



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Domain Decomposition of a Constructive Solid Geometry Monte Carlo Transport Code

M. J. O'Brien, K. I. Joy, R. J. Procassini, G. M.
Greenman

January 9, 2009

2009 International Conference on Advances in Mathematics,
Computational Methods, and Reactor Physics
Saratoga Springs, NY, United States
May 3, 2009 through May 7, 2009

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Domain Decomposition of a Constructive Solid Geometry Monte Carlo Transport Code

Matthew O'Brien, mobrien@llnl.gov; Ken Joy, joy@cs.ucdavis.edu;
Spike Procassini, procassini1@llnl.gov; Greg Greenman, greenman1@llnl.gov
November 18, 2008
LLNL-CONF-409739

Abstract:

Domain decomposition has been implemented in a Constructive Solid Geometry (CSG) Monte Carlo neutron transport code. Previous methods to parallelize a CSG code relied entirely on *particle parallelism*; but in our approach we distribute the geometry as well as the particles across processors. This enables calculations whose geometric description is larger than what could fit in memory of a single processor, thus it must be distributed across processors. 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 parts.

Table of Contents

Table of Contents	1
Introduction	1
Constructive Solid Geometry	2
Domain Decomposition: Mesh vs. CSG	3
Domain Decomposition of CSG	6
What is Distributed Across Domains	6
Scalability Issues	7
Scalability Solutions	7
Algorithms: Calculating a Cell's Bounding Box	7
Algorithms: Cell Parsing	9
Algorithms: Locate Coordinate	10
Algorithms: Nearest Facet	10
LIFE Problem	11
Preliminary Results	12
Dynamic Load Balancing	14
Conclusions	15
References	15
Acknowledgement	15

Introduction

Previous methods of parallelizing a CSG Monte Carlo neutron transport code implemented a method known as *particle parallelism*, meaning that the geometry information was redundantly stored on *all* of the processors, while the particle workload

was 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 mesh. There are well known techniques for partitioning a mesh into domains, the contrast here is that we don’t have an underlying “mesh”, we only have CSG surfaces and cells.

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 and we end up with a *scalable* algorithm.

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. These surfaces are stored as a list of the coefficients in the implicit equation satisfied by the points on the surface:

$$f(x, y, z) = \sum_{\substack{0 \leq i+j+k \leq 2 \\ i, j, k \geq 0}} a_{ijk} (x - x_0)^i (y - y_0)^j (z - z_0)^k = 0$$

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)
```

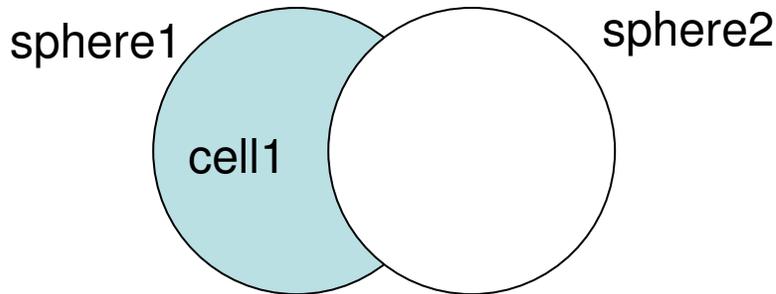


Figure 1: A simple example of creating a CSG cell that is inside of the sphere1 surface and outside of the sphere2 surface.

Using only these simple primitives, one can construct very complicated geometries. For example, in the pictures below, the NIF target chamber and support structures are modeled with CSG.

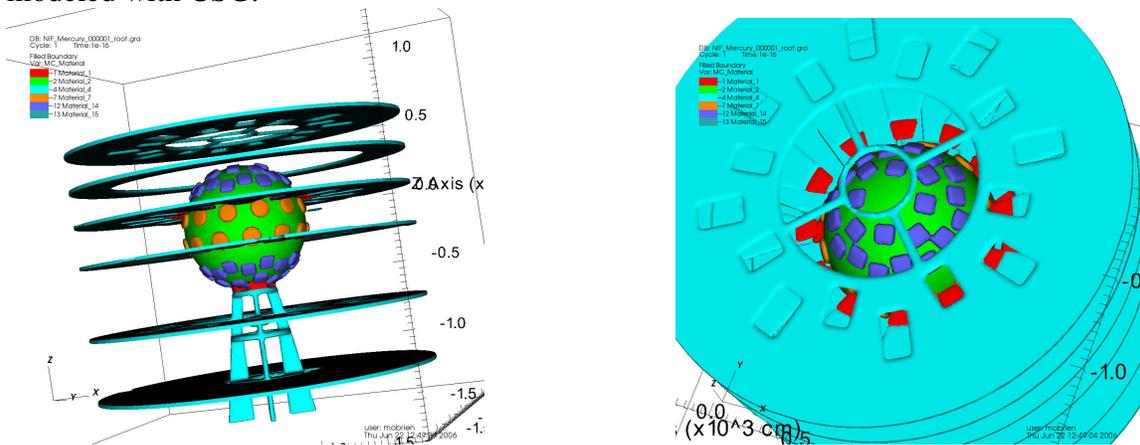


Figure 2: The NIF target chamber and support structures are modeled with CSG.

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 your geometry is *mesh* based, part of the description of the mesh is the *connectivity* of the of the mesh cells. If your mesh is topologically Cartesian, then you implicitly know the connectivity of the mesh by using indexing and striding to move in the *i*, *j* or *k* directions. If your mesh is unstructured, then you have a data structure that tells you the face neighbors of every face of every zone. This creates an underlying graph that is partitioned into domains.

Let $G=(V,E)$ where

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

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

In the case that the underlying discretization of the problem geometry is CSG based, then we do *not* have any connectivity information about the adjacency of any of the CSG cells. As a particle exits a bounding surface of one cell and enters an adjacent cell, (a priori) it does not know what cell it will enter. The algorithm dynamically *learns* the connectivity of the mesh as particles track through the mesh. 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, so there is no underlying cell-face-neighbor adjacency graph, 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. So the test for membership of a cell within a domain is a simple axis-aligned box-box intersection test.

	Mesh	CSG
Cell Boundary Crossing	Adjacent cells know.	Must check adjacent candidate cells, don't 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)

Table 1: This table compares the information at hand and underlying algorithms for mesh vs. CSG based domain decomposed particle tracking.

Example of CSG domain decomposition:

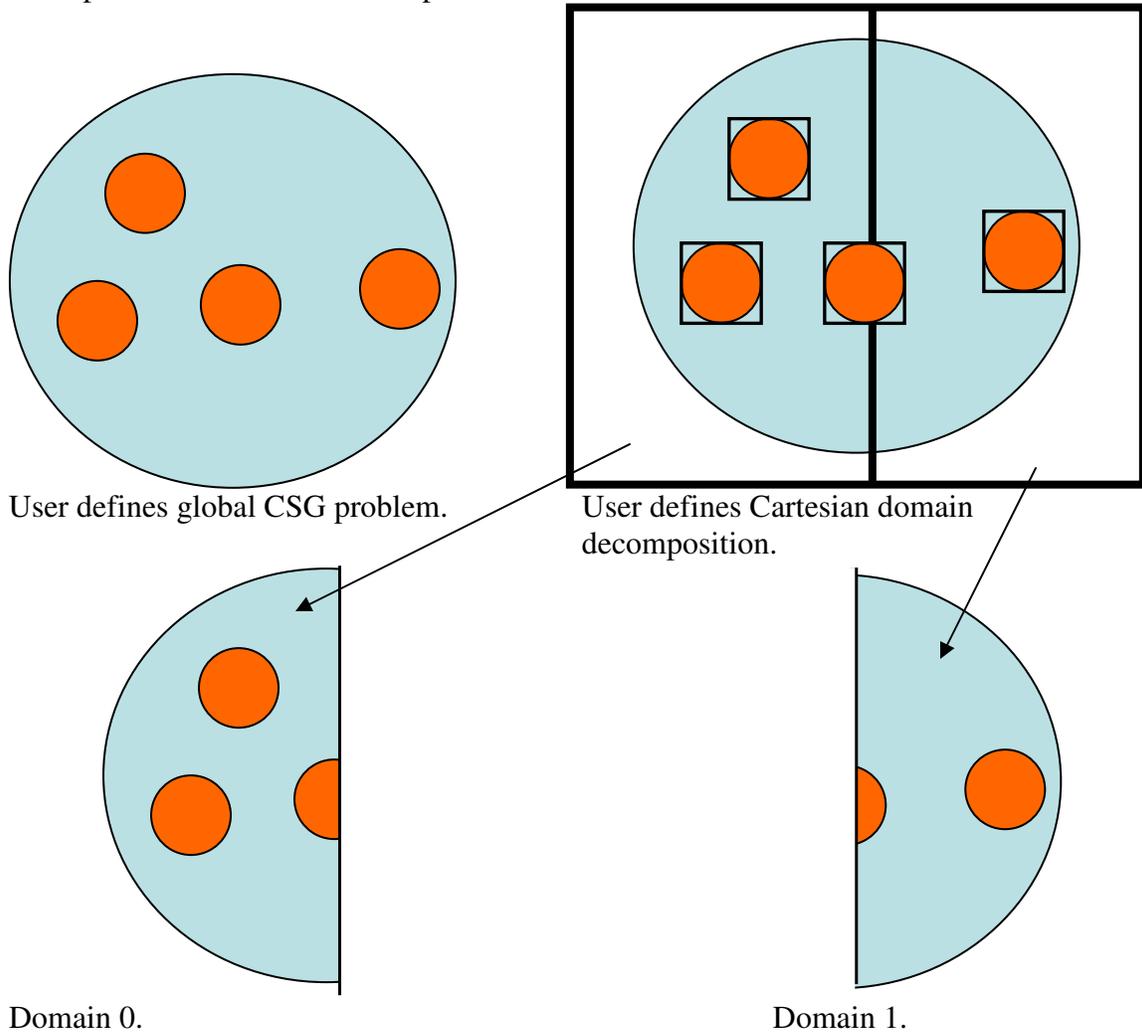


Figure 3:

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.

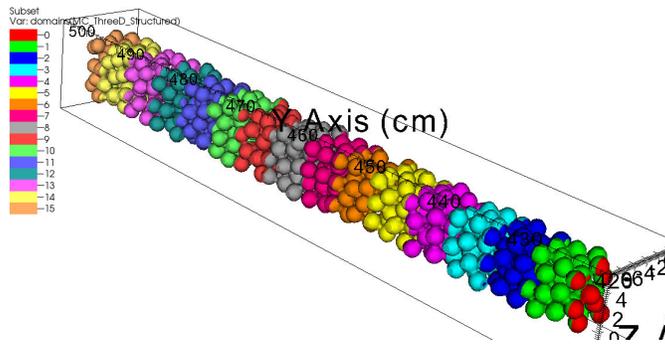


Figure 4: This problem has 16 domains, the CSG cells are colored by domain number.

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.

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. So we need a way to distribute this data across processors:

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

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.

The code 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, you 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.

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. Each processor never knows about the *global* description of the geometry. That 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.

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 you 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:

$$(1,000 \text{ houses/city}) * (2,500 \text{ cells/house}) * (7\text{K/cell}) = 17.5\text{GB/city}.$$

17.5GB is more memory than could fit on any single processor, but because of 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 scalability using input templates.

Algorithms: Calculating a Cell's Bounding Box

We need to calculate a bounding box for both CSG surfaces and cells. Our surfaces are just quadric surfaces, specified by coefficients a_{ijk} , such that $i, j, k \geq 0$ and $0 \leq i, j, k \leq 2$; 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 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 particulate, 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 so unbounded surfaces will be assigned to *all* processors.

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

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

So the axis aligned bounding box is given by:

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

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

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

CSG cells are built up from surfaces and we calculate cell bounding boxes from 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. Here 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
    }
}
}
}

```

We require all cells to be bounded, so `not (cell)` is unbounded, thus we return an infinite bounding box for that case.

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 onto its list of cells, if the cell's bounding box intersects with the domain's bounding box. This means that for cell's that straddle domain boundaries, they will be inserted into multiple domains.

```

foreach (input file cell)
{
    foreach (domain on this processor)
    {

```

```

temp_cell = inputFile.ParseCell(input file cell)
CalculateBoundingBox(temp_cell)
bool on_domain = domain.IsCellOnDomain(temp_cell)
if ( on_domain )
{
    domain.InsertCell(temp_cell)
}
}
}

```

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 of* the domain in question, that domain can immediately reject ownership of the particle. If the point in question is *inside of* the domain in question, then proceed with the existing algorithm.

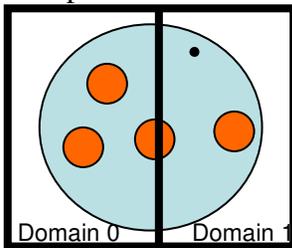


Figure 5: The bold black lines are domain boundaries. The black dot illustrates the position of a particle. The particle is outside of Domain 0, so Domain 0 can immediately reject ownership of the particle. The particle is inside of Domain 1, so Domain 1 must proceed as usual to test to see which cell the particle is in.

- During the *Is-Point-In-Cell* routine, if there is more than 1 domain, then the algorithm ensures that the input particle is inside of the input domain.
- If that test passes, continue as before.
- Otherwise the particle is definitely not on the input domain.

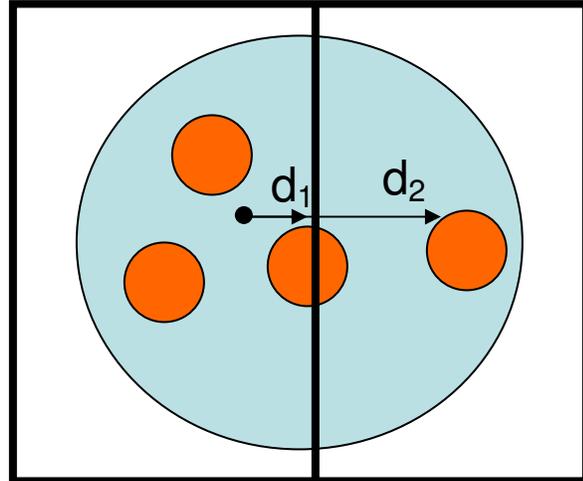
Algorithms: Nearest Facet

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 along through a CSG cell, it will eventually reach the current cell’s boundary and cross into the next cell. Given the particles position and velocity, the *Nearest Facet* algorithm will calculate the distance to all of the bounding surfaces of the cell, and it will select 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.

Our code already had domain decomposition for *mesh* problems, so we already have the infrastructure to buffer and communicate particles among adjacent domains. So after we determine that we are going to have a CSG domain boundary crossing, we use the existing infrastructure to communicate a particle from its current domain to the adjacent domain.



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 6: 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.

LIFE Problem

Lawrence Livermore National Laboratory is in the final stages of building the National Ignition Facility (NIF), 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) 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. It is a high tech, "fusion/fission" nuclear reactor.

To do 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° by 1° solid angle), requires 5.6 million CSG cells. To model the full four-Pi geometry would take billions of CSG cells.

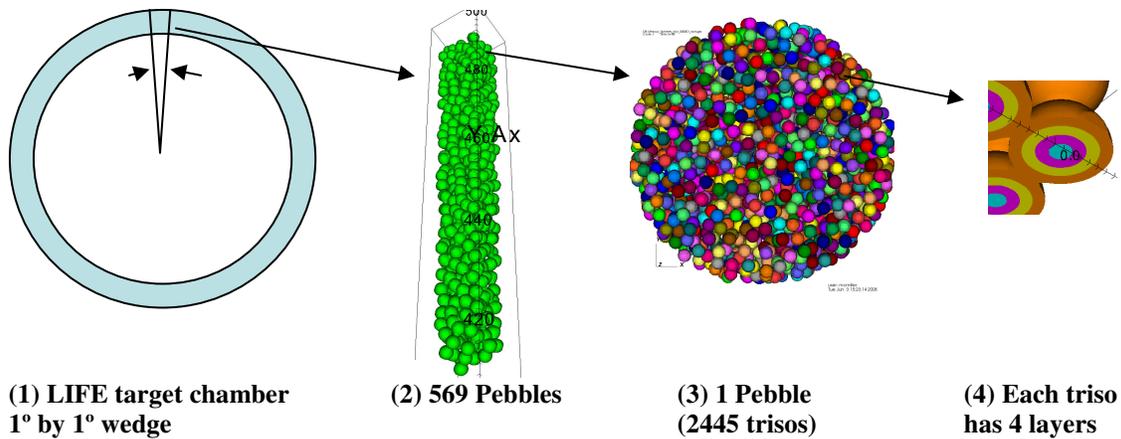


Figure 7: This shows the LIFE target chamber and burnable fuel.

- (1) The LIFE target chamber, the inner radius is 423 cm, the outer radius is 504 cm.
- (2) This is a 1° by 1° wedge of pebbles, which contains 569 pebbles, each pebble has a 1cm radius. The material in between the pebbles is Flibe Coolant, made of Li, Be and F.
- (3) 1 Pebble has a 1cm radius and contains 2445 Triso pellets. The material in between the trisos is Pebble Filler, Carbon.
- (4) Each triso pellet has a radius of 497 μm and has 4 layers. Layer 1: U²³⁸, O, C. Layer 2 and 3: C. Layer 4: C, Si.

The total CSG cell count in the LIFE problem is

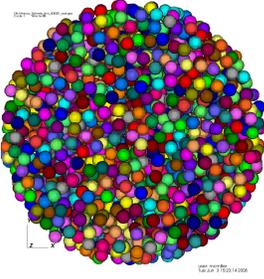
$$569 \text{ pebbles} * 2445 \text{ trisos} * 4 \text{ layers} = 5.6 \text{ Million CSG cells.}$$

We model the neutron scalar flux distribution as binned energy group data, with 175 energy groups, so 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 particles. This is more memory than any one processor has, so we must distribute the problem across processors if we ever hope to solve it. We are in a unique position to solve extremely large scale, detailed problems like this.

Preliminary Results

In this test, we transport particles though 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).

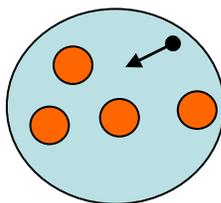
Figure 8: Pebble with 2445 CSG cells, homogenized trisos



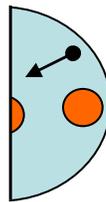
Seconds Spent Doing Particle Transport.					
	1 proc	2 procs	4 procs	8 procs	16 procs
1 domain	848	427	226	131	74
2 domains	736	235	148	82	52
4 domains	668	190	65	34	20
8 domains	659	162	57	20	12
16 domains	686	214	113	32	12
64 domains	732	207	116	37	18

Table 2: This shows the time in seconds spent doing particle transport, under various domain and processor configurations.

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.



(1) Without domain decomposition.



(2) With domain decomposition.

Figure 9:

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

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

When you add more domains to a problem, it localizes the geometry, but there is a competing effect of calculating the distance to the new domain boundaries you are introducing. For example, on one processor, when we go from 16 domains to 64 domains, the 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.

Now let's examine the column of data for 16 processors. We vary the number of domains that the problem is run on. When run on 1 domain, this 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. 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 again is due to localization of geometry. When a particle is inside of 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).

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. When you have 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 $64\text{processors}/16\text{domains} = 4$ 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.

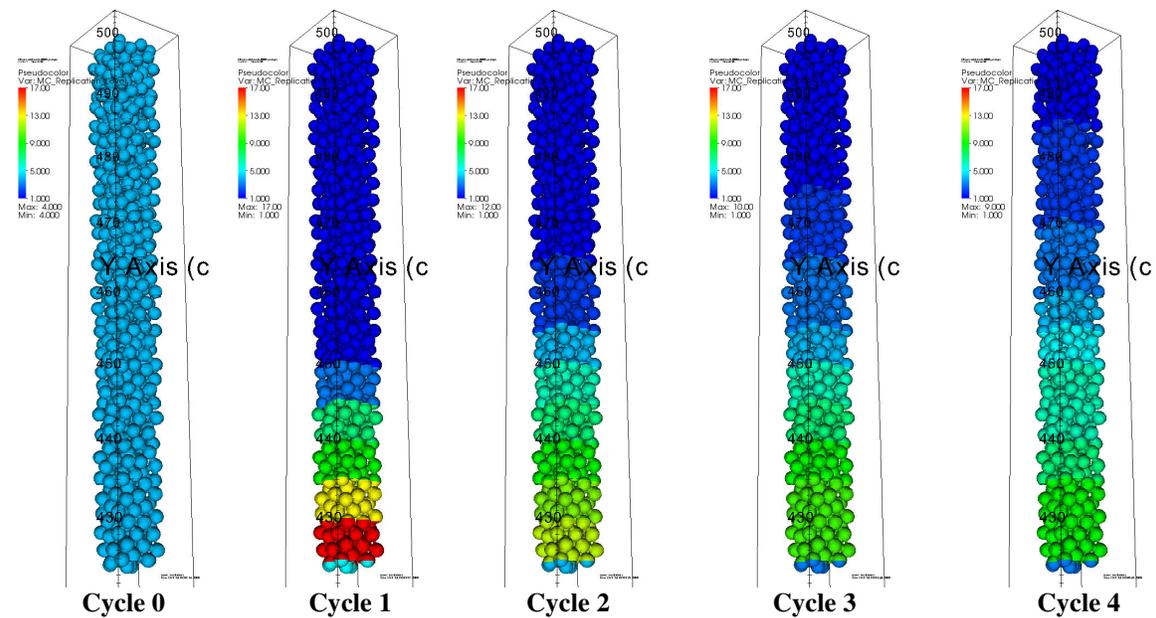


Figure 10:

- 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.
- You can processors "transport" with the particles.

Conclusions

We have implemented a domain decomposition algorithm in a constructive solid geometry Monte Carlo transport code, which 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 problem 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.

We were able to run a 5.6 million cell CSG simulation of the LIFE engine and will continue to use the domain decomposition feature for future LIFE calculations.

References

Mercury Web Page, www.llnl.gov/mercury. Lawrence Livermore National Laboratory.

VisIt Web Page, www.llnl.gov/visit. Lawrence Livermore National Laboratory.

R. Procassini, et al. *MERCURY User Guide*. Lawrence Livermore National Laboratory. UCRL-TM-204296. August 21, 2008.

M. O'Brien, G. Greenman and R. Procassini, *Domain Decomposition of a Combinatorial Geometry Monte Carlo Transport Code*, NECDC 2008, Livermore CA, October 20-24, 2008, (LLNL-PRES-407916).

Acknowledgement

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