Drainage computations on digital elevation models

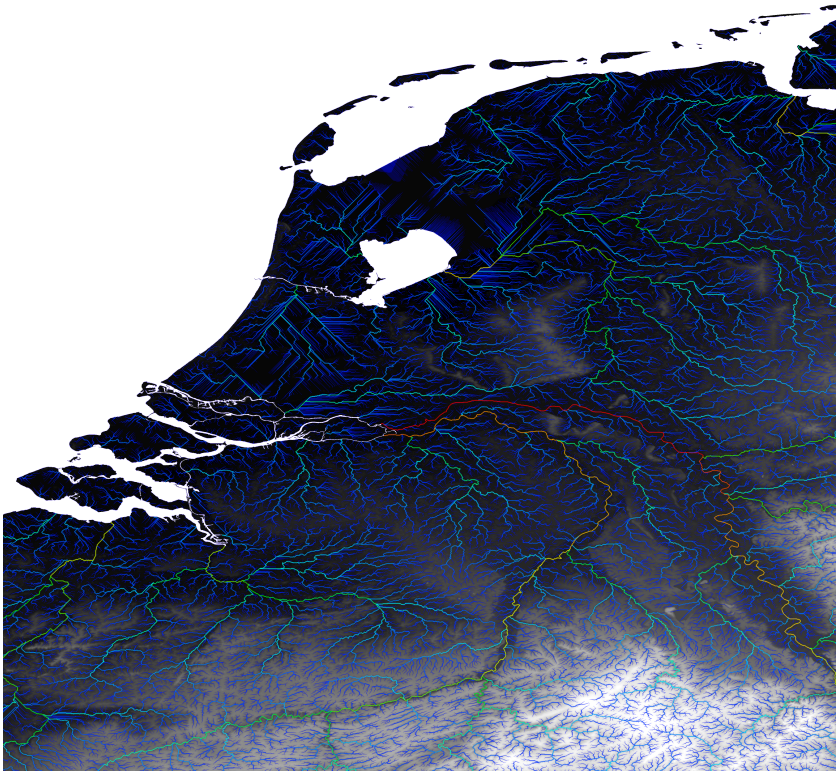
Janssen, J.H.M.

Award date:
2008

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Drainage computations on Digital Elevation Models

J.H.M. Janssen



Supervisors:
Herman Haverkort, Mark de Berg, Johan Lukkien

January 7, 2008

Abstract

In geographic information systems terrain models are stored using elevation samples. These can be used to compute where water flows, where flooding will occur, how the terrain can be decomposed into watersheds. As more detailed terrain data becomes more available these files become so large that they do not fit into the main memory anymore, resulting in poor performance for existing algorithms. Either because the operating is busy swapping pages in and out the memory constantly or the access pattern the algorithm has differs greatly from the order in which the data is stored on disk. In order to solve these problems in an acceptable amount of time I/O efficient algorithms are needed.

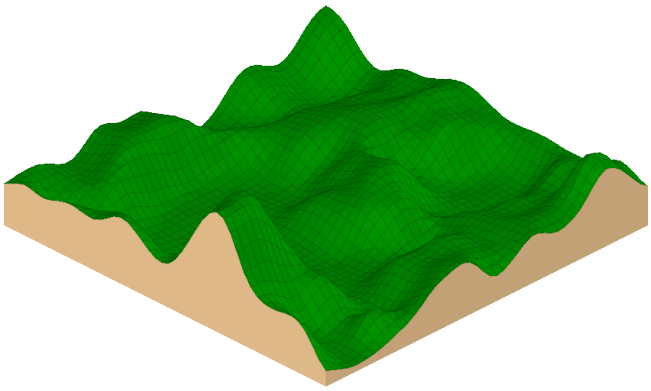
In this paper we will present an adaptation of a naive algorithm, a general cache-aware and a cache-oblivious algorithm for the flooding and flow-accumulation problems. These algorithms are designed to work on massive grid terrains. All algorithms have been implemented and were tested on a variety of real data files. Our fastest algorithm is able to flood a terrain of over 3.5 billion cells—over 13 gigabytes—in under 160 minutes. Flow accumulation for the same terrain is computed in less than 40 minutes.

Contents

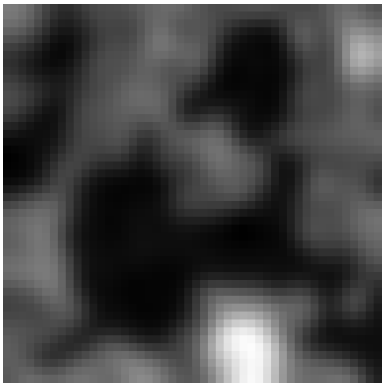
Chapter 1

Introduction

Geographical Information Systems (GIS) are information systems that work on or with terrain data, stored in files called Digital Elevation Models (DEM). These terrains are most commonly represented by a grid of elevation samples. Figure 1.1 shows a 3D terrain with a sampling grid superimposed on it and its corresponding height map. In the data file these height samples are stored as numerical values. These DEM files can be used to compute where water flows, where lakes will be formed, how the terrain can be decomposed into watersheds. Two of these problems—flooding and flow accumulation—will be discussed in this paper.



(a) 3D terrain



(b) 2D height map

Figure 1.1: Digital Elevation Model of a terrain

A grid is the most common way to store a DEM because of its simplicity, which makes it easy to store and process. Other ways to store a DEM are by using a *triangulated irregular network* (TIN) or *contour lines*[5][9].

The area outside the grid is called the ocean. This is defined in order to model the possibility of the water flowing off the terrain. This ocean has a height of negative infinity—meaning that it holds an infinite amount of water and will never overflow—and is connected to all cells on the border of the grid. Grid DEM files (including the ones we have used for testing) may contain cells that don't have a height value. These are called no-data cells. Usually these no-data cells represent large bodies of water on the terrain like seas, lakes or wide rivers.

In the flooding problem the height of the water level for each cell in the grid is computed when it is assumed that an infinite amount of rain has fallen on the terrain, forming puddles and lakes

where the water couldn't flow down the ocean.

Flow accumulation is about calculating how much water passes through every cell of the terrain when it flows down to the ocean, after each cell has initially received one unit of water.

Due to advances in satellite and other remote-sensing technology massive amounts terrain data for used in these GIS systems are readily available. In February of 2000 a NASA mission named SRTM[1] collected 30 meter resolution data (9 terabytes in size) for about 80% of the Earth's landmass. Data from this mission at 90 meter resolution are freely available to the public on the internet, as well as 30 meter data for the US.

The huge amounts of data makes that these problems need to be solved in an I/O[2] efficient manner, in order to guarantee sensible running times because hard drives are much slower than internal memory. This is because it takes time (in the order of ms) to position the head on the correct part of the disk. Reading consecutive blocks on the disk is faster, since this repositioning is not required. However, the maximum read speed of a hard drive is also only a fraction of the internal memory. For I/O efficient algorithms the number of I/O operations (reading/writing from disk) is kept to a minimum.

For each of the two problems three algorithms will be presented.

The first variant is an adaptation of a naive algorithm, to make it I/O efficient. A naive algorithm is usually designed to run in internal memory, and therefore does not take disk operations into account once the input data does not fit into memory anymore.

The second is a regular—cache-aware—I/O efficient algorithm that is based on the principle of dividing the input grid in smaller partitions. A cache-aware algorithm is designed to fully use the internal memory in the computer. The size of this memory must therefore be known before the algorithm is run.

The third variant for each of the problems is a cache-oblivious version of the cache-aware algorithm. Cache-oblivious means that the algorithm is not allowed to know the size of the memory or the size of the blocks on disk.

These algorithms are analysed on their I/O behaviour and computational complexity. All of the algorithms have been implemented in C++ and were run on real terrain data, in order to compare the running times and see which is the fastest.

Existing research into this subject (using I/O efficient algorithms) starts by creating a graph from the input grid, instead of taking advantage of the properties of a grid. TerraFlow[8] and TerraStream[2] are both systems that have—among others—implemented solutions for these two problems. Flooding in both systems is implemented by finding local minima and their corresponding watersheds (sinks) and overflow points to other sinks. From these a graph is created which is then flooded.

[1] Shuttle Radar Topography Mission—http://www2.jpl.nasa.gov/srtm/p_status.htm
[2] Input/Output

1.1 Terminology

The algorithms work on a grid of size $N = W \times H$, where W and H are respectively the width and height of the input grid. Some of the algorithms assume that the grid resembles a square, meaning that $W = \Theta(H)$. This is called the *square-input assumption*.

The standard two-level I/O model is used in this paper[1]. This means that there is an internal memory of size M, and the files on disk have a block size of B. Blocks are always transferred from disk to memory (and vice-versa) in their entirety, meaning that the I/O complexity is calculated as the number of block operations.

There are two special I/O complexity cases. The first one is defined as $\text{Scan}(N) \equiv \Theta\left(\frac{N}{B}\right)$, which is the amount of block operations needed to sequentially read a file of size N. The second is $\text{Sort}(N) \equiv \Theta\left(\frac{N}{B} \log_{\frac{M}{B}}\left(\frac{N}{B}\right)\right)$ representing the number of I/O operations needed to sort such a file.

Some algorithms use the so-called *tall-cache assumption*—$M = \Omega\left(B^2\right)$—stating that the cache (memory) is at least as high (number of blocks that it can store at the same time) as that it is wide (size of block).

Chapter 2

Flooding

Suppose an infinite amount of rain will fall onto a terrain. Water that cannot flow of the terrain directly will collect and form puddles and lakes. At some point enough water will be collected into a lake to fill it completely and it will overflow at a spill point. From this point on all water that rains into the lake directly or flows into the lake from other cells will flow out of this spill point. Eventually all lakes will be filled an all water that rains onto the terrain after this time is able to flow off the terrain into the ocean. At this point the terrain is flooded.

In the flooding problem we want to compute the flooded height for each cell, which is equal to the height of the lowest path from that cell to the ocean. The height of a path is defined as the height of the highest cell on that path. We can turn this around, and define the flooding problem as calculating the height of the lowest path from the ocean to each cell in the grid. We call this the SSLP (Single Source Lowest Path) problem, expressing the similarity to the SSSP (Single Source Shortest Path) problem. The only difference between these two problems is that in SSLP we take the maximum weight of all the edges in a path, whereas SSLP takes the sum of all edges on a path. The weight of an edge is the height of the lowest path between the two cells it connects. In the case an edge is between two adjacent cells, this is equal to the maximum of the heights of these two cells.

The input of the flooding problem is a grid, stored in a file referred to as *height* in the pseudo code of the algorithms, used for both the input and output. This grid forms a grid-graph, with (undirected) edges between each cell and all of its neighbours, meaning that water can only flow from a cell to one of its neighbouring cells. The neighbours of a cell c are the (up to) eight cells surrounding c.

No-data cells ignored by the implementation of these flooding algorithms. This means that no-data cells for which a path exists to the ocean that only consists of no-data cells will remain no-data after the terrain has been flooded. If such a path does not exists they will be flooded like regular cells.

2.1 Naive approach

One of the most well-known algorithms for the shortest path problem is Dijkstra's algorithm, and adapting it to compute the lowest path instead of the shortest path is trivial. In this algorithm the full grid is explored starting from the ocean, which is connected to all cells on the border of the grid. At each iteration the—at that point—lowest path to a cell is expanded. If we look at this process at a large point, it can be described as raising the water level outside the terrain,

where the flooded height of a cell is the height of the water level when the water first reaches this cell.

Essentially the terrain is flooded from the outside inwards giving an I/O access pattern that depends entirely on the terrain and that very likely jumps around "randomly" - resulting in a very poor I/O performance. In order to get a better performing algorithm, the idea is to substantially decrease the number of vertices (cells) in the terrain. This is done by first creating a watershed graph from the grid graph. A watershed is defined as a section of the terrain that flows into to the same sink (a cell that has no lower neighbouring cells). These watersheds are the vertices in the watershed graph. Edges are created between adjacent watersheds. The weight (height) of an edge in the watershed graph is defined as the minimum height of all edges between the watersheds in the grid graph.

In the first phase of the algorithm each cell in the grid is assigned to a watershed—stored in a file called *watershed*—and the watershed graph is created. The second phase floods the watershed graph, calculating the minimum height of all the cells each of the watersheds. Finally each cell is raised if it is lower than the minimum height of the watershed it belongs to. Giving the following outline of the algorithm:

1: **procedure** NaiveFlooding(grid G)
2: $W \leftarrow$ CreateWatersheds(G)
3: FloodWatersheds(W)
4: FloodGrid(G, W)
5: **end procedure**

2.1.1 Creating watersheds

In order to find the watershed to which a cell belongs, we follow the flow of water originating from this cell, until we either find a cell that already has a watershed assigned, or that we end up in a sink. The path followed is stored, so the watershed label can be assigned to each cell once the correct label has been determined. In the case that a cell is found that already has a watershed label assigned, this label is used to label each cell on the flow path. In the case that a sink has been found, we create a new watershed, and label each of the cells on the flow path with this watershed.

There are a few special cases that need to be handled differently. First there is the ocean, which can be considered as one big sink, so each cell flowing directly into the ocean, will be assigned to the ocean watershed. Secondly, it is possible that a cell has no lower neighbours, but some that are on the same height. In this case we look at the neighbours that have the same height. If one of them has a watershed assigned, the original cell will be assigned to that watershed. If none of the neighbours at the same height has a watershed assigned, a new watershed is created. This is done to avoid having to search a big area that has the same height for a spill point and prevent loops while doing so.

When a cell is labelled, also its neighbouring cells are examined to see if they belong to a different watershed. If this is the case, an edge between those watersheds is created. When such an edge already exists, only the lowest edge is kept.

In line 11 of the CreateWatersheds procedure a function FindLowestNeighbour is called. This function returns the lowest neighbour of a given cell, and if multiple cells have the same height it checks if one of them already has a watershed assigned.

Pseudo code

```
1: procedure CreateWatersheds(grid G)
2:     W ← empty watershed graph
3:     ocean ← CreateNewWatershed(W)
4:     for each cell c in G do
5:         if ¬watershed[c] then
               ▷ Follow flow path, until a watershed, sink or plateau is found. The cells in the
   flow path are stored in p.
6:             p ← ∅
7:             d ← c
8:             repeat
9:                 p ← p ∪ {d}                                  ▷ Add d to the current path
10:                e ← d
11:                d ← FindLowestNeighbour(e)
12:                if height[d] > height[e] ∨ (height[d] = height[e] ∧ ¬watershed[d]) then
13:                    watershed[e] ← CreateNewWatershed(W)
14:                end if
15:            until watershed[e]
               ▷ Label all cells on the flow path with the correct watershed label. Neighbouring
   cells are examined, and if they are labeled with a different watershed, an edge is created
   between the two watersheds.
16:            for each cell d in p do
17:                watershed[d] ← watershed[e]
18:                for each neighbour f of d do
19:                    if watershed[f] ∧ watershed[f] ≠ watershed[e] then
20:                        CreateEdge(W, watershed[e], watershed[f], max(height[f], height[d]))
21:                    end if
22:                end for
23:            end for
24:        end if
25:    end for
26:    return W
27: end procedure
```

I/O analysis

The algorithm iterates through all cells, which can happen in any order, including the file order, so this takes Scan(N) I/O operations. For each cell the flow path is followed, which does only requires extra I/O operations if we move outside the currently cached area. In order to follow any path efficiently, we need a file ordering that preserves spatially locality, like the z-order

ordering.

We conceptually divide the grid into partitions of size $\frac{\sqrt{M}}{3} \times \frac{\sqrt{M}}{3}$. We assume that when we iterate through the grid, the partition we are currently in, together with the surrounding partitions are cached into memory — see figure 2.1. This way an area of $\sqrt{M} \times \sqrt{M}$ cells is cached, and a minimum of $\Theta\left(\sqrt{M}\right)$ steps can be taken before extra I/O operations are needed.

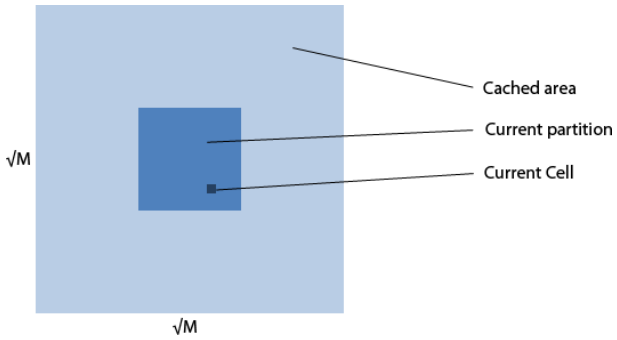


Figure 2.1: Cached area of naive algorithm

Outside of the cached area, an extra I/O operation is needed for each block on disk the path passes through. If we look at the disk blocks, and if there are—on average—only a constant number of these paths per block, the I/O complexity is unaffected. Formally, the I/O performance of this part of the algorithm is Scan(N) if the following holds:

$$\left(\sum_{\text{all blocks } B} \text{number of paths that are longer than } \sqrt{M} \text{ in } B \right) = \mathcal{O}\left(\frac{N}{B}\right)$$

This assumes that the watershed graph fits in memory. If it doesn't, extra I/O operations are needed to access the edges of the graph, in which case the I/O performance becomes Scan(N) + $SortE$, where E is the number of edges in the watershed graph.

Running time analysis

Cells are examined by this algorithm when it's their turn in the main iteration and when one of its neighbours examines it for determining the next cell in the flow path, or when it's watershed label is checked to find adjacent watersheds by one of its neighbours. Since each cell has a constant number of neighbours (at most eight), it is only examined a constant number of times. Efficiently updating the edges requires the use of a tree-like data structure, resulting in a running time of $\mathcal{O}\left(E \log(E) + N\right)$ where E is the number of edges in the watershed graph.

2.1.2 Flooding

After the first phase all watersheds and the edges between them have been defined. These are then flooded by using SSLP to find the lowest paths from the ocean watershed to all other watersheds. After this is completed, we have computed the minimum height for each of the watersheds. This information is then used to flood the entire grid, by setting the height of each cell to the maximum of the height of the cell itself and the height of the watershed it belongs to. The height of a vertex (watershed) v and edge e are represented by $height_W[v]$ and $height_E[e]$ respectively.

Pseudo code

Require: ($\forall$ vertices $v \in V :: completed[v] = \bot \wedge height_W[v] = -\infty$)

```
1: procedure FloodWatersheds(watershed graph W = (V, E))
2:     completed[ocean] = ⊤
3:     Sort(E)                                   ▷ Sort edges in ascending order
4:     for each edge e in E do
5:         if completed[source[e]] ∧ ¬completed[target[e]] then
6:             FloodEdge(e)
7:         end if
8:     end for
9: end procedure
```

```
1: procedure FloodEdge(edge e)
2:     s ← source[e]
3:     t ← target[e]
4:     height_W[t] ← max(height_W[t], height_E[e])
5:     completed[t] ← ⊤
6:     for each edge f in edges[t] do
7:         if ¬completed[target[f]] then
8:             if height_E[f] ≤ height_E[e] then
9:                 height_E[f] ← height_E[e]
10:                FloodEdge(f)
11:            end if
12:        end if
13:    end for
14: end procedure
```

```
1: procedure FloodGrid(grid G, watershed graph W = (V, E))
2:     for each cell c do
3:         height[c] ← max(height[c], height_W[watershed[c]])
4:     end for
5: end procedure
```

I/O analysis

If we assume that the watershed graph fits in memory, the only I/O operations needed for the flooding phase are reading each cell once, and writing it back at most once. Since the cells can be processed in any order (including the order in which the cells are stored on disk) the I/O complexity of this phase is Scan(N).

If the watershed graph does not fit in memory, extra I/O operations are needed to flood the watershed graph which increases it to Scan(N) + E I/O operations. Accessing the heights of the watersheds can also require extra I/O operations. However, since the grid is scanned in the same order as in the first phase of the algorithm, the watersheds are found in the same order, requiring Scan(E) I/O operations. For cells belonging to watersheds that were accessed before, we can assume that their height will remain cached in memory, since an spatially local ordering is used. For really long paths jumps in the z-order can introduce extra I/O operations, however we already assumed that the number of long paths was at most a constant per block, thus not affecting the I/O complexity here either.

The I/O complexity this second step is Scan(N) + E, if the watershed graph does not fit into memory.

Running time analysis

Flooding the graph requires all edges to be sorted, or the usage of a priority queue. In both cases the running time is $\mathcal{O}\big(E \log(E)\big)$, where E is the number of edges in the watershed graph.

Each of the cells can be flooded in constant time, therefore flooding all N cells requires $\mathcal{O}(N)$ time. This results in a total running time of $\mathcal{O}\big(E \log(E) + N\big)$.

2.1.3 Analysis

In the analysis for the sections we made a distinction whether the watershed graph was small enough to fit in memory or not. If it does fit in memory, we get the following complexities:

I/O complexity:	Scan(N)
Running time:	$\mathcal{O}\big(E \log(E) + N\big)$

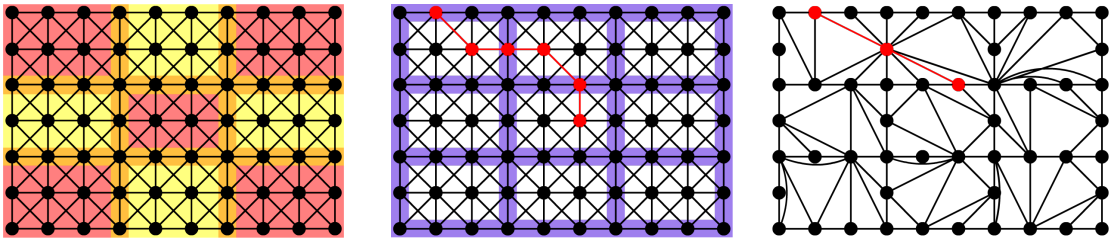
If the watershed graph does not fit in memory, and is stored on disk in its entirety, the bounds become the following:

I/O complexity:	Scan(N) + E
Running time:	$\mathcal{O}\big(E \log(E) + N\big)$

2.2 Cache-aware approach

Our cache-aware approach for flooding grids is based on the I/O efficient SSSP (Single Source Shortest Path) algorithm for grids, as described in chapter 5 of [4]. The approach used in this algorithm is partitioning the grid in such a way that each partition can be processed in memory.

The full grid graph is too big to flood it all at once. The principle of this algorithm is to first create a reduced graph (called G_R) from the full graph, containing significantly less vertices (cells) and edges. The edges for G_R are created in such a way that when G_R is flooded—which is the second phase of the algorithm—the flooded heights for the cells in G_R are the same as the full graph would have been flooded at once.



(a) Partitions with size 4×3 (b) G_R vertices (c) G_R graph after edge creation

Figure 2.2: Example grid graph

Water can only flow to cells that are directly adjacent. This means that if we take a rectangular section of the grid, the lowest path for all cells in this rectangle must pass at least one of the cells on the border. We divide the grid into partitions of $w \times h$ cells (see figure 2.2(a), each overlapping the adjacent partitions by one row or column.

If we consider the lowest path for any cell, the first section of the path is from the cell itself to the border of the partition containing this cell. The following sections of the path go from the previous endpoint—which is on a partition border—to another cell on the border of an adjacent (or the same) partition, only passing cells in that partition. An example of this is shown in figure 2.2(b). Note that nothing restricts the shortest path to go through the same partition multiple times.

We now define the vertices of the reduced graph G_R as the union of all the cells on the partition borders, an example of which is also shown in figure 2.2(b). In order to use this graph to compute lowest paths, edges need to be added that the height of the lowest path between each pair of cells on the partition border is equal in the original grid graph and in G_R. See figure 2.2(c) for an example G_R. In this figure the lowest path between the cells in G_R for the example path has been coloured. The edges for G_R are computed in the first phase of the algorithm.

In the second phase of the algorithm graph G_R is flooded, computing the heights of the lowest path from the ocean to each of the cells in G_R. After this is done, the final height is known for all border cells in each partition and the flooded height for the internal cells can be computed. This is done in the third phase of the algorithm. This gives the following outline for the cache-aware flooding algorithm:

```
1: procedure CacheAwareFlooding(grid G)
2:     G_R ← empty (new) separator graph for G
3:     CreateEdges(G)
4:     FloodGraph(G_R)
5:     FloodGrid(G, G_R)
6: end procedure
```

2.2.1 Creating edges

Algorithm description

We need to find the lowest paths between each of the cells on the border of partition P. We do this by flooding the P from the border inwards, tagging each cell in the partition with the cell on the boundary to which it has the lowest path, this tag is called the watershed label. This way we compute the watershed label for each cell in the interior of the partition. The lowest edge between each pair of adjacent watersheds is added to G_R.

We flood partition is by gradually raising the water level, and determine which cells will be filled from which point on the boundary. Which means add a cell to a watershed if and only if there is a path from that cell to the cell on the border that watershed belongs to, having a height lower or equal to the maximum of the current water level and the height of the cell itself.

We define $height[c]$ as the height of cell c, and $watershed[c]$ as the watershed of a cell c. In order to simulate the raising of the water level, we sort all cells from the lowest to the highest and process them in that order.

When we process cell c, the simulated water level $l = height[c]$. This cell can either be on the border or not.

If c is not on the border, and also has no watershed label yet, cell c cannot be flooded at the current water height (because if there was a path to c with height $height[c]$, cell c would have its watershed label set to the cell on the border this path is connected to. We therefore ignore this cell for now, and continue with the next cell.

If cell c is on the border, there are two possibilities, either cell c does not yet belong to a watershed, which means we have found a new spill point so we create a new watershed for cell c. This can be done by setting $watershed[c]$ to c. If cell c does belong to a watershed, we create an (undirected) edge from c to $watershed[c]$ with height $height[c]$. By doing this we ensure that each cell on the border of the current partition will have at least one edge.

If c is on the border, or it has a watershed label assigned, each of the neighbours d of c are examined:

If $watershed[d]$ has not been set, we add d to the watershed of c by setting $watershed[d]$ to $watershed[c]$. We then compare $height[d]$ to $height[c]$ and set $height[d]$ to l if $height[d] \leq height[c]$. The neighbours of cell d are then recursively processed.

If $watershed[d] \neq watershed[c]$, the watersheds of c and d are adjacent, so we add an (undirected)

edge between *watershed*[c] and *watershed*[d] with height max(*height*[d], l). If an edge already exists between *watershed*[c] and *watershed*[d], we keep only the edge with the lowest height.

We repeat this until all cells have been processed, in which case all cells in P have been assigned a watershed, and all adjacent watersheds have been found. Because edges are only added between adjacent watersheds, and watersheds are connected (unbroken) areas, the result is generally a planar graph. Under some circumstances it is possible for four watersheds to touch each other, introducing an edge that crosses an other. Though the highest edge can always be removed in this case (water flows will never cross)—the number of edges is still limited even with these crossing edges.

Pseudo code

The pseudo code for the algorithm that creates the edges for G_R is given below. For this pseudo code we assume that there exists a function EdgeExists(c, d) that checks if G_R contains an edge between the cells c and d, a function CreateEdge(c, d, j) that adds a new edge to G_R with height j and that there is a function LowerEdge(c, d, j) that updates the height of the edge between c and d if j is lower than the current height.

```
1:  procedure CreateEdges(graph G)
2:      for each partition P in G do
3:          Sort all cells in P with ascending height, and store them in Q.
4:          for each cell c in Q do
5:              if c on border of P ∨ watershed[c] then
6:                  ProcessCell(P, c)
7:              end if
8:          end for
9:      end for
10: end procedure
```

```
1:  procedure ProcessCell(partition P, cell c)
2:      if c on border of P then
3:          if ¬watershed[c] then
4:              watershed[c] ← c
5:          else
6:              CreateEdge(c, watershed[c], height[c])
7:          end if
8:      end if
9:      for each neighbour d of c do
10:         if ¬watershed[d] then
11:             watershed[d] ← watershed[c]
12:             if height[d] ≤ height[c] then
13:                 height[d] ← height[c]
14:                 ProcessCell(P, d)
15:             end if
16:         else if watershed[d] ≠ watershed[c] then
17:             if EdgeExists(watershed[c], watershed[d]) then
```

```
18:                 LowerEdge(watershed[c], watershed[d], max(height[c], height[d]))
19:             else
20:                 CreateEdge(watershed[c], watershed[d], max(height[c], height[d]))
21:             end if
22:         end if
23:     end for
24: end procedure
```

I/O analysis

Partitions are fully read from disk, one at a time. Neighbouring partitions share a boundary row or column, so the blocks these cells belong to are read once for each partition they belong to. Given the tall-cache assumption ($M = \Omega(B^2)$), these boundary blocks are read at most four times (in case the boundary cell is on the corner). This means that each block is read $\mathcal{O}(1)$ times, from which follows that the I/O volume is Scan(N). After processing a partition the edges for G_R are written to disk. Since G_R is smaller than the original graph, the amount of I/O operations required for this is also smaller, and does not affect the I/O performance.

Running time analysis

Each cell in the grid is processed once. When we process a cell, we take examine its neighbours, which are at most eight cells. However, all cells are sorted first, which takes $\mathcal{O}(w \cdot h \log(w \cdot h))$ time, per partition. Considering that each partition must fit onto memory, we have that $w \cdot h = \Theta(M)$. The total number of partitions is $\mathcal{O}(\frac{N}{M})$, giving a total running time of $\mathcal{O}(\frac{N}{M} \cdot M \log(M)) = \mathcal{O}(N \log(M))$.

2.2.2 Flooding G_R

Algorithm description

For flooding G_R we use a similar approach as in the first step, which is flooding inwards from the border, by simulating the raising of the water level. The main difference is that a graph is used instead of a grid.

First all cells are sorted on height, and all cells on the border of G_R are marked completed.

Define $height[e]$ as the height of edge e, $source[e]$ as one of the endpoints (cell, vertex) of e and $target[e]$ as the other endpoint.

The lowest edge e is removed from the queue, and the cell $t = target[e]$ is marked as completed (if t was already marked as completed, the algorithm continues with the next edge in the queue). The height of t is set to $height[e]$, if $height[e] > height[t]$. Finally, all outgoing edges f from t where $target[f]$ hasn't been completed yet are processed, and if $height[f] \leq height[e]$, we set $height[f]$ to $height[e]$ and recurse on edge f.

This process is repeated until each edge in the list has been processed.

Pseudo code

For the pseudo code for the flooding of G_R we assume that there is an adjacency list for each cell c containing all edges from c to other cells, which can be accessed by $edges[c]$. The state (completed or not completed) is represented by $completed[c]$.

```
1:  procedure FloodGraph(separator graph G_R)
2:      Sort all edges in G_R with ascending height, and store them in Q.
3:      for each cell c on border of G_R do
4:          completed[c] ← ⊤
5:      end for
6:      for each edge e in Q do
7:          if completed[source[e]] ∧ ¬completed[target[e]] then
8:              FloodEdge(P, e)
9:          end if
10:     end for
11: end procedure
```

```
1:  procedure FloodEdge(edge e)
2:      t ← target[e]
3:      completed[t] ← ⊤
4:      height[t] ← max(height[t], height[e])
5:      for each edge f in edges[t] do
6:          if ¬completed[target[f]] ∧ height[f] ≤ height[e] then
7:              FloodEdge(f)
8:          end if
9:      end for
10: end procedure
```

I/O analysis

The number of I/O that is needed for this step greatly depends on how the graph G_R is stored on disk. The most I/O efficient way to do this is by storing the cells and edges separate from the grid, instead of reading the part of the grid that is occupied by G_R.

The cells are read from disk before the algorithm starts, and are written back after the algorithm has finished. The size of graph G_R depends on the size and amount of partitions in the grid, so it contains $|G_R| \leq \left(\frac{H}{h} + 1\right) W + \left(\frac{W}{w} + 1\right) H$ vertices (cells), and at most three times as many edges (because G_R is planar).

The partition size is chosen such that it fits in the memory, which means that $w \cdot h = \Theta(M)$. If we assume that the partitions are square-like ($w = \Theta(h)$) this gives us, together with the tall-cache assumption, that $w = \Omega(B)$ and $h = \Omega(B)$.

$$
\begin{aligned}
|G_R| &\leq \left(\frac{H}{h}+1\right) W + \left(\frac{W}{w}+1\right) H \\
|G_R| &= \mathcal{O}\left(\frac{H}{h}W + \frac{W}{w}H\right) \\
&= \mathcal{O}\left(\frac{H}{B}W + \frac{W}{B}H\right) \\
&= \mathcal{O}\left(\frac{W \cdot H}{B}\right) \\
&= \mathcal{O}\left(\frac{N}{B}\right)
\end{aligned}
$$

For each vertex in G_R a constant number of I/O operations are needed to read and write the heights of the cells. This dominates the I/O performance of this part of the algorithm, given that G_R fits in memory, in this case the complexity is $\mathcal{O}(|G_R|) = \text{Scan}(N)$.

If G_R doesn't fit in memory, an external memory sorting algorithm is needed to flood it, giving an I/O bound of $\mathcal{O}(|G_R|) + \text{Sort}(|G_R|) = \text{Scan}(N) + \text{Sort}\left(\frac{N}{B}\right)$.

Running time analysis

Processing an edge takes constant time, but because they need to be sorted first, the complexity of this part is $\mathcal{O}\big(|G_R| \log(|G_R|)\big)$. $\mathcal{O}(1)$.

2.2.3 Flooding each partition

Algorithm description

The algorithm for flooding the grid is the same as the one given for creating the edges, except we are no longer interested in the watersheds, it suffices to mark the cells as completed once the height of the cell has been updated. Also the height of the cells on the border need to be updated after G_R has been flooded. Finally partitions need to be written back to disk after they have been flooded. This results in the following pseudo code:

Pseudo code

```
1: procedure FloodGrid(grid G, separator graph G_R)
2:     for each partition P in G do
3:         Sort all cells in P with ascending height, and store them in Q.
4:         for each cell c on border of P do
5:             height[c] ← height of c in G_R
6:             completed[c] ← ⊤
```

```
        end for

        for each cell c in Q do
            if c on border of P ∨ watershed[c] then
                FloodCell(P, c)
            end if
        end for
        Write P back to disk
    end for
end procedure
```

```
procedure FloodCell(partition P, cell c)
    for each neighbour d of c do
        if ¬completed[d] then
            completed[d] ← ⊤
            if height[d] ≤ height[c] then
                height[d] ← height[c]
                FloodCell(P, d)
            end if
        end if
    end for
end procedure
```

I/O analysis

The whole grid is read from disk, and written back after processing. This takes Scan(N) I/O operations. Also, graph G_R needs to be accessed to update the heights. Because the size (width and height) of G_R is the same as the full grid, but contains less cells, the time to access these cells is at most Scan(N). The I/O complexity of this algorithm therefore remains Scan(N).

Running time analysis

The running time of the algorithm is dominated by sorting the cells. This means that the running time is $\mathcal{O}\big(w \cdot h \log(w \cdot h)\big)$. Considering that each partition must fit onto memory, we have that $w \cdot h = \Theta(M)$. The total number of partitions is $\mathcal{O}\big(\frac{N}{M}\big)$, giving a total running time of $\mathcal{O}\big(\frac{N}{M} \cdot M \log(M)\big) = \mathcal{O}\big(N \log(M)\big)$.

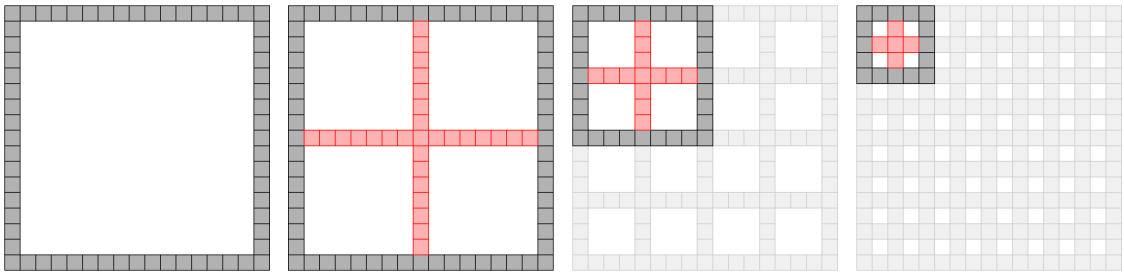
2.2.4 Analysis

Combining the results of the analyses from the individual sections gives the following bounds for the cache-aware flooding algorithm:

I/O complexity:	$\text{Scan}(N) + \text{Sort}\left(\frac{N}{B}\right)$
Running time:	$\mathcal{O}(N \log M)$

2.3 Cache-oblivious approach

The main problem in converting the cache-aware algorithm to a cache-oblivious variant lies in the fact that we cannot divide the grid into partitions in the same way, since we do not know the size of the memory. Instead of having a single G_R, with fixed partition sizes, we define a recursive version of the separator graph.



(a) Level 0 (b) Level 1 (c) Level 2 (d) Level 3

Figure 2.3: Recursive separator graph G_R at multiple levels—only cells, no edges

At the highest level (figure 2.3(a)) we have a graph only consisting of the boundary cells of the full grid. In each recursive step the current partition is divided into four (2×2) smaller partitions, with a column or row shared between the adjacent partitions. This is repeated until a partition has a size of 2×2 cells. The edges in the boundary graphs are—just as in the cache-aware algorithm—the lowest paths between the cells on the border of partitions. Flooding using this approach is done in two phases.

The first phase uses a bottom-up approach. Edges are created at the lowest level, where partitions have a size of 2×2. At this level it is trivial to determine the lowest paths between the cells of the partition border. Adjacent partitions are merged on the way upwards in the recursion, in each step removing the internal cells. These are cells that are on the border of a partition in a lower level, but not anymore on the current level. In figure 2.3 these are the red coloured cells. This removing is done by merging the internal cells (and its edges) with the cell on the border to which it has the lowest path.

After the first phase has been completed, all lowest edges from a border cell to the internal cells have been computed, for each level in the separator graph. The second phase uses these lowest paths to calculate the flooded height for the internal cells in a top-down direction, moving from the highest level to the lower ones. This is done by raising the height of internal cells if the height of the lowest path to the border is higher than the cell itself.

In contrast to the cache-aware algorithm there is no separate step needed for flooding the final

grid, since the separator graphs already go down to the cell level. Flooding the separator graphs is therefore the same as flooding the grid. The following pseudo-code gives the outline for the algorithm:

```
1: procedure CacheObliviousFlooding(grid G)
2:     CreateEdges(G)
3:     FloodEdges(G)
4: end procedure
```

2.3.1 CreateEdges

In the first phase we merge adjacent partitions. For each partition we consider only the cells on the border, as well as the edges between these cells, corresponding to the lowest paths between them. This information can be calculated easily for partitions of size 2×2 cells, by simply adding an edge between each two cells. Note that we can omit the diagonal edges unless the diagonal is lower than all four edges around the border. This can occur only if both of the cells on a diagonal edge are lower than each of the two remaining cells. In this case a single diagonal edge is added. This means that each partition of size 2×2 must be planar. Therefore if we look at graph consisting of all 2×2 sizes partitions, it is planar as well.

In order to remove the internal cells when merging adjacent partitions, we need to merge them with one of the cells on the border of the new partition. Therefore we have to find the lowest path from each of the internal cells to one of the cells on the border. The algorithm we use for this is the same as the algorithm used for flooding the G_R graph in section 2.2.2, except that instead of flooding the internal cells, they are added to the watershed of the cell on the border they were flooded by.

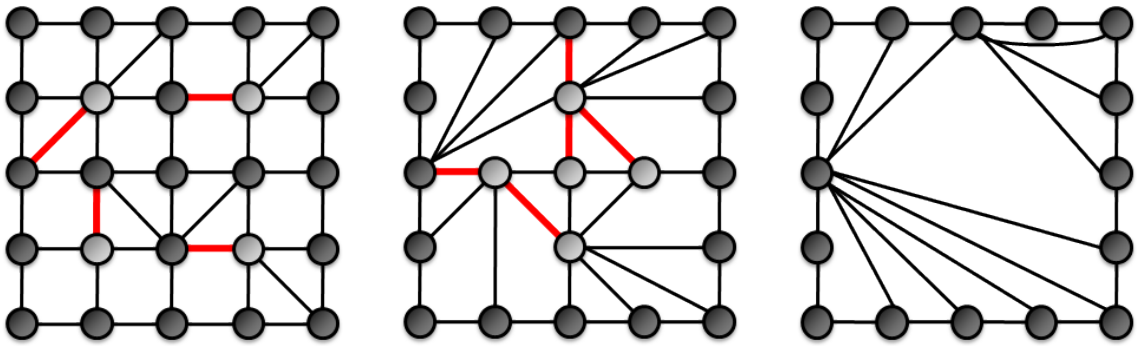
After this, each internal cell c has been assigned to a watershed d corresponding to one of the cells on the border. Also the height of the lowest path between the c and d has been calculated. Both of these values are stored to disk for cell c, so they can be reused in the next step of the algorithm. Cell c is merged with cell d, meaning that all edges going from or to cell c are updated so that they now go to cell d. During this process, it is possible that multiple edges connect the same two cells, in this case only the lowest edge needs to be kept. Also it is possible that edges are introduced that connect cells to themselves, these edges also need to be removed.

Figure 2.4 shows an example of the merge process. The red (thick) lines indicate to which cell on the border (dark coloured cells) the internal (light coloured cells) are merged. Level 0 is shown fully in this image, but it is actually processed in four separate partitions in the recursion.

Since vertices are merged over existing edges (on a planar graph), this process keeps the graph planar. It is repeated at each level until the full grid is processed.

Pseudo code

We present the pseudo code for creating the edges between the cells on the border of a grid (or partition). It assumes that there is a function GetEdges(G) that returns all straight (horizon-



(a) Level 0 (b) Level 1 (c) Level 2

Figure 2.4: Example of cache-oblivious merge step

tal/vertical) edges together with the lowest diagonal edge of a grid G that has a size of two by two cells in sorted order. It also makes use of a function CreatePartitions(G) that partitions a grid G into several smaller grids, with an overlapping row and column between adjacent grids.

When combining the edges of each partition in line 8 of CreateEdges function, this must be done in such a way that the result remains sorted. Since both parts that are combined are sorted initially, this can be done in linear time.

```
1:  procedure CreateEdges(grid G)
2:      if width[G] = 2 ∧ height[G] = 2 then
3:          return GetEdges(G)                    ▷ Edges are sorted with ascending height
4:      else
5:          E ← ∅
6:          P ← CreatePartitions(G)
7:          for each partition P in P do
8:              E ← E ∪ CreateEdges(P)            ▷ Combine edges, keeping them in sorted order
9:              for each cell c on border of P do
10:                 watershed[c] ← c
11:             end for
12:         end for
13:         return MergeEdges(E)
14:     end if
15: end procedure
```

The function MergeEdges merges adjacent partitions by removing the internal cells. This is done by replacing all edges to internal cells with an edge to a cell on the border.

This function consists of two steps, the first step is finding for each internal cell i the cell j on the border to which i has the lowest path. This is done using the same algorithm used for flooding the G_R separator graph in the cache-aware approach, for which it uses the function ProcessEdge.

The second step is moving the edges from the internal cells to the cells on the border, for which

it also uses a function RemoveDuplicates that removes all edges that connect a cell to itself, and duplicate edges where the lowest one needs to be kept.

```
1: procedure MergeEdges(edges E)
2:     for each edge e in E do
3:         if watershed[source[e]] ∧ ¬watershed[target[e]] then ProcessEdge(e)
4:         end if
5:     end for
6:     for each edge e in E do
7:         source[e] ← watershed[source[e]]
8:         target[e] ← watershed[target[e]]
9:     end for
10:    return RemoveDuplicates(E)
11: end procedure
```

```
1: procedure ProcessEdge(edge e)
2:     s ← source[e]
3:     t ← target[e]
4:     height[t] ← max(height[t], height[e])
5:     watershed[t] ← watershed[s]
6:     for each edge f in edges[t] do
7:         if ¬watershed[target[f]] then
8:             if height[f] ≤ height[e] then
9:                 height[f] ← height[e]
10:                ProcessEdge(f)
11:            end if
12:        end if
13:    end for
14: end procedure
```

I/O analysis

The CreateEdges algorithm is a recursive algorithm that works on a planar graph. This means the size of the sub problems in each recursive step is $\mathcal{O}(V)$, where V is the number of vertices in the current sub problem. In each step the graph is divided into four partitions, each half the amount of vertices as the original sub problem, since the internal vertices need to be added to cover the full border of the sub problems.

We assume that the input grid resembles a square, and that the partitioning preserves this square-like shape. This means that the amount of vertices V on the border of a sub problem of size N' is $\mathcal{O}\left(\sqrt{N'}\right)$.

Once a sub problem is small enough to fit in memory it requires Scan(M) I/O operations to process the entire sub problem. In larger sub problems we can have $\mathcal{O}(1)$ I/O operation per vertex, resulting in $\mathcal{O}\left(\frac{V}{2^i}\right) = \mathcal{O}\left(\frac{\sqrt{N}}{2^i}\right)$ I/O operations for a sub problem at level i. This gives us the following I/O complexity:

$$
\begin{aligned}
& \left(\sum_{i=0}^{\log_4\left(\frac{N}{M}\right)} 4^i \cdot \frac{\sqrt{N}}{2^i} \right) && + \text{Scan}(N) \\
= \quad & \mathcal{O}\left(\sqrt{N} \cdot \left(\sum_{i=0}^{\log_4\left(\frac{N}{M}\right)} 2^i \right) \right) && + \text{Scan}(N) \\
= \quad & \mathcal{O}\left(\sqrt{N} \cdot \left(2^{\log_4\left(\frac{N}{M}\right)} \right) \right) && + \text{Scan}(N) \\
= \quad & \mathcal{O}\left(\sqrt{N} \cdot \sqrt{\frac{N}{M}} \right) && + \text{Scan}(N) \\
= \quad & \mathcal{O}\left(\frac{N}{\sqrt{M}} \right) && + \text{Scan}(N)
\end{aligned}
$$

Given the tall-cache assumption $M = \Omega\left(B^2\right)$ the total I/O complexity of this part of the algorithm is Scan(N).

Running time analysis

The running time of MergeEdges is linear in the number of edges. However, the RemoveDuplicates function needs to sort the edges in order to efficiently find duplicate edges, and then resort the remaining edges in the original order, which takes $\mathcal{O}\left(V \log(V)\right)$ time.

This gives us the following recurrence: $T(v) = 4 \cdot T\left(\frac{v}{2}\right) + \mathcal{O}\left(v \log(v)\right)$. Solving this with the Master Theorem gives us a running time of $\mathcal{O}\left(V^2\right) = \mathcal{O}(N)$.

To flood the grid we process it in the reverse order of the previous phase. Initially we start with the full grid as a single partition, filling in the cells on the borders of the sub-partitions, and then recursively process each sub-partition, until the partitions no longer have internal cells (this happens when the partition is two cells wide or high.)

The height of the cell that is flooded is set to the maximum of the height of that cell, and the height of the cell on the border of the partition that floods into it. These values have been computed in phase one, and are reused here.

Pseudo code

```
1: procedure FloodEdges(grid G)
2:     if width[G] > 2 ∧ height[G] > 2 then
3:         P ← CreatePartitions(G)
4:         for each cell c in P that is inside of G do
5:             height[c] ← max(height[c], height[watershed[c]])
6:         end for
7:         for each partition P in P do
8:             FloodEdges(P)
```

```
 9:         end for
10:     end if
11: end procedure
```

I/O analysis

This second part of the algorithm follows the same recurrence as the first step and therefore has the same I/O complexity of Scan(N).

Running time analysis

This second part of the algorithm follows the same recurrence as the first step and therefore has the same running time of $\mathcal{O}(N)$.

2.3.2 Analysis

The combined complexities of the cache-oblivious flooding algorithm are:

$$\begin{aligned} \text{I/O complexity:} &\quad \text{Scan}(N) \\ \text{Running time:} &\quad \mathcal{O}(N) \end{aligned}$$

2.4 Algorithm comparison

Three different algorithms for the flooding problem have been presented in this chapter. In the table 2.1 we give a short summary of the complexities of each of these algorithms.

Algorithm	I/O complexity	Running time
Naive (watershed graph in memory)	Scan(N)[1][2]	$\mathcal{O}\big(E \log(E) + N\big)$
Naive (watershed graph on disk)	Scan(N) + E[1][2]	$\mathcal{O}\big(E \log(E) + N\big)$
Cache-aware	Scan(N) + Sort$\big(\frac{N}{B}\big)$	$\mathcal{O}\big(N \log(M)\big)$
Cache-oblivious	Scan(N)	$\mathcal{O}(N)$

Table 2.1: Flooding algorithm complexities

Where E is the number of edges in the watershed graph in the naive algorithm.

[1] Using a file ordering that preserves spatial locality, like z-order
[2] For "normal" maps

Chapter 3

Flow accumulation

The second problem we will be looking at is the flow accumulation problem. We assume that one unit of rain falls on each cell of the terrain. This water will flow downwards over the terrain until it falls off the edge of the terrain. We are interested in the amount of water that passes each cell.

Flow accumulation is generally done after the terrain has been flooded, so that there exists a descending path from each cell to the border of the grid. Between the flooding and flow accumulation there is another step, named flow routing. In this step the flow target(s) are assigned to all cells on the terrain, most importantly for the cells on flat parts of the terrain that have no lower neighbours, so that there is a descending path to the ocean from each cell.

There are two possible flow models for flow routing and flow accumulation[6][3]. The first one uses single flow direction (SFD), where all of the water on each cell will flow to the lowest neighbouring cell. The other one has multiple flow directions (MFD), which means that the water flow for each cell is divided among all lower neighbouring cells. SFD is used for the algorithms presented in this chapter. This means that each cell can have zero to seven *source* neighbours that flow towards the current cell, and one *target* neighbour towards which the water of the current cell flows. Section 3.5 of this chapter shows how the algorithms can be adapted for MFD.

The flow routing problem is not part of this paper, however in order to run tests on the flow accumulation algorithms, the flow routing step must have been completed. Therefore a flow routing program has been run on the flooded input, creating a SFD output file containing the flow direction for each cell. This flow direction indicates which of the eight neighbouring cells is the flow target. No-data cells are considered part of the ocean and therefore have no flow direction.

The input file (called *target*) stores for each cell c the position of its target cell in the same file. For cells that do not have a target cell (because it is no-data or it flows directly into the ocean) a value of *false* ($\perp$) is used. The output of the algorithms is a file which is named *flow*, where for each cell the accumulated flow is stored. Neither of these files is dependent on a specific file ordering, though it is assumed that the input and output file have the same ordering.

3.1 Naive approach

3.1.1 Algorithm description

The first algorithm we present for solving the flow accumulation problem is an adaptation of a naive algorithm. The basic principle is that for each cell c in the grid for which all the source neighbours have been processed, the flow accumulated on c is recursively pushed down, for as long as all the source neighbours for each cell on this path have been processed.

In order to detect if all source neighbours for a cell have been processed, we first run a pass through all the cells counting how many source neighbours each cell has.

3.1.2 Pseudo code

Require: ($\forall$ cells c : $count[c] = 0 \wedge flow[c] = 0$)

```
1:  procedure NaiveFlow(grid G)
    ▷ Pre-processing - count number of source cells for each cell
2:      for each cell c in G do
3:          Increase count[target[c]]
4:      end for
    ▷ Accumulate flow for each cell
5:      for each cell c in G do
6:          d ← c
7:          while count[d] = 0 ∧ target[d] do
8:              count[d] ← ∞                          ▷ Mark d as completed
9:              flow[d] ← flow[d] + 1                 ▷ Add flow for d
10:             flow[target[d]] ← flow[target[d]] + flow[d]   ▷ Push flow to the target of d
11:             Decrease count[target[d]]
12:             d ← target[d]                         ▷ Continue with the next cell of the path
13:         end while
14:     end for
15: end procedure
```

3.1.3 Analysis

I/O analysis

The algorithm scans through all cells twice, taking Scan(N) I/O operations. In the second part of the algorithm water is pushed down following the flow path. This only requires extra I/O operations if we move outside the currently cached area. In order to follow any path efficiently, we need a file ordering that preserves spatially locality, like the z-order ordering. As with the naive flooding algorithm, we have that a minimum of $\mathcal{O}\left(\sqrt{M}\right)$ steps can be taken before extra I/O operations are needed.

For paths longer than this, an extra I/O operation is needed for each block on disk the path passes through. If we look at the disk blocks, and if there are—on average—only a constant number of these paths per block, the I/O complexity is unaffected. Formally, the I/O performance of this part of the algorithm is Scan(N) if the following holds:

$$\left(\sum_{\text{all blocks } B} \text{number of paths that are longer than } \sqrt{M} \text{ in } B \right) = \mathcal{O}\left(\frac{N}{B}\right)$$

Running time analysis

Each cell is visited when the loop reaches it, or when one of its source neighbours is processed. Because each cell can be processed only once, and each cell has at most seven source neighbours, a cell can only be visited a constant number of times. This results in a running time of $\mathcal{O}(N)$.

3.2 Cache-aware approach

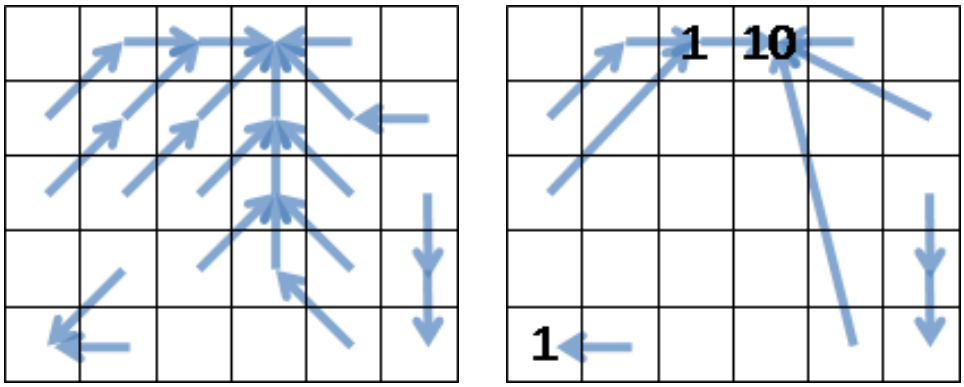
3.2.1 Algorithm description

As with the cache-aware flooding algorithm, for the cache-aware flow accumulation we partition the input graph in partitions that fit into memory, with each partition overlapping its neighbours with one row or column. These overlapping rows and columns, together with the border of the full grid form the vertices of the separator graph G_R. The algorithm then consists of three phases. In the first phase edges and intermediary results are created for the G_R graph, processing one partition at a time. The second phase combines and uses these results to calculate the accumulated flow for each cell in G_R. Finally, each partition is loaded and processed again, calculating the flow for each internal cell.

The principle of accumulating the flow is the same in each step. First we do a pre-processing step, counting the number of incoming edges for each internal cell. Next we iterate through all cells and process every cell c that doesn't have any incoming edges (thus having a count of zero). We then add the flow that has been accumulated in this cell to $t \equiv \mathit{target}[c]$. We also decrease the count of t by one because we now have removed the edge from c to t. Cell c is marked as completed, in order to prevent pushing down the water twice. If the count for t is zero, we recurse on t, thus following the flow path, until either the end of the path is reached, or one of the cells on the flow path has other incoming edges.

For the first phase we need to create edges and calculate the intermediate flow for the edges on the partition border, which we accomplish in a two-step process. First we push the flow for all internal cells down their flow-paths to the border cells, using the process described above. Next we follow the flow-paths for all border cells, and create an edge to the first target cell on the border we find. After this is done we end up with the flow graph G_R, which is significantly smaller than the original graph.

Figure 3.1(a) shows an example flow graph for a partition. Flow for the internal cells is pushed



(a) Partition with flow graph (b) Intermediate flow and G_R edges

Figure 3.1: Computing intermediate flow and edges for a partition

to the border cells, and edges for G_R are created between border cells. The result of this is shown in figure 3.1(b).

In the second phase we accumulate the flow for the G_R graph. The third phase consists of accumulating the flow for the internal cells for each partition again. Since we already have the correct flow for the cells on the border we can now calculate the total flow for each internal cell.

3.2.2 Pseudo code

Require: ($\forall$ cells c : $flow[c] = 0$) ▷ *flow* is the output file

1: **procedure** CacheAwareFlow(grid G)
▷ Phase one—create edges and calculate intermediate flow
2: **for each** partition P **do**
▷ Initialisation and reading from disk
3: Read partition from disk into array $target_P$
4: Clear (set to 0) arrays $flow_P$ and $count_P$
5: **for each** cell c in P, c not on border **do**
6: Increase $count_P[target_P[c]]$
7: $flow_P[c] \leftarrow 1$
8: **end for**
▷ Compute intermediate flow
9: **for each** cell c in G **do**
10: $d \leftarrow c$
11: **while** $count_P[d] = 0 \wedge target_P[d] \wedge d$ not on border **do**
12: $count_P[d] \leftarrow \infty$ ▷ Mark d as completed
13: $flow_P[target_P[d]] \leftarrow flow_P[target_P[d]] + flow_P[d]$ ▷ Push flow to the target of d
14: Decrease $count_P[target_P[d]]$
15: $d \leftarrow target_P[d]$ ▷ Continue with the next cell of the path
16: **end while**

```
17:         end for
            ▷ Create edges
18:         for each cell c in P, c on border do
19:             flow[c] ← flow[c] + flow_P[c]          ▷ Accumulate flow for the current G_R cell
20:             d ← target_P[c]
21:             while d not on border do
22:                 d ← target_P[d]                    ▷ follow flow path
23:             end while
24:             target_{G_R}[c] ← d                    ▷ Create edge from c to d
25:         end for
26:     end for
    ▷ Phase two—accumulate flow for G_R graph
27:     Clear (set to 0) array count_{G_R}
28:     for each cell c in G_R do
29:         Increase count_{G_R}[target_{G_R}[c]]
30:         flow[c] ← flow[c] + 1
31:     end for
32:     for each cell c in G_R do
33:         d ← c
34:         while count_{G_R}[d] = 0 ∧ target_{G_R}[d] do
35:             count_{G_R}[d] ← ∞                     ▷ Mark d as completed
36:             flow[target_{G_R}[d]] ← flow[target_{G_R}[d]] + flow[d]   ▷ Push flow to the target of d
37:             Decrease count_{G_R}[target_{G_R}[d]]
38:             d ← target_{G_R}[d]                    ▷ Continue with the next cell of the path
39:         end while
40:     end for
    ▷ Phase three—compute flow for internal cells
41:     Read partition from disk into array target_P
42:     Clear (set to 0) array count_P
43:     for each cell c in P do
44:         Increase count_P[target_P[c]]
45:         if c not on border then
46:             flow_P[c] ← 1
47:         end if
48:     end for
49:     for each cell c in G do
50:         d ← c
51:         while count_P[d] = 0 ∧ target_P[d] do
52:             count_P[d] ← ∞                         ▷ Mark d as completed
53:             if target_P[d] not on border then
54:                 flow[target_P[d]] ← flow[target_P[d]] + flow[d]   ▷ Push flow to the target of d
55:             end if
56:             Decrease count_P[target_P[d]]
57:             d ← target_P[d]                        ▷ Continue with the next cell of the path
58:         end while
59:     end for
60: end procedure
```

3.2.3 Analysis

I/O analysis

In the first and third phase, partitions are read and written to disk. Neighbouring partitions share a boundary row or column, so the blocks these cells belong to are read once for each partition they belong to. Given the tall-cache assumption ($M = \Omega\left(B^2\right)$), these boundary blocks are read at most four times (in case the boundary cell is on the corner). This means that each block is read $\mathcal{O}(1)$ times, from which follows that the I/O volume for reading and writing the partitions is Scan(N).

Flow accumulation for the graph G_R requires—worst-case—a constant number of I/O operations per cell in this graph. The number of cells in G_R depends on the size and amount of partitions in the grid. Given a grid of $N = W \times H$, and partition size of $w \times h$, we can express the size of the graph as follows: $|G_R| \leq \left(\frac{H}{h} + 1\right) W + \left(\frac{W}{w} + 1\right) H$.

The partition size is chosen such that it fits in the memory, which means that $w \cdot h = \Theta(M)$. If we assume that the partitions are square-like ($w = \Theta(h)$) this gives us, together with the tall-cache assumption, that $w = \Omega(B)$ and $h = \Omega(B)$.

We can now calculate the number of I/O operations needed for G_R:

$$
\begin{aligned}
|G_R| &\leq \left(\frac{H}{h} + 1\right) W + \left(\frac{W}{w} + 1\right) H \\
|G_R| &= \mathcal{O}\left(\frac{H}{h} W + \frac{W}{w} H\right) \\
&= \mathcal{O}\left(\frac{H}{B} W + \frac{W}{B} H\right) \\
&= \mathcal{O}\left(\frac{W \cdot H}{B}\right) \\
&= \mathcal{O}\left(\frac{N}{B}\right) \\
&= \text{Scan}(N)
\end{aligned}
$$

The amount of I/O operations required by the entire algorithm is therefore Scan(N).

Running time analysis

All three phases push water down the flow path, which visits every edge twice—once for counting the number of incoming edges each cell has, and once for the actual flow accumulation—and doesn't need any sorting. Since the flow graph is in fact a forest, the number of edges is at most the number of cells. Therefore the running time required by pushing water down a flow path is $\mathcal{O}(n)$, where n is the number of cells. The first and second phase do this for the whole grid (be it one partition at a time), taking $\mathcal{O}(N)$ time. In the second phase the flow for the G_R graph is

pushed down, taking $\mathcal{O}(|G_R|)$ time. Because the cells in the G_R graph are (a very small) subset of the full grid, the total running time of this algorithm is $\mathcal{O}(N)$.

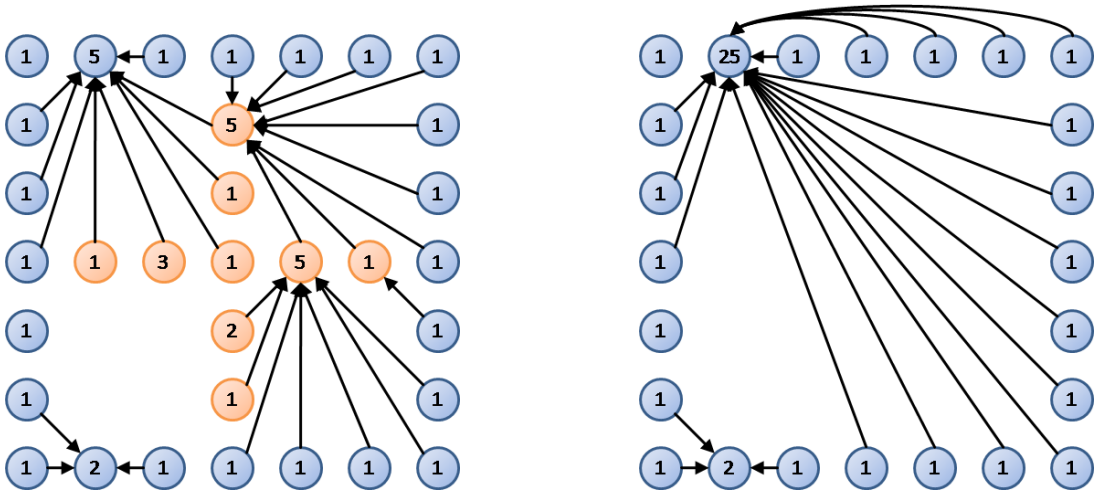
3.3 Cache-oblivious approach

3.3.1 Algorithm description

For the cache-oblivious version of the flow accumulation algorithm we use a divide-and-conquer strategy. In each recursive step the current sub-problem is partitioned into a number of smaller problems. The result for each recursive step are the flow edges between the cells on the border of the partition and for each cell on the border the amount of flow that has been accumulated.

Our base case is a partition of 2×2 cells, for which it is trivial to calculate both the edges between the border cells (since all cells are border cells) and the flow that has been accumulated (which is 1, since no flow from other cells has been pushed onto the current cell). However, since each cell can belong to up to four partitions of size 2×2, we need to make sure that the flow and edge are added exactly once. This can be accomplished by only adding the diagonal, south and east edges for such partitions, and only add the north edge if the partition is on the top-most row of the full grid, and only add the west edge if it is on the left-most column of the grid.

To calculate the flow edges for the combined problem, the internal cells are removed and creating new edges between the border cells. This is done by following the flow for each internal cell c until we reach the first cell that is on the border, we call this cell q, the flow target for c. The flow from c is then pushed to q, and for all cells s on the border that flow into c, we create a new edge from s to q. An example of this merge step can be seen in figure 3.2. In this example the internal cells are coloured orange, and in each cell the amount of flow accumulated so far is stated.



(a) Before merging

(b) After merging

Figure 3.2: Example showing the merge step of the cache-oblivious algorithm

In order to calculate the total amount of flow for these internal cells in the second pass of the algorithm, we have to push the flow from the border inwards to the internal cells, while recursively breaking up the partitions into the smaller sub-problems again. In order to do this efficiently (and not recalculate these edges) we store these edges on disk. Since in the second phase we have to break up the partitions, as opposed to combining them like in the first phase, these edges are visited in the reverse order as they were first phase, making the file listing these edges act as one big stack. Note that because edges are added ("pushed") only in the first phase and only removed ("popped") in the second phase, the I/O behaviour is exactly the same as scanning through a file twice (first forwards, then backwards), therefore we do not need an I/O efficient structure for this stack.

In order to do this we need to make sure that the edges are stored in topological order, so that we accumulate flow for the leaves of the flow tree first in the second phase, and then work our way back to the root. This means we have to create a forest of trees from the edges in each recursive step in the first phase. These flow trees are then visited from the root upwards (following the flow in the reverse direction) which we can also use to calculate the flow target for each cell by "remembering" the last cell on the border that was visited.

As stated before, the second phase consists of the algorithm pushes the flow from the border to the internal cells for which it uses the edges stored in the first phase.

3.3.2 Pseudo code

Require: ($\forall$ cells c : $flow[c] = 0$)

```
1:  procedure CacheObliviousFlow(grid G)
2:      S ← CreateEmptyStack
        ▷ Merging process, calculate flow for the border of the full grid and build the stack
3:      FlowPartition(G)
        ▷ Accumulate flow for internal cells in reverse order
4:      while edge (s, t) ← Pop(S) do
5:          flow[t] ← flow[t] + flow[s]
6:      end while
7:  end procedure
```

```
1:  procedure FlowPartition(partition P)
2:      if width[P] = 2 ∧ height[P] = 2 then
            ▷ Base case, create edges and add flow for current cells in P
3:          E ← ∅
4:          for each cell c in P do
5:              if target[c] in P∧ the edge between c and target[c] was not added before then
6:                  flow[c] ← flow[c] + 1
7:                  E ← E ∪ edge(c, target[c])
8:              end if
9:          end for
10:         return E
11:     else
            ▷ Recurse on sub-problems
```

```
12:         E ← ∅
13:         for each partition P′ in CreatePartitions(P) do
14:             E ← E ∪ FlowPartition(P′)
15:         end for
            ▷ Merge sub-problems
16:         F ← Create forest from edges in E
17:         E ← ∅
18:         for each tree T in F do
19:             FlowTree(Root(T), ⊥)
20:         end for
21:         return E
22:     end if
23: end procedure
```

```
 1: procedure FlowTree(vertex v, vertex q)
 2:     if v on partition border then
 3:         q ← v                                   ▷ Set v as the new flow target
 4:     end if
        ▷ Process tree edges
 5:     for each incoming edge e = (s, v) of v do
 6:         if v not on partition border then
 7:             Push(S, e)                  ▷ Add edge to stack, to calculate water flow
 8:         end if
 9:         if q ≠ ⊥ then
                ▷ There is a flow target for the current branch
10:             if s on partition border then
11:                 E ← E ∪ edge(s, q)                          ▷ Create new edge
12:             else
13:                 flow[q] ← flow[q] + flow[s]          ▷ push water to border vertex (q)
14:             end if
15:         end if
16:         FlowTree(s, q)                                  ▷ Follow tree upwards
17:     end for
18: end procedure
```

3.3.3 Analysis

I/O analysis

We can distinguish two I/O processes. The first one reads the input file and writes to the output file, which follows the same recursion as the cache-oblivious flooding algorithm. Note that even though the second phase isn't written like a recursion, it works on a stack of edges that are added in the recursive order of the first phase. From this follows that the I/O complexity of this first I/O process is Scan(N).

The second I/O process reads and writes the stack file. Since elements are only added in the

first phase, and then removed in the second phase, this stack file is scanned twice, giving an I/O complexity of Scan($|S|$), where $|S|$ is the size of the stack file. In each step of the recursion, in the worst case each edge can be added to the stack. Since the edges form a flow-tree, the amount of edges in each recursive step is at most the amount of vertices in that step. This gives the following recurrence: $S(v) = 4 \cdot S\left(\frac{v}{2}\right) + \mathcal{O}(v)$, which solves to $\mathcal{O}\left(V^2\right) = \mathcal{O}(N)$. This means the I/O complexity of this second process is also Scan(N).

This results in a total I/O complexity of Scan(N).

Running time analysis

The running time of FlowPartition is dominated by creating the forest from the edges, which takes $\mathcal{O}\big(V \log(V)\big)$ time. This gives us the following recurrence: $T(v) = 4 \cdot T\left(\frac{v}{2}\right) + \mathcal{O}\big(v \log(v)\big)$. Solving this with the Master Theorem gives us a running time of $\mathcal{O}\left(V^2\right) = \mathcal{O}(N)$.

3.4 Algorithm comparison

Three different algorithms for the flow accumulation problem have been presented in this chapter. In the following table we will give a short summary of the complexities of each of these algorithms:

Algorithm	I/O complexity	Running time
Naive	Scan(N) [1][2]	$\mathcal{O}(N)$
Cache-aware	Scan(N) + Sort$\left(\frac{N}{B}\right)$	$\mathcal{O}(N)$
Cache-oblivious	Scan(N)	$\mathcal{O}(N)$

Table 3.1: Flow accumulation complexities

3.5 Multiple flow directions

With SFD, the input represents a forest of trees, where each cell has either exactly one or no target cell at all. In MFD the water flow from a single cell is divided to flow to all lower neighbours, which means that each cell can have up to eight target cells, making it a forest of directed acyclic graphs (DAG). What this means in practice is that instead of having a river that has a width of one cell (in SFD), MFD rivers have a width of multiple cells.

Instead of a single $target[c]$ for a cell c, in MFD we have a set $targets[c]$, containing all the target cells of c. There also is a function $flowamount[c, d]$, giving the percentage of flow from cell c to cell d, given that $d \in targets[c]$. All flow that is accumulated on a cell must be passed down—and new flow cannot be created—which means that the following must hold:

[1] Using a file ordering that preserves spatial locality, like z-order
[2] For "normal" maps

$$(\forall \text{ cells } c :: \mathit{targets}[c] \neq \emptyset \Rightarrow \sum_{d \in \mathit{targets}[c]} \mathit{flowamount}[c, d] = 1)$$

$$(\forall \text{ cells } c, d : d \in \mathit{targets}[c] : 0 \leq \mathit{flowamount}[c, d] \leq 1)$$

In the cache-aware (section 3.2) and naive (section 3.1) algorithms the same two step approach is used for accumulating flow. The first step counts the number of incoming edges per cell, and in the second step flow is accumulated recursively for cells have no incoming edges (anymore). We will now present an adaptation of this approach for MFD, in which we note that because there no longer is a single flow path downwards, we need to change the procedure of following all flow paths downwards as well.

```
1:  procedure MFDFlow(grid G)
        ▷ Step 1—count number of source cells for each cell
2:      for each cell c in G do
3:          for each cell d in targets[c] do
4:              Increase count[d]
5:          end for
6:      end for
        ▷ Step 2—accumulate flow for each cell
7:      for each cell c in G do
8:          MFDFlowPaths(c)
9:      end for
10: end procedure

11: procedure MFDFlowPaths(cell c)
        ▷ Check if we are allowed to push down the flow for cell c
12:     if count[c] = 0 ∧ targets[c] ≠ ∅ then
13:         count[c] ← ∞                                  ▷ Mark c as completed
14:         flow[c] ← flow[c] + 1                         ▷ Add flow for c
15:         for each cell d in targets[c] do
16:             flow[d] ← flow[d] + flowamount[c, d] · flow[c]   ▷ Push flow to d
17:             Decrease count[d]
18:             MFDFlowPaths(d)          ▷ Continue with the next cell of the path
19:         end for
20:     end if
21: end procedure
```

The adaptation to the cache-oblivious algorithm from section 3.3 is of a similar fashion. Instead of topologically sorting a tree from the root to the leaves, each DAG needs to be topologically sorted in a similar fashion; namely that cell $c \prec$ cell d if there exists a sequence of cells $S = \{s_0, s_1, \ldots s_n\}$, such that $c \in \mathit{targets}[s_0] \wedge s_0 \in \mathit{targets}[s_1] \wedge \ldots \wedge s_n \in \mathit{targets}[d]$.

Another counter-effect of using MFD is that the resulting flow graphs in the cache-aware and cache-oblivious algorithm are no longer guaranteed to be planar.

For the cache-aware algorithm the number of edges in a partition (of size $w \times h$) can increase from $\mathcal{O}(w + h)$ to $\mathcal{O}\big((w + h)^2\big)$, increasing the worst-case size of the graph G_R from $\mathcal{O}\big(\frac{N}{B}\big)$ to

$\mathcal{O}(N)$.

In the cache-oblivious algorithm, a section at level i can hold up to $\mathcal{O}\left(\frac{N}{2^i}\right)$ edges, giving an I/O complexity of $\text{Scan}\left(N\sqrt{N}\right)$.

This means that both of these algorithms are no longer guaranteed to be efficient under all circumstances.

In the naive algorithm each cell still is visited at most a constant number of times, so this does not affect the analysis. Even though there are much more paths, it is not expected to affect performance since the rivers followed are the same, only wider.

Chapter 4

Testing

4.1 Test setup

All tests have been run on a Dell Optiplex GX620 computer, equipped with a 3.00 GHz Pentium 4 processor with 1 gigabyte of RAM. A single Seagate ST320506AS 250GB hard drive was used for input, output and temporary file storage. The operating system, Ubuntu 7.04 (kernel 2.60.20-16), was placed on a separate hard drive (Samsung HD080HJ, 80GB).

All programs were compiled using the GNU C++ Compiler (GCC) version 4.1.2, with the flags `-O3 -D_LARGE_FILE_SOURCE -D_FILE_OFFSET_BITS=64`.

Running times are logged using the `/usr/bin/time` command. Between each consecutive test the temporary and output directories are emptied.

Prior to the testing of runtime speeds, all algorithms were verified to be correct by comparing the output results of the to one another. Each algorithm is tested with six different input data sets:

Name	Size	# of cells	# data cells	% no-data
SRTM 37-2	$6,000 \times 6,000$	36,000,000	11,970,920	67%
SRTM 38-2	$6,000 \times 6,000$	36,000,000	27,541,900	23%
SRTM NLD	$12,000 \times 6,000$	72,000,000	39,512,820	45%
NEUSE 40FT	$17,630 \times 12,555$	221,334,650	78,016,734	65%
NEUSE 20FT	$35,260 \times 25,110$	885,378,600	312,064,767	65%
NEUSE 10FT	$70,520 \times 50,220$	3,542,378,600	1,248,258,876	65%

Table 4.1: Input data sets

The SRTM[1] data sets cover The Netherlands at a resolution of 90m. SRTM 37-2 contains the west side and SRTM 38-2 the east side. These maps have been combined into the SRTM NLD map.

The second series of data sets covers the Neuse river delta, located in North Carolina in the United States of America, at a resolution of 40, 20 and 10 feet.

The DEMs of these data sets are stored in binary files (one using row-major and one in z-order ordering) using a 32 bit floating point (the C++ *float* data type) value for storing the height

[1] Shuttle Radar Topology Mission

Algorithm	SRTM 37-2		SRTM 38-2		SRTM NLD	
	Time (s)	Eff (%)	Time (s)	Eff (%)	Time (s)	Eff (%)
Naive (R)	0:42	99%	0:48	100%	1:37	93%
Naive (Z)	0:41	99%	0:46	100%	1:41	86%
Cache-aware (8k × 4k)	0:42	99%	0:48	100%	1:44	84%
Cache-aware (4k × 4k)	0:38	99%	0:45	94%	1:29	88%
Cache-aware (2k × 2k)	0:39	84%	0:48	75%	1:22	84%
Cache-oblivious (RD)	3:10	99%	3:22	99%	11:17	58%
Cache-oblivious (RS)	3:01	99%	3:15	100%	10:43	56%
Cache-oblivious (ZD)	3:06	97%	3:19	97%	7:19	88%
Cache-oblivious (ZS)	3:10	100%	3:26	99%	6:58	91%
Naive (TZ)	0:47	87%	0:52	89%	1:43	88%
Cache-aware (T 8k × 4k)	0:41	100%	0:47	100%	1:41	86%
Cache-oblivious (TZS)	3:13	99%	3:26	100%	6:58	92%
Cache-aware (2 × 4k × 4k)	0:38	185%	0:38	184%	1:49	120%
Cache-aware (8 × 2k × 2k)	0:33	195%	0:33	196%	1:24	148%

Table 4.2: Flooding times for the SRTM datasets

of each cell. The output of the flooding algorithms is a similar file, but containing the flooded heights.

For the flow accumulation problem a flow routing step has been run on each input file, creating a file containing the flow direction for each cell, which are stored using 8 bit unsigned integers (bytes). The output of the flow accumulation algorithms are binary files containing the flow accumulation for each cell represented by 32 bit unsigned integers.

The cache-aware algorithms use partition sizes with a width and height of $2^n + 1, n \in \mathbb{N}$, which aligns the read/write operations nicely on the disk, giving a better performance.

4.2 Flooding

The tables 4.2 and 4.3 contain the running time of each of the algorithms for each dataset, as well as their efficiency (percentage of the time the CPU was busy). The cache-aware algorithm was run with partition sizes of 8193 × 4097 (the largest block that would fit in memory), 4097 × 4097 and 2049 × 2049. The cache-oblivious and naive algorithms have been run using both row-major (R) and z-order (Z) ordering. The cache-oblivious algorithm has two more variants, one using dynamic memory allocation (D) for the edges, and the other using static allocation (S).

Also, an extra series of tests has been run (marked with "T" in the tables), using both disks in the testing machine. For the cache-aware and cache-oblivious algorithms the output file was put on the operating system disk, while keeping the input and temporary file on the data disk. The naive algorithm used the operating system disk to store the temporary file since it requires simultaneous access to either the input and temporary file or the output and the

Algorithm	NEUSE 40FT		NEUSE 20FT		NEUSE 10FT	
	Time (s)	Eff (%)	Time (s)	Eff (%)	Time (s)	Eff (%)
Naive (R)	17:39	27%	14:28:07	2%	N/A	
Naive (Z)	9:40	49%	42:34	44%	7:14:29	17%
Cache-aware (8k × 4k)	7:49	68%	31:20	61%	2:36:34	47%
Cache-aware (4k × 4k)	7:44	64%	36:55	49%	3:39:37	32%
Cache-aware (2k × 2k)	8:22	51%	47:58	34%	5:00:35	21%
Cache-oblivious (RD)	2:22:24	14%	15:23:47	9%	63:08:13	9%
Cache-oblivious (RS)	2:20:56	13%	15:23:17	8%	62:47:45	8%
Cache-oblivious (ZD)	28:49	66%	2:02:37	65%	8:12:44	66%
Cache-oblivious (ZS)	29:33	66%	2:00:59	64%	8:04:04	65%
Naive (TZ)	7:07	67%	30:58	61%	6:49:51	18%
Cache-aware (T 8k × 4k)	7:14	73%	28:13	67%	2:12:44	55%
Cache-oblivious (TZS)	24:02	82%	1:37:29	81%	6:28:28	81%
Cache-aware (2 × 4k × 4k)	6:05	121%	29:82	82%	2:47:00	50%
Cache-aware (8 × 2k × 2k)	6:12	103%	42:57	45%	4:52:47	25%

Table 4.3: Flooding times for the Neuse datasets

temporary file. The cache-aware tests used a partition size of 8193 × 4097, testing on the cache-oblivious algorithm was done using the z-order version with static memory allocation and the naive algorithm used z-order as well.

The detailed results (appendix A.1.2) showed that for the cache-aware algorithm the flooding both computation and I/O use a significant amount of running time. While the program is busy reading (or writing) data from disk, the CPU is idling, and when the program is busy flooding a partition, no I/O operations are performed. This different from the cache-oblivious and naive algorithms, in which the I/O operations are intermixed in the computation, and therefore don't have such a regular structure. Because the partitions that are processed in this algorithm are independent of one another, it is very easy to create a multi-threaded version of the cache-aware flooding program. This way when one thread is busy flooding a partition, another thread could be reading the next partition already. Several threads could also be busy performing computations simultaneously, thus taking advantage of multiple cores and/or processors.

A multi-threaded version of the cache-aware program was implemented and tested with two different settings—one using 2 threads with a partition size of 4097 × 4097 and another one with a partition size of 2049 × 2049 on 8 threads. The program uses synchronisation code for the I/O, so that only one thread can read (or write) data at any time. Because the Pentium 4 processor in the test computer has HyperThreading support, multiple threads can perform calculations at the same time, thus allowing the efficiency to exceed 100%.

The naive, row-major test for the NEUSE 10FT data set was stopped test after 60 hours, at which point roughly 60% of the data had been processed, however the speed was down to less than 4% per day.

The z-order tests on the same file performed badly as well, since the number of watersheds and

edges found by the algorithm was so large they still fit in naive in the first part of the algorithm, which creates the watersheds and edges, but not anymore so for the part of the algorithm that floods the watershed graph, since it still used the same internal memory algorithm for flooding. This resulted in a really poor performance for this part of the algorithm, in fact it took more than 60% of the total running time. In contrast, the amount of time required for flooding a watershed graph in the other test cases, where it did fit into memory, was about 4% of the total running time.

TerraStream

We received the TerraStream[2] program from one of its authors. This way we could test the same datasets on the same machine, and be able to compare the running times of their program to the algorithms in this paper. Flooding (called hydrological conditioning) in TerraStream is done by two programs.

The first program (*CalcPersistence*) detects with sinks and their "saddle" vertices, which are the points where they will overflow in another sink or into the ocean. It also assigns a significance to each sink, called the persistence value. This can be used to only remove the small sinks and keep the large ones.

The second program (*Condition*) floods the sinks that have a persistence value lower than an user-specified bound τ, and then writes the result back to disk. In order to get the same flooding results from the TerraStream program and our algorithms the tests have been ran with $\tau = \infty$.

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
CalcPersistence	3:01	7:57	11:51	24:13	1:51:54	7:23:34
Condition	0:28	1:21	2:14	5:35	38:24	2:32:33
Total time	3:29	9:18	14:05	29:48	2:30:18	9:56:07

Table 4.4: Flooding times using the TerraStream algorithm

Also, the programs were configured such that the input, output and temporary files where all stored on the same disk. The results of these tests are shown in table 4.4.

4.3 Flow accumulation

Tables 4.5 and 4.6 show the running times and efficiency of the flow accumulation algorithms for each dataset. The cache-aware algorithm has been run on different partition sizes, the biggest being 8193 × 8193 cells in size. The cache-oblivious and naive algorithms have been run with both z-order (Z) and row-major (R) file ordering.

An extra series of tests was run (marked with "T" in the tables), where the operating system disk was used as well. For the cache-oblivious and naive algorithm the input file was in z-order,

and the cache-aware algorithm was run with partition sizes of 8193 × 4097, since these gave the best results in the regular tests.

Because the running time of the cache-aware algorithm is dominated by the I/O operations (appendix A.2.2), there is little improvement expected from a multi-threaded version, which therefore hasn't been created.

	SRTM 37-2		SRTM 38-2		SRTM NLD	
Algorithm	Time (s)	Eff (%)	Time (s)	Eff (%)	Time (s)	Eff (%)
Naive (R)	0:05	99%	0:07	99%	0:20	65%
Naive (Z)	0:06	99%	0:08	98%	0:23	59%
Cache-aware (8k × 8k)	0:04	96%	0:05	99%	0:09	89%
Cache-aware (8k × 4k)	0:04	100%	0:05	99%	0:09	99%
Cache-aware (4k × 4k)	0:04	99%	0:05	99%	0:23	35%
Cache-aware (2k × 2k)	0:03	99%	0:05	99%	0:20	41%
Cache-oblivious (R)	0:37	99%	1:20	99%	1:21	94%
Cache-oblivious (Z)	0:42	88%	1:25	93%	1:31	87%
Naive (TZ)	0:06	99%	0:08	99%	0:14	99%
Cache-aware (T 8k × 4k)	0:04	98%	0:05	100%	0:08	99%
Cache-oblivious (TZ)	0:37	99%	1:19	100%	1:55	99%

Table 4.5: Flow accumulation times for the SRTM datasets

	NEUSE 40FT		NEUSE 20FT		NEUSE 10FT	
Algorithm	Time (s)	Eff (%)	Time (s)	Eff (%)	Time (s)	Eff (%)
Naive (R)	1:58	32%	18:28	21%	1:51:07	22%
Naive (Z)	1:32	43%	10:18	26%	41:28	26%
Cache-aware (8k × 8k)	2:46	15%	12:36	14%	54:41	13%
Cache-aware (8k × 4k)	1:08	34%	8:17	21%	39:03	18%
Cache-aware (4k × 4k)	1:30	25%	11:14	16%	1:05:30	11%
Cache-aware (2k × 2k)	0:55	41%	10:17	17%	1:18:45	9%
Cache-oblivious (R)	8:42	50%	1:04:56	29%	6:20:13	22%
Cache-oblivious (Z)	6:17	70%	28:37	62%	1:58:54	62%
Naive (TZ)	1:24	47%	8:40	31%	33:50	32%
Cache-aware (T 8k × 4k)	0:54	43%	5:45	29%	25:34	26%
Cache-oblivious (TZ)	5:49	77%	25:08	71%	1:43:56	69%

Table 4.6: Flow accumulation times for the Neuse datasets

Chapter 5

Conclusion

We have presented three different algorithms for both the flooding and flow accumulation problems. All of these algorithms run—under normal conditions— in Scan(N) I/O operations and $\mathcal{O}(N)$ computation time. However, experimentation (chapter 4) has shown that there are still quite big differences between the actual running times of the different algorithms. The cache-aware algorithms are the quickest, followed closely by the naive algorithms. The cache-oblivious variants are the slowest.

In addition, the TerraStream flooding algorithm is outperformed by all of the flooding algorithms described in this paper, though the TerraStream flooding does provide support for flooding with a threshold value. According to the results from the TerraStream paper[2], their flow-accumulation program is slightly slower than their flooding program, whereas our flow-accumulation programs are approximately four times faster than the flooding programs.

A few remarks need to be made. The naive and cache-oblivious variants require a file ordering that preserves spatial locality. For our implementations z-order ordering is used (see chapter 6 for an in-depth overview), however it is to be expected that other orderings that preserve spatial locality—for example the Hilbert curve—have a comparable performance. If a different file ordering is used, the division of the partitions and the order in which they are visited in the cache-oblivious variant might need to be changed as well in order to better suit the file-ordering.

However, most grid data that is available comes in row-major order. This requires that the input data is first converted into the proper ordering, and an efficient way for doing so is needed.

The test results also show that the running time of the cache-aware algorithms greatly depends on the size of the partitions used. The detailed results show that these differences are caused by the difference in I/O time, but the exact reason for this behaviour is unknown. It is however consistent, so a smaller map can be used to determine the ideal partition size for a system.

5.1 Further research

The cache-aware and cache-oblivious algorithms use the same principles and techniques in both the flooding and flow-accumulation problems. Further research needs to be done to see if these techniques can also be adapted to other water flow problems, for example the flow-routing and watershed labelling problems.

Also flow accumulation using MFD needs to be explored further. While a short overview of the

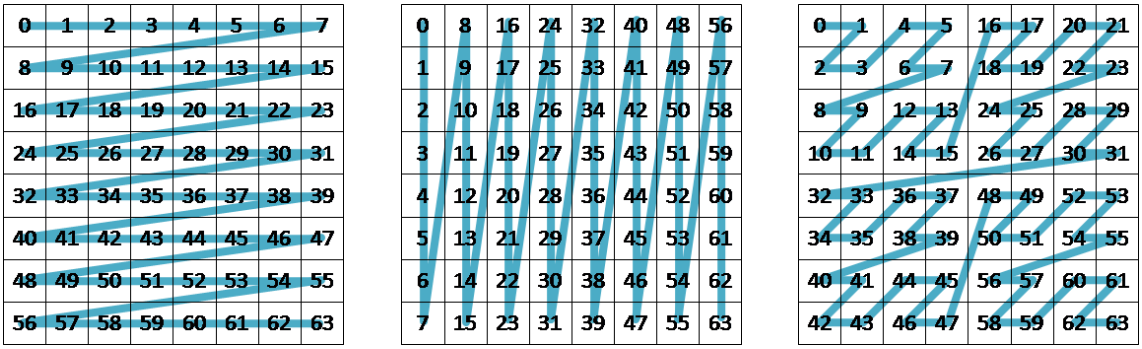
adaptations required to transform the algorithms from SFD into MFD are given in section 3.5, more in-depth analysis and an actual implementation still needs to be done. The graphs used in the cache-aware and cache-oblivious algorithms are no longer planar when MFD is used. This is because the source vertices have multiple target vertices, which can cross each other when connecting the source cells of rivers to their target cells. It might however be possible to create a planar flow-graph by combining the source and target vertices of these rivers.

Chapter 6

Z-order

The Digital Elevation Models (DEMs) used here are 2D matrices, with a height stored for each cell. These are stored on disk in a single. A file is essentially a 1D matrix (array), which means that such a DEM cannot be stored in a file in a straight-forward way. There are several ways to store a matrix in a file, most commonly row-major (figure 6.1(a)) or column-major (figure 6.1(b)) ordering is used, because of its simplicity. Row-major ordering stores the input file by concatenating rows, whereas in column-major ordering the columns are placed after each other. With both of these orderings it is easy to calculate the position of a cell in the file, or to calculate the position in the matrix, from a given file position.

A downside row-major and column-major ordering is that they have poor spatial locality. For example, the spatial distance between a cell and the one below or above it is very short, but if this DEM is stored in a row-major order, the distance between these two cells on disk is the width of an entire row, which makes it very inefficient to examine the grid in this way.



(a) Row-major (b) Column-major (c) Z-order

Figure 6.1: Three different file orderings for an 8×8 input file

An ordering that tries to preserve spatial locality is the so-called z-order. In which cells in each block of 2×2 cells are connected in a z-shaped fashion. These blocks are connected in the same way recursively, until the whole grid has been connected. An example of this can be seen in figure 6.1(c). The location of a cell at position $(x, y) \equiv (x_n \ldots x_1 x_0, y_n \ldots y_1 y_0)$, where x_i is the i-th bit of the x-coordinate, can be calculated by interleaving the bits: $y_n x_n \ldots y_1 x_1 y_0 x_0$. This operation can be reversed in order to calculate the position of a cell in the matrix, given a location in the file.

Bit interleaving is a costly operation on most platforms, so a faster way to calculate the location

of a given cell in a file is needed. For this two methods are used in the implementation of the algorithms—namely lookup tables and dilated arithmetic.

Using lookup tables, it is possible to efficiently calculate the position of a cell with coordinates (x, y) in a file (absolute positioning). This is done by looking at the individual x and y components in the interleaved file position:

$$\begin{aligned}
\text{BitInterleave}(x, y) &= y_n x_n y_{n-1} x_{n-1} \ldots y_1 x_1 y_0 x_0 \\
&= y_n 0 y_{n-1} 0 \ldots y_1 0 y_0 0 + 0 x_n 0 x_{n-1} \ldots 0 x_1 0 x_0 \\
&= 2 \cdot 0 y_n 0 y_{n-1} \ldots 0 y_1 0 y_0 + 0 x_n 0 x_{n-1} \ldots 0 x_1 0 x_0 \\
&= 2 \cdot \text{lookup}(y) + \text{lookup}(x), \text{ with lookup}(i) = 0 i_n 0 i_{n-1} \ldots 0 i_1 0 i_0
\end{aligned}$$

As can be seen from the equations above, it is possible to use the same lookup table for both the x and y coordinates, given that the size of the lookup table is at least bigger than the maximum of the width and height of the matrix that is stored.

The second method used is called *dilated arithmetic*[7] and can be used to calculate the position of cells that are near a given cell (relative positioning). The idea behind this method is that the interleaved x and y are extracted from a given z-order positioning. This can be done by a bitwise **and** operation, setting all the even or odd bits to 0 (6.4 and 6.5). In relative positioning the x or y coordinate will be increased or decreased by some value i generally $i = 1$ or $i = -1$ is used, since these will access the adjacent cells. Since the x and y coordinates are interleaved, i itself must also be interleaved, for example by using the lookup table from the first method. Before the x or y coordinate is increased, the unused bits (either the even or odd) must be set to 1 in order to propagate the carry over these gaps. This can be done by a bitwise **or** operation (6.6 and 6.8). This is not necessary for subtracting from the coordinate, since the unused bits are already 0 from the extraction operation, which will propagate the carry from subtracting (6.7 and 6.9). Finally the (altered) x and y coordinates are combined into a z-order position by adding them together (6.3).

$$ONES_0 \equiv 1010\ldots1010 \tag{6.1}$$

$$ONES_1 \equiv 0101\ldots0101 \tag{6.2}$$

$$\text{BitInterleave}(x, y) \equiv \text{BitInterleave}(x, 0) + \text{BitInterleave}(0, y) \tag{6.3}$$

$$\text{BitInterleave}(x, 0) = \text{BitInterleave}(x, y) \textbf{ and } ONES_1 \tag{6.4}$$

$$\text{BitInterleave}(0, y) = \text{BitInterleave}(x, y) \textbf{ and } ONES_0 \tag{6.5}$$

$$\text{BitInterleave}(x + 1, 0) = (\text{BitInterleave}(x, 0) \textbf{ or } ONES_0 + 1) \textbf{ and } ONES_1 \tag{6.6}$$

$$\text{BitInterleave}(x - 1, 0) = (\text{BitInterleave}(x, 0) - 1) \textbf{ and } ONES_1 \tag{6.7}$$

$$\text{BitInterleave}(0, y + 1) = (\text{BitInterleave}(0, y) \textbf{ or } ONES_1 + 2) \textbf{ and } ONES_0 \tag{6.8}$$

$$\text{BitInterleave}(0, y - 1) = (\text{BitInterleave}(0, y) - 2) \textbf{ and } ONES_0 \tag{6.9}$$

The restriction with the z-order ordering as presented above is that it only works on grids of size $2^n \times 2^n$, otherwise the bit-interleave properties are lost. In order to process DEMs that have different dimensions, the z-order must be adapted to cope with this. The easiest solution is to

insert no-data in the file, expanding the it to $2^n \times 2^n$ cells, where the minimum n is chosen such that $2^n \leq W \wedge 2^n \leq H$ holds. However, this can increase the file size by 300% for square-shaped ($W = \Theta H$) input, or even more for other shaped input.

This can be avoided by virtually expanding the grid instead of expanding the input file. Giving two distinct numbering schemes, one for the file order and the second for the expanded order.

The first is the file order, the order in which the data is stored in the file. Each cell is stored z-order, skipping over the gaps. An example of this can be seen in figure 6.2, in which the numbers show the file order for a 6×5 sized grid. In this figure the normal path is coloured blue, the gaps are green (dark) and the orange (light) lines show the expanded grid.

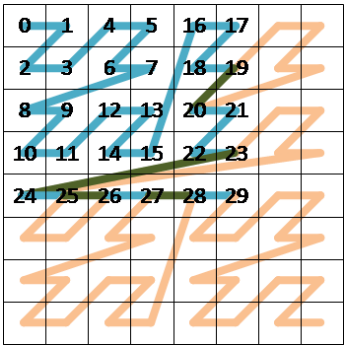


Figure 6.2: An 6×5 z-order on its 8×8 expansion

This expanded grid is the second numbering scheme. Which fits the input file in the smallest possible square of size $2^n \times 2^n$, which is then numbered in the regular z-order way.

In the expanded scheme, the methods described earlier for absolute and relative positioning can be used. Given a position in the expanded scheme, it is easy to check if this position is inside or outside (or even on the border) of the real grid by extracting the interleaved x and y coordinates, and comparing these to the interleaved width and height of the original grid.

Two more operations are needed before this system can be used. The first is calculating the file position from a given expanded position, and the second is a way to iterate through all cells using the expanded position (skipping over the gaps—the green (dark) lines in figure 6.2). Both of these operations can be done by using a skip-table, containing the offset between the real and file position at each gap, as well as the next expanded cell position that exists. Table 6.1 shows such a table for a 6×5 sized grid.

This table has three rows, the first row holds the positions (in the expanded numbering scheme) after which the gaps occur. The second row holds the first position after the gap has ended. The third row holds the difference between the expanded position and the actual file position, for all positions greater or equal than the "next" value in the same column.

last	19	27	33	37	49
next	24	32	36	48	N/A
offset	4	8	10	20	N/A

Table 6.1: Skip table for 6×5 z-order

Since the gaps can only occur at the left and bottom borders of the grid, the worst-case size of a skip table is $|S| = \mathcal{O}(W + H)$. Using this table to calculate a file position takes $\mathcal{O}\big(\log(|S|)\big)$ time if no assumptions are made about the position. Looking up $\Theta(N)$ cells takes a total of $\mathcal{O}\big(N \log(|S|)\big)$ time (running a binary search $\Theta(N)$ times). However, in order to keep the

algorithms in our paper I/O efficient, "random" file access is to be avoided. This means that subsequent lookups are spatially close together, it is expected that the elements (columns) in the skip-table that are visited for the lookup are close together as well—which is true for the majority of the grid, except for searches between the bottom row and the row above it—so we can run a linear search starting from the previous location. For searches between the bottom row and the row above it the binary search is still used. The expected time to look up all N cells therefore is likely to be very close to the lower bound of $\Omega\big(N + |S| + W \cdot \log(|S|)\big)$. Given the square-file assumption a lower bound of $\Omega(N)$ operations to look up $\Theta(N)$ cells is expected.

The easiest way to create such a skip-table is to do this while creating the z-order file (for example from a file in row-major order). At this time both the x and y coordinates for each cell are known, as well as the position of the cell in the z-order output file. The coordinates can be used to calculate the expanded position (using lookup tables), and comparing this to the expanded position of the previous cell so jumps can be detected, and the offset can be calculated by subtracting the file position from the expanded position. This table is stored to disk after it has been created so it can be loaded by the algorithms that use it.

For converting the input data to the output data we used a simple program that iterates through the cells in output (z-order) order, calculates the corresponding input position and copies the input value to the output file. The skip-table is calculated at the same time. Table 6.2 lists the times required to convert from row-major to z-order file ordering.

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Time required	0:10	0:10	0:20	2:59	18:27	1:27:30

Table 6.2: Conversion times in seconds

Bibliography

[1] Aggarwal, A., and Jeffrey, S. V. The input/output complexity of sorting and related problems. *Commun. ACM 31*, 9 (1988), 1116–1127.

[2] Danner, A., Molhave, T., Yi, K., Agerwal, P. K., Arge, L., and Mitasova, H. Terrastream: From elevation data to watershed hierarchies, 2007.

[3] Jenson, S. K., and Domingue, J. O. Extracting topographic structure from digital elevation data for geographic information system analysis, 1988.

[4] Meyer, U. Algorithms for memory hierarchies, 2003.

[5] Moore, I., Grayson, R., and Ladson, A. Digital terrain modelling: a review of hydrological, geomorphical, and biological applications, 1991.

[6] O'Callaghan, J. F., and Mark, D. M. The extraction of drainage networks from digital elevation data, 1984.

[7] Thiyagalingam, J., and Kelly, P. Is morton layout competitive for large twodimensional arrays, 2002.

[8] Toma, L., Wickremesinghe, R., Arge, L., Chase, J. S., Vitter, J. S., Halpin, P. N., and Urban, D. Flow computation on massive grids. In *GIS '01: Proceedings of the 9th ACM international symposium on Advances in geographic information systems* (New York, NY, USA, 2001), ACM, pp. 82–87.

[9] Wilson, J. P., and Gallant, J. C. Terrain analysis: Principles and applications, 1991.

Appendix A

Detailed measurements

In this chapter we present the detailed measurements of each test. These were measured using timing code embedded into the programs. The structured layout of the cache-aware algorithm allowed for timing specific portions of the code without a significant loss in performance, something that is not possible in the cache-oblivious and internal memory algorithms, which therefore have less detailed data.

File access in the cache-oblivious and naive algorithms is done by mapping parts of the files into memory. While this provides fast access to the files—the actual reading/writing and caching is completely done by the system—it requires that the space for the output and temporary files is allocated prior to using it. The timing tables of these algorithms therefore have an element named “allocating files” which states the time it took to allocate these files.

A.1 Flooding

A.1.1 Naive algorithm

Row-major order

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
# of watersheds	180,689	433,321	614,197	2,377,218	8,713,500	N/A
# of edges	433,521	1,047,249	1,481,728	7,135,010	25,359,241	N/A
Allocating files	0.43	0.43	0.87	8.58	54.74	N/A
Creating watersheds	40.16	46.34	92.79	850.75	50877.72	N/A
Flooding watersheds	0.33	0.83	1.25	8.71	306.32	N/A
Flooding grid	0.76	0.78	1.61	191.99	847.74	N/A
Total time	41.68	48.39	96.52	1059.03	52086.74	N/A

Z-order

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
# of watersheds	179,140	428,995	608,295	2,377,247	8,713,509	20,939,968
# of edges	428,864	1,023,981	1,453,831	7,061,072	25,148,617	59,051,297
Allocating files	0.43	0.43	0.88	8.67	55.10	248.05
Creating watersheds	39.02	44.27	96.85	384.86	1634.42	6542.89
Flooding watersheds	0.30	0.75	1.12	7.11	109.14	16543.31
Flooding grid	0.76	0.80	2.57	179.45	755.07	2734.39
Total time	40.52	46.25	101.42	580.08	2553.86	26068.88

Z-order—Output file on operating system disk

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
# of watersheds	179,140	428,995	608,295	2,377,247	8,713,509	20,939,968
# of edges	428,864	1,023,981	1,453,831	7,061,072	25,148,617	59,051,297
Allocating files	0.45	0.44	1.09	17.43	75.92	322.89
Creating watersheds	45.21	49.88	97.65	353.45	1458.45	5806.60
Flooding watersheds	0.31	0.76	1.02	7.07	108.58	17675.14
Flooding grid	1.04	0.84	2.70	48.98	214.72	785.04
Total time	47.01	51.92	102.57	426.92	1857.80	24590.54

A.1.2 Cache-aware algorithm

Partition size: 8193 × 4097

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Reading	0.18	0.18	4.51	33.60	222.41	1602.12
Initialising	2.02	2.15	2.57	7.33	15.87	40.75
Sorting	7.23	10.38	17.46	71.86	235.56	893.09
Computing edges	0.00	0.00	12.05	80.86	324.57	1297.86
Creating G_R	0.23	0.04	0.15	1.07	1.82	7.08
Flooding G_R	0.00	0.01	0.02	0.10	0.60	18.93
Processing phase	22.75	26.05	50.63	201.96	828.57	3971.14
Reading	0.19	0.19	8.71	77.11	446.32	3015.52
Initialising	1.43	1.32	2.23	6.74	18.96	72.99
Sorting	7.21	10.53	17.76	73.05	239.70	900.47
Flooding	0.00	0.00	9.10	57.31	231.68	926.99
Writing	0.53	0.51	5.24	46.84	91.36	416.12
Flooding phase	19.32	21.67	52.46	266.77	1050.45	5419.04
Total time	42.07	47.73	103.10	468.75	1879.49	9391.06

Partition size: 4097 × 4097

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Reading	0.17	0.17	7.17	56.04	400.18	3198.99
Initialising	0.95	0.94	1.00	3.65	10.07	37.78
Sorting	6.69	9.28	15.00	63.57	207.22	794.08
Computing edges	6.14	5.55	11.87	78.82	316.28	1329.01
Creating G_R	0.11	0.07	0.24	0.49	2.01	6.29
Flooding G_R	0.01	0.02	0.04	0.17	0.77	3.58
Processing phase	20.50	23.06	48.64	209.75	964.56	5411.32
Reading	0.17	0.17	0.32	83.80	716.87	5690.23
Initialising	1.00	1.01	1.18	4.42	13.07	49.67
Sorting	6.73	9.34	15.68	65.14	214.88	816.98
Flooding	4.53	4.13	9.04	57.10	231.21	968.10
Writing	0.55	3.02	4.73	37.64	50.15	205.06
Flooding phase	17.84	22.09	40.26	253.89	1249.24	7764.16
Total time	38.35	45.15	88.94	463.74	2214.08	13176.31

Partition size: 2049 × 2049

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Reading	0.16	0.16	0.32	93.14	756.83	6216.29
Initialising	0.39	0.39	0.74	2.44	9.17	33.95
Sorting	4.95	7.11	11.89	49.55	164.73	626.01
Computing edges	5.64	5.43	13.92	70.78	299.75	1174.93
Creating G_R	0.23	0.12	0.36	1.12	5.05	19.35
Flooding G_R	0.03	0.04	0.09	0.39	1.77	7.72
Processing phase	17.52	19.74	37.17	224.27	1247.67	8119.51
Reading	0.16	0.16	0.32	128.22	1063.25	8052.79
Initialising	0.48	0.48	0.95	3.15	12.21	47.51
Sorting	4.96	7.17	12.11	51.35	172.73	669.62
Flooding	4.34	3.98	10.60	51.27	220.77	873.88
Writing	6.42	12.39	13.99	38.51	152.16	235.55
Flooding phase	20.97	28.24	44.33	277.94	1629.40	9913.41
Total time	38.51	47.99	81.53	502.32	2877.49	18034.56

Partition size: 8193 × 4097**—Output file on operating system disk**

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Reading	0.19	0.18	3.57	33.76	222.58	1601.94
Initialising	1.44	1.53	2.56	7.33	15.73	40.86
Sorting	7.26	10.35	17.45	71.85	235.39	893.06
Computing edges	0.00	0.00	12.10	81.07	324.97	1298.76
Creating G_R	0.24	0.04	0.15	1.09	1.82	6.30
Flooding G_R	0.00	0.01	0.02	0.11	0.64	18.59
Processing phase	22.06	25.28	49.73	231.36	829.07	3970.84
Reading	0.19	0.19	7.25	58.99	254.12	1693.00
Initialising	1.27	1.29	2.22	6.56	16.10	81.40
Sorting	7.23	10.49	17.79	72.61	237.78	898.94
Flooding	0.00	0.00	9.13	57.17	230.90	925.68
Writing	0.53	0.51	5.00	30.41	102.56	305.62
Flooding phase	19.17	21.56	50.85	266.77	863.48	3991.66
Total time	41.23	46.84	100.58	433.76	1692.90	7963.36

Partition size: 4097 × 4097**—2 threads**

These multi-threaded timings have been split up in "cummulative" time, which is the sum off all time spent in a section over all threads. The "real" time is the actual time it took to complete a phase of the algorithm.

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Reading (cummulative)	0.37	0.33	9.33	66.09	421.63	3964.59
Initialising (cummulative)	1.51	1.52	1.36	4.63	12.46	41.50
Sorting (cummulative)	10.39	13.98	22.71	90.49	280.38	841.61
Computing edges (cummulative)	12.82	11.07	24.34	134.85	465.74	1656.56
Creating G_R (cummulative)	0.67	0.18	0.43	1.01	3.70	11.65
Reading (real)	0.27	0.26	6.80	56.29	401.12	3207.98
Flooding G_R (real)	0.03	0.03	0.07	0.27	1.19	38.90
Processing phase (real)	20.19	20.24	42.41	154.94	613.92	3286.49
Reading (cummulative)	0.51	0.24	20.65	132.80	861.03	5968.30
Initialising (cummulative)	1.40	1.45	1.57	5.54	16.54	129.12
Sorting (cummulative)	10.33	14.09	21.06	77.47	223.10	849.19
Flooding (cummulative)	10.42	8.93	20.92	90.59	295.25	1327.29
Writing (cummulative)	0.76	0.74	43.70	101.56	817.63	5054.97
Reading (real)	0.23	0.17	11.47	98.01	765.72	5831.53
Writing (real)	0.69	0.74	27.77	68.42	348.63	840.89
Flooding phase (real)	17.84	17.74	66.96	209.45	1124.25	6689.34
Total time (real)	38.06	38.00	109.43	364.66	1739.38	10014.82

Partition size: 2049×2049**—8 threads**

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Reading (cummulative)	2.65	4.77	121.85	580.58	5799.24	47742.98
Initialising (cummulative)	2.77	1.98	1.39	4.17	10.07	35.91
Sorting (cummulative)	25.84	43.18	35.27	108.78	173.01	643.80
Computing edges (cummulative)	51.17	34.76	71.30	318.59	379.36	1474.70
Creating G_R (cummulative)	2.01	2.94	1.20	5.81	5.35	24.29
Reading (real)	0.87	0.52	28.63	123.66	797.19	6246.86
Flooding G_R (real)	0.06	0.07	0.15	0.59	2.60	123.43
Processing phase (real)	18.09	17.48	37.46	131.18	798.18	6247.62
Reading (cummulative)	6.18	5.82	59.19	725.73	3043.95	20617.03
Initialising (cummulative)	1.64	1.03	3.68	4.38	16.08	425.41
Sorting (cummulative)	34.66	44.54	41.03	54.34	178.21	669.22
Flooding (cummulative)	32.78	28.40	85.84	67.33	287.28	1160.22
Writing (cummulative)	1.87	3.55	124.18	1061.89	10665.32	65917.11
Reading (real)	0.36	0.69	13.10	173.57	1431.62	10102.34
Writing (real)	1.87	2.40	18.28	65.49	343.74	1004.54
Flooding phase (real)	14.93	15.06	46.16	240.59	1775.61	11107.08
Total time (real)	33.08	32.61	83.77	372.35	2576.43	17478.42

A.1.3 Cache-oblivious algorithm

Row-major order, dynamic edge allocation

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Allocating files	0.52	1.21	7.54	37.47	175.28	634.82
Creating edges	184.64	195.60	507.92	4772.28	25656.00	106543.72
Flooding	4.94	5.25	161.44	3733.98	29595.05	120114.56
Total time	190.10	202.07	676.91	8543.73	55426.48	227293.24

Row-major order, static edge allocation

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Allocating files	0.52	0.52	9.96	39.53	172.46	636.48
Creating edges	175.79	188.73	468.27	4780.08	25547.37	104352.72
Flooding	4.97	5.23	164.28	3636.16	29677.41	121076.09
Total time	181.28	194.48	642.51	8455.76	55397.28	226065.32

Z-order, dynamic edge allocation

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Allocating files	4.54	4.55	9.67	39.85	184.13	651.79
Creating edges	175.06	187.59	370.89	1285.95	5267.80	21451.11
Flooding	6.42	6.91	58.63	452.75	1904.99	7460.48
Total time	186.03	199.05	439.18	1778.54	7357.04	29563.38

Z-order, static edge allocation

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Allocating files	0.52	0.54	8.32	39.85	182.31	649.51
Creating edges	182.92	199.00	365.63	1278.33	5183.73	20964.99
Flooding	6.45	6.64	43.88	454.48	1892.54	7429.55
Total time	189.90	206.18	417.83	1772.66	7258.64	29044.05

Z-order, static edge allocation—Output file on operating system disk

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Allocating files	0.53	0.52	7.94	27.98	112.02	387.55
Creating edges	185.97	199.28	370.79	1220.32	4914.85	19673.41
Flooding	6.40	6.51	39.10	193.14	821.79	3246.40
Total time	192.89	206.31	417.82	1441.44	5849.06	23307.44

A.1.4 TerraStream

Times marked with $_{i}$ are done using internal memory algorithms, whereas the mark $_{e}$ is used to indicate if an external memory algorithm was used.

CalcPersistence

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Creating edges	18.96	40.10	60.49	133.17	541.60	2114.13
Sorting items	26.33	122.03	181.40	381.79	1827.29	6602.66
Removing nodes	134.77	313.90	467.17	822.35	3609.64	15107.23
Union Find	$_{i}$0.26	$_{i}$0.55	$_{i}$0.68	$_{e}$113.03	$_{e}$723.99	$_{e}$2750.16
Total time	180.73	477.15	711.34	1453.52	6714.14	26613.91

Condition

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Sorting peristence tree					11.15	48.14
Sorting wlabel stream					763.52	3136.82
Raise elevation	$_{i}$0.59	$_{i}$1.24	$_{i}$1.81	$_{i}$4.65	$_{e}$68.51	$_{e}$162.11
Flooding	$_{i}$3.24	$_{i}$13.01	$_{i}$24.45	$_{i}$53.54	$_{e}$232.76	$_{e}$894.94
Writing out	22.72	64.69	106.05	273.63	1219.98	4879.90
Total time	27.84	81.40	133.85	335.29	2303.82	9152.93

A.2 Flow Accumulation

A.2.1 Naive algorithm

Row-major order

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Allocating files	0.64	0.65	1.71	25.02	118.76	461.01
Counting incoming edges	2.02	2.42	11.07	24.08	119.95	564.75
Accumulating flow	2.46	3.78	7.26	69.50	869.97	5641.66
Total time	5.12	6.84	20.03	118.61	1108.69	6667.46

Z-order

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Allocating files	0.65	0.65	1.69	25.17	120.04	461.57
Counting incoming edges	2.39	3.23	14.59	27.74	149.94	616.12
Accumulating flow	2.75	4.35	7.16	39.33	347.96	1410.23
Total time	5.79	8.23	23.44	92.24	617.98	2487.96

Z-order—Temporary file on operating system disk

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Allocating files	0.67	0.67	1.40	21.13	122.30	489.03
Counting incoming edges	2.36	3.06	5.68	28.98	158.71	605.28
Accumulating flow	2.73	4.37	7.32	33.80	238.43	935.07
Total time	5.77	8.10	14.41	83.90	519.86	2029.62

A.2.2 Cache-aware algorithm

Partition size: 8193 × 8193

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Reading	0.10	0.10	0.13	3.68	58.75	412.12
Initialising	0.93	1.09	1.75	4.48	16.15	60.54
Computing edges	0.59	1.17	1.77	4.16	17.89	76.30
Creating G_R	0.02	0.02	0.04	0.12	0.44	1.84
Flowing G_R	0.00	0.00	0.00	0.00	0.01	0.04
Processing phase	1.64	2.38	3.69	12.45	93.25	550.83
Reading	0.05	0.05	0.10	17.66	120.16	841.84
Initialising	0.77	0.87	1.65	4.98	19.68	76.55
Computing flow	0.55	1.13	1.70	3.97	17.32	74.77
Writing	0.52	0.52	2.09	127.41	506.01	1736.88
Flooding phase	1.90	2.58	5.55	154.03	663.17	2730.05
Total time	3.53	4.96	9.25	166.47	756.44	3280.88

Partition size: 8193 × 4097

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Reading	0.09	0.09	0.13	4.19	48.10	412.86
Initialising	0.83	0.99	1.63	4.25	15.83	60.06
Computing edges	0.66	1.31	1.96	4.63	19.59	82.17
Creating G_R	0.02	0.03	0.06	0.15	0.60	2.53
Flowing G_R	0.00	0.00	0.00	0.01	0.02	0.06
Processing phase	1.60	2.42	3.78	13.22	84.14	557.68
Reading	0.05	0.05	0.10	6.51	124.49	853.29
Initialising	0.77	0.87	1.65	5.00	19.76	76.82
Computing flow	0.62	1.27	1.91	4.48	19.33	81.57
Writing	0.52	0.52	1.05	39.40	249.17	773.30
Flooding phase	1.90	2.71	4.72	55.39	412.74	1784.99
Total time	3.51	5.13	8.50	68.61	496.90	2342.69

Partition size: 4097 × 4097

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Reading	0.10	0.10	0.16	2.44	83.10	741.74
Initialising	0.75	0.91	1.50	4.08	15.61	59.89
Computing edges	0.61	1.18	1.79	4.17	17.14	70.21
Creating G_R	0.03	0.04	0.06	0.20	0.79	3.56
Flowing G_R	0.00	0.00	0.00	0.01	0.02	0.07
Processing phase	1.48	2.23	3.51	10.91	116.66	875.48
Reading	0.08	0.08	0.14	1.83	183.94	1598.97
Initialising	0.77	0.88	1.65	5.00	19.88	77.84
Computing flow	0.58	1.15	1.76	4.10	17.02	70.28
Writing	0.57	0.57	16.18	68.96	336.47	1306.76
Flooding phase	2.00	2.68	19.74	79.89	557.31	3053.85
Total time	3.49	4.92	23.25	90.80	673.97	3929.34

Partition size: 2049 × 2049

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Reading	0.11	0.11	0.23	1.27	127.60	1378.81
Initialising	0.64	0.80	1.40	3.95	15.45	59.58
Computing edges	0.56	1.09	1.65	3.83	15.56	62.87
Creating G_R	0.04	0.07	0.12	0.35	1.47	6.60
Flowing G_R	0.00	0.00	0.00	0.01	0.04	0.17
Processing phase	1.36	2.07	3.40	9.41	828.57	1508.04
Reading	0.11	0.11	0.24	0.91	240.94	2215.87
Initialising	0.77	0.87	1.65	5.08	19.77	78.67
Computing flow	0.54	1.06	1.62	3.78	15.25	62.96
Writing	0.62	0.63	12.92	36.09	181.52	858.71
Flooding phase	2.04	2.66	16.43	45.86	457.48	3216.22
Total time	3.40	4.74	19.83	55.27	617.61	4724.28

Partition size: 8193 × 4097**—Output file on operating system disk**

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
Reading	0.14	0.08	0.13	4.11	47.88	413.00
Initialising	0.83	0.99	1.64	4.25	15.82	60.07
Computing edges	0.66	1.32	1.97	4.59	19.51	82.26
Creating G_R	0.02	0.03	0.06	0.15	0.63	2.59
Flowing G_R	0.00	0.00	0.00	0.01	0.02	0.06
Processing phase	1.65	2.42	3.79	13.11	83.87	557.99
Reading	0.05	0.05	0.10	14.93	130.46	564.38
Initialising	0.77	0.87	1.67	5.04	19.71	76.61
Computing flow	0.62	1.27	1.92	4.44	18.99	80.77
Writing	0.53	0.52	1.06	16.72	92.28	254.00
Flooding phase	1.97	2.72	4.75	41.13	261.42	975.76
Total time	3.62	5.14	8.54	54.23	345.29	1533.77

A.2.3 Cache-oblivious algorithm

Row-major order

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
# remaining edges	11,930,783	27,553,362	39,467,634	78,270,514	312,972,565	1,254,027,350
Allocating files	0.44	0.43	0.87	13.18	82.75	327.22
Creating edges	35.77	77.48	109.26	367.57	2441.85	15723.89
Accumulating flow	1.09	2.57	10.57	141.75	1370.81	6761.95
Total time	37.29	80.48	120.69	552.53	3895.41	22813.18

Z-order

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
# remaining edges	11,930,783	27,553,362	39,467,634	78,270,514	312,972,565	1,254,027,350
Allocating files	0.43	0.44	0.87	13.22	85.62	329.66
Creating edges	40.35	80.86	120.81	292.30	1191.98	4955.29
Accumulating flow	1.20	4.07	9.80	71.22	439.94	1849.38
Total time	41.98	85.37	131.47	376.74	1717.54	7134.44

Z-order—Output file on operating system disk

	SRTM			NEUSE		
	37-2	38-2	NLD	40FT	20FT	10FT
# remaining edges	11,930,783	27,553,362	39,467,634	78,270,514	312,972,565	1,254,027,350
Allocating files	0.44	0.43	0.86	17.23	77.97	326.25
Creating edges	35.84	75.69	109.62	280.93	1110.72	4499.09
Accumulating flow	1.15	3.30	5.22	51.55	319.41	1410.24
Total time	37.44	79.43	115.71	349.71	1508.13	6235.93