Report on the paper: "A Parallel Hashed Oct-Tree N-Body Algorithm"

Andreas Enz
supervisor: Prof. Peter Widmayer
Seminar: Algorithms for Database Systems
May 29, 2015

Abstract
In this report I will look at the concepts discussed in the paper "A Parallel Hashed Oct-Tree N-Body Algorithm" [1]. An N-Body algorithm is used to simulate the behaviour of N particles exerting some physical force on each other. Because N is usually very large (several millions) an efficient approach is needed. While some ideas from physics and mathematics are important to build and understand this algorithm, the main focus will be the decomposition of the particle domain and constructing a clean tree data structure such that the parallel computation can be done as efficiently as possible. Some tools that we will use include the concept of the Morton order, a spatial tree data structure without using pointers and a hash table. At the end I will provide insights on how this works together as a parallel algorithm running on a message passing architecture.

1 Introduction

N-Body algorithms simulate the behaviour of N particles exerting some force on each other. The forces can be different ones depending on which scale we simulate and what the particles represent. Examples are gravitational, coulombic and van der Waals forces. In this report I will use the example of stars ("particles") exerting gravitational force on each other. In practice we would look at this in a 3 dimensional space, but for the sake of simplicity and the clarity of figures we will stick to a 2 dimensional example. It is important to note that when we talk about the N-Body algorithm we only mean the algorithm that calculates the force interactions for a single timestep. A naive N-Body algorithm would therefore run in $O(N^2)$ and calculate every possible pair of particle interaction, then change the position of all particles depending on the calculations and the timestep. Since applications often have N reaching into several millions, reducing the $O(N^2)$ time complexity has seen much research effort. The paper [1] presents an algorithm very much influenced by popular methods like the Barnes-Hut algorithm [2]
1.1 Background

How is an improvement on $O(N^2)$ even possible if we have to calculate all possible particle interactions for an exact result? Well, we don’t have to do the exact calculation to stay within a reasonable error bound. N-Body Algorithms are based on truncated series approximation which allows aggregation of particles under some circumstances. I am only going to provide an intuitive explanation on this, since it is not in the scope of this report (and the paper for that matter). Imagine a dense cluster of stars (like a galaxy). For particles that are far away the force of this cluster can be approximated as coming from a single point source (center of mass). The same approximation can be done for smaller cluster within this cluster as depicted in Fig. 1 on the left.

The decision if a cluster of bodies can be approximated as a single entity in regards to some particle depends on the distance $d$ between particle and body (center of mass of the cluster), cluster radius $r$ and a maximal approximation error bound. The predicate that evaluates these parameters is usually called MAC (multipole acceptance criterion). I will write $MAC_p(c)$ which means the predicate is evaluated for some particle $p$ on body $c$ (which can be a cluster or single particle).

To represent the particle data and the aggregation of clustered particles in a hierarchical
manner, a spatial tree data structure is used. The leaves of this tree store data of particles and the nodes data of aggregations of particles (clusters) that are represented by values like center of mass, radius of the cluster. An important concept of this paper is, that no pointers are used to address the nodes of this tree. Instead a clever key scheme provides the structure of the tree. This will be especially important for the interprocessor communication where a pointer structure would lead to a lot of complications.

1.2 Overview of the algorithm

The Algorithm can be structured into the following steps:

- Decomposing the spatial domain
- Building the tree data structure
- Computing interacting bodies by traversing the tree
- Force evaluation of interacting bodies
- Load Imbalance

First the domain of particles is decomposed and all particles are distributed among processors. After that, particles are inserted into a tree data structure as leaf nodes. Inner nodes are constructed which represent an aggregation of its children. The root node of this tree represents an aggregation of the whole space of the particle domain. The most important step for us is the traversal of the tree, where all the bodies a particle needs to interact with are calculated. In the force evaluation step we will compute the physical interactions. Finally the Load Imbalance step will assign a weighting to particles based on how much computational work they were involved in.

I will concentrate on the first three steps which take as the input all the particle data. The output at the end of the third step will be an interact list for every particle. This list contains all the bodies (either particles or cluster) for which we have to do the force evaluation.

2 Decomposing the spatial domain

Here I will introduce the concept of a binary key which defines the position of a cell in the tree. This key will also be used to sort all particles in a linear list which allows for easy data decomposition for the parallel machine. Furthermore this key can be used in a simple hash table that provides access to an object in constant time.

2.1 Addressing the nodes in the tree data structure

The key for the particles is constructed by mapping the d-dimensional coordinates to a single binary number. This is done by first translating the floating point numbers of the coordinate vector into integers and then interleaving the bits. At the end a ‘1’ is
 prepended to the most significant bit. This helps to remove ambiguity between particle keys with a lot of zeros in the most significant bits and keys of internal tree nodes. An example of a bit interleaving for the two dimensional case would be:

\[
\begin{array}{c}
000011 \\
001100 \\
1.00.00.01.11.10
\end{array}
\]

Note that this key would be the same as the key 011110 of an internal tree node, if the '1' was not prepended.

**The key of internal nodes** can be addressed as follows:

- the root node has key '1'
- the key of daughter nodes of a cell with key k can be constructed by left-shifting k by d (dimension) bits and then adding numbers from 0 to \(2^d - 1\). This is shown in an example below for the values \(k = 101101\) and \(d = 2\).

<table>
<thead>
<tr>
<th>k</th>
<th>key daughter 1</th>
<th>key daughter 2</th>
<th>key daughter 3</th>
<th>key daughter 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>101101</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

The implicit tree structure (see Fig. 2) that is given through the key and the way we construct daughter keys allows us to address tree nodes without using pointers. This in turn can be used to access nodes quickly through a hash table.

**Figure 2:** The root cell represents the whole space. At the bottom a domain decomposition can be seen where every particle inhabits a single cell that is as large as possible. The tree is built to represent that decomposition. Cells either carry particle data (which are always tree leaves) or aggregated data of daughter cells like center of mass (drawn as a star).
2.2 Using the Morton order to sort particles

The way we constructed the particle key is identical to the Morton order, which allows us to sort them in a one dimensional list in a certain manner (see Fig. 3). This provides us with some properties that will come in handy in the data decomposition:

- Thanks to the Morton order, locality of the data is preserved: particles that are close to each other in space will be close in the sorted list. Particles which are far from each other in space are not close in the list. Morton Ordering is not perfect since there are some discontinuities.
- Having a linear list lets us split the data easily to distribute it to different processors
- The sorted list allows us to construct the tree in an efficient manner

3 The Hash Table storing the tree cells

The hash table plays a central role in accessing tree nodes on a single processor and especially across different processors. Every processor has its own table in which it stores local tree nodes. The length of the table that is used is much smaller than the key space. The hash function reduces the key to an h bit long string by selecting the h least significant bits of the key. This means there should be a lot of collisions, but those are not a big problem because of two reasons:

- We use a linked list to resolve collisions for each hash table entry
- More importantly: because of the data decomposition across several processors and the way we partitioned the data, a lot of collisions never occur because the tree nodes inhabit different hash tables. By decomposing the data with the help of the Morton order, we ensured that most tree nodes on a certain processor have keys with very similar most significant bits.
3.1 Hash table entry - an ‘hcell’

An entry in this Hash Table is called an ‘hcell’. (see Fig. 4) It consists of:

- One pointer to the tree node data (center of mass, mass, cluster radius, a.s.o)
- One pointer to a linked list to resolve collisions
- 2^d many one-bit flags that describe which daughters of the cell actually exist. This is redundant information but it lets us avoid one Hash Table lookup.
- If the node is a branch node then the address of the processor owning the branch is stored.

3.2 Accessing Data on another processor

All that is needed to request non-local data is the address of the processor that holds the data and the key of the tree node we are looking for. The key combined with the hash table is a very effective way for a global data accessing scheme using message passing. Fetching non-local data would be much more complicated if we were to use pointers to address tree nodes. Because accessing a daughter node on processor B from a node on processor A would require some sort of address translation.

4 Building the tree data structure

Using the Morton sorted list, the data gets split evenly among processors. These first build trees from their local data and then communicate with each other to connect their local trees to a global tree data structure. See Fig. 5 for an example.

4.1 Splitting the particle data among processors

The Morton sorted list is split into an equal part for every processor using an estimate on the expected workload. This estimation is done in a previous timestep in the Load Imbalance part of the algorithm by counting the number of interactions of every single
particle. The list is then split such that the number of interactions is about the same on every processor. This can mean that some processors have to store a lot more particle data but those particles in turn should not interact that much.

4.2 Building the local tree data structure

A simple idea would be to insert particles into the tree by starting at the root and traversing the tree until a leaf node is encountered. This leaf is then converted into an inner node which is stored in the hash table. The particles are then inserted into the tree as daughter nodes of the newly created inner node. We need to insert $N_{\text{proc}}$ particles and one insertion takes expected time of $O(\log N_{\text{proc}})$. Total time to build the local tree with $N_{\text{proc}}$ particles would be in $O(N_{\text{proc}} \log N_{\text{proc}})$.

The actual method that was chosen to build the tree in the paper [1] builds again heavily on the Morton sorted list. Using this list we can start the traversal for the particle insertion at the latest created node instead of the root. Since we know the data in the list is spatially close it will also be close in the tree. Inserting one particle into the tree then takes expected $O(1)$ time. But since we have to sort the list first we are back at $O(N \log N)$ running time to build the tree. We will need this Morton sorted list also in the data decomposition step, so this second method is an improvement in processing speed.

4.3 Connecting locally built trees across all processors

Now that every processor has built the data structure for the local data we need to connect these in a sensible manner. We do essentially the same as we already did for the local tree and finish building the whole tree in the same manner up to the root node '1' that represents the whole space. This is done on every processor. But of course we don’t want to duplicate the local data subtree of every processor, this is where the concept of branch nodes comes in.

**Branch nodes** describe the data decomposition on the coarsest level possible. A single processor may have several branch nodes - and therefore several disjoint local trees. Essentially, a branch node is much like the root node of the local tree. If we tried to construct a single branch node for every processor - which means building the local tree from bottom up until all nodes are connected to a single root, some processors would have the same branch node. Even worse, a branch of one processor could be inside (below) the branch of another processor. An example of those multiple branch nodes can be seen in Fig. 5 for processor P1.

Processors will construct these branch nodes after building their local tree and then exchange them with every other processor. Based on this information they will locally construct the top of the tree.
Figure 5: At the bottom a domain decomposition can be seen where the particles are connected in Morton order.
On the top is the corresponding tree for data points 1 to 9 which are distributed among two processors.
Assume the data decomposition would be such that processor P1 has particles [1,2,3,4] and processor P2 has particles [5,6,7,8,9].
P1 has two branch nodes B1 while P2 has only one branch node B2. Nodes below branches are only constructed on the processor owning the branch. Nodes marked with A (above branches) and all branch nodes B need to be constructed locally by every processor.
Processor P1 for example would store the tree consisting of the nodes marked with A, everything below and including B1 nodes and only the branch node B2. If it wanted to access particle 8 it would traverse its local tree down to B2 and then send a request to processor P2.

Figure 6: The hash table and collision lists of processor P1 after storing the tree in Fig. 5. Local trees of other processors are not stored, only their branch nodes. In this example values are hashed by their 2 least significant bits.
5 Computing interacting bodies by traversing the tree

Looking back at the big picture I want to remind the reader about what we really want to do, which is simulating the behaviour of \( N \) particles exerting some force on each other. Instead of using a naive algorithm and calculating all \( N^2 \) interactions we set up a tree data structure which consists of particles in the leaves of the tree and aggregations (cluster of particles approximated as single body) in the inner nodes of the tree. Furthermore I intuitively introduced the concept of the MAC, which is a predicate that decides for some particle if it may interact with an aggregation of bodies or needs to look at the daughter nodes of the aggregation.

The Tree Traversal part of the algorithm will create an interact\_list for every particle \( p \). In this list all particles or aggregate bodies are stored which have to interact with particle \( p \) according to the evaluation of the \( MAC_p() \).

This is done by a breadth-first traversal of the whole tree for every \( p \). Written here in pseudocode:

```
start at root cell, add it to walk\_list;
while walk\_list not empty do
    pick any cell c from walk\_list;
    for every daughter cell d of c do
        if MAC\_p(d) then
            add d to interact\_list;
        else
            add d to walk\_list;
        end
    end
end
```

When this traversal routine encounters branch cells the computation will stall while non-local data is retrieved. This is a huge performance concern. The paper [1] discusses a method to prevent this. The idea is to keep on working on local data and gather responses from other processors in a request\_list. Periodically this list will be copied to the walk\_list. The actual method is of course a bit more intricate and uses flags and book-keeping to i.e. prevent duplicate data requests.
6 Conclusion

We presented a way to decompose a spatial domain of particles by using the Morton order. This decomposition was used to distribute the computation on particle data evenly among several processors. To reduce the amount of interactions we need to calculate, cluster of particles were abstracted as single bodies. A tree data structure was used where particles are stored in the leaves and aggregations of particles as inner nodes. Using breadth first search we traversed the tree for every particle and calculated the bodies it has to interact with. This calculation of interacting bodies happens in every single timestep of the simulation.

The algorithm discussed in the paper [1] consists mainly of very basic methods of computer science which makes it a simple and yet effective tool for the task. The more complex ideas that were used were largely based on previous papers ([2] and [3]). That is why I feel the authors of the paper [1] should have discussed the performance of their algorithm more in comparison to existing ones.

References

