# Octrees & OpenMP

Doug Woodward

## 1 Introduction

Modeling objects in space and their gravitational interplay by solving the classic N-Body physics problem over a given time period presents a number of immediately apparent problems from a computational perspective. The core problem is simple, calculate the combined gravitational effect of some number of objects N, on a single body using the formula $Fg = -Gm_i \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{m_j}{r_{ji}^3}(r_{ji})$ where $G$ is the gravitational constant,$n$ is the number of bodies in the system, $m_i$ is the mass of the body in question, and $r_{ij}$ is the difference of the position vectors $r_j - ri$. For some small system composed of only a few objects, it is straightforward to make the calculations in $O(n^2)$ time. However, space is massive, in fact, it is limitless as far as we know. And within that infinite space, the number of bodies approaches infinity. Given this, solving the N-Body problem for a given object at a given point in time means $\infty$ calculations for the entire universe. To avoid this, the area of space in question can be confined - for example to just our Solar system. In this context, the gravitational effects of objects outside the solar system are considered negligible. This makes the problem much more manageable, but with the amount of object in the solar system being roughly 735,830 [1] the computational effort of $O(n^2)$ on that many objects is unfeasible, especially over time. One solution to this is the Barnes-Hut simulation, which relies on dividing the space into octrees, and for each object in the tree, traverse the tree, and if a node is sufficiently far away, that entire section of the tree and therefore the space is combined into one object with a the position set as the combined center of mass.

The aim of this work is the develop an octree algorithm from scratch, first in serial, and then apply OpenMP to parallelize the creation and traversal of the tree.

## 2 Octree Implementation

An Octree is the 3-dimensional equivalent of a binary tree for single dimension space and a quadtree for 2-dimensional space. Each node is either a leaf node, or an internal node. A leaf has no children. An internal node has eight children, octants, each of which can be a leaf or an octree itself. Only leaves contain actual points in this implementation. To build the octree, the required parameters are a maximum allowed depth, cubic space size, and a list of points in the three dimensional space. It is important to note that octrees scale to a possible size of $8^m$ nodes in the tree where $m$ is the maximum possible depth. This implementation does not guarantee balanced octrees because if there is only one point left to place in the node, it will become a leaf node. Therefore, for a $m = 7$, a maximum of 2,097,152 nodes can be created, however, the number of total nodes will kept much lower than this on average by the number of points to place. For example, the simulation run with $m = 7, numPoints = 100,000$ yields just over 368,000 octree nodes on average. So it can be noted that the number of nodes in the octree will to be limited by the number of points when the number of points is less than $8^m$.

The implementation of the octree algorithm used here means that each Node in the tree is an object of the Octree class. Each Octree has a vector of it's children, which is either filled with 8 children, or empty if it is a leaf node, a pointer to it's parent octree, which is null for the root node, and a list of NodePoint structs (x,y,z coordinates) that represent bodies. In addition, the Octree has upper and lower x,y,z bounds which fall within the parent node bounds such that the lower down the tree a node is the smaller the area contained within it.

During tree construction, a list of randomly generated points is created, then the root node is created and all NodePoints passed into it. From here, the recursive tree building begins using the Octree::Split method:

- If there is only one NodePoint left or Maximum Depth is reached:
    1. Place all remaining NodePoints in this Octree
- Else:
    1. Create the 8 Children Octrees for this Octree
    2. Iterate over them and assign each one the NodePoints that fall within its bounds
    3. Recursively call the Split method on each Child

With a completed Octree, traversals can now be conducted. The Traversal algorithm is analogous to that of a Binary tree, with the only difference being the number of children for each internal node. It takes in an Octree and will traverse the tree downwards from that node.

Starting from the root node:

- Iterate through each child node, call the Traverse method on each
- Process the current node

For our current purposes, the Process function does essentially nothing as the aim of this work is to establish the gains from using OpenMP and as such, the process taken is not relevant in the context. A future implementation of the Barnes-Hut algorithm would see this step comprised of calculating the $Fg$ on this node.

## 3 Parallelization

### 3.1 Child Creation

Following the implementation of the Octree algorithm, different sections of the code were identified as potential targets for parallelization. The first section addressed was the child creation section of the code. This section was selected for the lack of dependence on any globally scoped variables, pointers, or other potential complications. The OpenMP section construct was employed here - an ideal use case of the OpenMP sections due to the ability to construct each node in a fully independent section from the others. However, two issues occurred here. The first stemmed from the scoping of OpenMP variables. Due to the need to first construct the centerpoint of each node, then construct the Octree with that center node, and then insert the Octree into a vector only the centerpoint construction was successfully parallelized. OpenMP has variable scoping rules such that a variable declared inside a section cannot be accessed outside of it. This meant the nodes could not be constructed and then inserted after the parallel region. And because the each Octree must be accessible by the current node's children vector to be appended, these must be in the same scope as the list append - which is the critical region for this code block. An implementation explicitly built for OpenMP parallelization would be able to find a more elegant solution here, but that would not address the second, and more pertinent issue — this parallelization gives a speedup of negligible gain to the program. The reason for this is that the overhead required to create the team of threads, assign to sections, and then reconcile is of greater or equal cost to the gains realized by completing the task in parallel. This is a poor use of parallel processing as the computations performed are trivial and as such the overhead outweighs any gain.

## 3.2 Octree Creation

Next, a core component of the program was targeted for parallelization with OpenMP - the creation of the Octree. The code block is contained within the `Octree::split()` method. Within this method, a computationally heavy nested loop was targeted. A while loop nested within a for loop that is responsible for iterating through each NodePoint for each child and inserting them where the NodePoint falls in bounds of the child. Firstly, an attempt was made to employ the standard `#pragma omp parallel` directive. This unsurprisingly was met with a segmentation fault as the loop relies on pointers for insertion into the child Octree, among other issues. After more investigation into the cause of the failures, it was realized that despite being a computationally intensive for loop, this was also a memory critical section of the code. Because the NodePoint list is not only shared during iterations of the children, but also modified, a NodePoint is removed from the list when placed into an Octree, and the two iterator objects are modified during iterations, this section of code can not be parallelized using OpenMP. During attempts to parallelize this section, it was realized that the C++ STL containers are not thread safe when it comes to simultaneous writes of the same object in the container despite being thread safe for reading the same object, or writing different objects. Additional enhancements to this work could center on the parallel construction of this tree — one potential approach would be to build the tree bottom up and reconcile the resulting forest into a single tree in a critical section.

## 3.3 Traversal

After the unsuccessful attempts at parallelization in other parts of the algorithm, the traversal of the tree was targeted for parallelization. From the insights gained about thread safety in STL containers during the Octree creation OpenMP attempt, traversal fit all the criteria for parallelization. The code referred to here is in the `octree_traverse` and `octree_traverse_p` methods (same algorithm, p is parallel) and the `main` function. Analysis of the tree traversal method from an OpenMP standpoint led to the use of OMP tasks. Because the algorithm here is recursive, it is important to open the parallel block in the `main` function, outside the recursive call, thereby spawning the team of threads, and then immediately calling the `#pragma omp single` directive, still before the call to `octree_traverse_p`. Then, within the function `octree_traverse_p`, `#pragma omp task` is called inside the for loop above the recursive call to `octree_traverse_p`. Doing so means that only one thread enters the first call to `octree_traverse_p` from `main`, creates all the tasks and then the team executes them. This means that there is only one traversal of each child, and handily, most modern computers have four hyper-threaded cores, allowing for 8 threads to match the 8 children. The last piece of the task parallelization to consider is the `#pragma omp taskwait` directive. It is important to understand that using tasks in OpenMP means that the master thread, thread(0), enters the function, and each time it encounters a `#pragma omp task` directive, it will assign a waiting thread to that task and continue until it reaches either a `#pragma omp taskwait` directive or the end of the parallel section. Initially, the `#pragma omp taskwait` directive was placed the recursive call to `octree_traverse_p` inside the for loop of that method. This resulted in no speedup from serial execution because the first thread was waiting for the task to be completed while before continuing on the next loop iteration — essentially traversing serially with some added overhead from OpenMP. After this issue was identified, the `#pragma omp taskwait` was moved out of this loop, out of the function and placed after the initial call from the `main` function. This meant that the master thread entered the recursion, created and queued up each task without waiting for any to finish, leaving the thread team to execute them as quickly as they could. A key feature of tasks should be highlighted here - they can be created dynamically, ideal for recursion, and can be queued up for the thread team to handle. The end result of this implementation can be found in the following table of results, with serial and parallel times aligned, (time measured in seconds). These results were calculated using a four core, hyper threaded machine, effectively 8 cores.

| Serial | Parallel | Speedup | Efficiency |
|---|---|---|---|
| 0.01239 | 0.006024 | 2.06 | 0.26 |
| 0.012389 | 0.005649 | 2.19 | 0.27 |
| 0.012676 | 0.006117 | 2.05 | 0.26 |
| 0.012174 | 0.007373 | 1.6511 | 0.206 |
| 0.012163 | 0.0069 | 1.77 | 0.22 |

Amdahl's law was not employed here because the only section that was parallelized and measured was the traversal - in which either all of it or none of it was run in parallel, and the use of recursion renders it difficult to break down the code into serial and parallel by percentage.

## 4 Conclusion

Parallelization is a powerful technique and one that can yield great gains — however, not all applications are well suited for it. A generalization can be drawn here that sections of code that require writing to an object or memory are ill-suited for parallelization. However, searching some data structure and only accessing memory for reads presents an excellent potential area for the use of parallelization.

## 5 Code

Code can be found at `https://github.com/rugggg/CADS/tree/master/CS540/nBody`

## References

[1] How Many Solar System Bodies
`https://ssd.jpl.nasa.gov/?body_count` *NASA JPL*.

[2] OpenMP `http://www.openmp.org/` [*OpenMP 4.5 Specifications*]. Annalen der Physik, 322(10):891921, 1905.

[3] The Barnes-Hut Algorithm *Tom Ventimiglia & Kevin Wayne*
`http://arborjs.org/docs/barnes-hut`

[4] The Barnes-Hut Algorithm *Tom Ventimiglia & Kevin Wayne*
`http://arborjs.org/docs/barnes-hut`