buildings such as floors, walls, steel beams, doors, and windows. This test problem has 89 million CG cells. All of the surfaces are aligned with the coordinate axes, so it is a “cube-like” approximation of all the buildings. This problem was decomposed into 100 domains in the X-direction, 100 domains in the Y-direction and 1 domain in the Z-direction for a total of 10,000 domains. By dividing the geometry into 10,000 domains, the geometry is localized so that when particles track from the air outside of the buildings, they only have to track to the *local* surfaces in the current domain instead of to every surface. This problem was visualized on 128 processors using the Mercury inline ray caster, which we describe in section 6.4.5.

Figure 13 shows 4 different views/magnifications of the Boston test problem. The roofs of the buildings are not shown, which reveals the inside of the rooms of the buildings and a steel beam in the center of each room. These pictures were created with the inline Mercury Ray Caster.

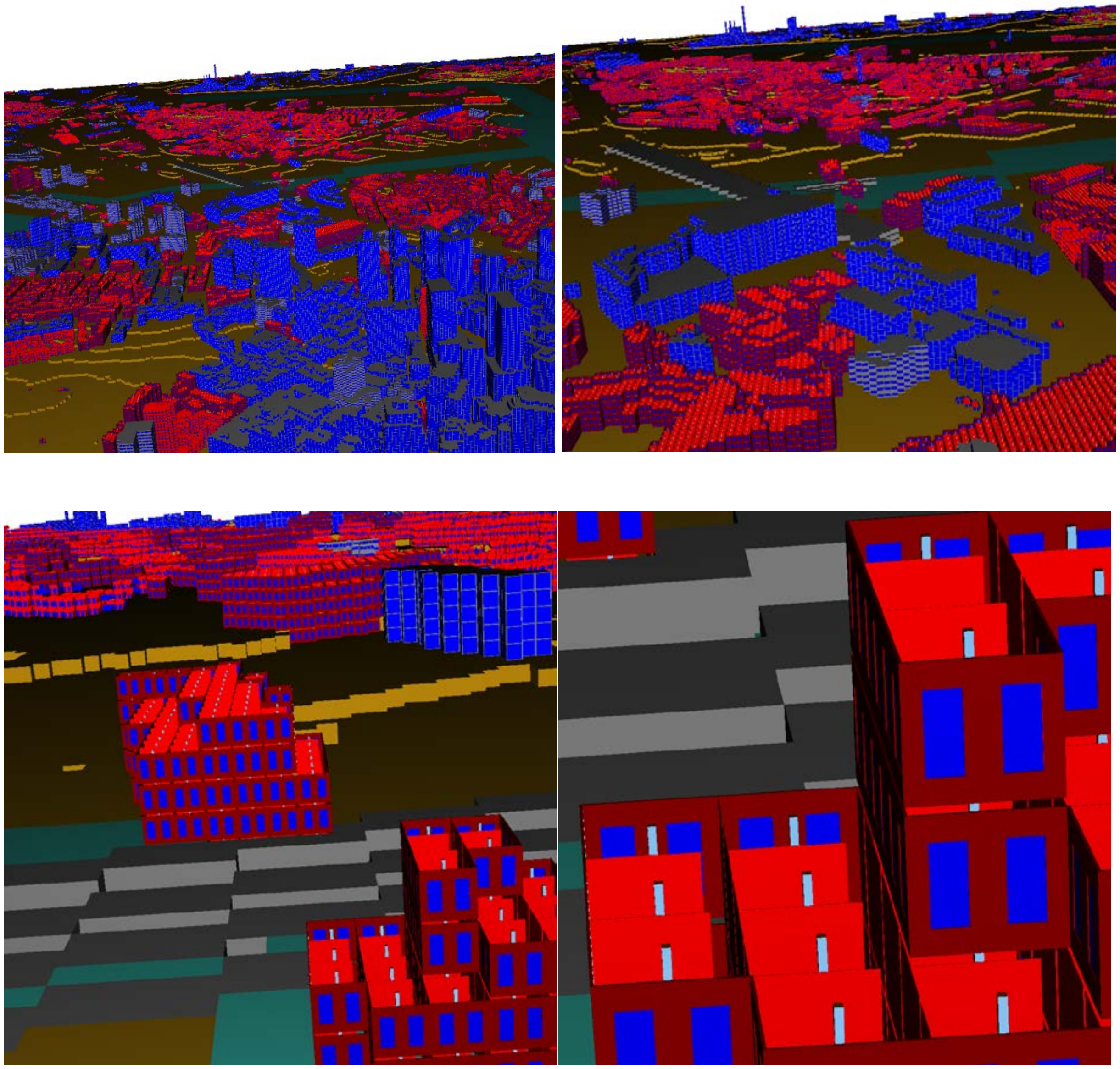


Figure 13: Ray casting visualization of Boston.

2.8 Conclusions

We have implemented a domain decomposition algorithm in the Mercury constructive solid geometry Monte Carlo transport code. This capability allows us to solve large problems that are not possible to solve without domain decomposition due to large memory requirements. We have also shown that domain decomposition can be faster than particle parallelism. Typically neutron transport problems have a non-uniform distribution of particles in space and time, and our existing dynamic load balancer works with the new CSG domain decomposition.

We have tried to keep the implementation simple, using the idea of calculating axis aligned bounding boxes for surfaces and cells, and then localizing the geometry by intersecting bounding boxes and filtering non-local geometry.

Domain decomposition enabled Mercury to visualize an 89 million CG model of the city of Boston.

3. Domain Decomposed Load Balancing

The performance of parallel Monte Carlo transport calculations which use both spatial and particle parallelism is increased by dynamically assigning processors to the most worked domains. Since the particle work load varies over the course of the simulation, this algorithm is performed each cycle. If load balancing is required, particle communications are initiated in order to achieve load balance. This method has decreased the parallel run time by more than a factor of two for certain criticality and sourced calculations shown later in this chapter, see also [41] and [42].

3.1. Introduction

Monte Carlo particle transport calculations can be very time consuming, especially for problems which require large particle counts or problem geometries with many zones. Calculations of this magnitude are normally run in parallel, since a single processor does not have enough memory to store all of the particles and/or zones. Several parallel execution modes are employed in Mercury. The first mode involves spatial decomposition of the geometry into domains, and assignment of individual processors to work on specific domains. This method, known as *domain decomposition*, is a form of spatial parallelism. The second mode, which is the easiest way to parallelize a Monte Carlo transport code, is to store the geometry information redundantly on each of the processors, and assign each processor a different set of particles. This method is termed *domain replication*, which is a form of particle parallelism. In many cases, problems are so large that domain replication alone is not sufficient. For these problems, a combination of both spatial and particle parallelism is employed to achieve a scalable parallel solution.

Since particles often migrate in space and time between different regions of a problem, a natural consequence of domain decomposition is that not all spatial domains will require the same amount of computational work. Hence, the calculation is load imbalanced. In many applications, one portion of the calculation (cycle, iteration, etc.) must be completed by all processors before the next phase can commence. If one processor has more work than any of the other processors, the less-loaded processors must wait for the most-loaded processor to complete its work.

In an attempt to reduce this form of particle-induced load imbalance, a technique has been developed which allows the number of processors assigned to a domain, known as the domain's *replication level*, to vary in accordance with the amount of work on that domain. This technique requires the use of both spatial and particle parallelism. The particles that are located in a given spatial domain are divided evenly among the processors assigned to work on that domain, known as the domain's work group.

Alme, Rodrigue, and Zimmerman [37] also investigated load balancing for Monte Carlo applications. They describe *static* load balancing in which a work estimate is used to determine the number of times a domain is replicated relative to the others. In this chapter, we consider *dynamic* load balancing where the number of times a domain is replicated may change each cycle in response to the particle workload.

This chapter describes a dynamic load balancing algorithm which minimizes the computational work of the most loaded processor by off loading part of the work to other processors. The chapter is organized as follows. The parallel architecture of the Mercury Monte Carlo particle transport code is described Section 3.2. This is followed by a discussion of a problem that illustrates the need for some form of load balancing in spatially-decomposed parallel calculations in Section 3.3. A discussion of the optimal number of processors that should be assigned to the domains is then presented in Section 3.4. The various algorithms used to implement dynamic load balancing are also described in Section 3.4. This is followed by results from parallel calculations which illustrate the advantage of enabling the dynamic load balancer in Section 3.5. Finally, the conclusions of this chapter are presented in Section 3.6.

3.2. The Architecture of the Mercury Parallel Monte Carlo Code

Mercury supports two modes of parallelism: spatial parallelism via domain decomposition, and particle parallelism via domain replication. These modes may be used individually or in combination. Spatial parallelism involves spatial decomposition of the problem geometry into domains and the assignment of each processor to work on a different (set of) domain(s).

In particle parallelism, the problem geometry is replicated on each processor, and the particles are divided among each of the processors. At the end of each cycle, the tally results need to be summed to the boss processor and presented to the user.

These modes can also be used in combination, where the problem is spatially decomposed into domains, and then within a domain, the particle load is divided among multiple processors. Each domain can be assigned a different number of processors (replication level), depending on the particle work load. Figure 14 shows an example of a problem that is divided into 4 domains, each with a different replication level. The lower left domain has the most work and is replicated

3 times; the upper right domain has the least work and is only replicated once. The other two domains have a workload somewhere in-between and are replicated two times each.

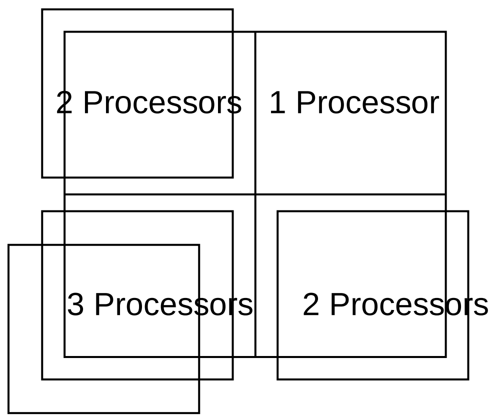
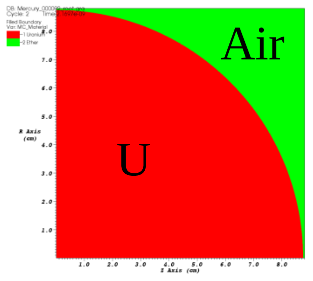


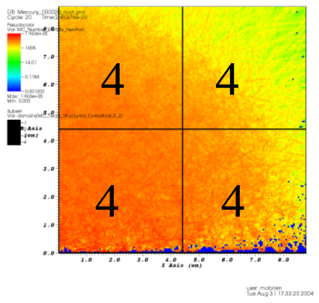
Figure 14: Illustration of hybrid domain decomposition with domain replication.

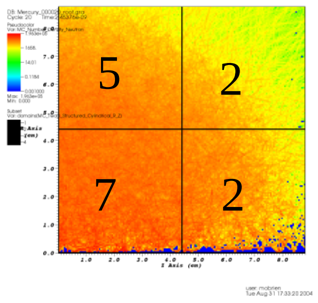
3.3. The Requirement for Dynamic Load Balancing

The requirement for some form of active management of the particle work load in a spatially-decomposed parallel transport calculation is illustrated in Figure 15. Figure 15 (A.) shows the geometry of the double-density Godiva supercritical system, a highly-enriched uranium sphere of radius r = 8.7407 cm and density of 37.48 g/cm3. Particles are sourced at the origin and an eigenvalue calculation is performed to find the alpha-eigenvalue of the system using the pseudo dynamic alpha algorithm [43]. This calculation is run on a 2D mesh with 4-way spatial parallelism: a 2 by 2 spatial decomposition, as indicated by the black domain boundary lines in Figure 15 (B.) and Figure 15 (C.).

Figure 15 (B.) and Figure 15 (C.) compare two different ways of distributing 16 processors to 4 spatial domains. The first approach (Figure 15 (B.)) is to uniformly assign 4 processors to each domain. This configuration does not take into account the actual work load of the domain, so it is less efficient (60% parallel efficiency, for a single cycle) than an approach that considers the domain work load when deciding how many processors should be assigned to each domain. The second configuration (Figure 15 (C.)) assigns processors to domains based on the work load of the domain. As a result, the parallel efficiency of this calculation is much higher (91%, for a single cycle). Figure 15 (B.) and Figure 15 (C.) are pseudo color plots of particle number density, redder areas are more computational work.







(A.) Problem setup.

(B.) 60% efficient.

(C.) 91% efficient.

Figure 15: Godiva criticality problem, uniform and load balanced replication levels.

Now in Figure 16 we examine the *dynamic* nature of the workload as the problem evolves over time. Figure 16 shows pseudo color plots of particle number density. Redder areas indicate more computational work. Clearly we have an uneven workload over time. The black lines indicate domain boundaries. Time increases to the right in the five plots (for five different cycles) in Figure 16. Initially all of the work is on the lower-left domain (Domain 0), since the particles were sourced in at the origin. As time evolves, the particles migrate to the other domains.

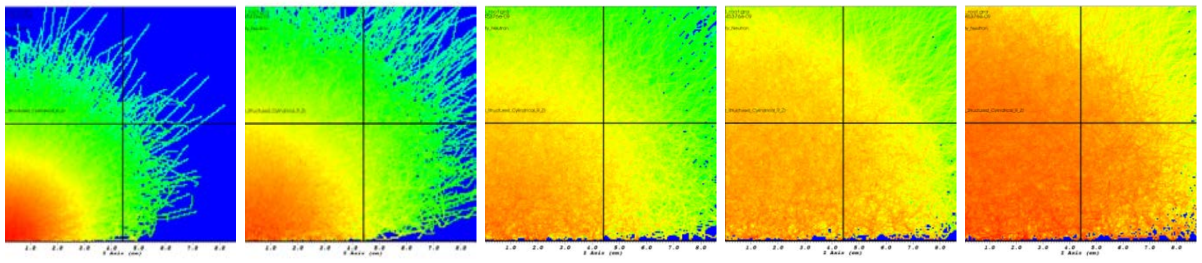


Figure 16: Pseudo-color plot of particle number density at various times during the simulation.

As used here, the *parallel efficiency* is defined to be the average computational work over all processors, divided by the maximum computational work on any processor. Let *W(p)* be the computational work on processor *p,* for *p=1,2,…,P*. We define the parallel efficiency as follows:

$$ParallelEfficiency = \frac{\frac{1}{P}\sum_{p=1}^{P} W(p)}{\underset{p=1\ldots P}{Max}\ W(p)}$$

Note that the parallel efficiency is inversely proportional to the maximum work load of any processor, so having even a single processor that is over worked can dramatically slow down the entire calculation. Since the calculation run time is inversely proportional to the parallel efficiency, the goal of load balancing is to maximize the parallel efficiency and minimize the run time to run the problem. The parallel efficiency of the calculation changes as the problem evolves over time or as the replication level of the domains changes.

3.4. Load Balancing Algorithm: The Optimal Number of Processors per Domain

The figures in the previous section clearly show that the computational work load in a parallel Monte Carlo transport calculation changes over the course of the problem. This implies a change in the work load of any given domain. As a result, the number of processors assigned to work on a domain (replication level) should respond according to the work load of that domain.

Figure 17 is a graph showing the dynamic nature of the work load from cycle to cycle in the double-density Godiva problem. The calculation was run with 4 domains on 16 processors. The calculation begins with a uniform assignment of processors to domains: each domain has 4 processors working on its particles. After the first cycle, the code responds to the large number of (sourced) particles in Domain 0 by assigning 13 processors to it (3 processors are reassigned from each of Domains 1, 2 and 3). As the calculation proceeds, the work load per domain changes, leading the code to redistribute the number of processors working on each domain. At the end of the simulation, there are 6 processors working on Domains 0 and 2, while 2 processors are working on Domains 1 and 3.

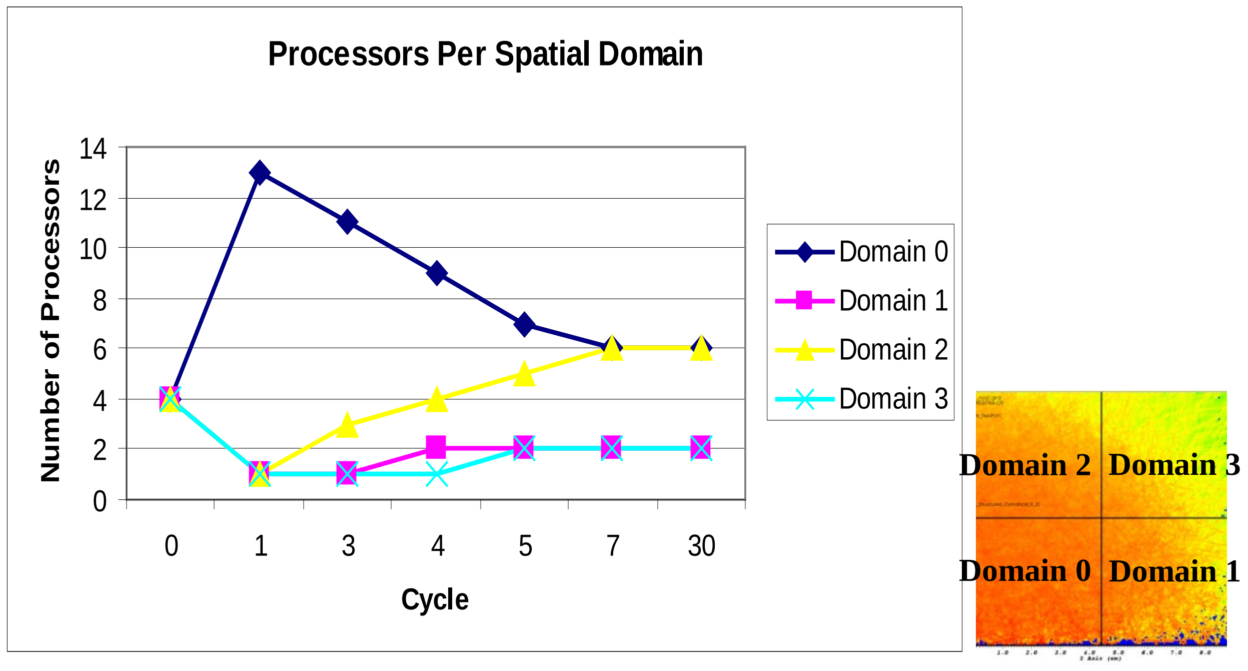


Figure 17: Replication level (number of assigned processors) of each domain as a function of Cycle.

The computational work performed by each processor *W(p)* is approximately equal to the number of particle *segments* that occurred on each processor during the previous cycle. A *segment* is defined to be one of the following particle events:

```
(1) Facet Crossing
(2) Collision
(3) Thermalization
(4) Census
(5) Energy-group Boundary Crossing
```

The computational work performed on each processor, represented by a single integer per processor, is then globally communicated, such that each processor knows the work load of all other processors. This step makes this algorithm not scalable to millions of processors, but this algorithm has been run effectively up to 65,536 processors. Future work will address modifying the algorithm to make it scalable. Since the domain that each processor is currently assigned to is known, determining the most worked domain is straightforward.

3.4.1 Number of Processors per Domain Algorithm.

This algorithm determines *how* we decide how many processors should work on each domain, the goal being to minimize the work on the most worked processor. This algorithm is analogous to what a CEO of a company might do when assigning employees to different projects. In this case

```
P  = total number of employees,
N  = number of projects,
wi = total work for project i (1…N), and
pi = number of employees working on project i (1…N).
```

Initialization: each project gets 1 employee. Iterate until there are no more employees: find the project with the most work per employee and give them another employee.

INPUT:

P = total number of processors.

N = total number of domains. ($P \geq N$)

$w_i : i=1\ldots N$, w_i = work on domain i.

OUTPUT:

p_i : i=1…N, p_i = number of processors assigned to domain i, such that:

1. $p_i \geq 1$ *for i=1…N.* (Each domain is assigned at least one processor)

2. $\sum_{i=1}^{N} p_i = P$ (The sum of all the processors assigned to all the domains must equal the total number of processors.)

3. $\underset{i=1\ldots N}{Max} \frac{w_i}{p_i}$ is minimized. (The maximum work per processor is minimized.)

Algorithm:

Keep a maximum priority queue of the work per domain w_i/p_i.

```
for i = 1 … N:
      p_i = 1                          // give 1 processor to each domain
      domainQueue.push(w_i/p_i, i)    //  put on a max priority queue
for i = N+1 … P:   // for all the extra processors (we have more processors
than domains, P ≥  N)
      (w_imax/p_imax, imax)= domainQueue.pop()  // Get the domain with the most
work/processor
      p_imax++                             // Give that domain one more processor
      domainQueue.push(w_imax/p_imax, imax)  // Put the updated domain back on the
queue
```

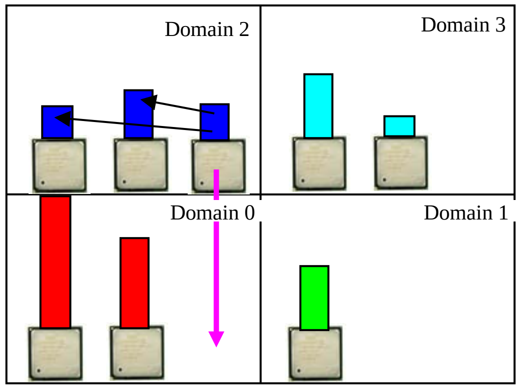
A proof by mathematical induction on *P* shows that this algorithm minimizes $\underset{i=1\ldots N}{Max} \frac{w_i}{p_i}$.

Proof: Fix $N \geq 1$. For the base case, we have *P* processors and $P = N$. There are not any degrees of freedom so $p_1 = p_2 = \ldots = p_N = 1$ and $\underset{i=1\ldots N}{Max} \frac{w_i}{p_i} = \underset{i=1\ldots N}{Max}\, w_i$. Inductively assume that for *P* processors $\underset{i=1\ldots N}{Max} \frac{w_i}{p_i}$ is minimized. Now if we have *P+1* processors, we use the inductive hypothesis to find the solution for *P* processors. With only *P* processors, say the maximum value of w_i/p_i occurs at $i=imax$. Then $w_{imax}/p_{imax} \geq w_i/p_i$. We have 1 extra processor so we can reduce the maximum value of w_i/p_i by incrementing p_{imax} by 1, so now $w_{imax}/(p_{imax} + 1) < w_{imax}/p_{imax}$. So $\underset{i=1 \ldots N}{Max} \frac{w_i}{p_i}$ is minimized. The run time of this algorithm is $\Theta(P\log(P))$ since it has P total iterations, and each iteration does $log(P)$ work, since both push() and pop() have logarithmic complexity for a priority queue.

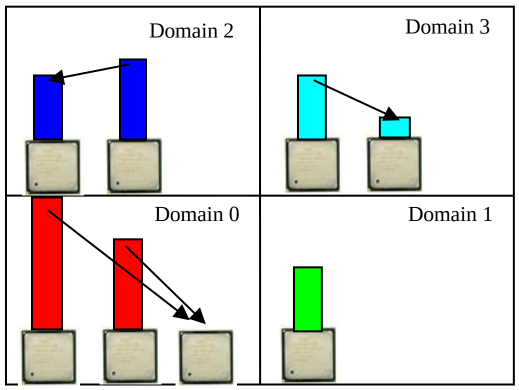
3.4.2 The Particle Communication Algorithm

Once the per-domain particle work load has been used to determine the optimal number of processors to assign to each domain, particles must be communicated between processors in order to move from the current (load imbalanced) state to the desired (load balanced) state. This task is accomplished by (a) finding the changes to the per processor particle count that need to be communicated (or transferred) to another processor followed by (b) sending that small set of particles to other processors in order to achieve load balance.

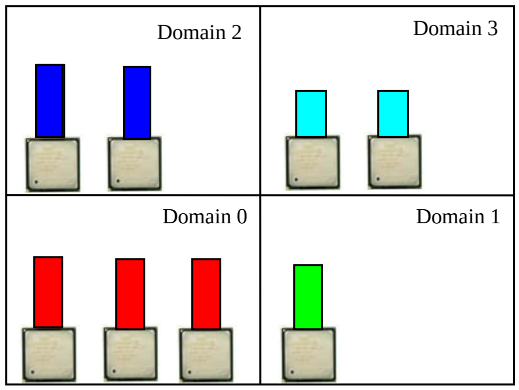
The operation of this algorithm is illustrated in Figure 18. The first step is to deal with the processors that are *leaving* a domain. The particles on a processor leaving a domain must be communicated to the processors that remain on that domain. That is illustrated in Step 1 of Figure 18. The leaving processor simply sends all of its particles evenly to the remaining processors working on the domain. The next step is to balance the particle workload within each domain, illustrated in Step 2 of Figure 18. Chapter 4 explains how to balance the particles within a given domain. This is the *homogeneous* load balancing problem. Among the processors assigned to a domain, any processor can operate on any particle. We have developed a scalable algorithm for this step that has been run up to 2^{21} = 2,097,152 MPI processes. The final load balanced state is shown in Step 3 of Figure 18. Note that the work per processor on *different* domains is not necessarily balanced. The work per domain w_i/p_i may be different for each domain, depending on the distribution of particles, the domain decomposition, and the number of processors. The work per processor on the *same* domain will be load balanced.



Step 1. Domain 2 loses one processor that goes to work for Domain 0. The particles on the processor that moves to Domain 0 must be communicated to the processors that remain on Domain 2.



Step 2. This is the communication necessary to achieve load balance within each domain. See Chapter 4 for full details.



Step 3.   The end result of load balancing. We have changed the number of processors per domain so that the maximum work per processor in minimized.

Figure 18: Illustration of particle communication algorithm.

3.5. Results

The efficacy of dynamic load balancing in the context of parallel Monte Carlo particle transport calculations is illutrated by running one criticality problem and one sourced problem. These problems are chosen because they exhibit substantial particle-induced dynamic load imbalance during the course of the calculation. Each of these problems is time dependent, and the particle distributions also evolve in space, energy and direction over many cycles. Two calculations were made for each of these problems, with the dynamic load balancing feature either disabled or enabled.

3.5.1 Criticality Test Problem

The criticality problem chosen for this test is one of the benchmark critical assemblies compiled in the International Handbook of Evaluated Criticality Safety Benchmark Experiments [44]. This particular critical assembly is a known as HEU-MET-FAST-017: a right-circular cylindrical system comprised of alternating layers of highly-enriched uranium and beryllium, with beryllium end reflectors. The assembly is $L = 35.31$ cm in length and has a radius of $r = 9.995$ cm, as shown in Figure 19. The central cavity contains a neutron source, and the two halves of the assembly are separated by a 1.52 cm air gap. In Figure 19 the green regions are highly enriched uranium, while the red and maroon regions are two different forms of beryllium.

This problem was run on a 2D $r – z$ cylindrical mesh that was spatially-decomposed into 14 domains, axially along the axis of rotation. Figure 20 shows an illustration of the domain boundaries. Parallel calculations were run on 28 processors of the MCR machine (a Linux-cluster parallel computer with 2-way symmetric multiprocessor nodes) at the Lawrence Livermore National Laboratory (LLNL). These calculations were run with 2×10^6 particles, using a “pseudo-dynamic” algorithm that iterates in time to calculate both the k_{eff} and alpha eigenvalues of the system [43].

The nature of particle induced load imbalance in this calculation is clearly seen in Figure 21. Pseudocolor plots of the particle number density are shown at five cycles during the evolution of the time iteration algorithm. Redder areas indicate a greater particle density, and hence a larger work load, than blue areas. The domain boundaries are indicated by the black lines in Figure 21. The particles are initially sourced into the problem at the center of the source cavity, as shown in the Cycle 1 plot. As the cycles progress, the particles transport through all of the domains, but it is clear the heterogeneous nature of this assembly results in uneven particle densities at all cycles.

The run in which the dynamic load balancing feature was disabled had a uniform, static replication level of two processors assigned to each domain. When load balancing is enabled, the replication level of each domain varies in accordance with the work load per domain, as shown in Figure 22. The initial load imbalance (see the Cycle 1 plot in Figure 21) results in 12 processors being assigned to work on the domain containing the source cavity (Domain 5) during cycle 1: a 12-to-2 (6-to-1) max-to-mean domain processor count. As the work load evens out over time, the replication level becomes less peaked at the center of the system. By cycle 15, some central domains are assigned 3 processors, while the peripheral domains are assigned 1 processor, with the remaining domains being assigned 2 processors: a 1.5-to-1 max-to-mean ratio.

The non-load-balanced and load balanced per-cycle run times and the parallel efficiencies are presented as a function of the cycle count in Figure 23. These figures show only the first 14 iterative cycles. When the load balancing feature is enabled, the per-cycle run times are reduced by more than a factor of 2.0 for Cycle 2, and the reduction in run time is a factor of 1.3 at Cycle 14. Similarly, the parallel efficiency is increased by a factor of 2.4 at Cycle 2, and then falls off to a factor of 1.2 at Cycle 14. The cumulative run time is reduced by 39% when the load balancer is enabled, as shown in Table 6.

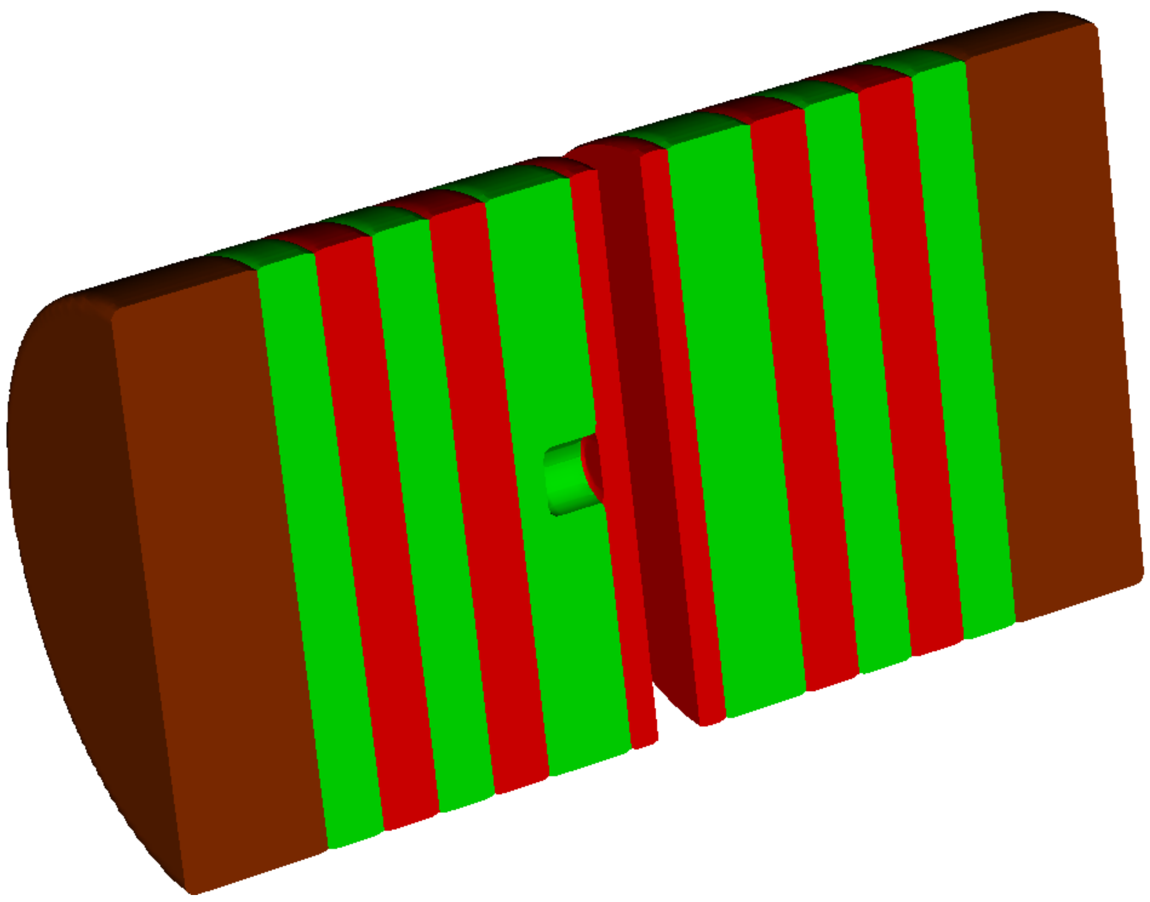


Figure 19: Geometry of the critical assembly test problem HEU-MET-FAST-017.

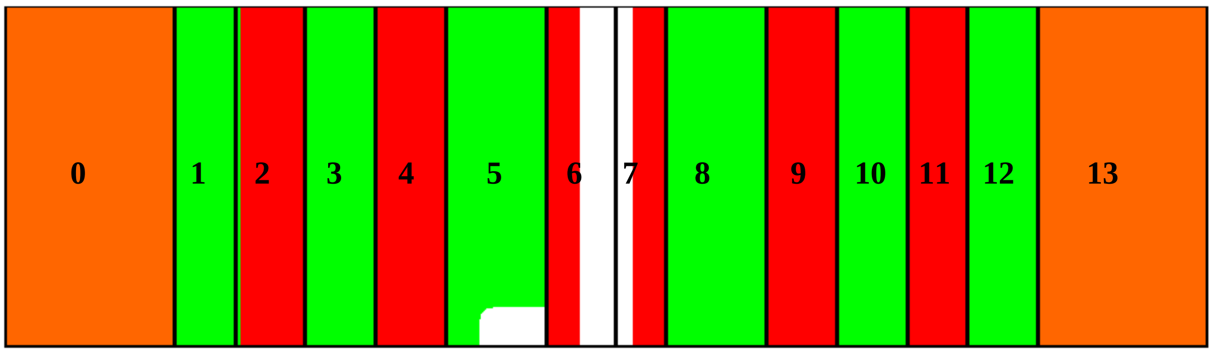


Figure 20: Domain decomposition of the critical assembly test problem HEU-MET-FAST-017.

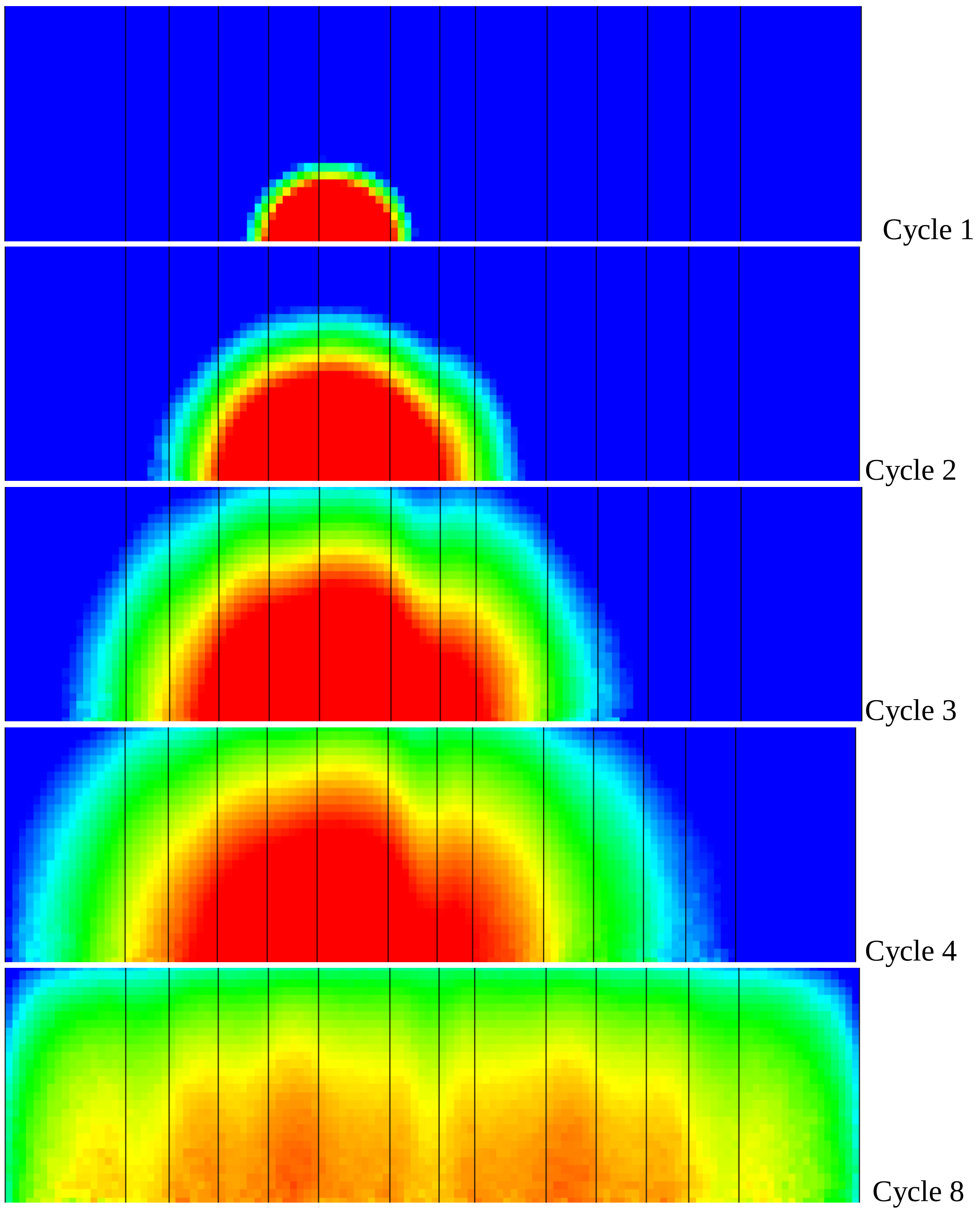


Figure 21: Evolution of neutron number density in critical assembly test problem HEU-MET-FAST-017.

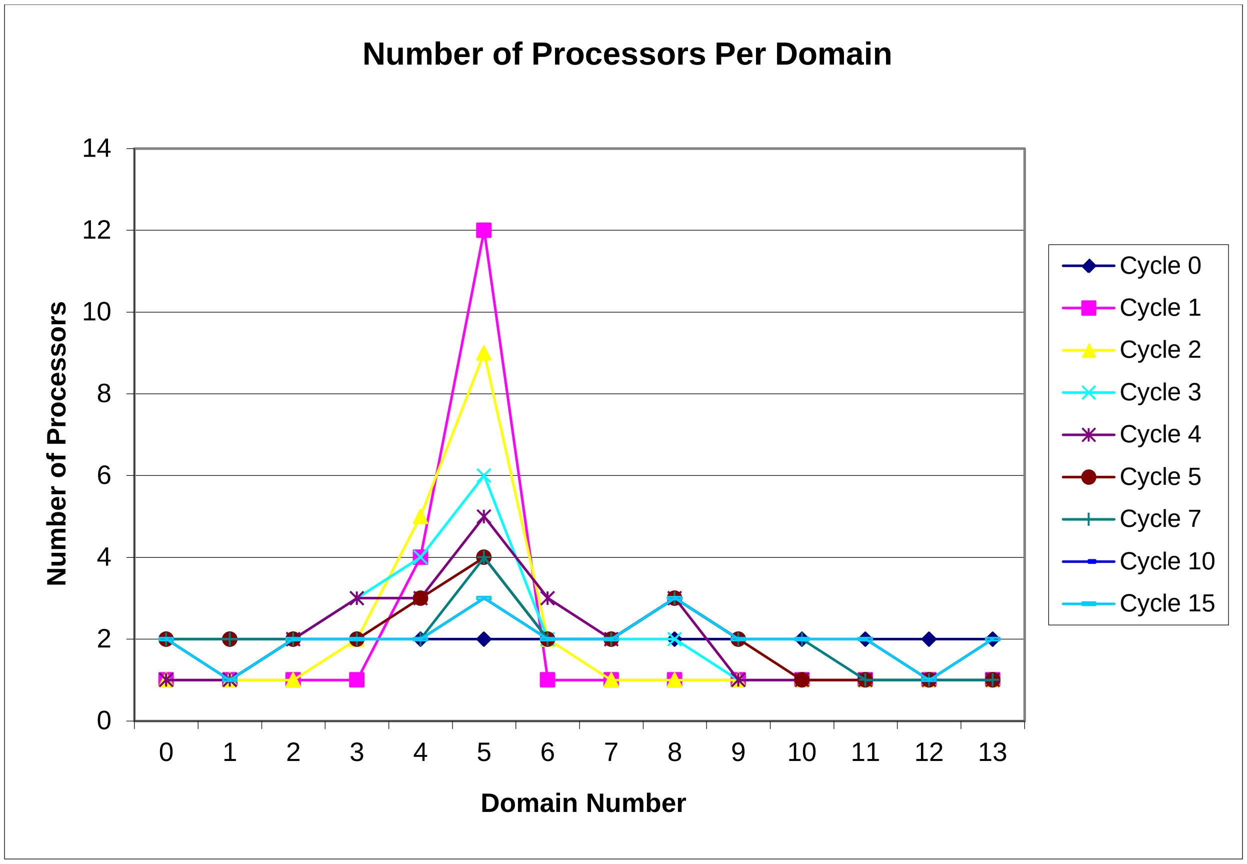
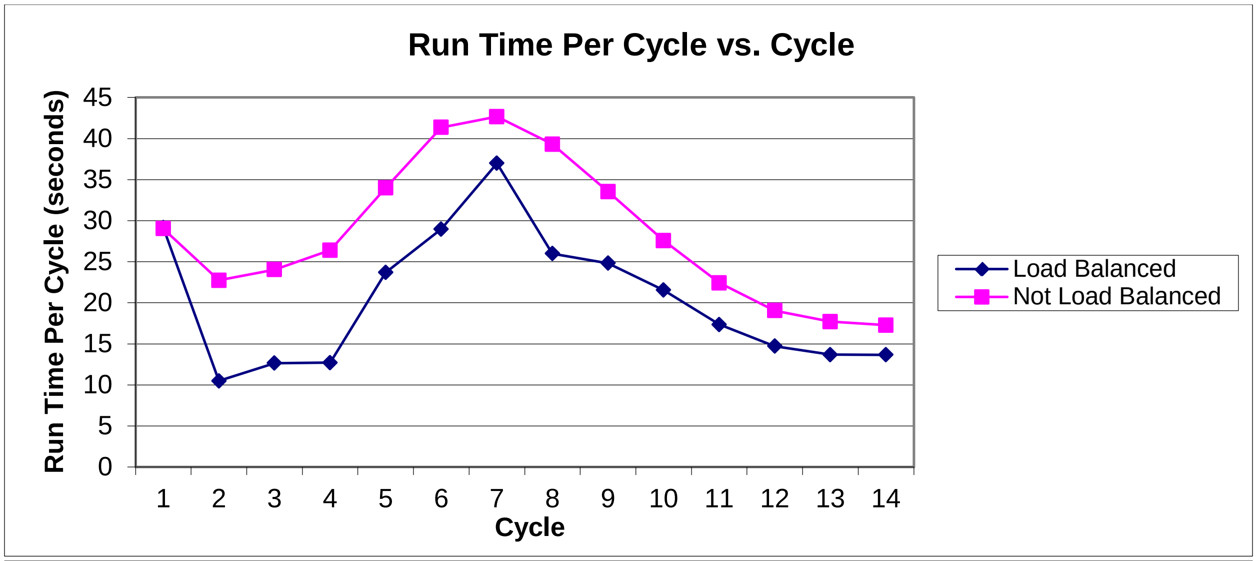


Figure 22: Evolution of number of processors per domain in critical assembly test problem HEU-MET-FAST-017.



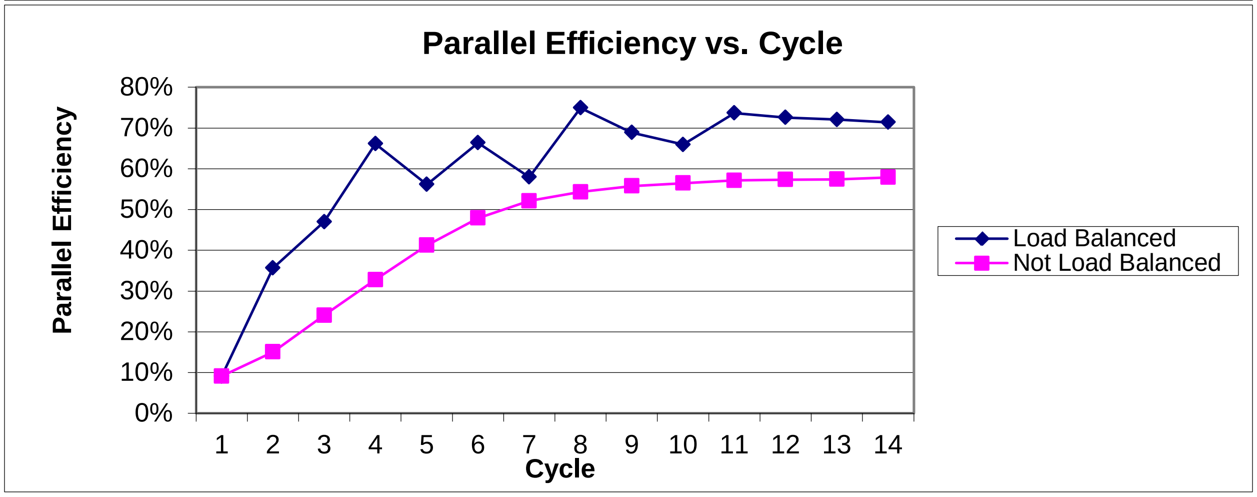


Figure 23: Per-cycle run time and parallel efficiency as a function of cycle from the critical assembly test problem HEU-MET-FAST-017.

Table 6: Problem run time from critical assembly test problem HEU-MET-FAST-017.

Problem Run Time (sec)			
Cycle Range	Not Load Balanced	Load Balanced	Speedup
1 to 4	102.2	65.0	1.57
1 to 14	397.1	286.4	1.39

3.5.2 Sourced Test Problem

The time-dependent sourced problem chosen for this test is a spherized version of a shielding configuration that has been considered as a candidate for the neutron shield surrounding a fusion reactor. The shield consists of alternating layers of stainless steel and borated polyethylene, as shown in Figure 24. The stainless steel is green, the borated polyethylene is blue, and the air is red. The radius of the inner steel sphere is r = 35.56 cm, and the thickness of each of the other shells is 5.08 cm. Monoenergetic E = 14.1 MeV particles are sourced into the center of the system from an isotropic point source. During each of the first 200 cycles, 1×10^{5} particles are injected into the system. The source is then shut off, and the particles continue to flow through the shield for the next 800 cycles. The size of the time step is $\Delta t = 1\times10^{-8}$ sec.

This problem was also run on a 2-D r – z cylindrical mesh that was spatially-decomposed in 4 domains, 2 domains along each of the axes. Parallel calculations were made using 16 processors of the MCR machine. The main tally for these calculations is the time history of the particles leaking across the outer steel layer into the air.

The evolution of the particle positions is shown for six cycles in the scatter plots of Figure 25. The particles are color coded according to their kinetic energy: red is 14.1 MeV, green is 1×10^{-5} MeV and cyan is 1×10^{-8} MeV. The interior (exterior) shell boundaries are shown in blue (red), while the domain boundaries are shown in black. It is clear from these figures that the majority of the particle work load "flows" from the bottom-left domain (Domain 0) early in time (Cycle 2), to the domains on the periphery (Domains 1 and 2) later in time (Cycles 301 through 1001).

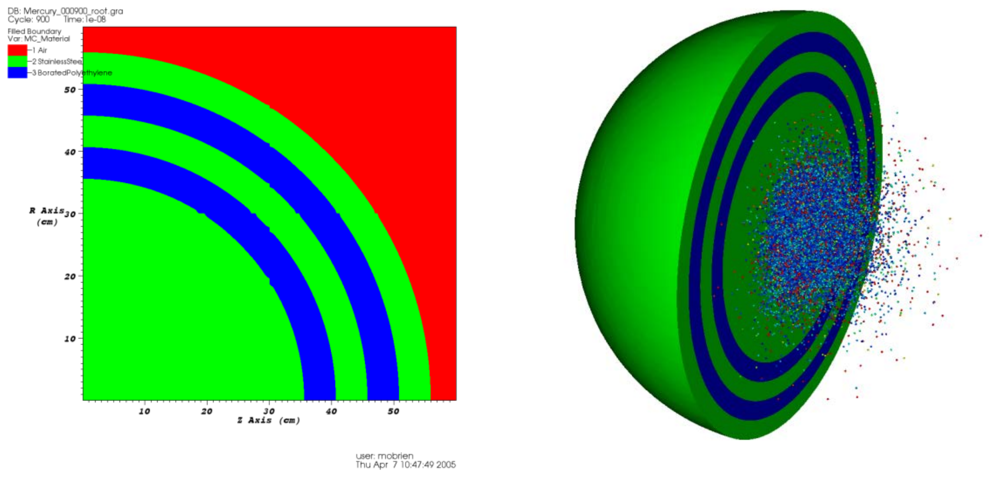


Figure 24: Geometry of the spherical shield sourced test problem.

Figure 26 shows a plot of the number of processor per domain for each of the 4 domains in this problem. Domain 0 has the particle point source and had significantly more work than the other domain, especially during the first 200 cycles when the source is on. Therefore Domain 0 has many more replicated processors to divide the particle workload.

Figure 27 is a plot of the Cycle Run Time vs. Cycle. When we compare load balanced with non-load balanced, we see for the first 200 cycles the load balanced case runs in about 3 seconds per cycle, but the non-load balanced case runs in about 7 seconds per cycle. After the source turns off, for the next 800 cycles both cases take about 1 second per cycle. When there is not much particle workload, load balancing has less of an impact because we have to pay fixed costs per cycle, independent of the particle workload.

Figure 28 shows the Parallel Efficiency vs. Cycle for load balanced and non-load balanced runs. We see for the first 200 cycles, the average parallel efficiency is about 85% for the load balanced case and only about 30% for the non-load balanced case, which is why load balancing makes the calculation run significantly faster.

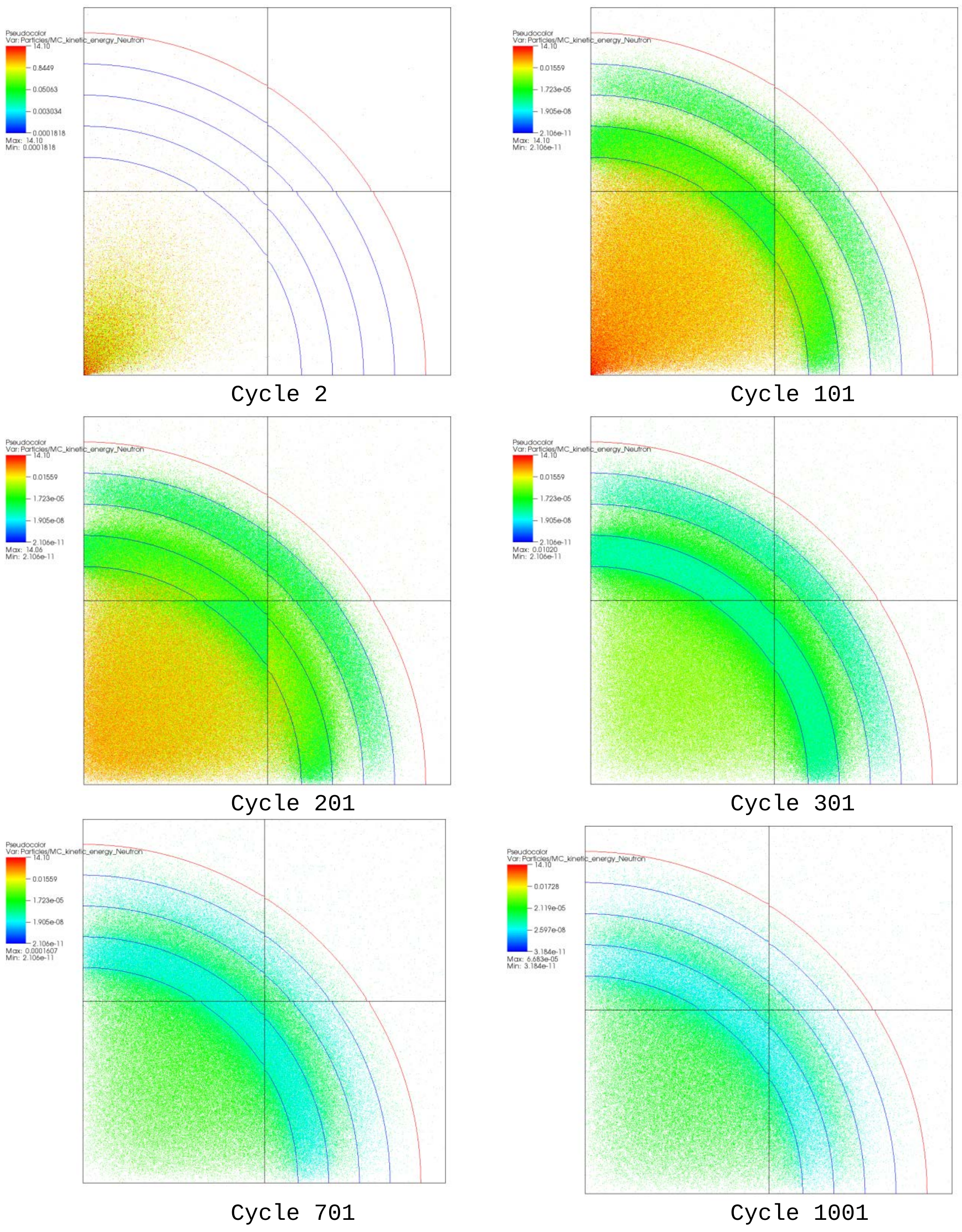
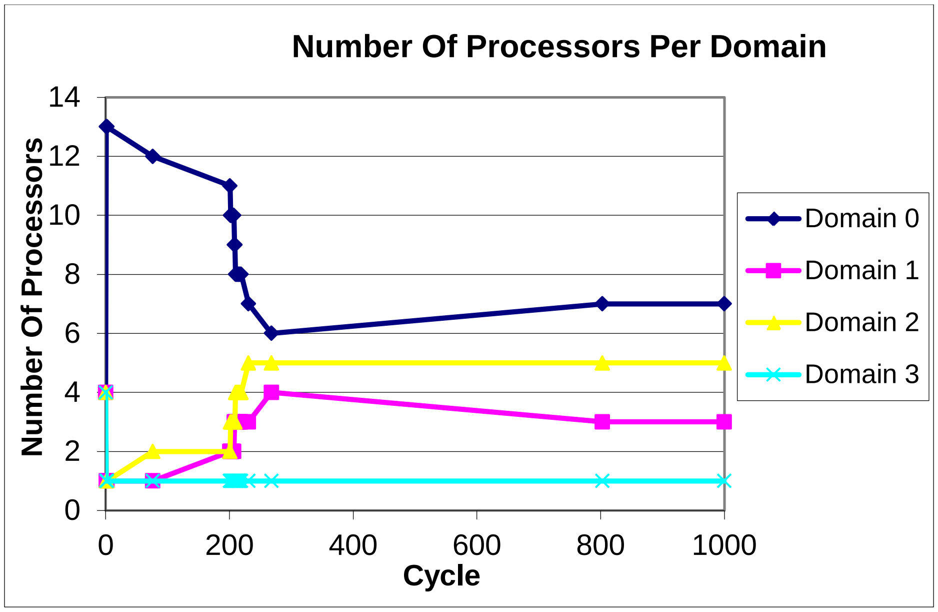


Figure 25: Plot of particle position, colored by energy for spherical shield sourced problem.



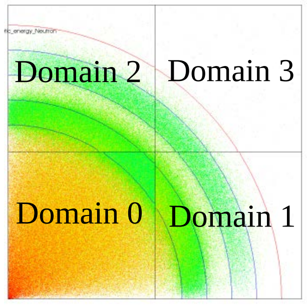


Figure 26: Number of processors per domain for spherical shield sourced problem.

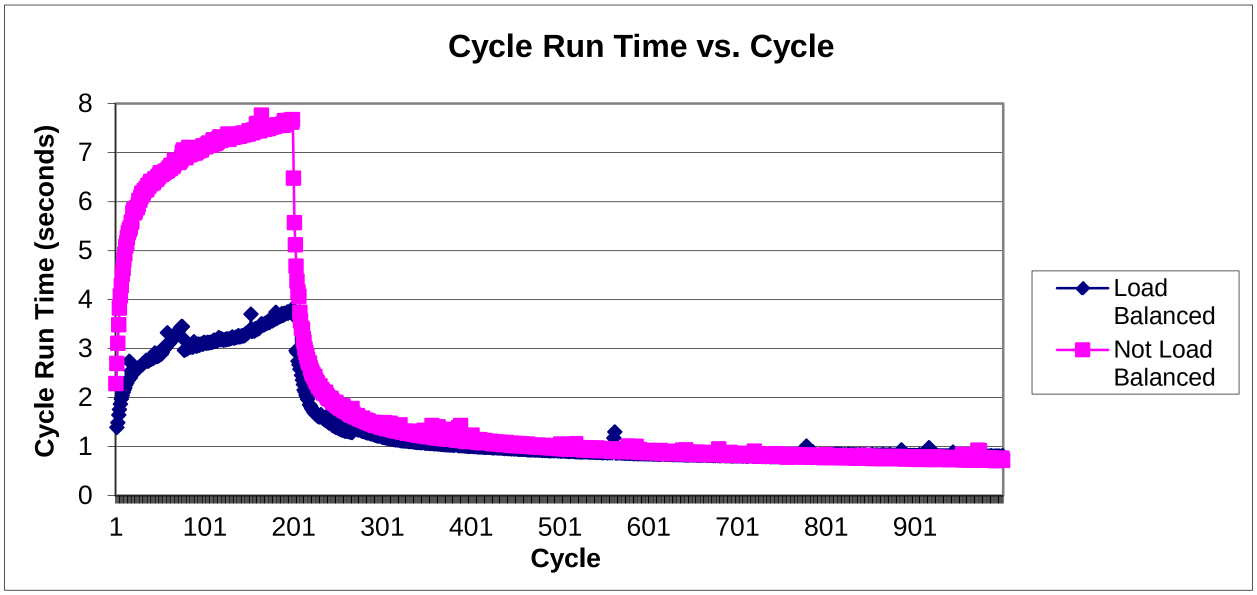


Figure 27: Cycle Run Time vs. Cycle for spherical shield sourced problem.

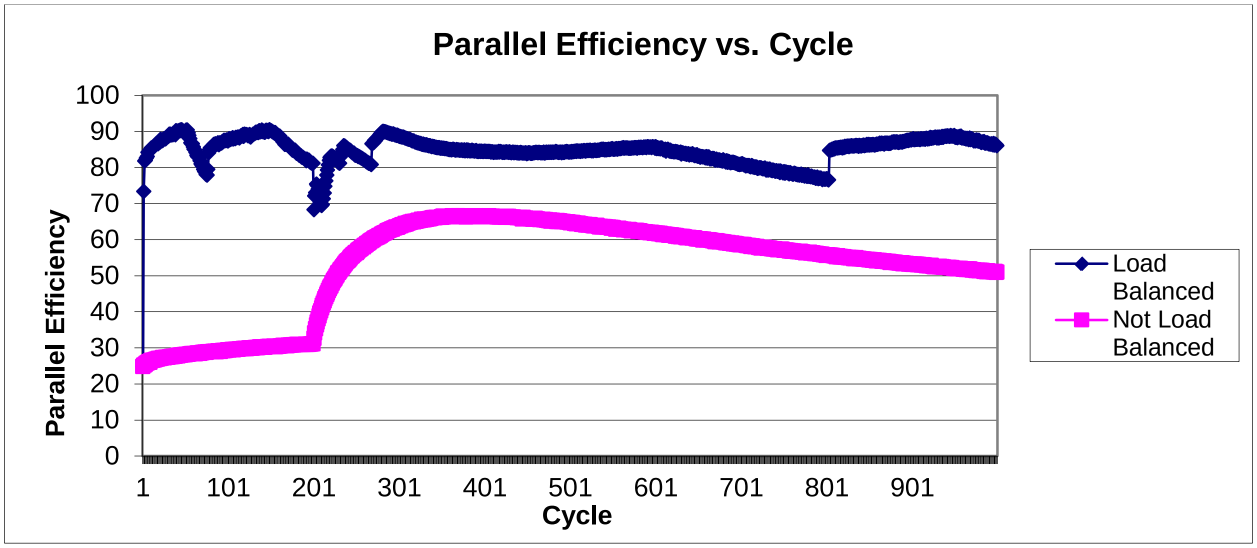


Figure 28: Parallel Efficiency vs. Cycle for spherical shield sourced problem.

Table 7 shows the problem run time for the spherical shield sourced problem. Particles are actively source in for the first 200 cycles, so the particle workload is very uneven for those cycles and the load balanced case has a speedup of 2.20 compared to the non-load balanced case. The remaining 800 cycles are not as computationally intensive and the runtime is about the same in either case, diluting the speedup factor of the entire calculation to be 1.58.

Table 7: Problem run time for spherical shield sourced problem.

Problem Run Time (sec)			
Cycle Range	Not Load Balanced	Load Balanced	Speedup
1 to 201	1355	615	2.20
1 to 1001	2221	1404	1.58

3.6 Conclusions

The particle work load in a spatially-decomposed, parallel Monte Carlo transport calculation has been shown to be dynamic and non-uniform across domains. This particle-induced load imbalance results in a reduction of the computational efficiency of such calculations. In an effort to overcome this shortcoming, the Mercury Monte Carlo code has been extended to include a dynamic particle load balancing algorithm. The method uses a variable number of processors that are assigned to each domain (replication level) in an attempt to balance the number of particles per processor. The algorithm includes logic that determines the optimal number of processors per domain, as well as how to perform the load balancing particle communications between processors. This method has been applied to the parallel calculation of one criticality and one sourced problem, where it has yielded more than a two-fold increase in the parallel efficiency.

4 Scalable Load Balancing For Particle Parallelism

In this chapter we consider the *homogeneous load balancing* problem, where each unit of work (particle) can be processed on *any* processor. Also each particle requires about the same amount of computational work to process it. This problem occurs when distributing the particles in domain replicated particle parallel Monte Carlo transport problems.

4.1 Introduction

In order to run computer simulations efficiently on massively parallel computers with hundreds of thousands or millions of processors, care must be taken that the calculation is load balanced across the processors. Examining the workload of *every* processor leads to an unscalable algorithm, with run time at least as large as $O(N)$, where N is the number of processors. We present a scalable load balancing algorithm, with run time $O(\log(N))$, that involves iterated processor-pair-wise balancing steps, ultimately leading to a globally balanced workload [45]. We demonstrate scalability of the algorithm up to 2 million MPI processes on the Sequoia [7] supercomputer at Lawrence Livermore National Laboratory.

We assume we have a distributed memory parallel supercomputer, using the *Message Passing Interface* (MPI) [19] for inter-process communication. The work described in this section is aimed at developing a scalable load balancing technique for a massively parallel Monte Carlo particle transport code [8], where the particle workload is distributed across processors. The assumptions for this chapter are that the computational cost of all the Monte Carlo particles is the same, and that any processor can process any particle. In our previous load balancing algorithm [41], [42], a description of each processor's workload was gathered to the 0th ranked processor, where a *global* communication graph was constructed to achieve a load balanced state. This algorithm performed efficiently up to thousands of processors, but for larger processor counts, the load balancing step itself took longer than the computation portion of the calculation. As soon as one array of length proportional to the number of processors is required, the algorithm is already not scalable. We define an algorithm to be ***scalable*** if its run time is at most proportional to some power of the logarithm of the number of processors. This definition rules out any global algorithm that needs to simultaneously know the workload of every processor.

We have developed and implemented a scalable load balancing algorithm in the *Mercury* Monte Carlo particle transport code [8], [34], [14]. Mercury is written in C++ with a Python user interface and uses distributed memory parallelism with MPI. Mercury models dynamic neutron, gamma and light charged particle transport and also solves neutron criticality problems. The geometry information through which the particles are transported is stored redundantly on all of the processors and is not domain decomposed in this case. (As described in Chapter 2, Mercury has domain decomposition, but this chapter only addresses particle replication). The number of Monte Carlo particles can be very large, and the particles are load balanced and distributed across processors.

The run time of our previous load balancing algorithm [41] was $O(N^2)$, where N is the total number of processors. This algorithm performed efficiently up to several thousand processors, but on 2^{17} = 131,072 processors, the load balancing step itself took 90 times longer than the computation part of the calculation. The load balancing step should take only a small fraction of the computation part of the calculation. The load balancing algorithm was not initially written with scalability in mind, so we have revisited load balancing with a focus on scalability.

Romano and Forget [46], [47] address a similar problem with a different set of constraints and assumptions. Their algorithm has the constraint that particles must be processed in a certain order, but in our case each particle has its own random number seed and may be processed on any processor in any order. This is a much less constrained problem and allows for an efficient, scalable, load balancing solution. Other large scale load balancing work has been done with the CHARM++ [48] library in [49] and [50].

The remainder of this chapter is organized as follows. In Section 4.2, we describe the need for and goals of load balancing for Monte Carlo calculations. We describe in Section 4.3 the scalable load balancing algorithm that we have developed and implemented in Mercury. In Section 4.4, we present numerical results from weak scaling studies that demonstrate the need for load balancing as well as the scalability of the load balancing algorithms we have developed. In Section 4.5, we describe MPI communicator creation, which defines the set of processors involved in each step of the load balancing algorithm. In Section 4.6 we present scaling results. Finally we conclude this chapter in Section 4.7 and offer suggestions for future work.

4.2 Load Balancing For Monte Carlo Calculations

The *load balance efficiency* of a calculation is the average amount of computational work per processor divided by the maximum amount of computational work on any processor. Let w_0, $w_1, \ldots, w_{N-1}$ be the amount of computational work per processor; then the load balance efficiency is:

$$\text{Load Balance Efficiency} = \frac{ave(w_i)}{max(w_i)} = \frac{\frac{1}{N}\sum_{0\le i<N} w_i}{\max_{0\le i<N} w_i} \qquad (1)$$

The goal of load balancing is to maximize the load balance efficiency of a calculation. By moving work from one processor to another, we cannot change the *average* amount of work per processor, but we can change the *maximum* amount of work on any processor. The goal of load balancing then becomes trying to minimize the amount of work on the processor that has the most work, thereby maximizing the load balance efficiency.

At the start of each computational physics cycle, each processor starts with some number of particles. The goal of the load balancing problem is for each processor to have the *same* number of particles (or have the maximum difference of particle counts be at most one if the number of particles is not a multiple of the number of processors). The result of the load balancing algorithm is to have particles communicated between processors so that after the communication the particle counts are balanced. Then the computational physics cycle occurs, which may induce new load imbalance, and the load balance step is repeated.

Running problems *without* load balancing results in the load balance efficiency decreasing as a function of generation in an eigenvalue calculation. The load balance efficiency also decreases as the number of processors increases. Load balancing the problem enforces that all processors have essentially the same number of particles, so the efficiency remains high.

4.3 Scalable Partner Processor Algorithm

An iterative load balancing algorithm was developed such that at each iteration, every processor finds a unique *partner* processor. The partner processors send and receive the number of particles they own to each other. Then both processors compute the average of these two numbers. The partner that is above the average sends particles to the partner that is under the average, so both processors end up having the average number of particles. If both processors have the same number of particles, then they are already load balanced and have nothing to do. This algorithm has pair-wise interactions between processors, never knowing what the global workload distribution looks like. We define the processor ***rank*** to be the unique processor number in *{0, 1, 2, …, N-1},* when there are *N* total processors. After each iteration, the partner processors are *individually* balanced. By choosing the partner processor appropriately, on the k^{th} iteration, all processor ranks with the same binary representation up to the last *k* digits will have exactly the same number of particles, i.e. processors in groups of 2^k are balanced.

We now define how the processors are paired. Processors are paired based upon their processor rank and the current iteration number *k* of the algorithm. We choose the partner processor on the k^{th} iteration of the algorithm by defining the partner function f_k, and the rank of the partner processor is given by: *partner* = $f_k(rank)$, where

$$f_k(rank) = \begin{cases} rank + 2^k & \text{if the } k^{th} \text{ binary digit of } rank \text{ is } 0 \\ rank - 2^k & \text{if the } k^{th} \text{ binary digit of } rank \text{ is } 1 \end{cases} \qquad (2)$$

Another interpretation of f_k is to flip the k^{th} binary digit of the input argument. If $a_n \ldots a_0$ are the binary digits of the processor rank, then

$$\text{rank} = a_n a_{n-1} \ldots a_{k+1} a_k a_{k-1} \ldots a_1 a_0 = \sum_{i=0}^{n} a_i 2^{\mathrm{i}} \quad \text{where } \mathrm{a_i} \in \{0,1\} \qquad (3)$$

$$f_k(rank) = f_k(a_n a_{n-1} \dots a_{k+1} a_k a_{k-1} \dots a_1 a_0) = a_n a_{n-1} \dots a_{k+1} \overline{a_k} a_{k-1} \dots a_1 a_0 \qquad (4)$$

$$where\ \ \overline{a_k} = 1 - a_k \qquad (5)$$

f_k has the appealing property that it is self-inverting: $f_k(f_k(rank)) = rank$, i.e. f_k is an *involution*, so $f_k^{-1} = f_k$. As a result of this property,

$$f_k(rank) = partner \text{ and } f_k(partner) = rank. \qquad (6)$$

Figure 29 shows a graph of the partner function f_k for k=0,1,2,3,4. Note the functions are symmetric about the identity line ("y=x" line) at integer values, which characterizes involutions (self inverse functions).

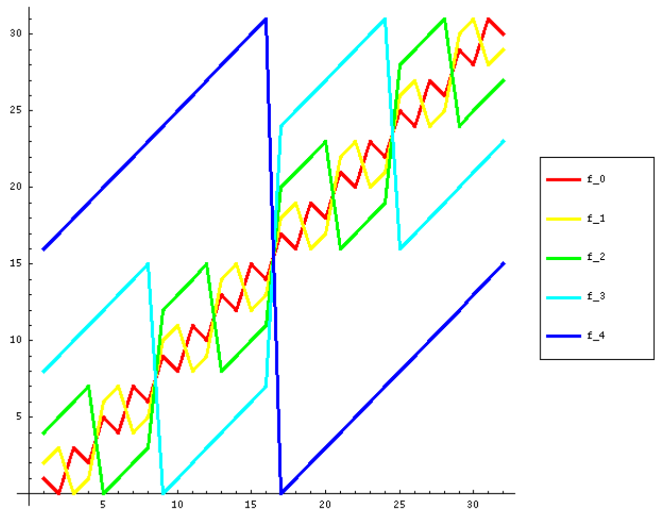


Figure 29: The partner function *f*$_k$ for k=0,1,2,3,4.

Algorithm 1 shows pseudocode for the implementation of a scalable load balancing algorithm.

```
BalanceWithPartnerWrapper()
{
    int NumRounds = ceiling(log2(numProcessors));

    for ( int k = 0; k < NumRounds; k++ )
    {
        BalanceWithPartner(k);
    }
}

BalanceWithPartner(int binaryDigit)
{
    // rank is this processor's MPI rank
    // all binary digits agree except binaryDigit is flipped in partner
    int partner = rank ^ (1 << binaryDigit);

    // Send and Recv with partner processor the number of particles each has
    int aveNumParticles = ( myNumParticles + partnerNumParticles ) / 2;

    if ( myNumParticles > partnerNumParticles ) // I am sending
    {
        // send (myNumParticles – aveNumParticles) particles to partner
    }
    else if ( myNumParticles < partnerNumParticles ) // I am receiving
    {
        // recv (partnerNumParticles – aveNumParticles) particles from partner
    }
}
```

Algorithm 1: Pseudocode showing the implementation of a scalable load-balancing algorithm

Claim: Let N = the number of processors, when $N=2^n$ is a power of 2, then **`BalanceWithPartnerWrapper()`**(Algorithm 1) globally balances the number of particles per processor so that all processors end up with exactly the same number of particles at the end of the algorithm. (Or the processor with the most number of particles has $\Theta(log(N))$ more particle than the processor with the least number of particles, which is small relative to the number of particles per processor). Furthermore the number of communication steps of this algorithm is $\Theta(log(N))$.

Proof: This follows by mathematical induction. See Figure 30 for a graphical illustration of how the algorithm works.

If $N = 1$, then the problem is already load balanced. If $N=2$, then after $log_2(2) = 1$ iteration, each processor ends up having the average number of particles per processor (or they differ by 1 if there are an odd number of particles), and the simulation is load balanced. Inductively assume that the claim holds for $N = 2^n$ processors. When we have 2^{n+1} processors, divide the processors into two groups:

Group (A) ranks 0 to 2^n-1

Group (B) ranks 2^n to 2^{n+1}-1

After *n* iterations of the algorithm, by induction each of groups A and B are independently load balanced (and processors in each group have *not* communicated between the groups). So all of the processors in group A have w_A work each, and all of the processors in group B have w_B work each. On the (n+1)st iteration of the algorithm, each processor from group A partners with a processor from group B in a one-to-one fashion so that every processor ends up with $(w_A+w_B)/2$ work, so the calculation is load balanced. The number of communication steps of the algorithm is $\Theta(log(N))$ since `BalanceWithPartnerWrapper()` calls `BalanceWithPartner()` $\Theta(log(N))$ times and `BalanceWithPartner()` does the communication between the two partner processors. Since we cannot have a *fraction* of a particle, it may be the case that after calling `BalanceWithPartner()`, one partner processor has 1 more particle than its partner.

This may happen $\Theta(log(N))$ times (once for each round), so the processor with the most number of particles may have $\Theta(log(N))$ more particles than the processor with the least number of particles. Note that $\Theta(log(N))$ is small compared to the number of particles per processor, so the calculation is still load balanced. QED.

Figure 30 illustrates the initial workload of each processor *before* load balancing, and what the workload looks like after each round of load balancing. The processor ranks are colored so that processors of the same color are "partners" and are communicating with each other; this partnership is also indicated with an arc drawn between the processor ranks communicating. After k rounds of load balancing, processors in groups of 2^k have the exact same workload.

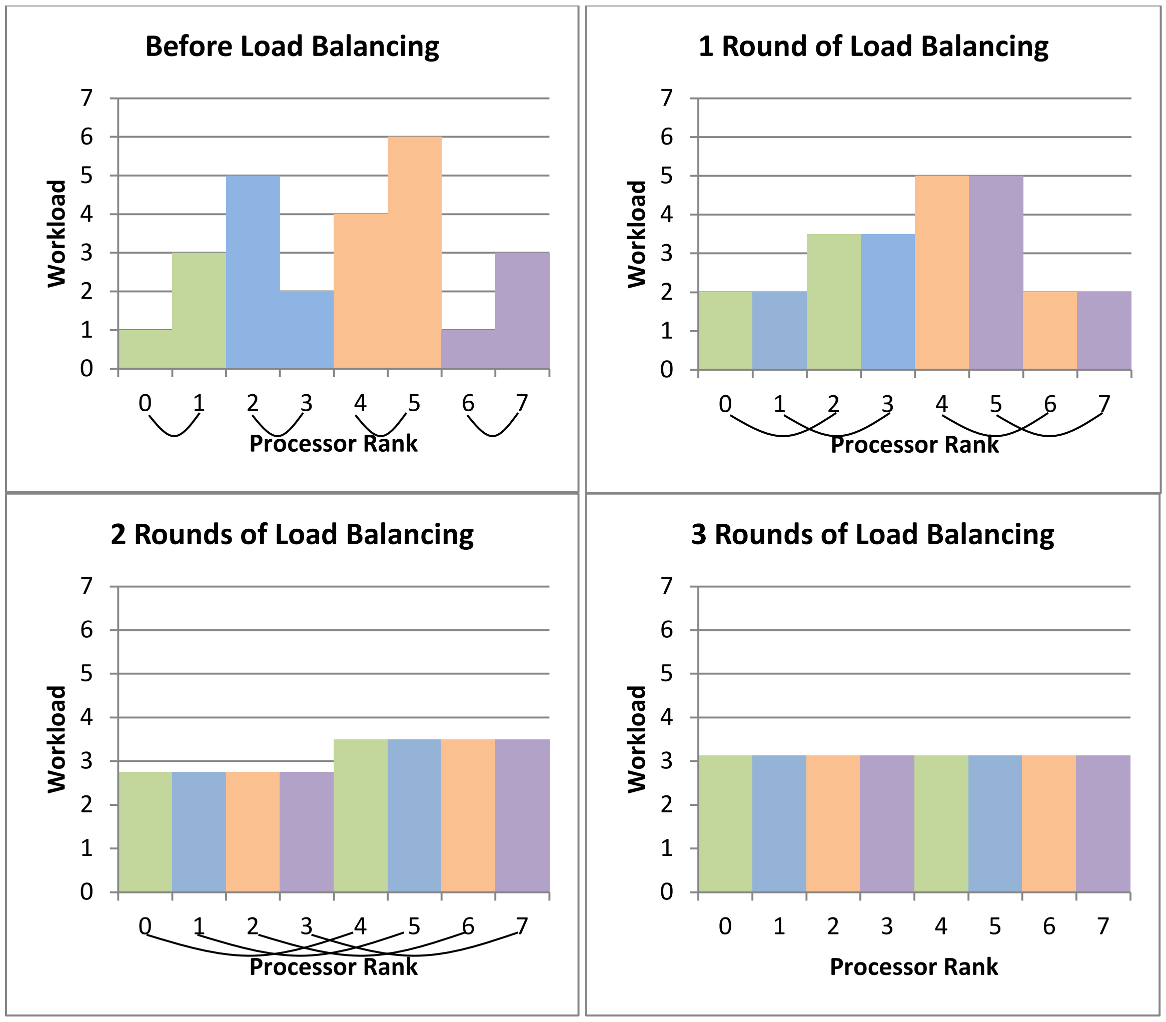


Figure 30: Illustration of processor workloads during load-balancing.

We have also developed and implemented a generalized version of this load balancing algorithm that groups processors together in groups of size 2^w, where w is a user-settable parameter of the algorithm. The generalized algorithm has the advantage that the larger w is, the fewer rounds of load balancing are required to achieve global load balance (but each round takes longer). See Algorithm 2, the `BalanceWork` algorithm, that balances the particle workload across N different processors. This algorithm has a run time of $O(N \log(N))$. We do *not* call this algorithm to balance the workload on *all* of the processors at once, since that would not be scalable. Instead we call this algorithm *$log_2(N)/w$* times, in processor groups of size 2^w, and balance the workload in stages. The detailed description of the general algorithm is described in Section 4.5.

```
BalanceWork(work[0…N-1],  /* Input: Number of particles per processor */
            communicator  /* Input: MPI communicator */ ) {
   Max_priority_queue maxQueue;
   Min_priority_queue minQueue;
   ave = (work[0] + work[1] + ... + work[N-1])/N;
   for ( int rank = 0; rank < N; rank++ ) {
      maxQueue.push(work[rank], rank);
      minQueue.push(work[rank], rank);
   }
   for ( int iteration = 0; iteration < N; iteration++ ) {
      (maxWork, maxRank) = maxQueue.pop();
      (minWork, minRank) = minQueue.pop();
      numParticles = min(ave – minWork, maxWork – ave);
      minWork += numParticles;
      maxWork -= numParticles;
      Processor maxRank sends    numParticles particles to   processor minRank
      Processor minRank receives numParticles particles from processor maxRank
      maxQueue.push(maxWork, maxRank);
      minQueue.push(minWork, minRank);
   }
}
```

Algorithm 2: Pseudocode for the `BalanceWork` algorithm, which balances the particle workload across N processors.

Claim: `BalancWork()` balances the workload of *N* processors and its computational complexity is *$\Theta(N \log(N))$*.

Proof: The idea of the algorithm is for the processor that has the most work (`maxRank`) to send work to the least worked processor (`minRank`), so that the most (or least) worked processor ends up having the average amount of work, and the other processor ends up having *max+min-ave* work. The most and least worked processors are recomputed and re-inserted into the priority queues and the algorithm is iterated *N* times. At each iteration of the algorithm, one processor ends up having the average amount of work, so after at most N iterations, all N processors have the average amount of work and the calculation is load balanced.

We use a max priority queue and a min priority queue which are data structures that have the property that data can be "`pushed`" onto them with $\Theta(log(N))$ cost and the max (or min) can be "`popped`" off with $\Theta(log(N))$ cost. Because we do at most $log(N)$ work each iteration of the loop, the computational complexity of this algorithm is $\Theta(N\ log(N))$, where N is the number of processors in the input communicator. QED.

4.4 Numerical Scaling Results

The test problem that we use in this chapter to investigate parallel scalability is the Godiva critical assembly test problem (HEU-MET-FAST-001) [44], a uranium sphere of radius 8.7407 cm and density 18.74 g/cm^3 (see Figure 31). The isotopic atom fractions are U234 = 0.01025002, U235 = 0.9376829 and U238 = 0.05206708. The Godiva problem was modeled using one spherical combinatorial geometry cell. The book *Computational Methods of Neutron Transport* by Lewis and Miller [51] describe the K-eigenvalue that is computed using the Static-K algorithm.

4.4.1 Effects of Load Imbalance

In this section, we demonstrate the effects of load imbalance for the Godiva critical sphere problem when run with large numbers of processors. Lawrence Livermore National Laboratory has the Sequoia [7] supercomputer, which we use to study the processor scaling of the Godiva Static-K criticality calculation up to 2 million (2^{21}) processors. We run Godiva as a *weak scaling* problem, that is, with fixed work per processor as we increase the number of processors. Each processor has 10,000 particles, so the total number of particles increases as we increase the processor count up to 2^{21} = 2,097,152 processors and 21 billion particles. We demonstrate that running this problem *without* load balancing results in the load balance efficiency decreasing as a function of Static-K generation. The load balance efficiency decreases as the number of processors increases, because the amount of work on the most worked processor increases as the number of processors increases. If each processor starts with the same number of particles, the number of particles on a processor at the end of an iteration follows a normal distribution [46]. As the number of processors increases, the normal distribution is sampled more often. As a result, the number of particles on the most worked processor continues to increase and the efficiency decreases.

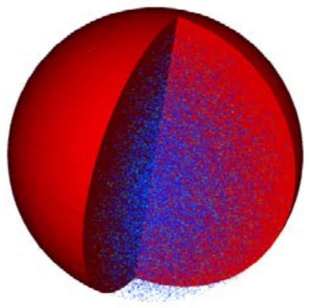


Figure 31: The Godiva critical assembly test problem.

Figure 32 shows the load balance efficiency for 6 Godiva Static-K calculations that are NOT load balanced and run on 2^1, 2^4, 2^8, 2^{16}, and 2^{21} processors. In general, the load balance efficiency decreases as the number of processors increases and decreases with the number of iterations (Static K generations).

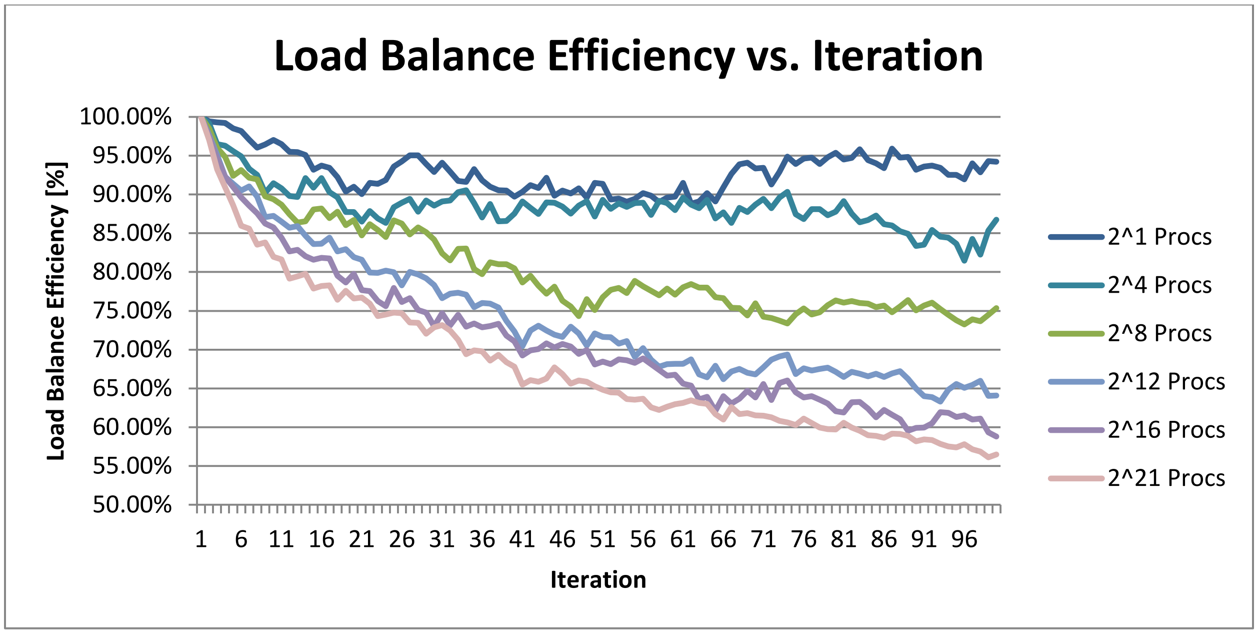


Figure 32: Load balance efficiency for 6 Godiva Static-K calculations that are NOT load balanced.

Figure 33 shows StdDev(K) vs. $\log_2$(Num_Processors). By the law of large numbers, the StdDev(K) should be proportional to 1/sqrt(Num_Particles). Since Num_Particles = 10,000 * Num_Processors, StdDev(K) should be proportional to 1/sqrt(Num_Processors), which is what we observe in this log-log plot.

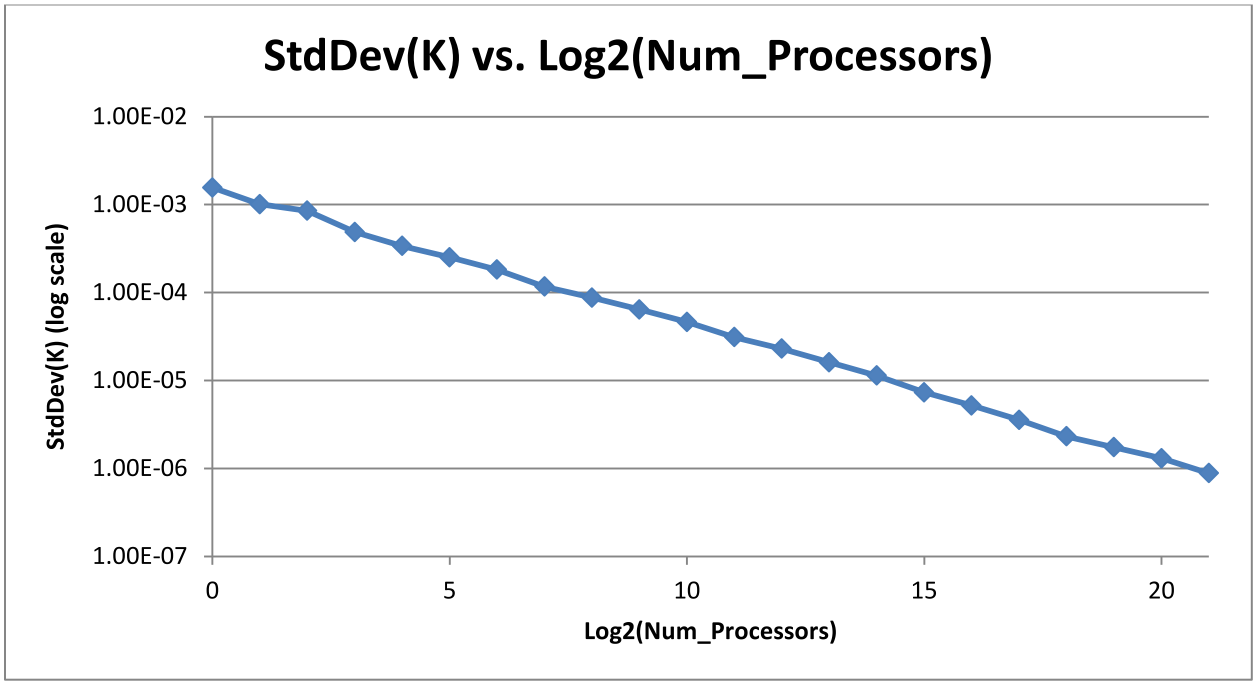


Figure 33: StdDev(K) vs. $\log_2$(Num_Processors).

4.4.2 Partner Processor Algorithm Results

In this section, we investigate the parallel scalability of the partner processing load balancing algorithm. We ran a weak scaling study for N=1, 2, 4, 8, 16, …, 2^{17} = 131,072 processors on the Dawn supercomputer (IBM Blue Gene/P [6]) at Lawrence Livermore National Laboratory. The test problem is the Godiva critical assembly test problem again run with 10,000 particles per processor. So every time we doubled the number of processors, we doubled the total number of particles. We ran 5 iterations of a Static-K calculation.

Figure 34 plots the wall time spent executing the load balancing algorithm at processor counts of 2^0, 2^1, 2^2, …, 2^{17} = 131,072. This plot is almost linear on a log-linear scale. The linear regression line through the data has slope 0.0633s/(doubling of processors). Each time we double the number of processors, we pay a fixed cost of 0.0633s, regardless of the number of processors. These calculations confirm that the load balancing algorithm wall time is in fact proportional to the log of the number of processors.

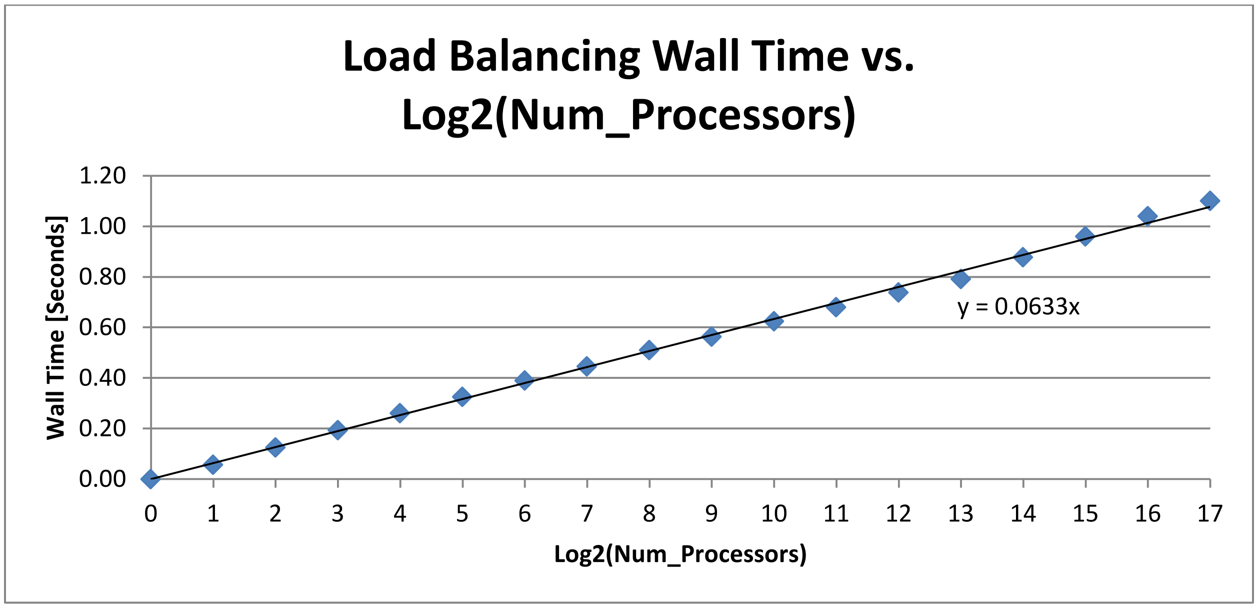


Figure 34: Wall time spent executing the load-balancing algorithm.

4.4.3 Generalized Load balancing Algorithm Results

We study the processor scaling of the Godiva Static-K criticality calculation up to 2 million (2^{21}) processors using the Sequoia supercomputer. The results in this section used the generalized load balancing algorithm described in Section 4.5. We examine how the load balance efficiency scales, shown in Figure 35, and how the particle tracking time scales, shown in Figure 36. We also examine how the load balancing step itself scales with the number of processors, shown in Figure 37. We have proven the algorithm scales like *O(log(N))*, which is what we observe when we time the algorithm.

Figure 35 is a plot of the average load balance efficiency vs. *log_2(Num_Processors)*, comparing load balancing against not load balancing. With load balancing, the efficiency remains above 95%, even at 2 million processors. Without load balancing, the efficiency decreases with processor count down to 68% at 2 million processors.

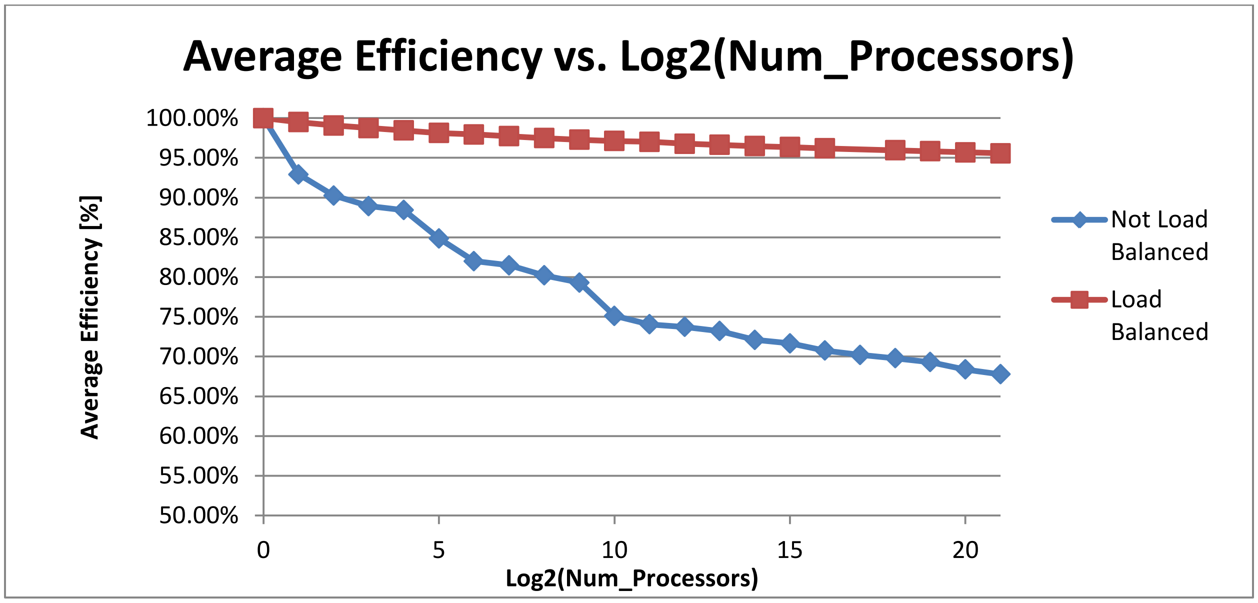


Figure 35: Average load balance efficiency vs. $\log_2$(Num_Processors)

Figure 36 shows the wall time spent tracking particles vs. *$\log_2$(Num_Processors)*, comparing load balancing against not load balancing. With load balancing, we see essentially perfect scaling up to 2 million processors. Since each processor has the same amount of work, the calculation takes the same amount of time at any scale. Without load balancing, dispersion in the number of particles per processor occurs, and the calculation cannot proceed until the most worked processor has finished. This dispersion in processor workload results in an increase in tracking time.

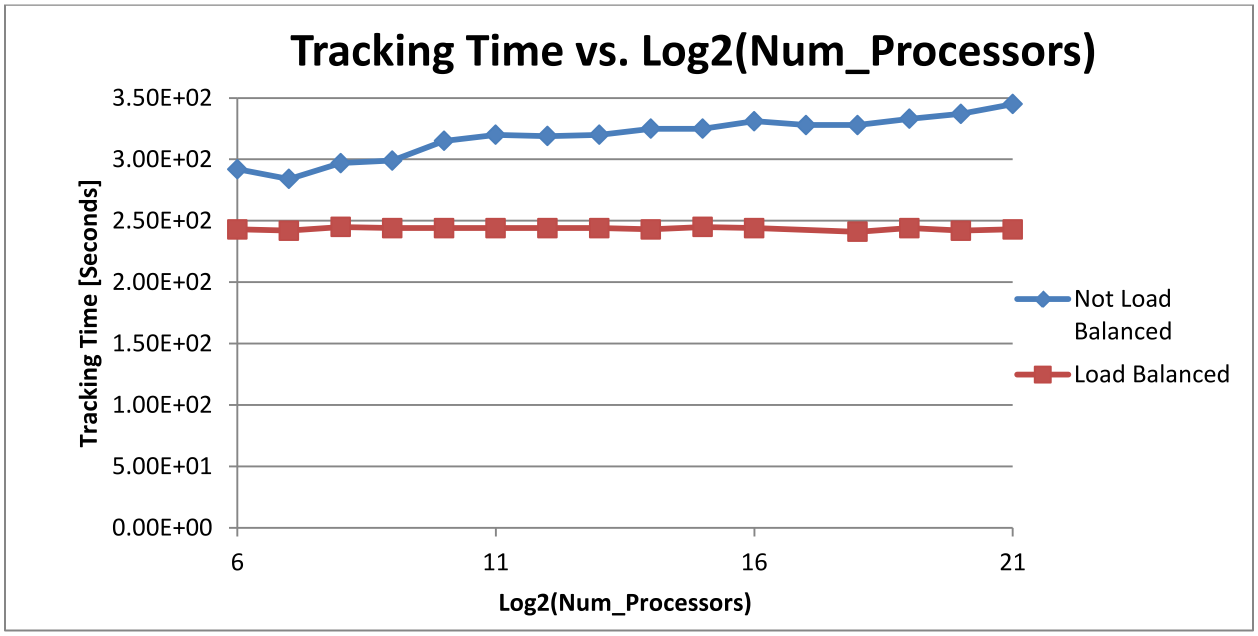


Figure 36: Wall time spent tracking particles vs. $\log_2$(Num_Processors)

Figure 37 shows the scaling behavior of the load balancing algorithm by plotting the load balancing time vs. *log_2(Num_Processors).* On this plot the wall time should look roughly like a straight line, since the algorithm is *O(log(N)).* The algorithm balances particle workloads in processors of group size 2^9, so we see an extra added expense at 2^9 and 2^{18}, since the algorithm needs to do an additional round of communication.

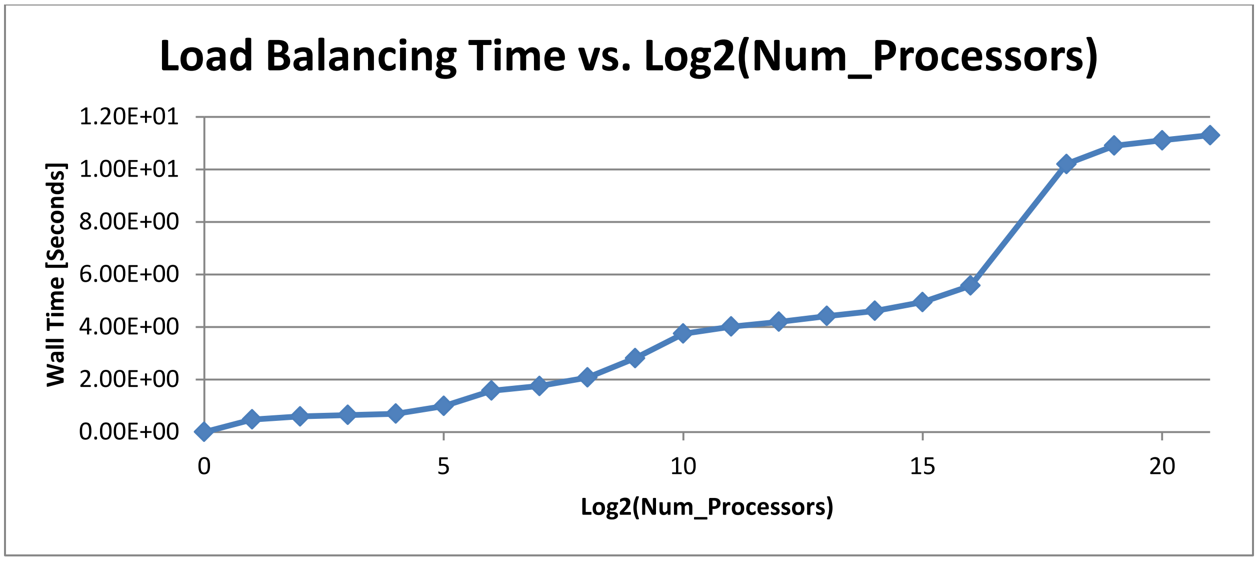


Figure 37: Scaling of the generalized load-balancing algorithm.

4.5 Generalized Load Balancing Algorithm

In this section we describe a generalization of the partner processor load balancing algorithm. Instead of grouping processors together in groups of size two, the generalized algorithm balances particles in processor groups of size 2^w, where w is a parameter of the algorithm. One of the necessary steps in this algorithm is to create an MPI Communicator (which is a group of communicating processors) for the processors involved in communication. Since there are many rounds of communication required to achieve load balance, this algorithm creates an MPI Communicator for each round.

4.5.1 MPI Communicator Creation

The next algorithm describes how to create MPI communicators of size 2^{width}, where *width* is a parameter of the algorithm. We create one communicator for each round of the load balancing algorithm; the total number of rounds depends on the total number of processors being used for the calculation, and the *width* parameter:

$$NumRounds = \left\lceil \frac{\log_2(totalNumProcessors)}{width} \right\rceil$$

totalNumProcessors must be a power of 2, *totalNumProcessors* = 2^n, so

$$NumRounds = \left\lceil \frac{n}{width} \right\rceil$$

See Table 8 for an example of the processor ranks in each communicator when *width=2.* Let the binary digits of the rank of a processor be: rank = $a_n a_{n-1} \ldots a_5 a_4 \; a_3 a_2 \; a_1 a_0$.

Table 8: Processor ranks in each communicator when *width=2.*

Round 0: *communicator[0]*	*Round 1:* *communicator[1]*	…	*Last Round:* *communicator[NumRounds-1]*
$a_n a_{n-1} \ldots a_5 a_4 \; a_3 a_2 \; 00$	$a_n a_{n-1} \ldots a_5 a_4 \; 00 \; a_1 a_0$		$00 \ldots a_5 a_4 \; a_3 a_2 \; a_1 a_0$
$a_n a_{n-1} \ldots a_5 a_4 \; a_3 a_2 \; 01$	$a_n a_{n-1} \ldots a_5 a_4 \; 01 \; a_1 a_0$		$01 \ldots a_5 a_4 \; a_3 a_2 \; a_1 a_0$
$a_n a_{n-1} \ldots a_5 a_4 \; a_3 a_2 \; 10$	$a_n a_{n-1} \ldots a_5 a_4 \; 10 \; a_1 a_0$		$10 \ldots a_5 a_4 \; a_3 a_2 \; a_1 a_0$
$a_n a_{n-1} \ldots a_5 a_4 \; a_3 a_2 \; 11$	$a_n a_{n-1} \ldots a_5 a_4 \; 11 \; a_1 a_0$		$11 \ldots a_5 a_4 \; a_3 a_2 \; a_1 a_0$

Table 8 shows the processor ranks in each communicator. The ranks in each communicator are different for each processor since they depend on the rank of each processor. The same communicator is created on 2^{width} processors.

Note that within each round (column), each processor creates exactly one communicator. Processors in groups of size 2^{width} create the *same* communicator. The processors have been partitioned into disjoint communicators. The size of each communicator is 2^{width}, and we have 2^n

total processors; so across *all* processors, we have $2^{n-width}$ different communicators per round. The processors in a communicator have *identical* binary digits except for *width* digits. In the 0th round, the least significant *width* digits are allowed to vary, and each round the variable digits "shift to the left". Note that when *width=1*, this algorithm reduces to the *partner processor* balancing algorithm from the beginning of this chapter.

Below is the pseudocode for the creation of the MPI communicators and for using the communicators to load balance the particle workload.

```
vector<MPI_Comm> CreateCommunicators(int width /* input */ ) {
    int NumRounds = ceiling(log2(totalNumProcessors)/width);
    vector<MPI_Comm> communicator(NumRounds);
    for ( int k = 0; k < NumRounds; k++ ) {
        int communicatorSize = 1 << width;
        int mask = ( communicatorSize-1 ) << (round * width );
        if ( round == NumRounds - 1 ) //deal with the remainder on the last round
            communicatorSize = totalNumProcessors >> (round*width);
        vector<int> processors(communicatorSize);
        for ( processor = 0; processor < communicatorSize; processor++ )
            processors[processor] = (rank & ~mask) + (processor<<(round*width));
        communicator[k] = MPI_Comm_create(processors);


    }
    return communicator;
}
LoadBalance(int width,                  /* input */
            communicator[0…NumRounds-1] /* input: MPI communicators */ ) {
    int NumRounds = ceiling(log2(totalNumProcessors)/width);
    for ( int k = 0; k < NumRounds; k++ ) {
        work[0…2^w′-1] = MPI_Allgather(MyNumParticles, communicator[k]);
        BalanceWork(work, communicator[k]);


    }
```

Algorithm 3: `CreateComunicators()` is done once per simulation, but `LoadBalance()` is done every cycle of the simulation, since each cycle can potentially introduce load imbalance.

Claim: The `LoadBalance()`algorithm balances the particle workload and has computational complexity *$\Theta(2^w \log_2(N))$*, where *$N=2^n$* is the total number of processors in the simulation and *w=width* is a user settable parameter of the algorithm.

Proof: The proof follows by induction on the number of rounds. Fix *w*. The base case is when we have only *one* round. In that case `communicator[0]` contains *all* of the processors and we call `BalanceWork()` on that communicator so all of the processors will end up having the same amount of work. Inductively assume the algorithm balances the work for *(NumRounds-1)* rounds, and now examine what happens if we have one more round. The total number of processors *$N=2^n$* may not be a power of *2^w*, ie. *n* may not be evenly divisible by *w,* so the last round will have a communicator of size *$2^{w'}$*, where *w'* is given by:

$$w' = \begin{cases} w, & \text{if } n \bmod w = 0 \\ n\, mod\, w, & \text{if } n \bmod w \neq 0 \end{cases}$$

Table 9 shows the binary digits of processor ranks. Each column in the table is already load balanced by the inductive hypothesis. The inductive step is the last round of load balancing which balances each row. There are *$2^{n-w'}$* rows. Since each column is already balanced and then we balance each row, we end up with all processors being load balanced.

Table 9: Binary digits of processor ranks.

Size of last communicator = $2^{w'}$. ($w' = 3$ in this example)

Already Balanced Group 000	Already Balanced Group 001	Already Balanced Group 020	Already Balanced Group 011	Already Balanced Group 100	Already Balanced Group 101	Already Balanced Group 110	Already Balanced Group 111
000 0..00	**001** 0..00	**010** 0..00	**011** 0..00	**100** 0..00	**101** 0..00	**110** 0..00	**111** 0..00
000 0..01	**001** 0..01	**010** 0..01	**011** 0..01	**100** 0..01	**101** 0..01	**110** 0..01	**111** 0..01
000 0..10	**001** 0..10	**010** 0..10	**011** 0..10	**100** 0..10	**101** 0..10	**110** 0..10	**111** 0..10
000 0..11	**001** 0..11	**010** 0..11	**011** 0..11	**100** 0..11	**101** 0..11	**110** 0..11	**111** 0..11
…							
000 $a_j..a_1a_0$	**001** $a_j..a_1a_0$	**010** $a_j..a_1a_0$	**011** $a_j..a_1a_0$	**100** $a_j..a_1a_0$	**101** $a_j..a_1a_0$	**110** $a_j..a_1a_0$	**111** $a_j..a_1a_0$
…							
000 1..11	**001** 1..11	**010** 1..11	**011** 1..11	**100** 1..11	**101** 1..11	**110** 1..11	**111** 1..11

Table 9 illustrates the processor groupings. In the last round we will have $2^{w'}$ columns of processors. Each column has $2^{n-w'}$ rows, and the processors within each column are all balanced and have the same amount of work by the inductive hypothesis. A communicator is constructed for each row and contains one processor from each balanced group. `BalanceWork()` is then called on that communicator to balance each row, and since the processors in each column contain the same amount of work by the inductive hypothesis, now the *all* the processor ranks are load balanced.

We have just proved correctness of the algorithm. Now we will address computational complexity of each round of the algorithm. Each round of the algorithm calls `MPI_Allgather()` for a message of size 1, on a communicator of size 2^w, so the cost of that call is $\Theta(2^w)$. We also call `BalanceWork()` on a communicator of size 2^w, and we have already proved that has computational complexity of $\Theta(w2^w)$. So the computational complexity of each round of the algorithm is $\Theta(w2^w)$.

The total computational cost of the algorithm can be broken down into the cost of each round, times the number of rounds. The pseudocode clearly shows that

$$NumRounds = Ceiling(\ log_2(N)/w\ )$$

and we just proved that the cost of each round is $\Theta(w2^w)$. As a result,

Total Cost = (Cost of each round)*(NumRounds) = $\Theta(w2^w) * \Theta(log_2(N)/w) = \Theta(2^w\ log_2(N))$

QED.

4.6 Results

The cost of the load balancing algorithm is $\Theta(\ 2^w\ log_2(N)\ )$, where w is a user settable parameter. So the question is how should we select the optimal values of w? When $w=1$ we get the *"partner processor"* load balancing algorithm described in Section 4.3, with cost $\Theta(log_2(N))$. When $w=log_2(N)$, we get $\Theta(N\ log_2(N))$, which is not scalable.

Since each round of the algorithm has to do communication, there is some latency, λ, associated with each round of communication that depends on the network interconnect of the supercomputer being used. We therefore model the time for each round as

$$\text{Time of each round} = w2^w + \lambda$$

The total number of rounds is $\log_2(N)/w$ (leaving off the ceiling for simplicity), so the total time is $(w2^w + \lambda)\log_2(N)/w$. The total time will be minimized when its derivative with respect to *w* is 0:

$$\frac{d}{dw}\left(\frac{w2^w + \lambda}{w}\right) = 0$$

which happens when

$$w^2\, 2^w \ln 2 = \lambda$$

Neglecting w^2 which is small relative to 2^w for large *w* this is approximately:

$$w \approx \log_2(\lambda / \ln 2)$$

The total time will be minimized when $w \approx \log_2(\lambda/\ln 2)$. So if the network latency λ is large, this suggests a larger value for the parameter *w*. A larger value for *w* causes the algorithm to have *fewer rounds*. Since the network latency is large, this will reduce total runtime since we pay the network latency cost less often. Conversely if the network latency is small, then we select a smaller *w*, do more rounds of communication, and pay less per round since the latency is small.

We examined the full parameter space for w=1,2,3,…,17 for a scaling study on the Dawn supercomputer [6] for processors counts $2^0, 2^1, 2^2, \ldots, 2^{17}$. Figure 38 shows the results of this scaling study.

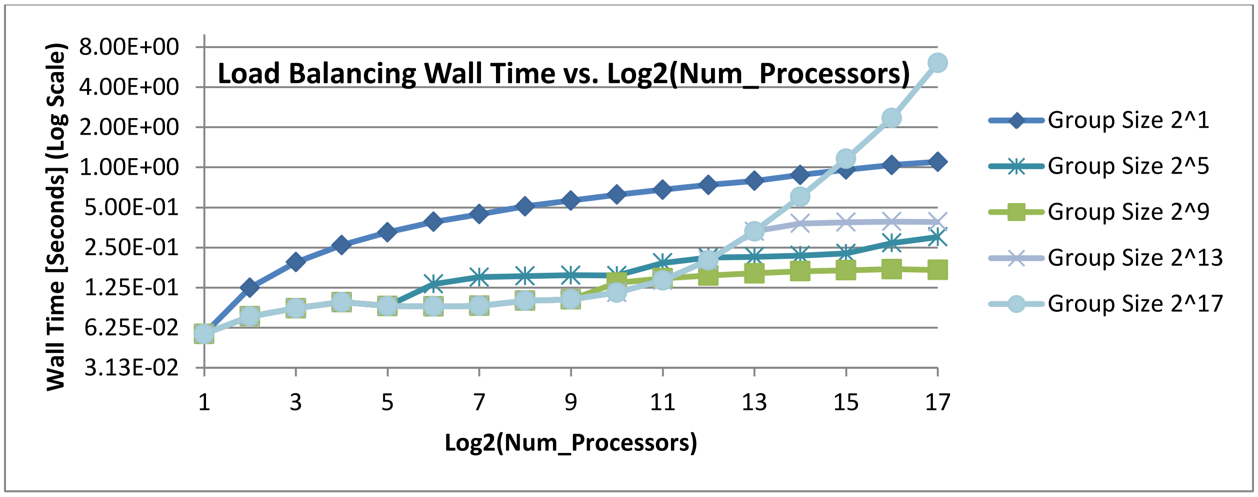


Figure 38: Load balancing wall time vs. $\log_2$(Num_Processors).

Figure 38 shows 5 curves, each curve has a different value for the $w = width$ parameter. When the width parameter is w, the Group Size is 2^w. When the Group Size = 2^1, this is exactly the same as the *partner processor* algorithm described in the beginning of this chapter, and we obtain the exact same result (this time plotted on a log-log scale, whereas before it was plotted on a log-linear scale in Figure 34). We see the logarithmic runtime $\Theta(log_2(N))$ when $w=1$, just as before. The group size cannot be *larger* than the total number of processors, so we limit the group size to be at most the number of processors. When the group size is 2^{17}, we see that the algorithm performs scalably up to about 2^{10} processors, but at large numbers of processors we see non-scalability of that parameter choice, since in this case the runtime scales like $\Theta(N \log(N))$. When the group size is 2^5, we see the "step function" behavior of the run time at the multiples of 5 on the $\log_2$(Num_Processors) scale. These discontinuities in run time happen when we need to do one more round of communication; recall $NumRounds = Ceiling(log_2(N)/w)$, so we see the discontinuities of the ceiling step function show up in the graph. The same discontinuities occur with Group Size 2^9 when $\log_2$(Num_Processors)=9 and with Group Size 2^{13} when $\log_2$(Num_Processors) =13. On the Dawn supercomputer, the best choice of Group Size is

2^9, since that curve is almost always the fastest for all processor counts (except when $\log_2$(Num_Processors)=10 in which case Group Size 2^{17} (which gets floored to 2^{10}) is slighly faster). Therefore, instead of using the basic *partner processor* load balancing algorithm, we balance in groups of size 2^9, and pay slightly more per round, but with the big advantage that we have 9 times fewer rounds.

Now we will look at the same data from a different perspective. Instead of each curve representing the same group size, in Figure 39 each curve represents the same number of processors, and we vary the group size as a parameter and look for a minimum.

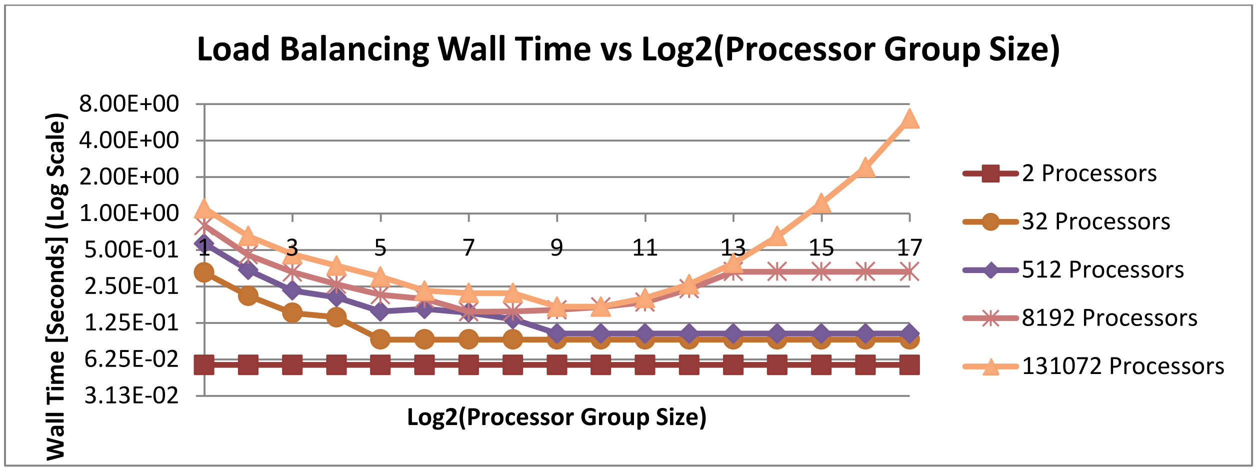


Figure 39: Load balancing wall time vs. $\log_2$(Processor Group Size)

Figure 39 shows 5 curves, varying the Processor Group Size to find the value of that parameter that minimizes the run time. The Processor Group Size cannot exceed the total number of processors, so the Processor Group Size is always bound by the number of processors. Examine the plot for 2^{17}=131072 processors, recall *NumRounds = Ceiling($\log_2(N)/w$)*, so when

$N=2^{17}$, NumRounds = Ceiling(17/w). As $w=Log_2(ProcessorGroupSize)$ increases from 1 to 9, we see the time decreasing like *Ceiling(17/w)* (in fact we see a plateau at *w=6,7,8* since *Ceiling(17/6) = Ceiling(17/7) Ceiling(17/8) = 3*). The cost of each round is $w2^w + \lambda$, and $w2^w$ is small compared to λ when *w=1,2,…,9*. As *w* continues to increase, the cost of each round becomes significant, and the fact that we need fewer rounds does *not* make up for the extreme cost of each round. For *w=10,11,12,…,17,* the run time increases since each round is so expensive. For all processor counts, the minimum is attained at $w = Log_2(ProcessorGroupSize) = 9$, so we have chosen a default value of $w = 9$ for the Dawn supercomputer.

4.7 Conclusions

We have developed, implemented, and demonstrated correctness of a scalable load balancing algorithm that has a computational complexity of *O(log(N))*. This algorithm globally balances all of the processors' work without globally knowing what the work distribution looks like. Through a sequence of local operations, global load balance can be achieved. We ran a scaling study up to 2^{21}=2,097,152 processors on the Sequoia (IBM BG/Q) supercomputer at Lawrence Livermore National Laboratory, and the observed results agree very well with the theoretical predictions. We believe that this algorithm applies to a broad class of homogeneous load balancing problems where the units of work all cost the same amount and any processor can process any unit of work. This algorithm allows for load balanced computations on the next generation of supercomputers with millions of cores. The new algorithm represents a significant improvement over our previous algorithm, which would have taken 17 days on 2 million processors of Sequoia, while this new algorithm takes less than 1 second.

We implemented a parameterized version of the load balancing algorithm where the user may select the *Processor Group Size* parameter which is the number of processors to group together when doing a local load balance step. Instead of load balancing processors 2 at a time, the general algorithm load balances processors in groups of size 2^w. We analyzed a performance model of the algorithm to select the best choice of this parameter to minimize the run time. We ran a scaling study exploring the entire parameter space and found that a Processor Group Size of 2^9 was the best choice for the Dawn (IBM BG/P) supercomputer.

5 Scalable Global Particle Find

The problem that we are trying to solve in this chapter is how to communicate particles to the correct processor that owns the geometric space in which the particle is located. Particles may be sourced on a processor that does *not* own the background geometry of the particle, so the particle needs to be communicated to the processor that owns the background geometry. If a particle is created as a result of a nuclear collision of an in-flight particle with a background material, then the created particle is already on the correct processor, since that is a local operation. Particles can also be created from an *external user defined source*, where the particle's coordinates are defined by sampling a user specified probability distribution function. In this case we cannot guarantee that the particle's coordinate will be *local* to the processor that samples it, in which case the new particle will have to be communicated to the correct processor before it can be tracked.

The remainder of this chapter is organized as follows. Section 5.1 describes the algorithms needed to communicate particles to the correct processor. Section 5.2 shows the results of a scaling study where the algorithm was run up to 2^{18}=262,144 MPI processes. Section 5.3 gives the conclusions of this chapter and suggests future areas of work.

5.1 Globally Resolving Particle Locations on the Correct Processor

In the new particle sourcing algorithm, each processor samples from the user-specified spatial distribution which can create particles on the wrong processor in domain decomposed problems. (A particle is on the *right* processor if the particle's coordinate is owned by a domain on the processor, conversely a particle is on the *wrong* processor if its coordinate is not on a domain owned by the processor. Processors can only operate on particles that reside *within* the domains the processor owns.) Before Mercury can begin the particle transport, it must first communicate every particle to the correct processor that owns that portion of the geometry [52], [53].

Each processor owns some set of spatial domains that determines the background geometry on which the processor can track particles. If a particle is within that geometry, then that processor can work on that particle; if not, the particle must be communicated to the correct processor that owns that background geometry. Our old (non-scalable) algorithm for communicating particles to the correct processor stored bounding box information for *all* of the domains in the problem. This algorithm is not scalable, since this memory requirement grows linearly with the number of processors.

Figure 40 shows a particular test case where each processor sources in particles uniformly over the entire problem. Consider the lower left (red) domain in the figure. That domain sources particles uniformly over the entire problem, which can create particles that are outside of the red domain. Those particles need to be communicated to the correct processor that owns that section of geometry. Every processor is in a similar situation, so all of the particles must globally resolve their locations to end up on the correct processor.

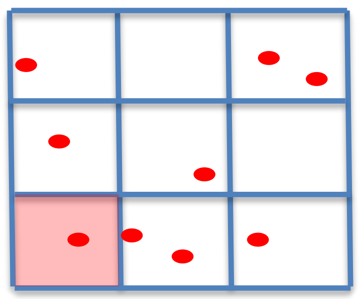


Figure 40: The lower left (red) processor can source in particles over the *entire* problem, not necessarily restricted to be within its geometry.

One possible solution to this problem is to reuse the existing particle tracking software to communicate particles to the correct processor. Figure 41 shows a diagram of this approach. Each particle is marked as being in the red domain, then the particle's trajectory and time to finish is reset and it is tracked to the correct location. When it reaches the correct final location, the trajectory is reset to the original value. This approach is appealing, because it reuses the existing particle tracking code and works well if the particles are "close" to the correct domain. But we will show that this algorithm is not scalable if most of the particles are "far" from the correct domain. This algorithm also does not work for non-convex problems.

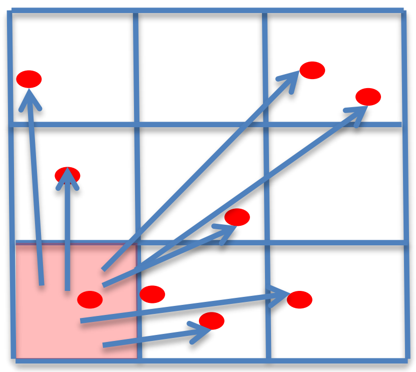


Figure 41: Use the existing particle tracking software to communicate particles to the correct processor.

<u>Claim:</u> Using particle tracking to resolve particle locations is not scalable as the processor count increases.

<u>Proof:</u> Consider a problem that has a 1D domain decomposition and a spatially homogeneous source, as show in Figure 42. Each processor creates particles uniformly over the entire problem, so the processors on the left half create NumParticles/2 total particles, NumParticles/4 of which are created on the right half, which is the half not owned by the processors! Similarly NumParticles/4 particles are created by the right half of the processors with particle coordinates on the left half, hence they are also on the wrong processor. So that means NumParticles/2 particles must be communicated through the center red domain, which is not scalable. QED.

The center processor sees too much of the global flow of particles, because particles can only flow to adjacent domains. One solution is to allow particles to "hop" to "distant" neighbors. By carefully choosing a domain's distant neighbors, we have designed an algorithm that globally resolves the particle locations in a scalable fashion.

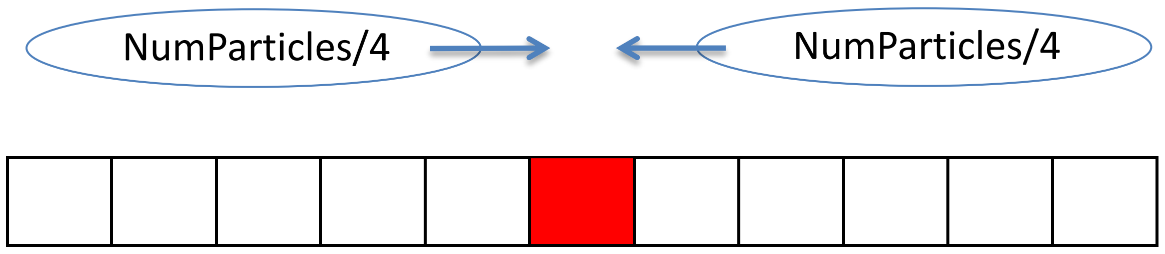


Figure 42: 1D domain decomposed problem showing a "bottleneck" at the center domain.

For the new scalable algorithm, each processor generates *O(log(N))* "distant" neighbors (where *N* = number of processors), to form a communication graph for sending particles to the correct processor. Each particle on the "wrong" processor is then sent to its *nearest* neighbor (among the domain face neighbors and the distant neighbors). This algorithm is iterated until each particle is on the correct processor.

This algorithm creates a hypercube graph to connect the processors, so in $O(\log(N))$ iterations, each particle is on the correct processor. A *hypercube* is a graph $G=(V, E)$ where

$V = \{0,1,\ldots,N-1\}$ = Set of processor ranks

$E = \{(a, b)$: a and b differ in exactly one binary digit$\}$

Figure 43 shows an example of a hypercube graph on 8 nodes. Let $N = |V|$ be the number of nodes in the graph; then there are $\log_2(N)$ digits in the binary representation of the processors' ranks $\{0 \leq rank < N\}$. Each node has $\log_2(N)$ neighbors. All nodes are at most $\log_2(N)$ hops apart, and this maximum distance occurs when two processor ranks a and b differ in *every* binary digit.

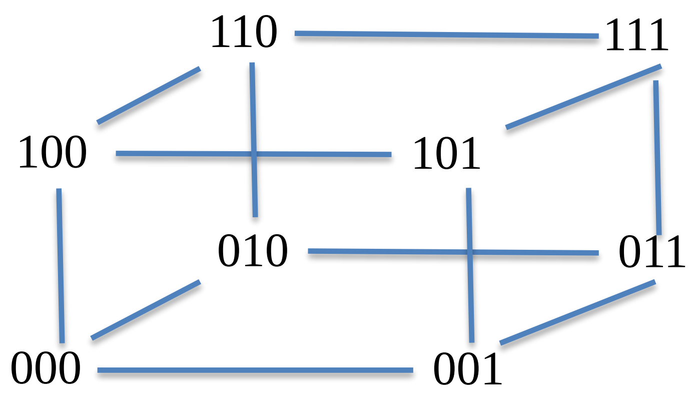


Figure 43: A hypercube on 8 nodes.

If a particle is not on the correct processor, then we must first locate the “closest” neighboring domain to which to send the particle. We are only dealing with Cartesian domain decompositions, so we have implemented a greedy algorithm that always selects the closest neighboring domain at every step. At each iteration, the particle gets closer to its home processor, so the algorithm is guaranteed to terminate. Figure 44 shows an example of the communication path a particle takes to get from the processor it was created on to the processor it belongs on.

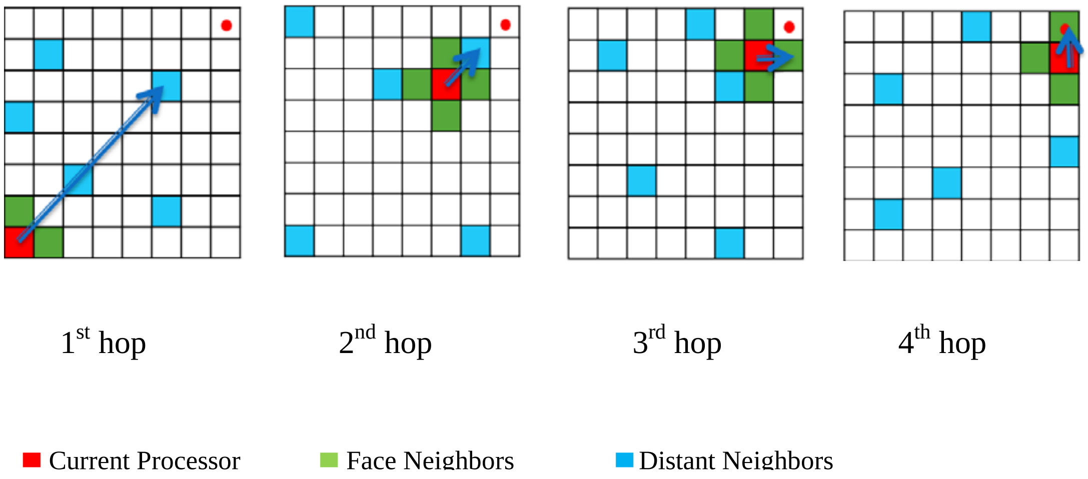


Figure 44: Example of a particle finding the “closest” neighbor to find its way to the correct processor.

The algorithm first checks to see if the particle coordinate is owned by the current processor; if so, keep the particle and begin the particle tracking. If not, we must find the neighboring domain that is “closest” to the particle’s coordinate. A naïve implementation would be to linearly search through a domain’s six face neighbors (a cube has six faces) and its $log_2(N)$ distant neighbors. But instead of linearly searching through the list of neighbors, we can put all $6 + log_2(N)$ neighbors in a *k-d tree*, and use that to search for the closest neighbor. A *k-d tree* is a data structure used for quickly finding an object based on position [54]. The objects can be partitioned in either the x, y or z direction, depending on which direction gives the most even partition of objects. Figure 45 shows an example of domains stored in a *k-d tree.* The vertical red line is the highest level partition, which divides the neighbors into 2 groups, those *less than* the line and those *greater than* the line. We proceed recursively, dividing each side. The horizontal green lines partition each half into domains *less than* and *greater than* the green lines. This is an efficient partitioning of the domains so that the algorithm can find the closest domain with a run time proportional to the logarithm of the number of entries in the *k-d tree*.

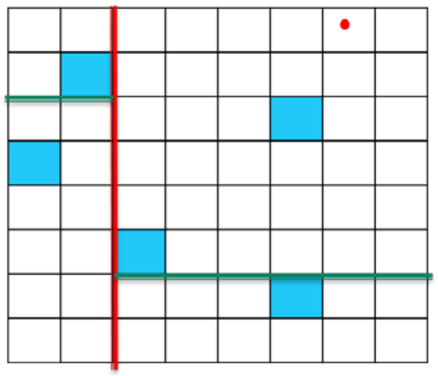


Figure 45: An example of domains stored in a *k-d tree.*

The time required to find the *nearest* neighbor is proportional to the log of the number of neighbors; we have *O(log(N))* neighbors, so the algorithm requires *O(log(log(N)))* time. The total runtime is the number of iterations (maximal distance between vertices in a hypercube graph = *O(log(N))*) times the cost per iteration or

$$\text{Total runtime} = O(\log(N)\log(\log(N)))$$

5.2 Results

In this section we examine an infinite medium weak scaling test problem. The test problem uses 3D domain-decomposed combinatorial geometry and is a Uranium-235 infinite medium (reflecting boundary conditions) criticality test problem. Each processor has 1 domain which is a 1cm cube. 10,000 particles are tracked per processor. Each time we double the processor count, we double the size of the geometry as illustrated in Figure 46. Each processor sources particles uniformly over the entire geometry, so the particles need to be communicated to the correct processor before transport can begin.

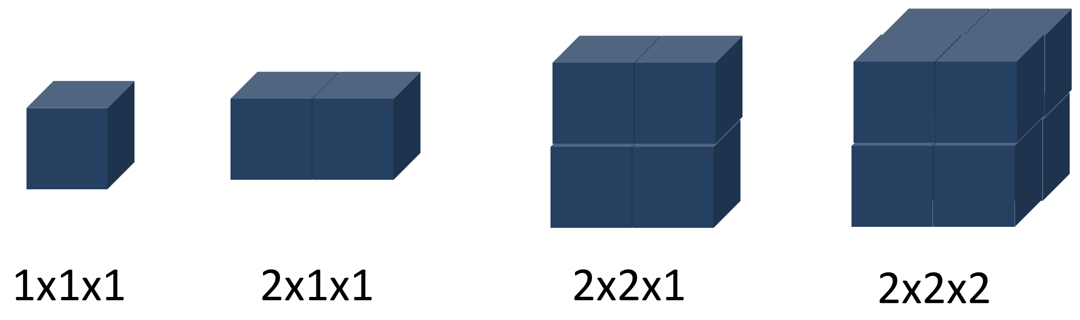


Figure 46: Illustration of geometry scaling with processor count scaling.

Figure 47 shows a graph of the distribution of the number of communication hops that are required to put every particle on the correct processor for different numbers of processors. Figure 47 is plotted on a linear scale and seeing the tails of the distribution is difficult. So Figure 48 plots the same information on a log scale. The log scale clearly shows the tails of the distribution. As the number of processors increases, more hops are required to communicate the particles to the correct processor. In fact, the *maximum* number of hops required by any particle is what determines the total number of communication iterations required to send the particle to the correct processor.

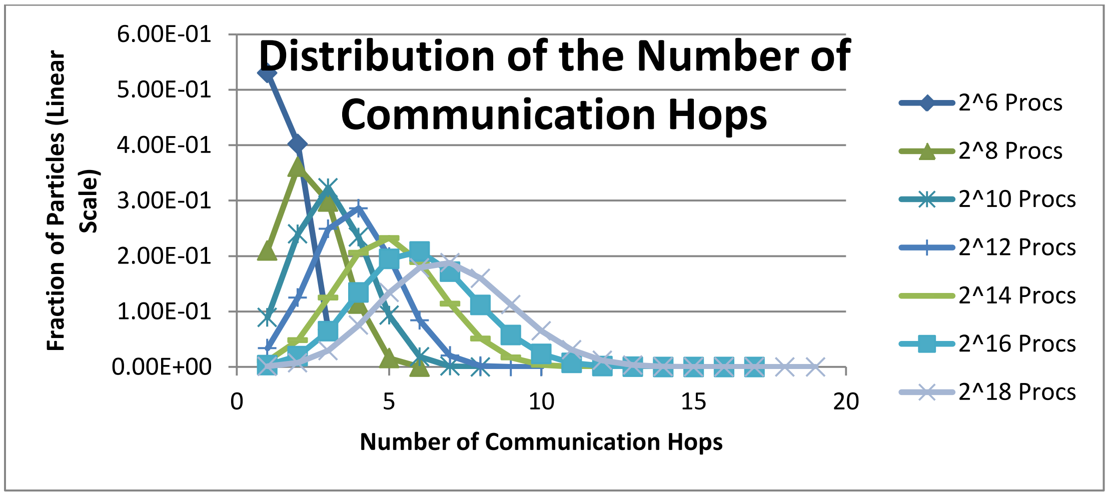


Figure 47: Distribution of the number of communication hops on a *linear* scale.

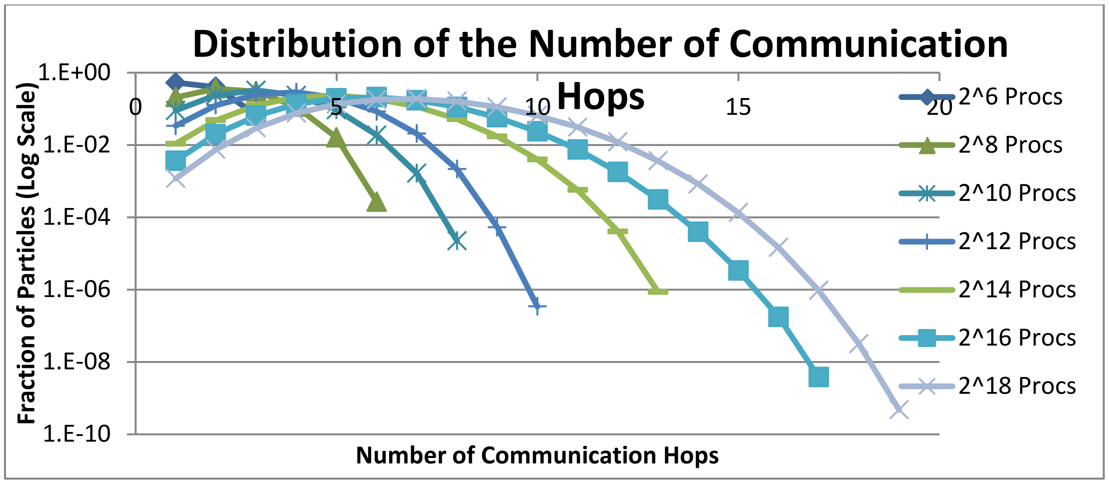


Figure 48: Distribution of the number of communication hops on a *log* scale.

Figure 49 is a plot of the maximum number of communication hops required to communicate all of the particles on the correct processor versus the number of processors. This graph shows the approximate relationship:

$$\text{Max number of communication hops} \approx \log_2(\text{Num\_Processors})$$

This result is a very nice property for scalability. We want the total runtime of the algorithm to be scalable, and in order for that to be possible, the total number of communication hops must also be scalable.

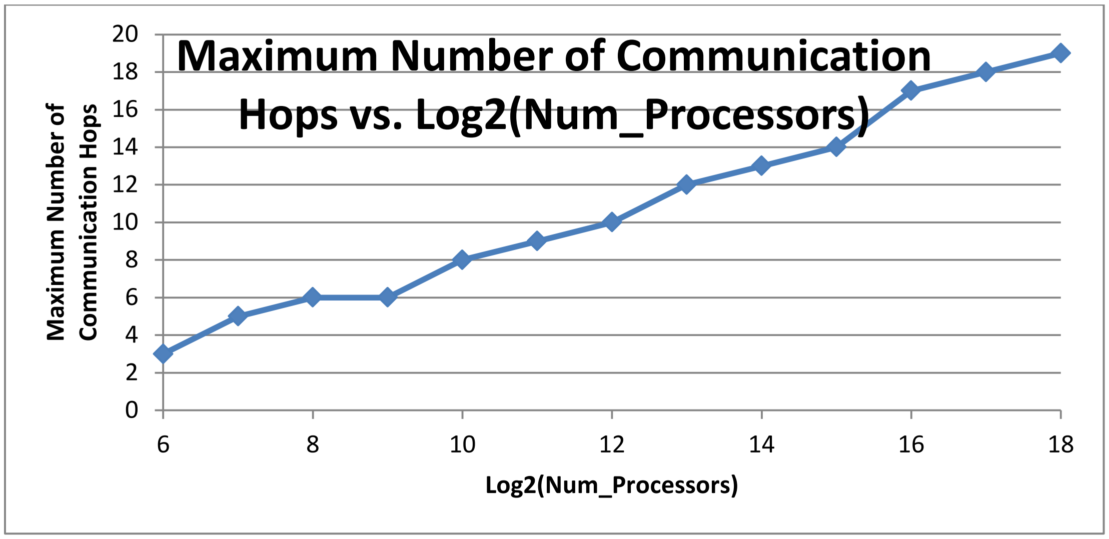


Figure 49: Maximum number of communication hops vs. Log2(Num_Processors).

Figure 50 shows the particle tracking time vs $\log_2$(Num_Processors) for this infinite medium problem. The particles are sourced on the *first* cycle. The first cycle is more expensive than subsequent cycles, since particles must first be communicated to the correct processor before the particle physics can begin. The scaling behavior of the first cycle shows the particle tracking cost plus the cost of communicating particles to the correct processor. This test problem is a weak scaling test problem, so the particle tracking cost should be exactly the same independent of the number of processors. As a result, any increase in wall time when the number of processors is increased is due to the overhead of the communication algorithm to send particles to the correct processor. At the low end of the scale, $2^6 = 64$ processors, the first cycle takes 19 seconds. At the high end, 2^{18} = 262,144 processors, the first cycle takes 199 seconds. This domain decomposed Monte Carlo particle transport on a quarter of a million processors is a challenging problem. Future work will focus on improving this algorithm. The second cycle and last cycle scale very well. The only parallel overhead in those cases is the particle streaming communication and the "test for done" algorithm, which is described in Chapter 7. No particle source exists after the first cycle, so the "particle hopping" algorithm is not used after the first cycle of this problem. The second cycle and last cycle curves demonstrate excellent scaling of the particle streaming communication and of the new scalable test for done algorithm.

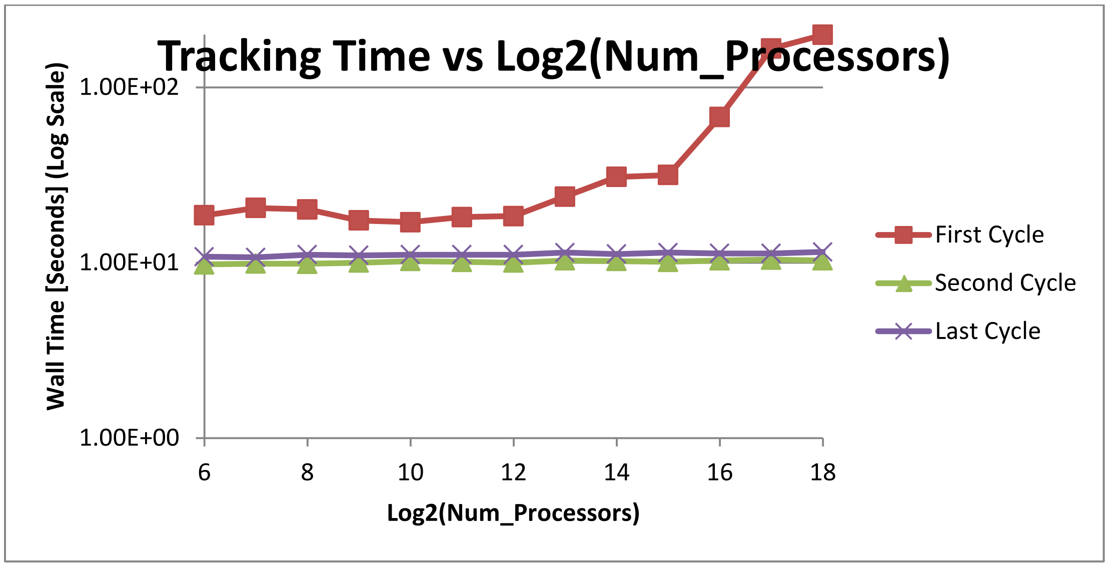


Figure 50: Tracking Time vs. Log2(Num_Processors).

Figure 51 shows the results of a weak scaling test for which we run up to 2^{15}=32,768 processors. Each processor owns 1 out of N domains, but sources particles uniformly over the entire problem space. Therefore most particles are initially on the wrong processor and need to be communicated to the correct processor. This plot shows the scaling behavior of communicating particles to the correct processor. Up to 32,768 processors, this algorithm looks very scalable. However, Figure 50 shows three more data points at 2^{16}, 2^{17}, and 2^{18} processors that show significant increase in time compared with the smaller processor counts.

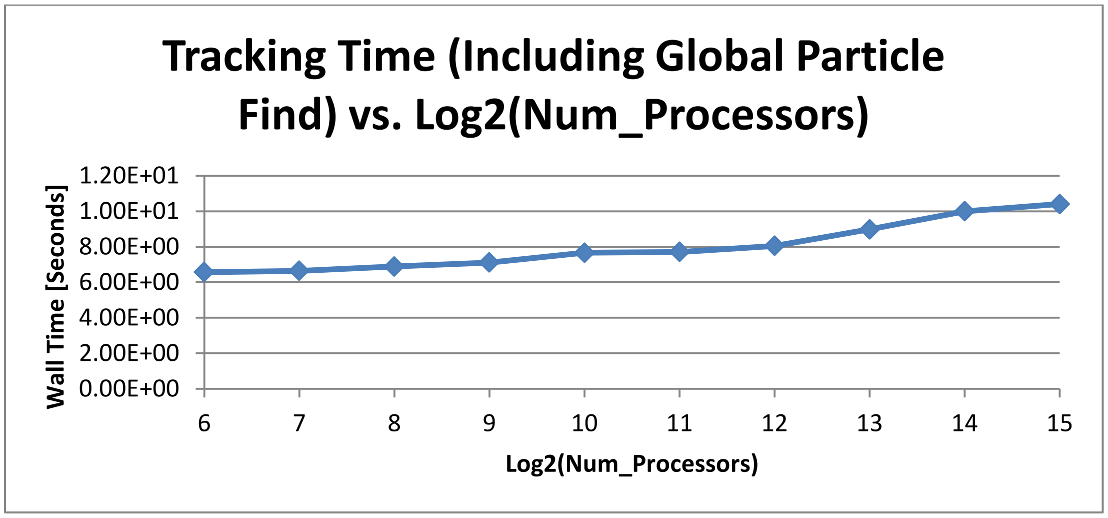


Figure 51: Scaling results of the new global particle find algorithm.

5.3 Conclusion

We have implemented a scalable algorithm to globally resolve particle locations and to communicate particles to the correct processor. A crucial part of the algorithm is the introduction of "distant neighbors" that allow particles to take larger "jumps" toward their correct domain. The distant neighbors are formed by creating a hypercube graph, where the maximum distance between any two nodes in the graph is *O(log(N))*. A weak scaling study was run on the algorithm up to 2^{18} = 262,144 MPI processes on an infinite medium test problem. We observed excellent scaling behavior up to 2^{15} = 32,768 MPI processes and observed some slowdown after that. In future work we will experiment with different distant neighbor networks to determine if the scalability to large numbers of processors can be improved.

6 Visualizing Constructive Solid Geometry

6.1 Introduction

Validation of the problem definition and analysis of results (tallies) produced during a Monte Carlo particle transport calculation can be a complicated, time-intensive processes. The time required for a person to create an accurate, validated combinatorial geometry (CG) or mesh-based representation of a complex problem, free of common errors such as gaps and overlapping cells, can range from days to weeks. The ability to interrogate the internal structure of a complex, three-dimensional (3-D) geometry, prior to running the transport calculation, can improve the user's confidence in the validity of the problem definition. With regard to the analysis of results, the process of extracting Monte Carlo physics tally data from printed tables within a file is laborious and not an intuitive approach to understanding the results. The ability to display tally information overlaid on top of the problem geometry can decrease the time required for analysis and increase the user's understanding of the results.

In this chapter, we discuss the integration of VisIt [55], [56], [57], [58], [59] a parallel, production quality full featured visualization and data analysis tool into Mercury, a massively parallel Monte Carlo particle transport code. VisIt provides an API for real time visualization of a simulation as it is running. A user may select which plots to display from the VisIt GUI or by sending VisIt a Python script from Mercury [60]. A user may set the frequency to update the plots and watch the simulation evolve as it is running.

Rather than reinventing the wheel by writing custom software to visualize our Monte Carlo particle transport calculation results, we chose to use VisIt, an existing scientific visualization and data analysis tool. Mercury was already able to write restart files and graphics files that could be opened for post-processing visualization in VisIt. This chapter focuses on connecting Mercury and VisIt *in memory*, through VisIt API function calls which give VisIt the data that it needs for plots, based on user requests.

VisIt is capable of visualizing domain decomposed mesh based data on structured and unstructured meshes. This capability is used to visualize Mercury's *mesh*-based geometry. VisIt can also visualize "point" based data, which is used to visualize Mercury's *particle*-based data. A relatively new feature in VisIt is the ability to discretize and visualize Constructive Solid Geometry (CSG) data (a.k.a. Combinatorial Geometry or CG). An API exists to give VisIt the coefficients of the surfaces that define the cells and to tell VisIt how the surfaces are combined together to form cells. VisIt will then automatically discretize and visualize the CG data. Before VisIt had the ability to directly visualize CG data, Mercury would convert its internal CG data into mesh based data by introducing a graphics mesh over the CG and sampling the CG at the mesh points. If a mesh cell intersected multiple CG cells, a "mixed" mesh cell is created that contains the volume fractions of the partial CG cells. This mesh sampling algorithm will be described in detail later in Section 6.4.

Mercury has an interactive Python prompt, and a user may issue the "visit()" command to launch VisIt and have it connect to the simulation. Then a user may select the plots they would like from VisIt's GUI. Alternatively, a user may request plots directly from the Mercury Python prompt by feeding VisIt a Python script, such as: visit('myScript.py'). This will hand the contents of the python script 'myScript.py' to VisIt for visualization. As part of the Mercury input, a user may set the frequency at which the VisIt plots are updated as the simulation is running.

The remainder of this chapter is organized as follows. Section 6.2 highlights some of the most useful plots for Mercury users of VisIt, such as a particle pseudocolor plot and a background material plot. Section 6.3 discusses the inline Mercury-VisIt interface by which a user may visualize the simulation results in real time with VisIt. Section 6.4 covers all of the available methods for visualizing constructive solid geometry. Some of the methods are based on introducing a *graphics mesh* on which to overlay the CG, while other methods use ray casting or rely on VisIt itself to discretize the CG. Section 6.5 provides an analysis of the numerical convergence of the volume calculation of CG cells using the mixed cell method. Section 6.6 discusses geometry error detection by the automatic detection of *gaps* and *voids.* Section 6.7 is a gallery of images that have been produced with Mercury and VisIt. Finally Section 6.8 provides the conclusion of the combinatorial geometry visualization chapter.

6.2 Types of Visit Plots

VisIt has many different types of plots such as mesh, material, pseudocolor, domain, curve, histogram, contour, particle, vector, label and volume rendering. VisIt also has many operators that a user can apply to the plots such as clipping and material sub selection. Material sub selection is a very useful operator in VisIt. This approach is used for turning on or off any material in the problem, so a user can focus on exactly what they are looking for while hiding what they are not interested in.

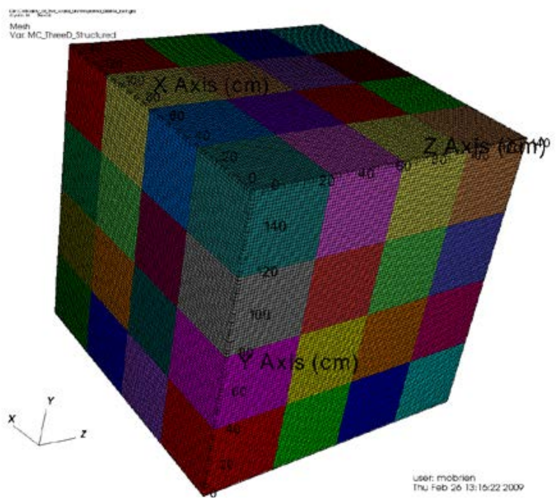
We will now illustrate many of the available plots in VisIt as applied to the **"Criticality of the World"** test problem. The problem definition is a 7 x 7 x 7 lattice of Uranium-235 spheres, each of radius 5.0cm, except for the center sphere which is of radius 8.7407cm. The density of all of the spheres is 19.1 g/cm^3. The centers of all of the spheres are 24cm apart. Low density Uranium-235 (10^{-10} g/cm^3) surrounds the lattice. All of the spheres are subcritical except the center sphere. The initial source of particles is the lower corner sphere. We run a Static-K calculation and watch the particles “find” the center sphere. We run the problem with 100,000 simulation particles per iteration. Note this is *not* the same Criticality of the World test problem described in Chapter 2.

Mesh Plot

As part of the user input, we have requested a 100x100x100 cell graphics mesh to be overlaid on the CG for visualization. See Figure 52 for two plots:

a) We ran this problem on 64 processors, so the graphics mesh is automatically divided into 64 domains; each processor samples the CG for only its domain.

b) Here we are plotting the mesh and materials, but we have used VisIt's material sub selection feature to turn off the low density Uranium.



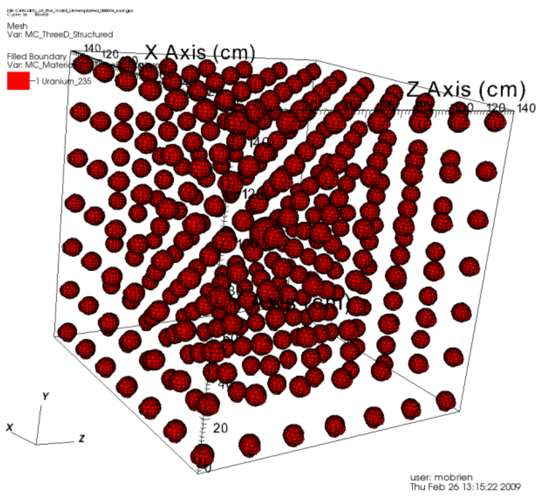


Figure 52: (a) Mesh + Domain plot. (b) Mesh + Material plot, low density U turned off.

Material Plot

Figure 53 is a material plot of both the underlying geometry and of the particles. Every graphics mesh zone has a material associated with it (or multiple materials for mixed cells). Every particle also has a background material associated with it. We have lowered the opacity of the low density filler Uranium to make it look transparent. We have turned off the high density Uranium completely. We are also plotting the particles by material, so we see the particles in place of the high density Uranium. The particle distribution has not yet converged, so we can see some remnants of the initial particle source in the lower-left sphere.

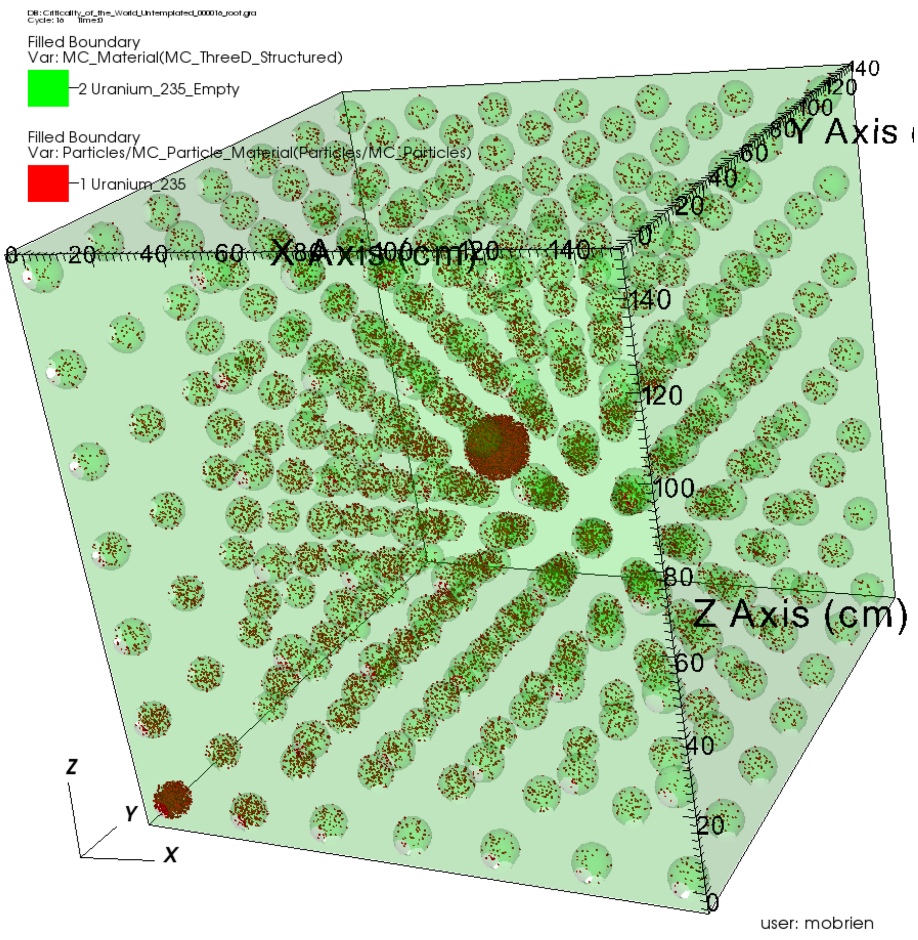
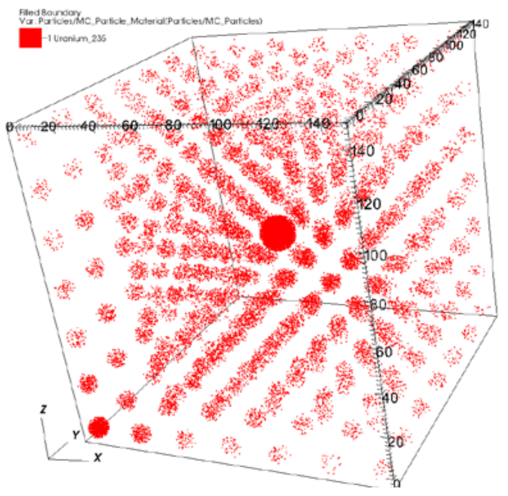
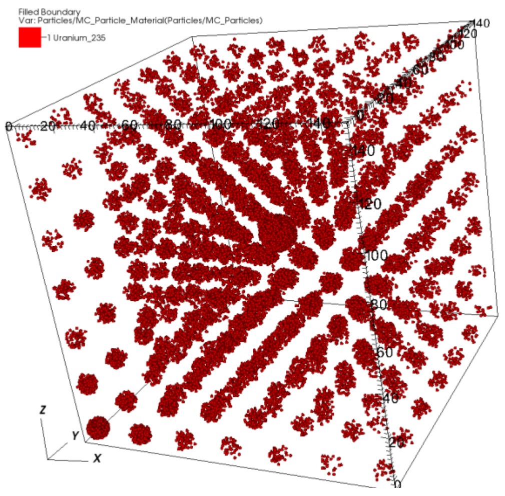


Figure 53: Transparent material plot.

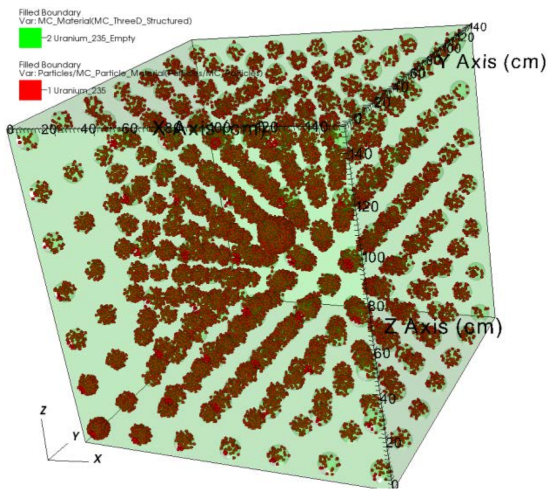
When plotting particles in VisIt, a user may plot them simply as “points” as in Figure 54 (a), or they may plot them as “spheres” as in (b). In (c) we are plotting both the particle material and the background material. In (d) we are plotting the background material, with the low density Uranium with reduced opacity so we can see through it.



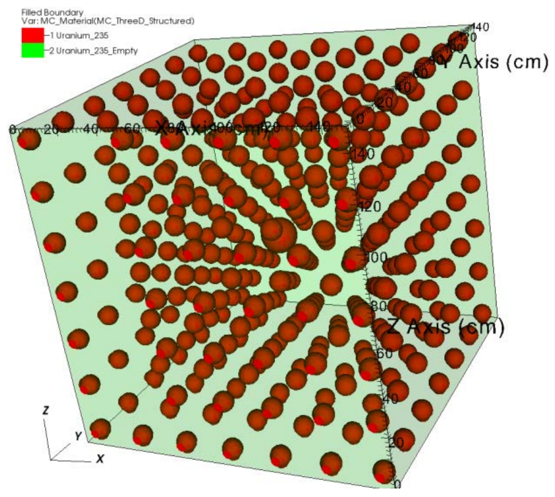
(a) Particle Material, plotted as “points”.



(b) Particle Material, plotted as “spheres”.


Filled Boundary
Var: MC_Material(MC_ThreeD_Structured)
2 Uranium_235_Empty
Filled Boundary
Var: Particles/MC_Particle_Material(Particles/MC_Particles)
1 Uranium_235
Y Axis (cm)
X Axis (cm)
Z Axis (cm)

(c) Particle and background material.



(d) Background material.

Figure 54: Various material plots.

Pseudocolor Plot

A pseudocolor plot maps values to colors. We have used the material sub selection to turn off the filler material. Figure 55 has two plots:

(a) We are plotting CG cell index. We can see that the range of values is 0 to 343, for a total of 344 cells, which is $7^3 + 1$ (7 by 7 by 7 lattice + 1 for filler). This plot is extremely useful for debugging. This plot along with VisIt's "Pick" tool allows a user to click on a cell, and VisIt will tell the user the CG cell index.

(b) This is a pseudocolor plot of the log of neutron number density, which is essentially the eigenvector associated with the K_{eff} eigenvalue. VisIt allows linear or logarithmic color scales, which can be used to "spread" out the color values, depending on the range of values in the data. In this case the "log" scale shows more color range than the "linear" scale.

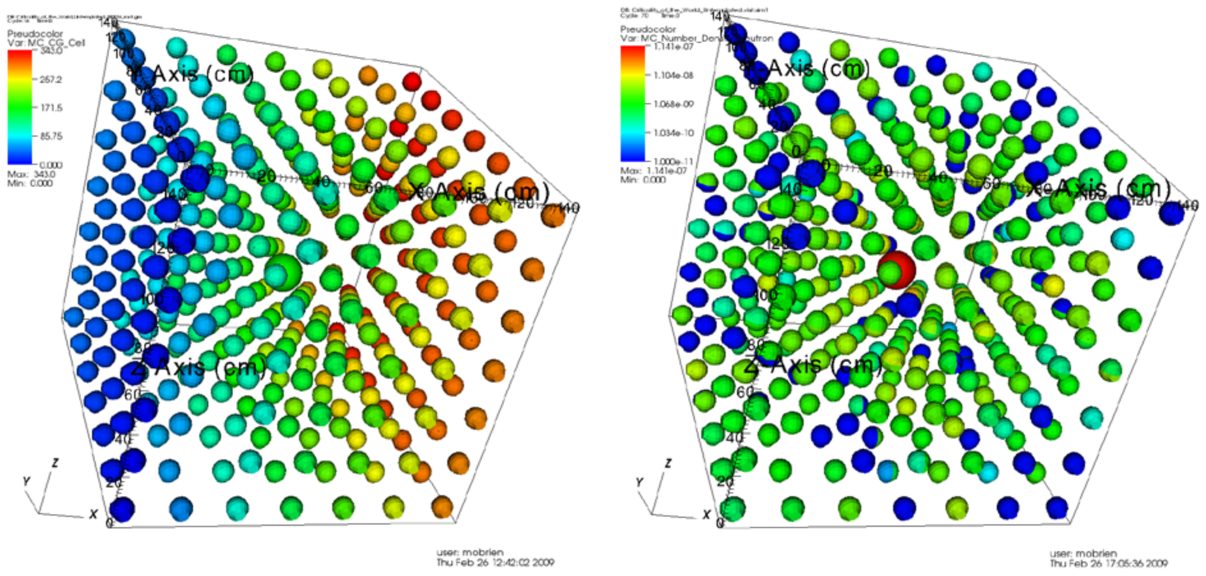


Figure 55: (a) Pseudocolor plot of CG cell index. (b) Pseudocolor plot of log(neutron number density)

Domain Plot

The user specifies the physical extents of the graphics mesh and the number of zones in each (x,y,z) direction. Then Mercury automatically domain decomposes the *graphics mesh* into N domains, where N is the number of processors the simulation is run on. In this case, we ran the problem on 64 processors, so we have 64 domains. In Figure 56, we are coloring each domain a different color to show the domain decomposition of the graphics mesh. This makes the generation of the graphics mesh faster, since each processor has to discretize only a portion of the entire problem. Chapter 2 describes the combinatorial geometry domain decomposition in detail. In Figure 56(b), we have turned off the low density filler material and are coloring the spheres by domain. Figure 56(b) also shows a wireframe plot of the domain boundaries.

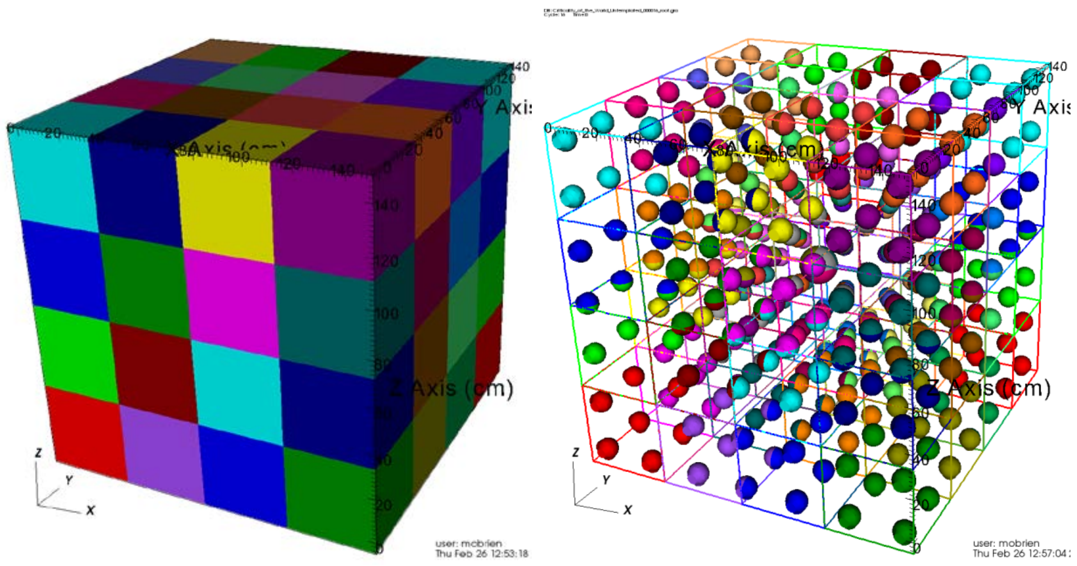


Figure 56: (a) Domain plot. (b). Wireframe domain plot with filler material hidden.

Curve Plot

Figure 57 is a plot of the iteration history of:

```
1. Instantaneous K (red)
2. Average K (green)
3. Average K + Standard Deviation (cyan).
4. Average K – Standard Deviation (blue).
```

As the iteration count increases, the standard deviation decreases and Average K converges [43]. A "discontinuity" of Average K occurs at 34 iterations, because we do 34 initial "transient" iterations to let the particle distribution become stationary. We then throw out the transient iterations and compute Average K and the Standard Deviation of K *after* the transient iterations.

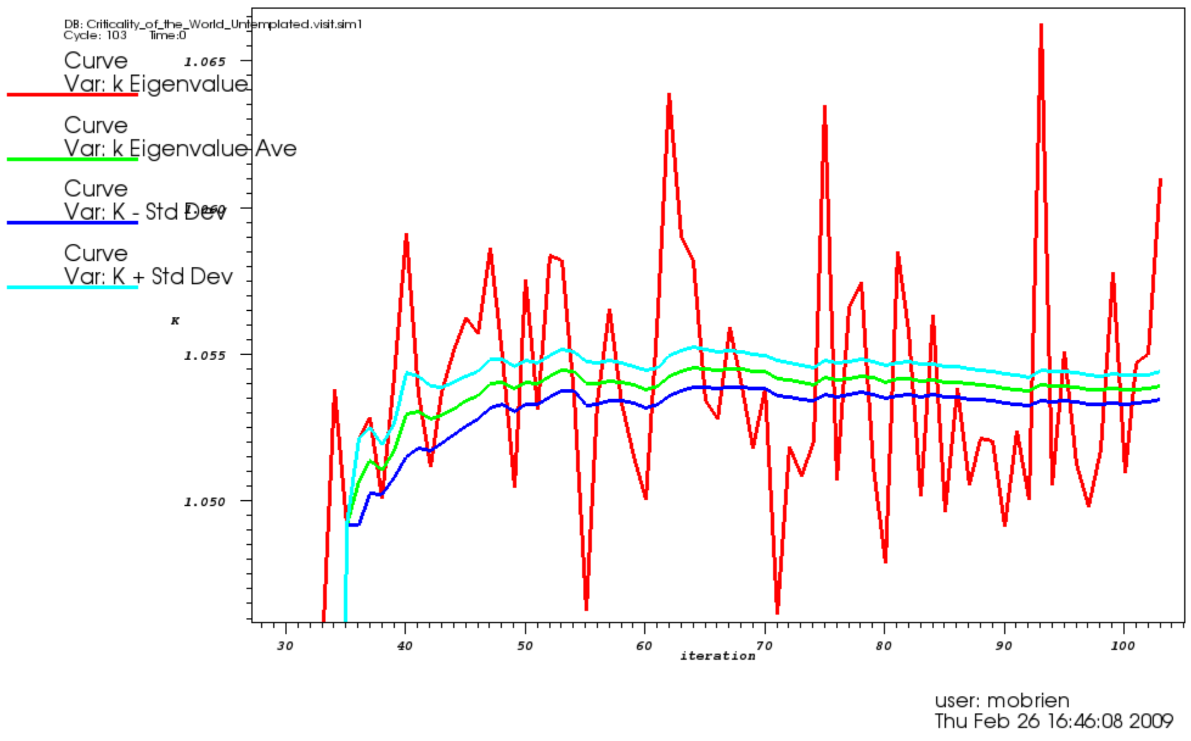


Figure 57: Curve plot.

Histogram Plot

Figure 58 is a histogram of the number of particles at a given kinetic energy, on a log-log scale. This type of plot is very useful for seeing the energy distribution of the particles. The histogram plot is also useful for looking at the particle weight distribution. A user can make a histogram plot of any particle or mesh based data.

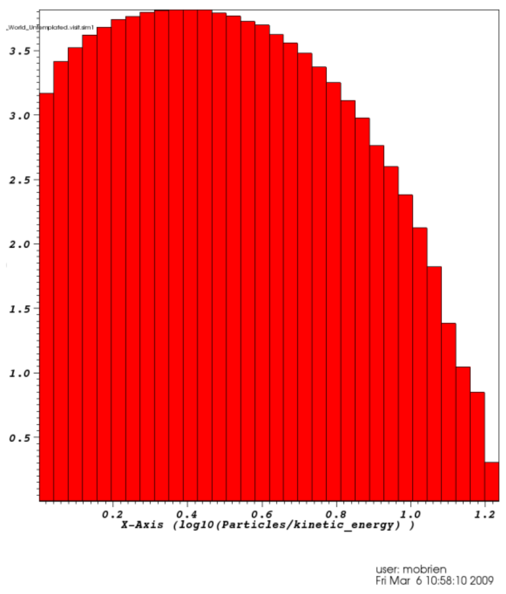
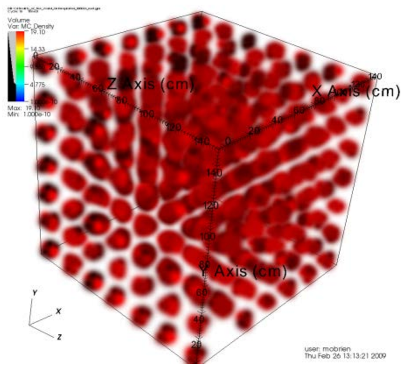
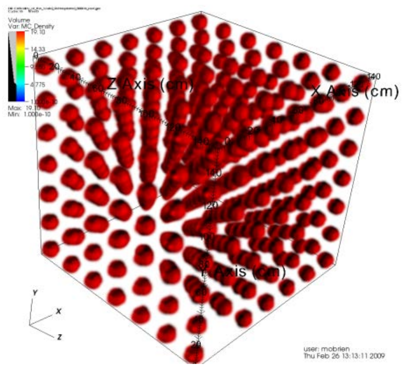


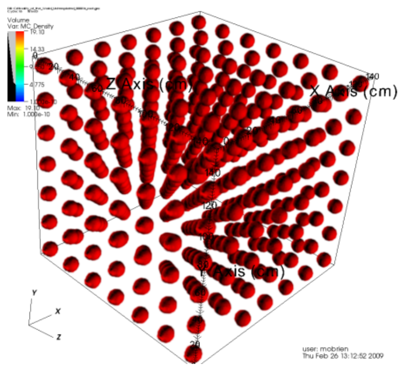
Figure 58: Histogram plot.

Volume Rendering Plot

Visit can do volume rendering of *mesh* based data as in Figure 59 (a), (b), (c) and (e), or *particle* data as in (d). In (a), (b) and (c) we are looking at mass density, as we vary the number of volume rendering samples. With a low number of samples (as in (a)), the image looks fuzzy. As a user increase the number of samples, the image becomes sharper (as in (c)).


(a) 50,000 Samples.


(b) 500,000 Samples.


(c) 5,000,000 Samples.

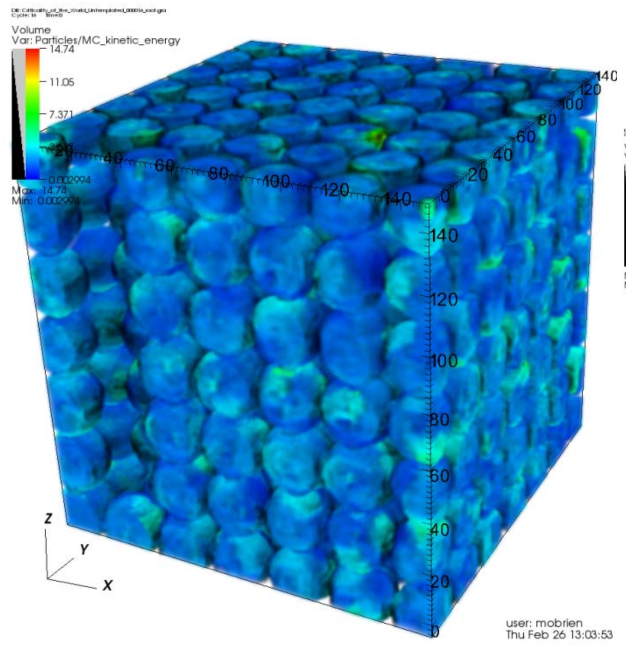
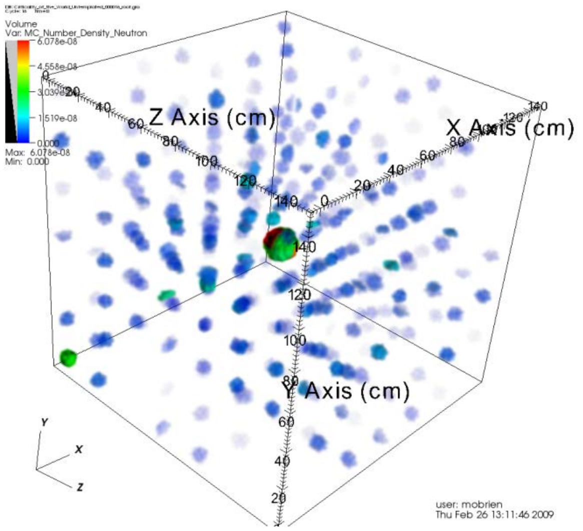


Figure 59: (d) Volume rendering of particle KE.

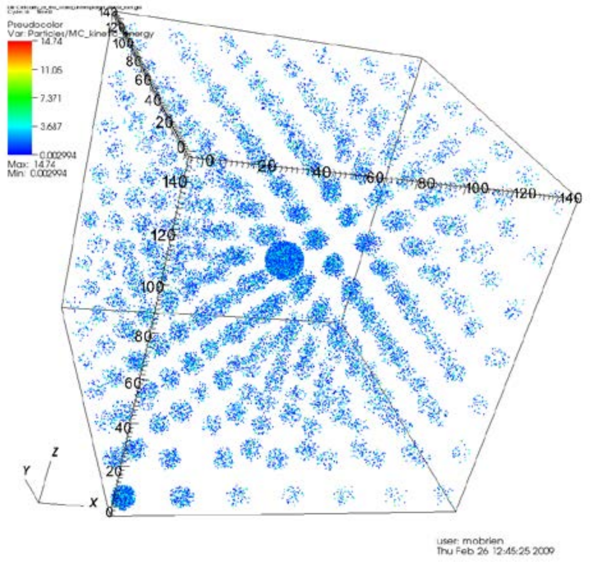

(e) Volume rendering of neutron number density.

Particle Plot

The Monte Carlo particles can be colored by any field defined on the particles. Figure 60 shows two plots:

(a) Shows the particles colored by Kinetic Energy. This gives a user an indication of the particle distribution in energy and space. A concentration of particles exists in the center sphere, and since the eigenvector has not yet converged, some remnants of the initial particle source exist in the lower left sphere.

(b) Shows the particles colored by which CG cell they are in. This is very useful for finding particle tracking bugs. The human eye easily spots an incorrectly colored particle. A user can also use "discontinuous" color scales so adjacent cells have drastically different colors instead of continuously varying colors, which would make it harder to identify a particle colored by the incorrect cell index.



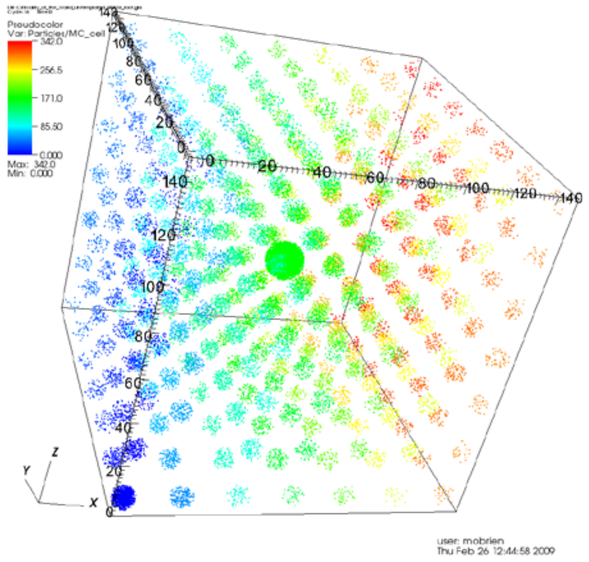


Figure 60: (a) Particles colored by KE. (b) Particles colored by CG cell index.

Vector Plot

Figure 61(a) shows each particle's velocity vector. Since this data can be very dense, VisIt has an option to specify the stride of which particles to draw. For example, in Figure 61(b), the stride is 16 and we are plotting every 16th particle. The vectors are colored by magnitude.

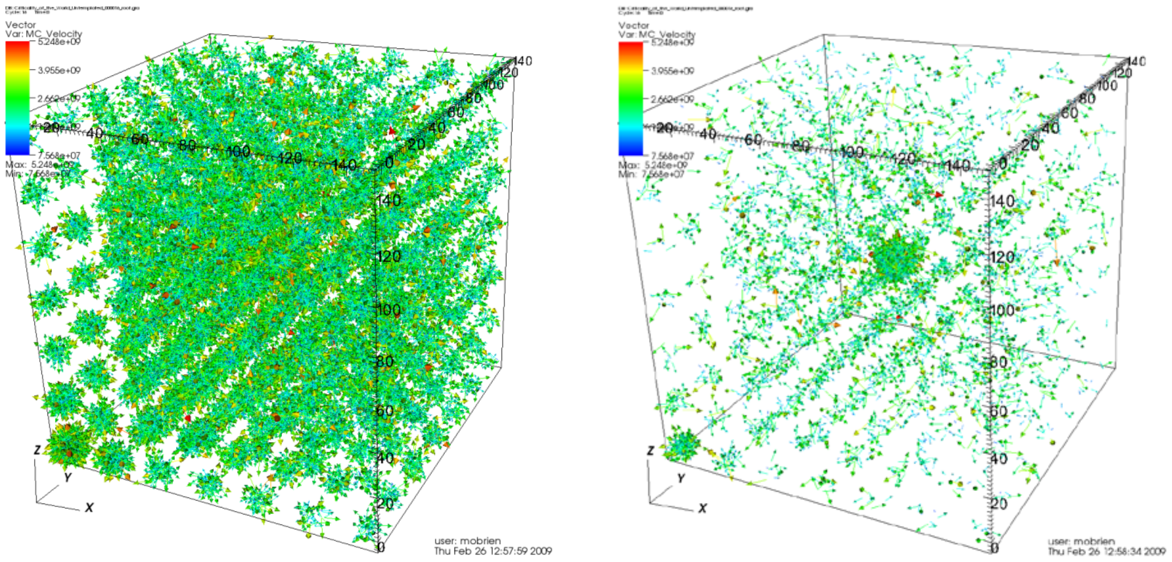


Figure 61: (a) Every particle. **(b) Every 16th particle.**

6.3 Using the Inline Interface

6.3.1 Time Evolution of the Criticality of The World Problem

Figure 62 shows screen captures of a running simulation, as the Static-K iterations are increasing. At each iteration, we plot four quantities in different VisIt windows.

Upper left: Pseudocolor plot of log(neutron number density)	Upper right: Curve plot of iteration history of: K, Average K, Average K ± StdDev
Lower left: Particle plot of log(particle KE)	Lower right: Curve plot of Flux Entropy (eigenvector convergence diagnostic).

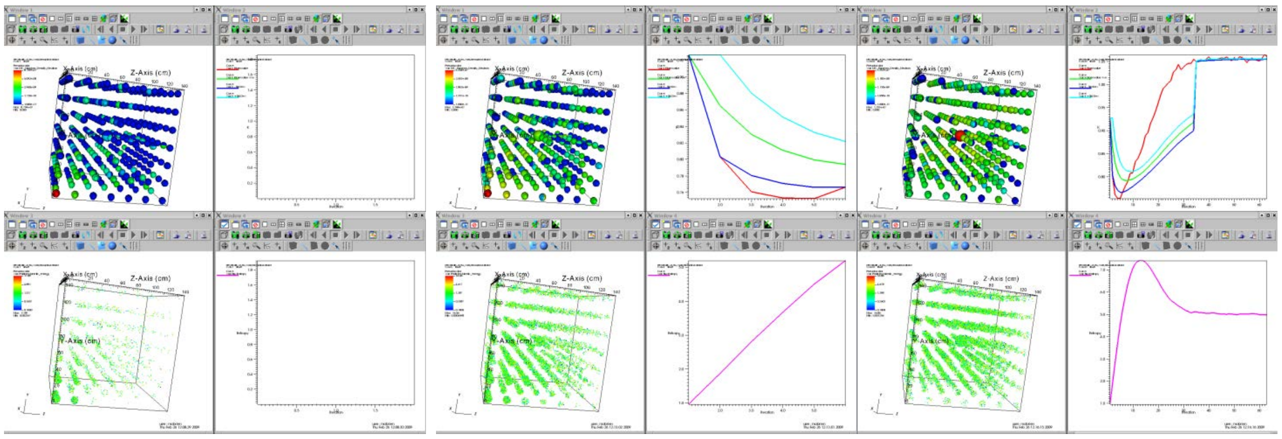


Figure 62: (a) Iteration 1 **(b) Iteration 6** **(c) Iteration 63**

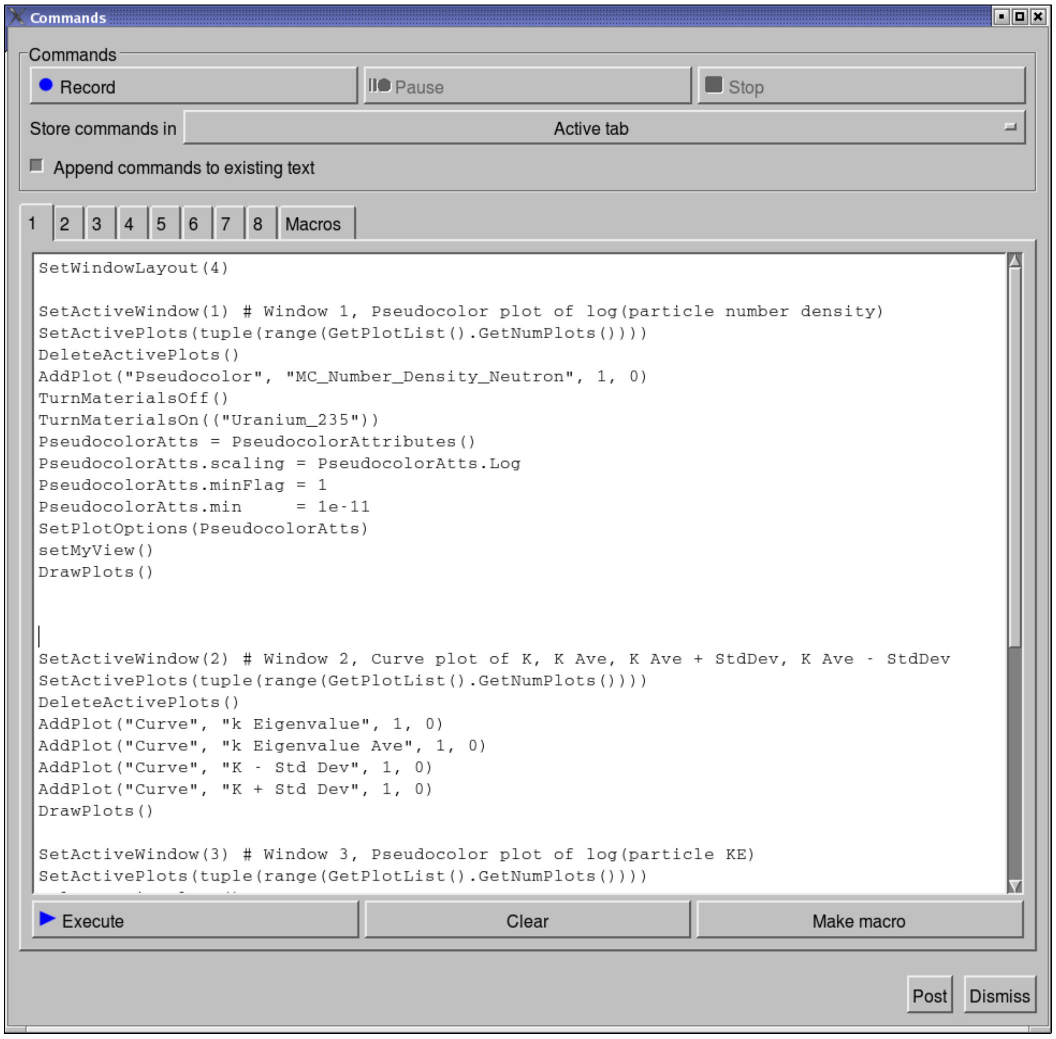
Figure 62 shows the evolution of these plots at iterations 1, 6, and 63.

(a) Iteration 1: We do not have enough data for the curve plots, so they are blank. After only 1 iteration, the particles are still concentrated around where they were initially sourced, in the lower left sphere.

(b) Iteration 6: Particles are still wandering around, being concentrated around the lower sphere**.**

(c) Iteration 63: The particles have “found” the center sphere and the particle distribution is stationary. A discontinuity exists at iteration 34 of Average K, since we told Mercury to do 34 initial “transient” iterations so the particles could find their natural distribution for this problem. After iteration 34 we begin averaging to calculate Average K.

Figure 63 is VisIt's user interface window that allows a user to *record* their mouse clicks, and it will translates a user's mouse clicks into a python script. The script can then be "played" back to achieve the exact same behavior as the mouse clicks. A user can also save the text to a python file that can be handed back to VisIt from the interactive Mercury python prompt, as shown in Figure 64. For example, we save the script to a file named 'cow.py', and then at the Mercury python prompt, we type visit('cow.py') to feed the python script to VisIt. As a result, we can quickly recover all of the plots, without having to re-select them in the VisIt GUI.



```
SetWindowLayout(4)

SetActiveWindow(1) # Window 1, Pseudocolor plot of log(particle number density)
SetActivePlots(tuple(range(GetPlotList().GetNumPlots())))
DeleteActivePlots()
AddPlot("Pseudocolor", "MC_Number_Density_Neutron", 1, 0)
TurnMaterialsOff()
TurnMaterialsOn(("Uranium_235"))
PseudocolorAtts = PseudocolorAttributes()
PseudocolorAtts.scaling = PseudocolorAtts.Log
PseudocolorAtts.minFlag = 1
PseudocolorAtts.min     = 1e-11
SetPlotOptions(PseudocolorAtts)
setMyView()
DrawPlots()



SetActiveWindow(2) # Window 2, Curve plot of K, K Ave, K Ave + StdDev, K Ave - StdDev
SetActivePlots(tuple(range(GetPlotList().GetNumPlots())))
DeleteActivePlots()
AddPlot("Curve", "k Eigenvalue", 1, 0)
AddPlot("Curve", "k Eigenvalue Ave", 1, 0)
AddPlot("Curve", "K - Std Dev", 1, 0)
AddPlot("Curve", "K + Std Dev", 1, 0)
DrawPlots()

SetActiveWindow(3) # Window 3, Pseudocolor plot of log(particle KE)
SetActivePlots(tuple(range(GetPlotList().GetNumPlots())))
```

Figure 63: VisIt's Commands GUI.

```
################################################################################
# File Name: cow.py
# Purpose: This is an example python script that sets up 4 VisIt Windows.
# 1. Pseudocolor plot of: log(particle number density)
# 2. 4 curve plots: K, K Ave, K Ave + StdDev, K Ave - StdDev.
# 3. Pseudocolor plot of log(particle KE)
# 4. 1 curve plot: Flux Entrop
# Usage:  Mercury>  visit('cow.py')
################################################################################
def setMyView():
  # Set view
  View3DAtts = View3DAttributes()
  View3DAtts.viewNormal    = (-0.973501, 0.0961496, -0.207485)
  View3DAtts.focus = (72.5, 72.5, 72.5); View3DAtts.viewUp = (0.12,0.99, -0.10)
  View3DAtts.viewAngle  = 30;        View3DAtts.parallelScale = 134.234
  View3DAtts.nearPlane  = -268.468;  View3DAtts.farPlane      = 268.468
  SetView3D(View3DAtts)
SetWindowLayout(4)

SetActiveWindow(1) # Window 1, Pseudocolor plot of log(particle number density)
SetActivePlots(tuple(range(GetPlotList().GetNumPlots())))
DeleteActivePlots()
AddPlot("Pseudocolor", "MC_Number_Density_Neutron", 1, 0)
TurnMaterialsOff()
TurnMaterialsOn(("Uranium_235"))
PseudocolorAtts = PseudocolorAttributes()
PseudocolorAtts.scaling = PseudocolorAtts.Log
PseudocolorAtts.minFlag = 1
PseudocolorAtts.min     = 1e-11
SetPlotOptions(PseudocolorAtts)
setMyView()
DrawPlots()

SetActiveWindow(2) # Window 2, Curve plot of K, K Ave, K Ave + StdDev, K Ave - StdDev
SetActivePlots(tuple(range(GetPlotList().GetNumPlots())))
DeleteActivePlots()
AddPlot("Curve", "k Eigenvalue", 1, 0)
AddPlot("Curve", "k Eigenvalue Ave", 1, 0)
AddPlot("Curve", "K - Std Dev", 1, 0)
AddPlot("Curve", "K + Std Dev", 1, 0)
DrawPlots()

SetActiveWindow(3) # Window 3, Pseudocolor plot of log(particle KE)
SetActivePlots(tuple(range(GetPlotList().GetNumPlots())))
DeleteActivePlots()
AddPlot("Pseudocolor", "Particles/kinetic_energy", 1, 0)
PseudocolorAtts = PseudocolorAttributes()
PseudocolorAtts.scaling = PseudocolorAtts.Log
PseudocolorAtts.minFlag = 1
PseudocolorAtts.min     = 0.1
SetPlotOptions(PseudocolorAtts)
setMyView()
DrawPlots()

SetActiveWindow(4) # Window 4, Curve plot of Flux Entropy
SetActivePlots(tuple(range(GetPlotList().GetNumPlots())))
DeleteActivePlots()
AddPlot("Curve", "Flux Entropy", 1, 0)
DrawPlots()
```

Figure 64: Example VisIt python script.

6.4 Combinatorial Geometry Visualization Options

Before VisIt had the ability to natively discretize combinatorial geometry, Mercury had to do the discretization itself, and write out a *mesh*, that VisIt could visualize. The problem at hand is given the analytic surfaces that define the CG, how does one visualize them in a mesh visualization tool like VisIt? We translate the CG representation to a mesh based representation by sampling the CG onto a graphics mesh which is then visualized in VisIt.

6.4.1 Clean Cell Method

The very first implementation looked like "Legos". The user defined a graphics mesh, and each graphics cell was assigned exactly *one* CG cell whose identity was chosen by asking "which CG cell is the center of each graphic cell in?" Figure 65 shows an example of the output of the clean cell algorithm.

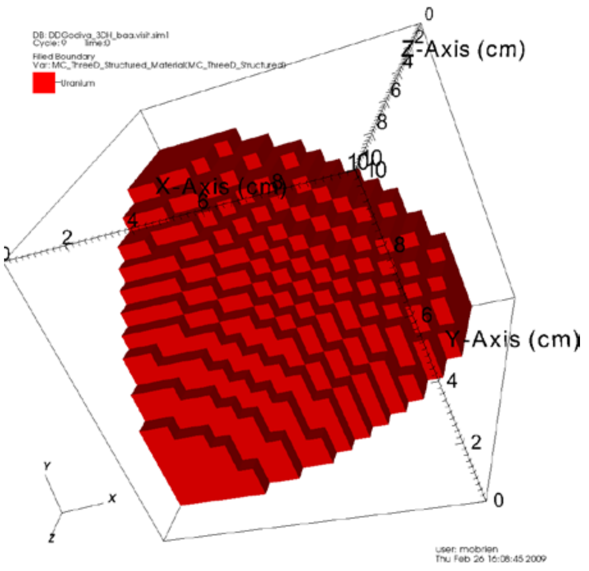


Figure 65: Octant of a sphere visualized with the Clean Cell Method.

6.4.2 Mixed Cell Method

In the mixed cell method, Mercury recursively samples points within each zone to get an accurate volume fraction of each CG cell within each graphics mesh cell. Mercury only writes out *volume fraction* information and then relies on VisIt's *Material Interface Reconstruction* algorithm to subdivide the graphics cells into parts corresponding to the volume fraction of the parts of the cell. Figure 66 shows an example of the output of the mixed cell algorithm.

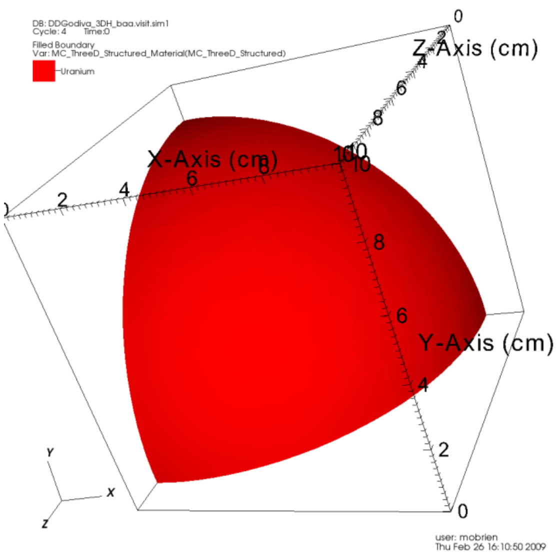


Figure 66: Octant of a sphere visualized with the "Mixed Cell" method, relying on Visit's MIR algorithm.

Figure 67 illustrates how the **Mixed Cell Method** works. If the four corners (eight corners in 3D) of a graphics cell all contain the same CG cell, the algorithm terminates. If any of the corners contain different CG cells, the algorithm recursively subdivides the cell into four parts (eight parts in 3D) and continues sampling. User settable parameters exist for controlling the minimum and maximum recursion limits. The goal of the algorithm is to compute the *volume fraction* of each CG cell within each graphics cell. VisIt uses the volume fraction information to do *Material Interface Reconstruction* to attempt to reconstruct the position of the CG cell interface that divides the graphics cell. Assume Figure 67 represents only *one* graphic mesh zone. The mesh lines are for illustration only; they are used for the algorithm to compute the volume fraction. The output of the algorithm will converge to the volume fraction of the quarter circle in the square cell, volume fraction = (area of quarter circle)/(area of square) = $\pi/4$.

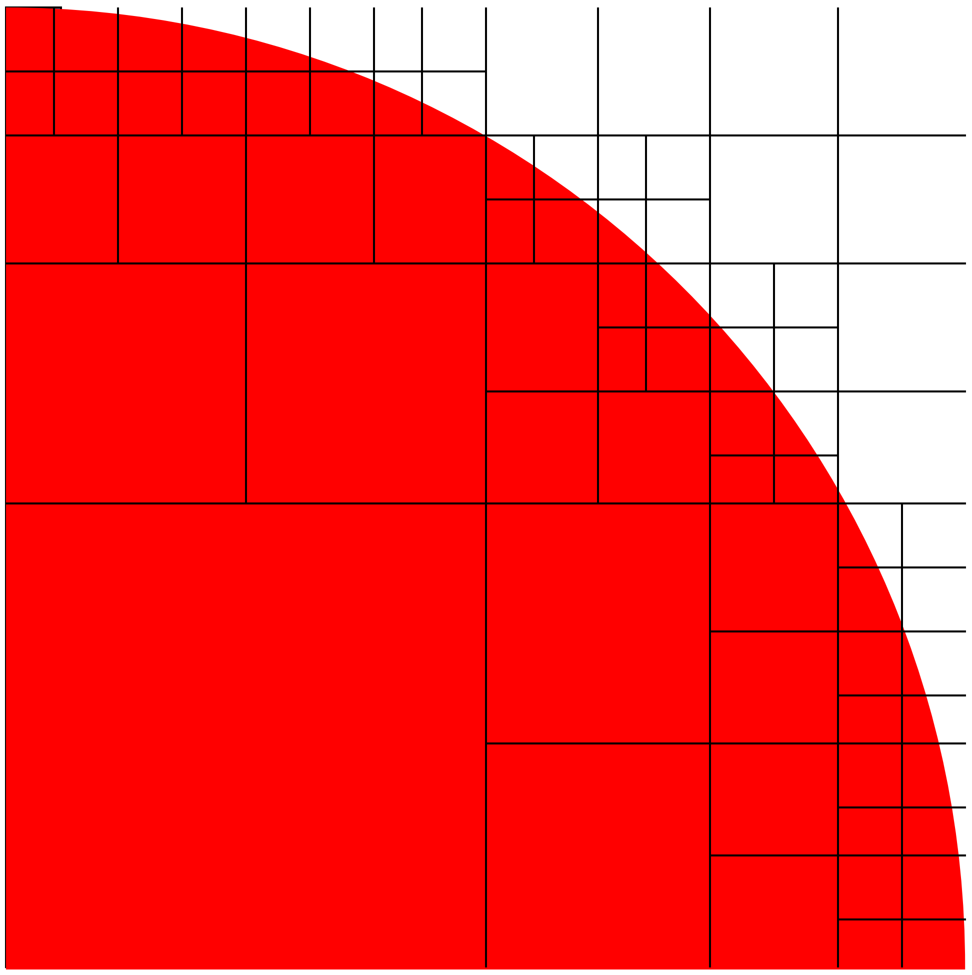


Figure 67: Example of recursive volume fraction calculation.

6.4.3 Numerical Integration Method

Instead of sampling *points* within each mesh zone to calculate the volume fraction of a CG cell within the mesh zone, the numerical integration method uses Mercury's particle tracker to find the position of the CG surface that cuts through the mesh zone. This algorithm uses a 2D numerical integration to calculate the volume fraction of a CG cell within the graphics mesh zone. Then VisIt uses the volume fractions for its own Material Interface Reconstruction algorithm. Given the height of the surface above the base of the graphics mesh cell, the algorithm uses a quadrature rule to find the volume below the surface. See Figure 68 for an illustration of how this algorithm works.

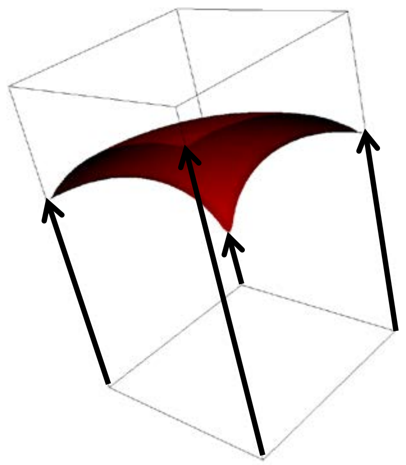


Figure 68: Illustration of numerical integration volume calculation.

6.4.4 Conformal Method

The conformal algorithm starts with a Cartesian graphics mesh and then "launches a particle" from every node in the graphics mesh. If the particle intersects a CG surface before it reaches the neighboring node in the graphics mesh, that mesh node is moved to lie on the CG surface. The idea is to conformally move the graphics nodes directly onto the CG surfaces so the graphics mesh contains cells of exactly one material. Since the mesh is only used for visualization and not for a finite difference or finite element calculation, it does not matter how distorted the graphics cells are. We have developed an implementation of this algorithm that works well if the graphics mesh cell is cut by only one CG surface. If more than one CG surface intersects the graphics mesh cell, then we revert to the "mixed" method and compute volume fractions of CG cells within the graphics mesh cell.

Figure 69 compares the mixed cell method with the conformal method. In the mixed cell method, we can see that the mesh is Cartesian. Mercury has computed volume fractions of each CG cell within each graphics cell, and then VisIt is using its Material Interface Reconstruction to draw the CG cell boundaries. In the conformal method, we can see that we started with a Cartesian mesh, and then moved the nodes onto CG cell boundaries.

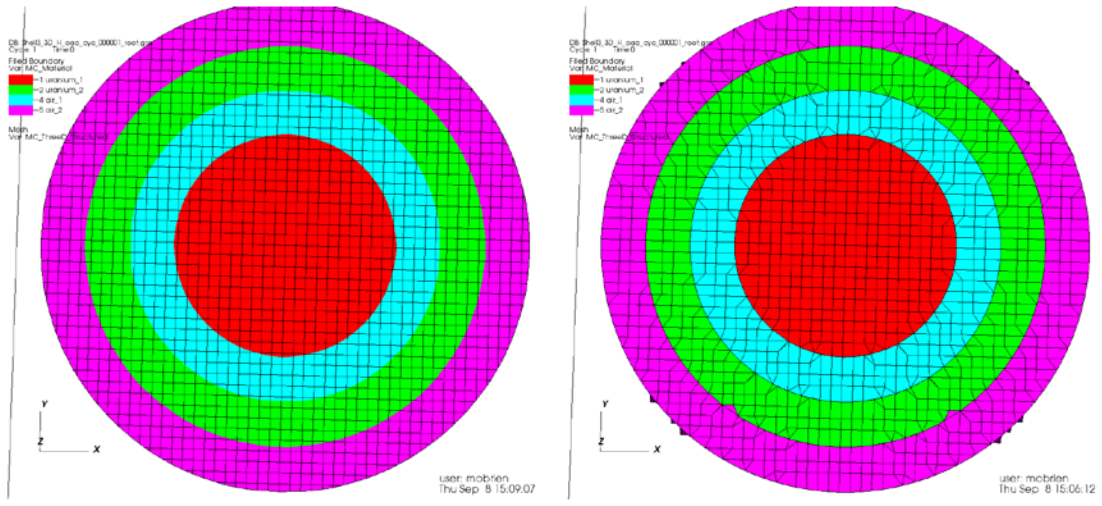
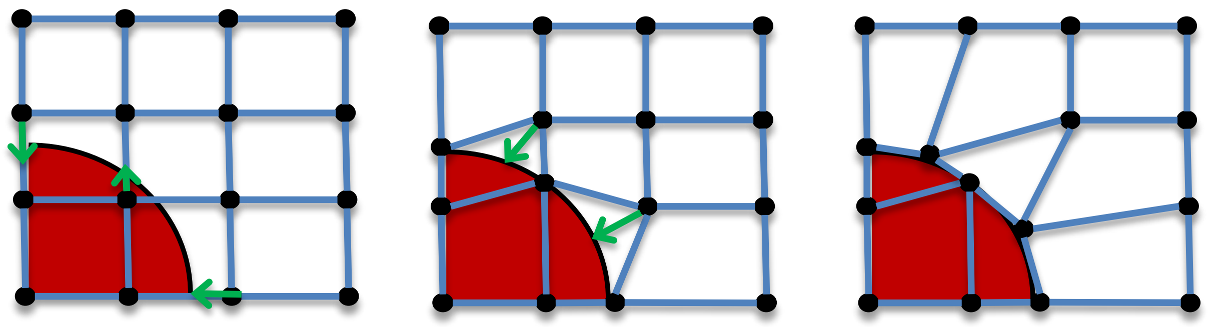


Figure 69: (a) Mixed Cell Method (b) Conformal Method.

Figure 70 shows a description of how the conformal algorithm works.



(A). Move nodes of cut edges to nearest surface.	(B). Now no edges are cut, but zones may be cut diagonally.	(C). Move nodes diagonally to form clean zones.

Figure 70: Description of conformal mesh algorithm.

Figure 71 shows an example of starting with a Cartesian Mesh (shown in a dashed line) and then moving the mesh nodes to the CG surface so that each mesh zones is clean. Figure 72 shows the 2D algorithm comparing the conformal method with the mixed cell method; the top half of the picture is the conformal method, and the bottom half is the mixed cell method. The conformal method has all clean zones, and the zone boundaries lie along the CG surface boundaries. The mixed method calculates volume fractions of each mixed cell, then VisIt's MIR algorithm draws the material interfaces. Figure 73 shows the 3D algorithm comparing the conformal method with the mixed cell method. The mixed cell method is shown on the left half of the hemisphere, and the conformal method is shown on the right half of the hemisphere. The conformal method has all clean zones, and the zone boundaries lie along the CG surface boundaries. The mixed cell method calculates volume fractions, and then VisIt's MIR algorithm draws the material boundaries.

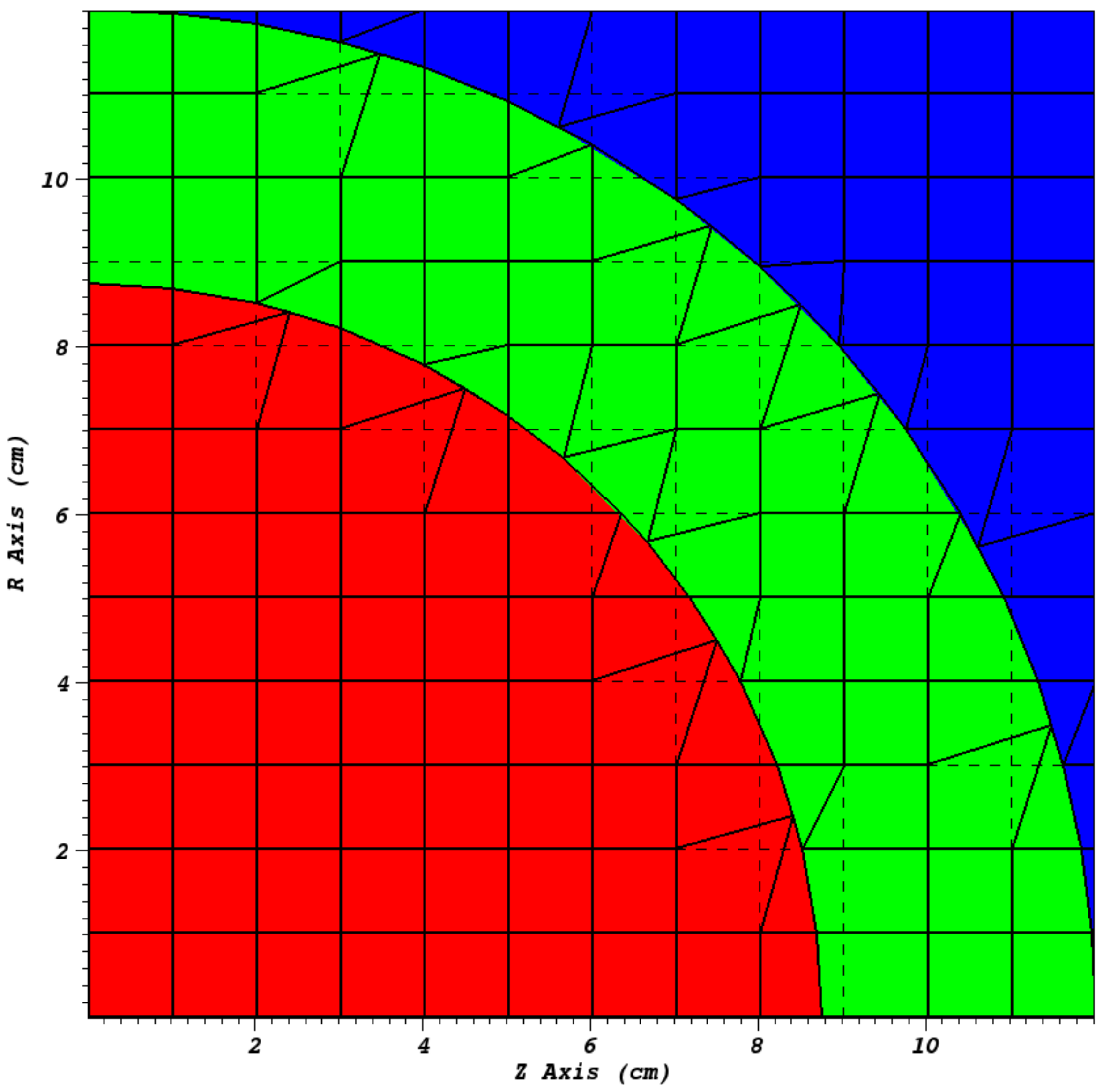


Figure 71: Example of the Conformal method on a 2D mesh.

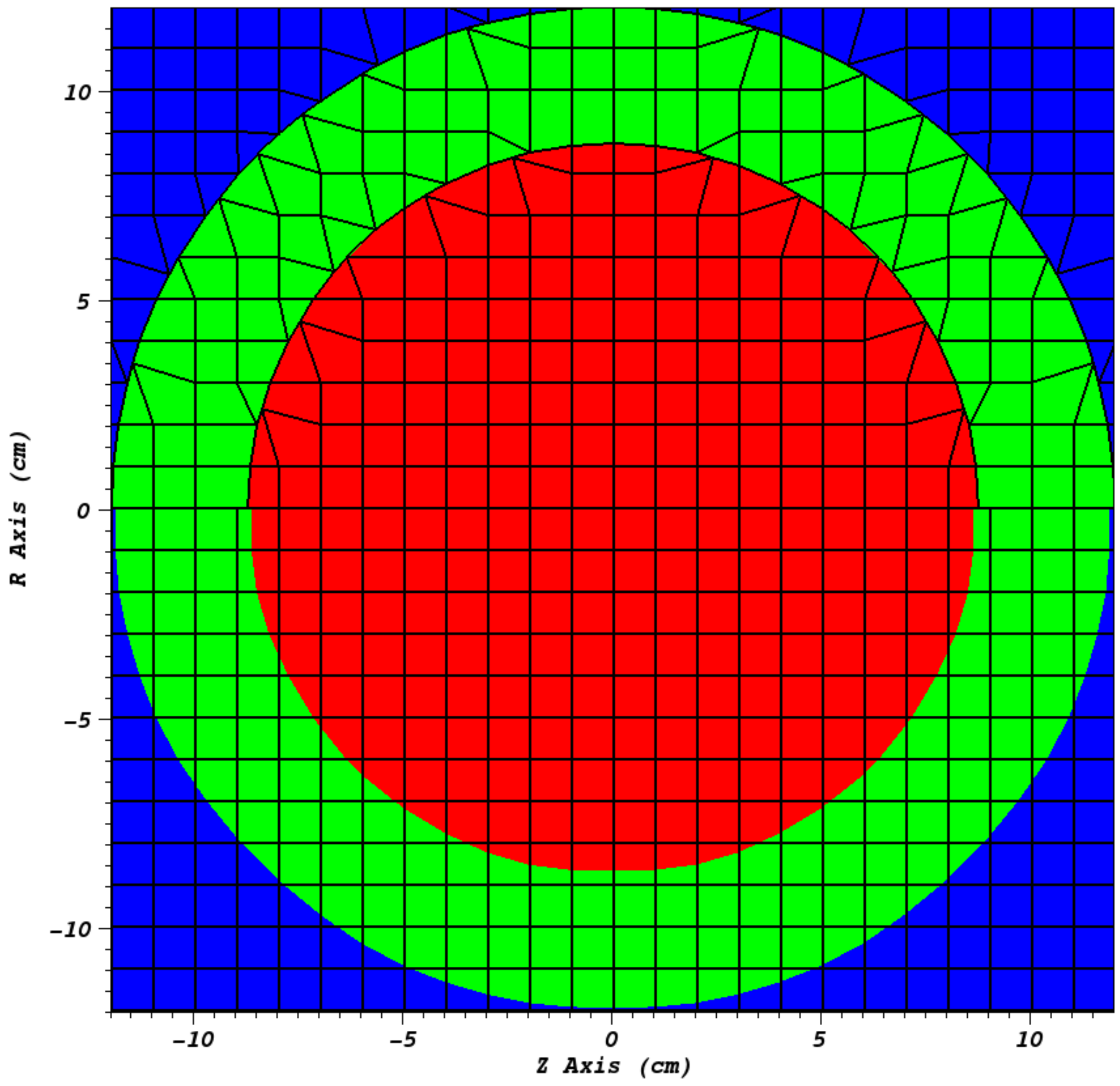


Figure 72: 2D Conformal method (on top) vs. mixed method (on the bottom).

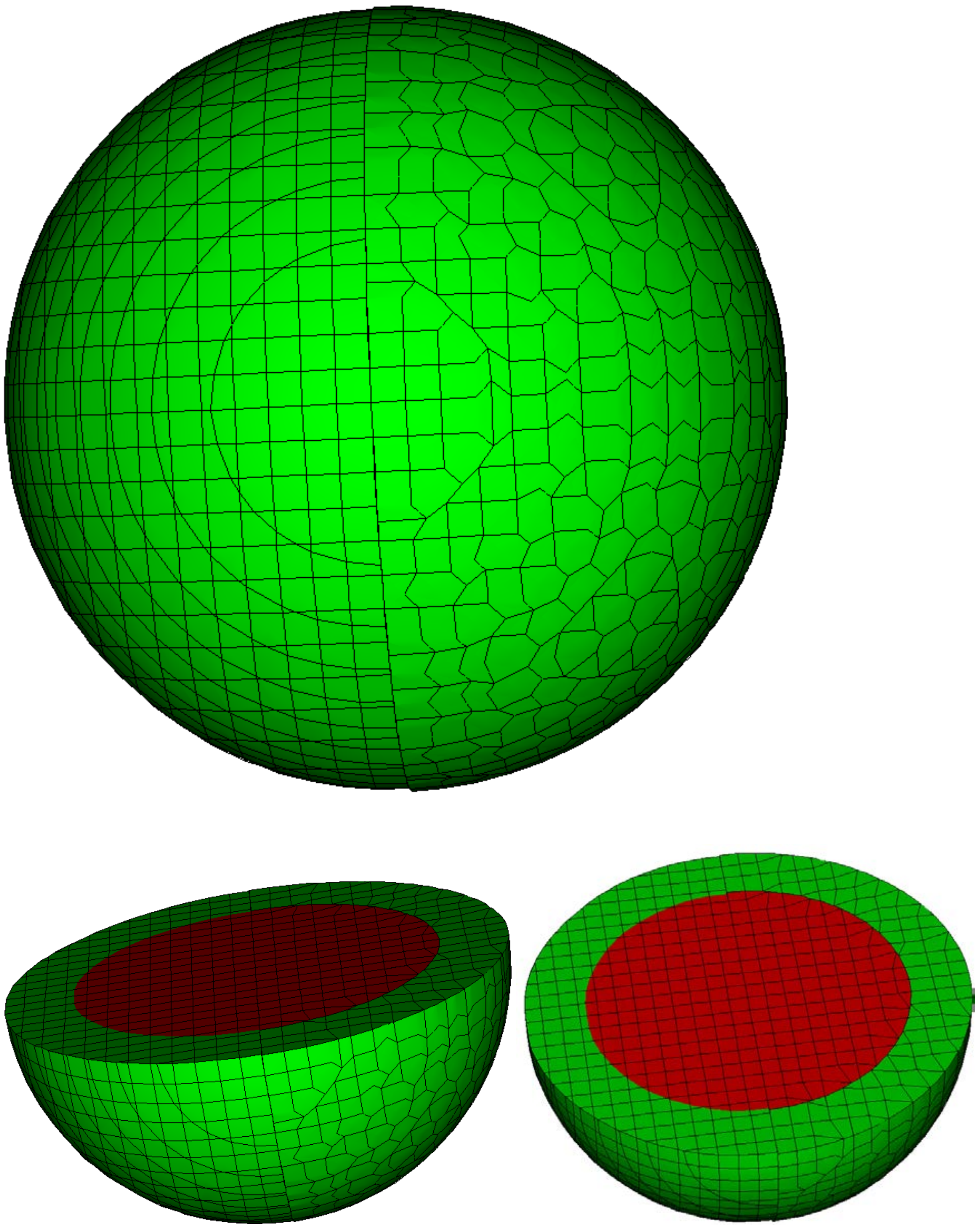


Figure 73: 3D Conformal (right half) vs. Mixed (left half).

Now we examine the convergence of the volume of a sphere calculated by the Conformal Mesh algorithm. Assume we have a sphere that has a volume of exactly 1.0, and then we examine the numerical error as a function of the number of zones. We start with a uniform mesh around the sphere, and then run the Conformal Mesh algorithm to move nodes to the surface of the sphere. Figure 74 is a graph of the volume error vs. the number of zones. This was performed in both 2D cylindrical geometry and 3D Cartesian geometry. 2D cylindrical geometry has infinite azimuthal resolution, so it has less error than 3D for the same number of zones

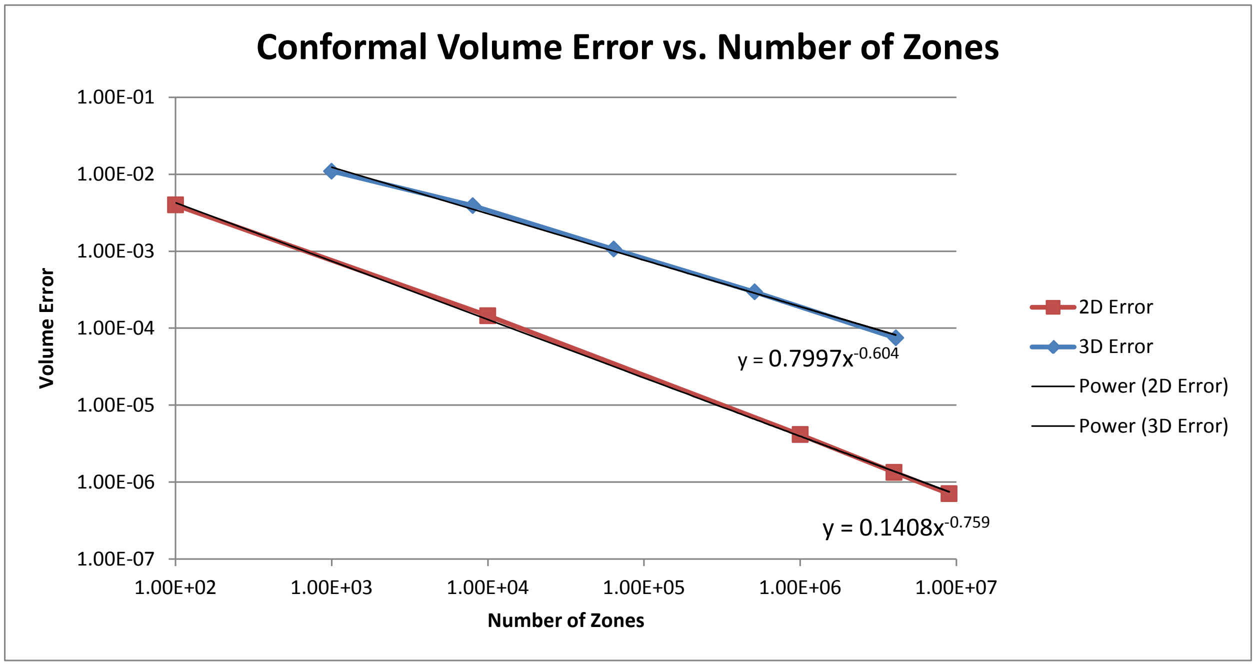


Figure 74: Conformal Volume Error vs. Number of Zones; for 2D and 3D test problem.

. Let N = the total number of zones, then the total error = (Error for one boundary zone)*(number of boundary zones). See Table 10 for the theoretical analysis of how the error should behave as a function of the number of zones. The error for one boundary zone can be calculated by considering the volume of a "spherical cap", that is the error associated with approximating a spherical surface with a plane within one zone. Similarly the area of a 2D circular segment can be calculated and then revolved to calculate the volume error for a single boundary zone in 2D. The non-boundary zones are entirely within the sphere and have zero volume error.

Table 10: Theoretical analysis for 2D and 3D conformal mesh volume error.

	Error for one Boundary Zone	Number of Boundary Zones	Total Error
2D	$N^{-3/2}$	$N^{1/2}$	N^{-1}
3D	$N^{-4/3}$	$N^{2/3}$	$N^{-2/3}$

6.4.5 Direct Visualization Through Ray Casting

In this case we do not use VisIt for visualization at all. Instead, Mercury draws directly to an image buffer that we display with the Python package *pylab* [61], [62]. We translate user mouse clicks into rotations, translations and magnifications by moving the particle source (which is the camera, in this case) and redoing a Mercury calculation to redraw the new frame. Mercury knows how to track particles to CG surfaces and how to calculate the surface normal, so we color each CG cell by some field and then reduce the brightness of the color by the dot product of the incoming particle and the local surface normal:

ShadedColor = CellColor * |Dot(IncomingVisualizingParticle, FacetNormal)|

Figure 75 shows an example of the output of the Mercury ray caster used to visualize various shapes.

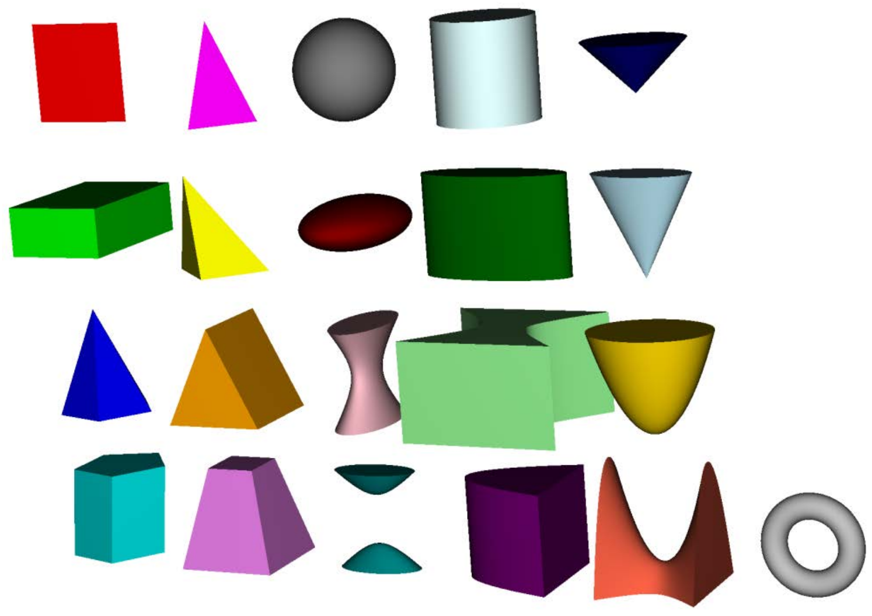


Figure 75: Direct visualization through ray casting.

No linear approximations are made with this method. There is also no length scale requirement for this method. If a user wants more detail, they can just continue to zoom in on a feature, and they will see features at any length scale. This feature overcomes a disadvantage of the "mesh based" methods, which require a length scale for the mesh. If a user continues to zoom in with mesh based methods, the resolution is limited by the finest mesh resolution.

By adding separate left eye and right eye sources and image planes, we can also create a 3D image with separate left eye and right eye images. With special 3D hardware, we can visualize the left eye and right eye images independently. Without the help of hardware, we can composite the images to create a 3D anaglyph that can be viewed with red-cyan glasses. The composite image is formed by taking the red component of the left image and the green and blue components of the right image:

$$(\text{Red}_{\text{composite}}, \text{Green}_{\text{composite}}, \text{Blue}_{\text{composite}}) = (\text{Red}_{\text{left}}, \text{Green}_{\text{right}}, \text{Blue}_{\text{right}})$$

Figure 76 shows an example of a 3D anaglyph.

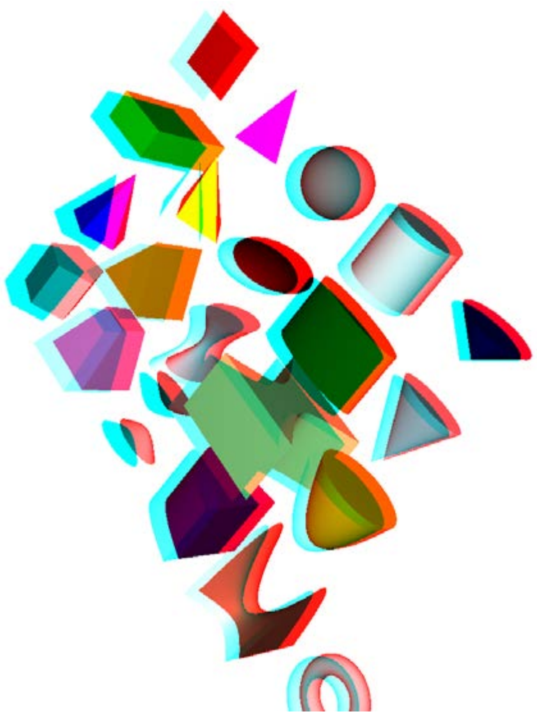


Figure 76: 3D anaglyph created with left-eye and right-eye ray casting.

6.4.6 Native VisIt Discretization and Visualization

In the native VisIt discretization and visualization method, Mercury does not convert its internal CG representation to a mesh representation, rather it gives the CG representation directly to VisIt. Mercury gives VisIt the coefficients of the analytic surfaces that define the cells, and how the surfaces are combined to form cells. Then VisIt does an AMR discretization of the CG and creates an AMR mesh to visualize the CG. Figure 77 shows an example of the output of this method.

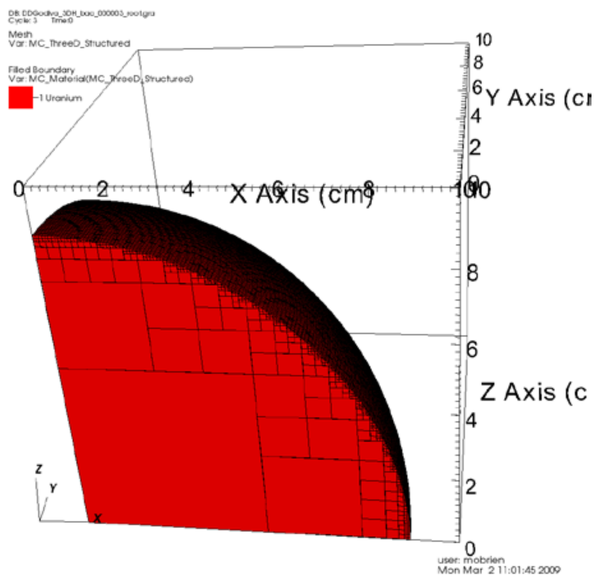


Figure 77: Native VisIt discretization. The AMR mesh gets finer near the cell boundary.

6.5 Mixed Cell Method: Algorithm Error Analysis

In the Mixed Cell Method, Mercury recursively samples each graphics mesh cell, attempting to calculate an accurate volume fraction for each CG cell in the graphics mesh cell. This functionality can be used to calculate the volume of each CG cell in the problem (for an alternate volume calculation, see [63] and [64]). In this error analysis, we have a sphere of radius 8.7407cm, centered at (0,0,0), contained in a *single* graphics mesh cell [0, 10] x [0, 10] x [0, 10]. So an octant of the sphere will be sampled. We vary the min and max refinement level from 1 to 11. We enforce min refinement = max refinement. At refinement level n, there are 8^n sample points, i.e. in 3D we cut the cell in half in each of the 3 coordinate directions, $2^3 = 8$. This is an octant of a sphere of radius 8.7407cm, so the exact volume is :

Volume(Sphere(8.7407)) = 4/3 * Pi * 8.7407^3 / 8 = 349.6530057623cm^3.

As we increase the refinement level by 1, we expect the error to go down roughly by 1/8th since each graphics cell is subdivided into 8 sub-cells. Table 11 shows the volume error at each refinement level n=1,2,3,…,11.

Define the error at refinement level n, E[n] = | exact volume – calculated volume |

Table 11: Error analysis of recursive oct-tree algorithm for calculating volume fractions.

Refinement Level	Volume Error, E[n]	Error Ratio E[n-1]/E[n]	Calculated Volume
1	4.36057623183E-02		349.6094000000
2	1.64010057623E+01	2.65872489469E-03	333.2520000000
3	4.78942376817E-02	3.42442150793E+02	349.7009000000
4	9.59594237682E-01	4.99109267240E-02	350.6126000000
5	5.94057623183E-02	1.61532181430E+01	349.5936000000
6	2.20942376817E-02	2.68874460274E+00	349.6751000000
7	1.63942376817E-02	1.34768313786E+00	349.6694000000
8	3.69423768171E-03	4.43778638360E+00	349.6567000000
9	9.87387681732E-04	3.74142573384E+00	349.6539931500
10	1.36472318275E-04	7.23507663837E+00	349.6528692900
11	2.10223182648E-05	6.49178252157E+00	349.6529847400
Exact Vol	0.00000000000E+00	N/A	349.6530057623

Figure 78 is a plot of the Error At Refinement Level n vs. the Refinement Level. After some initial noise at small refinement levels, the error decreases linearly on a log scale, so the error is decaying exponentially.

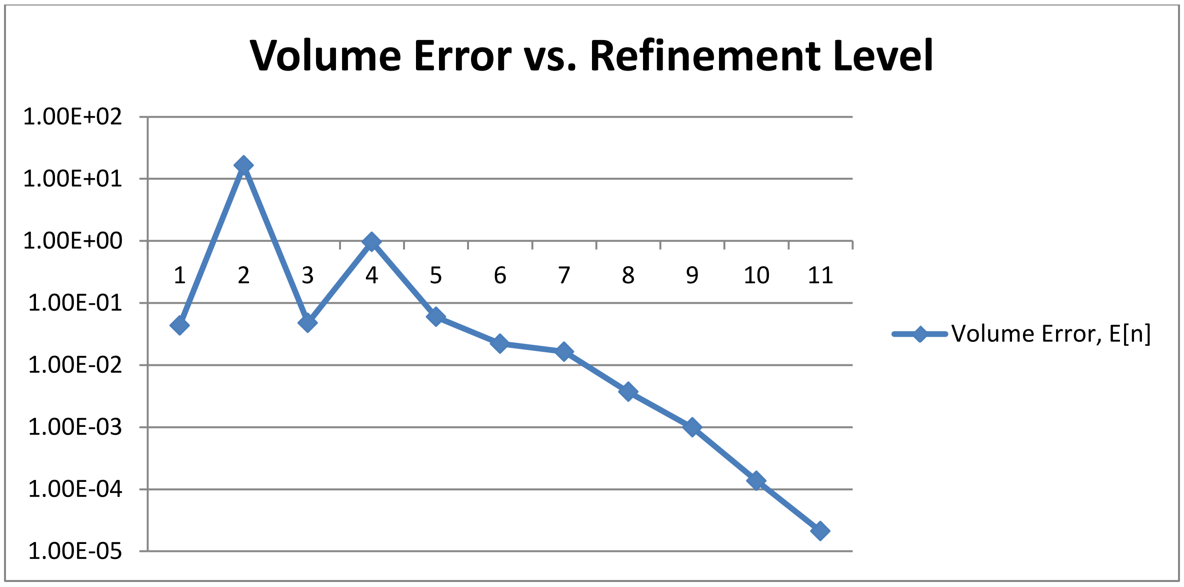


Figure 78: Error vs. refinement level.

Figure 79 is a plot of the Error At Refinement Level n vs. the Refinement Level, for n = 7, 8, 9, 10, 11. The expoential fit for the error is $E[n] = 0.1028 * (1/5.27)^n$, which means that the error goes down by a factor of about (1/5.27) when the refinement level is increased by one. This error reduction is roughly what we expected, since each graphics cell is divided into 8 sub-cells as the refinement level increases. The error should go down by at most a factor of (1/8) when the refinement level is increased by one.

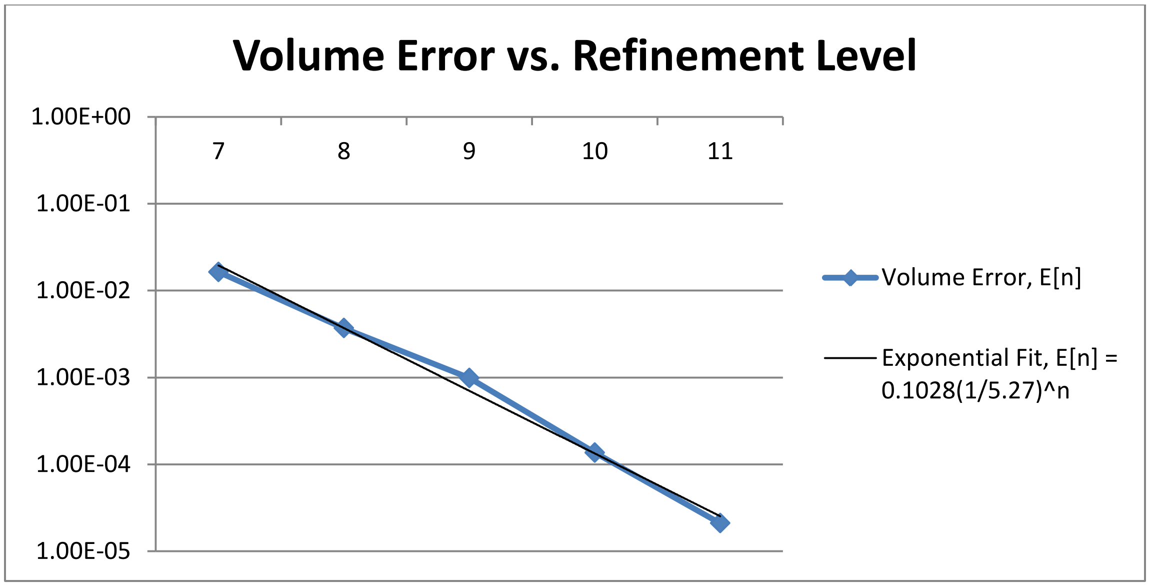


Figure 79: Volume error vs. refinement level.

6.6 Geometry Error Detection

Ensuring that the CG input is setup correctly can be very difficult. Mercury has “gap” and “overlap” detection built into its graphics mesh options. Mercury will automatically find “gaps” or “void” in a problem, which is defined to be any point in space where *no* CG cell claims ownership of that point. The void and overlap detection are based on the graphics mesh resolution that the user requests, and Mercury samples the CG at the graphics mesh nodes. If the voids or overlaps are smaller than the mesh resolution, this method will not find those errors.

Anytime a user creates a graphics mesh, “void” detection is automatically enabled by calling the Mercury routine “Which CG cell is this point in” at each graphics mesh point. If no CG cell claims ownership of the point, then “void” has been found.

The overlap detection must be requested by the user since it is more expensive to check. For each graphics mesh point, the algorithm loops over all CG cells and asks if the CG cell thinks it owns that point in space. If more than one CG cell claims ownership of the point, then "overlapping cells" have been found.

Figure 80 is a simple example of two red Uranium spheres that are overlapping. We have a blue "void" sphere that has been excluded from the green Air sphere. The user must decide if the "void" is correct or not. For example, in this problem, we have a vacuum boundary condition around the green Air sphere, so it is correct to have "void" outside of the green sphere. But we do not have a boundary condition for the small internal blue "void" sphere, so this is a geometry setup error.

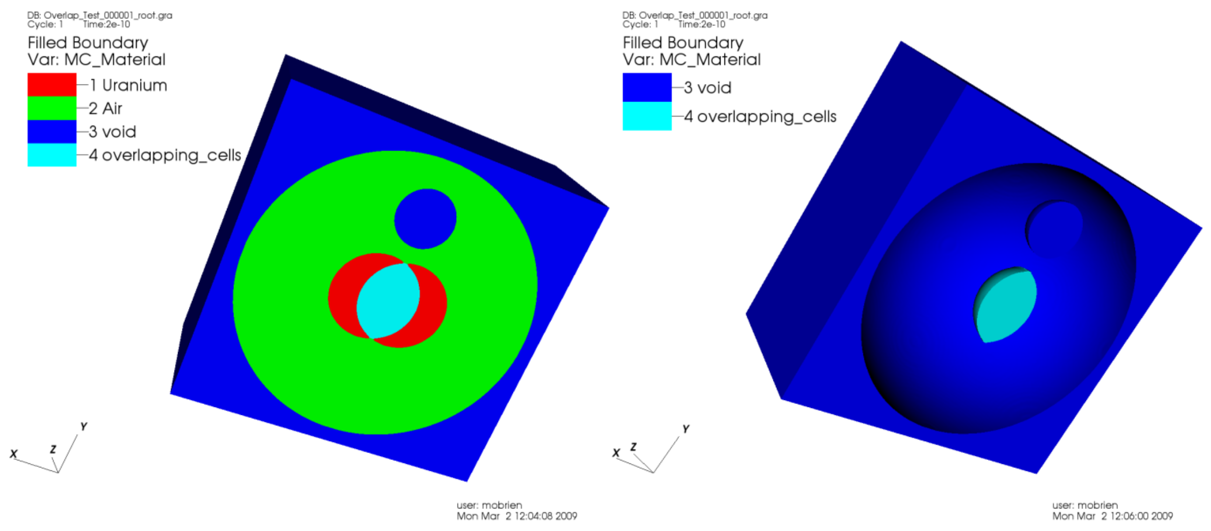


Figure 80: (a) All materials are shown. (b) only "void" and "overlapping_cells" are shown.

The code gives a printout of the mass and volume of all materials in the problem, including any “void” or “overlapping_cells” found. Here is an example of the output for this test problem:

```
Material Uranium            Mass 2.0408300111e+03 Volume 5.4451174255e+01
Material Air                Mass 2.3926665283e+00 Volume 1.9938887736e+03
Material void               Mass 0.0000000000e+00 Volume 3.2463340822e+03
Material overlapping_cells  Mass 0.0000000000e+00 Volume 2.9325969955e+01
--------------------------------------------------------------------------
Total                       Mass 2.0432226776e+03 Volume 5.3240000000e+03
```

6.7 Gallery

In this section we show how visualization with VisIt has been used to aid in understanding and verification of code development issues.

6.7.1 Dynamic Load Balancing

In Figure 81 we are looking at the domain decomposition to verify that it intuitively makes sense. This graphics is an octant of a supercritical Uranium sphere, so we would expect more work to be at the center of the sphere than farther out radially. We see that Mercury has chosen a domain decomposition that balances the work in each domain [65]. This spatial redecomposition algorithm is discussed in detail in Section 7.4.

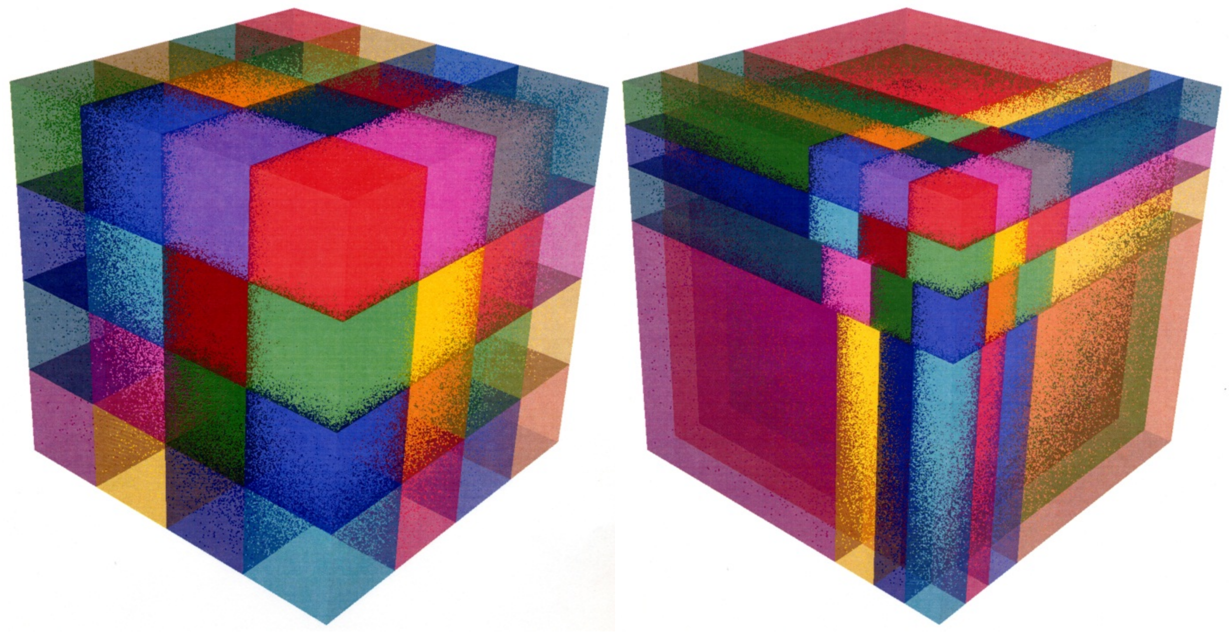


Figure 81: (a) Uniform domain decomposition. (b) Load balanced domain decomposition.

In Mercury, particles are tracked in 3D. Figure 82 shows particles colored by the domain that they are in. We are applying a "cylindrical projection"; a particle has Cartesian coordinates (x,y,z) which we plot at cylindrical coordinates (z, r) = (z, sqrt(x^2 + y^2)). The domain decomposition is varying dynamically and responding to balance the particle workload.

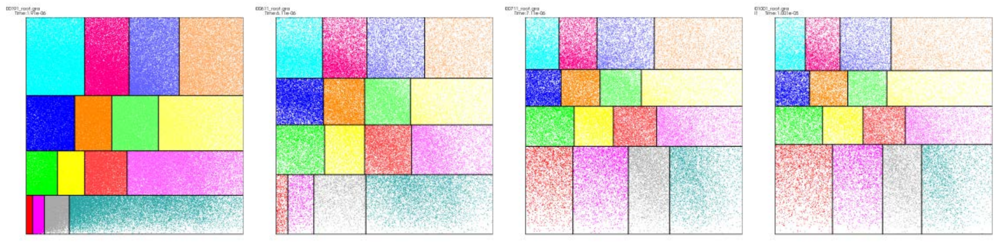


Figure 82: Time evolution of domain decomposition.

6.7.2 Material Interface Reconstruction

The underlying mesh in this problem is a 2D cylindrical mesh, so we are again applying the "cylindrical projection" to the particle coordinates. Figure 83 was used to verify Mercury's Material Interface Reconstruction algorithm [66].

First note the bold black line that identifies the material interface. That line was calculated to be normal to the gradient of the material volume fraction. The position of the line was found to match the input volume fractions of the materials in the underlying mesh.

Next note that each particle is colored by the material that it is in. Many iterations of code development followed by visualization were required to get the plot to look correct. It was very easy to spot particles colored the wrong color given their location. This type of visualization is extremely valuable for validating code development.

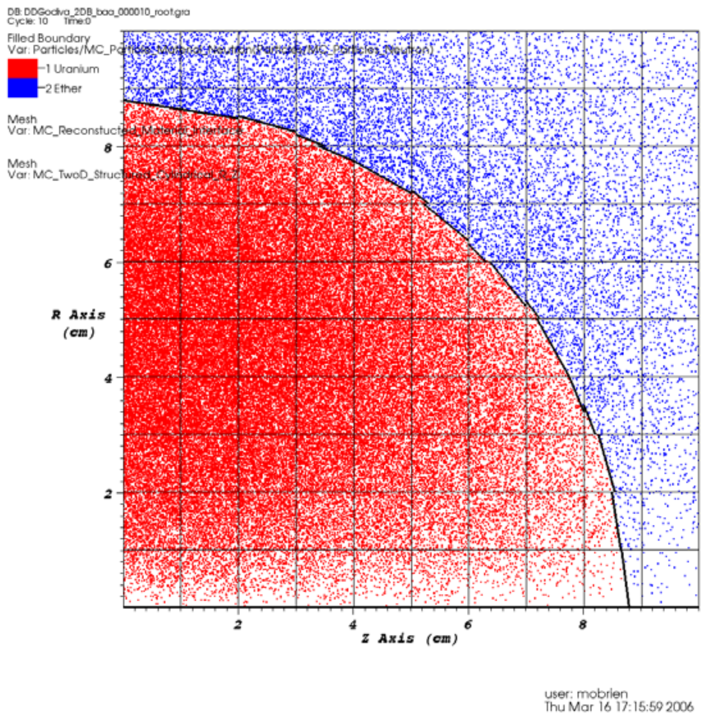


Figure 83: Material Interface Reconstruction

6.7.3 Embedded Mesh in Combinatorial Geometry

Figure 84 has two images:

(a) Shows the combinatorial geometry with 2 embedded meshes within it. The particles are colored by "region", meaning which mesh or CG region they are in. In this case particles are blue if they are in the CG (none present), green if they are in the cylindrical region (2D mesh), and red if they are in the cube region (3D mesh).

(b) This is a plot colored by the CG cell index for the CG, and it also shows the 2D cylindrical embedded mesh, and a 3D Cartesian embedded mesh.

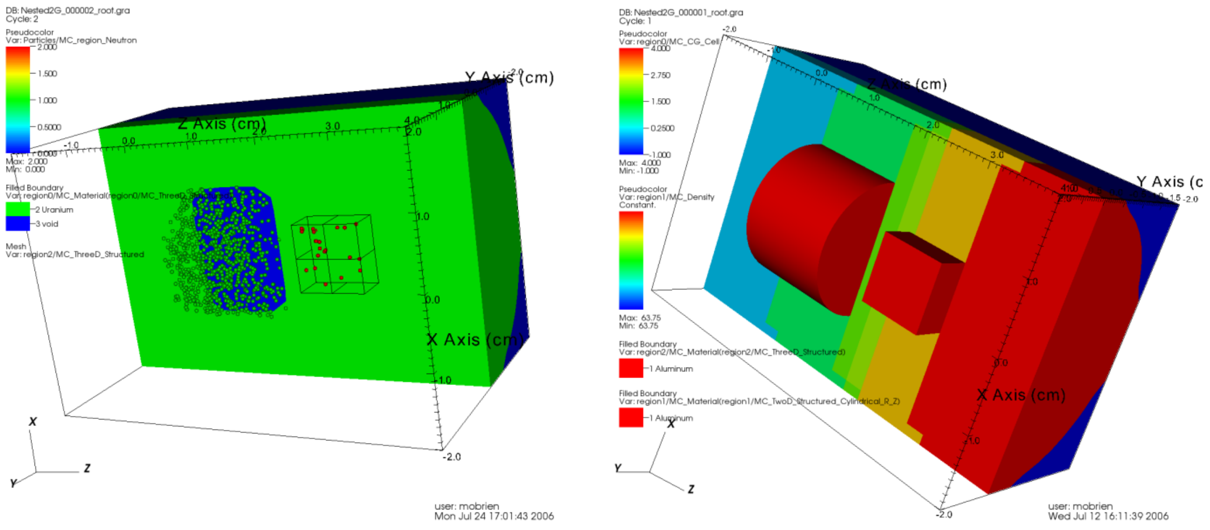
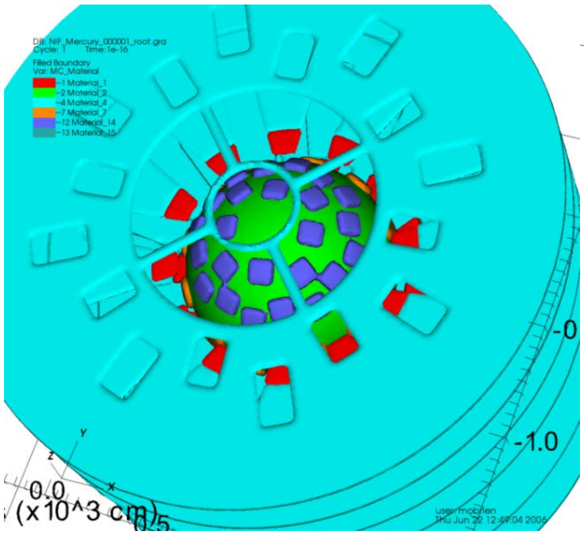
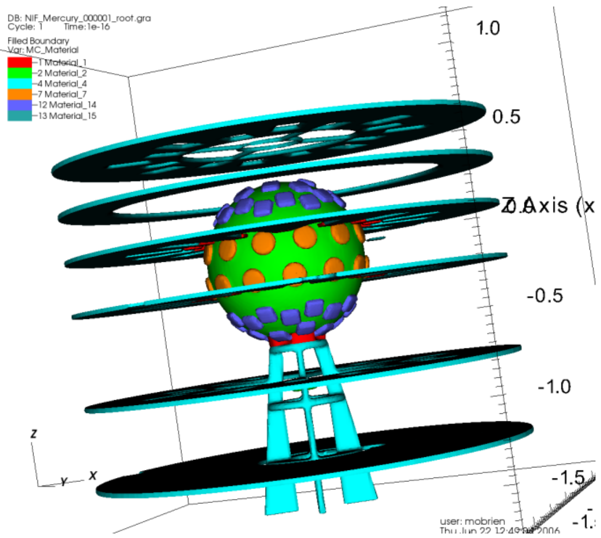


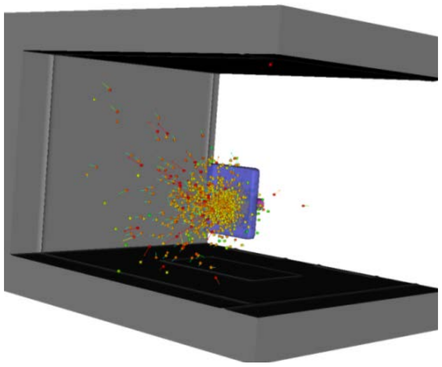
Figure 84: (a) Particles colored by region. (b) CG and 2 embedded meshes.

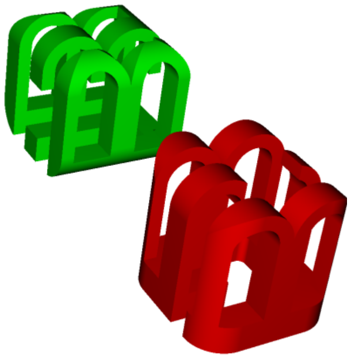
6.7.4 Complicated Geometry Setup Verification.

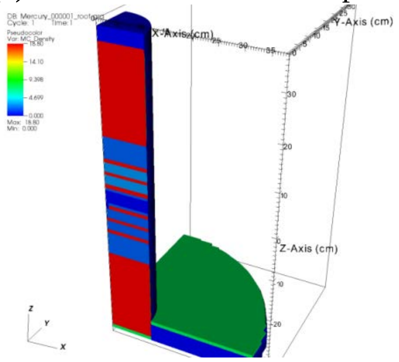
Figure 85 shows test problems where visualization was used to verify the problem setup.


(a) NIF target chamber and supports.


(b) NIF target chamber and supports.


(c) Fusion shield room test problem.


(d) Godel, Escher, Bach test problem.


(e) Critical assembly test problem.

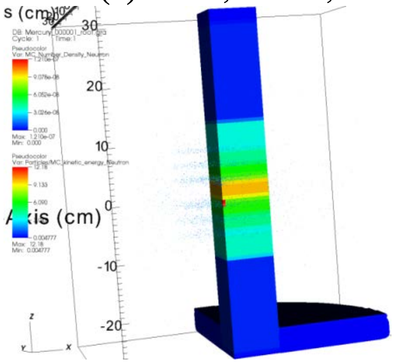
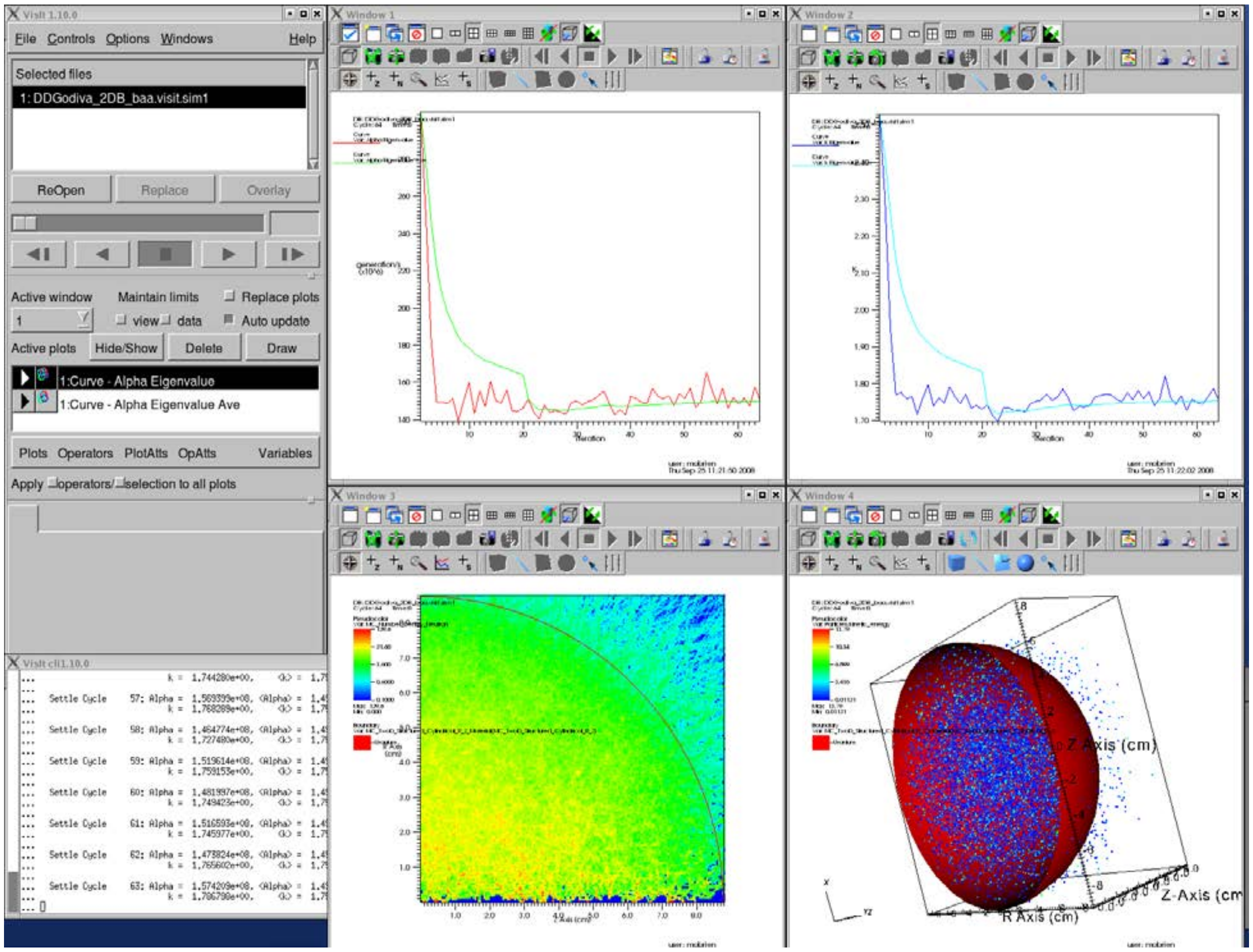

(f) Critical assembly test problem.

Figure 85: Various test problems; visualization is used to verify correct problem setup.

6.7.5 Running VisIt Inline in Mercury

In Figure 86 the VisIt graphical user interface (GUI) is shown on the left side of the image, while four sets of tally data are displayed in individual windows on the right side of the image. The screen snapshot shown in Figure 86 was obtained while the tally data was being updated in real-time during a Mercury simulation of a criticality problem. This calculation is a 2-D axisymmetric r-z (mesh-based) model of the gedanken double-density Godiva assembly. This example demonstrates the wide variety of data visualization capabilities available within VisIt. The two upper windows are curve plots representing the iteration histories of the instantaneous and averaged α (left) and Keff (right) eigenvalues. A pseudocolor representation of the neutron number density (including the effects of individual particle tracks), superimposed on the outer boundary of the spherical assembly, is shown in the lower left window. Finally, the lower right window shows the 3-D point mesh collection of the Monte Carlo particles superimposed upon one hemisphere of the assembly, where the color of each particle represents its kinetic energy.



```
...                              k =  1.744280e+00,      <k> =  1.7
...
...   Settle Cycle     57: Alpha =  1.563339e+08, <Alpha> =  1.4
...                              k =  1.768289e+00,      <k> =  1.7
...
...   Settle Cycle     58: Alpha =  1.464774e+08, <Alpha> =  1.4
...                              k =  1.727480e+00,      <k> =  1.7
...
...   Settle Cycle     59: Alpha =  1.519614e+08, <Alpha> =  1.4
...                              k =  1.759153e+00,      <k> =  1.7
...
...   Settle Cycle     60: Alpha =  1.481997e+08, <Alpha> =  1.4
...                              k =  1.749423e+00,      <k> =  1.7
...
...   Settle Cycle     61: Alpha =  1.516593e+08, <Alpha> =  1.4
...                              k =  1.745977e+00,      <k> =  1.7
...
...   Settle Cycle     62: Alpha =  1.473824e+08, <Alpha> =  1.4
...                              k =  1.765602e+00,      <k> =  1.7
...
...   Settle Cycle     63: Alpha =  1.574209e+08, <Alpha> =  1.4
...                              k =  1.786798e+00,      <k> =  1.7
...
```

Figure 86: Running inline VisIt within Mercury.

6.8 Conclusions

We have shown how VisIt is connected *inline* to a running Mercury Monte Carlo simulation for real time visualization and analysis of simulation results. This functionality is invaluable for code developer verification of algorithms, user verification of problem geometry setup, and visualization of simulation results. We have described our algorithm for converting combinatorial geometry data to mesh data for visualization with mesh based tools. VisIt has also been enhanced to natively discretize and visualize CG data. Mercury has an embedded Python prompt which can be used to launch and attach VisIt to Mercury and to hand VisIt Python scripts which contain plot commands to automatically create plots in VisIt.

7 Other Scalable Algorithms

This final chapter contains some additional parallel algorithms required for Monte Carlo particle transport. Sections 7.4 on Spatial Redecomposition and Section 7.5 on Domain to Processor Assignment are preliminary investigations of these topics and are areas for future research.

The remainder of this chapter is organized as follows. Section 7.1 discusses sourcing particles scalably. The new scalable sourcing algorithm maintains reproducibly, and source particle random number seeds are independent of the number of processors used. Section 7.2 highlights scalable particle streaming communication and a scalable algorithm for deciding that particle streaming communication has finished. We present scaling results of domain-decomposed Monte Carlo particle transport up to 2^{21}=2,097,152 MPI processes. Section 7.3 highlights some of the generality of the Mercury particle tracker by tracking to user defined arbitrary surfaces. This capability enables a user to write a C-function *f(x,y,z)* that is the implicit function definition of a surface and to use that surface in the definition of the problem geometry. Section 7.4 is a preliminary investigation of an alternative load balancing technique, called *spatial redecomposition*, where domains are dynamically resized to balance the particle workload. Section 7.5 is a preliminary investigation of domain to processor assignment. We demonstrate with simple test problems that domain to processor assignment can significantly affect the runtime of a calculation.

7.1 Sourcing Particles

In each Mercury Monte Carlo calculation, a particle *source* must be present that starts the calculation. For a criticality eigenvalue calculation, the source should be some approximation of the final converged particle distribution. If the source distribution is closer to the converged distribution, then fewer iterations are required to achieve convergence. Another type of calculation is called a “source” calculation in which the user specifies the particle source and Mercury then simulates the particle transport through the background material either dynamically as a function of time or statically (independent of time). The user specifies the particle source distribution in terms of: time (t), position (x,y,z), energy (E), and angle (θ, φ). Any subset of the variables can be either correlated (up to 7 dimensional space) or uncorrelated. The important point from a parallel algorithms point of view is that the code is sampling some user defined spatial coordinate distribution, and the algorithm cannot predict where the coordinate will be, making it difficult to domain decompose. The domain decomposition of the background materials prevents an algorithm from knowing if a particle will be on a processor without actually *generating* the coordinates of the particle and *testing* to see if that coordinate is on each processor.

The original (non-scalable) particle sourcing algorithm would redundantly have every processor loop over all of the particles in the problem, generate each particle’s spatial coordinates according the user-requested spatial distribution, and then check to see if each processor owned that point in space. This algorithm had the property that the particles were always generated in the same order, so the particle properties were sampled identically, independent of the number of processors the calculation was run on. This feature is a desirable property for reproducibility. This algorithm also has the desirable property that each particle is on the correct processor that owns that point in the geometry, so no communication is required. The problem with this algorithm is that it loops over the *global* number of particles, which is not scalable.

The pseudocode of the original sourcing algorithm is shown in Algorithm 3. The important point to note is that the loop is over the total number of particles on *all* processors, so as the total number of particles increases, the algorithm takes longer and longer. We want an algorithm that will loop over the local number of particles on each processor, so each processor has a constant amount of work for a weak scaling test problem.

```
OldSourcingAlgorithm()
   seed = UserInput.SourceSeed
   for ( i = 0; i < totalNumGlobalParticles; i++ )
      particle.seed = SpawnSeed(&seed)
      particle.coordinate = SampleSourceCoordinate(&particle.seed)
      if ( IsOnThisProcessor(particle.coordinate) )
         KeepThisParticle(particle)
```

Algorithm 3: Old sourcing algorithm.

Figure 87 shows the difference between the outcome of the old sourcing algorithm (left) and the new sourcing algorithm (right). With the old sourcing algorithm, the particles are *already* on the correct processor, at the expense of testing *every* particle to see if it is on *each* processor. The new sourcing algorithm does *not* enforce that particles are created on the correct processor (the processor that owns the background geometry). The new algorithm evenly divides up the source particles, making it scalable, but then requires a subsequent communication step to put particles on the correct processor.

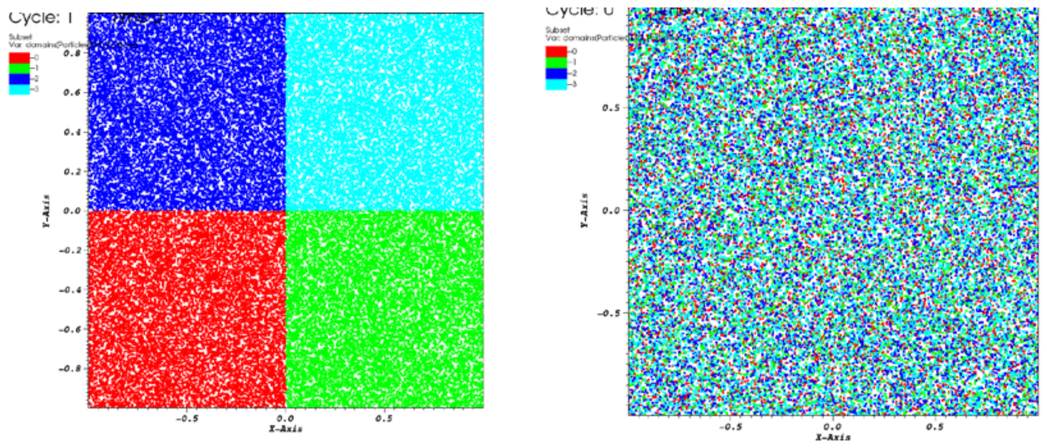


Figure 87: Old sourcing algorithm (left) and new sourcing algorithm (right).

The new sourcing algorithm maintains reproducibility, but does not necessarily generate particles on the processor that owns that coordinate of geometry, therefore introducing a communication requirement to send the particle to the correct processor that owns that section of space. Each particle is given a global particle number, and then each MPI rank assumes ownership of some range of global particle numbers. A particle's random number seed is spawned from the global particle number. Since the particle's global number determines its random number seed, which is independent of the number of processors being used, the algorithm is reproducible, independent of the number of processors. After a particle has a random number seed, the seed is used to sample the spatial distribution of the particle. This sampling may produce a coordinate that is *not* owned by the current processor, so the algorithm may produce particles on the wrong processors, which requires communication to resolve.

We have a scalable (and reproducible) particle sourcing algorithm (shown in Algorithm 4), but particles may be generated on the "wrong" processor, so we need a way to communicate them to the correct processor. We previously described the communication step in Chapter 5 on globally resolving particle locations on the correct processor.

```
NewSourcingAlgorithm()
   // Each processor calculates myStartNumber and myEndNumber
   (myStartNumber, myEndNumber) = GetMyParticleRange()
   for ( i = myStartNumber; i < myEndNumber; i++ )
      seed = UserInput.SourceSeed + i
      particle.seed = SpawnSeed(&seed)
      particle.coordinate = SampleSourceCoord(&particle.seed)
      KeepThisParticle(particle)


GetMyParticleRange()
    // This is a trivial algorithm that divides global
    // particle numbers contiguously among numProcessors
    // processors.  Each processor has a different value
    // for rank, 0 <= rank < numProcessors.
    myNumParticles = totalNumGlobalParticles / numProcessors
    myStartNumber = rank * myNumParticles
    remainder = totalNumGlobalParticles % numProcessors
    if ( rank < remainder )
        myNumParticles++
        myStartNumber += rank
    else
        myStartNumber += remainder
    myEndNumber = myStartNumber + myNumParticles
    return (myStartNumber, myEndNumber)
```

Algorithm 4: New scalable and reproducible sourcing algorithm.

7.2 Deciding That Particle Streaming Communication has Finished

For domain decomposed Monte Carlo transport calculations, particles must be communicated from one processor to another when particles reach domain boundaries, see [67], [68], [37] and [69]. This communication is handled by buffering the particles and using non-blocking point-to-point communication. The local communication is inherently scalable since domains only couple to their nearest neighbors. A challenging aspect of the communication is deciding when the calculation is finished processing all of the particles and finished communicating. Particles can be created and destroyed as they are being tracked, and they are streaming from one processor to another, making it difficult to determine when the calculation is done. We have implemented essentially the same algorithm described by Brunner and Brantley [70]. Brunner and Brantley count the total number of particles started and finished, but we simply count the total number of messages sent and received. We chose this approach is for the practical reason that there are 12 places in the code where particles are created and 22 places in the code where particles are finished; to avoid the maintenance problem, we just count sends and receives.

The idea of the algorithm is to implement a "non-blocking" reduce and a "non-blocking" broadcast using tree-based point-to-point communication. As particles are tracking, we "reduce" the total number of messages sent and received. When these counts tentatively match, we broadcast down the tree a message to perform a blocking reduce. Analysis of this algorithm predicts that it scales logarithmically (the depth of the tree), and that is logarithmic scaling we observe empirically.

Figure 88 shows a weak scaling test of the time it takes to decide the calculation is done communicating and processing particles. This is the exact same domain decomposed infinite medium weak scaling test problem described in Section 5.2, except that particles are source locally instead of globally. This calculation was run on IBM BG/Q rzuseq (same architecture as sequia, but 1/192 its size) up to 2^{15}=32,768 processors. The plot is approximately linear on a log-linear scale, so the runtime is O(log(N)), which is what we would expect to communicate messages up and down a binary tree.

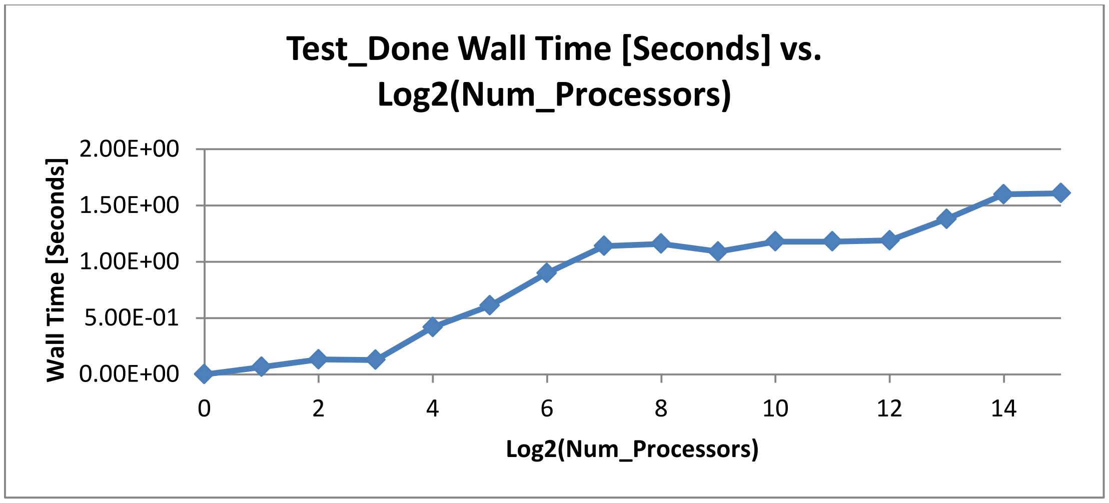


Figure 88: Weak scaling test results on IBM BG/Q rzuseq, up to 2^{15}=32,768 processors.

Figure 89 shows the particle tracking time for the largest domain decomposed Monte Carlo particle transport calculation that has been performed with Mercury. The is the same domain decomposed infinite medium weak scaling test problem described in Section 5.2, except in this test problem, particles are sourced *locally*, so we do *not* run the algorithm to globally communicate particles to the correct processor, since they are already created on the correct processor. This problem tests the particle streaming communication and the “test for done” algorithm. We see amazing scaling results out to 2^{21} = 2,097,152 MPI processes. We do not know the cause of the spike in runtime at 2^{20} processors, this calculation was only done once and the sequoia supercomputer was not yet in production mode.

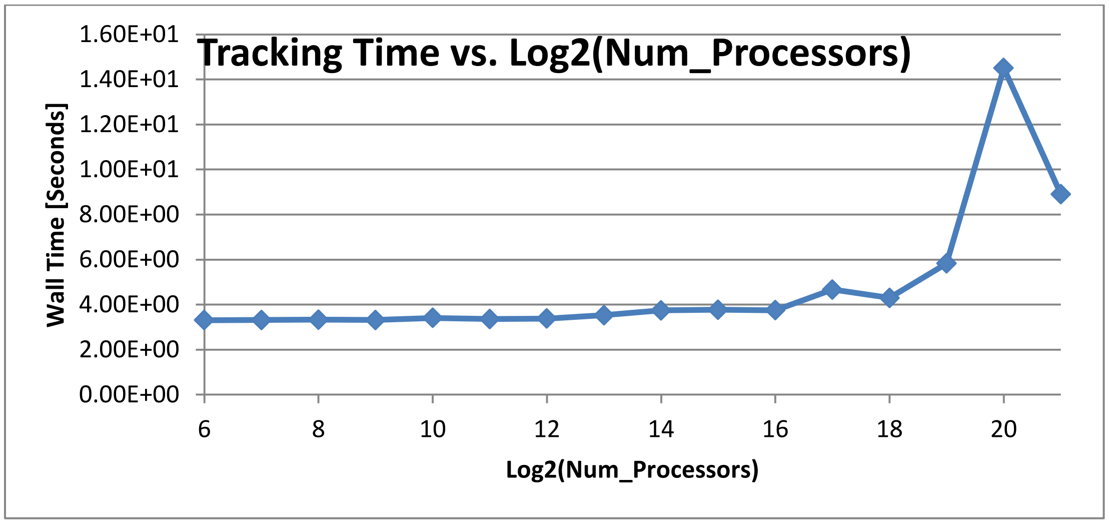


Figure 89: Tracking Time vs. Log2(Num_Processors).

For load balanced problems, this algorithm has performed quite well, as seen in the above example run up to 2^{21} processors. But for load-imbalanced problems, we found that some processors were doing too much "test for done" communication and this decision algorithm was called over one billion times for a single cycle. In an attempt to reduce the amount of non-blocking reduction communication up the tree, every 1024 messages up the tree we call MPI_Issend() instead of MPI_Isend(). Note the double "s" in MPI_Issend(). The extra "s" stands for *synchronous* and enforces more synchronization between the sender and receiver. This synchronization does not allow the sender to get too far ahead of the receiver and makes the algorithm perform significantly better for load imbalanced problems.

Conclusion

We have re-implemented our "test for done" parallel algorithm to be scalable, argued that the theoretical runtime of the algorithm should be logarithmic, and showed the results of weak scaling studies to over 2 million processors. This work enables the code to run efficiently on extremely large processor counts.

7.3 Tracking to Arbitrary Surfaces

As part of the built in particle tracker, Mercury can track to 4th order surfaces. 4th order surfaces were added to support tracking to a torus, with major radius *R* and cross sectional minor radius *r*, which has an equation

$$f(x,y,z) = (x^2 + y^2 + z^2 + R^2 - r^2)^2 - 4R^2(x^2 + y^2)$$

The equation of a torus falls into the general class of 4th order surfaces:

$$f(x,y,z) = \sum_{\substack{0 \le i+j+k \le 4 \\ i,j,k \ge 0}} a_{ijk}(x-x_0)^i (y-y_0)^j (z-z_0)^k = 0$$

Most of the typical problems encountered can be modeled with only second order surfaces which include the sphere, ellipsoid, cone and cylinder (among others).

Generalizing this, Mercury allows arbitrary user defined surfaces. The user inputs the implicit function equation of the surface *f(x,y,z)* as either a python function or a C-function. Below is an example of how to do that for a sphere of unit radius.

```
python

def PyShere(x, y, z):

      return x*x + y*y + z*z – 1.0

end_python

c_code

double C_Sphere(double x, double y, double z) {
        return x*x + y*y + z*z – 1.0; }
end_c_code
```

The user may then use the following functions to define surfaces and cells:

$$Surface = \{(x,y,z) : f(x,y,z) = 0\}$$

$$Inside = \{(x,y,z) : f(x,y,z) < 0\}$$

$$Outside = \{(x,y,z) : f(x,y,z) > 0\}$$

Using this form to describe the surface makes it very easy to answer the question of whether or not any given point is inside of the surface. We just evaluate the point in the surface equation and examine the sign of the answer. The function is zero *on* the surface, negative *within* the surface, and positive *outside* of the surface. In order to track a particle to a surface boundary, we must implement a numerical root solver. Given a particle with position (x,y,z) going in direction (α,β,γ), we must solve for the smallest positive t such that:

$$f(x + \alpha t, y + \beta t, z + \gamma t) = 0$$

Solving this equation requires a one-dimensional numerical root solver that must evaluate the function many times as the solver iterates to converge to a solution. Our initial implementation used a python user defined function, and a significant amount of overhead exists in calling the python function repeatedly from the C++ root solver. We also implemented a way for the user to input a C-function that is compiled into a dynamic library; this dynamic library is loaded and the function pointer is obtained to call that function from the numerical root solver. Much less overhead exists in calling the C-function compared to the python function, so the C-function is much faster than the python function.

We set up a test problem of a single sphere and compared our built in sphere tracker to both the python sphere and the C-sphere. See Table 12 for the results of both Samplings Points (which is just evaluating the function) and Tracking (which is the numerical root solve). We can see that when just sampling points, the user defined C-function (1.8 seconds) is slightly faster than the built in evaluate (2.0 seconds), but the overhead of calling a python function is too high and the python is slower (5.1 seconds). The built in evaluation requires more memory indirection to retrieve the equation coefficient from memory and that probably explains why it is slightly slower than the C-function evaluation, where the coefficients are directly stored in the function definition. For the **Tracking** column, our built in quadratic equation solver (2.2 seconds) is faster than the C-numerical root solver (4.7 seconds), and in the case of the Python function (86.2 seconds), the numerical solve is 18 times slower than the C-function because of the overhead of calling the python function evaluation so many times.

Table 12: Timing results for sphere test problem.

	Sampling Points	Tracking
Built In Quadratic Solver	2.0	2.2
User Defined in Python	5.1	86.2
User Defined in C	1.8	4.7

The next test problem exercises tracking to a torus. It has four tori as shown in Figure 90. Solving a 4th order equation explicitly requires a large number of floating point operations, so we see that we can solve it faster using a numerical root solver (17.1 seconds for explicit solve vs. 8.2 seconds for the numerical solve; the excessive python overhead comes in at 477.5 seconds). The extra memory indirection required to retrieve the equation coefficients for the built in torus evaluation (64.7 seconds) makes it slower than just directly evaluating the function using the user defined function (37.8 seconds), the excessive python overhead comes in at 351.6 seconds.. See Table 13 for timing data for the Four-Torus test problem.

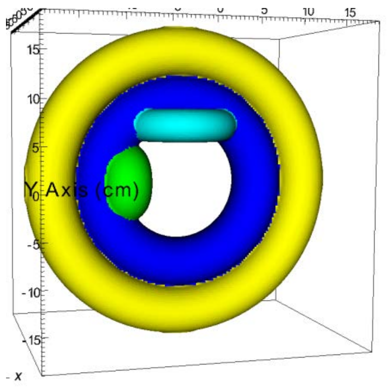


Figure 90: 4-Torus test problem.

Table 13: Timing results for the 4-Torus test problem.

	Sampling Points	Tracking
Built In 4th Order Solver	64.7	17.1
User Defined in Python	351.6	477.5
User Defined in C	37.8	8.2

We have used the generality of the user defined arbitrary surfaces to track to extruded surfaces, surfaces of revolution, and other implicitly defined shapes. This feature also works with domain decomposition; see Figure 91 for a gallery of examples. When the figure is all red, we are showing the inside of the surface. When the surface is multi-colored, we are showing the domain decomposition of the surface. The implicit function definition is stored redundantly on all processors, but particles are distributed and localized to domains.

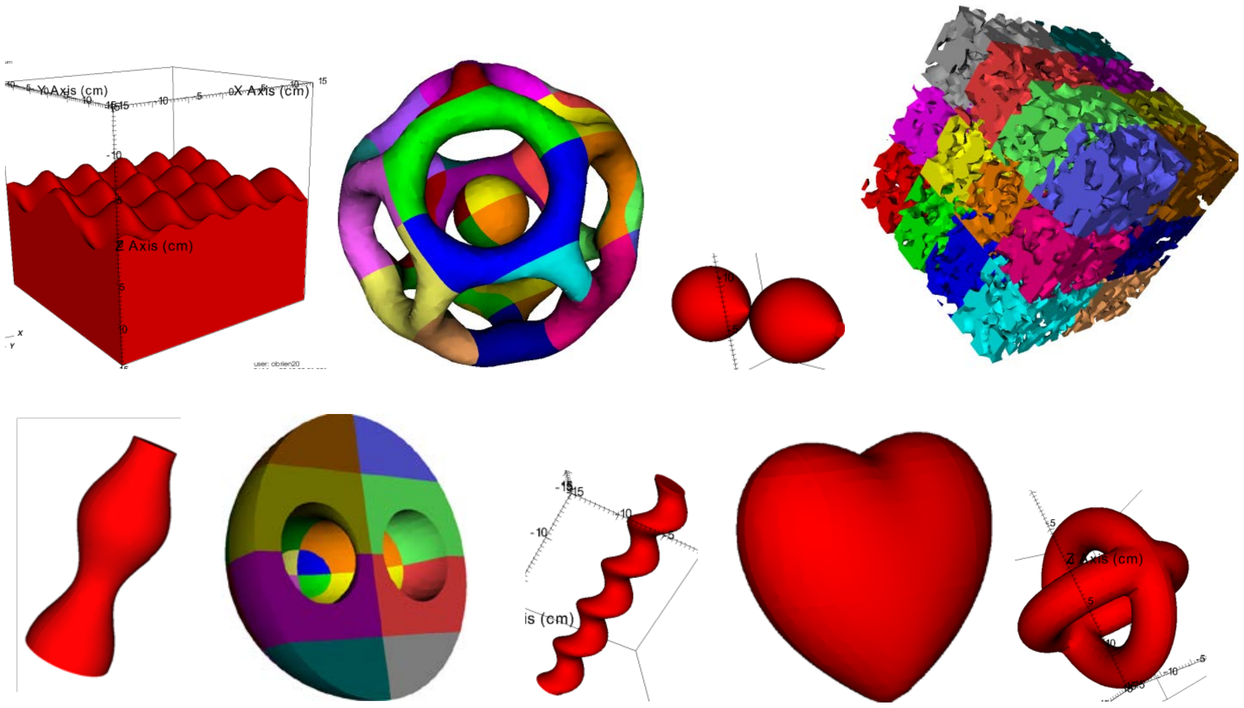


Figure 91: Gallery of user defined implicit functions.

7.4 Spatial Redecomposition

As a Monte Carlo particle transport calculation evolves over time, the particle distribution and workload also changes. This creates the requirement of some type of dynamic load balancing, which can respond to the current particle workload and reassign processing resources where they are needed the most. See Figure 92 for an example of an evolving workload as a function of time.

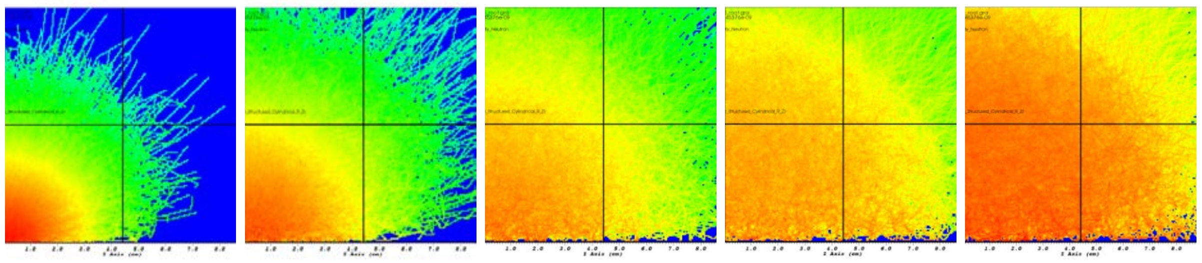


Figure 92: Evolving workload as a function of time.

By dynamically resizing domains to attempt to balance the workload in each domain, we can make the calculation run faster. Figure 93 shows an octant of a Godiva Uranium sphere criticality problem. The uniform domain decomposition is only 28% efficient; some domains have too much or too little work. The spatially-weighted domain decomposition is 85% efficient; the algorithm attempts to resize domains to balance the particle workload.

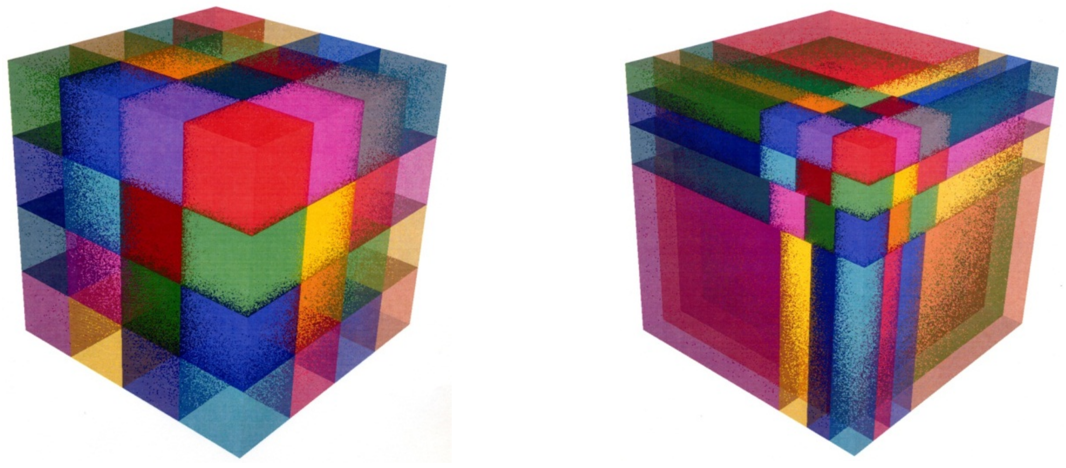


Figure 93: Uniform (left) vs. spatially weighted (right) domain decomposition.

The way the algorithm works is by summing the work along one coordinate axis and then placing domain decomposition planes to divide the workload evenly along that coordinate axis. As a result each partition has about the same amount of work. This process is repeated independently in each partition along a different coordinate axis. Those sub-partitions now have about the same amount of work, and a third and final partitioning step is performed along the remaining coordinate axis. Figure 94 shows an illustration of the partitioning algorithm in 1 dimension.

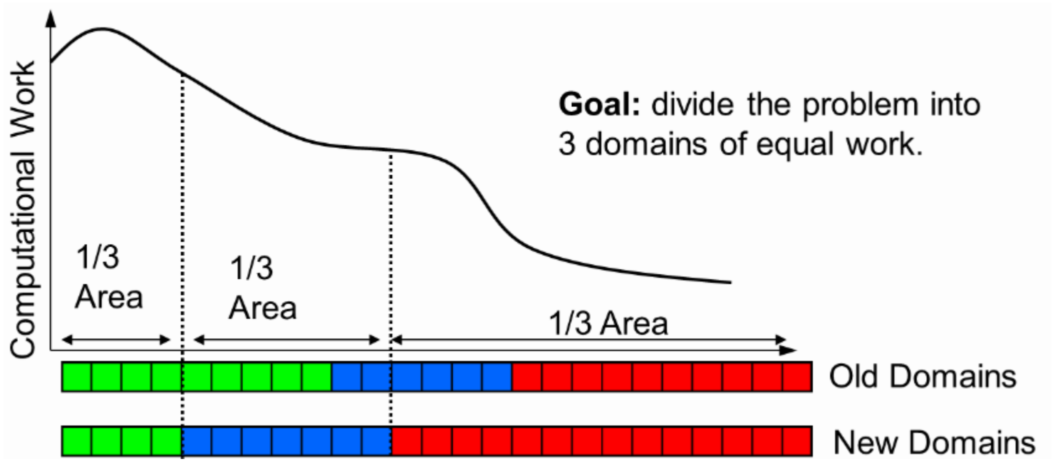


Figure 94: The load balancing algorithm finds new domain boundaries so that each domain has the same amount of work.

Figure 95 shows an example of the partitioning algorithm load balancing a 2D problem. The first step is to balance the work along the vertical axis. This process creates three partitions, each with about the same amount of work. Next, each of these partitions is balanced independently in the horizontal direction, shown on the right of Figure 95.

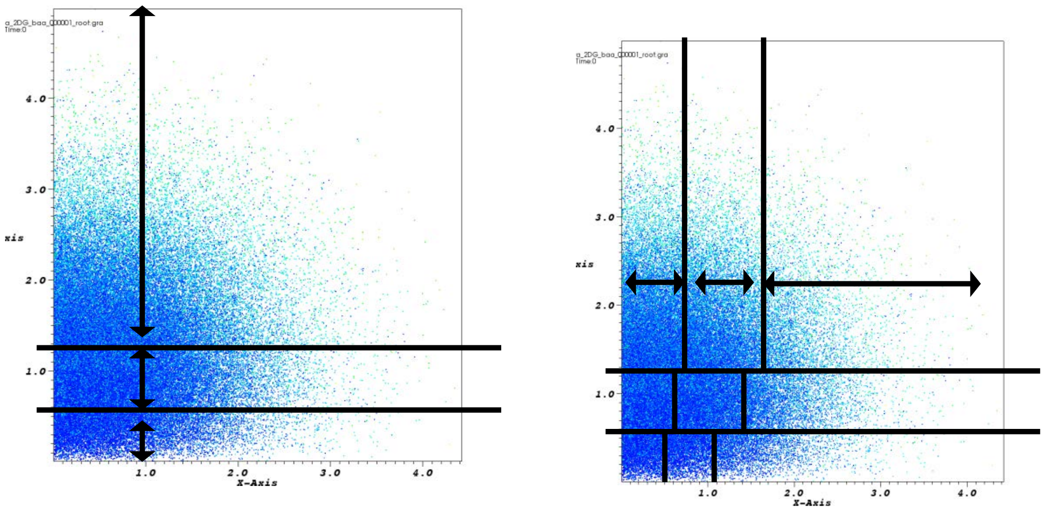
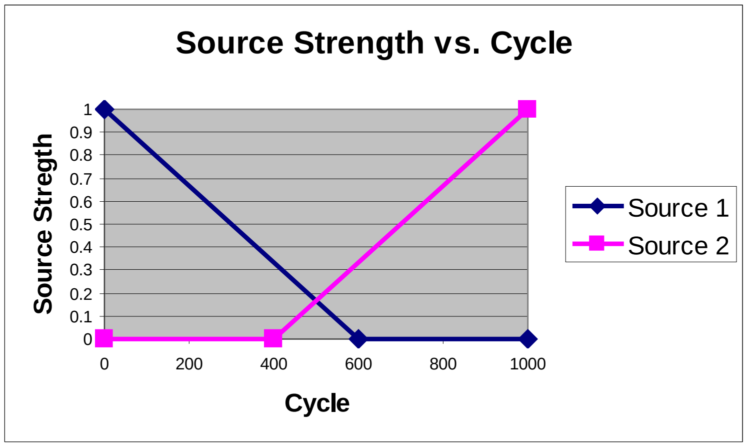


Figure 95: 2D example of partitioning algorithm.

In the following example, we have two strong particle sources at opposite ends of the problem. This test problem is a contrived example to have a concentration of work in the lower left corner in the beginning of the problem and a concentration of work in the upper right corner at the end of the problem, as seen in Figure 96. The domains resize according to the particle workload. Figure 97 shows small domains in the lower left corner in the beginning of the problem, when there is a large concentration of work there. Later in time, we have large domains in the lower left corner and small domains in the upper right corner as work shifts there.



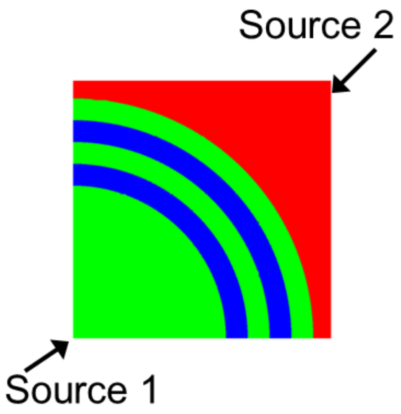
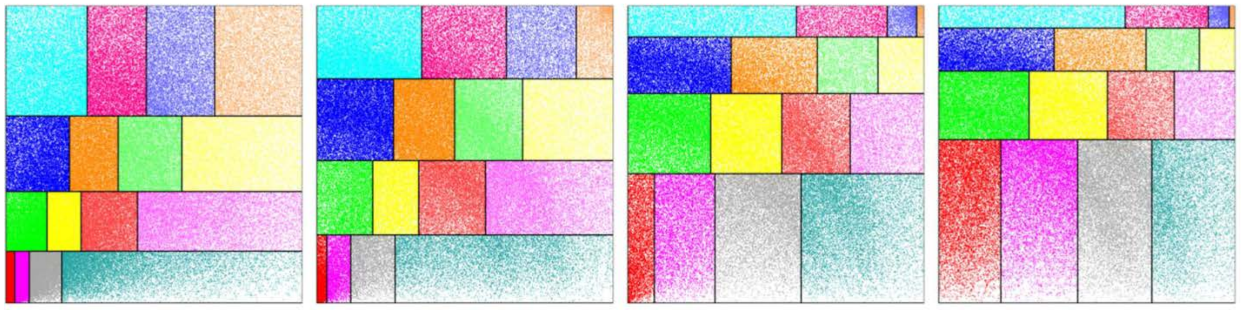


Figure 96: Source Strength (left) and Source positions (right).



Cycle 11 Cycle 481 Cycle 581 Cycle 611

Figure 97: Domain boundaries adjust to balance particle workload.

Figure 98 shows a comparison of running the same problem (Spherical Shield Sourced Problem from Section 3.5.2) four different ways. Each run used 16 processors.

1. **Fixed Decomposition:** 16 domains, no load balancing.
2. **Spatial Redecomposition:** 16 domains that are allowed to resize to balance the workload.
3. **Fixed Replication:** 4 domains, each replicated 4 times, no load balancing.
4. **Dynamic Replication**: 4 domains, the replication level is allowed to change to balance workload.

Fixed Decomposition runs the slowest, because the *most worked* processor controls the overall runtime. A concentration of work exists in the lower left corner of this problem. The processor in the lower left corner has the most work, so the efficiency = “ave/max” is minimized and run time is maximized. Fixed replication runs somewhat faster, because the size of the domains are increased and the workload is spread evenly over four replications. Spatial Redecomposition runs the fastest, because it has the most degrees of freedom to balance the workload. With dynamic replication, we have 16 processors that we need to assign to domains, so assigning an entire processor to a domain is a relatively large granularity movement to balance the workload. With dynamic redecomposition, we move individual domain boundaries by one zone at a time. As a result, we have finer granularity to balance the workload, and we see the highest average efficiency and the lowest total runtime.

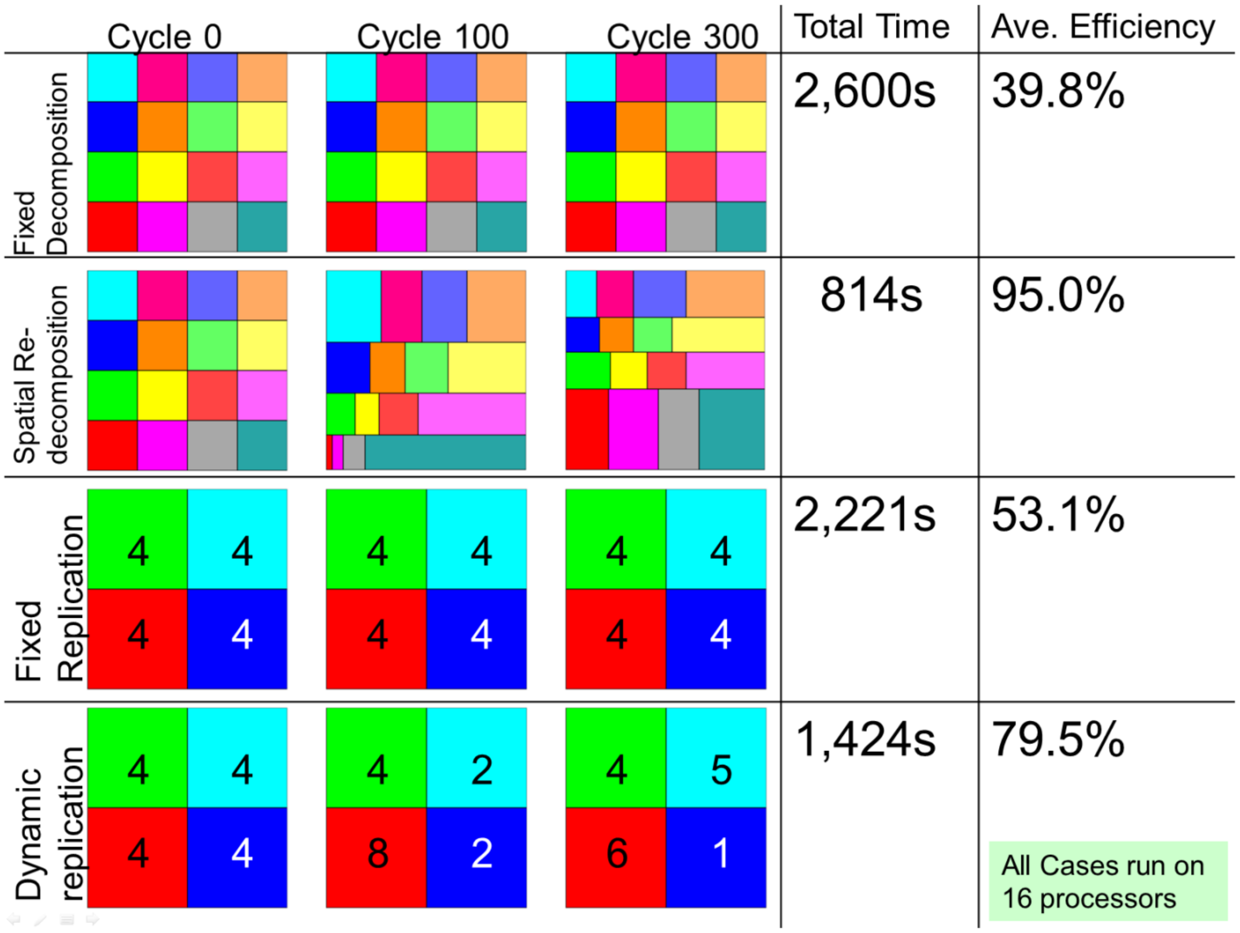


Figure 98: Comparison of load balancing methods.

Many problems have a non-uniform distribution of particles in space and time. This creates the need for a dynamic load balancing algorithm. Two dynamic load balancing methods have been implemented: dynamic replication (discussed in Chapter 3) and dynamic spatial redecomposition (discussed in this section). Both methods can typically speed up a problem by a factor of 2-3. Spatial redecomposition can be more effective since it has finer grained control by moving zones instead of processors. Future work involves scaling these methods to large processor counts and combining the two methods, which gives more degrees of freedom to optimize over which should lead to improved load balance.

7.5 Domain to Processor Assignment

When running large scale parallel, domain decomposed Monte Carlo particle transport calculations, the assignment of domains to processors is important for load balancing and for minimizing the inter-processor particle communication. These factors are important for using a supercomputer efficiently and finishing calculations as quickly as possible. In this section, we examine the impact of various assignments of domains to processors and present an algorithm for minimizing inter-processor communication. Other high performance computing applications may only have one domain per processor, but it is common for Mercury to have more than one domain per processor, and we want to localize the domains assigned to the same processor.

Introduction

Currently we simply assign domains to processors via an "unintelligent" round-robin process. The assignment is not based on any criterion that tries to optimize anything; adjacency information is being ignored. By investing some effort in defining to which processor each domain is assigned, we can dramatically speed up the code. In this case, by intelligently defining the domain to processor mapping, and making *zero* changes to the rest of the code, we can achieve a 10-20% speedup in code performance. More investigation needs to be done for large processor counts; presumably the speedup will increase with processor count.

Example 1: Checkerboard

In this first example, we need to assign 256 domains to two processors. In order to minimize the surface area of domain boundaries on different processors, we should group domains together that are assigned to the same processor. We look at two basic methods: (a) "*Short Stride*" meaning we assign domains "round-robin" to processors, and (b) "*Long Stride*" meaning we assign the first half of the domains to one processor, and the second half to the other processor. The *Short Stride* method requires the most amount of particle streaming communication, since as a particle streams through the problem, it must be communicated at *every* domain boundary crossing. The *Long Stride* method is optimal, since it requires the least communication as particle stream through the problem.

Figure 99 shows 16 x 16 domains, run on 2 processors, with blue domains assigned to processor 0 and red domains assigned to processor 1. This test problem is an infinite medium (reflecting boundary conditions) double density Uranium-235, pseudo-dynamic alpha eigenvalue criticality calculation, with 100,000 particles.

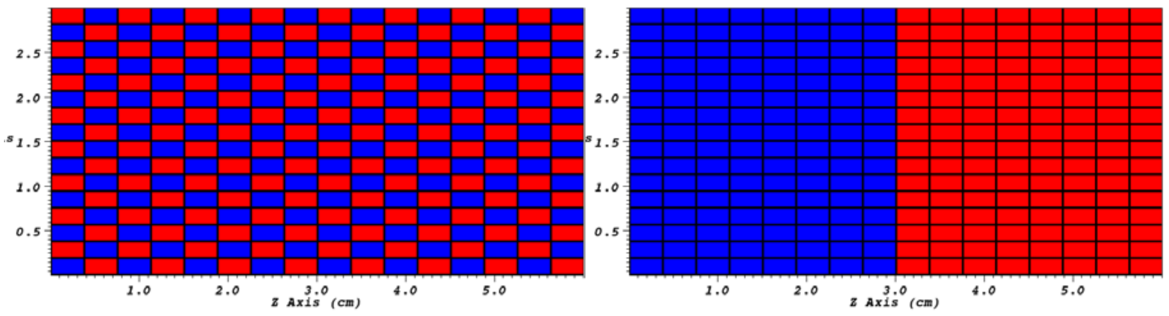


Figure 99: Short Stride (left), Long Stride (right).

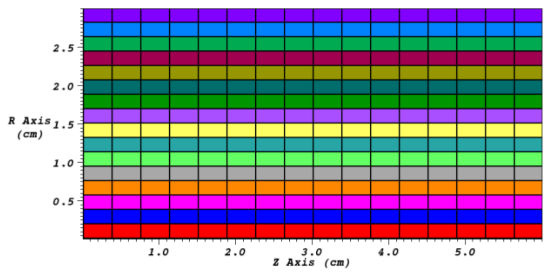
Table 14 shows the run time for various numbers of zones per domain. As the number of zones per domain increases, more time is spent tracking to zone boundaries. As a result the total time increases, and a smaller fraction of the time is spent doing communication. The speedup therefore decreases with more zones per domain.

Table 14: Particle Tracking Time for Short Stride vs. Long Stride.

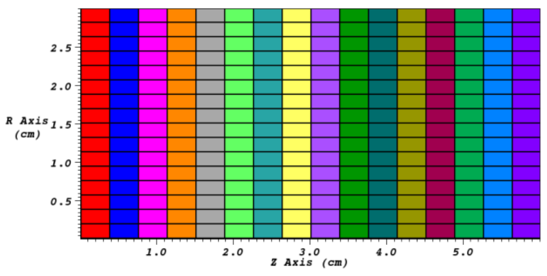
	Particle Tracking Time (sec)		
Domain size	Short stride	Long stride	% speedup
2 x 2 zones, 2D	114	88	23%
4 x 4 zones, 2D	197	162	18%
8 x 8 zones, 2D	386	334	13%
16 x 16 zones, 2D	805	695	13%
2 x 2 zones, 3D	28	18	36%
4 x 4 zones, 3D	57	35	39%
8 x 8 zones, 3D	110	70	36%
16 x 16 zones, 3D	153	132	14%
32 x 32 zones, 3D	290	275	5%

Example 2: Unequal Volumes

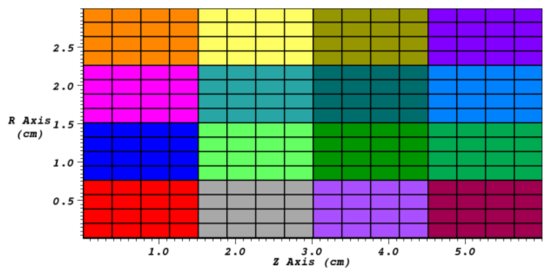
The next example shown in Figure 100 investigates the effect that 2D cylindrical coordinates have on the problem. Both (a) 1 x 16 and (b) 16 x 1 *appear* to be load balanced. However, with the cylindrical geometry, these domains are revolved around the horizontal axis, so for a fixed area, the *revolved volume* gets larger with larger radial (vertical) component. The problem we are running is homogeneous with reflecting boundary conditions, so the computational workload is proportional to the domain volume. Therefore (a) 1 x 16 is *not* load balanced, since the volume of the domains assigned to each processor is different, while (b) 16 x 1 *is* load balanced since all the volumes of domains assigned to each processor is the *same*. Also (c) 4 x 4 is *not* load balanced, since the volume of domains assigned to each processor are different, while (d) Diagonal *is* load balanced, since the volume of domains assigned to each processor is the same.



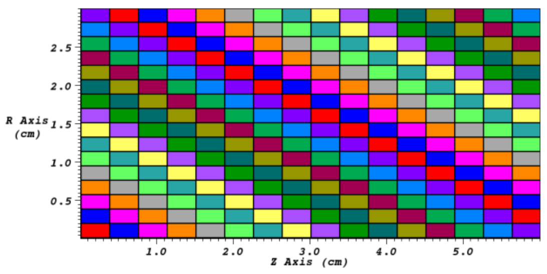
(a) 1 x 16



(b) 16 x 1



(c) 4 x 4



(d) Diagonal

Figure 100: Comparison of various domain decompositions.

See Table 15 for the wall time results for running the various domain to processor assignments.

Table 15: Comparison of runtimes of various assignments of domains to processors.

	Particle Tracking Wall time in seconds. 256 domains, 16 processors.			
	(a) 1x16	(b) 16x1	(c) 4x4	(d) Diagonal
Cylindrical 2D	35	19	27	24
Cartesian 3D	5.2	3.7	3.3	5.3

The numbers in Table 15 (for 2D cylindrical) are hard to interpret since there is more than one effect occuring on here. The problem is that we are running the R-Z cylindrical geometry, the axis of the cylinder is the horizontal axis and the vertical axis is the radial axis. The volume of a domain is "r Δr Δz" instead of the Cartesian area "Δx Δy", so we have an extra radial factor. Looking at the pictures is misleading. We are modeling a homogeneous infinite slab of material, so the computational work is proportional to the volume. We have different volumes assigned to processors, hence different workloads per processor, so the calculation is not load balanced. (Load balancing is turned off). Note that the 1x16 case has the largest discrepancy in the processor volumes and hence runs the slowest. Followed by 4x4, the second slowest. But 16x1 has equal volume per processor and is the fastest. "diagonal" also has equal volume per processor, but has more communicating surface area and hence runs slower. So it *is* a fair comparison to compare 16x1 to "diagonal", since both problems are load balanced, the only difference is the communication, which is what we are studying in this section.

We also ran this problem in 3D Cartesian space, with reflecting boundary conditions so all processors have the same volume and hence the same workload, now the only difference is in the communicating surface area, which is the effect we indend to study in this section. Now we see that 1x16 is slower than 16x1. This is because there is more communicating surface area in the 1x16 case. 4x4 runs the fastest, since the communicating surface area is minimized. Diagonal runs the slowest since it has the largest communicating surface area.

Conclusion

This brief investigation was meant to demonstrate that the assignment of domains to processors matters and can significantly affect the particle communication time and hence the overall runtime of the problem. We illustrated that a simple "round-robin" assignment of domains to processors may not be optimal and with a small amount of work, a better assignment can be realized. This algorithm will be left for future work, where we intend to do a scaling study to large processor counts studying the effect of domain to processor mapping, and look for optimal mappings which minimize total run time.

Conclusion

In this dissertation, we described the parallel algorithms necessary to efficiently run domain decomposed Monte Carlo particle transport on large numbers of processors.

Chapter 2 described how to domain decompose CSG for Monte Carlo particle transport. Many mesh-based physics simulations have been parallelized via domain decomposition, but as far as we know, Mercury was the first CSG Monte Carlo particle transport code to be domain decomposed. Domain decomposition enables truly large calculations. Without domain decomposition, the size of the calculation is limited by what can fit in the memory of *one* node of a computer. With domain decomposition, you are only limited by the *global memory* of the *entire* supercomputer.

Chapter 3 presented a load balancing algorithm for domain decomposed problems. This novel algorithm has significantly improved the performance of almost all of our Mercury simulations. Performance improvements of a factor of 2-3 compared to not load balancing are common. Some aspects of this algorithm are not scalable and future research will address this issue.

Chapter 4 described a scalable *homogenous* load balancing algorithm in which any processor can operate on any of the particles. This is the case in particle parallel (domain replicated) problems. This algorithm has *O(log(N))* communication steps and is therefore scalable. We presented a scaling study of this algorithm up to 2^{21} = 2,097,152 MPI processes on the IBM BG/Q *Sequoia* supercomputer that agreed with our theoretical predictions.

Chapter 5 described a scalable global particle find algorithm. Particles may be sourced on a processor that does not own the background geometry required to process the particle. So the particles must be communicated to the correct processor that owns the background geometry. A hypercube graph was constructed over the processors to efficiently communicate particles from processor to processor, ultimately enabling particles to make it to the correct processor.

Chapter 6 considers the problem of visualizing CSG, given mesh-based visualization tools. Therefore, the problem becomes how to convert the CSG to a mesh. We implemented several algorithms for this, including:

- recursive zone sampling to calculate volume fractions
- numerical integration techniques to calculate volume fractions
- conformal mesh generation by moving mesh nodes onto the CSG surface

As an alternative to converting the CSG to a mesh, we can directly visualize the CSG using in-line ray casting.

Chapter 7 covered some remaining parallel algorithms and suggested two areas for future research: load balancing via spatial redecomposition and domain to processor assignment. We presented a scalable and reproducible particle sourcing algorithm. We presented a scaling study up to 2^{21} = 2,097,152 MPI processes, which exercises domain decomposed particle streaming communication and the algorithm to decide if particle streaming communication has finished.

The major contributions of this dissertation are (by Chapter):

1. Motivated the need for scalable algorithms. Non-scalable algorithms are not feasible and parallel overhead takes significantly longer than useful computation.

2. Designed and implemented *domain decomposition* for combinatorial geometry.
    a. Enables extremely large cell counts: ran 89 million cell problem.
3. Designed and implemented a domain decomposed load balancing algorithm.
    a. A domain's *replication level* is determined each cycle based on the particle workload relative to the other domains.
    b. Some aspects of this algorithm are not scalable and will be addressed in future research.
4. Designed, implemented and proved correct a scalable homogeneous load balancing algorithm.
    a. Demonstrated scalability of this algorithm up to 2 million MPI processes.
    b. This algorithm is used for particle parallelism (domain replication) only.
5. Designed, implemented and proved correct a scalable global particle find algorithm.
    a. Relying on this algorithm allows for scalable particle sourcing.
6. Implemented several novel methods for visualizing combinatorial geometry.
    a. Convert CG to mesh via recursive sampling, numerical integration, or conformal mesh generation. Or visualize geometry via in-line ray casting.
7. Other Scalable Algorithms:
    a. The new particle sourcing algorithm is scalable, and particle random number seeds are insensitive to the number of processors used.

b. Demonstrated scalability of particle streaming communication and *test for done* algorithm up to 2 million MPI processes.

c. Added the ability to track to arbitrary user defined surfaces for flexibility in problem definition.

d. Demonstrated spatial redecomposition as an alternative load balancing technique. More research should be done in this area.

e. Demonstrated runtime sensitivity to domain to processor assignment. More research should be done in this area.

These algorithms form the foundation for scalable domain decomposed Monte Carlo particle transport and are used in the production quality software tool Mercury developed at LLNL.

Table 16 shows the computational complexity of the old run time and the new run time of various algorithms in Mercury. This table assumes weak scaling in which the calculation has constant work per processor; if the number of processors is doubled, the number of particles is also doubled.

Table 16: Runtimes of old and new parallel algorithms.

Algorithm (n = number of processors)	**Old Run Time**	**New Run Time**
Particle Sourcing	$\Theta(n)$	$\Theta\ (1)$
Global particle find: Cartesian Domains	$\Theta\ (n)$	$\Theta\ (\ (\log n)\ (\log \log n)\ )$
Load Balancing within a workgroup	$\Theta\ (n^2)$	$\Theta\ (\log n)$
Test for done with particle communication	$\Theta\ (n^2)$	$\Theta\ (\log n)$

We have eliminated non-scalable algorithms from Mercury and we are left with algorithms that are at most proportional to powers of *log(n)*, which was the goal of this dissertation.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

This dissertation is document number LLNL-TH-647500.

Bibliography

[1] N. Metropolis and S. Ulam, "The Monte Carlo Method," *J. Am. Stat. Assoc., 44, 335*, 1949.

[2] F. Brown, "Fundamentals of Monte Carlo Transport, LA-UR-05-4983," Los Alamos National Laboratory, 2005.

[3] A. F. Bielajew, "Fundamentals of the Monte Carlo method for neutral and charged particle transport," The University of Michigan, Ann Arbor, Michigan, 2001.

[4] C. Englemann and A. Geist, "Super-Scalable Algorithms for Computing on 100,000 Processors.," *Proceedings of ICCS,* (2005).

[5] Lawrence Livermore National Laboratory, "BlueGene/L," 5 January 2011. [Online]. Available: https://asc.llnl.gov/computing_resources/bluegenel. [Accessed 2 September 2013].

[6] B. Barney, "Using the Dawn BG/P System," (2011). [Online]. Available: https://computing.llnl.gov/tutorials/bgp.

[7] "Advanced Simulation and Computing- Sequoia," Lawrence Livermore National Laboratory, (2012). [Online]. Available: https://asc.llnl.gov/computing_resources/sequoia/.

[8] P. Brantley and M. McKinley, "Mercury Web Site," (2011). [Online]. Available: https://wci.llnl.gov/codes/mercury/.

[9] R. J. Procassini, J. M. Taylor, G. M. Greenman, M. J. O'Brien and D. E. Cullen, "Design, Implementation and Optimization of a Parallel Monte Carlo Particle Transport Code," in *Computational Methods in Transport Workshop*, Tahoe City, 2004.

[10] R. J. Procassini, "Mercury User Guide (Version c.2)," Lawrence Livermore National Laboratory, UCRL-TM-204296, Livermore, 2008.

[11] R. Procassini, D. Cullen, G. Greenman, C. Hagmann, K. Kramer, S. McKinley, M. O'Brien and J. Taylor, "New Capabilities in Mercury: A Modern, Monte Carlo Particle Transport Code.," in *Joint International Topical Meeting on Mathematics & Computation and Supercomputing in Nuclear Applications.*, Monterey, 2007.

[12] R. Procassini, P. Brantley, S. Dawson, G. Greenman, S. McKinley, M. O'Brien, S. Sepke, D. Stevens, B. Beck and C. Hagmann, "New Features of the Mercury Monte Carlo Particle Transport Code," in *Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2010 (SNA + MC2010)*, Tokyo, 2010.

[13] R. Procassini, J. Taylor, S. McKinley, G. Greenman, D. Cullen, M. O'Brien, B. Beck and C. Hagmann, "Update on the Development and Validation of MERCURY: A Modern, Monte

Carlo Particle Transport Code," in *ANS Topical Meeting in Mathematics and Computations*, Avignon, 2004.

[14] P. S. Brantley, S. A. Dawson, M. S. McKinley, M. J. O'Brien, D. E. Stevens, B. R. Beck, E. D. Jurgenson, C. A. Ebbers and J. M. Hall, "Recent Advances in the Mercury Monte Carlo Particle Transport Code," in *International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013).*, Sun Valley, Idaho, 2013.

[15] Python Software Foundation, "Python," 2013. [Online]. Available: http://www.python.org. [Accessed 2 September 2013].

[16] F. Iandola, M. J. O'Brien and R. J. Procassini, "PyMercury: Interactive Python for the Mercury Monte Carlo Particle Transport Code," in *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering*, Rio de Janeiro, 2011.

[17] Python Software Foundation, "Python/C API Reference Manual," 2013. [Online]. Available: http://docs.python.org/2/c-api. [Accessed 29 September 2013].

[18] Python Software Foundation, "Extending and Embedding the Python Interpreter," 2013. [Online]. Available: http://docs.python.org/2/extending/index.html. [Accessed 29 September 2013].

[19] "MPI: A Message-Passing Interface Standard," (2011). [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi.

[20] P. Balaji, D. Buntinas, D. Goodell and W. Gropp, "MPI on a Million Processors," in *EuroPVMMPI'09*, Helsinki, Finland, (2009).

[21] OpenMP Architecture Review Board, "OpenMP," 2013. [Online]. Available: http://openmp.org. [Accessed 2 September 2013].

[22] Lawrence Livermore National Laboratory, "Silo," 2013. [Online]. Available: https://wci.llnl.gov/codes/silo/. [Accessed 29 September 2013].

[23] S. Koranne, "Hierarchical Data Format 5: HDF5," in *Handbook of Open Source Tools*, Springer, 2011, pp. 191-200.

[24] Oak Ridge National Laboratory, "Radiation Safety Information Computational Center," 2013. [Online]. Available: https://rsicc.ornl.gov. [Accessed 14 October 2013].

[25] B. L. Kirk, "Overview of Monte Carlo radiation transport codes," *Radiation Measurements,* vol. 45, pp. 1318-1322, 2010.

[26] X-5 Monte Carlo Team, "MCNP - A General Monte Carlo N-Particle Transport Code, Version 5. LA-UR-03-1987," Los Alamos National Laboratory, Los Alamos, 2008.

[27] T. Sutton, T. Donovan, T. Trumbull, P. Dobreff, E. Caro, D. Griesheimer, L. Tyburski, D. Carpenter and H. Joo, "The MC21 Monte Carlo transport code," in *Joint International Topical Meeting on Mathematics and Computation and Supercomputing in Nuclear Applications*, Monterey, California, 2007.

[28] B. Walker, J. Figgins and J. Comfort, "A Framework for Monte Carlo simulation calculations in GEANT4," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, and Associated Equipment*, vol. 768, pp. 889-895, 2006.

[29] D. E. Cullen, "TART2005 - A Coupled Neutron-Photon 3-D, Combinatorial Geometry, Time Dependent Monte Carlo Transport Code," Lawrence Livermore National Laboratory, UCRL-SM-218009, Livermore, November 22, 2005.

[30] R. M. Buck and E. M. Lent, "COG User's Manual: A Multiparticle Monte Carlo Transport Code, 5th Edition," Lawrence Livermore National Laboratory, Livermore, California, USA, 2002.

[31] P. K. Romano and B. Forget, "The OpenMC Monte Carlo particle transport code," *Annals of Nuclear Energy*, vol. 51, pp. 274-281, 2013.

[32] P. K. Romano, "Parallel Algorithms for Monte Carlo Particle Transport Simulation on Exascale Computing Architectures," Massachusetts Institutes of Technology, PhD Dissertation, Cambridge, MA, 2013.

[33] VTT Technical Research Centre of Finland, "Serpent a continuous-energy Monte Carlo reactor physics burnup calculation code," 2013. [Online]. Available: montecarlo.vtt.fi. [Accessed 10 November 2013].

[34] G. Greenman, M. O'Brien, R. Procassini and K. Joy, "Enhancements to the Combinatorial Geometry Particle Tracker in the Mercury Monte Carlo Transport Code: Embedded Meshes and Domain Decomposition," in *Proceeding from the ANS Mathematics and Computation 2009 Meeting*, Saratoga Springs, (2009).

[35] M. J. O'Brien, G. M. Greenman and R. J. Procassini, "Domain Decomposition of a Combinatorial Geometry Monte Carlo Transport Code," in *LLNL-PRES-407916*, Livermore, 2008.

[36] F. Brown, "MCNP Monte Carlo & Parallel Computing," in *University of New Mexico Workshop on Monte Carlo for Particle Therapy Treatment Planning*, Albuquerque, NM, 16-18 May 2011.

[37] H. J. Alme, G. H. Rodrigue and G. B. Zimmerman, "Domain Decomposition for Parallel Monte Carlo Transport," *The Journal of Supercomputing*, no. 18, pp. 5-23, 2001.

[38] G. E. Whitesides, "A Difficulty in Computing the k-effective of the Word," *Transactions of the American Nuclear Society*, vol. 14, no. 2, p. 680, 1971.

[39] Lawrence Livermore National Laboratory, "The National Ignition Facility: Ushering in a New Age for Science," 2013. [Online]. Available: https://lasers.llnl.gov/about/nif/. [Accessed 28 August 2013].

[40] Lawrence Livermore National Laboratory, "Laser Inertial Fusion Energy (LIFE): Tackling the Global Energy Crisis," 2013. [Online]. Available: https://lasers.llnl.gov/about/missions/energy_for_the_future/life/. [Accessed 28 August 2013].

[41] M. O'Brien, J. Taylor and R. Procassini, "Dynamic Load Balancing of Parallel Monte Carlo Transport Calculations," in *ANS Monte Carlo 2005: The Monte Carlo Method: Versatility Unbounded In A Dynamic Computing World*, Chattanooga, TN, (2005).

[42] R. Procassini, M. O'Brien and J. Taylor, "Load Balancing of Parallel Monte Carlo Transport Calculations," in *Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Application*, Palais des Papes, Avignon, France, (2005).

[43] D. E. Cullen, C. J. Clouse, R. J. Procassini and R. C. Little, "Static and Dynamic Criticality: Are They Different?," Lawrence Livermore National Laboratory, Livermore, CA, 2003.

[44] "International Handbook of Evaluated Criticality Safety Benchmark Experiments," Nuclear Energy Agency, on CD-ROM (2010).

[45] M. J. O'Brien, P. S. Brantley and K. I. Joy, "Scalable Load Balancing For Massively Parallel Distributed Monte Carlo Particle Transport," in *International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013)*, Sun Valley, Idaho, 2013.

[46] P. Romano and B. Forget, "Parallel Fission Bank Algorithms in Monte Carlo Criticality Calculations," *Nuclear Science and Engineering,* vol. 170, no. 2, pp. 125-135, (2012).

[47] A. Siegel, K. Smith, P. Romano and B. F. K. Forget, "The effect of load imbalances on the performance of Monte Carlo algorithms in LWR analysis," *Journal of Computational Physics,* pp. 901-911, 2012.

[48] Parallel Programming Laboratory, Department of Computer Science, University of Illinois, "Charm++ Parallel Objects," 2013. [Online]. Available: http://ppl.cs.illinois.edu/research/charm. [Accessed 2 September 2013].

[49] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction," Ph. D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, (2005).

[50] G. Zheng, A. Bhatele, E. Meneses and L. V. Kale, "Periodic Hierarchical Load Balancing for Large Supercomputers," *International Journal of High Performance Computing Applications,* (2011).

[51] E. E. Lewis and W. F. J. Miller, Computational Methods of Neutron Transport, La Grange Park, Illinois: American Nuclear Society, Inc., 1993.

[52] M. J. O'Brien, B. Bihari, P. S. Brantley and K. I. Joy, "Computing on Sequoia: Designing Scalable Algorithms," in *Joint Mathematics Meetings, American Mathematical Society and Mathematical Association of America*, San Diego, 2013.

[53] M. J. O'Brien, S. A. Dawson, P. S. Brantley and K. I. Joy, "Scalable Algorithms for Monte Carlo Particle Transport," in *LLNL-CONF-643319*, Livermore, 2012.

[54] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM,* vol. 18, no. 9, pp. 509-517, 1975.

[55] Lawrence Livermore National Laboratory, "Visit Web Page," 2013. [Online]. Available: https://wci.llnl.gov/codes/visit/. [Accessed 24 August 2013].

[56] Lawrence Livermore National Laboratory, "VisIt User's Manual," Lawrence Livermore National Laboratory, UCRL-SM-220449, Livermore, 2005.

[57] Lawrence Livermore National Laboratory, "Getting Data Into VisIt," Lawrence Livermore National Laboratory, UCRL-SM-224277, Livermore, 2006.

[58] VisIt Developers, "VisItUsers.org," 2013. [Online]. Available: http://www.visitusers.org. [Accessed 24 August 2013].

[59] Lawrence Livermore National Laboratory, "VisIt Python Interface Manual," Lawrence Livermore National Laboratory, UCRL-SM-209589, Livermore, 2005.

[60] M. O'Brien, R. Procassini and K. Joy, "Mercury + VisIt: Integration of a Real-Time Graphical Analysis Capability into a Monte Carlo Transport Code.," in *2009 International Conference on Advances in Mathematics, Computational Methods, and Reactor Physics.*, Saratoga, 2009.

[61] Matplotlib Development Team, "Matplotlib," 2013. [Online]. Available: http://matplotlib.org. [Accessed 29 September 2013].

[62] The Scipy community, "Numpy C-API," 2013. [Online]. Available: http://docs.scipy.org/doc/numpy/reference/c-api.html. [Accessed 29 September 2013].

[63] D. L. Millman, D. P. Griesheimer, B. R. Nease and a. J. Snoeyink, "Robust Volume Calculations For Constructive Solid Geometry (CSG) Components In Monte Carlo Transport Calculations," in *PHYSOR 2012 - Advances in Reactor Physics - Linking Research, Industry, and Education, April 15-20, 2012*, Knoxville, 2012.

[64] D. L. Millman and J. Snoeyink, "Degree-Driven Algorithm Design for Computing Volumes of CSG Models," in *CG:YRF*, Chapel Hill, NC, USA, June 16-20, 2012.

[65] M. J. O'Brien, "Dynamic Load Balancing of Parallel Monte Carlo Transport Calculations Via Spatial Redecomposition," in *M&C + SNA 2007*, Monterey, 2007.

[66] M. J. O'Brien, "Material Interface Reconstruction for Monte Carlo Particle Tracking," Lawrence Livermore National Laboratory, UCRL-TR-219915, Livermore, 2006.

[67] T. Brunner, T. Urbatsch, T. Evans and N. Gentile, "Comparison of four parallel algorithms for domain decomposed implicit Monte Carlo," *Journal of Computational Physics*, vol. 2, no. 212, pp. 527-539, 2006.

[68] A. Siegel, K. Smith, P. Fischer and V. Mahadevan, "Analysis of communication costs for domain decomposed Monte Carlo methods in nuclear reactor analysis," *Journal of Computational Physics*, vol. 231, pp. 3119-3125, 2012.

[69] J. C. Wagner, S. W. Mosher, T. M. Evans, D. E. Peplow and J. A. Turner, "Hybrid and Parallel Domain-Decomposition Methos Development to Enable Monte Carlo for Reactor Analyses," *Progress in NUCLEAR SCIENCE and TECHNOLOGY*, vol. 2, pp. 815-820, 2011.

[70] T. Brunner and P. Brantley, "An efficient, robust, domain-decomposed algorithm for particle Monte Carlo," *Journal of Computational Physics*, 2009.