Sorting Point Clouds along an Axis

by Shawn Hillstrom

Thesis Advisor - Nikolay Strigul

Mathematics and Computer Science Departments

Abstract:

The research here details a more efficient method of finding volume given the partial point cloud of a roughly cylindrical object. Specifically, this research project will use sorting to analyze enormous data sets more efficiently. After sorting a data set, it becomes easy to separate it into several very small segments. The volumes of these segments can also be calculated very easily because we are dealing with the partial point cloud of a roughly cylindrical object. In turn, this means we can find the entire volume of the complete object by summing up the volumes of fitted cylinders to each segment. This being the end goal, determining an efficient method for sorting partial point clouds is the first challenge to overcome.

Introduction:

There are many reasons why one may want to know the volume of an object in real-time. Perhaps one wishes to calculate the mass of an object given an approximate volume and average density. One could also desire to understand approximately how much space an object (or objects) takes up. There are countless other examples, but the underlying point is that this concept of finding the volume of an object in real-time has applications in the real world. To name a few fields where tools to find volume of irregular shapes are useful: forestry (e.g. Gatziolis et al., 2015), paleontology (e.g. Falkingham, 2012, Falkingham et al., 2014), architecture and archaeology (e.g. Kersten and Lindstaedt, 2012).

A traditional approach to modelling an object is to assume that a point cloud is complete (e.g. you can scan an object from all sides) and then establish a mesh fit which acceptably balances complexity and geometry fitting errors (Berger et al., 2014). Several methods which use this general idea exist. For example, Ball pivoting (Bernardini et al., 1999), Marching cubes (Lorensen and Cline, 1987), Poisson surface reconstruction (Kazhdan et al., 2006), and the alpha-hull approach (Edelsbrunner et al., 1983). All these methods have strengths and weaknesses. For example, with the convex hulls of fossil bone structures (Sellers et al., 2012); however, these general methods for mesh reconstruction fail when you cannot scan an object from all sides which leads to missing information in the point cloud. Extremely noisy point clouds can cause issues as well, as geometric shapes cannot be accurately fitted.

The Problem

Problems arise when using traditional methods for modelling three-dimensional objects in that they assume a partial point cloud is complete and try to fit a singular geometric mesh which accurately represents both the complexity of the object and its geometric fitting error (Berger et al., 2014). These methods all break down when faced with incomplete point clouds or highly noisy point clouds (very irregular shapes).

Many algorithms used to reconstruct an object reduce the complexity of a point cloud by resampling 3D points into small clusters and then performing local optimization on each cluster (Schnabel et al., 2007). The fact that there are a smaller number of points to deal with in this approach means that the algorithm will have very little work to do and will, therefore, be extremely computationally inexpensive; however, a concentration of points which is not dense enough will lead to a failure to construct the real structure of the object which will, in turn, lead to inaccurate volume approximations. We, therefore, will need an algorithm which can work with the given point cloud or can accurately determine the lower limit on required complexity so that resampling does not affect the result. The former will most likely be easier.

Proposed Solution

Initially to solve this problem, we used a program which tried to fit numerous smaller geometric objects to the main object. This is more accurate for highly noisy or incomplete point clouds because irregular objects become much more regular the more you split them up into small segments. In addition, these segments do not necessarily have to be parallel to one another. We can dissect an object along non-parallel lines to enhance the overall accuracy of the approximation. The specific algorithm we used, though, was highly inefficient in this process. The approach used was sound, but the actual algorithm was not. Our task is to develop a method for more efficiently fitting shapes to smaller segments of the object.

Instead of trying to fit any one geometric shape to an object, we will split an object into fine segments and fit individual geometric shapes to each segment. The resulting volume sum should be a highly accurate approximation to the actual volume. A similar method was already tried but proved ineffectively slow so our task in this project is to create a new, more computationally efficient software package for finding the volume of any roughly cylindrical object given, at worst, a highly noisy partial point cloud.

Significance of the Approach

Specifically, we believe that sorting a point cloud will produce the results we desire. Sorting makes parsing easier and simplifies the structure of the point cloud for use in the computer. We can sort along a spatial axis initially, either the x-, y-, or z- axis, depending on the structure of the point cloud and then parse the sorted results into equal sections. From here it is easy to calculate a simple, rough object skeleton for the point cloud and then refine based on that rough skeleton using further perpendicular splits. This is what our specific research here focuses on: sorting the point cloud quickly and efficiently for parsing.

Methodology & Research:

The end goal of this line of research is to produce an algorithm which can quickly and efficiently find an accurate skeleton of a given partial point cloud so that volume can be easily re-constructed. In our case, this partial point cloud represents a single tree trunk as shown below in *Figure 1*.

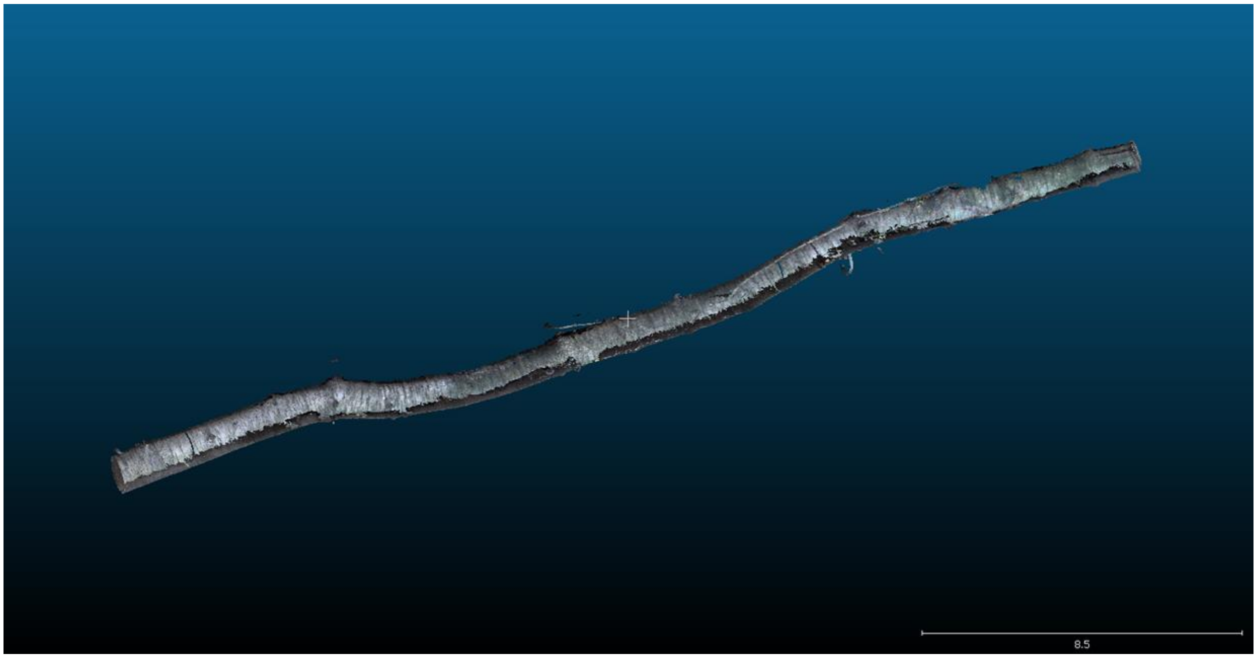

Figure 1

The proposed procedure to accomplish this goal is as follows: First the point cloud is split into sections along a spatial axis (e.g. the x, y, or z axes). Sorting the point cloud along said spatial axis and storing it in an array makes this easy for the algorithm to accomplish. For example, say the point cloud in *Figure 1* is longer along the x axis than it is along either of the other basic spatial axes. We then sort the point cloud along the x axis and split into even sections using indexes in the given sorted array of points. After sorting and splitting are completed, the next step is finding a basic skeleton. Using the centers of mass for each section and connecting the points, we can form a rough skeleton for the object, simplifying its shape for further calculation. From here, we can use lines perpendicular to our first simple skeleton to split the tree trunk more accurately, which we assume to be neither straight or of uniform shape. This process is then repeated to form increasingly accurate object skeletons.

The primary focus of this stage in our research is to formulate a method for quickly identifying and sorting along the most appropriate spatial axis and then implement it in Java. We chose Java because of its simplicity and ease of use, although we are also interested in building an identical algorithm later in C++, as this language is faster and more light-weight. Additionally, the algorithm is built and tested on one local machine with adequate specifications to run a Java development environment.

Identifying the appropriate spatial axis to sort by is a simple matter. While reading the points into memory, we simply find the minimums and maximums for each coordinate axis and compare when finished. The coordinate axis with the highest differentiation is the one we intend to choose to sort along. In simpler terms, we identify which axis the tree trunk is longest on, whether it be the x-, y-, or z-axis. The more complicated task is to formulate and implement an efficient sorting algorithm.

Initially, we decided to pursue sorting with Linked Lists. Linked Lists are simple and easy to implement data structures. Each node in the Linked List contains a data section in which we store spatial coordinates and a section with a memory address for the next node in the list. Linked Lists are easy to sort because inserting a new node between two other nodes is quite simple. One must only update the memory addresses in the previous node and the new node. Essentially, Linked List nodes can exist ambiguously in memory, not relying on physical continuity within adjacent blocks of memory but rather relying simply on virtual continuity, storing memory addresses. We then store the resulting sorted list in an array for easy parsing.

While this method is a very simple solution to the problem, it does have its drawbacks. The first drawback is that Linked Lists take up more memory than a simple array does. Also, by converting our Linked List into an array after sorting, we end up using even more memory until we release the memory used for the Linked List after conversion. Another drawback lies in the very nature of Linked Lists. When inserting a new item into a sorted Linked List of length n items, it will take at the worst-case O(n) comparisons to insert. In other words, in the worst case, we will have to insert at the end of the list each time, comparing to every node as we go. When inserting k items into a Linked List starting at zero items in the list, the worst-case comparison total is $n_0 + n_1 + n_2 + \ldots + n_{k-1} + n_k \leq n_k + n_k + n_k + \ldots + n_k + n_k \leq k^2$. Simplified, this means that in big-O notation, given k items to insert into a sorted Linked List, our worst-case run-time is $O(k^2)$. As we insert more items into our Linked List, our runtime should theoretically grow exponentially. This is undesirable for large data sets such as the one we are working with.

Because of these drawbacks we decided to switch to sorting with Hash Tables. Hash Tables are almost always implemented using only arrays which are, by their nature, very easy to parse; however, because of the non-uniform nature of our data, we chose to implement our Hash Table using an array of self-sorting Linked Lists so that each slot can hold multiple data points. Additionally, the Hash Table array size is set at a specified factor less than the actual data size. What this means is that we expect collisions but try to distribute collisions roughly equally over the entirety of the array. This is different from how Hash Tables are conventionally implemented. Generally, collisions are avoided but here we embrace them and use them to our advantage.

Hash Tables map data to array indexes using a function which mathematically converts data entries into integers falling within the bounds of the array. In this way data can be sorted in one easy step. The goal with our algorithm was to evenly distribute collisions over a Hash Table of a size which is a specified factor less than the total size of the data-set. We let s_h be the size of the Hash Table and s_d be the size of the data set. In our case, we let $s_h = s_d/100$. Additionally, we know the minimum and maximum along our axis of sorting for the data-set we are working with after reading it, so we let d_{min} be said minimum and d_{max} be said maximum. For any spatial coordinate along our axis of sorting *d*, the following equation accurately and evenly converts data points into array indexes:

$$(d - d_{min})(\frac{s_h - 1}{d_{max} - d_{min}})$$

Because we have all of the data available for the calculation above, this has a run-time of O(1). Constant run-time is perfect for the quantity of data points we are dealing with. Additionally, if we calculate k data indices using said algorithm, we have a total run-time falling within O(k). Linear growth in run-time is a clear step up from our Linked List run-time.

Using Hash Tables in our research proved effective, but we took it one step further by also implementing multi-threading. Multi-threading in Hash Tables is normally impossible because they use a single array to store data. Having multiple separate threads accessing one memory block is extremely dangerous. When multiple threads access one point in memory, it is completely up to the hardware which one writes its data. It is important then that we have measures in place so that threads do not interfere with each other. We can effectively make our entire Hash Table thread-safe by making each Linked List in our Hash Table object thread-safe. This is accomplished by queuing up threads on each Linked List object and running them as the Linked List becomes available. Essentially what this means is that threads must wait their turn to access each individual Linked List; however, threads can access different Linked Lists simultaneously. The proposed process still has linear runtime falling within the bounds of O(k) when inserting k items, but it has the potential to reduce the slope of said linear growth because some processes can run concurrently.

Data & Analysis:

Our analysis of sorted Linked Lists indicated that it may not be the most ideal method of sorting for our data set; however, it was extremely simple to implement and as such we tested its viability. We tested the sorting algorithm with various numbers of randomly generated point distributions. We used the average run-time in milliseconds from 10 trials and graphed against its associated number of items sorted. The number of items sorted was incremented by 10,000 each test, starting at 10,000.

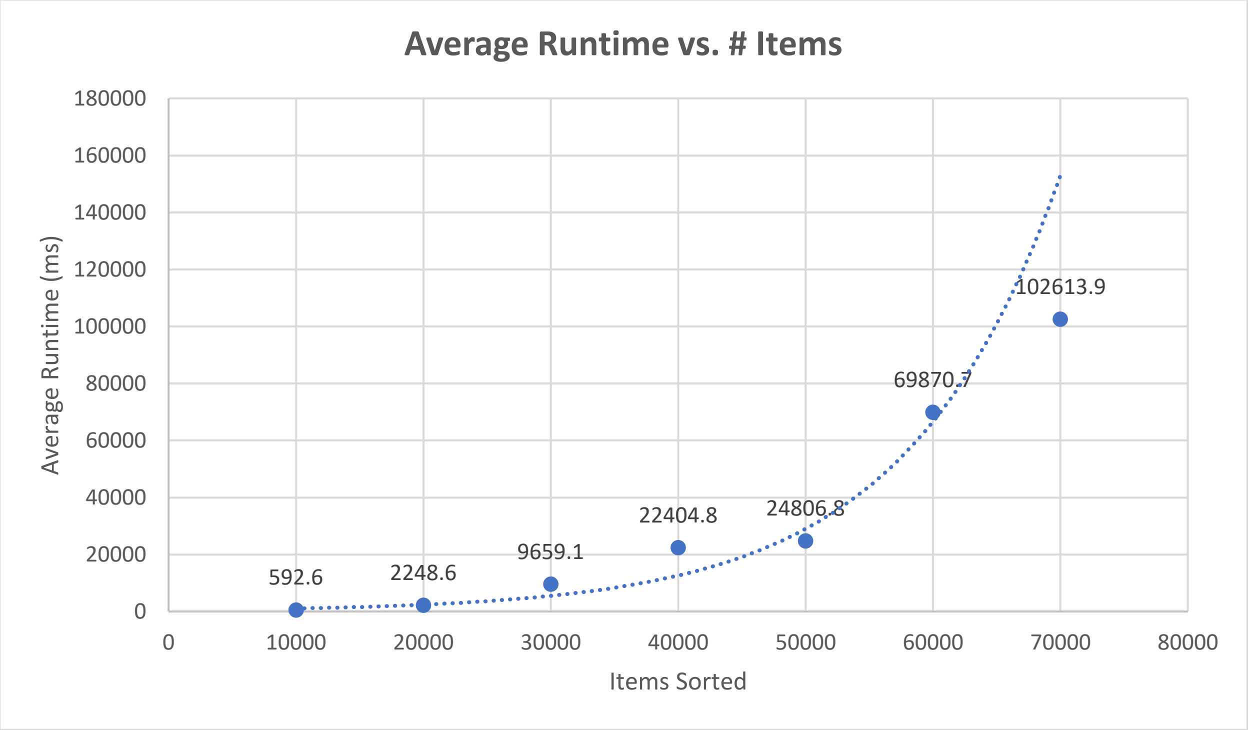

Figure 2

As can be seen from *Figure 2* Linked List sort demonstrated exponential growth, which was expected. By 70,000 items sorted, run-time was approaching two minutes, which is unacceptable considering the number of items we need to sort in our point cloud is close to 3,000,000.

The results from Linked List sort were not promising, so we move on to testing run-time from our Hash Table sort. Again, we tested the algorithm with increasing numbers of randomly generated points. We used the average run-time in milliseconds from 10 trials and graph against its associated number of items sorted. The number of items sorted was incremented by 10,000 each test, starting at 10,000 and we went up to 100,000 items sorted.

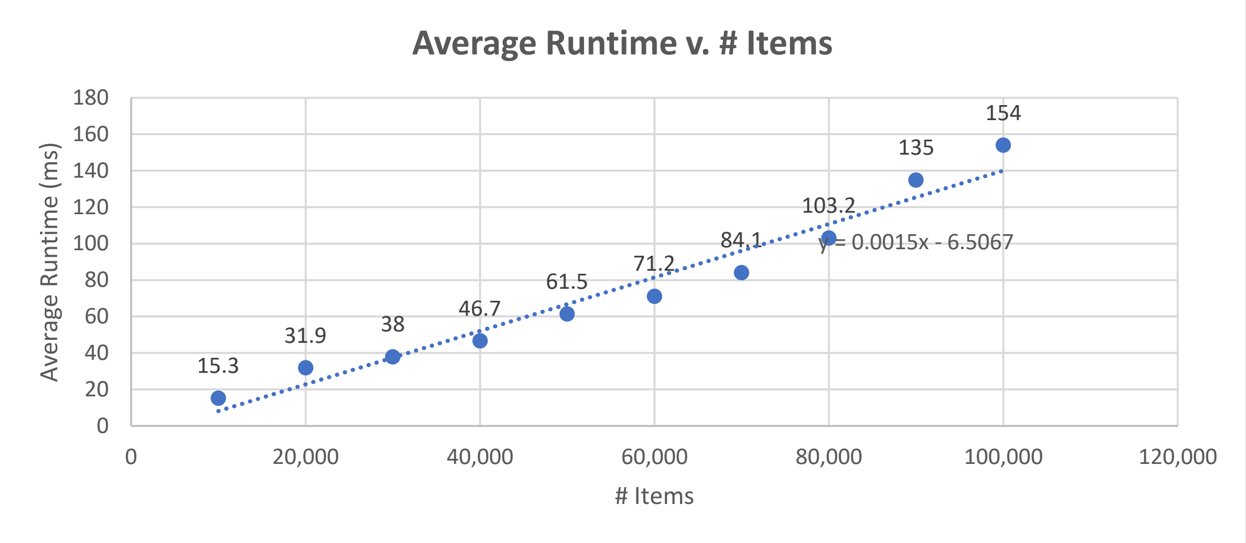

Figure 3

Figure 3 demonstrates the linear nature of our Hash Table algorithm's run-time. By 100,000 items sorted we are still under 1 second for run-time. This is a massive improvement from Linked List Sort so we try it with more points.

Here we start with 100,000 items sorted and increment by 100,000 up to 1,000,000 items. Just like the last test, run-time average from 10 trials is graphed against its associated number of items sorted.

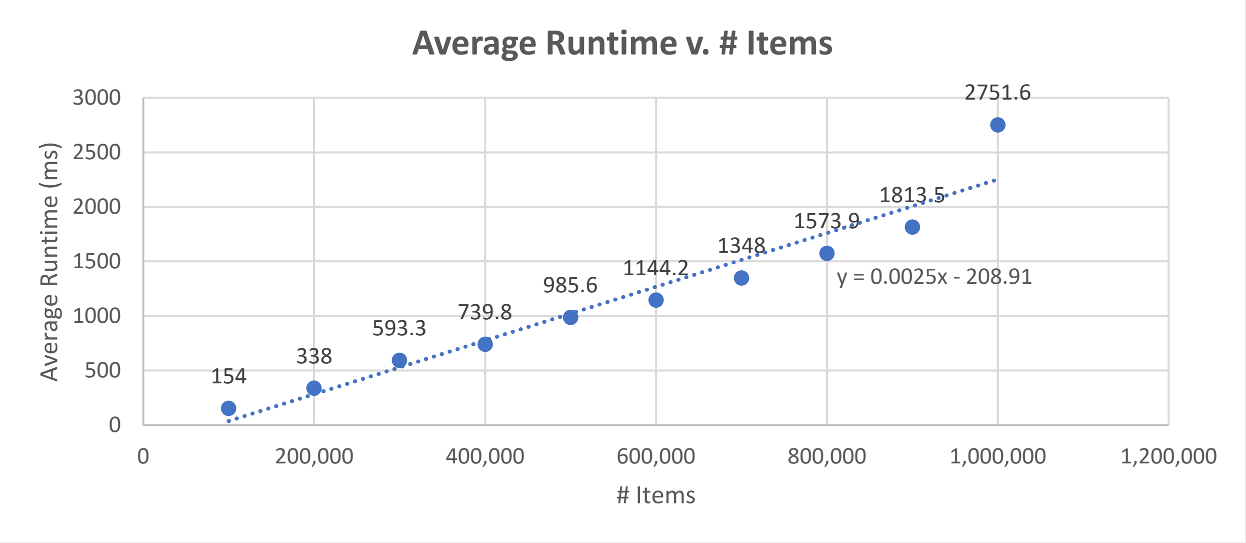

Figure 4

Although the last data point from *Figure 4* is somewhat of an outlier, the data is still consistently linear. It is increasing at a slightly higher rate which can be explained by the memory and computing restrictions on the local machine the algorithm was run on. Our highest run-time at

1,000,000 items sorted is still extremely low, so we decide to conduct more tests with increased number of items.

Here we start with 1,000,000 items sorted, and we increment upwards by 1,000,000 each time up to 10,000,000. We are now in the range of item totals we will be dealing with in our partial point clouds.

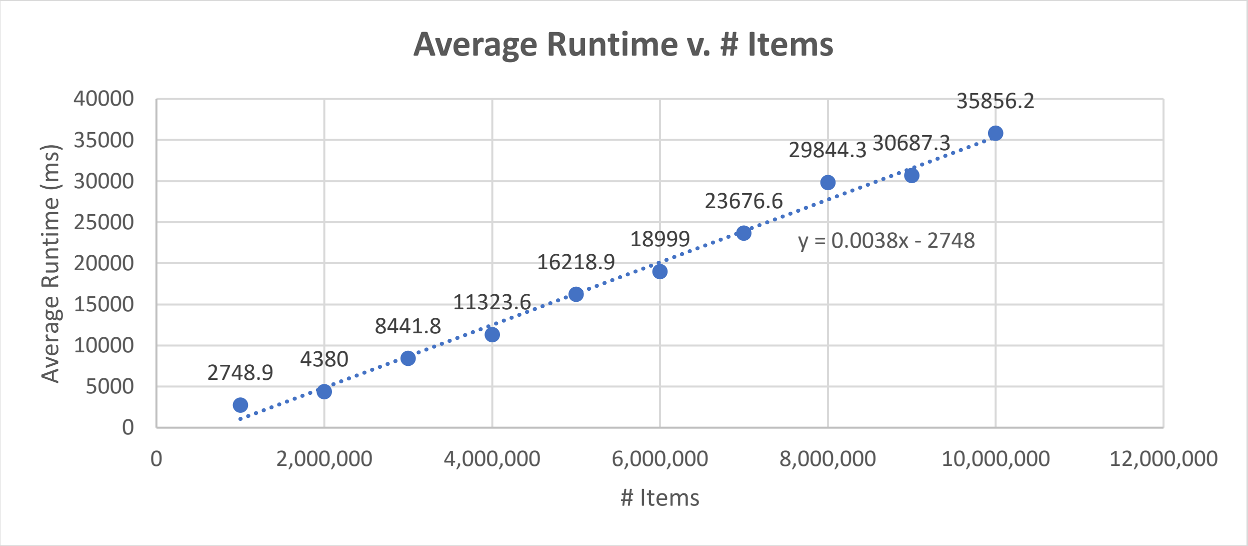

Figure 5

Run-time is still consistently linear, which is encouraging. Again, we have a slight increase in the slope of the line, but this is expected as we approach higher numbers of points sorted and we use more memory and computing power. The run-time averages here are acceptable for our purposes. Sorting 3,000,000 points at an average of 8.4418 seconds is fast enough to justify using sorting to eventually reconstruct the volume of a partial point cloud with approximately as many points. It would be interesting, though, to see if implementing multi-threading in our Hash Table Algorithm would produce measurable improvement. We reproduce the last three tests using our multi-threaded version of the Hash Table algorithm in the following figures.

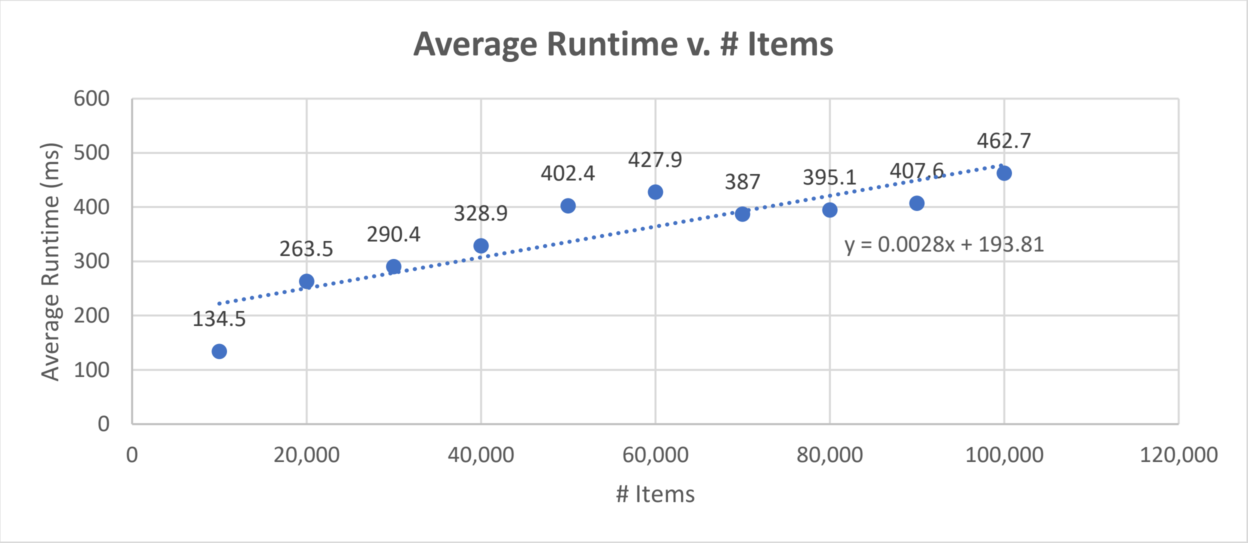

Figure 6

From the first test with our multi-threaded implementation of the Hash Table algorithm in *Figure 6* we see a measurable decrease in performance compared to the results in *Figure 3*. Run-time increases at a slightly faster rate, and average run-times are higher. These initial results are discouraging.

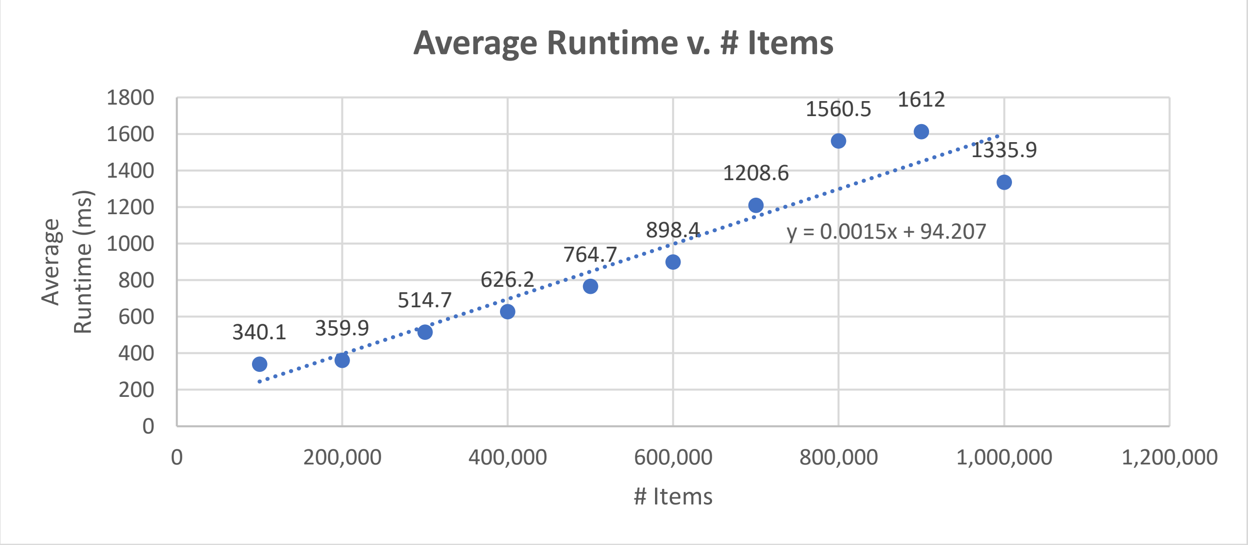

Figure 7

By the second test, however, going from 100,000 items up to 1,000,000, we see improvement. *Figure 7* demonstrates a slower rate of increase and lower average run-times than *Figure 4*. The initially discouraging results could have been due to the extra over-head generated by using multiple threads. Sorting more points, however, could potentially make this over-head irrelevant as is indicated by *Figure 7*.

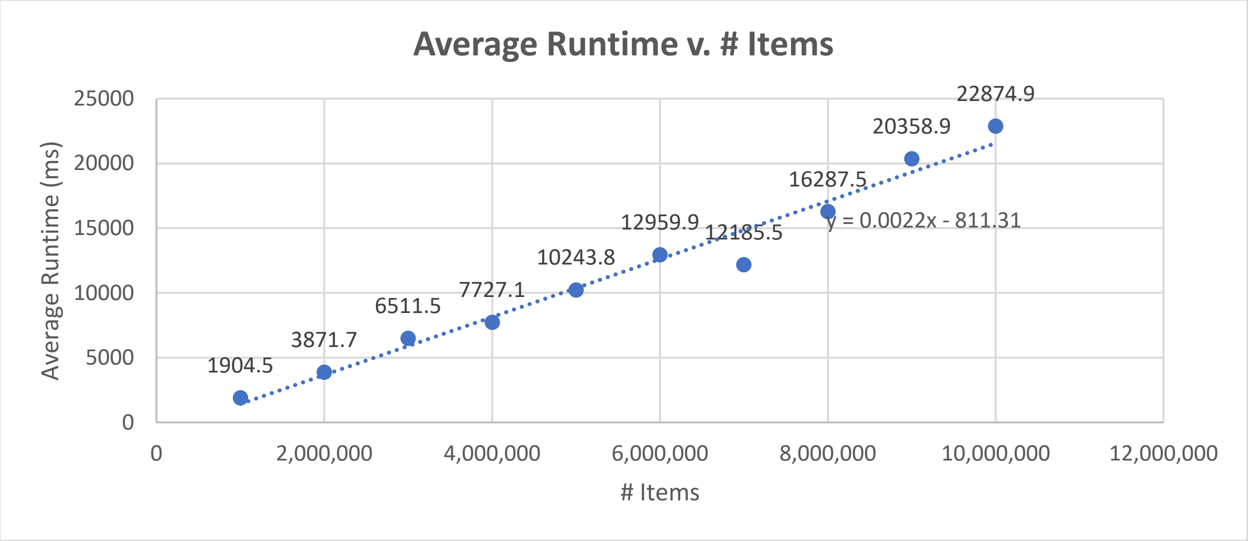

Figure 8

Just as in the last test, we see improvement from the normal Hash Table implementation. Using multi-threading in our tests from *Figure 8* remarkably decreased average run-times and overall line slope compared to the results from *Figure 5*. In *Figure 5* at 3,000,000 points sorted, we had an average run-time of 8.4418 seconds. In *Figure 8* at the same amount of points sorted we see an average-run-time of 6.5115 seconds. Additionally, we can see that at 10,000,000 points sorted, *Figure 5* displays an average run-time of 35.8562 seconds while *Figure 6* displays an average run-time of 22.8749 seconds.

The multi-threaded Hash Table sort displays improved results for high numbers of points but cannot compete with the normal implementation at lower numbers of points. This is almost certainly due to the high over-head cost of multi-threading on the operating system. For high numbers of points, though, increased performance and efficiency out-weigh this over-head cost. For our purposes, dealing with partial point clouds containing millions of points, multi-threaded Hash Table sort is the more desirable solution.

Conclusions:

Sorting partial point clouds into arrays along a spatial axis for easy parsing is extremely effective as it provides easy methods for computation later. Finding object skeletons becomes easier, and the skeleton can be refined through further parsing. Our initial idea of sorting with Linked Lists proved ineffectively slow but using Hash Tables in combination with Linked Lists to evenly distribute collisions proved highly efficient and fast. For small data sets, Hash Tables are superior, but for large data sets implementing multi-threading in said Hash Table produces even faster results.

Not only are multi-threaded Hash Table operations promising for this line of research, they are also promising in other applications where large data set management is important. Hash Tables are equally as speedy with insertions as they are with searches and deletions which means they can be used virtually anywhere large sorted data sets are important. To name a few specific industries in which this could apply: Social media, online television, and search engine mechanics. Anything which involves large, sorted data sets could potentially benefit from increased speed in sorting and access time. The only thing which would need to change is the algorithm which generates array index integers, as each data set involves different distributions and data types. The algorithm used here was specifically built to operate with floating point 3D-point coordinates.

Eventually the end goal is to build algorithms for computing object skeletons, refining them, and then using them to reconstruct volume, but our primary focus is to make the algorithm developed here even more efficient. In the future, it could be beneficial to build an algorithm using a more efficient, light-weight programming language, such as C++ or perhaps Python. These are both languages which could prove promising, although whether they will provide an improvement on performance compared to Java has yet to be seen. Additionally, it would be interesting to see how the algorithm would run on a clean server machine. The algorithm built here was both built and tested on a local machine, which means that outside program over-head on the operating system is much more variable than on a server computer.

References:

Berger, M., Tagliasacchi, A., Seversky, L., Alliez, P., Levine, J., Sharf, A., and Silva, C. (2014). State of the art in surface reconstruction from point clouds. In EUROGRAPHICS star reports, volume 1, pages 161–185.

Bernardini, F., Mittleman, J., Rushmeier, H., Silva, C., and Taubin, G. (1999). The ball pivoting algorithm for surface reconstruction. IEEE transactions on visualization and computer graphics, 5(4):349–359.

Edelsbrunner, H., Kirkpatrick, D., and Seidel, R. (1983). On the shape of a set of points in the plane. IEEE Transactions on information theory, 29(4):551–559.

Falkingham, P. L. (2012). Acquisition of high resolution three-dimensional models using free, open-source, photogrammetric software. Palaeontologia electronica, 15(1):15.

Falkingham, P. L., Bates, K. T., and Farlow, J. O. (2014). Historical photogrammetry: Bird's paluxy river dinosaur chase sequence digitally reconstructed as it was prior to excavation 70 years ago. PloS one, 9(4):e93247.

Gatziolis, D., Li´enard, J. F., Vogs, A., and Strigul, N. S. (2015). 3d tree dimensionality assessment using photogrammetry and small unmanned aerial vehicles. PloS one, 10(9):e0137765.

Kazhdan, M., Bolitho, M., and Hoppe, H. (2006). Poisson surface reconstruction. In Proceedings of the fourth Eurographics symposium on Geometry processing, volume 7.

Kersten, T. and Lindstaedt, M. (2012). Potential of automatic 3d object reconstruction from multiple images for applications in architecture, cultural heritage and archaeology. International Journal of Heritage in the Digital Era, 1(3):399–420.

Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. In ACM siggraph computer graphics, volume 21, pages 163–169. ACM.

Schnabel, R., Wahl, R., and Klein, R. (2007). Efficient ransac for point-cloud shape detection. In Computer graphics forum, volume 26, pages 214–226. Wiley Online Library.

Sellers, W., Hepworth-Bell, J., Falkingham, P., Bates, K., Brassey, C., Egerton, V., and Manning, P. (2012). Minimum convex hull mass estimations of complete mounted skeletons. Biology Letters, page rsbl20120263.