Optimising Performance of Diffusion-Limited Aggregation Simulations

S.G.Morgan and K.A. Hawick

Computer Science, Massey University, Albany, North Shore 102-904, Auckland, New Zealand

http://complexity.massey.ac.nz

Diffusion Limited Aggregation

Diffusion-Limited Aggregation (DLA)[1] is the growth process model and associated theories that help in explaining some physical form particles. An example of this are electrodepostition[2] and mineral deposits such as, gold clusters and other forms including soot and pollen. This process can be replicated through modern computer processing technology which can result into very large fractal clusters. These clusters can be grown to 10 million individual particles or more. Replication of this process can be achieved in a variety of ways in growing the clusters. Implementation of a selected process can vary based on what the individual user desires, however this can result in varied performance based on the implementation which effects memory usage, processing time and the users current available hardware[3].

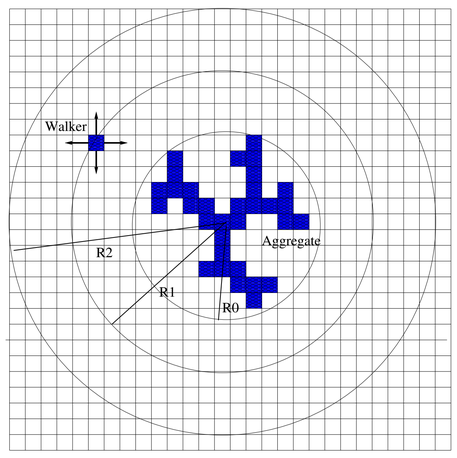

Figure 1: Basic Steps of the DLA Algorithm

Implementation of a simple DLA can be represented in a simulation either by growing the aggregates (clusters) in continuous space[4] or applied to a lattice (a grid). Figure 1 represents a lattice based forming of a DLA in which a walker(a particle that, if collides with the aggregate, will attach itself to it and become apart of it causing the aggregate to grow) is introduced in a random controlled area around the current aggregate. This walker is allowed to diffuse randomly in either direction until either it has collided with the aggregate or has been considered to have "fallen off". This simulation has an additional random control border that represents a maximum walking distance from the current aggregate this allows some form of limit for the walkers to help improve performance as shown by r2 in figure 1. This can create some bias towards the aggregate with having smaller random control areas for walkers to fall off or spawn from. Having a very large fall off area can result in very slow generation of a DLA due to the distance need for the walker to travel per time step. Additionally with the factor of each time step being diffuse randomisation can waste unnecessary processing time. Introducing these are very important as they can cause some issues with bias.

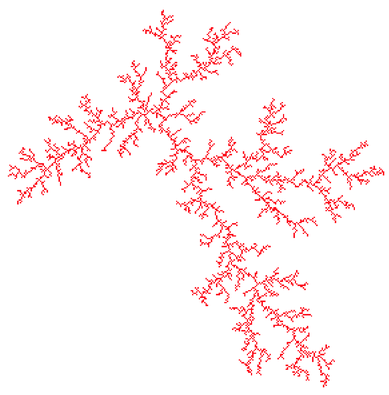

Figure 2: Non-Weighted DLA Aggregate

Figure 2 and Figure 3 show the way in which the choice of shifting the central point of the aggregation based on the mass can effect its generation process. While these can be represented with bias, this can also result in some very interesting patterns in the way aggregates can form based on controlled conditions. Figure 2 shows the effect of a stationary point of central mass. Figure 3 shows a possible result in which the walker attaches randomly to one side. As the central mass is shifted this creates two dominant branches of the aggregate in which both sides have more chance of being attached to by the walker.

Figure 3: Weighted DLA Aggregate

Generating DLA

As stated previously a DLA can be implemented in multiple ways. Each of these ways can effect the performance at which the current DLA can be generated and the current resources required in memory to store all the information about each particle. Shown below are a selection of ways to represent the difference between using a lattice based DLA and continuous space DLA and different implementations of each. While all these use different methods to read and write the information, however they use the same basic loops to access the values as shown in Figure 5. For each iteration a walker will be generated on the outer loop. The 1st step will check if the walker has fallen off, if it has the walkers current x and y values will reset to the current spawn area. If not the walker will check if there is any nearby values. If a value has been found to be near the walker, this will cause it to attach to the current aggregate and will stop the internal loop resulting in another walker spawning. If there is nothing around the walker, the walker will diffuse randomly in a random direction and then repeating the current process.

Algorithm 1 DLA Generation Steps

```
declare WalkerAmount
for i in WalkerAmount do
  Spawn Walker
  while true do
    if Fallenoff then
      Reset Particle
    end if
    if HasNeighbours then
      Add Particle
      Break
    end if
  end while
  Extend Bounds
end for
```

Figure 4: Basic Steps of the DLA Algorithm

The lattice DLA is based on a simple global grid, as shown in Figure 1. In this implementation of a DLA the grid is used to store the current points that have attached to the current aggregate. The central point of the lattice is set to give the generated walker an aggregate to attach too. When a walker is initialised it has it's own co-ordinates generated. The walker will diffuse randomly in either of the eight directions given, however the walker itself will not be applied to the current gird until it has attached to the aggregate. The walker will be stuck in a continuous loop until it has attached to the aggregate. When a walker has fallen off however, the walker will be reset to the random controlled spawn area. This will loop until the walker has attached itself to the aggregate. When the walker has been considered attached, the walker's position will be set on the current grid.

$$K - Index = (Y + maxWidth\ 2) \times maxWidth + (X + maxWidth\ 2) \quad (1)$$

The second implementation is based on continuous space. This implementation uses the unordered set container(this is only available though c++11 or the addition of the boost library) to store the points. A walker is generated in a similar fashion that the previous grid based DLA's walker has been implemented. For each time step however, due to the internal hash map of the unordered set, the "find" function is called to locate the necessary co-ordinate to check if it has attached itself to the aggregate. All these points are stored in a K-Indexed format as shown in Equation 1 to utilise a singular index accessor instead of comparing sets of cartesian points. Once the walker as been attached to the aggregate the k-indexing of the walkers final co-ordinates will be inserted into the unordered set as a K-Indexed value.

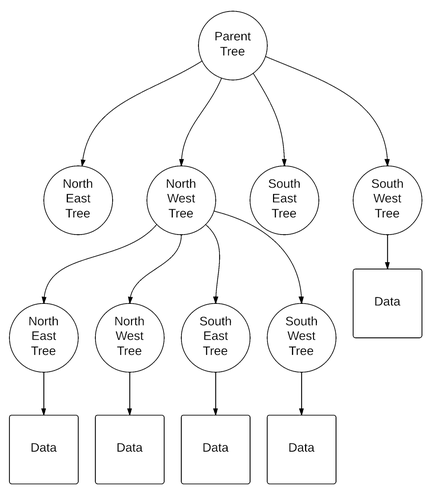

Figure 5: Diagram on how DLA is stored in a Quad Tree

The final implementation shown is also based on continuous space, however this is done using a Quad Tree algorithm for storing the points. Each tree represent a set area in 2D space and contains pointers to its possible child trees and the data stored in this node. The tree algorithm as shown in Figure 6 represents the way that the information is stored. The parent tree initially stores the information until the max capacity is reached (this is just a variable set by the user). Once reached, the parent will split into four equal subdivided sections as shown in Figure 7, and the parents information will be split and given to the corresponding child trees.

The intention of using the Quad Tree algorithm is to help limit the processing time in which points are checked. When a comparison or check is needed, the tree is recursively searched until the correct area is found and the check is done though all the variables in the tree to see if the walker needs to be attached to the current aggregate.

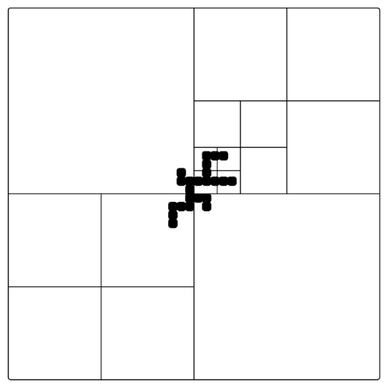

Figure 6: Representation of the DLA and being split into smaller trees once the point capacity has been reached

Simulations

With the different implementations of the DLA, basic timing tests were done to check the time in which the DLA has taken to form. Each style of DLA code had a basic implementation of timing in which all variables, such as particle amount, would be initialised. Timing would begin for the DLA and would continue until all particles have attached to the aggregate. To help not effect the consistency for the DLA, the fall off area variables and random spawn area variables were kept constant throughout all the different implementations of the DLA.

	Lattice	Quad Tree	Unordered Set
1000	0.01	4	11
5000	1.7	26	269
10000	16.6	127	998
15000	45.2	239	1996
20000	128.3	535	2770

Figure 7: Table for timings based on particle amount average over 5 trials

The performance differences between implemented continuous space segregation algorithms is clear, especially when the simulated participle population exceeds 2,000. The unordered set does use hashing to increase its performance, however it is clear that using a quad tree to store the information gives a far greater performance. By the time the unordered set has reached 20,000 points, is has taken 2,770 seconds—while the quad tree can establish the same actions in roughly one fifth of the time.

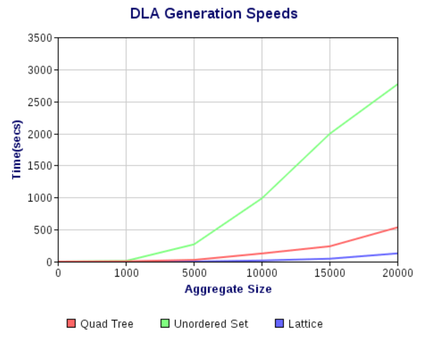

Figure 8: Chart comparison for methods for time vs particle amount

Timing for the lattice shows much faster speeds for the aggregate generation compared to all of the other methods implemented. The comparison of the unordered set and the lattice shows that it takes close to $1/21$ of the time take to generate an aggregate as shown on Figure 8. The trends on Figure 9 conveys that as the aggregate gets bigger, the required time per particle to join the aggregate increases.

	Lattice	QuadTree	Unordered Set
1000	250KB	8.3KB	8KB
10000	25MB	83.2KB	80KB
20000	50MB	166.4KB	160KB

Figure 9: Comparison table for the size in memory that each implementation take based off amount of particles used

A prominent outlier for the memory footprint between different implementation of these methods is the lattice. For example, when the lattice was given a 2000 by 2000 grid, each of which contains of one boolean which has the size of 1 byte regardless of the aggregate.

This results to roughly 4MB (megabyte) for the grid alone, plus the additional information in which the point's K-Indexed values are stored to be written. While the unordered set only has overhead + values (roughly 4 bytes per pointer+ 4 bytes for the stored value) which will usually result in less wasted memory. The quad tree however this does have a higher overhead for the quad tree per point(4 pointers for the each tree + for pointers for the value) than the unordered set.

With this information shown in figure 8 and figure 9 it is clear that the time for generating a particle is much faster when using a lattice based method for generation than the other types. However if memory is needed it seems that storing the information in a structure such as a quad tree or unordered set seems to be the best option.

Discussion

With the information given from the simulations, a clear result is seen in performance. Lattice based generated DLA's will always be considered the fastest due to the fact cells only need to check their corresponding neighbours as the information is stored in a grid regardless of the size of the aggregate. However, while this has a very powerful ability to generate the DLA at considerable speeds, once the size of the DLA starts to become bigger, the lattice must be given more memory for full or empty spaces. This will result in far more memory taken per particle than using a regular data structure.

It seem that choosing a way in which a DLA is created seems to be dependant on what the user has available in hardware and the size of the DLA needing to be generated. If creating a small DLA, the lattice would be recommended. If trying to create a larger DLA, it seems that a lattice would consume too much memory—for example, using a bit packed boolean(1 bit), creating a 1 billion point DLA would require roughly 120MB. While this does seem small to todays standards, this hasn't included the actual lattice itself. If a lattice is of 1,000,000 x 1,000,000 this would roughly take up 110GB(gigabyte) of physical memory, which is not readily available.

While the quad tree has far superior performance compared to the unordered set, this will still take a considerable amount of time to generate an aggregate that has a high amount of particles. Improvements, however, could be added to increase the performance. Some of the improvements could be to replace the pointers of the data in the quad tree with just a sub-divided grid and only having the grid be generated when a point occupies that area. This would result in removing the internal loop in which the data is searched and compared with the current walker's value. Other modifications that could be introduced is using bit packing for the point information as this is only stored but not rendered until it has been fully generated. This should help in saving as much memory as possible for larger generations.

It is important to do multiple runs of the generation of the aggregate due to the chaotic nature of the structure caused by the randomisation when the walker is diffusely moving. The impact of a walker causing a particle to be introduced could potentially affect following particles resting place and insertion time. For example, since the walker has equal chance to move any direction this can result in looping movement and wasted processing time.

Expanding on the current DLA is possible. Initial steps forward seem to be moving from the basic 2D generation of a DLA to a 3D or even a HyperDimensional DLA(4 dimension or greater). With this however, performance will be a key aspect as this will require far more processing power and memory for the additional axes. However, this does seem possible for a user to generate given the available hardware and creation of efficient code to generate the aggregate[**?**].

References

[1] T. A. Witten and L. M. Sander, "Diffusion Limited Aggregation, a Kinetic critical Phenomenon," *Phys.Rev.Lett.*, vol. 47, pp. 1400–1403, Nov 1981.

[2] G. Ackland and E. Tweedie, "Microscopic model of diffusion limited aggregation and electrodeposition in the presence of leveling molecules," *Phys. Rev. E*, vol. 73, pp. 011606–1–5, 2006.

[3] S. G. Morgan and K. A. Hawick, "High performance simulations and visualisations of hyper-dimensional diffusion-limited aggregation models," Tech. Rep. CSTN-244, Computer Science, Massey University, Auckland, New Zealand, 2013.

[4] P. Meakin, *Fractals, Scaling and Growth far From Equilibrium*. No. ISBN 0-521-45253-8, Cambridge University Press, 1998.

[5] K. A. Hawick, "Simulating and visualising sedimentary cluster-cluster aggregation," in *Proc. International Conference on Modeling, Simulation and Visualization Methods (MSV'10)*, no. CSTN-012, (Las Vegas, USA), pp. 3–9, CSREA, 10-15 July 2010. MSV3277.

[6] K. Hawick, "Morphology and epitaxial growth in a directed diffusion model," in *Proc. International Conference on Modelling, Identification, and Control (MIC2011)*, (Innsbruck, Austria), pp. 110–117, IASTED, 14-16 February 2011.