Delineating Sea-Level Rise Inundation: An Exploration of Data Structures and Performance Optimization

By

Charles Grady

B.Sc., University of Kansas, 2006

Submitted to the graduate degree program in Geography and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

_____

Chair: Xingong Li

_____

James Miller

_____

Ting Lei

Date Defended: 10 May 2017

The thesis committee for Charles Grady certifies that this is the approved version of the following thesis:

Delineating Sea-Level Rise Inundation: An Exploration of Data Structures and Performance Optimization

Chair: Xingong Li

Date Approved: 10 May 2017

Abstract

Based on a conservative projection by the IPCC (IPCC 2007), inundation caused by sea level rise will likely disrupt the physical, economic, and social systems in coastal regions around the world. This research proposed an innovative method to calculate the minimum sea level rise required to inundate a cell in a Digital Elevation Model (DEM). The method, which accounts for water connectivity when determining inundation height for each cell, performs better than the simple "bathtub" approach, especially with sea level rises below 1 m. Several implementation data structures are proposed and compared. The combination of a binary heap and hash table data structure gives the most efficient implementation. The implementation is further parallelized using a master / worker paradigm. The parallel approach significantly outperforms serial implementations with respect to running time and memory footprint. Performance can be further improved with additional processing cores and using the supercomputing resources in the XSEDE (Towns, et al., 2014) program.

Acknowledgements

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575. Allocation TG-ASC160011. This work was enabled, in part, by U.S.NSF BIO/DBI #1356732.

Table of Contents

Chapter 1: Introduction 1

Chapter 2: Delineating Sea Level Rise Inundation Using Dijkstra's Algorithm 3

Introduction 3

Methods 5

Inundation by a Specific Sea Level Rise 6

Inundation Height 8

Data Structures 12

Data 19

Results and Discussions 20

Conclusions 25

Chapter 3: Parallelizing Inundation Height Calculation 27

Introduction 27

Motivation 27

Existing methods 28

Data 34

Method 35

A complete example 39

Handling tiles with different spatial resolutions 47

Implementation 49

Results 50

West Coast Experiment 51

Entire NGDC Dataset Experiment 53

Conclusions 55

Chapter 4: Conclusions 58

References 60

Appendix 65

Appendix A: West coast inundation height benchmarks. 65

Appendix B: NGDC experiment running time in seconds. 66

Chapter 1: Introduction

If sea-level continues to rise, as the Intergovernmental Panel on Climate Change (IPCC) predicts (Intergovernmental Panel On Climate Change -- IPCC, 2007), it is important to determine at what point and which coastal areas will be inundated. Coastal inundation could have significantly impacts as nearly a quarter of the world's population lives at lower than 100m and within 100km of the coast (Nicholls, et al., 2011). It is critical to know at what sea-level height and which coastal areas might be inundated in order to predict and mitigate economic and environmental impacts.

Inundation height is not simply the elevation of a given location. If there are terrain barriers to prevent water from reaching a region, then the inundation height is actually greater than the elevation at the location. There are multiple methods for delineating potentially inundated areas while accounting for barriers to water propagation. An example of one of these methods is region grouping (Li, et al., 2009) where cells in a raster that have elevation values less than a specific value are inspected and those that are connected to ocean cells are marked as inundated. A major drawback of the approach is that it is computed for specific sea-level rises. To address this limitation, a new method is proposed in chapter two that calculates the minimum sea-level rise required to inundate every cell. By creating a raster of the minimum sea-level rise required to inundate every cell, we can perform the computations once and then query the areas of inundation at specific sea-level rises rather than computing for each sea-level rise. The performance of this method, both execution time and memory usage, is the major concern and the focus of this research.

The second chapter presents the method for calculating inundation height on a DEM. The method uses Dijkstra's algorithm (Dijkstra, 1959) for graph traversal by treating each raster cell as a node and creating edges between each cell and its adjacent cells. Dijkstra's algorithm relies on a priority queue data structure to explore a graph. Two different heap implementations, binary and Fibonacci heaps, are evaluated to determine which is better suited for our method.

Chapter three presents an extension to the method that utilizes a parallel approach. The new method uses a master-worker paradigm to spread computations across multiple CPU cores, which addresses the traditional limitations of Dijkstra's algorithm. The parallel method is also evaluated on supercomputing resources using the Stampede supercomputer at the Texas Advanced Computing Center through the XSEDE program (Towns, et al., 2014).

Chapter 2: Delineating Sea Level Rise Inundation Using Dijkstra's Algorithm

Introduction

During the 20th century, world sea levels rose by 0.17 ± 0.05 m (Intergovernmental Panel On Climate Change -- IPCC, 2007). The Intergovernmental Panel on Climate Change (IPCC) estimates that the rate of sea level rise will roughly double over the next century due to increasing global temperatures, with a conservative projection of global sea level rise of 0.18-0.59 m by 2100 (Intergovernmental Panel On Climate Change -- IPCC, 2007). The IPCC report, however, acknowledges that its sea level rise appraisal does not take into account the recent rapid changes to the ice sheets that have been observed since 2003. The Greenland Ice Sheet contains a volume of water equivalent to 6 m of sea level rise, and the West Antarctic Ice Sheet, an unstable ice mass grounded well below sea level, contains a volume of water equivalent to 5 m of sea level rise (Bindschadler, 1998). Both the Greenland Ice Sheet and the West Antarctic Ice Sheet are currently showing rapid increases in mass loss that may significantly increase sea level if such mass loss continues (Thomas, et al., 2004; Rignot & Kanagaratam, 2006). Overpeck et al. (2006) indicated that warming polar temperature may reach a level this century similar to that of the Last Interglaciation (LIG), which led to sea levels about 4 to 6 meters higher than present.

Nearly a quarter of the world's population lives at elevations below 100 m and within 100 km from a coast (Nicholls, et al., 2011). Coastal regions also have the greatest concentration of economic activities. Inundation caused by sea level rise will likely disrupt the physical processes, economic activities, and social systems in coastal regions around the world (Nicholls & Tol, 2006). Numerous assessments of present and future coastal impacts of sea level rise have been conducted at global, regional, and local scales with different focuses on population, land

use/cover, flora, fauna, and biodiversity (Nicholls & Tol, 2006; Hopkinson, Lugo, Alber, Covich, & Van Bloem, 2008; Legra, Li, & Townsend, 2008; Li, et al., 2009; Craft, et al., 2009; Virah-Sawmy, Willis, & Gillson, 2009; Menon, Soberón, Li, & Peterson, 2010; Peterson, Navarro-Sigüenza, & Li, 2010). One of the key requirements in those analyses is the delineation of potentially inundated areas by different sea level rises (SLRs). Delineation of inundation from SLR differs from existing event flood models that describe short-term, pulsed-flood events (Poulter & Halpin, 2008). These flood models depend on parameters that include surface roughness coefficients and the magnitude and duration of the flooding (Marks & Bates, 2000). The long-term and near steady-state inundation resulting from SLR is less dependent on surface roughness features and individual storm characteristics, and research is needed to improve our understanding on hydraulic characteristics of coastal flooding from SLR (Poulter & Halpin, 2008).

Inundation by a specific SLR could be simply identified as the areas whose elevation is below or equal to the SLR. Although this "bathtub" approach is simple, it has one important shortcoming: water connectivity is not considered when delineating the inundation. It is possible that there are areas that have an elevation below the SLR, but they will not be inundated because terrain barriers exist between them and ocean water. Several inundation delineation methods, which do consider water connectivity, have been proposed (Poulter & Halpin, 2008; Li, et al., 2009). Li *et al.* (2009) discussed two methods of delineating inundation by a specific SLR. One of the methods uses the region group GIS function (also called component labeling in image processing) to delineate spatially contiguous inundation. The other method is similar to the method used by Poulter and Halpin (2008), where water propagates from current oceans and inundates the cells whose elevation is below a specific SLR. While the efficiency of their method

was not documented in Poulter and Halpin (2008), Li *et al.* (2009) found the method significantly slower than the method based on the region group GIS function. One common limitation among those existing methods is that they can only delineate the inundation by a specific SLR. Although those methods could be called multiple times to compile a map of inundation by several SLRs, this approach is not efficient, and more importantly, it does not take the full advantage of the vertical resolution, which continues to improve, of the DEM.

The research reported here first proposed yet another method of delineating inundation by a specific SLR. The method used the cost distance GIS function and could be easily implemented in any raster GIS having the function available. Inspired by the cost-distance based method, this research also developed an innovative method which calculated the minimum SLR needed to inundate each cell in a DEM. Once calculated, this inundation height raster layer can be used to query the inundation by any SLRs. Several data structures were proposed and compared for efficient implementation of the method.

Methods

While the "bathtub" method simply selects the cells whose elevation is less than or equal to a specific SLR as the inundation, methods that consider water connectivity only include those cells in inundation if their elevation is less than or equal to the SLR and can be reached by ocean water through spatial propagation. Two methods are proposed in this research, both considering water connectivity through either 4- or 8-neighbors. The first method delineates inundation by a specific SLR while the second method calculates the minimum SLR to inundate each cell in a DEM.

Inundation by a Specific Sea Level Rise

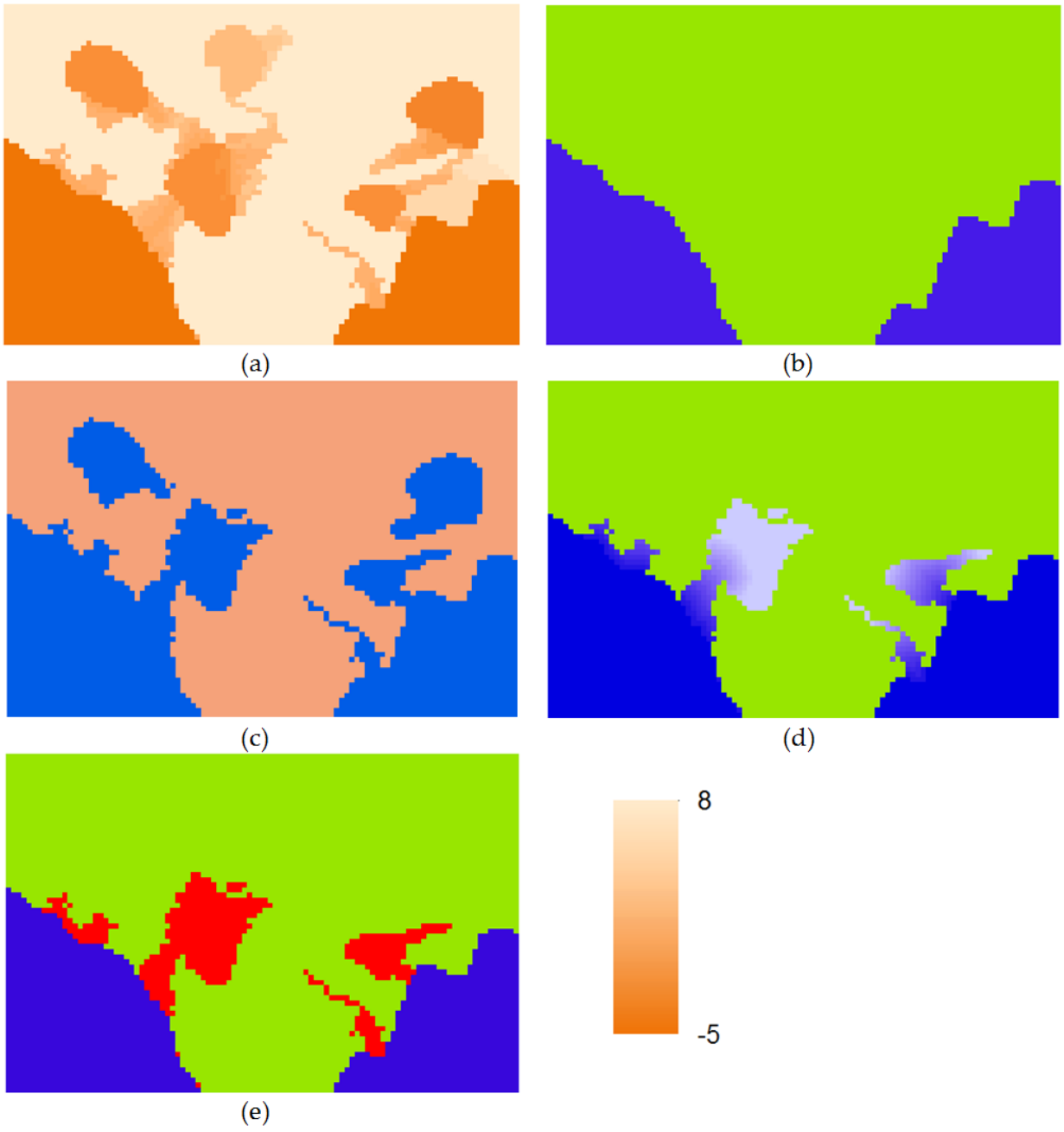


Figure 2.1. Delineating inundation by a specific sea level rise using the cost distance GIS function. (a) An example DEM with elevation ranging from -5 to 8 m; (b) current oceans (blue) and land (green); (c) the friction surface where blue cells are less than or equal to 2 m and have a friction of 1 and brown cells are physical barriers; (d) shortest distance calculated by the cost distance function where light blue indicates farther away from current oceans; € inundation by 2 m SLR (red)

Li *et al.* (2009) used the region group GIS function to test whether the cells under a specific SLR are spatially connected to oceans. As illustrated in Figure 2.1, our first method here used the cost distance GIS function to delineate spatially connected cells under a specific SLR. The cost distance GIS function calculates the least cost from each cell to some source cells (i.e., cells to which least cost is calculated) based on a friction surface. The function requires two input raster layers, one for the friction surface and the other one for the source cells. For the purpose of delineating the inundation by a specific SLR, current ocean cells (Figure 2.1b) are treated as the source cells from which ocean water propagates. The friction surface (Figure 2.1c) is created by setting a friction of 1 to the cells whose elevation is below and equal to the SLR and physical barriers to the cells whose elevation is above the SLR. With the source and friction raster layers thus created, the cost distance GIS function can then be used to calculate the least cost distance (Figure 2.1d) from current oceans to the cells to which ocean water can "travel", i.e., the cells that are not blocked by any physical barriers (i.e., terrain that is above the SLR). The inundation cells (Figure 2.1e), therefore, are the cells whose distance to the current oceans is greater than zero (current ocean cells have the distance of zero because they are the source cells). Figure 2.2

illustrates how this method can be implemented as a ModelBuilder© model with the ESRI ArcGIS© software.

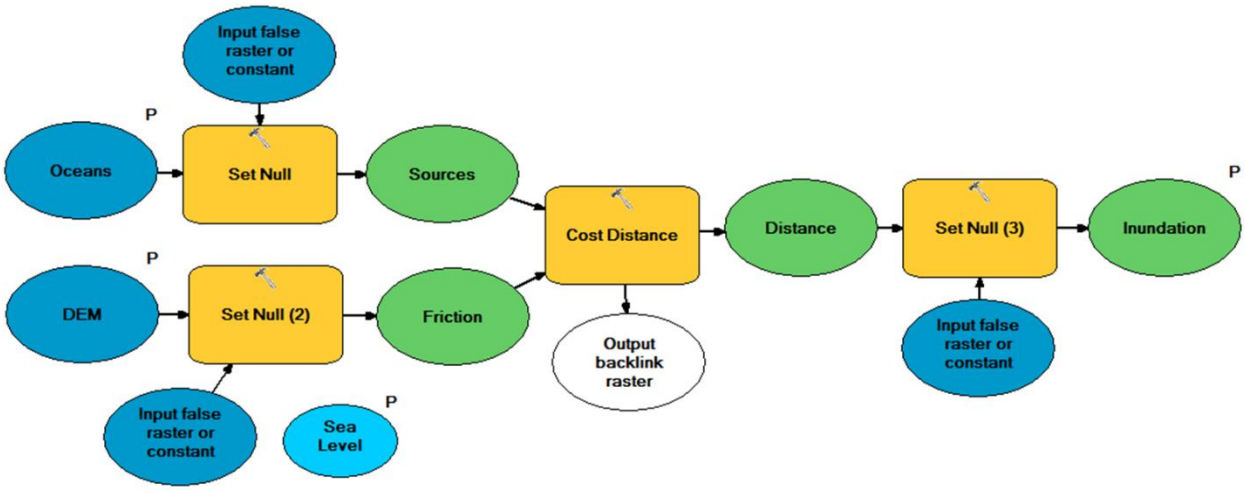


Figure 2.2. Cost distance based inundation model built using ModelBuilder of the ArcGIS software. Rectangles are GIS operations. The blue and green ellipses connected to the rectangles are, respectively, the inputs and outputs of the GIS operations. Ellipses with a letter "P" indicate the parameters of the model.

Inundation Height

Our second method was inspired by the cost distance function in the first method. Instead of just taking advantage of the implicit spatial propagation in the cost distance GIS function, we developed a method which calculates the minimum SLR needed to inundate each cell in a DEM. We call the minimum SLR inundation height.

Our method is based on the Dijkstra algorithm which is a graph traversal algorithm that finds the minimum cost and path on a graph with non-negative costs (Dijkstra, 1959). It begins at a start node and adds all nodes directly connected to it to a set of possible nodes (front nodes hereafter). Then, the node that has the least cost in the front nodes is extracted and added to the set of visited nodes. All the nodes connected to the extracted node are added to the front nodes, and

any nodes in the front that can be reached (through the extracted node) with less cost than before have their costs reduced. The process repeats until all nodes have been visited.

Dijkstra's algorithm can be adapted for computing inundation height for a DEM raster layer by treating each cell center as a node and creating edges between each cell and its neighbors. Our method supports both 4- and 8-connetivity neighbors, although all the analyses were based on 8-connectivity neighbors. With this graph view on a raster layer, current ocean cells are the start nodes and the algorithm iteratively spreads from the ocean cells by maintaining a set of land cells which are the immediate neighbors of ocean cells. We call this set of land cells the *inundation front* hereafter. Initially, the inundation front contains all the land cells next to current ocean cells with their elevation as their inundation height. In each iteration, the algorithm extracts from the inundation front the cell with the lowest inundation height (low-cell hereafter), writes the inundation height on the output raster layer, and updates the inundation front by finding the low-cell's neighbor cells, calculating their inundation heights, and inserting those cells into the inundation front. When there are several cells having the same lowest inundation height in the inundation front, the low-cell is the cell that enters the inundation front the first. The inundation height for a cell is calculated as:

$$Inundation\ height = \max(low - cell's\ inundation\ height, elevation\ of\ the\ cell) \quad (2.1)$$

The above process repeats until all the cells have their inundation height calculated. Figure 2.3 gives an example DEM (Figure 2.3a) and a current ocean (Figure 2.3b) raster layer from which inundation height is calculated. The example DEM has six rows and six columns and the ocean layer has three current ocean cells. Figure 2.3c shows the initial cells in the inundation front and

their inundation heights. Figure 2.3d to 2.3j illustrate the snapshots of running the algorithm a few more iterations. The primary output of the algorithm is a raster layer where the value at a cell represents the minimum SLR required to inundate the cell. A possible secondary product is a raster layer that stores the shortest path by which a cell is inundated by ocean water.

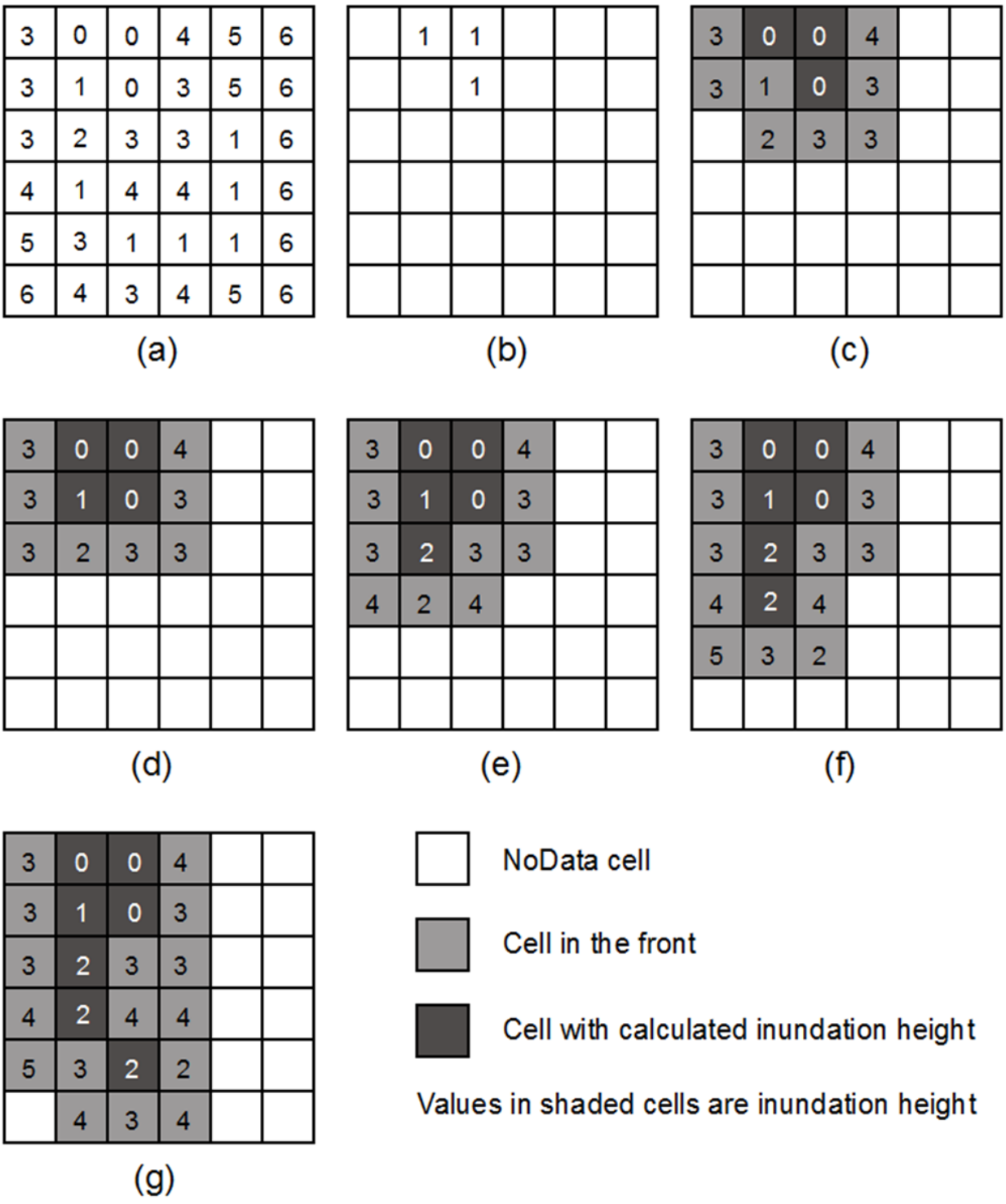


Figure 2.3. Calculating the inundation height on an example DEM. (a) The DEM; (b) current ocean raster layer with 3 ocean cells; (c) initial inundation front and their inundation height; (d) – (g) inundation height calculation for four cells and the evolution of the inundation front.

Data Structures

The key data structure in the algorithm is to represent and manage the inundation front from which ocean water propagates. Any data structure used to manage the inundation front must provide two operations: the *extraction operation* which finds the cell with the lowest inundation height in the front and removes it from the front; and the *insertion operation* which either adds a new cell into the front if the cell does not exist in the front or updates the cell's inundation height if necessary. The front expands and shrinks during its propagation. As the size of the raster layer increases, so does the number of cells in the front, and the number of iterations will increase quadratically. This means the extraction and insertion operations will be called more times with more cells in the front. Designing efficient data structures for the operations on the inundation front becomes the most crucial component of implementing the algorithm.

The front behaves like a minimum priority queue (Cormen, Leiserson, Rivest, & Stein, 2009), where each item in the queue has an associated priority, which, in our case, is inundation height. One way to build a minimum priority queue is to use a binary heap. A binary heap is a *complete binary tree* which has as many nodes as it can hold at all levels except for the leaves. Any nodes on the leaves are pushed to the left. In a minimum binary heap (min-heap hereafter), each tree node is, at most, as large as its two children. While the two children must be at least as large as their parent, either may be larger than the other. Because each node is at most as large as the two nodes below, the root node is always the smallest node in a min-heap. This makes min-heaps a good data structure for implementing a minimum priority queue. A binary heap also has several important properties. First, it is the shortest tree that can hold a given number of nodes. Second, if a binary heap contains N nodes, its height is log(N). Last, a binary heap can be stored very compactly in an array.

To remove the smallest items from a min-heap, the last item is moved to the top of the tree. The item is then pushed down until it reaches its final position and the tree is again a valid min-heap. Because the tree has a height of log(N), this process can take, at most, log(N) steps. This means a min-heap based priority queue takes Θ(log N) run time to extract its smallest item. To add a new item to a min-heap, the item is placed at the bottom of the tree and then pushed upward until the tree is again a valid min-heap. Because the tree has a height of log(N), this process also can take, at most, Θ(log N) run time. Decreasing an item's priority is similar to adding a new item. After the priority of the item is reduced, the item is pushed upward until the tree is again a valid min-heap. This will take, at most, Θ(log N) run time too. Figure 2.4b and c give the examples of extracting a node and decreasing the priority of a node in a min-heap.

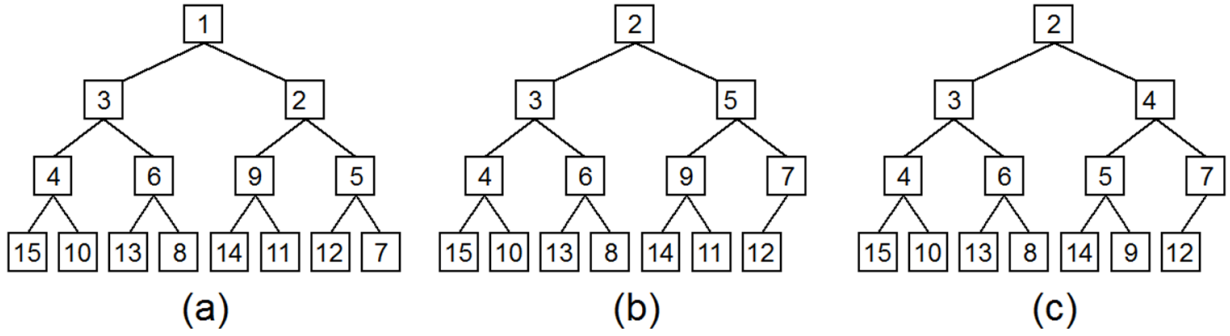


Figure 2.4. An example min-heap and some operations on the heap. (a) The min-heap after items 15, 14, 13, 9, 10, 12, 4, 3, 1, 8, 6, 7, 11, 2, and 5 are inserted into the heap; (b) the min-heap after item 1 is extracted; (c) the min-heap after 11's priority in (b) is decreased to 4.

Another way to build a minimum priority queue is to use a Fibonacci heap (Fredman & Tarjan, 1987). A Fibonacci heap is a more complicated data structure than a binary heap but may provide potential performance gains. Figure 2.5a shows an example Fibonacci heap. In a Fibonacci heap, every node is a sub-heap and each node has four pointers: one to its parent (null if it is a root node), one to its left sibling, one to its right sibling, and one to its child with the

smallest key. Each node has a key, which is the value used to determine its position in the heap, which, in our case, is inundation height. Additionally, each node has a rank that indicates how tall the sub-heap is at that location and a bit indicating if the node has been marked, meaning it has lost a child.

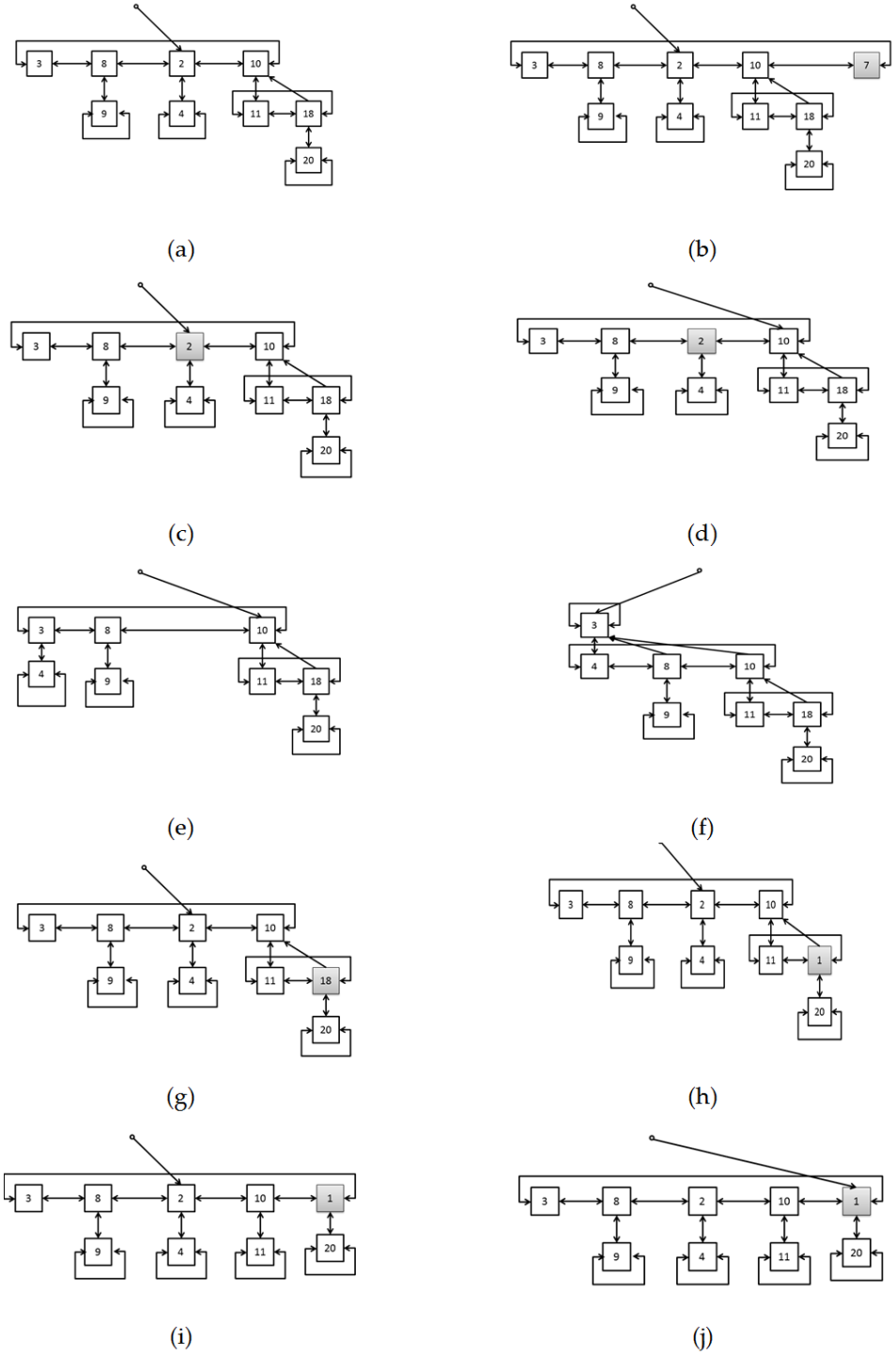


Figure 2.5. An example Fibonacci heap and operations on the heap. (a) The original Fibonacci heap; (b) the heap after item 7 is inserted; (c)-(f) the extract minimum operation. The minimum pointer is moved to a sibling and the minimum node's children are promoted to the root level. Sub-heaps with the same degree are merged until their root level no longer contains any two sub-heaps with the same degree (height); (g)-(j) the Reduce Key operation. Item (18) has its key reduced and is cut from its parent and moved to the root level. Then the minimum pointer is reset to the new heap minimum.

The insert operation for a Fibonacci heap creates a new sub-heap and inserts it into the heap's ring of root sub-heaps and performs a merge operation. If the newly inserted item has the lowest key for the entire heap, the minimum pointer changes to point at it and the operation is complete. The efficiency of this operation is $\Theta(1)$ and runs in constant time. Figure 2.5b illustrates the heap after a new item (7) is inserted into the heap shown in Figure 2.5a.

To remove the smallest item from a Fibonacci heap, the children of the item pointed at by the minimum pointer are added to the root ring of sub-heaps. In the worst case, this can result in all items being in the root ring of sub-heaps and become very inefficient. To combat this, when the extract operation takes place, sub-heaps with the same rank are linked together by making one a child of the other so that the heap property is maintained. If a node is marked, it is moved to the root level. Once all of the sub-heaps are linked, the minimum pointer is set to the sub-heap with the item with the smallest key. The amortized cost of this operation is $O(\log N)$. The operation is demonstrated in Figure 2.5c-f, where the minimum item (2) is deleted by first moving the root minimum pointer to the sibling on the right (10). Then, the children of node (2) are promoted to the root level of the heap. From there, link operations are performed on sub-heaps with the same rank until no two sub-heaps at the root level have the same rank. At that point, the root minimum pointer is reset to the smallest root item (3).

To reduce an item's key in a Fibonacci heap is similar to insert an item. First, the key of the item in question is reduced. If the heap property is no longer satisfied, the parent of the item is marked and the entire sub-heap is cut from its parent and inserted at the top level of the heap. From there, the same procedure used for insert is performed. The amortized efficiency of this operation is $\theta(1)$. Figure 2.5g-j demonstrate the process, where item 18 has its key reduced to 1.

The parent of the item 18 is marked and the entire sub-heap is cut and moved to the root level. Finally, the root minimum pointer is reset to item 1.

The data structures used to implement the minimum priority queue can greatly influence the performance of the method. Both binary heap and Fibonacci heap have advantages and drawbacks. While binary heaps are simpler to implement, Fibonacci heaps provide better theoretical computational complexity as shown in Table 1, where N is the number of items in a heap.

Operation	Binary Heap	Fibonacci Heap
Insert an Item	$\Theta(\log N)$	$\Theta(1)$
Remove the Minimum Item	$\Theta(\log N)$	$O(\log N)$
Reduce Key	$\Theta(\log N)$	$\Theta(1)$

Table 1. Theoretical computational complexity of the three main operations in the heap implementation of a minimum priority queue

Whether a min-heap or a Fibonacci heap is used to implement the minimum priority queue, a crucial step in the insertion operation is to determine whether a low-cell's neighbor cells are already in the inundation front before adding them to the front or updating their inundation height. The operation has to search through the front based on the location of a cell. Depending on how the spatial connectivity is defined, this operation is called four or eight times in each iteration. As the number of cells in the front increases, this operation becomes a bottleneck if a sequential search is performed. Therefore, we built a location index for front cells using a hash table. With the hash table index, the average time to access a cell in the front is $\Theta(1)$ and the worst-case time is $\Theta(N)$ where N is the number of cells in the front.

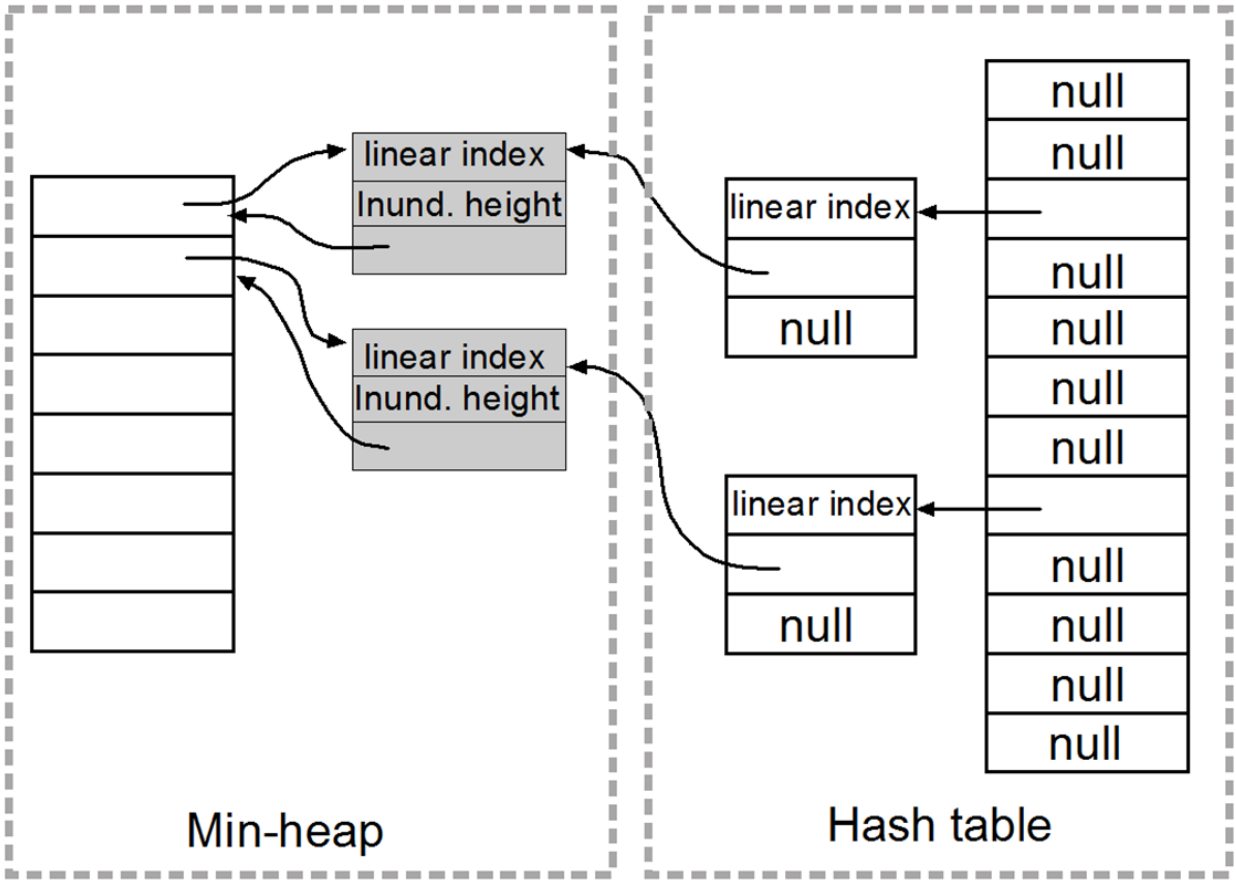


Figure 2.6. Data structures used in the min-heap implementation of the inundation height method

Our min-heap implementation used a dynamic array which expands and shrinks as cells are inserted and extracted from it (Figure 2.6). An item in the array is a pointer to a front cell object. Each front cell object stores inundation height, a linear index, and its position index in the min-heap array. Inundation height is the key in the min-heap. A hash table is used to create a location index to front cell objects. Each hash table node stores a location key, which is the linear index of a cell on a DEM raster layer, and a pointer to a front cell object. Maintaining a position index in a front cell object adds extra complexity to the extraction and insertion operations which move front cells in the min-heap array. However, this effort is rewarded when the insertion operation

has to quickly find a cell in the min-heap array by its location index, and when the inundation height of a cell needs to be decreased.

The two heaps and the hash table data structures were implemented in C++ and wrapped to work with MATLAB. All the calculations were performed using a Lenovo Thinkpad x220t machine with a 2.70 Intel Core i7-2620M processor with 8 GB RAM running MATLAB R2007b.

Data

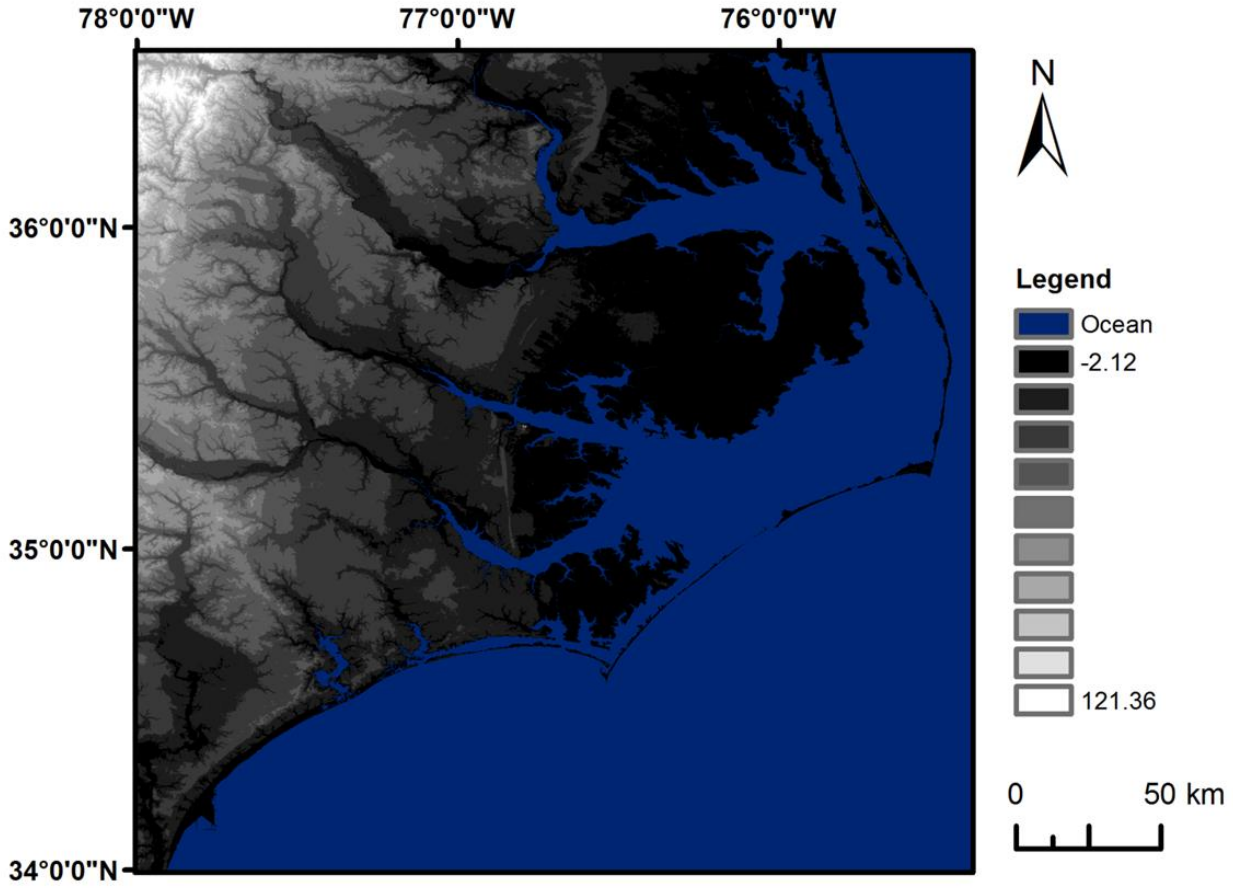


Figure 2.7. A DEM covering the east coast of the state of North Carolina, USA. Ocean is in blue and elevation is in shades of gray, the darker the lower.

To test our methods and to evaluate the efficiency of different implementation data structures, we used a DEM covering the East coast zone of North Carolina (Figure 2.7). The DEM was downloaded from the USGS National Elevation Dataset (NED) website. Several individual tiles of the DEMs were first downloaded then pieced together to cover the entire region at a spatial resolution of 30 meters. Since LiDAR data was used to generate the DEM, it has high vertical accuracy and resolution. The DEM grid has a dimension of 9188 x 9360 (about 86 million) cells. Among them, 54% are land cells, which have a mean elevation of 14.83 m and range between -0.12 m and 121.36 m.

Results and Discussions

To compare the simple "bathtub" approach and our methods, we used Equation 2.2 to calculate an error percentage. We used the equation and the term "error" here because the "bathtub" approach overestimates inundation and our methods are theoretically correct, and the true inundation is not available.

$$error\ \% = \frac{Bathtub\ Inundation - Our\ Inundation}{Our\ Inundation} \qquad (2.2)$$

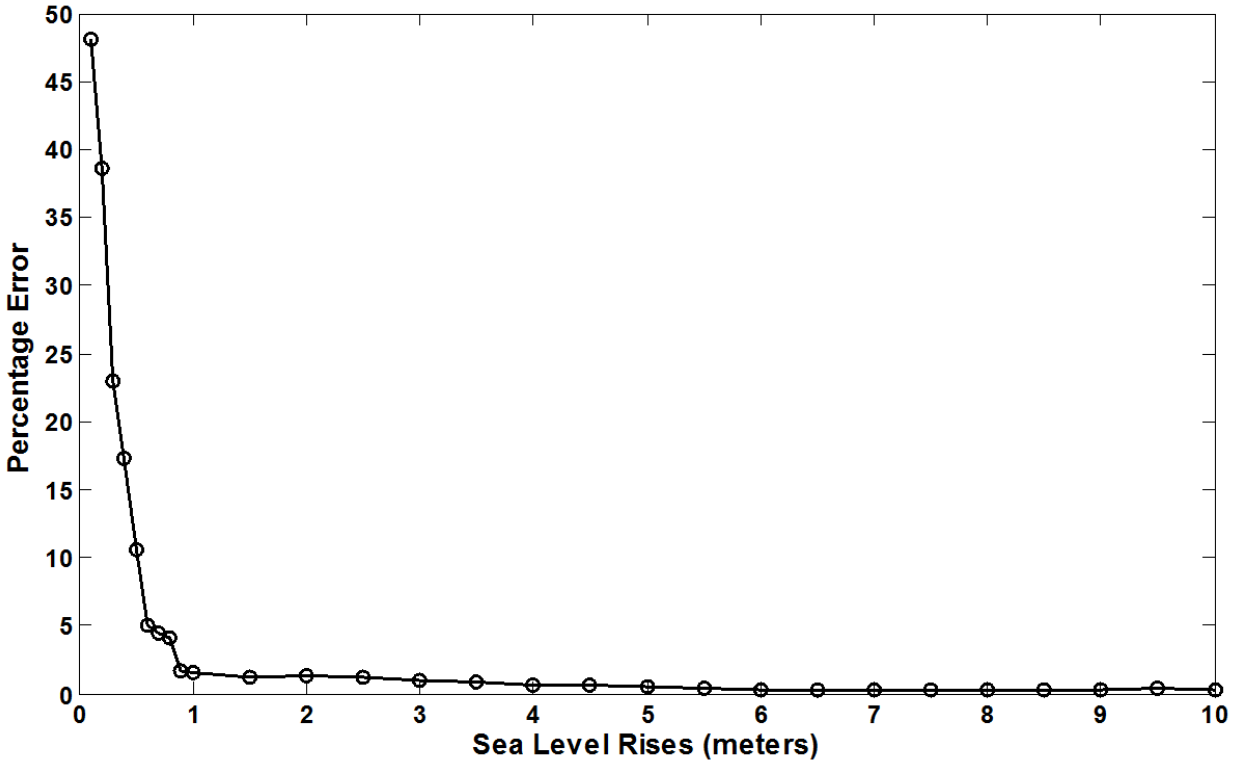


Figure 2.8. Percentage error of the "bathtub" method with different sea level rises.

Figure 2.8 shows the error percentages calculated using Equation (2.2) with 28 SLRs. The error percentage decreases from 48% to 0.3% as SLR increases from 0.1 to 10 m. The average error percentage for the SLRs of less than 1 m is 17%. However, the error percentage drops drastically to 1.5% when SLR reaches 1 m and reduces to an average of 0.6% for SLRs above 1 m. With IPCC's projection of global SLR of 0.18-0.59 m by 2100, the error percentage ranges between 40% and 6% on the North Carolina DEM. Those results indicate that it is especially important to consider water connectivity in delineating inundation with small and conservative SLR scenarios.

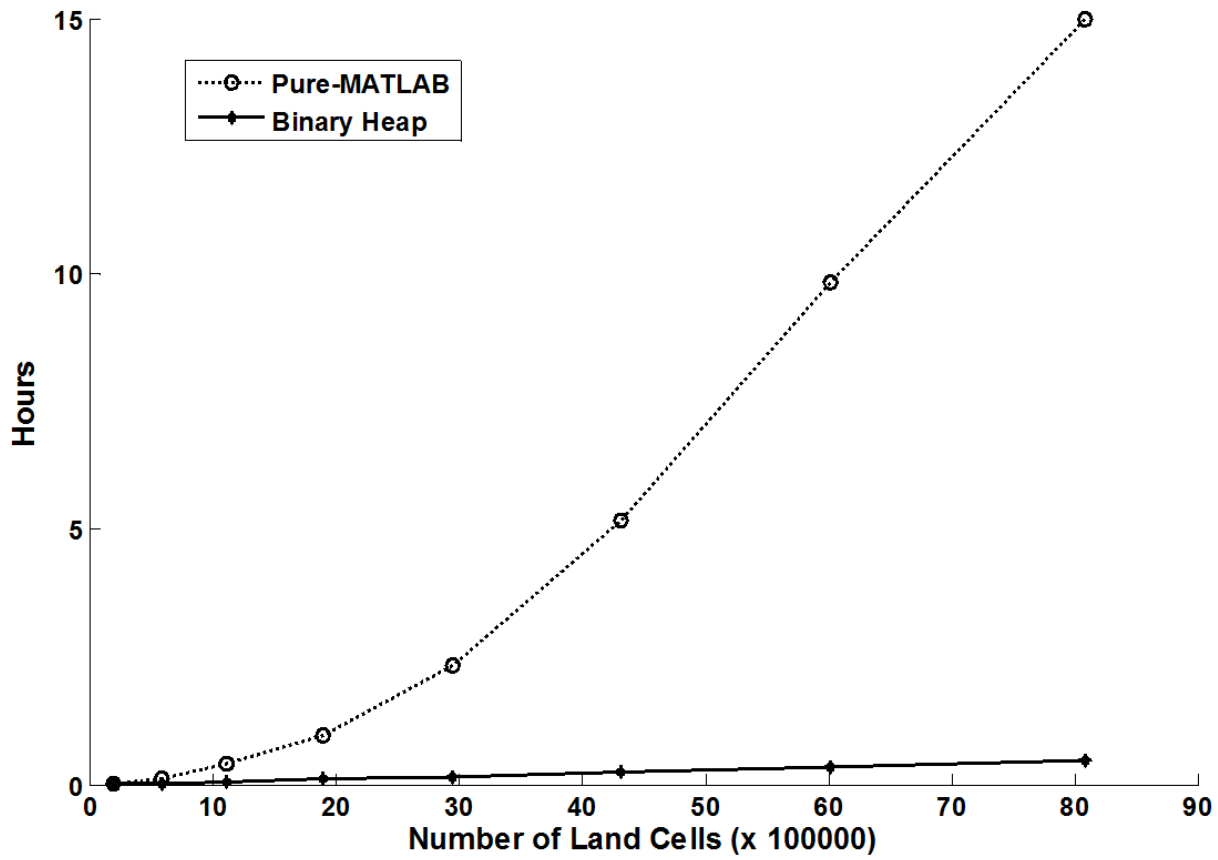


Figure 2.9. Running time comparison between a binary-heap-based implementation and a pure-MATLAB implementation.

To see how the choice of data structures may affect the efficiency of calculating inundation height, we also implemented the method using only matrix functions (i.e., no heap and hash table data structures) in MATLAB to perform the operations (pure-MATLAB implementation hereafter). Figure 2.9 shows the running time for an implementation using a binary heap and hash table and the pure-MATLAB implementation with 8 different sizes of sub-DEMs from the North Carolina DEM. The sub-DEMs have the land cells range from 197,000 to 8 million. With the increase of land cells on those sub-DEMs, running time also increased in both implementations. While the running time for the heap based implementation increased linearly

with the increase of land cells, the running time for the pure-MATLAB implementation, however, increased quadratically. For example, with the smallest sub-DEM, which has 197,000 land cells, the running time for the heap based and pure-MATLAB implementation are 38 and 84 seconds, respectively. The heap based implementation is only twice faster than the pure-MATLAB implementation. However, when the sub-DEM with 8 million land cells was used, the running time for the heap based and pure-MATLAB implementations are 0.47 and 15 hours, respectively, a difference of over 30 times. This comparison clearly indicated the importance of the data structures used in implementing the method.

We calculated inundation height for the entire North Carolina DEM using both binary heap and Fibonacci heap based implementations. The raster layer has more than 46 million land cells. The binary heap implementation ran in 5,821 seconds and the Fibonacci heap version ran in 50,358 seconds. Since both heap implementations used the same hash table, the computational complexity of the two heaps (Table 1) would indicate that the results should be the opposite of what they really were. However, there are two reasons for these results. First, as shown by Goldberg and Tarjan (1996), the constant factor for the extract minimum operation of the Fibonacci heap is much greater than that of the binary heap, thus making the operation more computationally expensive for the Fibonacci heap. Second, much of the performance gains found by using a Fibonacci heap over a binary heap are due to the efficiency of the reduce key operation. However, the binary heap outperforms the Fibonacci heap in calculating inundation height because the reduce key operations did not take place. This is true due to the function used to calculate inundation height (i.e., equation (2.1)) and the property of Dijkstra's algorithm. If a low-cell has an inundation height of IH_{lc} and one of its neighbors has an inundation height of IH_{nbr} and an elevation of E_{nbr}, and already exists in the inundation front, then the relationship E_{nbr}

$\leq IH_{nbr} = IH_{lc}$ or $IH_{lc} < IH_{nbr} = E_{nbr}$ must be true. When the low-cell is extracted from the front, the new inundation height (IH'_{nbr}) of the neighbor is calculated, using Equation (2.2), as $IH'_{nbr} = \max(IH_{lc}, E_{nbr})$, which is always equal to its existing inundation height (IH_{lc}) in the front. Therefore, the neighbor's inundation height in the front will never need to be reduced. These factors eliminated the benefit of using a Fibonacci heap for this particular application.

A hash table was used to create an index for the cells in the inundation front to quickly locate them in the minimum priority queue. In addition to hash table, there are several other data structures which can be used to search the cells in the front (Li, Larson, & Rex, 2005). However, with the observation that the inundation height of front cells are never reduced, our binary heap based implementation can be further improved. One possibility is to remove the hash table at all and flag the front cells on the output raster layer using some special value, for example, a negative inundation height. This way, the algorithm can check the output raster to see whether the neighbors of the low-cell are already in the front. Only the neighbors which are not in the front will be added and no other action is needed.

The first method was implemented as a model tool in ESRI ArcGIS and the ModelBuilder diagram was shown in Figure 2.2. The method can also be implemented in any raster GIS software with the cost distance function available. The second method was implemented using C++ and MATLAB and can be accessed through MATLAB. We are considering implementing the second method using the Python programming language and making it available in ESRI ArcGIS as a Python script tool in the future.

One important limitation of the existing inundation delineation methods, including the two methods developed here, is that they can only handle single DEM raster layer. As higher spatial resolution DEMs become more and more available and are commonly distributed as DEM tiles,

creating a single DEM raster layer for inundation calculation is not feasible. The process of creating a single raster layer from DEM tiles is tedious and the size of the DEM can easily exceed computer memory, which makes the calculation impossible if the methods are not disk-based (which means entire raster layers have to be loaded into memory in order to perform the calculation). Even if we can process single large DEM, the output raster layers usually have to be tiled, more often than not, to distribute the results. We are currently developing parallelized inundation delineation methods which can handle tiled DEMs and also take the advantages of multicore and cluster computing systems.

As indicated in the introduction, delineation of inundation from SLR differs from short-term and pulsed-flood inundation modeling which depends on surface parameters and the characteristics of individual storm events. Inundation from SLR, instead, reflects long-term and near steady-state flooding, which completely ignores storm effects and the possible changes of storm characteristics caused by SLRs.

Conclusions

One of the key requirements in the assessments of present and future impacts of SLR is the delineation of potentially inundated areas by different SLR scenarios. This research developed two methods for delineating potential inundation by SLRs. The methods require a DEM and current ocean as the inputs and can be used to delineate inundation for different types of coastal landscapes and for any SLRs as long as DEM's vertical resolutions are comparable with the SLR scenarios.

Both methods consider water connectivity and performed better than the simple “bathtub” approach. The error (overestimate of inundation) from the “bathtub” method, based on the North Carolina DEM, is the largest with SLRs below 1 m, ranging from 48% to 1.5% with an average of 17%. With SLRs above 1 m, the average error is only 0.6%. The methods, therefore, make the biggest difference in low SLRs, which are the most likely future projections according to the IPCC (2007).

The first method is based on the cost distance GIS function and was implemented using the ESRI ArcGIS software. The method, like other existing methods, however, can only delineate the inundation by a specific SLR. Inspired by the cost distance function, we also developed an innovative method which calculates the minimum SLR (i.e., inundation height) to inundate each cell in a DEM. The importance of data structures used in implementing this method is clearly demonstrated in our research. While the running time of a binary heap based implementation increased linearly with the increase of DEM size, the running time of a non-heap implementation increased quadratically. The min-heap implementation outperformed the non-heap implementation significantly, especially with large DEMs. The Fibonacci heap data structure has a better theoretical computational complexity than the binary heap. However, the Fibonacci heap based implementation did not perform well because the reduce key operation is not necessary in this particular application.

Several improvements can be further explored ranging from using a better location index for searching the cells in the inundation front to removing the location index entirely. More important, all the existing methods, including the two we developed, are limited to handle single DEM. We need parallelized methods which can handle tiled DEMs and take the advantages of multicore and cluster computing systems.

Chapter 3: Parallelizing Inundation Height Calculation

Introduction

There is a class of computationally expensive raster GIS functions which propagate a property of interest from certain cells (often called source cells) to all the other cells in a raster (Gao & Sudhakar, 1994). Examples of these raster GIS propagation functions include cost distance and coastal inundation height (Li, Grady, & Peterson, 2014). Graph traversal algorithms, such as Dijkstra's algorithm (Dijkstra, 1959) and the Bellman-Ford algorithm (Bellman, 1958; Ford Jr, 1956; Moore, 1959) can be used for implementing these calculations when each cell in a raster is treated as a node and the links between a cell and its adjacent cells are edges. As input raster files become large, the time and resources required to perform these calculations with traditional implementations becomes unsustainable. This work aims to improve the running time and reduce the memory footprint of these calculations as well as to enable new analyses that are not possible with the traditional implementations.

Motivation

Using the Dijkstra's algorithm, Li et al. (2014) calculated inundation height due to sea level rise from a raster that had approximately 46 million cells and took almost two hours. The calculations were for one tile from the National Geophysical Data Center Coastal Relief Model (NGDC) dataset that has 537 one degree by one degree tiles (NOAA National Geophysical Data Center, 2014). Extrapolating for the entire dataset suggests that the amount of time and resources needed for the existing approach would not be feasible without specialized, high-memory, hardware. Even if a machine were able to handle the large data size, the running time

required to perform the calculations would likely take longer with better and finer resolution data.

Computing resources have multiplied over time, today even commodity computing hardware resources have multiple processing cores that can be used for computations. Additionally, the accessibility of supercomputing resources has become more prevalent through the U.S. Extreme Science and Engineering Discovery Environment (XSEDE) program (Towns, et al., 2014). These hardware resources provide great potential for reducing total running time for many applications through parallelization if their algorithms can be adapted to utilize these resources. In recent years, computing processor speed has not made the large jumps forward as it has in the past and instead has increased the quantity of processing units, be that computing cores, GPUs, or coprocessors, to enable greater levels of parallelism for improving overall speed (National Research Council, 2011). It is imperative that new algorithms be developed with an eye towards parallelism and that existing algorithms be assessed to see how they might be adapted to take advantage of this computing architectures potential, not only to improve running time, but also to more efficiently utilize the available resources. In this research, we parallelized the algorithm for calculating inundation from sea level rise. Although the algorithm is a special version of the cost distance calculation, we have design hooks in place to expand the algorithm for more general cost distance computations.

Existing methods

There have been various efforts to speed up least cost path and cost distance computations. Some of these efforts relax requirements, such as being absolutely correct versus approximately correct (Hart, Nilsson, & Raphael, 1968). Others precompute every possible scenario in order to optimize “on demand” requests (Sun & Sun, 2016). Still others explore alternative data

structures and heap implementations in order to optimize computations as much as possible (Crauser, Mehlhorn, Meyer, & Sanders, 1998; 2014; Solka, Perry, Poellinger, & Rogers, 1995; Träff & Zaroliagis, 1996).

Accuracy versus performance is an important consideration when selecting an implementation for the proposed function. Dijkstra's algorithm was chosen because it guarantees that the least cost path will be calculated (Cormen, Leiserson, Rivest, & Stein, 2009). There are faster and less resource-intensive algorithms for determining the cost from a source cell to a destination cell. One of these algorithms is "A*" (Hart, Nilsson, & Raphael, 1968). A* produces an approximation of the least cost path between two points by using a heuristic function to estimate the cost left to reach the destination and then explores the route with the smallest cost. There are two issues with this algorithm. First, the result is only an approximation, meaning that the true cost and path may not be found. This is a critical shortcoming for applications that require accurate results. A second issue is that the algorithm is designed for a single origin/source and a single destination. It may be possible to add multiple source locations to this algorithm, but the heuristic function can only estimate the cost to a specific destination. Propagation functions in general and the coastal inundation height function, specifically, require calculating the cost (height in this case) to many destinations.

Another alternative algorithm is the Bellman-Ford algorithm (Bellman, 1958; Ford Jr, 1956; Moore, 1959). Like Dijkstra's algorithm, the Bellman-Ford algorithm guarantees that the least cost path will be found. This algorithm works in a similar way to Dijkstra's algorithm as it determines the least cost required to reach, at least one, node in each iteration. In each iteration, every node is analyzed to see if the cost to reach it has changed. If it has, then the cost is updated, and this update may trigger the update of the connected nodes in the next iteration. A

primary difference between Dijkstra's algorithm and the Bellman-Ford algorithm is that Dijkstra's algorithm holds all of the next possible nodes in memory and selects one at a time, while the Bellman-Ford algorithm explores all connected nodes at every iteration. The absence of a priority queue data structure means the memory footprint of the Bellman-Ford algorithm is smaller than Dijkstra's algorithm. However, the number of calculations required by the Bellman-Ford algorithm is much more than Dijkstra's algorithm as every cell is evaluated at every step. One advantage of the Bellman-Ford algorithm is that it allows negative weights on edges. However, this is not an important consideration for most, if not all, propagation functions as the edge weights (costs) are not negative.

Gao and Sudhakar (1994) have attempted to reduce the calculations involved with the Bellman-Ford algorithm. They divided a raster into chunks and then processing adjacent chunks once their edges were reached, allowing only a portion of the study area to be in memory. This was designed to reduce the number of calculations and memory footprint but not for parallelization. The research method presented here approaches the problem in a similar fashion but will parallelize and operate on the calculations in the chunks concurrently. There is a possibility to expand the proposed method with the Bellman-Ford algorithm but the number of required calculations may outweigh memory savings.

There have been prior efforts to parallelize Dijkstra's algorithm (Crauser, Mehlhorn, Meyer, & Sanders, 1998; Solka, Perry, Poellinger, & Rogers, 1995; Träff & Zaroliagis, 1996). These studies focused on parallelizing the processing of items in the heap that are not likely to collide. Crauser (1998) processed edges costs within a certain threshold in parallel. Solka (1995) processed edges in parallel based on the angle between them. This implementation is more similar to Träff and Zaroliagis (1996) that divided a graph into regions and then applied

Dijkstra's algorithm to each of them. However, the above approaches are more concerned with creating a work-efficient algorithm, whereby the total number of calculations is minimized, and where all of the processors need to communicate with each other to find the boundary node with the least cost. These implementations use a centralized heap or a different priority queue data structure, which allows for a traditional implementation of Dijkstra's algorithm to operate in the same manner. This, however, limits their implementations to architectures with shared memory. This research decentralizes the priority queue structure so that there is an instance with each tile. This allows for the implementation to reduce the memory footprint required for computations, as the size of the tiles can be controlled and with a focus on speed, but the method may add more total work to be done as it can require the cost to be calculated multiple times for each tile.

Jasika et al. (2012) implemented a parallel version of Dijkstra's algorithm that they ran on multiple processing cores using OpenMP as well as GPUs using OpenCL but found that it was not significantly faster than a serial implementation. They parallelized the computations by running separate processes for each source cell and processing all the connected edges for a vertex in parallel. They suggest that the overhead added for parallelization was not offset by the speed up from the parallelization itself. The overhead of parallelization was a concern for the presented research as well and was addressed by configuring the tool with multiple tile sizes.

Meyer and Sanders (2003) presented a shortest path algorithm that also allows for recalculation of path costs. Their algorithm creates "buckets" that each contains path nodes that fall within a specified cost window. The algorithm allows for nodes that are near each other, cost-wise, to be explored in parallel and if a node is reached with a smaller cost, it can be re-inserted into the bucket. This approach creates a similar result to Crauser's (1998) method. At first glance, the buckets that Meyer and Sanders used seem similar to the chunks, or regions, used here.

However upon deeper examination it is clear that this research is very different from theirs. They used multiple processors to explore the nodes in a bucket, one bucket at a time. This research uses one processor per region and works on multiple regions simultaneously. While they do allow for a node to be re-inserted into a bucket if it is reached with a smaller cost, the methods described here allow entire, or partial regions to be recalculated as necessary.

Sun and Sun (2016) explored options for speeding up Dijkstra's algorithm computations with parallelism and region decomposition but they focused on road networks and pre-computed intermediate results with the goal of speeding up dynamic requests. They used single source computations and pre-computed all of the paths between each node and every other node. They then focused on compressing those results and speeding up "online" requests for retrieval of any particular path between two nodes. They decomposed networks into regions using k-means clustering. Their approach is different from the presented research because it focuses (1) on single source / single destination pairs and (2) on speeding up the final step of retrieving those paths between those pairs. This work focuses on speeding up computations from source nodes to all other nodes.

The most similar effort to this proposed method is that of the MrGeo software (https://github.com/ngageoint/mrgeo/wiki). MrGeo uses a single source version of Dijkstra's algorithm to compute cost distance values for a set of tiles. As the computations reach the edge of a tile, they report the adjacent tiles that should be computed in the next iteration. The cost of each cell is determined by a friction surface resulting in cost per meter values for each cell. It does not seem to be possible to compute inundation height using MrGeo, but conceptually, the computations are similar. The primary difference between MrGeo and our method is that MrGeo is based on the MapReduce framework and is locked into discrete iterations (Li, Hu, Li, Wu, &

Yang, 2016). This means that all active tiles are started at once and no new computations can begin until all existing computations are finished. Our method propagates computations between tiles as soon as the computations on a tile finishes. The advantage of our approach is that we can begin new computations immediately rather than waiting for the slowest computation / tile to complete at each iteration.

There have been many efforts to both improve the performance of cost distance computations and Dijkstra's algorithm that we rely on. These optimizations of cost distance calculations may not maintain the same requirements for results. Examples of these concessions include only providing an approximation of the least cost path and limiting computations to a single source cell. Other least cost algorithms, such as the Bellman-Ford algorithm (Bellman, 1958; Ford Jr, 1956; Moore, 1959), make trade-offs of a reduced memory footprint at the cost of potentially many recomputations for the same cell. The efforts to optimize Dijkstra's algorithm have produced limited gains as they have mainly focused on parallelizing and / or optimizing the priority queue data structure that Dijkstra's algorithm relies on. There have been other efforts that resemble ours and relax the requirements of a strict implementation of Dijkstra's algorithm, but those too have their limitations that we will address in our approach.

Here we present a parallel approach for computing inundation height for coastal regions of the United States. Our approach provides the flexibility to work with datasets with multiple resolutions and can be deployed on heterogeneous hardware. To evaluate our approach, we compare it to a serial implementation for the same data. We then deploy our method with a larger dataset on supercomputing resources to see if the same patterns hold.

Data

The Digital Elevation Model (DEM) data used for this method is the Coastal Relief Model (CRM) dataset from the National Geophysical Data Center (NGDC) (NOAA National Geophysical Data Center, 2014). Shown in Figure 3.1, the dataset is comprised of ten volumes representing different coastal regions of the United States. For nine of these regions, the data is made available as a large raster file for each volume. For the tenth volume (Volume 6 in Figure 3.1), the data is available as a set of one degree by one degree tiles. There are two versions of these tiles, the first set has a resolution of three arc seconds, matching the resolution of the other volumes. The second option has a resolution of one arc second. For this project, this dataset was manipulated into multiple forms: (1) A set of raster files, one per region, (2) sets of half degree, one degree, and two degree tiles for the entire dataset, and (3) a single aggregated raster file where all of the volumes are merged into one raster file.

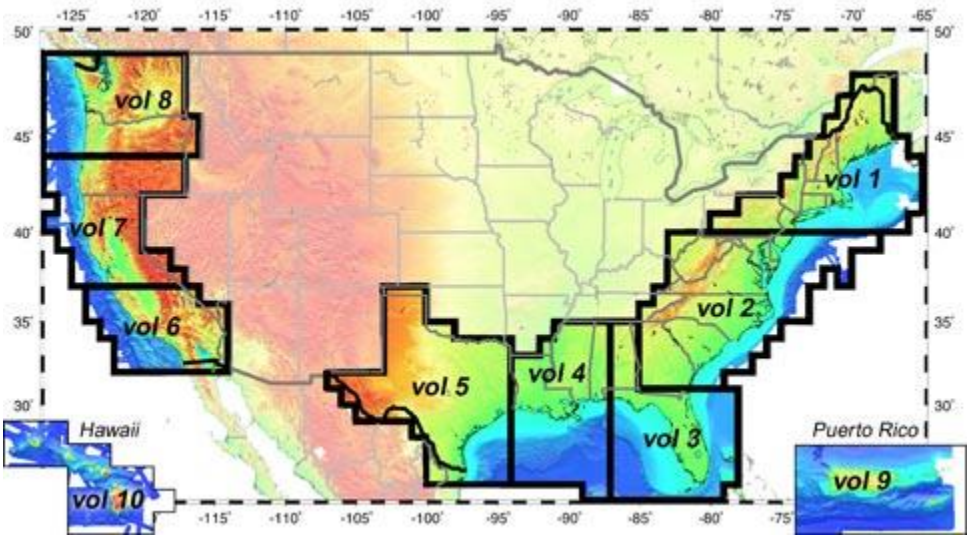


Figure 3.1. Image of NGDC Coastal Relief Model Dataset.

Method

This research focuses on the calculation of coastal inundation height, i.e., the minimum sea level rise required to inundate a cell. This operation is a specialized version of cost distance function, which has a few caveats that allow for greater optimization than standard cost distance computations (Li, Grady, & Peterson, 2014). The cost equation used in calculating inundation height is:

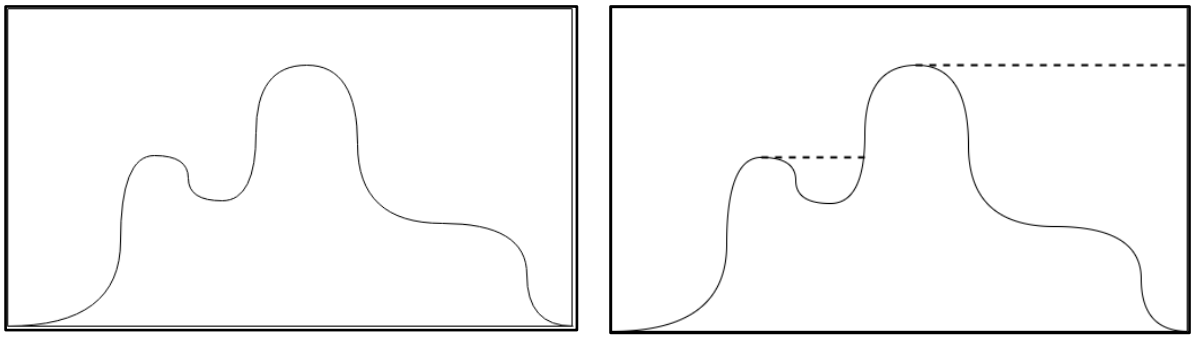
$$cost = \max(inundation\ height\ of\ lowest\ neighbor, elevation\ of\ cell) \quad (3.1)$$

Coastal inundation height was chosen for this research due to the simplicity of the cost equation (3.1) and the limits of the methods in processing large DEM (Li, Grady, & Peterson, 2014). It would not take much effort, however, to generalize this method for other propagation functions.

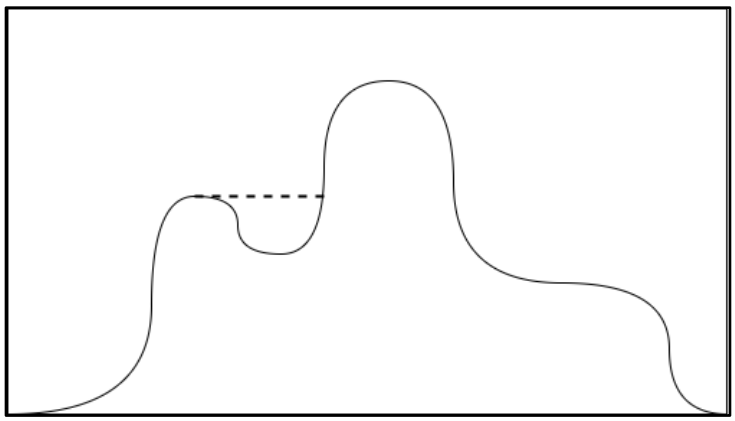
Parallelizing the calculation of coastal inundation height relies on three core concepts in order to break free of the limits constraining the serial implementations based on Dijkstra's algorithm. The first concept in our parallelization is region decomposition. To enable parallelization and to limit computational resources required at any given time, a study area is decomposed into smaller sub-regions. This is similar to the region decomposition approach used by Sun and Sun (2016) but the sub-graphs generated from a raster dataset are more dense, regularly-spaced, and uniform, which offers other advantages for parallelization.

The second concept is a calculate-and-correct approach, where calculations are performed with the information available at the time they run and then corrected when new information is available. The re-calculations are limited so that they stop when there are no additional updates to perform. The calculate-and-correct approach can be visualized using a profile of a sample

surface as illustrated in Figure 3.2(a). In this example, the profile is inundated from the left by sea level rise. Since inundation height is the maximum between the elevation and the minimum sea level rise to inundate an adjacent cell (Equation 3.1), it follows the profile until the first depression from the left is reached. At that point, the inundation height remains constant as the depression is filled until a greater elevation is reached. The inundation height follows the profile again until it reaches its highest point. Then, for this left inundation of the profile, the inundation height remains constant from that point to the right edge. Figure 3.2(b) illustrates the inundation height on the profile calculated by sea level rise from the left, where dotted lines indicating locations where the inundation height computed is greater than the elevation of those locations. Next, consider that the profile is inundated again from the right. Moving from right to left along the profile, the inundation height follows the profile until it reaches the peak. Inundation height is not further reduced moving to the left, so the calculations stop from this direction. The result is an inundation height represented by the profile shown in Figure 3.2(c). This example illustrates how the calculate-and-correct approach works. Inundation heights are calculated using the available ocean fronts at any given iteration and it is then updated if new ocean fronts arise in future iterations.



(a) (b)



(c)

Figure 3.2. (a) A profile of a sample surface; (b) Inundation height calculated from left inundation; (c) Updated inundation height after inundation from the right.

For a raster dataset which is decomposed into regions, only source cells located within a region are used for computation at the beginning. It is possible, and likely, that some regions cannot be calculated because they do not have source cells. However, the regions that do have source cells can carry out the computation. Once the computation has completed for the initial run over a region, the edges of the region can be used as source cells for adjacent regions. Figure 3.3 shows how the edge of one tile is exported and added to an adjacent tile for further computations. In the figure, the right edge of the left tile is exported and added as the left edge of the right tile (shown in blue). This approach allows the calculations to propagate to the regions without initial source cells. As the computations continue, calculated regions may be reached from other regions with a potentially lower inundation height. Once this happens, the inundation heights of the region are re-computed using the edge from the adjacent region(s) ("calculate-and-correct"). Using these values, the calculations are run again but stop when no additional heights are reduced. This allows for the total number of computations to be reduced, as only changed cells are visited and calculations propagate to adjacent regions only if edge inundation heights are

changed. Computations for the entire area continue until no additional propagations occur among the regions.

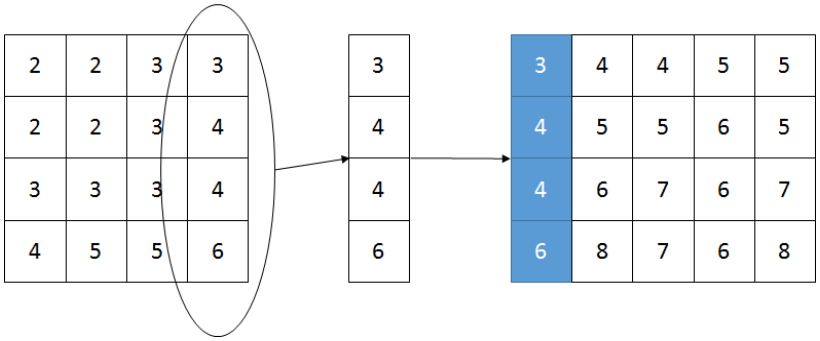


Figure 3.3. Export vector as the source cells of an adjacent tile

By splitting large grids into smaller grids, the resources needed at any given time can be limited. Potentially, multiple region computations can be fit on a single machine to take advantage of multiple processing cores, achieving the third core concept of parallelization. For this work, we accomplish parallelism by deploying worker processes on each of the available processing cores that connect to the master process. These workers perform calculations for a single tile at a time, send the results back to the master process, and then request more work. Parallelization can be taken a step farther by addressing how the computations are arranged. This means that the architecture of the environment where the computations are run can be taken advantage of to potentially distribute the computations across multiple machines.

By using region decomposition and calculate-and-correct, these calculations can be parallelized at the region level, be that a tile, or potentially within a tile. It is more obvious how the parallelization takes place between tiles. Each tile is stored as individual files and then each instance of a computational worker will load a single raster tile. The computations can be done one at a time on a single machine. This is useful for commodity hardware that may not have a

lot of processing power. Computations will be slower, but it is still possible to perform the calculations on a large dataset that would be impossible with methods that need to load the entire study area at one time.

A complete example

Here we use an example to show how region decomposition, parallelism, and calculate-and-correct, work together to perform the computation. The initial ten cell by ten cell surface is split into tiles, or regions. Each of the regions is operated on at the same time for parallelism. Finally, the results of each of the region computations are propagated to adjacent regions and used as new source cells to make corrections to the computed values for those regions if necessary. For this example, intermediate results are shown in discrete iterations. In reality, computations can begin on adjacent tiles as soon as they finish for a tile.

Figure 3.4 shows the example surface with each value representing the elevation value a cell. Elevation values of zero are used as source cells in this example.

3	5	3	2	1	1	0	1	2	2
2	2	3	4	4	5	4	1	2	3
2	1	9	9	7	9	8	5	5	4
0	1	8	3	3	2	7	4	6	4
1	3	7	3	3	2	9	8	7	5
1	4	6	4	3	3	2	8	4	4
1	3	5	2	2	2	2	2	2	3
1	1	6	2	2	2	9	2	2	3
1	1	1	8	9	8	6	1	1	1
0	1	1	2	2	1	1	1	1	1

(a)

3	5	3	2	1	1	0	1	2	2
2	2	3	4	4	5	4	1	2	3
2	1	9	9	7	9	8	5	5	4
0	1	8	3	3	2	7	4	6	4
1	3	7	3	3	2	9	8	7	5
1	4	6	4	3	3	2	8	4	4
1	3	5	2	2	2	2	2	2	3
1	1	6	2	2	2	9	2	2	3
1	1	1	8	9	8	6	1	1	1
0	1	1	2	2	1	1	1	1	1

(b)

Figure 3.4. An example surface and its decomposition

The first step is to perform a decomposition of the surface. For this example, the surface is split into four equally sized five cell by five cell regions. But it is not required that the regions are the same size. An exploded view of the surface (Figure 3.5) will be used for visual clarity.

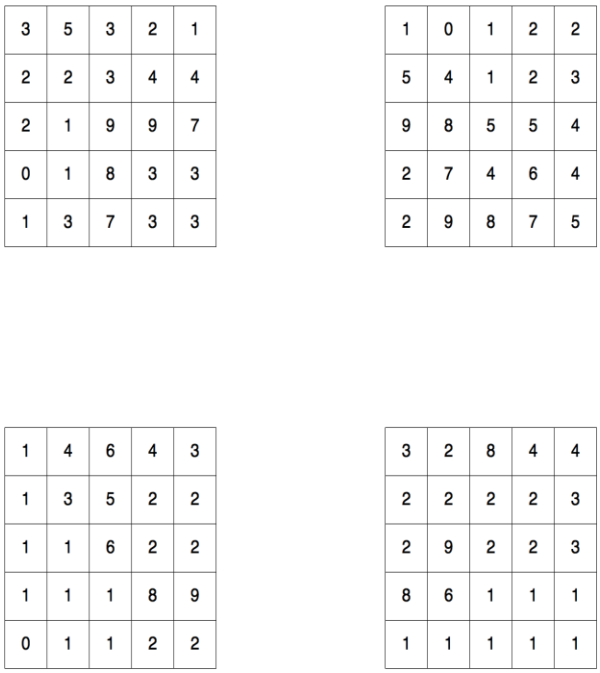


Figure 3.5. Exploded view of region decomposition

Now that the surface is divided into regions, the computation can be run on each region, concurrently, starting from the source cells (i.e., cells with elevation values of zero) within each

region. Figure 3.6 shows the first iteration of calculations of inundation height on each of the regions.

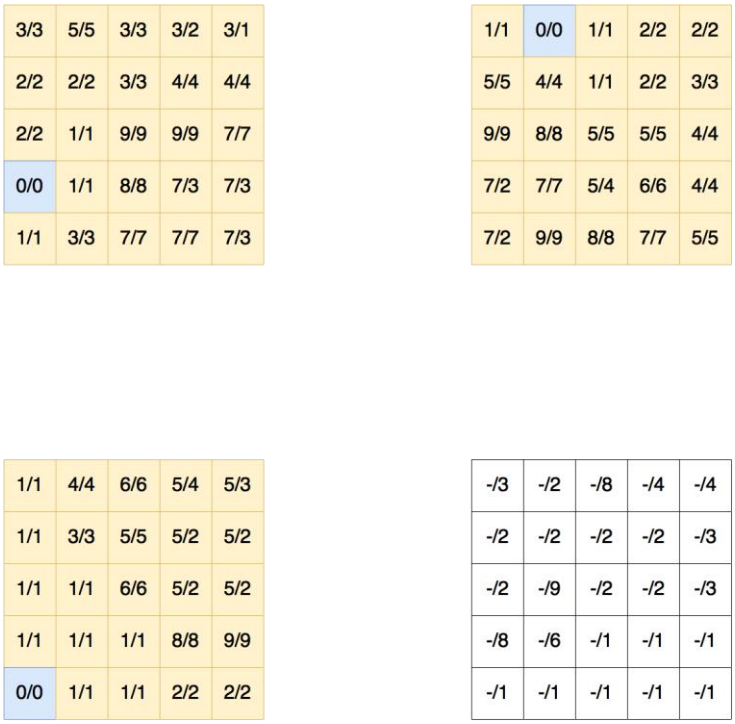


Figure 3.6. First iteration of example computations. Each tile is explored for source cells and any found are used for computations.

Three of the regions have source cells (one in each) which are shown in blue. Cells with changed inundation height values in any given iteration are shown in yellow and at the end of the first iteration, every cell in the regions that have source cells have been changed. White cells did not change in this iteration. The value in each cell is the sea level rise required to inundate a particular cell from the information currently available. These cell values are stylized as the current inundation height on the left, a slash, followed by the elevation of the cell on the right. This provides a clear visual representation of the cells that have inundation heights that are greater than the elevation of the cells and therefore could potentially have their values reduced.

In the first iteration, the top-left, top-right, and bottom-left regions had a single source cell and every cell in those regions were inundated. The bottom-right region does not have a source cell, therefore, none of the cells were inundated in the first iteration.

3/3	5/5	3/3	2/2	1/1	1
2/2	2/2	3/3	4/4	4/4	5
2/2	1/1	9/9	9/9	7/7	9
0/0	1/1	8/8	5/3	5/3	7
1/1	3/3	7/7	7/7	5/3	7
1	4	6	5	5	

3	1/1	0/0	1/1	2/2	2/2
4	5/5	4/4	1/1	2/2	3/3
7	9/9	8/8	5/5	5/5	4/4
7	7/2	7/7	5/4	6/6	4/4
7	7/2	9/9	8/8	7/7	5/5

1	3	7	7	7
1/1	4/4	6/6	5/4	5/3
1/1	3/3	5/5	5/2	5/2
1/1	1/1	6/6	5/2	5/2
1/1	1/1	1/1	8/8	9/9
0/0	1/1	1/1	2/2	2/2

	7	9	8	7	5
5	3/3	2/2	8/8	4/4	4/4
5	2/2	2/2	2/2	2/2	3/3
5	2/2	9/9	2/2	2/2	3/3
9	8/8	6/6	2/1	2/1	2/1
2	2/1	2/1	2/1	2/1	2/1

Figure 3.7. Second iteration of example computations. Vectors exported from adjacent regions are represented in blue. Cells changed are in yellow.

In the second iteration, shown in Figure 3.7, and every subsequent iteration, source cells are no longer determined by inspecting a region looking for specific values, and instead come from edge vectors exported from previous computations. This is how computations, inundation height in this example, propagate between tiles and enable calculate-and-correct. The above Figure 3.7 shows how the edge vectors from adjacent regions are used as source cells. These values come from adjacent regions and are shown as blue rows and columns. In this iteration, the top-left region receives source cells from the top-right region and the bottom-left region. These source values result in five cells changing in the top-left region. The top-right and bottom-left regions

receive source cells from the top-left region, but these source cells do not result in any changes for either region. The bottom-right region, which was not inundated in the first iteration, receives source cells from the top-right and bottom-left regions and all of the cells are inundated from these source cells.

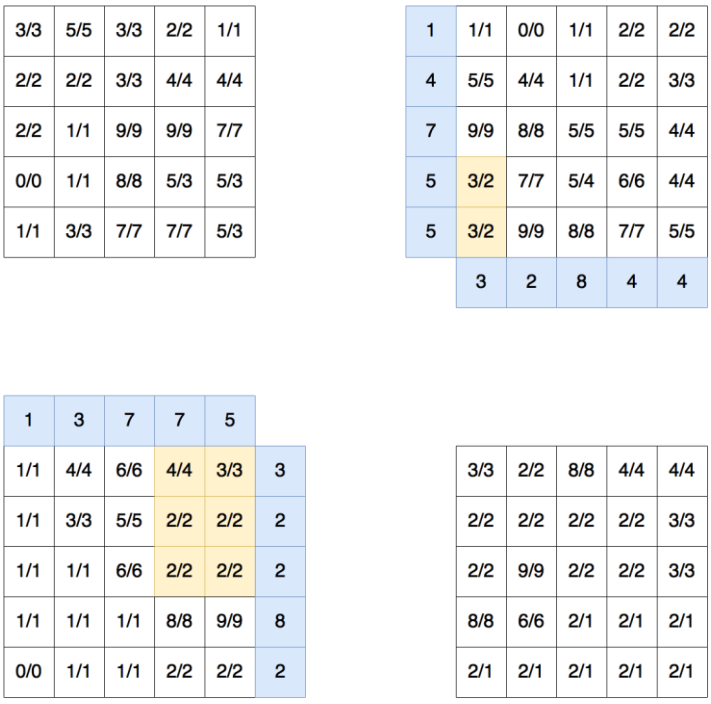


Figure 3.8. Third iteration of example computations. Only the top-right and bottom-left tiles are active in this iteration.

Figure 3.8 shows the third iteration of computations. Since the top-right and bottom-left regions did not have any changes in the second iteration, they do not export source edges in the third iteration. They do, however, receive source edges from the top-left and bottom-right regions. These source edges result in changes in six cells in the bottom-left region and two cells in the top-right region.

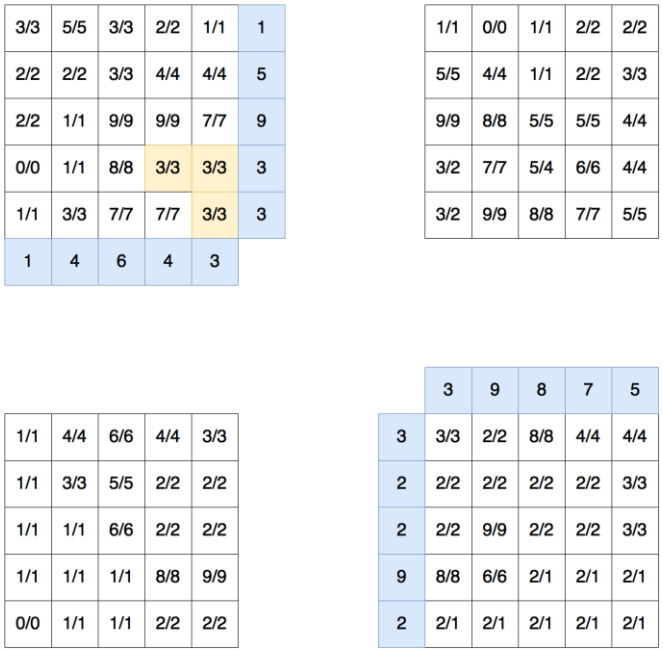


Figure 3.9. Results of fourth iteration of example computations. Cells are changed in the top-left tile but none are changed in the bottom-right tile.

The bottom-left and top-right regions produced source edges that are sent to the top-left and bottom-right regions in the fourth iteration. None of the inundation heights change in the bottom-right region and the inundation height of three cells is reduced in the top-left region (Figure 3.9).

3/3	5/5	3/3	2/2	1/1
2/2	2/2	3/3	4/4	4/4
2/2	1/1	9/9	9/9	7/7
0/0	1/1	8/8	3/3	3/3
1/1	3/3	7/7	7/7	3/3

1	1/1	0/0	1/1	2/2	2/2
4	5/5	4/4	1/1	2/2	3/3
7	9/9	8/8	5/5	5/5	4/4
3	3/2	7/7	5/4	6/6	4/4
3	3/2	9/9	8/8	7/7	5/5

1	3	7	7	3
1/1	4/4	6/6	4/4	3/3
1/1	3/3	5/5	2/2	2/2
1/1	1/1	6/6	2/2	2/2
1/1	1/1	1/1	8/8	9/9
0/0	1/1	1/1	2/2	2/2

3/3	2/2	8/8	4/4	4/4
2/2	2/2	2/2	2/2	3/3
2/2	9/9	2/2	2/2	3/3
8/8	6/6	2/1	2/1	2/1
2/1	2/1	2/1	2/1	2/1

Figure 3.10. The fifth iteration does not change any cells so the computations are finished.

Since only the top-left region produced source edges in the fourth iteration, the bottom and right edges from the region are exported to the top-right and bottom-left regions. Figure 3.10 shows that these source vectors do not result in any changes in the fifth iterations and therefore the computations are complete.

3/3	5/5	3/3	2/2	1/1	1/1	0/0	1/1	2/2	2/2
2/2	2/2	3/3	4/4	4/4	5/5	4/4	1/1	2/2	3/3
2/2	1/1	9/9	9/9	7/7	9/9	8/8	5/5	5/5	4/4
0/0	1/1	8/8	3/3	3/3	3/2	7/7	5/4	6/6	4/4
1/1	3/3	7/7	7/7	3/3	3/2	9/9	8/8	7/7	5/5
1/1	4/4	6/6	4/4	3/3	3/3	2/2	8/8	4/4	4/4
1/1	3/3	5/5	2/2	2/2	2/2	2/2	2/2	2/2	3/3
1/1	1/1	6/6	2/2	2/2	2/2	9/9	2/2	2/2	3/3
1/1	1/1	1/1	8/8	9/9	8/8	6/6	2/1	2/1	2/1
0/0	1/1	1/1	2/2	2/2	2/1	2/1	2/1	2/1	2/1

Figure 3.11. Completed computations using parallel computation method.

Comparing the inundation height surface produced by the parallel approach (Figure 3.11) and the surface generated by the traditional serial technique shows that the results are the same. This indicates that our method produces correct results with the parallel, calculate-and-correct approach.

Handling tiles with different spatial resolutions

The edges of each inundation height tile are exported to adjacent tiles for computations. For large surfaces distributed as tiles, adjacent tiles may have different spatial resolutions. This happens with the NGDC dataset where the resolution of nine volumes is three arc seconds and the resolution of one volume is one arc second. Without special handling, it would require that the resolution of the entire surface be artificially increased to one arc second or for the higher resolution dataset to be resampled down to three arc seconds to use our method.

We handle this situation by modifying the adjacency vectors as necessary. If the adjacent tile has a higher resolution, the adjacency vector is densified and the new cell values are interpolated from the adjacency vector. If the adjacency vector has a higher resolution than the adjacent tile, the vector reduced to match the adjacent tile. The new adjacency vector is defined by Equation 3.2.:

$$newVector[i] = oldlVector\left[int(\frac{oldVectorLength*i}{newVectorLength})\right], for\ i = 0\ to\ newVectorLength - 1 \qquad (3.2)$$

Equation 3.2 is a simple scheme that does not interpolate between values, but other interpolation schemes are possible. Figure 3.12 shows how adjacency vectors may be contracted (on the left) or expanded (on the right) using nearest-neighbor to determine cell values.

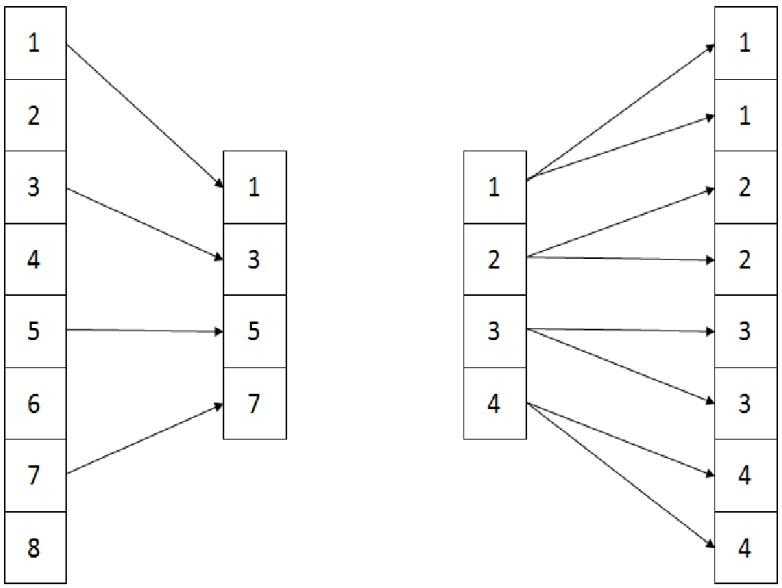


Figure 3.12. Example of contracting or expanding exported vectors as necessary. The right example shows a simple contraction of an exported vector from eight cells to four. The right side shows an expansion from four cells to eight.

Implementation

The approach described above has been implemented using the Python programming language (http://python.org, version 2.7). Python was chosen because of the availability of libraries that were utilized in the implementation (NumPy and Work Queue) and author familiarity. Integration tools through GitHub, such as Travis CI and ability to automatically test code commits are also attractive reasons for choosing Python. Multiple other languages, such as C / C++, would have been appropriate, but Python allowed for faster development and demonstration of the validity of the method.

At the computational level, raster tiles are read and stored as ASCII grids. This was done for simplicity and ease of debugging but should be extended in the future to other raster formats. The raster files are read into, and manipulated as, NumPy arrays (van der Walt, Colbert, & Varoquaux, 2011) because they can be addressed easily and they are a more optimized data structure than a traditional Python list. The source code for the project is available as a GitHub repository (https://github.com/cjgrady/parallel-cost-distance). Documentation is available as part of the repository and by utilizing Travis CI, build testing and test code coverage information is available at a glance from the GitHub repository page.

A pool of a configurable number of workers is created and calculations are performed on each region in a tile. As calculations complete, calculations for adjacent regions are submitted to the pool for computation when worker threads become available.

For multiple-tile parallelism, computations are managed by using Work Queue. Work Queue is part of the CCTools package produced by The Cooperative Computing Lab at the University of

Notre Dame, Indiana (Bui, Rajan, Abdul-Wahid, Izaguirre, & Thain, 2011). Work Queue provides a master / worker paradigm for computations and handles the communication between the workers and the master process. Work Queue allows for workers to be distributed across a single or multiple machines and still connect to a single master process. As the calculations on a single tile complete, any edge that was altered is exported as a registered Work Queue output to the master process. A controlling process takes these exported edges and initializes computation on the tile adjacent to that edge. These computations are submitted as tasks to a Work Queue master process and the computation is picked up when a worker is available.

We carried out several experiments to show under what, if any, scenarios this implementation performs better than existing, serial, implementations. Better can be defined as running in less time, a smaller memory footprint, or potentially measured with other metrics. The primary metric that will be used for comparison for these experiments is running time with additional commentary about memory footprint where appropriate.

Results

To evaluate the effectiveness of the proposed method, an experiment was designed to gather benchmarks for comparison, specifically, total running time and memory footprint. The independent variables for these experiments were the size of the regions within the study area and the number of workers (one per CPU) working on the experiment at once.

The NGDC dataset is split into ten volumes. Nine of these volumes are organized as a single raster for each volume and the last volume is split into one degree by one degree raster files. A serial implementation of Dijkstra's algorithm cannot operate on the entire dataset in that

condition so the raster files were merged into a single raster for computation. The merged raster file, in ASCII grid format, was 25 gigabytes in size and caused the serial implementation of Dijkstra's algorithm to fail because the machine ran out of memory. The experimental machine has twelve processing cores and 32 gigabytes of primary memory with an additional 32 gigabytes of swap memory. The serial experiment with the entire raster used all 64 gigabytes of available memory before the process was killed by the operating system.

West Coast Experiment

In response to the failure with the serial implementation of Dijkstra's algorithm using a raster that covers the entire study area, smaller raster files were created by limiting the data to each coast of the Continental United States. The resulting files were 6.8 gigabytes for the east coast and 6.5 gigabytes for the west coast. The east coast dataset was tested with the serial implementation and it used all of the memory available on the system and failed again. The west coast dataset was tested and it was able to fit into approximately 60 gigabytes of memory and the inundation height for the western dataset was computed in 34 minutes and 36 seconds. Running time was determined by a timer embedded in the software starting the clock before starting the computations and stopping when they are finished and memory footprint was determined by watching the system monitor as the measurement did not need to be exact and assessing the memory footprint of the parallel application is difficult without additional overhead.

For the parallel implementation, the western United States dataset was divided into half degree, one degree, and two degree tiles. The experiments were all run on the same machine that produced the initial benchmark for the serial experiment but with one, two, four, eight, and twelve workers / CPUs. The goal of this experiment was to determine if there is an optimal

configuration of tile size / number of workers to perform the computations as fast as possible with a manageable memory footprint.

The savings on memory footprint of the experiment proved to be very significant. For two degree tiles the memory footprint was less than one gigabyte per core, a very manageable amount for current hardware. For smaller tiles the memory to processing core ratio was smaller still. This showed that memory footprint would not be an issue with the tile sizes chosen for these experiments and therefore memory monitoring was discontinued for future experiments.

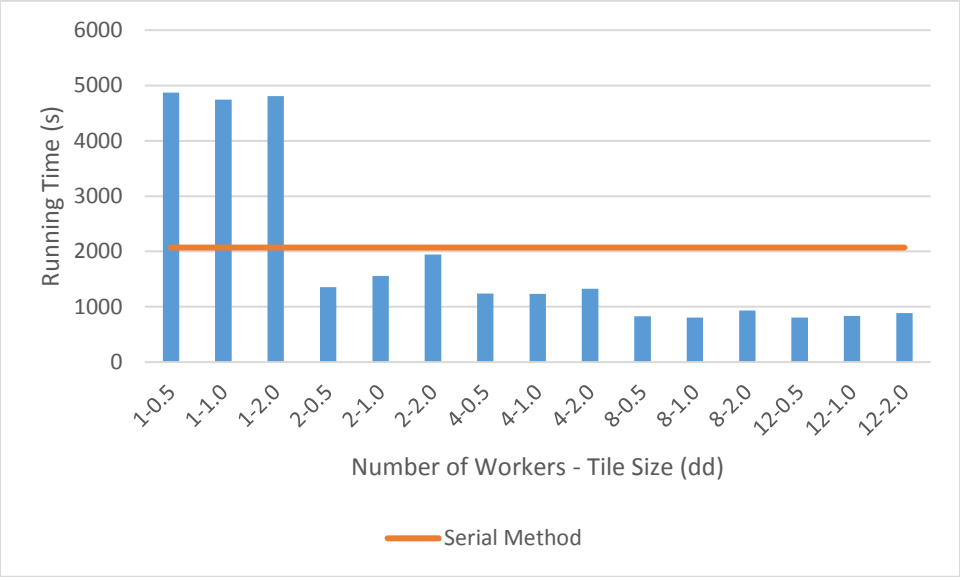


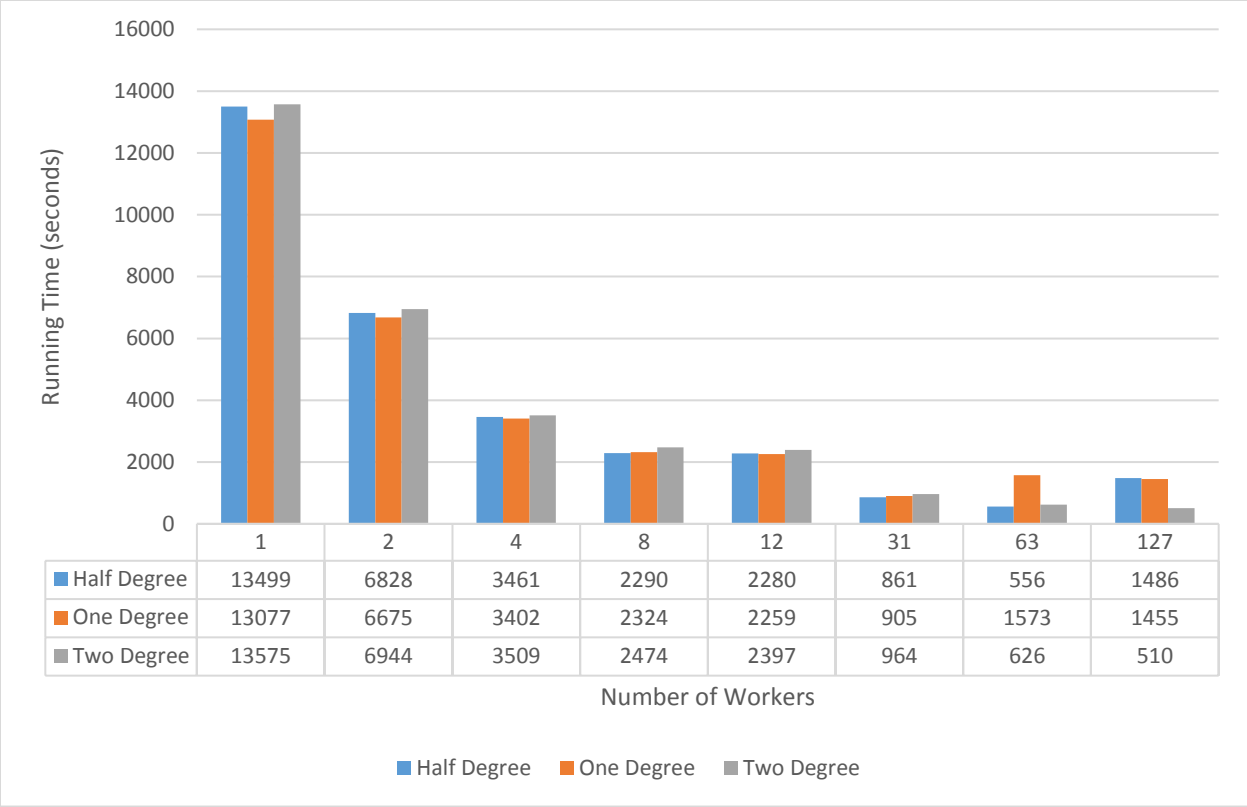
Figure 3.13. Running time of west coast dataset using various parallel configurations. The red line is the running time using serial computations.

Figure 3.13 shows that running time was reduced each time the number of workers was increased, however the pattern between the various tile sizes was not completely consistent. Unsurprisingly, the one worker configuration was consistently slower than the serial method. This can be explained by the overhead of reading and writing files multiple times. As seen in the supplemental materials, each tile is read approximately three times. While each file read and

write is smaller than the serial method, the aggregate reads and writes are three times that of the serial method and is a significant source of overhead. This overhead is overcome with additional parallelism, but the difference between eight and twelve workers is not as drastic as the difference between other numbers of workers. Additional benchmarks would be needed to determine if this is due to limits of the hard disk hardware, processing at the management layer, or other factors. The relationship between tile sizes generally favored half or one degree tiles but it was not always the same between experiments. Full results for this experiment can be found in Appendix A.

Entire NGDC Dataset Experiment

A second experiment was performed over the entire NGDC dataset to determine if the same patterns hold for running time. Additionally, the application was deployed on XSEDE (Towns, et al., 2014) resources, specifically the Stampede Supercomputer at The Texas Advanced Computing Center, to see if and how those patterns change with multiple machines, more processing cores, and a dedicated, parallel file system. The same tile sizes were used for this experiment with one, two, four, eight, and twelve processing cores on the single machine and 32, 64 and 128 cores (two, four, and eight machines) on Stampede. In the Stampede experiments, one core was reserved for the master process, leaving 31, 63, and 127 cores for workers.



	1	2	4	8	12	31	63	127
Half Degree	13499	6828	3461	2290	2280	861	556	1486
One Degree	13077	6675	3402	2324	2259	905	1573	1455
Two Degree	13575	6944	3509	2474	2397	964	626	510

Figure 3.14. Running time for computations over entire NGDC dataset.

The results of this experiment, available in Appendix B and summarized in Figure 3.14, are similar to those of the west coast dataset. The total running time decreases as the number of workers increased but the difference between eight and twelve workers is significantly less than the difference between the previous concurrency steps for single machine experiments.

Results from Stampede machines show that running time generally decreases as more workers are added, though with some exceptions. The inconsistencies are likely due to a bottleneck with the Lustre file system on Stampede. Stampede is a community resource, and while users are allocated entire machines for each job, jobs are not given exclusive access to the parallel file system and thus performance is likely influenced by other user tasks as well as the concurrency

within the user's application. There are many articles discussing the performance of Lustre file systems under concurrent load but that is outside of the scope of this research.

It is interesting to see large increases in running time in the Stampede experiments as well as continued improvements in running time. Unfortunately, performance information, more than running time, was not collected about the file system that could reveal what other factors may contribute to the increase in running time in some of the experiments. This is an area for future exploration to determine if there are configuration changes that can be made for more predictable performance on Stampede or if a different XSEDE resource may be more suitable for this application. Overall, performance gains are very promising. Half degree tiles had fastest running time with 63 cores at 556 seconds. One degree tiles were fastest at 31 workers with 905 seconds. Finally, two degree tiles performed best with 127 workers at 510 seconds. Comparing these with the single worker running times of 13499 seconds, 13077 seconds, and 13575 seconds, this represents a speed up 24.3x, 14.4x, and 26.6x for half degree, one degree, and two degree tiles respectively.

Conclusions

To claim that this research is an appreciable improvement upon a standard implementation of Dijkstra's algorithm for calculating inundation height, the results must show that at least one of the following is true: (1) the computations run in less time, (2) the computations require fewer resources to run, or (3) new computations are enabled that were not previously possible. Our results show that this research meets all three criteria.

This parallel implementation of inundation height calculations runs in significantly less time than a serial implementation as the number of processing cores increases. For the west coast raster surface, the serial calculations ran in 34 minutes and 36 seconds. Of the parallel configurations, the half degree tiles and twelve processing cores combination gave the best running time of 13 minutes and 24 seconds. The parallel approach ran in less than half of the time required for the serial approach and these experiments were only done on a single machine. In fact, all experiments that utilized two or more processing cores ran in less time than the serial experiment. Experiments running with only a single processing core did run in more time than the serial method due to the overhead added for calculate-and-correct computations.

The serial version of the west coast experiment required approximately 60 gigabytes of memory to perform the calculations. Each of the parallel experiments required a gigabyte or less of memory per core. The exact value was determined by the tile sizes with the maximum occurring with the two degree tiles. With 12 processing cores, the memory required at any given time is 12 gigabytes or less. This number can be reduced by utilizing smaller tiles or fewer processing cores, however, one gigabyte per core is a reasonable ratio with current hardware. An interesting experiment would be to configure a tile size such that the required memory per core more closely matches the memory per core available on the machine.

The third criterion is met because the research enabled computations that were larger than previously possible. This is directly related to the second criterion as the memory required to perform computations is reduced. This is shown by the experiment over the entire NGDC dataset. With a merged raster size of 25 gigabytes, the memory required to perform the calculations for the serial implementation was more than what was available on the machine. While there may be “high memory” machines with enough local memory to perform these

calculations, the memory per core balance would be very poor at potentially hundreds of gigabytes needed with only a single processing core utilized. Furthermore, a raster size limit still exists that would not fit, even on these high memory machines. Our approach also allows for computations over raster tiles with variable resolutions. Serial implementations require a single resolution on a merged raster, thus requiring at least a portion of the tiles to be resampled to fit. Our approach adjusts the resolution of the individual tiles at computation time.

This research also enables the deployment of our approach on highly-parallel XSEDE resources, such as Stampede at TACC. This takes the parallelization beyond what is available on a single machine and allows it to scale to much higher levels of concurrency as the calculations can potentially expand to more than one supercomputing resources through the use of Work Queue. The experiments show that these resources present different bottlenecks to overcome, such as parallel disk performance. Addressing these and optimizing the calculations is still an open question. Even with the new bottlenecks, deploying this tool on Stampede provided a very significant improvement on running time on the NGDC dataset, more than 26 times faster at peak.

While we have accomplished our goals with respect to inundation height computations, we still need to expand our development to work for the cost distance function and other raster propagation functions. We would also like to experiment with GPU computations for individual tiles, most likely by utilizing the Bellman-Ford algorithm rather than Dijkstra's algorithm. Furthermore, continued research is needed to determine which, if any, of the other XSEDE resources would provide a better architecture, especially with respect to parallel I/O, for our computations in order to enable additional parallelism and improved performance.

Chapter 4: Conclusions

We have shown that our approaches overcome the limitations of previous approaches for determining coastal inundation due to sea-level rise. First, we presented a new approach for computing the minimum sea level rise needed to inundate every cell in a raster, addressing the limits of previous techniques that only reported what cells are inundated by specific sea-level rises. We then addressed the performance limitations of our approach by creating a parallel version that is able to run across processing cores on a single machine as well as be deployed on supercomputing resources.

The second chapter discussed our approach to determining the minimum sea-level rise required to inundate every cell for a surface and explored the effects of different data structures on performance. We showed that by treating a raster as a graph, with each cell being a node and adding edges between the adjacent cells, Dijkstra's algorithm could be used to create a surface representing the minimum sea-level rise required to inundate each cell. This allows the computations to be run one time to determine the inundation height for every cell. This is an improvement upon older methods that only answered the question of which cells were inundated at a specific sea-level rise and therefore required the computations to be run for every sea-level rise in question.

We then explored the performance effects of different priority queue implementations, binary and Fibonacci heaps. We found that, while Fibonacci heaps have better theoretical performance for many operations, in practice, those operations were not used in our calculations. We determined that the remove minimum operation, called once for each cell in a raster, had the

same order of asymptotic performance (Big O) but the constant associated with binary heaps is smaller, resulting in faster running times for binary heaps.

In the third chapter, we presented a new parallel implementation of calculating inundation height for the NGDC dataset. Our method was shown to be an effective alternative to a traditional implementation of Dijkstra's algorithm when used for inundation height calculations. We showed that our parallel approach enabled us to work with larger datasets than traditional methods and it is able to complete the computations in less time. We then deployed the method on XSEDEs Stampede Supercomputer at the Texas Advanced Computing Center where we explored how our method can be scaled. We were able to run our method on multiple machines and see performance improvements. While we encountered new bottlenecks to continued performance improvement, we were able to achieve a peak performance that was 26 times faster than using a single core.

We have shown that our approaches provide effective methods for determining the inundation height for all cells in a raster. Moreover, we have presented a parallel method that addresses the performance limitations and allows for the method to be deployed in multi-processor and supercomputing resources. The processing time savings that we were able to achieve show that we were able to effectively incorporate parallelism into the inundation height computations and allow us to explore additional supercomputing resources for future performance improvements.

References

Bellman, R. E. (1958). On a routing problem. *Quarterly of applied mathematics, 16*(1), 87-90.

Bindschadler, R. (1998). Future of the West Antarctic ice sheet. *Science, 282*(5388), 428-429.

Bui, P., Rajan, D., Abdul-Wahid, B., Izaguirre, J., & Thain, D. (2011). Work queue+ python: A framework for scalable scientific ensemble applications. *Workshop on python for high performance and scientific computing at sc11*.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Third ed.). Cambridge, Massachusetts: The MIT Press.

Craft, C., Clough, J., Ehman, J., Joye, S., Park, R., Pennings, S., . . . Machmuller, M. (2009). Forecasting the effects of accelerated sea-level rise on tidal marsh ecosystem services. *Frontiers in Ecology and the Environment, 7*(2), 73-78.

Crauser, A., Mehlhorn, K., Meyer, U., & Sanders, P. (1998). A parallelization of Dijkstra's shortes path algorithm. *Mathematical foundations of computer science 1998*, 722-731.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik, 1*(1), 269-271.

Ford Jr, L. R. (1956). *Network flow theory.* DTIC Document.

Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM), 34*(3), 596-615.

Gao, P., & Sudhakar, M. (1994). A Divide and Iterate Approach to a Class of Propagation Functions. *Advances in GIS Research: proceedings of the Sith International Symposium on Spatial Data Handling*, pp. 117-189.

Goldberg, A. V., & Tarjan, R. E. (1996). Expected performance of Dijkstra's shortest path algorithm. *NEC Research Institute Report*.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, *4*(2), 100-107.

Hopkinson, C. S., Lugo, A. E., Alber, M., Covich, A. P., & Van Bloem, S. J. (2008). Forecasting effects of sea-level rise and windstorms on coastal and inland ecosystems. *Frontiers in Ecology and the Environment*, *6*(5), 255-263.

Intergovernmental Panel On Climate Change -- IPCC. (2007). Climate Change 2007: The physical science basis: Summary for policymakers. *Intergovernmental Panel on Climate Change, Geneva, Switzerland*.

Jasika, N., Alispahic, N., Elma, A., Ilvana, K., Elma, L., & Nosovic, N. (2012). Dijkstra's shortest path algorithm serial and parallel execution performance analysis. *MIPRO, 2012 proceedings of the 35th international convention* (pp. 1811-1815). IEEE.

Legra, L., Li, X., & Townsend, P. A. (2008). Biodiversity consequences of sea level rise in New Guinea. *Pacific Conservation Biology, 14*(3), 191-199.

Li, R., Hu, H., Li, H., Wu, Y., & Yang, J. (2016, August). MapReduce Parallel Programming Model: A State-of-the-Art Survey. *International Journal of Parallel Programming, 44*(4), 832-866. doi:10.1007/s10766-015-0395-0

Li, X., Grady, C., & Peterson, A. T. (2014). Delineating Sea Level Rise Inundation Using a Graph Traversal Algorithm. *Marine Geodesy*, *37*(2), 267-281.

Li, X., Larson, C. M., & Rex, A. B. (2005). Creating buffers on surfaces. *Cartography and Geographic Information Science, 32*(3), 195-212.

Li, X., Rowley, R. J., Kostelnick, J. C., Braaten, D., Meisel, J., & Hulbutta, K. (2009). GIS analysis of global impacts from sea level rise. *Photogrammetric Engineering & Remote Sensing*, *75*(7), 807-818.

Marks, K., & Bates, P. (2000). Integration of high-resolution topographic data with floodplain flow models. *Hydrological Processes*, *14*(11-12), 2109-2122.

Menon, S., Soberón, J., Li, X., & Peterson, A. T. (2010). Preliminary global assessment of terrestrial biodiversity consequences of sea-level rise mediated by climate change. *Biodiversity and Conservation*, *19*(6), 1599-1609.

Meyer, U., & Sanders, P. (2003). Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, *49*(1), 114-152.

Moore, E. F. (1959). *The shortest path through a maze.* Bell Telephone System.

National Research Council. (2011). *The Future of Computing Performance: Game Over or Next Level?* National Academies Press.

Nicholls, R. J., & Tol, R. S. (2006). Impacts and responses to sea-level rise: a global analysis of the SRES scenarios over the twenty-first century. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 364*(1841), 1073-1095.

Nicholls, R. J., Marinova, N., Lowe, J. A., Brown, S., Vellinga, P., De Gusmao, D., . . . Tol, R. S. (2011). Sea-level rise and its possible impacts given a 'beyond 4 C world' in the twenty-first century. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 369*(1934), 161-181.

NOAA National Geophysical Data Center. (2014). *Coastal Relief Model*. Retrieved from http://www.ngdc.noaa.gov/mgg/coastal/crm.html

Overpeck, J. T., Otto-Bliesner, B. L., Miller, G. H., Muhs, D. R., Alley, R. B., & Kiehl, J. T. (2006). Paleoclimatic evidence for future ice-sheet instability and rapid sea-level rise. *Science, 311*(5768), 1747-1750.

Peterson, A. T., Navarro-Sigüenza, A. G., & Li, X. (2010). Joint effects of marine intrusion and climate change on the Mexican avifauna. *Annals of the Association of American Geographers, 100*(4), 908-916.

Poulter, B., & Halpin, P. N. (2008). Raster modeling of coastal flooding from sea-level rise. *International Journal of Goegraphical Information Science, 22*, 167-182.

Rignot, E., & Kanagaratam, P. (2006). Changes in the velocity structure of the Greenland Ice Sheet. *Science, 311*, 986-990.

Solka, J. L., Perry, J. C., Poellinger, B. R., & Rogers, G. W. (1995). Fast computation of optimal paths using a parallel Dijkstra algorithm with embedded constraints. *Neurocomputing, 8*(2), 195-212.

Sun, J., & Sun, G. (2016). SPLZ: An efficient algorithm for single source shortest path problem using compression method. *GeoInformatica, 20*(1), 1-18.

Thomas, R., Rignot, E., Casassa, G., Kanagaratnam, P., Acuña, C., Akins, T., . . . Krabill, W. (2004). Accelerated sea-level rise from West Antarctica. *Science, 306*(5694), 255-258.

Towns, J., Cockerill, T., Dahan, M., Foster, I., Gaither, K., Grimshaw, A., . . . Peterson, G. D. (2014). XSEDE: accelerating scientific discovery. *Computing in Science & Engineering, 16*(5), 62-74.

Träff, J. L., & Zaroliagis, C. D. (1996). A simple parallel algorithm for the single-source shortest path problem on planar digraphs. *International Workshop on Parallel Algorithms for Irregularly Structured Problems* (pp. 183-194). Berlin Heidelberg: Springer.

van der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering, 13*(2), 22-30.

Virah-Sawmy, M., Willis, K. J., & Gillson, L. (2009). Threshold response of Madagascar's littoral forest to sea-level rise. *Global Ecology and Biogeography, 18*(1), 98-110.

Appendix

Appendix A: West coast inundation height benchmarks.

Tile Size (dd)	Number of workers	Number of tiles	Number of tiles computed	Cells changed	Running time (s)
0.5	1	804	2558	335446065	4870
1	1	218	648	338422580	4742
2	1	68	184	379099967	4809
0.5	2	804	804	206005204	1354
1	2	218	218	249750002	1557
2	2	68	68	338763611	1946
0.5	4	804	2594	336386812	1242
1	4	218	659	338406323	1233
2	4	68	193	386091306	1324
0.5	8	804	2591	336263039	830
1	8	218	638	331456587	806
2	8	68	189	385300333	931
0.5	12	804	2569	329762652	804
1	12	218	657	337004449	832
2	12	68	197	386090007	888

Appendix B: NGDC experiment running time in seconds.

Number of Workers	Half Degree	One Degree	Two Degree
1	13499	13077	13575
2	6828	6675	6944
4	3461	3402	3509
8	2290	2324	2474
12	2280	2259	2397
31	861	905	964
63	556	1573	626
127	1486	1455	510