Geometric Acceleration Structures to Speed Up Cell Searches in OpenMC

Saad Amin[1], Gavin Ridley [2,*], Benoit Forget[2]

[1]Mission San Jose High School, Fremont, California;
[2]Massachusetts Institute of Technology, Cambridge, Massachusetts

doi.org/10.13182/PHYSOR24-43886

ABSTRACT

Initial cell searches in Monte Carlo calculations can be a computationally expensive task for problems with many cells not arranged in a lattice or hierarchical set of universes. For example, an Advanced Test Reactor (ATR) model used in this work has 5,968 cells. Determining the cell occupied by fission source and secondary particles took 38% of program wall clock time before using our new algorithms. Each of our new methods reduced this initial cell search to less than 1% of wall clock time with negligible time taken to traverse the acceleration structure. In a problem exhibiting a large number of secondary particles or coupled photon transport, even larger speedups could be expected.

Keywords: OpenMC, Octrees, KD-Trees, Acceleration Structures, Geometry

1. INTRODUCTION

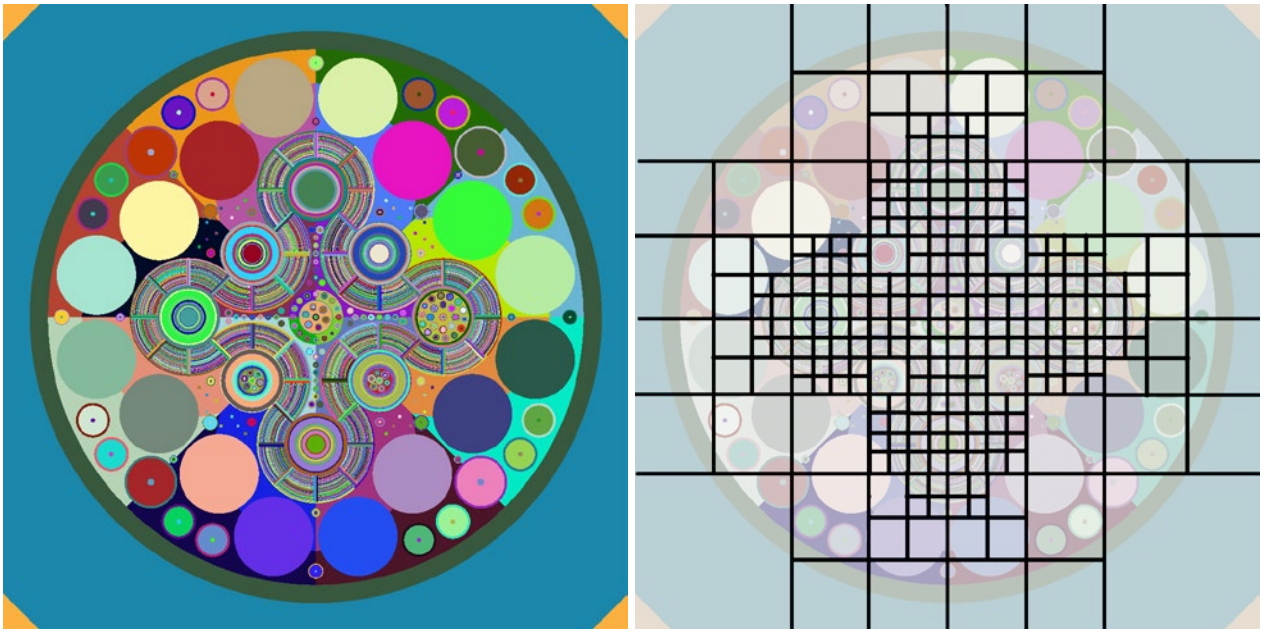
In Monte Carlo particle transport calculations, the cell search for source and secondary particles requires an exhaustive search in the absence of acceleration structures. By exhaustive search, we mean the process of finding the cell which contains a given point by evaluating the constructive solid geometry expressions for each cell until the first which satisfies the criteria is found. In contrast, the majority of cell searches in a Monte Carlo transport code use neighbor lists [1] because particles contiguously traverse from one cell to another. Hence, only neighboring cells need to be searched most of the time, and OpenMC maintains a shortlist of these neighbors to check.

On the other hand, the fission sites saved in the method of successive generations [2] do not have cached information about the nested cells of a universe-based [3] geometry formalism. Caching the full geometry state of each fission site would be undesirable due to the extensive communication of fission sites between parallel processes [4]. Secondary particles and fixed source sites are treated in a similar manner, and the computational demand of these initial searches consequently can vary tremendously between problems.

At the start of this work, OpenMC allowed only one ad-hoc method of partitioning the cells to search in a problem. In fact, simply rotating a geometry would cause the previous technique to fail [1]. OpenMC divides the exhaustive cell search between any z-axis-aligned planes that separate the cells. This aspect of OpenMC clearly should be improved.

*ridley@mit.edu

[1]To clarify, we mean by changing z-planes to y-planes and so on. A transformation matrix would not effect this failure.



(a) Coloring by cell ID **(b) Octree overlay bounding cell count per leaf**

Figure 1. OpenMC CSG model of ATR

In Serpent, the cell search list may be sorted by the most likely cell first [5]. We are unaware of the techniques MCNP employs for finding the cell a particle initially occupies, and in fact have found this topic to be infrequently described in open literature. For most reactor geometries where the large number of CSG cells tend to occupy nested hierarchical universes consisting of a small number of cells in each, advanced geometric acceleration structures offer little benefit. In this work, we developed a few robust exhaustive cell search acceleration strategies for an example problem representative of a large number of cells in a single CSG universe: the advanced test reactor (ATR). Fig. 1a shows a slice plot of the model.

2. BUILDING AND TRAVERSING THE PARTITIONERS

Traversing a geometric acceleration structure is solved problem described in myriad pieces of computer science literature. Efficient construction in the context of a Monte Carlo transport code, however, is more nuanced. Construction of hierarchical partitioner structures requires us to know what objects are within each node. If the objects satisfy a certain requirement (e.g. there are too many objects within a node), then we subdivide the node. It is problematic to do this within OpenMC, which uses constructive solid geometry instead of primitives (e.g. triangles or other simple shapes) to represent cells. Unlike with primitives, it is nontrivial to determine if a shape constructed with CSG is inside a node's bounding box. Thus, the well-established methods in computer graphics for constructing octrees, K-D trees, and BVHs from a set of primitives [6] no longer apply.

As an initial bootstrapping step to start our acceleration structure, we first sample a point cloud. From there, secondary sites which do not result in a successful search can be appended to the structure on-the-fly, similar to our approach for neighbor lists [1]. With each of the initially sampled locations, we carry out a preliminary brute-force geometry search. To determine what cells are in a specified bounding box, we can sample points inside the bounding box and return a list of associated cell IDs. This makes construction far easier, as we can use the point cloud instead of the cells themselves.

2.1. Binned Point Search

With the point cloud approach, we might miss small lengthscale details in a large CSG model. As a result, the final partitioner might not find the cell a specified point is located in. In the case that this occurs, we have to fall back to an expensive brute force search or z-plane partitioner. Since this type of cell search is slow, we need a higher sampling resolution. However, a higher sampling resolution means more exhaustive searches during construction and more memory to store the point cloud.

We devised an algorithm we call binned point search to learn over time where complex, smaller geometry is located and concentrate more samples there, as well as provide a cell cache to avoid slow brute force searches. Binned point search splits the partitioner bounding box into a grid, with each ”box” on the grid being referred to as a bin. Each bin contains a list of unique cells found so far within it, known as the cell cache. Our algorithm is an loose adaptation of [7] to geometry sampling.

Binned point search begins by picking a bin randomly, and then randomly and uniformly picking a point inside that bin. The probability of picking a bin is proportional to its score, which is the maximum of the size of the cell cache and the number 1. At the beginning of the algorithm, there are no cells in the cell cache, so each bin as an equal probability of being picked. After sampling a fixed number of points, we update the probability of sampling all bins according to the new cell cache sizes. We refer to this process of sampling points and then updating the probabilities as an iteration. Algorithm 1 presents the specifics of this technique.

Because the bin location is sampled on-the-fly, we do not update the probabilities after each cell's completion. Rather, the probability update is carried out after a batch of cell searches. We create an array that stores each bin's score. We then apply a prefix sum and divide each element by the sum of all elements in the array. Then, to sample a bin, we generate a random number in [0, 1) and use binary search to find which bin's CMF value is the least value that is greater than our random number. It is difficult to update the CMF array during sampling efficiently, so we do it after sampling.

When we randomly sample a point inside a bin, we first check if the point is located in any of the cells in the cell cache. If we don't find it there, we then do a brute force search over all cells then append the correct cell ID to the cache. This is a more minor detail, and the paper is too long. It can be gleaned from the pseudocode. Unlike the CMF array, we update the cell cache right away, so other points during the same sampling iteration can take advantage of the new knowledge.

2.2. Constructing the Partitioner from a Point Cloud

We can utilize the bin grid directly as a partitioner. We found that this partitioner type is relatively quick to construct and provides similar results to octrees and KD-trees. Nonetheless, for larger models we may want a more efficient partitioning approach. The construction for octrees and KD-trees is largely the same. We start at the root and count how many unique cells would be in each child node. Then, if the number of unique cells is greater than a threshold in at least one of the children nodes or if the sum of the volume heuristic in the children nodes is smaller than the volume heuristic of the parent node, we subdivide. The volume heuristic of a node is equal to the number of unique cells in the node multiplied by the probability of us traversing this node. Since we do not have the exact probability of coming across the node during traversal, we use the volume as a stand-in value that is proportional to the actual probability. To prevent issues with infinite subdivision, we multiply the volume heuristic of the children nodes by a factor to not favor sudivision at deeper depths. If we do choose to subdivide, we add the children nodes to a stack. We then pull a child node off this stack and attempt to continue the process of subdivision. We repeat this process until the stack is empty.

Since the construction process has to process millions of points multiple times in order to determine what node a particular point falls into, it is a memory bandwidth intensive process. Since each point requires 3 double precision floating point variables to store position and a 16 bit integer to store the cell ID, we can easily reach a memory bottleneck. To alleviate this bottleneck, we need to reduce the storage size of each point. In the case of octrees, we can observe that we don't need to store the position. Since we know the octree bounds ahead of time, we know which child node the point will fall into at each depth. Since we only need 3 bits to encode the child node index, we can instead store a single 64-bit integer with 16 bits dedicated to the cell ID and the remaining 48 bits dedicated to child node indices that can be used for up to 16 levels of depth. The exact mapping from child node index to child node will be discussed later in the traversal section. Unfortunately, in the case of KD-trees, we cannot apply a similar optimization because we are unaware of the split planes ahead of time.

To efficiently determine the number of unique cells in a node, we iterate over all points, storing the cell IDs in the corresponding child node. We then sort the list of cell IDs in each child node and count the number of different consecutive elements to count the number of unique elements.

2.3. Random Refinement

If we utilize our octree directly, we often will find that it is unable to find many points. We use random refinement to refill many of those lost points. Random refinement takes heavy inspiration from the binning methodology of the binned point search. A quick summary of random refinement is that it is a stochastic algorithm that samples points in the octree and checks if we found the point. If we did not, we brute force search the point and then add it to the octree.

Random refinement makes use of probability bins, which contain a list of leaves and a list of all unique cell IDs found in those leaves. Like binned point search, probability bins comprise a grid that encases the model. We pick a random probability bin according to a probability that is proportional to the number of leaves conatined in that probabilty bin. Once we pick a bin, we randomly pick a leaf inside that bin with equal probability. Inside that leaf, we randomly pick a point and if we don't find the cell it is inside of the leaf, we check the probability bin's cell cache and add it to the leaf. If we don't find it there, we do a brute force search and add the cell to the probability bin and the leaf. Algorithm 2 lays out the details.

2.4. Information Refilling

Although random refinement does find a lot of missing points, it still leaves out a significant chunk of them. We hypothesize that a major reason why octrees are unable to find points is because subdivision causes points to be lost across subdivision boundaries, thus causing many unique cell IDs to be lost. To circumvent this problem, adding the list of cell IDs from surrounding nodes to a particular leaf node should help, and we call this process information refilling. By including surrounding cell IDs into a node, we essentially refill any cell IDs that were lost during node subdivsion back to a node. Information refilling does go against the original idea of partitioning by including extra cells we do not need in each node. To remove this added cost, we append the refilled cell IDs to the end of our cell ID array. This way, OpenMC first checks the subarray of nonrefilled cell IDs before checking the refilled cells. This is far cheaper than resorting to a brute force fallback or the z-plane partitioner.

In our implementation of information refilling, for every leaf node, we go up a few depths and reach the collection node. Then, we traverse the octree down from the collection node, storing all cell IDs we find in leaves in a a list. We then append this list to our leaf node. We store this appended list separatly to ensure that information refilling does not use the results of previously information refilled nodes and leads to an extreme spike in memory usage. However, information refilling is still very expensive, as we need to do a

complete traversal of a subtree for every single child node. Instead, we separate the process into two distinct stages: forward propagation and down propagation.

2.4.1. Forward Propagation

In forward propagation, we want to create a cache of collected cell IDs for each parent node. This way, the down propagation step does not have to retraverse the tree every time we want to refill a node's cell ID list. Although we can create a collected cell ID cache during subdivsion, we would lose out on information regained via random refinement. Thus, we chose to rebuild these caches.

We first create an array of all nodes sorted by their depths such that deeper nodes come first. For each node, we store a pointer to its parent (or nullptr if it is the root). For each node with a parent, we append the child node's cell IDs to the parent. Once we iterate over to the parent, we take the combined cell ID array and sort it. We then store elements that are only different from the previous element, this making the array consisting of unique elements only. We then repeat the process of writing cell IDs to the parent.

2.4.2. Down Propagation

In down propagation, we take advantage of the cell ID caches built in forward propagation to refill information in leaf node. We traverse the tree, searching for leaf nodes. If we come across a leaf node, we traverse up a fixed number of steps to the collection node. We then use the cache built during forward propagation for information refilling. We make sure not to duplicate cells that already exist in the leaf node's array. Algorithm 3 details the combined method to propagate information in octrees to higher levels of detail.

3. TRAVERSAL

In this section, we will discuss how we traverse the octrees, KD-trees, and bin grid partitioners. We also discuss how we optimized the memory representation of nodes for better cache efficiency.

3.1. Traversal Algorithms

To efficiently traverse octrees, we use a special layout to map child node indices to child nodes. The child node a particular point goes into is determined by its position relative to the center of the parent node. The x, y, and z components of the point can be greater or less than the x, y, and z components of the center position respectively. These three boolean values taken together form a 3-bit integer, with each specific combination mapping to a different child node. We can use this 3-bit integer as the child index mapping. We can also use this 3-bit integer to update the node center position if we know the bounds of the parent node, which we can determine via the octree bounds and current depth.

To efficiently traverse KD-trees, we use a standard traversal algorithm that compares against the split axis. To traverse bin grid partitioners, we simply determine the bin a point maps to with integer division.

3.2. Node Compression

Since traversal is a task that often suffers from accessing memory addresses that are far from each other, it is important to reduce the size of each indivual node to improve cache efficiency. For octrees, we store all nodes in a contigous array. We can store all cell ID vectors of the leaves in a seperate array. We can then pack each node as a 32 bit integer. The sign bit is dedicated to determining if the node is a leaf or node. If it is a leaf, the remaining 31 bits correspond to an index in the cell ID vector array. Otherwise, it corresponds to the first child's index in the node array.

Algorithm 1 Binned point search

```
1:  struct Point {
2:      Position pos
3:      int cell_id
4:  }
5:  struct Bin {
6:      double relative_probability
7:      vector<int> cell_cache
8:  }
9:  procedure BINNED_POINT_SEARCH(universe)
10:     Bin grid[WIDTH][HEIGHT][DEPTH]
11:     vector<Point> points_arr
12:     for cycles = 0; cycles<max_cycles; cycles++ do
13:         for bin in grid do
14:             bin.relative_probability = max(1, bin.cell_cache.size)
15:         end for
16:         for points = 0; points<max_points_per_cycle; points++ do
17:             Bin bin = pick uniform random bin from grid
18:             Position pos = bin.sample_uniform_point()
19:             Cell cell = search bin.cell_cache for cell at pos
20:             if cell not found then
21:                 cell = universe.exhaustive_search(point)
22:                 bin.cell_cache.push_back(cell)
23:             end if
24:             points_arr.push_back(pos, cell)
25:         end for
26:     end for
27: end procedure
```

Algorithm 2 Random octree refinement

```
struct OctreeLeaf {
    Box bounds
    vector<Cell> cells
}
struct ProbabilityBin {
    double relative_probability
    vector<OctreeLeaf> contained_nodes
    set<Cell> cell_cache
}
procedure RANDOM_REFINEMENT(universe, octree)
    ProbabilityBin grid[WIDTH][HEIGHT][DEPTH]
    for leaf in octree do
        Position center = leaf.box.center()
        ProbabilityBin bin = bin that contains center
        bin.contained_nodes.push_back(leaf)
        bin.cell_cache.extend(leaf.cells)
    end for
    for bin in grid do
        bin.relative_probability = bin.contained_nodes.size()
    end for
    while time spent in refinement not over allotted time do
        ProbabilityBin bin = pick uniform random bin from grid
        OctreeLeaf = pick uniform random leaf in bin
        Position pos = pick uniform random position in leaf.box
        cell = search leaf.cells for cell located at pos
        if cell not found then
            cell = search bin.cell_cache for cell located at pos
            if cell not found then
                cell = universe.exhaustive_search(pos)
                bin.cell_cache.push_back(cell)
            end if
            leaf.cells.push_back(cell)
        end if
    end while
    deallocate(grid)
end procedure
```

Algorithm 3 Octree propagation

```
1:  struct OctreeNode {
2:      vector<Cell> cells                          ▷ initially empty for all nonleaf nodes
3:      OctreeNode* parent                          ▷ nullptr for root
4:  }
5:  procedure INFORMATION_REFILLING(OctreeNode* octree)
6:      forward_propagation(octree)
7:      down_propagation(octree)
8:  end procedure
9:
10: procedure FORWARD_PROPAGATION(OctreeNode* octree)
11:     vector<OctreeNode*> nodes = get list of nodes in octree
12:     sort nodes by depth
13:     for node in nodes do
14:         if node is not leaf then
15:             make all elements in node.cells unique
16:         end if
17:         if node is not root then
18:             node.parent.cells.extend(node.cells)
19:         end if
20:     end for
21: end procedure
22:
23: procedure DOWN_PROPAGATION(OctreeNode* octree)
24:     for leaf in octree do
25:         OctreeNode* collection_point = leaf
26:         int depth_increase = 0
27:         while (depth_increase < max depth increase) and (collection_point ≠ root) do
28:             collection_point = collection_point.parent
29:             depth_increase++
30:         end while
31:         for cell in collection_point.cells do
32:             if leaf.cells does not contain cell then
33:                 leaf.cells.push_back(cell)
34:             end if
35:         end for
36:     end for
37: end procedure
```

	Octree	K-D tree	Bin grid	Z-plane
Setup time (s)	21.3	18.6	6.1	0.002
Calculation rate (p/s)	85,900	85,600	84,500	63,400
% of time in find_cell	0.94	1.88	2.81	38.0

Table I. Performance of the different methods on the ATR model

For KD-trees, we use a similair strategy. In the case of KD-trees, we need to store a the split axis, split position, and first child index in the case of a parent, and cell vector index in the case of a leaf. We can store a single 32 bit integer and a 32 bit float for this purpose. The 2 most significant bits of the integer correspond to a combined split-axis and leaf marker. If the value of the top 2 bits are 0, 1, or 2, they correspond to the x, y, and z axis respectively. If the value is 3, then the node is a leaf. In the case of a parent, the remaining 30 bits correspond to the first child index, where as in the case of a lead they correspond to the cell vector index. The parent also uses the floating point variable to store the split location. The child does not use this variable.

4. RESULTS

Table I shows the performance of each of our implemented methods. Any acceleration technique beyond z-plane partitioning tends to effectively reduce the time spent in the initial cell finding subroutine. Among the new methods, the bin grid technique requires the least setup time, but takes triply long to complete initial cell finding compared to an octree. Of course, the performance can vary tremendously depending on the choice of parameters for the acceleration structure, for instance the maximum number of cells per leaf in the case of the octree, or the number of initial sample points. The K-D tree requires slightly less setup time than an octree, but takes about twice as long to run initial cell finding calls.

5. CONCLUSIONS

We have presented three universe partitioning techniques applicable to Monte Carlo particle transport simulations: octrees, K-D trees, and Cartesian binning. The former two techniques are not employed in mainstream Monte Carlo codes, to our knowledge, and have been clearly demonstrated to enhance simulation performance on a CSG model of the ATR.

There are a plethora of future directions to take this research. Firstly, we expect these methods can accelerate simulations based on CAD geometry, and hope to also investigate performance there. The new techniques could also be modified to operate in a thread-safe, on-the-fly fashion, obviating the need for an initial setup step. The different methods can also be combined, for example octrees that fill Cartesian bins or KD-tree partitioning of octree leaves. Due to the large number of user-tunable parameters for the acceleration structures, methods for automatically determining an efficient parameter set based on model characteristics should also be investigated at some point in the future. The new methods should also greatly enhance problems with a large number of secondary particles, so we hope to test performance on a problem with coupled photon transport in the future.

ACKNOWLEDGEMENTS

The second author was funded by a DOE-NE Integrated University Program Graduate Fellowship during this work. We thank Patrick Shriwise and Paul Romano for their patience in code reviews. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not

necessarily reflect the views of the Department of Energy Office of Nuclear Energy.

REFERENCES

[1] S. Harper, P. Romano, B. Forget, and K. Smith. "Efficient Dynamic Threadsafe Neighbor Lists for Monte Carlo Ray Tracing." In *M&C 2019 Topical Meeting*. Portland, OR (2019).

[2] J. Lieberoth. "MONTE CARLO TECHNIQUE TO SOLVE. THE STATIC EIGENVALUE PROBLEM OF THE BOLTZMANN TRANSPORT EQUATION." *Nukleonik, 11: 213-19(Sept 1968)* (1968).

[3] J. A. Kulesza et al. "MCNP® Code Version 6.3.0 Theory & User Manual." Technical Report LA-UR-22-30006, Los Alamos National Lab. (LANL), Los Alamos, NM (United States) (2022).

[4] P. K. Romano and B. Forget. "Parallel Fission Bank Algorithms in Monte Carlo Criticality Calculations." *Nuclear Science and Engineering*, **volume 170**(2), pp. 125–135 (2012).

[5] J. Leppänen. "On the Performance of the Particle Tracking Routine in Serpent 2." (2022).

[6] M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering: From Theory to Implementation (3rd ed.)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition (2016).

[7] H. W. Jensen. "Importance Driven Path Tracing using the Photon Map." In P. M. Hanrahan and W. Purgathofer, editors, *Rendering Techniques '95*, pp. 326–335. Springer Vienna, Vienna (1995).