Evaluation of MapReduce for Gridding LIDAR Data

Sriram Krishnan, Chaitanya Baru, Christopher Crosby
San Diego Supercomputer Center, UC San Diego
9500 Gilman Dr MC 0505
La Jolla, CA 92093, USA
{sriram, baru, ccrosby}@sdsc.edu

***Abstract* — The MapReduce programming model, introduced by Google, has become popular over the past few years as a mechanism for processing large amounts of data, using shared-nothing parallelism. In this paper, we investigate the use of MapReduce technology for a local gridding algorithm for the generation of Digital Elevation Models (DEM). The local gridding algorithm utilizes the elevation information from LIDAR (Light, Detection, and Ranging) measurements contained within a circular search area to compute the elevation of each grid cell. The method is data parallel, lending itself to implementation using the MapReduce model. Here, we compare our initial C++ implementation of the gridding algorithm to a MapReduce-based implementation, and present observations on the performance (in particular, price/performance) and the implementation complexity. We also discuss the applicability of MapReduce technologies for related applications.**

Keywords: MapReduce, Gridding, Digital Elevation Models, LIDAR

I. INTRODUCTION

The MapReduce programming model [1], introduced by Google, has become popular over the past few years. Apart from Google's proprietary implementation of MapReduce, there are several popular open source implementations available such as Apache Hadoop MapReduce [2] and Disco [3]. MapReduce technologies have also been adopted by a growing number of groups in industry (e.g., Facebook [19], and Yahoo [20]), and there are several database vendors such as GreenPlum [4] and AsterData [5], who leverage concepts of MapReduce in their data warehousing solutions. In academia, researchers are exploring the use of these paradigms for scientific computing, for example, through the Cluster Exploratory (CluE) program, funded by the National Science Foundation (NSF).

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. The MapReduce paradigm is designed to be scalable and conducive to data-intensive and data-parallel applications, is fault-tolerant by design, and meant to be run on non-specialized commodity hardware. Hence, in theory, MapReduce technologies could provide better price/performance ratio than traditional HPC or database technologies, which may require proprietary hardware and software configurations.

In this paper, we evaluate the use of MapReduce technology to implement a local gridding algorithm for the generation of Digital Elevation Models (DEM) from dense measurements acquired by LIDAR (Light, Detection and Ranging) remote sensing technology [17], and compare the implementation to an initial C++ implementation. The overall performance is of interest since the algorithm is a key processing tool offered by the OpenTopography Facility (http://www.opentopography.org), an NSF-funded data facility hosted at the San Diego Supercomputer Center, for online access to Earth science-oriented LIDAR topography data.

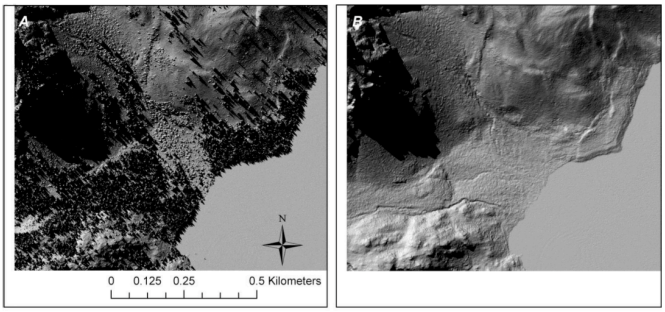

Figure 1. Teton Fault zone: Hillshade imges constructed from: A) Full feature DEM (1x1 km wide, 0.5 m grid resolution) - includes all vegetation details, and B) Bare earth DEM – constructed from ground returns only

Digital Elevation Models (DEMs – also called Digital Surface Models (DSM) and Digital Terrain Models (DTM)) are a digital continuous representation of the landscape (see Figure 1), where each (X, Y) position is represented by a single elevation value. A DEM can be represented as a raster (a grid of squares) or as a triangular irregular network (TIN), and can be generated from remotely sensed or directly surveyed elevation information. DEMs are used for a range of scientific and engineering applications, including hydrologic modeling, terrain analysis, and infrastructure design. One of the fundamental processing tasks in the OpenTopography system is generation of DEMs from very dense (multiple measurements per square meter) LIDAR topography data. To accomplish this task, OpenTopography uses a simple local gridding approach, where points within a given circular local "bin" are used to calculate value of a given DEM node.

This work is funded by the National Science Foundation's Cluster Exploratory (CluE) program under award number 0844530, and the Earth Sciences Instrumentation and Facilities (EAR/IF) program & the Office of Cyberinfrastructure (OCI), under award numbers 0930731 & 0930643.

Here, we compare our initial C++ implementation of this gridding algorithm to a MapReduce-based implementation, which leverages the open source Apache Hadoop framework. We present some early results and provide our observations on the performance and implementation complexity, and discuss our thoughts on the applicability of MapReduce technologies for applications of a similar nature.

The rest of this paper is organized as follows. In Section II, we provide background on the local gridding algorithm that is used for the Digital Elevation Model generation. In Section III, we present the initial C++ and Hadoop-based implementations for this algorithm. We describe our various experiments and their observed performance in Section IV, and discuss our observations and inferences from these experiments in Section V. Finally, we describe our ongoing and future work in Section VI, and conclusions in Section VII respectively.

II. Algorithm Overview

The process of surface gridding of irregularly spaced elevation measurements involves estimating the elevation at a specific grid cell based on the surrounding measurements [18]. Gridding methods can be broadly classified into 1) global approaches such as Kriging [22], which use the measurements from a large area around the grid node, or 2) local, which utilize only the measurements immediately adjacent to the grid node to calculate the value. Given that natural topography is not dependant on regional trends (i.e. the elevation at a given point is not influenced by the elevation elsewhere in the region), a local gridding methodology for topographic measurements is appropriate.

The nature of LIDAR point cloud data also lend themselves to a local gridding approach. Because these data are collected at 10s to 100s of kilohertz from a low-flying aircraft, they typically sample elevation at a spacing of significantly less than a meter. However, researchers typically perform their analyses on DEMs with resolutions of half a meter or more. Thus, in many research grade LIDAR datasets, each pixel in the DEM may have been sampled several times. Thus, fitting a surface between points to estimate elevation at the DEM node is not required, and induces unnecessary computational burden. However, the LIDAR returns are not spatially uniformly distributed while the DEM, by definition, is a regular gridded representation of the surface.

In this paper, we study and evaluate the performance of the local gridding algorithm described in [8], and implemented by the OpenTopography Facility. The algorithm computes a circular neighborhood defined around each *grid cell* based on a radius provided by the user (see Figure 2). This neighborhood is referred to as a *bin*, while the grid cell is referred to as a *DEM node*. Up to four values — minimum, maximum, mean, or inverse distance weighted (IDW) mean — are computed for points that are in the bin. These are then assigned to the corresponding DEM node, and used to represent the elevation variation over the neighborhood represented by the bin [9]. The grid resolution, and the radius of the bin are input parameters selected by the user.

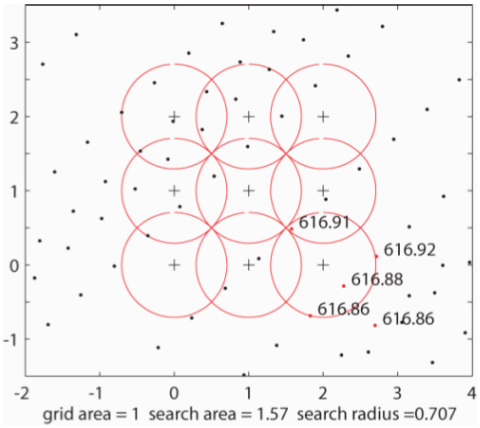

Figure 2. Illustration of local binning geometry. Black dots are actual LIDAR returns. Red dots in lower right have elevations shown. Plus symbols indicate locations of DEM nodes at which elevation is estimated. Each circle represents a local "bin" for a particular DEM node. The DEM values are computed by applying simple mathematical functions on all points that fall within the local bin. Units are arbitrary, but typically are meters.

III. Implementation Overview

The initial implementation of the local gridding algorithm for DEM generation was in C++, and is currently being used in production mode by the OpenTopography Facility. We have also implemented the same algorithm using the Java-based MapReduce framework provided by Apache Hadoop. Both implementations use the same input and output formats, described below.

For the purposes of this discussion and analysis, the input point cloud data can be represented simply as {X, Y, Z} tuples, X and Y represent the latitude and longitude of the point location, and Z represents the elevation at that point. In practice, however, the data tuple also includes other elements stored in the point cloud database, such as *classification* (ground versus non-ground) and *acquisition time*.

The DEM output is generated by both implementations in the ESRI ASCII Grid format (also known as ArcASCII - [7]). This format consists of header information, including the number of rows and columns in the grid, the coordinates for the lower left (or southwest) corner of the grid, and the grid size, followed by the elevations for each grid cell, starting with the upper left corner (northwest).

A. C++ Implementation

The initial C++ implementation of the local gridding algorithm for DEM generation was first discussed in [8]. There are two versions of this implementation – in-core and out-of-core. The in-core version loads the entire DEM into memory, whereas the out-of-core version uses secondary storage for saving intermediate results when the size of the DEM exceeds available memory. The details of the in-core and out-of-core versions of the implementation are as follows.

1) In-Core Version

Figure 3 shows the implementation of the C++ in-core version. If the entire grid representing the DEM (i.e. the output to be generated) can be stored in memory, then the required memory for the DEM (represented as a 2D array) is allocated using a *malloc* statement, and initialized. Each entry in this 2D array corresponds to a DEM node – and is represented by a C++ structure called *GridPoint*, which includes fields for the *min*, *max*, *mean*, *idw* and *count* values for that DEM node. After initialization, the point cloud data is read line by line from an input file – for every point in the file, the bin radius specified by the user is used to compute the set of DEM nodes that fall within its neighborhood. Next, the values in these DEM nodes are correspondingly updated. Every line in the input file is read only once, thus minimizing the I/O operations involved. Once all the input points have been processed, the values stored in the GridPoint data structure for each DEM node are finalized and a DEM is generated in the ArcASCII Grid format.

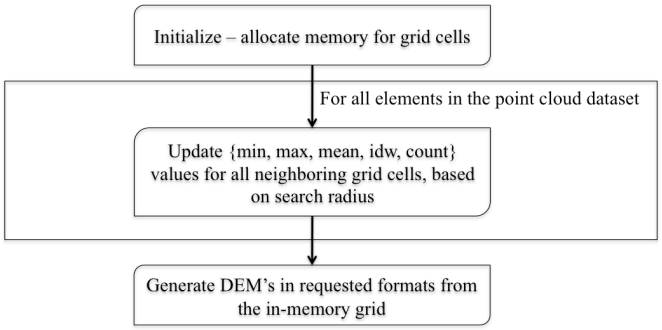

Figure 3. Overview of the C++ in-core implementation. Every point in the input file is read just once, and assigned to the local bins of the grid cells in its neighborhood. The grid cells are finally updated in memory, and results are written out in ArcASCII format after all the input points have been processed.

If N is the number of points in the point cloud dataset and G is the number of cells in the output DEM grid, then the time required to read the entire input is $O(N)$, and to initialize, compute and write out the output is $O(G)$. Hence, the total algorithm runtime is $O(N+G)$.

2) Out-of-Core Version

If the available memory on the system is less than the size of the entire grid, then the grid is partitioned into multiple blocks (sub-grids) and the algorithm is run "out-of-core". A simplified overview of this out-of-core implementation is shown in Figure 4. The block size is determined by the available memory such that an entire block fits into main memory. If M is the block size, in terms of number of grid cells in a block, then the total number of blocks is given by $\lceil G/M \rceil$.

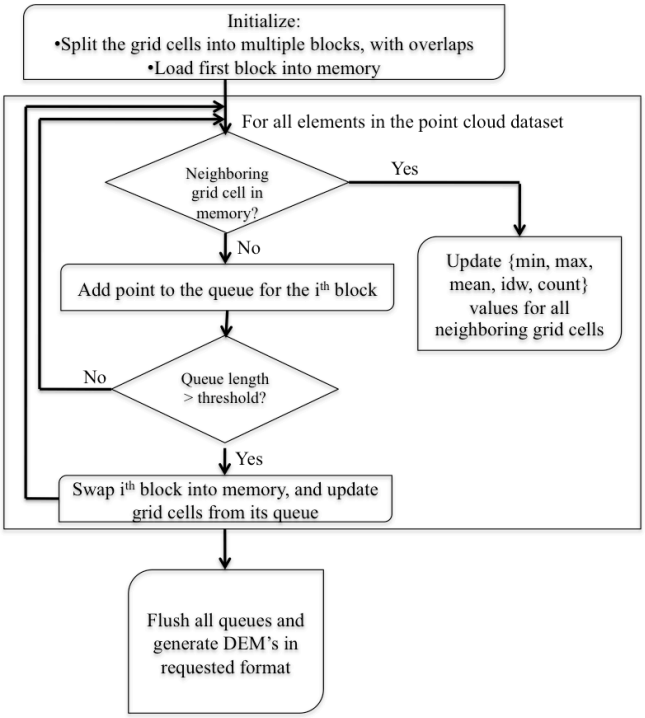

Figure 4. Overview of the C++ out-of-core implementation. Individual blocks of the grid are loaded into memory and updated one at a time. After all input points are processed, the final DEM is generated by merging the individual grid blocks, and the results are written out in ArcASCII format

To begin, the first block (sub-grid) is loaded into memory. The input point cloud dataset is read line by line from the input file. As before, for each point, the corresponding DEM nodes are computed, based on the search radius. If the DEM node is in the block that is currently in memory, the corresponding GridPoint values are updated. If not, the values are queued for processing when the corresponding block is loaded into memory at a future time. If the length of any queue reaches a defined threshold, the current block in memory is written out to a file, and the new block is loaded in memory. To allow for computation of values at the edge of each block, blocks are created with some overlap – the size of which depends upon the search radius. Once the entire input has been read, all the blocks are updated sequentially, the queues are flushed, and the final DEM is written out in the ArcASCII Grid format.

Since the input file is read only once, the read time is $O(N)$. The time to initialize the memory for each block is $O(M)$ and the time to write each block is also $O(M)$. If each block is initialized only once and written once (i.e. the best case, when input data are sorted), then the time to initialize and write output is $O(\lceil G/M \rceil \bullet M)$, or $O(G)$. Thus, in the best case, the overall complexity is the same as that of the in-core version, viz. $O(N+G)$. However, these blocks may have to be written out to disk and re-read in the case when the input is not well sorted (though, in actuality, the input data tend to be relatively well sorted). If C_{write_M} is the cost of writing each block of size M, and C_{read_M} is the cost of reading the block, then the overheads due to swapping of blocks caused as a result of non-

sorted inputs is, $O(f \bullet (\lceil G/M \rceil \bullet (C_{write\_M}+C_{read\_M}))$, where f is a "fudge factor" that accounts for the level of unsortedness of the input.

B. MapReduce Implementation using Hadoop

The MapReduce implementation of the local gridding algorithm using Apache Hadoop is shown in Figure 5.

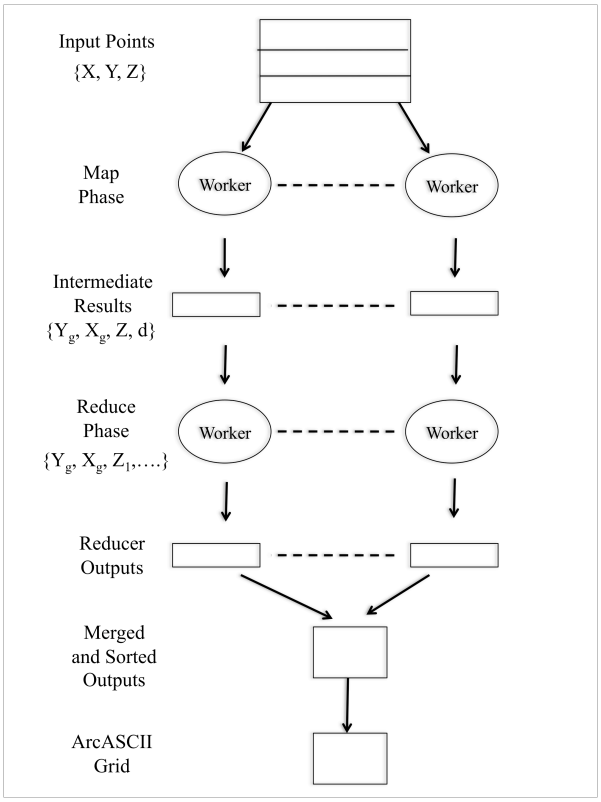

Figure 5. Hadoop-based DEM implementation. In the Map Phase, input points are assigned to corresponding grid cells (local bins), and in the Reduce phase the corresponding elevations for each grid cell are computed from the local bins. The reduced outputs are merged and sorted, and the DEM is generated in the ArcASCII grid format.

The implementation consists of three phases as follows:

Map Phase: In the Map phase, a Hadoop program generates a set of intermediate key/value pairs from a set of input key/value pairs. In our implementation, the input key is the line number for every line in the point cloud, and the value is the content of the line (the *{X, Y, Z}* tuple). For every such key/value pair, we compute the set of DEM nodes for which this point falls in the corresponding bin, and the corresponding distances from these nodes. The intermediate key is of the form $\{Y_g, X_g\}$, and the values are of the form *{Z, d}*, where $\{Y_g, X_g\}$ correspond to the coordinates of the DEM node, the set of Z values correspond to the elevation of the points in the bin, and d is the distance to each point from the corresponding grid cell. We choose to emit $\{Y_g, X_g\}$, rather than $\{X_g, Y_g\}$ to aid in the generation of the outputs – it is beneficial if the outputs are ordered with decreasing Y and increasing X values because of the way the points are laid out in the ArcASCII grid file – starting from the upper left corner to the lower right corner of the grid. The map phase takes $O(N/M)$ time, where N is the input size and M is the number of map processes.

Reduce Phase: In the Reduce phase, a Hadoop program generates the final set of output key/value pairs from the intermediate set. In our implementation, the input key is $\{Y_g, X_g\}$, and the input value is an array of *{Z, d}*, corresponding to that DEM node. Using the Z and d values, we compute the average, inverse distance weighted (IDW) mean, minimum, maximum and counts for a particular grid point, and emit $\{Y_g, X_g, Z_1, ...\}$ values for every DEM node. The reduce phase takes $O(N/R)$ to read the input and $O(G/R)$ to write outputs, where N is the size of the input dataset, G is the size of the output grid, and R is the number of reduce processes.

Output Generation Phase: Each reducer generates one single output file. The reduced outputs need to be first merged, and then sorted by descending Y and ascending X values to aid in output generation. The sorted and merged values are then fed to a Java program that generates the DEM in the required format. If the number of reducers is one, then there is no need to do any merging – in fact, the reducer can emit the outputs in the required order using Hadoop's *KeyFieldBasedComparator* class. If the number of reducers is greater than one, then we use utilities provided by Hadoop itself to merge and sort the outputs (using Hadoop Streaming [23], *Identity* Mappers and Reducers, the *KeyFieldBasedComparator* to sort the reduce outputs, and setting the number of Reducers to one). In the worst case, the merging and sorting takes $O(G \bullet logG)$ time.

Note that we have used a Java program to generate the output file for consistency purposes, because the rest of the Hadoop code is written in Java. Although there is some potential to speed up the output generation using compiled code such as C or C++, we did not do so for ease of programming and manageability of code. This phase takes $O(G)$ time to produce the output.

IV. Experiment Evaluation

A. Overview of Experiments

The overall goal of our experiments was to get a better understanding of how our implementations perform for different data sizes and input parameters, on different types of hardware. In particular, we were interested in comparing the performance (and price/performance) of the initial C++ implementation on both commodity and HPC resources, versus the Hadoop implementation running on the commodity cluster.

In terms of the algorithm itself, we were interested in studying how the two implementations scaled when we increased the number of points in the input point cloud (from a few million to a few hundred million); and, when we increased the size of the DEM output by increasing grid resolution. Note that changing the grid resolution from *1x1m* grid cells to

0.5x0.5m grid cells, for example, quadruples the grid size. We also wanted to investigate the effect of the grid resolution on performance for both the initial C++ and the Hadoop implementations, and if the effect was similar or different for the two implementations.

For our HPC platform, we used a traditional high performance SMP resource, accessed via the TORQUE resource manager (also known as PBS [10]). The resource featured 28 Sun x4600M2, eight-processor quad-core nodes. Eight had 512 GB of memory, and the remaining 20 had 256 GB. The total system bandwidth was 112 GB per second. The 256GB nodes had eight GB per core and the 512GB nodes had 16 GB per core. Each node had eight AMD 8380 Shanghai 4-core processors running at 2.5 GHz. Costs for these nodes varied from $30K to $70K USD, depending on the available memory. For our experiments, we focused on the 512 GB nodes.

For the commodity resources, we used an 8-node cluster that we had assembled from off-the-shelf components. Each node had a quad-core AMD Phenom™ II X4 940 Processor at 800MHz, and had 8GB of memory. They were connected together by Gigabit Ethernet. Every node cost around $1,000 USD each.

For our experiments, we used four input data sets varying from 1.5 million to 150 million points, which are available for download from the OpenTopography.org portal. The overall point density was 6-8 points per square meter. The 1.5 million points input was around 74 MB in size, while the 150 million points input was around 6.9 GB in size. The two intermediate input datasets were 13 million points (628 MB) and 67 million points (3.2 GB) points, respectively. With this selection, we had a set of inputs that varied from small to large point clouds, and experiments based on them were sufficient to evaluate the overall performance of our implementations.

B. C++ Implementation

The performance of the C++ implementation on the commodity and HPC resources are shown in Figure 6. As described in Section III A, the code can run in either in-core or out-of-core mode depending on the memory availability. Because the HPC node had 512GB of memory available, all runs on that were in in-core mode. On the other hand, the commodity node only had 8GB of memory available. Hence, the runs on the commodity node ended up in out-of-core mode for the larger grid sizes. The threshold for the number of grid cells for switching to out-of-core mode was set to 140 million. This threshold can be set in the C++ code, and is a function of the available memory on the resource. The size of the GridPoint data structure is 52 bytes (5 doubles and 3 integers) – hence, the total size of 140 million grid cells was set to be slightly lower than the total memory on the commodity resource.

As evident from figure 6, the performance on the commodity resource and the HPC resource are very similar for the smallest input size (1.5 million input points) – on both resources, the grids can be easily fit into memory for all three grid resolutions (0.25m, 0.5m and 1m). For this input size and grid resolution, the performance is more I/O-bound than CPU or memory-bound. Even for 13 million input points, the performances on both resources were in the same neighborhood for resolutions of 0.5m and 1m.

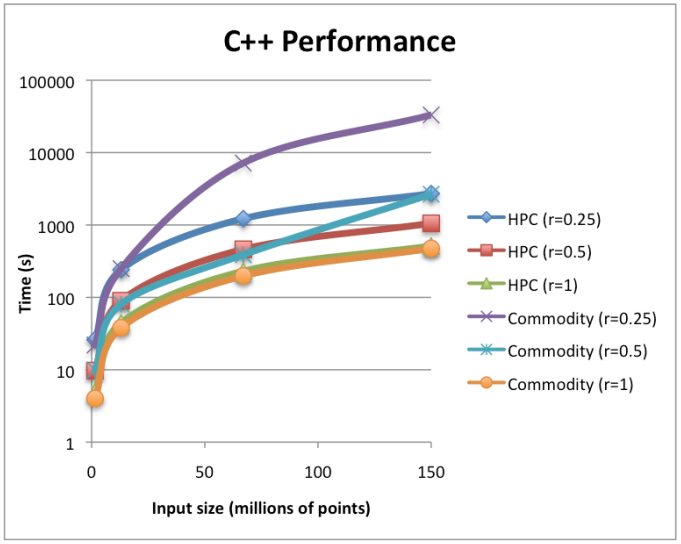

Figure 6. C++ performance comparison of commodity versus HPC nodes. Performance on the HPC node is about the same for smaller jobs, but an order of magnitude faster for the largest jobs. Grid resolution is represented by "r".

The performance on the HPC resource is significantly better once the implementation goes out of out of core on the commodity resource – i.e. for DEMs that contain more than 140 million grid cells. For instance, the 150 million point input file at a 0.25m resolution produces around 595 million grid cells – and the execution time on the HPC node was around 2,711s, while that on the commodity resource was around 33,113s. In other words, the DEM generation for the largest grid was more than 12 times faster on the HPC resource, in comparison to the commodity resource. The memory availability is the defining factor in the performance of the C++ implementation for large grids – hence; the HPC resource could run larger jobs much faster than the commodity resource.

C. Hadoop Implementation

Next, we evaluated the performance of our Hadoop implementation on our commodity cluster, by varying some of the parameters, such as the number of nodes in the Hadoop cluster, and the number of Reducers. The purpose of this experiment was to evaluate if the Hadoop implementation running on an inexpensive cluster put together out of commodity off-the-shelf components, compared favorably to the C++ performance on the HPC resource.

We used the Apache Hadoop version 0.20.2 with the default parameters – no attempts were made to optimize or tune the parameters for each of the resources. The replication parameter for the Hadoop Distributed File System (HDFS [11]) was set to two on both systems – i.e. the data are replicated twice.

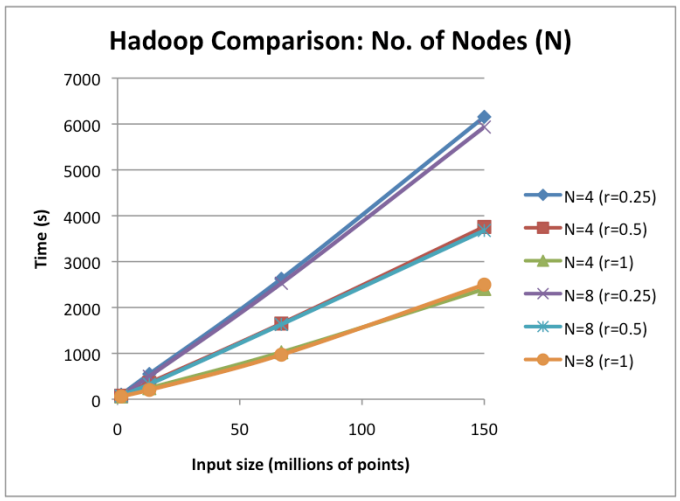

Figure 7. Hadoop perfortmance comparison on commodity cluster: 4 vs 8 nodes. The Hadoop performance is not significantly different between 4 or 8 nodes. Grid resolution is represented by "r".

Figure 7 shows the performance of our Hadoop implementation on our commodity cluster, with 4 and 8 nodes. Two characteristics of this graph deserve attention. First, the execution time using either 4 or 8 nodes was significantly less than that of the (single-node) C++ implementation on the same resource for large jobs. For example, our largest job (150 million points at a resolution of 0.25m) took around 5,933s to run on the 8-node Hadoop cluster, and 6,156s on the 4-node cluster. The Hadoop runs were around 5.5 times as fast as the C++ run on the same resource. However, neither run was faster than the C++ run on the HPC resource for the same inputs (2,711s).

The second characteristic to note from the figure is that the performance for both the 4 and 8 node implementations were very similar. This is because of two factors – first, the serial output generation step, which converts the Hadoop outputs ($\{Y, X, Z\}$ tuples) to the ArcASCII format, dominates the overall performance. For instance, the output generation time for 150 million input points at a grid resolution of 0.25m was found to be 2,878s, out of a total execution time of 6,156s (i.e. 47% of total time). Secondly, the number of Reducers chosen by the Hadoop framework for these runs defaulted to 1. The reduce phase is time consuming since all the Z values are computed from the intermediate key/value pairs in this phase – use of a single Reducer deprives us of the parallelism that can be achieved in the computation of the Z values.

Since the number of Reducers for the Hadoop implementation can be modified, we performed some experiments by varying their number. Figure 8 shows the performance of the Hadoop implementation with one and four Reducers, on a 4-node cluster. As seen in the graph, the performance was also very similar for one or four Reducers. There is indeed some parallelism that is gained by increasing the number of Reducers, but the benefit is counteracted by the necessity to merge and sort the outputs from the Reducers, which is needed for DEM generation in the ArcASCII format. Furthermore, as discussed earlier, the output generation time is still the dominating factor in the overall execution time.

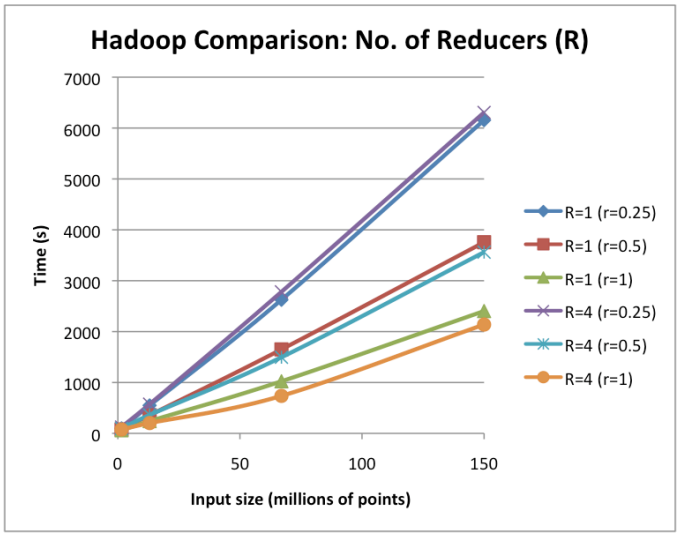

Figure 8. Hadoop performance comparison on commodity cluster: 1 vs 4 Reducers on 4 nodes. The performance with four reducers is worse than the performance with a single reducer for very small and very large grids. Grid resolution is represented by "r"..

It is worthwhile to note a few other characteristics of Figure 8. The performance of the run with a single Reducer was slightly better than the one with four Reducers in a couple of instances – when the inputs and grid resolutions are very low, and when they are very high. The performance of four Reducers was better in all other cases. This is because in the first case, the time required for bootstrapping the extra Reducers was greater than the benefit that could be gained due to the parallelism. And in the second case, the time required to sort and merge the output from the Reducers nullified the speedup achieved due to the parallelism. Although it may be possible to choose an optimal number of Reducers for the best possible execution time, the benefits were not significant enough to justify any further investigation for this particular application.

In summary, the Hadoop-based implementation provided significant speedup on the commodity resource, but was still slower than the C++ implementation on the single HPC node. Even though the algorithm was parallelizable, the serial output generation step was found to be the bottleneck for large grids.

V. Discussion

As our experiments have shown, the performance of both the Hadoop and C++ implementations depend upon a number of factors, including the size of the input point cloud dataset, and the resolution of the output grid. For the C++ implementation, the grid resolution is the dominating factor because it determines whether the resulting grid can be stored in memory or not. When the code switches to out of core, the performance is significantly worse because grid blocks may need to be swapped to and from secondary storage. For the

Hadoop implementation, the grid resolution is also a factor, but has less of an impact because *the number of intermediate key value pairs after the Map phase is equal to the total number of points in the input point cloud, irrespective of the grid size*. The bottleneck for the Hadoop implementation, however, appears to be the output generation phase, which unfortunately cannot be parallelized.

In general, if the entire grid can be fit in memory, the C++ implementation significantly outperforms the multi-node Hadoop implementation. While this might make the case for large memory systems, they can also be expensive. The cost of a single node on the HPC resource is around $30-$70K USD. On the other hand, the cumulative costs of the 4-node commodity Hadoop cluster was only around $4K. Thus, while the performances on HPC versus commodity hardware are in the same order of magnitude, the cost of the HPC node is an order of magnitude greater than the commodity Hadoop cluster. This means that the price performance ratio for our application is an order of magnitude better on the commodity cluster, in comparison to the HPC resource.

In terms of implementation effort, the Hadoop-based version of our algorithm is significantly easier to implement than the C++ version. This is because we only have to write two core functions for the Hadoop implementation – the Map and Reduce. Hadoop takes care of the partitioning the data into multiple nodes (via HDFS), and executing the algorithm in parallel. As for the C++ implementation, we have to perform the memory management by hand, leading to relatively complex code. The C++ implementation, including both the in-core and out-of-core versions of the algorithm, is around 2900 lines of code. The Hadoop based implementation, including the output generation, is around 700 lines of code.

In summary, for a similar class of applications, we recommend that traditional HPC machines be used if raw performance is desired. If cost or accessibility of such resources is a factor, then a Hadoop-based implementation on commodity clusters can be an option, since it provides performance of a similar order of magnitude as compared to traditional HPC resources, at a significantly lower cost. Also note that it might require significantly more effort to write HPC versions of such codes because of the overhead required to manage memory and optimize implementation, leading to increased personnel costs as well.

VI. Ongoing and Future Work

As part of our ongoing work, we are investigating the deployment of "Hadoop On-demand" on traditional HPC resources. The installation of Hadoop on HPC resources comes with its own set of challenges. In particular, access to the nodes on such resources is via batch scheduling systems, such as PBS [10]. Logging on to the individual nodes via secure shell (ssh) is prohibited – instead, jobs must be launched via the batch queuing interface provided by PBS. This means that the Hadoop configuration and setup has to be done "on-demand" via PBS scripts. The scripts must perform a set of pre-processing steps before any Hadoop code is run – including staging of all prerequisite software on to the compute nodes, formatting the Hadoop Distributed File System (HDFS), configuring the master and slave nodes, and starting all Hadoop daemons. Once everything is set up, the input files must be copied over to HDFS from a high performance shared file system, such as Lustre [12]. After Hadoop processing is complete, the outputs must be copied back over to the shared file system. Since there is a lot of overhead involved in the pre- and post-processing, we plan on running experiments to measure these overheads, and find out whether running Hadoop on an HPC resource is a worthwhile approach.

We are also investigating the use of a traditional parallel HPC approach for the local gridding algorithm in C++, using the Message Passing Interface (MPI) [14]. Apart from comparing the performance of the C++ implementation against the Hadoop implementation, we will investigate scaling, fault tolerance and software development time for both implementations. We are also looking into reducing the I/O overhead of the implementations by reading the input point cloud data in the ASPRS LAS binary format [15], which is the industry standard for LIDAR data exchange. In addition, we are investigating other algorithms for the generation of Digital Elevation Models, such as streaming TIN [16], and investigating whether they can be parallelized using MapReduce or MPI technologies.

Finally, we are also evaluating the use of User Defined Functions (UDF) for an alternative DEM implementation, running on the multi-node partitioned IBM DB2 database [13], which hosts the LIDAR point cloud datasets for the OpenTopography Facility [24]. Pushing the DEM generation into the database, where the points are hosted, could potentially improve the performance of our overall workflow because the datasets do not have to be exported to the file system to be later processed by the C++ and the Hadoop codes.

VII. Conclusions

In this paper, we investigated the use of MapReduce technology for a local gridding algorithm for the generation of Digital Elevation Models (DEM) being used by the NSF-funded OpenTopography Facility. We compared the traditional C++ implementation of this gridding algorithm to a MapReduce-based implementation, and presented our observations on the performance (in particular, price/performance) and implementation complexity. In general, we discovered that the MapReduce version was easier to implement than the C++ version, and provided a significant performance boost over the C++ version running on a commodity resource. We also found that the single-node C++ implementation on a traditional HPC resource, having access to significantly greater memory, out-performed the multi-node Hadoop implementation on the commodity resources for large jobs. However, the HPC resource costs significantly more, thus leading to lower price performance ratio than the commodity cluster. In general, depending on the budgets, accessibility and need for raw performance, we believe that both approaches may be applicable for different sets of users, for other applications in the same class.

VIII. ACKNOWLEDGMENTS

We acknowledge Han Kim and Ramon Arrowsmith for their efforts in designing the original C++ version of the local gridding algorithm. We also acknowledge Mahidhar Tatineni for his help with the work on Hadoop on-demand for HPC clusters, and the OpenTopography development team, specifically Charles Cowart and Viswanath Nandigam, for their ongoing efforts on the UDF-based DEM implementation on the IBM DB2 database.

REFERENCES

[1] J. Dean, and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Communications of the ACM, vol. 51, no. 1, pp. 107-113, 2008

[2] Apache Software Foundation, "Hadoop MapReduce Framework", http://hadoop.apache.org/mapreduce/, 2010

[3] Nokia Research Center, "Disco MapReduce Framework", http://discoproject.org/, 2010

[4] Greenplum, "Greenplum MapReduce: Bringing Next-Generation Analytics Technology to the Enterprise" http://www.greenplum.com/resources/mapreduce/, 2010

[5] Aster Data Systems, Inc., "In-Database MapReduce for Rich Analytics", http://www.asterdata.com/product/mapreduce.php, 2010

[6] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis", Proceedings of the 35th SIGMOD International Conference on Management of Data, 2009

[7] ESRI, "The ESRI Grid Format", http://en.wikipedia.org/wiki/ESRI_grid, 2010

[8] H. Kim, et al, "An Efficient Implementation of a Local Binning Algorithm for Digital Elevation Model Generation of LiDAR/ALSM Dataset", Eos Trans. AGU, 87(52), Fall Meet. Suppl., Abstract G53C-0921, 2006

[9] El-Sheimy, N, Valeo, C., and Habib, A., "Digital terrain modeling: acquisition, manipulation, and applications", Artech House: Boston, MA, 257pp, 2005

[10] Cluster Resources Inc, "The TORQUE Resource Manager", http://www.clusterresources.com/products/torque-resource-manager.php, 2010

[11] D. Borthakur, "The Hadoop Distributed File System: Architecture and Design", http://hadoop.apache.org/core/docs/current/hdfs_design.pdf, 2007

[12] P. Schwan, "Lustre: Building a File System for 1,000-node Clusters", Proceedings of the Linux Symposium, 2003

[13] IBM DB2 Information Center, http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp, 2010

[14] W. Gropp, E. Lusk, and A. Skjellum, "Using MPI: Portable Parallel Programming with the Message-Passing Interface", Cambridge: MIT Press, 1994

[15] LAS Industry Initiative, "Common LIDAR data exchange format," http://www.asprs.org/society/committees/LIDAR/LIDAR_format.html

[16] M. Isenburg, Y. Liu, J. Shewchuk, J. Snoeyink, and T. Thirion, "Generating Raster DEM from Mass Points via TIN Streaming", GIScience'06 Conference Proceedings, pp. 186-198, 2006.

[17] Carter, W.E., Shrestha, R.L., and Slatton, K.C., "Geodetic Laser Scanning", Phys. Today 60, 41, 2007

[18] El-Sheimy, N, Valeo, C., and Habib, A., "Digital terrain modeling: acquisition, manipulation, and applications", ArtechHouse: Boston, MA, pp 257, 2005

[19] J. S. Sarma, "Hadoop", Facebook Engineering Note, http://www.facebook.com/note.php?note_id=16121578919, June 2008

[20] Yahoo Inc, "Hadoop at Yahoo!", http://developer.yahoo.com/hadoop/, 2010.

[21] L. A. Barroso, J. Dean and U. Holzle, "Web search for a planet: The google cluster architecture", Micro IEEE, 2003.

[22] M. L. Stein, "Interpolation of spatial data: some theory for kriging", Springer Verlag, 1999

[23] Apache Hadoop Wiki, "Hadoop Streaming", http://wiki.apache.org/hadoop/HadoopStreaming, 2010

[24] V, Nandigam, C. Baru, C., and C. J. Crosby, "Database Design for High-Resolution LIDAR Topography Data", in SSDBM 2010, Lecture Notes in Computer Science 6187, pp. 151-159, 2010