From Point Cloud to Grid DEM: A Scalable Approach

Pankaj K. Agarwal [⋆1], Lars Arge [⋆⋆12], and Andrew Danner [⋆⋆⋆1]

[1] Department of Computer Science, Duke University, Durham, NC 27708, USA
{pankaj, large, adanner}@cs.duke.edu
[2] Department of Computer Science, University of Aarhus, Aarhus, Denmark
large@daimi.au.dk

Summary. Given a set $\mathcal{S}$ of points in $\mathbb{R}^3$ sampled from an *elevation* function $H : \mathbb{R}^2 \to \mathbb{R}$, we present a scalable algorithm for constructing a grid digital elevation model (DEM). Our algorithm consists of three stages: First, we construct a quad tree on $\mathcal{S}$ to partition the point set into a set of non-overlapping segments. Next, for each segment q, we compute the set of points in q and all segments neighboring q. Finally, we interpolate each segment independently using points within the segment and its neighboring segments.

Data sets acquired by LIDAR and other modern mapping technologies consist of hundreds of millions of points and are too large to fit in main memory. When processing such massive data sets, the transfer of data between disk and main memory (also called I/O), rather than the CPU time, becomes the performance bottleneck. We therefore present an *I/O-efficient* algorithm for constructing a grid DEM. Our experiments show that the algorithm scales to data sets much larger than the size of main memory, while existing algorithms do not scale. For example, using a machine with 1GB RAM, we were able to construct a grid DEM containing 1.3 billion cells (occupying 1.2GB) from a LIDAR data set of over 390 million points (occupying 20GB) in about 53 hours. Neither ArcGIS nor GRASS, two popular GIS products, were able to process this data set.

1 Introduction

One of the basic tasks of a geographic information system (GIS) is to store a representation of various physical properties of a terrain such as elevation, temperature, precipitation, or water depth, each of which can be viewed as a real-valued bivariate function. Because of simplicity and efficacy, one of the widely used representations is the so-called grid representation in which a functional value is stored in each cell of a two-dimensional uniform grid. However, many modern mapping technologies

⋆ Supported by NSF under grants CCR-00-86013, EIA-01-31905, CCR-02-04118, and DEB-04-25465, by ARO grants W911NF-04-1-0278 and DAAD19-03-1-0352, and by a grant from the U.S.–Israel Binational Science Foundation.
⋆⋆ Supported in part by the US Army Research Office through grant W911NF-04-1-0278 and by an Ole Rømer Scholarship from the Danish National Science Research Council.
⋆⋆⋆ Supported by the US Army Research Office through grant W911NF-04-1-0278.

do no acquire data on a uniform grid. Hence the raw data is a set $\mathcal{S}$ of N (arbitrary) points in $\mathbb{R}^3$, sampled from a function $H : \mathbb{R}^2 \rightarrow \mathbb{R}$. An important task in GIS is thus to interpolate $\mathcal{S}$ on a uniform grid of a prescribed resolution.

In this paper, we present a scalable algorithm for this interpolation problem. Although our technique is general, we focus on constructing a grid digital elevation model (DEM) from a set $\mathcal{S}$ of N points in $\mathbb{R}^3$ acquired by modern mapping techniques such as LIDAR. These techniques generate huge amounts of high-resolution data. For example, LIDAR[3] acquires highly accurate elevation data at a resolution of one point per square meter or better and routinely generates hundreds of millions of points. It is not possible to store these massive data sets in the internal memory of even high-end machines, and the data must therefore reside on larger but considerably slower disks. When processing such huge data sets, the transfer of data between disk and main memory (also called *I/O*), rather than computation, becomes the performance bottleneck. An *I/O-efficient* algorithm that minimizes the number of disk accesses leads to tremendous runtime improvements in these cases. In this paper we develop an I/O-efficient algorithm for constructing a grid DEM of unprecedented size from massive LIDAR data sets.

Related work. A variety of methods for interpolating a surface from a set of points have been proposed, including inverse distance weighting (IDW), kriging, spline interpolation and minimum curvature surfaces. Refer to [12] and the references therein for a survey of the different methods. However, the computational complexity of these methods often make it infeasible to use them directly on even moderately large points sets. Therefore, many practical algorithms use a segmentation scheme that decomposes the plane (or rather the area of the plane containing the input points) into a set of *non-overlapping areas* (or *segments*), each containing a small number of input points. One then interpolates the points in each segment independently. Numerous segmentation schemes have been proposed, including simple regular decompositions and decompositions based on Voronoi diagrams [18] or quad trees [14, 11]. A few schemes using *overlapping* segments have also been proposed [19, 17].

As mentioned above, since I/O is typically the bottleneck when processing large data sets, I/O-efficient algorithms are designed to explicitly take advantage of the large main memory and disk block size [2]. These algorithms are designed in a model in which the computer consists of an internal (or main) memory of size M and an infinite external memory. Computation is considered free but can only occur on elements in main memory; in one *I/O-operation*, or simply *I/O*, B consecutive elements can be transfered between internal and external memory. The goal of an I/O-efficient algorithm is to minimize the number of I/Os.

Many $\Theta(N)$ time algorithms that do not explicitly consider I/O use $\Theta(N)$ I/Os when used in the I/O-model. However, the "linear" bound, the number of I/Os needed to read N elements, is only $\Theta(\mathrm{scan}(N)) = \Theta(\frac{N}{B})$ in the I/O model. The number of

[3] In this paper, we consider LIDAR data sets that represent the actual terrain and have been pre-processed by the data providers to remove spikes and errors due to noise.

I/Os needed to sort N elements is $\Theta(\mathrm{sort}(N)) = \Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ [2]. In practice, B is on the order of 10^3–10^5, so $\mathrm{scan}(N)$ and $\mathrm{sort}(N)$ are typically much smaller than N. Therefore tremendous speedups can often be obtained by developing algorithms that use $O(\mathrm{scan}(N))$ or $O(\mathrm{sort}(N))$ I/Os rather than $\Omega(N)$ I/Os. Numerous I/O-efficient algorithms and data structures have been developed in recent years, including several for fundamental GIS problems (refer to [4] and the references therein for a survey). Agarwal et. al [1] presented a general top-down layered framework for constructing a certain class of spatial data structures–including quad trees–I/O-efficiently. Hjaltason and Samet [9] also presented an I/O-efficient quad-tree construction algorithm. This optimal $O(\mathrm{sort}(N))$ I/O algorithm is based on assigning a Morton block index to each point in $\mathcal{S}$, encoding its location along a Morton-order (Z-order) space-filling curve, sorting the points by this index, and then constructing the structure in a bottom-up manner.

Our approach. In this paper we describe an I/O-efficient algorithm for constructing a grid DEM from LIDAR points based on quad-tree segmentation. Most of the segmentation based algorithms for this problem can be considered as consisting of three separate phases; the *segmentation* phase, where the decomposition is computed based on $\mathcal{S}$; the *neighbor finding* phase, where for each segment in the decomposition the points in the segment and the relevant neighboring segments are computed; and the *interpolation* phase, where a surface is interpolated in each segment and the interpolated values of the grid cells in the segment are computed. In this paper, we are more interested in the segmentation and neighbor finding phases than the particular interpolation method used in the interpolation phase. We will focus on the quad tree based segmentation scheme because of its relative simplicity and because it has been used with several interpolation methods such as thin plate splines [14] and B-splines [11]. We believe that our techniques will apply to other segmentation schemes as well.

Our algorithm implements all three phases I/O-efficiently, while allowing the use of any given interpolation method in the interpolation phase. Given a set $\mathcal{S}$ of N points, a desired output grid specified by a bounding box and a cell resolution, as well as a threshold parameter $k_{\max}$, the algorithm uses $O(\frac{N}{B}\frac{h}{\log\frac{M}{B}} + \mathrm{sort}(T))$ I/Os, where h is the height of a quad tree on $\mathcal{S}$ with at most $k_{\max}$ points in each leaf, and T is the number of cells in the desired grid DEM. Note that this is $O(\mathrm{sort}(N) + \mathrm{sort}(T))$ I/Os if $h = O(\log N)$, that is, if the points in $\mathcal{S}$ are distributed such that the quad tree is roughly balanced.

The three phases of our algorithm are described in Section 2, Section 3 and Section 4. In Section 2 we describe how to construct a quad tree on $\mathcal{S}$ with at most $k_{\max}$ points in each leaf using $O(\frac{N}{B}\frac{h}{\log\frac{M}{B}})$ I/Os. The algorithm is based on the framework of Agarwal et. al [1]. Although not as efficient as the algorithm by Hjaltason and Samet [9] in the worst case, we believe that it is simpler and potentially more practical; for example, it does not require computation of Morton block indices or sorting of the input points. Also in most practical cases where $\mathcal{S}$ is relatively nicely distributed, for example when working with LIDAR data, the two algorithms both

use $O(\mathrm{sort}(N))$ I/Os. In Section 3 we describe how to find the points in all neighbor leaves of each quad-tree leaf using $O(\frac{N}{B}\frac{h}{\log\frac{M}{B}})$ I/Os. The algorithm is simple and very similar to our quad-tree construction algorithm; it takes advantage of how the quad tree is naturally stored on disk during the segmentation phase. Note that while Hjaltason and Samet [9] do not describe a neighbor finding algorithm based on their Morton block approach, it seems possible to use their approach and an I/O-efficient priority queue [5] to obtain an $O(\mathrm{sort}(N))$ I/O algorithm for the problem. However, this algorithm would be quite complex and therefore probably not of practical interest. Finally, in Section 4 we describe how to apply an interpolation scheme to the points collected for each quad-tree leaf, evaluate the computed function at the relevant grid cells within the segment corresponding to each leaf, and construct the final grid using $O(\mathrm{scan}(N)) + O(\mathrm{sort}(T))$ I/Os. As mentioned earlier, we can use any given interpolation method within each segment.

To investigate the practical efficiency of our algorithm we implemented it and experimentally compared it to other interpolation algorithms using LIDAR data. To summarize the results of our experiments, we show that, unlike existing algorithms, our algorithm scales to data sets much larger than the main memory. For example, using a 1GB machine we were able to construct a grid DEM containing 1.3 billion points (occupying 1.2GB) from a LIDAR data set of over 390 million points (occupying 20GB) in just 53 hours. This data set is an order of magnitude larger than what could be handled by two popular GIS products–ArcGIS and GRASS. In addition to supporting large input point sets, we were also able to construct very large high resolution grids; in one experiment we constructed a one meter resolution grid DEM containing more than 53 billion cells—storing just a single bit for each grid cell in this DEM requires 6GB.

In Section 5 we describe the details of the implementation of our theoretically I/O-efficient algorithm that uses a regularized spline with tension interpolation method [13]. We also describe the details of an existing algorithm implemented in GRASS using the same interpolation method; this algorithm is similar to ours but it is not I/O-efficient. In Section 6 we describe the results of the experimental comparison of our algorithm to other existing implementations. As part of this comparison, we present a detailed comparison of the quality of the grid DEMs produced by our algorithm and the similar algorithm in GRASS that show the results are in good agreement.

2 Segmentation Phase: Quad-Tree Construction

Given a set $\mathcal{S}$ of N points contained in a bounding box $[x_1, x_2] \times [y_1, y_2]$ in the plane, and a threshold $k_{\max}$, we wish to construct a quad tree $\mathcal{T}$ [8] on $\mathcal{S}$ such that each quad-tree leaf contains at most $k_{\max}$ points. Note that the leaves of $\mathcal{T}$ partition the bounding box $[x_1, x_2] \times [y_1, y_2]$ into a set of disjoint areas, which we call *segments*.

Incremental construction. $\mathcal{T}$ can be constructed incrementally simply by inserting the points of $\mathcal{S}$ one at a time into an initially empty tree. For each point p,

we traverse a root-leaf path in $\mathcal{T}$ to find the leaf v containing p. If v contains less than $k_{\max}$ points, we simply insert p in v. Otherwise, we split v into four new leaves, each representing a quadrant of v, and re-distribute p and the points in v to the new leaves. If h is the height of $\mathcal{T}$, this algorithm uses $O(Nh)$ time. If the input points in $\mathcal{S}$ are relatively evenly distributed we have $h = O(\log N)$, and the algorithm uses $O(N \log N)$ time.

If $\mathcal{S}$ is so large that $\mathcal{T}$ must reside on disk, traversing a path of length h may require as many as h I/Os, leading to an I/O cost of $O(Nh)$ in the I/O-model. By storing (or *blocking*) the nodes of $\mathcal{T}$ on disk intelligently, we may be able to access a subtree of depth $\log B$ (size B) in a single I/O and thus reduce the cost to $O(N \frac{h}{\log B})$ I/Os. Caching the top-most levels of the tree in internal memory may also reduce the number of I/Os needed. However, since not all the levels fit in internal memory, it is hard to avoid spending an I/O to access a leaf during each insertion, or $\Omega(N)$ I/Os in total. Since $\mathrm{sort}(N) \ll N$ in almost all cases, the incremental approach is very inefficient when the input points do not fit in internal memory.

Level-by-level construction. A simple I/O-efficient alternative to the incremental construction algorithm is to construct $\mathcal{T}$ level-by-level: We first construct the first level of $\mathcal{T}$, the root v, by scanning through $\mathcal{S}$ and, if $N > k_{\max}$, distributing each point p to one of four leaf lists on disk corresponding to the child of v containing p. Once we have scanned $\mathcal{S}$ and constructed one level, we construct the next level by loading each leaf list in turn and constructing leaf lists for the next level of $\mathcal{T}$. While processing one list we keep a buffer of size B in memory for each of the four new leaf lists (children of the constructed node) and write buffers to the leaf lists on disk as they run full. Since we in total scan $\mathcal{S}$ on each level of $\mathcal{T}$, the algorithm uses $O(Nh)$ time, the same as the incremental algorithm, but only $O(Nh/B)$ I/Os. However, even in the case of $h = \log_4 N$, this approach is still a factor of $\log_{\frac{M}{B}} \frac{N}{B} / \log_4 N$ from the optimal $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/O bound.

Hybrid construction. Using the framework of Agarwal et. al [1], we design a hybrid algorithm that combines the incremental and level-by-level approaches. In-

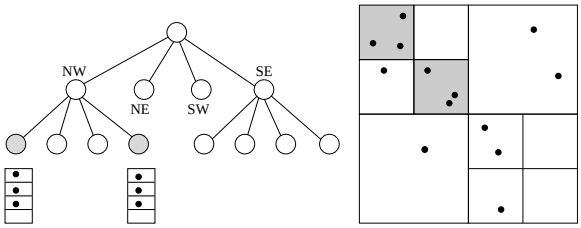


Fig. 1. Construction of a quad-tree layer of depth three with $k_{\max} = 2$. Once a leaf at depth three is created, no further splitting is done; instead additional points in the leaf are stored in leaf lists shown below shaded nodes. After processing all points the shaded leaves with more than two points are processed recursively.

stead of constructing a single level at a time, we can construct a *layer* of $\log_4 \frac{M}{B}$ levels. Because $4^{\log_4 \frac{M}{B}} = M/B < M$, we construct the layer entirely in internal memory using the incremental approach: We scan through $\mathcal{S}$, inserting points one at a time while splitting leaves and constructing new nodes, except if the path from the root of the layer to a leaf of the layer is of height $\log_4 \frac{M}{B}$. In this case, we write all points contained in such a leaf v to a list L_v on disk. After all points have been processed and the layer constructed, we write the layer to disk sequentially and recursively construct layers for each leaf list L_i. Refer to Figure 1.

Since a layer has at most M/B nodes, we can keep a internal memory buffer of size B for each leaf list and only write points to disk when a buffer runs full (for leaves that contain less than B points in total, we write the points in all such leaves to a single list after constructing the layer). In this way we can construct a layer on N points in $O(N/B) = \mathrm{scan}(N)$ I/Os. Since a tree of height h has $h/\log_4 \frac{M}{B}$ layers, the total construction cost is $O(\frac{N}{B} \frac{h}{\log \frac{M}{B}})$ I/Os. This is $\mathrm{sort}(N) = O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os when $h = O(\log N)$.

3 Neighbor Finding Phase

Let $\mathcal{T}$ be a quad tree on $\mathcal{S}$. We say that two leaves are neighbors if their associated segments share part of an edge or a corner. Refer to Figure 2 for an example. If $\mathcal{L}$ is the set of segments associated with the leaves of $\mathcal{T}$, we want to find for each $q \in \mathcal{L}$ the set $\mathcal{S}_q$ of points contained in q and the neighbor leaves of q. As for the construction algorithm, we first describe an incremental algorithm and then improve its efficiency using a layered approach.

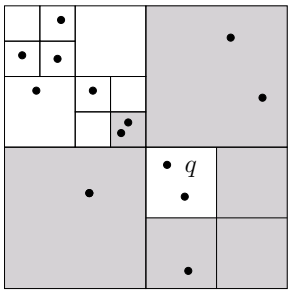


Fig. 2. The segment q associated with a leaf of a quad tree and its six shaded neighboring segments.

Incremental approach. For each segment $q \in \mathcal{L}$, we can find the points in the neighbors of q using a simple recursive procedure: Starting at the root v of $\mathcal{T}$, we compare q to the segments associated with the four children of v. If the bounding box of a child u shares a point or part of an edge with q, then q is a neighbor of at

least one leaf in the tree rooted in u; we therefore recursively visit each child with an associated segment that either neighbors or contains q. When we reach a leaf we insert all points in the leaf in $\mathcal{S}_q$.

To analyze the algorithm, we first bound the total number of neighbor segments found over all segments $q \in \mathcal{L}$. Consider the number of neighbor segments that are at least the same size as a given segment q; at most one segment can share each of q's four edges, and at most four more segments can share the four corner points of q. Thus, there are at most eight such neighbor segments. Because the neighbor relation is symmetric, the total number of neighbor segments over all segments is at most twice the total number of neighbor segments which are at least the same size. Thus the total number of neighbor segments over all segments is at most 16 times the number of leaves of $\mathcal{T}$. Because the total number of leaves is at most $4N/k_{\max}$, and since the above algorithm traverses a path of height h for each neighbor, it visits $O(Nh)$ nodes in total. Furthermore, as each leaf contains at most $k_{\max}$ points, the algorithm reports $O(Nk_{\max})$ points in total. Thus the total running time of the algorithm is $O((h + k_{\max})N) = O(hN)$). This is also the worst case I/O cost.

Layered approach. To find the points in the neighboring segments of each segment in $\mathcal{L}$ using a layered approach similar to the one used to construct $\mathcal{T}$, we first load the top $\log_4 \frac{M}{B}$ levels of $\mathcal{T}$ into memory. We then associate with each leaf u in the layer, a buffer B_u of size B in internal memory and a list L_u in external memory. For each segment $q \in \mathcal{L}$, we use the incremental algorithm described above to find the leaves of the layer with an associated segment that completely contains q or share part of a boundary with q. Suppose u is such a layer leaf. If u is also a leaf of the entire tree $\mathcal{T}$, we add the pair $(q, \mathcal{S}_u)$ to a global list Λ, where $\mathcal{S}_u$ is the set of points stored at u. Otherwise, we add q to the buffer B_u associated with u, which is written to L_u on disk when B_u runs full. After processing all segments in $\mathcal{L}$, we recursively process the layers rooted at each leaf node u and its corresponding list L_u. Finally, after processing all layers, we sort the global list of neighbor points Λ by the first element q in the pairs $(q, \mathcal{S}_u)$ stored in Λ. After this, the set $\mathcal{S}_q$ of points in the neighboring segments of q are in consecutive pairs of Λ, so we can construct all $\mathcal{S}_q$ sets in a simple scan of Λ.

Since we access nodes in $\mathcal{T}$ during the above algorithm in the same order they were produced in the construction of $\mathcal{T}$, we can process each layer of $\log_4 \frac{M}{B}$ levels of $\mathcal{T}$ in $\mathrm{scan}(N)$ I/Os. Furthermore, since $\sum_q |\mathcal{S}_q| = O(N)$, the total number of I/Os used to sort and scan Λ is $O(\mathrm{sort}(N))$. Thus the algorithm uses $O(\frac{N}{B} \frac{h}{\log \frac{M}{B}})$ I/Os in total, which is $O(\mathrm{sort}(N))$ when $h = O(\log N)$.

4 Interpolation Phase

Given the set $\mathcal{S}_q$ of points in each segment q (quad tree leaf area) and the neighboring segments of q, we can perform the interpolation phase for each segment q in turn simply by using any interpolation method we like on the points in $\mathcal{S}_q$, and evaluating the

computed function to interpolate each of the grid cells in q. Since $\sum_q |\mathcal{S}_q| = O(N)$, and assuming that each $\mathcal{S}_q$ fits in memory (otherwise we maintain a internal memory priority queue to keep the $n_{\max} < M$ points in $\mathcal{S}_q$ that are closest to the center of q, and interpolate on this subset), we can read each $\mathcal{S}_q$ into main memory and perform the interpolation in $O(\mathrm{scan}(N))$ I/Os in total. However, we cannot simply write the interpolated grid cells to an output grid DEM as they are computed, since this could result in an I/O per cell (or per segment q). Instead we write each interpolated grid cell to a list along with its position (i, j) in the grid; we buffer B cells at a time in memory and write the buffer to disk when it runs full. After processing each set $\mathcal{S}_q$, we sort the list of interpolated grid cells by position to obtain the output grid. If the output grid has size T, computing the T interpolated cells and writing them to the list takes $O(T/B)$ I/Os. Sorting the cells take $O(\mathrm{sort}(T))$ I/Os. Thus the interpolation phase is performed in $O(\mathrm{scan}(N) + \mathrm{sort}(T))$ I/Os in total.

5 Implementation

We implemented our methods in C++ using TPIE [7, 6], a library that eases the implementation of I/O-efficient algorithms and data structures by providing a set of primitives for processing large data sets. Our algorithm takes as input a set $\mathcal{S}$ of points, a grid size, and a parameter $k_{\max}$ that specifies the maximum number of points per quad tree segment, and computes the interpolated surface for the grid using our segmentation algorithm and a regularized spline with tension interpolation method [13]. We chose this interpolation method because it is used in the open source GIS GRASS module `s.surf.rst` [14]—the only GRASS surface interpolation method that uses segmentation to handle larger input sizes—and provides a means to compare our I/O-efficient approach to an existing segmentation method. Below we discuss two implementation details of our approach: thinning the input point set, and supporting a *bit mask*. Additionally, we highlight the main differences between our implementation and `s.surf.rst`.

Thinning point sets. Because LIDAR point sets can be very dense, there are often several cells in the output grid that contain multiple input points, especially when the grid cell size is large. Since it is not necessary to interpolate at sub-pixel resolutions, computational efficiency improves if one only includes points that are sufficiently far from other points in a quad-tree segment. Our implementation only includes points in a segment that are at least a user-specified distance ε from all other points within the segment. By default, ε is half the size of a grid cell. We implement this feature with no additional I/O cost simply by checking the distance between a new point p and all other points within the quad-tree leaf containing p and discarding p if it is within a distance ε of another point.

Bit mask. A common GIS feature is the ability to specify a bit mask that skips computation on certain grid cells. The bit mask is a grid of the same size as the output grid, where each cell has a zero or one bit value. We only interpolate grid cell

values when the bit mask for the cell has the value one. Bit masks are particularly useful when the input data set consists of an irregularly shaped region where the input points are clustered and large areas of the grid are far from the input points. Skipping the interpolation of the surface in these places reduces computation time, especially when many of the bit mask values are zero.

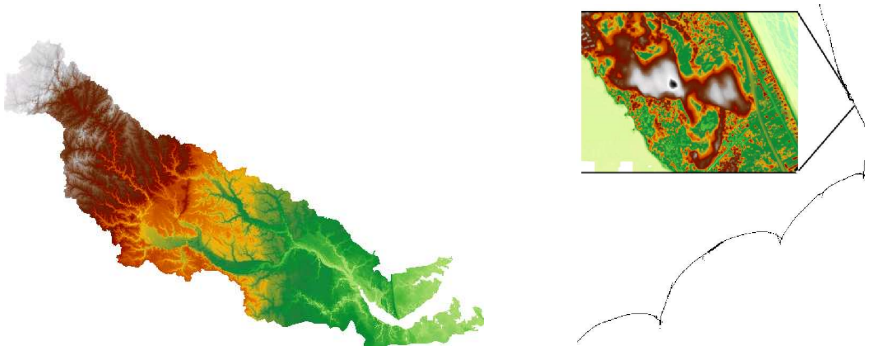
For high resolution grids, the number of grid cells can be very large, and the bit mask may be larger than internal memory and must reside on disk. Randomly querying the bit mask for each output grid cell would be very expensive in terms of I/O cost. Using the same filtering idea described in Section 2 and Section 3, we filter the bit mask bits through the quad-tree layer by layer such that each quad-tree segment gets a copy of the bit mask bits it needs during interpolation. The algorithm uses $O(\frac{T}{B}\frac{h}{\log \frac{M}{B}})$ I/Os in total, where T is the number of cells in the output grid, which is $O(\mathrm{sort}(T))$ when $h = O(\log N)$. The bits for a given segment can be accessed sequentially as we interpolate each quad-tree segment.

GRASS Implementation. The GRASS module `s.surf.rst` uses a quad-tree segmentation, but is not I/O-efficient in several key areas which we briefly discuss; constructing the quad tree, supporting a bit mask, finding neighbors, and evaluating grid cells. All data structures in the GRASS implementation with the exception of the output grid are stored in memory and must use considerably slower swap space on disk if internal memory is exhausted. During construction points are simply inserted into an internal memory quad tree using the incremental construction approach of Section 2. Thinning of points using the parameter ε during construction is implemented exactly as our implementation. The bit mask in `s.surf.rst` is stored as a regular grid entirely in memory and is accessed randomly during interpolation of segments instead of sequentially in our approach.

Points from neighboring quad-tree segment are not found in advance as in our algorithm, but are found when interpolating a given quad-tree segment q; the algorithm creates a window w by expanding q in all directions by a width δ and querying the quad tree to find all points within w. The width δ is adjusted by binary search until the number of points within w is between a user specified range $[n_{\min}, n_{\max}]$. Once an appropriate number of points is found for a quad-tree segment q, the grid cells in q are interpolated and written directly to the proper location in the output grid by randomly seeking to the appropriate file offset and writing the interpolated results. When each segment has a small number of cells, writing the values of the T output grid cells uses $O(T) \gg \mathrm{sort}(T)$ I/Os. Our approach constructs the output grid using the significantly better $\mathrm{sort}(T)$ I/Os.

6 Experiments

We ran a set of experiments using our I/O-efficient implementation of our algorithm and compared our results to existing GIS tools. We begin by describing the data sets on which we ran the experiments, then compare the efficiency and accuracy of our algorithm with other methods. We show that our algorithm is scalable to over 395



(a) (b)

Fig. 3. (a) Neuse river basin data set and (b) Outer Banks data set, with zoom to very small region.

million points and over 53 billion output grid cells–well beyond the limits of other GIS tools we tested.

Experimental setup. We ran our experiments on an Intel 3.4GHz Pentium 4 hyper-threaded machine with 1GB of internal memory, over 4GB of swap space, and running a Linux 2.6 kernel. The machine had a pair of 400GB SATA disk drives in a non-RAID configuration. One disk stored the input and output data sets and the other disk was used for temporary scratch space.

For our experiments we used two large LIDAR data sets, freely available from online sources; one of the Neuse river basin from the North Carolina Floodmaps project [15] and one of the North Carolina Outer Banks from NOAA's Coastal Services Center [16].

Neuse river basin. This data set contains 500 million points, more than 20 GB of raw data; see Figure 3(a). The data have been pre-processed by the data providers to remove most points on buildings and vegetation. The average spacing between points is roughly 20ft.

Outer banks. This data set contains 212 million LIDAR points, 9 GB of raw data; see Figure 3(b). Data points are confined to a narrow strip (a zoom of a very small portion of the data set is shown in the figure). This data set has not been heavily pre-processed to remove buildings and vegetation. The average point spacing is roughly 3ft.

Scalability results. We ran our algorithm on both the Neuse river and Outer Banks data sets at varying grid cell resolutions. Because we used the default value of ε (half the grid cell size) increasing the size of grid cells decreased the number of points in the quad tree and the number of points used for interpolation. Results

are summarized in Table 1. In each test, the interpolation phase was the most time-consuming phase; interpolation consumed over 80% of the total running time on the Neuse river basin data set. For each test we used a bit mask to ignore cells more than 300ft from the input points. Because of the irregular shape of the Outer Banks data, this bit mask is very large, but relatively sparse (containing very few "1" bits). Therefore, filtering the bit mask and writing the output grid for the Outer Banks data were relatively time-consuming phases when compared to the Neuse river data. Note that the number of grid cells in the Outer Banks is roughly three orders of magnitude greater than the number of quad-tree points. As the grid cell size decreases and the total number of cells increases, bit mask and grid output operations consume a greater percentage of the total time. At a resolution of 5ft, the bit mask alone for the Outer Banks data set is over 6GB. Even at such large grid sizes, interpolation—an internal memory procedure—was the most time-consuming phase, indicating that I/O was not a bottleneck in our algorithm.

Dataset	Neuse		Outer Banks	
Resolution (ft)	20	40	5	10
Output grid cells ($\times 10^6$)	1360	340	53160	13402
quad-tree points ($\times 10^6$)	395	236	128	66
Total Time (hrs)	53.0	24.4	17.7	6.9
Time spent to... (%)				
Build tree	2.0	3.8	4.5	8.6
Find Neighbors	10.6	15.1	14.5	16.4
Filter Bit mask	0.2	0.3	13.1	8.0
Interpolate	86.4	80.4	52.6	57.8
Write Output	0.8	0.4	15.3	9.2

Table 1. Results from the Neuse river basin and the Outer Banks data sets.

We also tried to test other available interpolation methods, including `s.surf.rst` in the open source GIS GRASS; kriging, IDW, spline, and topo-to-raster (based on ANUDEM [10]) tools in ArcGIS 9.1; and QTModeler 4 from Applied Imagery [3]. Only `s.surf.rst` supported the thinning of data points based on cell size, so for the other programs we simply used a subset of the data points. None of the ArcGIS tools could process more than 25 million points from the Neuse river basin at 20ft resolution and every tool crashed on large input sizes. The topo-to-raster tool processed the largest set amongst the ArcGIS tools at 21 million points.

The `s.surf.rst` could not process more than 25 million points either. Using a resolution of 200ft, `s.surf.rst` could process the entire Neuse data set in six hours, but the quad tree only contained 17.4 million points. Our algorithm processed the same data set at 200ft resolution in 3.2 hours. On a small subset of the Outer Banks data set containing 48.8 million points, `s.surf.rst`, built a quad tree on 7.1 million points and computed the output grid DEM in three hours, compared to 49 minutes for our algorithm on the same data set.

The QTModeler program processed the largest data set amongst the other methods we tested, approximately 50 million points, using 1GB of RAM. The documentation for QTModeler states that their approach is based on an internal memory quad tree and can process 200 million points with 4GB of available RAM. We can process a data set almost twice as large using less than 1GB of RAM.

Overall, we have seen that our algorithm is scalable to very large point sets and very large grid sizes and we demonstrated that many of the commonly used GIS tools cannot process such large data sets. Our approach for building the quad tree and finding points in neighboring segments is efficient and never took more than 25% of the total time in any of our experiments. The interpolation phase, an internal step that reads points sequentially from disk and writes grid cells sequentially to disk, was the most time-consuming phase of the entire algorithm.

Comparison of constructed grids. To show that our method constructs correct output grids, we compared our output on the Neuse river basin to the original input points as well as to grid DEMs created by `s.surf.rst`, and DEMs freely available from NC Floodmaps. Because `s.surf.rst` cannot process very large data sets, we ran our tests on a small subset of the Neuse river data set containing 13 million points. The output resolution was 20ft, ε was set to the default 10ft, and the output grid had 3274 rows and 3537 columns for a total of 11.6 million cells. Approximately 11 million points were in the quad tree.

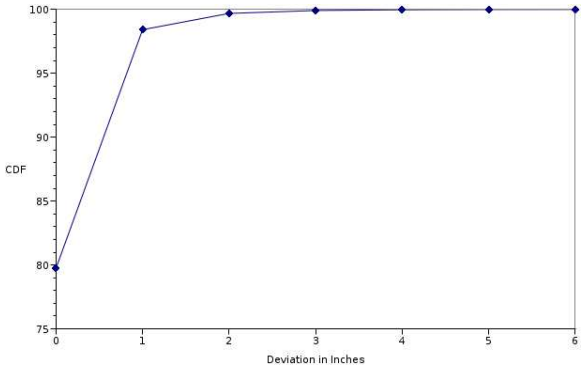
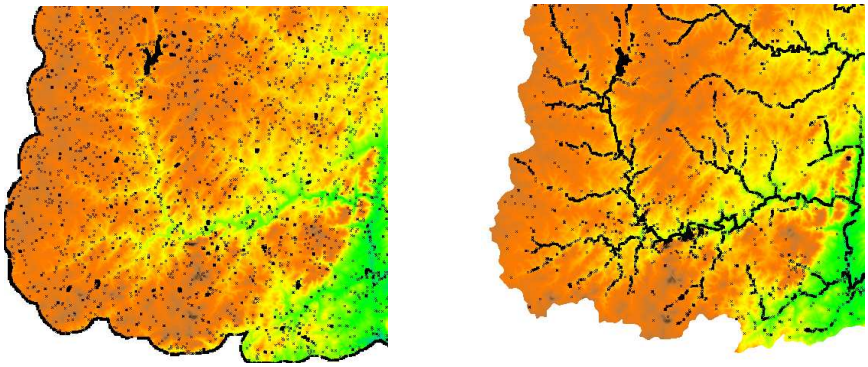


Fig. 4. Distribution of deviations between input points and interpolated surface ($k_{\max} = 35$).

The interpolation function we tested used a smoothing parameter and allowed the input points to deviate slightly from the interpolated surface. We used the same default smoothing parameter used in the GRASS implementation and compared the distribution of deviations between the input points and the interpolated surface. The results were independent of $k_{\max}$, the maximum number of points per quad-tree segment. In all tests, at least 79% of the points had no deviation, and over 98% of the points had a deviation of less than one inch. Results for `s.surf.rst` were similar.

Since the results were indistinguishable for various $k_{\max}$ parameters, we show only one of the cumulative distribution functions (CDF) for $k_{\max} = 35$ in Figure 4.

Next, we computed the absolute deviation between grid values computed using `s.surf.rst` and our method. We found that over 98% of the cells agreed within 1 inch, independent of $k_{\max}$. The methods differ slightly because `s.surf.rst` uses a variable size window to find points in neighboring points of a quad-tree segment q and may not choose all points from immediate neighbors of q when the points are dense and may expand the window to include points in segments that are not immediate neighbors of q when the points are sparse. In Figure 5(a) we show a plot of the interpolated surface along with an overlay of cells where the deviation exceeds 3 inches. Notice that most of the bad spots are along the border of the data set where our method is less likely to get many points from neighboring quad-tree leaves and near the lake in the upper left corner of the image where LIDAR signals are absorbed by the water and there are no input data points.



(a) (b)

Fig. 5. Interpolated surface generated by our method. Black dots indicate cells where the deviation between our method and (a) `s.surf.rst` is greater than three inches, (b) ncfloodmap data is greater than two feet.

Finally, we compared both our output and that of `s.surf.rst` to the 20ft DEM data available from the NC Floodmaps project. A CDF in Figure 6 of the absolute deviation between the interpolated grids and the "base" grid from NC Floodmaps shows that both implementations have an identical CDF curve. However, the agreement between the interpolated surfaces and the base grid is not as strong as the agreement between the algorithms when compared to each other. An overlay of regions with deviation greater than two feet on base map shown in Figure 5(b) reveals the source of the disagreement. A river network is clearly visible in the figure indicating that something is very different between the two data sets along the rivers. NC Floodmaps uses supplemental break-line data that is not part of the LIDAR point set

to enforce drainage and provide better boundaries of lakes in areas where LIDAR has trouble collecting data. Aside from the rivers, the interpolated surface generated by either our method or the existing GRASS implementation agree reasonably well with the professionally produced and publicly available base map.

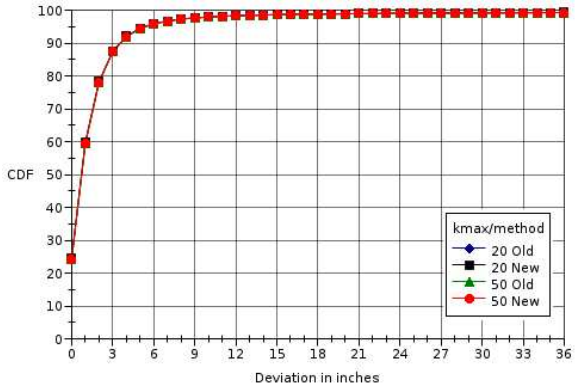


Fig. 6. Cumulative distribution of deviation between interpolated surface and data downloaded from ncfloodmaps.com. Deviation is similar for both our method and `s.surf.rst` for all values of $k_{\max}$.

7 Conclusions

In this paper we describe an I/O-efficient algorithm for constructing a grid DEM from point cloud data. We implemented our algorithm and, using LIDAR data, experimentally compared it to other existing algorithms. The empirical results show that, unlike existing algorithms, our approach scales to data sets much larger than the size of main memory. Although we focused on elevation data, our technique is general and can be used to compute the grid representation of any bivariate function from irregularly sampled data points.

For future work, we would like to consider a number of related problems. Firstly, our solution is constructed in such a way that the interpolation phase can be executed in parallel. A parallel implementation should expedite the interpolation procedure. Secondly, as seen in Figure 5(b), grid DEMs are often constructed from multiple sources, including LIDAR points and supplemental break-lines where feature preservation is important. Future work will examine methods of incorporating multiple data sources into DEM construction. Finally, the ability to create large scale DEMs efficiently from LIDAR data could lead to further improvements in topographic analysis including such problems as modelling surface water flow or detecting topographic change in time series data.

References

1. P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proc. International Colloquium on Automata, Languages, and Programming*, pages 115–127, 2001.
2. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
3. Applied Imagery. http://www.appliedimagery.com, 5 March 2006.
4. L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, pages 213–254. Springer-Verlag, LNCS 1340, 1997.
5. L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.
6. L. Arge, R. Barve, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickremesinghe. *TPIE User Manual and Reference (edition 0.9.01a)*. Duke University, 1999. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.
7. L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. European Symposium on Algorithms*, pages 88–100, 2002.
8. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry – Algorithms and Applications*. Springer Verlag, Berlin, 1997.
9. G. R. Hjaltason and H. Samet. Speeding up construction of quadtrees for spatial indexing. *VLDB*, 11(2):109–137, 2002.
10. M. F. Hutchinson. A new procedure for gridding elevation and stream line data with automatic removal of pits. *Journal of Hydrology*, 106:211–232, 1989.
11. S. Lee, G. Wolberg, and S. Y. Shin. Scattered data interpolation with multilevel B-splines. *IEEE Transactions on Visualization and Computer Graphics*, 3(3):228–244, July–September 1997.
12. L. Mitas and H. Mitasova. Spatial interpolation. In P. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind, editors, *Geographic Information Systems - Principles, Techniques, Management, and Applications*. Wiley, 1999.
13. H. Mitasova and L. Mitas. Interpolation by regularized spline with tension: I. theory and implementation. *Mathematical Geology*, 25:641–655, 1993.
14. H. Mitasova, L. Mitas, W. M. Brown, D. P. Gerdes, I. Kosinovsky, and T. Baker. Modelling spatially and temporally distributed phenomena: new methods and tools for GRASS GIS. *Int. J. Geographical Information Systems*, 9(4):433–446, 1995.
15. NC-Floodmaps. http://www.ncfloodmaps.com, 5 March 2006.
16. NOAA-CSC. LIDAR Data Retrieval Tool-LDART http://www.csc.noaa.gov/crs/tcm/missions.html, 5 March 2006.
17. J. Pouderoux, I. Tobor, J.-C. Gonzato, and P. Guitton. Adaptive hierarchical RBF interpolation for creating smooth digital elevation models. In *ACM-GIS*, pages 232–240, November 2004.
18. R. Sibson. A brief description of natural neighbor interpolation. In V. Barnett, editor, *Interpreting Multivariate Data*, pages 21–36. John Wiley and Sons, 1982.
19. H. Wendland. Fast evaluation of radial basis functions: Methods based on partition of unity. In C. K. Chui, L. L. Schumaker, and J. Stöckler, editors, *Approximation Theory X: Wavelets, Splines, and Applications*, pages 473–483. Vanderbilt University Press, Nashville, 2002.