Fast Generation of Multiple Resolution Instances of Raster Data Sets

Lars Arge
MADALGO*
Aarhus University, Denmark
large@madalgo.au.dk

Herman Haverkort
Department of Computer Science
TU Eindhoven, the Netherlands
cs.herman@haverkort.net

Constantinos Tsirogiannis
MADALGO*
Aarhus University, Denmark
constant@madalgo.au.dk

ABSTRACT

In many GIS applications it is important to study the characteristics of a raster data set at multiple resolutions. Often this is done by generating several coarser resolution rasters from a fine resolution raster. In this paper we describe efficient algorithms for different variants of this problem.

Given a raster $\mathcal{G}$ of $\sqrt{N} \times \sqrt{N}$ cells we first consider the problem of computing for every $2 \leq \mu \leq \sqrt{N}$ a raster $\mathcal{G}_\mu$ of $\sqrt{N}/\mu \times \sqrt{N}/\mu$ cells such that each cell of $\mathcal{G}_\mu$ stores the average of the values of $\mu \times \mu$ cells of $\mathcal{G}$. We describe an algorithm that solves this problem in $\Theta(N)$ time when the handled data fit in the main memory of the computer. We also provide two algorithms that solve this problem in external memory, that is when the input raster is larger than the main memory. The first external algorithm is very easy to implement and requires $O(\text{sort}(N))$ data block transfers from/to the external memory, and the second algorithm requires only $O(\text{scan}(N))$ transfers, where $\text{sort}(N)$ and $\text{scan}(N)$ are the number of transfers needed to sort and scan N elements, respectively.

We also study a variant of the problem where instead of the full input raster we handle only a connected subregion of arbitrary shape. For this variant we describe an algorithm that runs in $\Theta(U \log N)$ time in internal memory, where U is the size of the output. We show how this algorithm can be adapted to perform efficiently in the external memory using $O(\text{sort}(U))$ data transfers from the disk.

We have also implemented two of the presented algorithms, the $O(\text{sort}(N))$ external memory algorithm for full rasters, and the internal memory algorithm that handles connected subregions, and we demonstrate their efficiency in practice.

*Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

ACM SIGSPATIAL GIS '12, November 6-9, 2012. Redondo Beach, CA, USA

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*

General Terms

Theory

Keywords

I/O-efficient algorithms, Raster processing, Multiscalar analysis,

1. INTRODUCTION

Scale is one of the most important parameters of spatial analysis. In many GIS applications, examining the same data set at different scales may lead to different interpretations. One example is the identification of landforms in real-world landscapes; looking at a scale of a few meters, a studied region may appear flat, but at a larger scale it might only be a small part of a wide peak, or appear on the spine of a ridge.

In many cases, there is no definite answer as to which scale is the most appropriate for examining a given data set. For this reason, it is quite common that the analysis of the data set is repeated for a large range of scales, once for each different scale. Then, a conclusion is drawn given the results of each different repetition of the analysis. In the most common setting the studied data set is a raster, a grid of square cells where each cell is associated with a scalar value. In this paper, we use the terms "grid" and "raster" interchangeably to refer to this type of data set. To examine a raster data set at multiple scales, the standard approach is to create multiple rasters, each one representing the original raster in a different "coarser" resolution.

The above approach is widely used in numerous GIS studies with applications that range from terrain analysis to remote sensing. For example, Fisher *et al.* [6] use this method in their landform classification algorithm. In their study the input raster represents a real-world terrain, and the value of each cell represents the terrain elevation at a specific location. Their algorithm first constructs multiple grids G_μ, where a cell in G_μ represents a region of $\mu \times \mu$ cells of the original data set, and the value of the cell is the average of the values of the original $\mu \times \mu$ cells. Then, for each constructed raster they classify each cell to a landform category according to the curvature of the terrain surface at this scale.

Examining a grid in different resolutions is a very popular approach also in remote sensing. Woodcock and Strahler [12] introduce a method for identifying objects in an image. In this setting, the image is represented as a raster where each cell is assigned a grayscale value. The method that they consider involves constructing multiple versions of the same grid in different resolutions. Then, the average local variance is computed for each of the constructed grids; given these variance values, it is then decided which is the appropriate resolution for analysing the image further. This method has received ample attention within the remote sensing community, and there exist many variants of the original algorithm [4, 5, 8, 9].

In all of the above applications, the same problem arises; given a fine resolution raster we want to create multiple versions of this raster of coarser resolutions. More formally, given a grid $\mathcal{G}$ of $\sqrt{N} \times \sqrt{N}$ cells the goal is to compute for every $2 \leq \mu \leq \sqrt{N}$ grids $\mathcal{G}_\mu$ of $\sqrt{N}/\mu \times \sqrt{N}/\mu$ cells such that each cell of $\mathcal{G}_\mu$ stores the average of the values of $\mu \times \mu$ cells of $\mathcal{G}$. We call this problem the Full Grid Scale problem.

A trivial algorithm that solves the Full Grid Scale problem is the following: for every $\mu \in \{2, 3, \ldots, \sqrt{N}\}$ we compute the value for each cell of $\mathcal{G}_\mu$ by explicitly summing up the values of all cells in $\mathcal{G}$ that are covered by this cell of $\mathcal{G}_\mu$, and then dividing this sum by the number of these cells. Although this is a very simple algorithm to implement, it is very inefficient; for each instance $\mathcal{G}_\mu$ the algorithm executes $O(N)$ additions to compute the values of all cells of this instance, which leads to $O(N\sqrt{N})$ total running time.

In many applications it is important to examine only a subset of the input grid $\mathcal{G}$ [4, 8]. This subset is usually determined by a shape-file that indicates which regions within $\mathcal{G}$ are interesting for the current application, and that need to be examined in different scales. More specifically, let S be a given connected region that has been extracted from a raster $\mathcal{G}$. Let $\mathcal{G}_\mu$ be a raster such that $\mu \in \{2, 3, \ldots, \sqrt{N}\}$ and each cell of $\mathcal{G}_\mu$ covers a region of $\mu \times \mu$ cells of $\mathcal{G}$. In this variant of the problem, we want to compute all cells of $\mathcal{G}_\mu$ that cover at least one cell of S, and for each such cell of $\mathcal{G}_\mu$ we also want to compute the average of the values of the cells in S that it covers. In fact we want to compute that for every $\mathcal{G}_\mu$ such that $\mu \in \{2, 3, \ldots, \mu\}$. We call this problem the Subregion Scale problem.

The Subregion Scale problem is far more demanding than handling a full raster; even if we have an efficient algorithm that works on the full raster, this algorithm may not be directly adapted to work efficiently on regions of arbitrary shape. To describe this better, consider a region S that we want to process and let $\mathcal{G}_{\text{box}}(S)$ be the subset of $\mathcal{G}$ that consists of all the cells in $\mathcal{G}$ that are contained in the bounding box of S. Given an algorithm that processes a full raster in multiple resolutions, a simple approach would be to apply this algorithm on $\mathcal{G}_{\text{box}}(S)$. However, if S is quite "skinny", it can be the case that $\mathcal{G}_{\text{box}}(S)$ contains a quadratic number of cells compared to the actual size of S. The situation becomes even worse if we want to examine more than one such regions in $\mathcal{G}$. In theory, it can be possible that out of all regions outlined in a shape-file (imposed over $\mathcal{G}$) there exist $\Theta(\sqrt{N})$ regions such that for each of these regions $\mathcal{G}_{\text{box}}(S)$ has dimension $\Theta(N)$.

The External Memory Model.

The Full Grid Scale problem and the Subregion Scale problem become much more complicated when considering the size of present-day data sets. During the last years, the size of the available GIS data sets has exploded. A single raster that represents a terrain or a remote image can be as large as several gigabytes, or even terrabytes, and may not fit into the main memory of a computer. Thus the largest part of the data set has to reside in the hard disk. Since an algorithm can only process the part of the data set that resides in the main memory, blocks of data have to be copied from the hard disk to the main memory, get processed by the algorithm, and then copied back to the disk to make room in the memory for new blocks to be processed. But moving a single block of data between the hard disk and the memory, a so-called *I/O-operation* or simply *I/O* , takes the same time as roughly hundreds of thousands of basic operations in the CPU. Therefore, in this setting, instead of minimising the number of CPU operations, it becomes important to design algorithms that minimise the number of I/Os.

The standard model for analysing the perfomance of an algorithm in terms of I/Os is the model introduced by Aggarwal and Vitter [1]. In this model the main memory of the computer has size M, while the hard disk has unbounded size. Data are transferred between the disk and the memory in blocks of B elements. In many applications it is reasonable to assume that $M = \Omega(B^2)$. This is the so-called *tall-cache assumption*, which implies that the main memory of the computer can store roughly at least as many blocks as the amount of data that fit to a single block.

The two most fundamental procedures that have been studied in the I/O model are scanning and sorting. Scanning a stream of N consecutive records in the disk can be done in $O(\text{scan}(N))$ I/Os, where $\text{scan}(N) = N/B$. Sorting N consecutive records can be performed using $O(\text{sort}(N))$ I/Os, where $\text{sort}(N) = N/B \log_{M/B} N/B$. Scanning and sorting are frequently used as the basic procedures for designing more involved algorithms in the I/O-model. Hence, the running times of several external memory algorithms are expressed in terms of $\text{scan}(N)$ and $\text{sort}(N)$.

Our Results.

In this paper we describe efficient internal and external-memory algorithms for the Full Grid Scale problem and the Subregion Scale problem. In particular, we provide an algorithm that solves the Full Grid Scale problem in main memory in $\Theta(N)$ time. We also describe two algorithms that solve this problem efficiently in external memory; the first algorithm requires $O(\text{sort}(N))$ I/Os and is very simple to implement, while the second algorithm requires only $O(\text{scan}(N))$ I/Os.

For the Subregion Scale problem, we present an internal memory algorithm that runs in $\Theta(U \log N)$ time, where U is the size of the output. We also show how this algorithm can be adapted to solve the problem in the external memory using $O(\text{sort}(U))$ I/Os.

We have implemented two of the presented algorithms, and we demonstrate their efficiency in practice. More specifically, we have implemented the $O(\text{sort}(N))$ external memory algorithm that solves the Full Grid Scale problem and the $\Theta(U \log N)$ time algorithm that solves the Subregion Scale problem in internal memory. We have con-

ducted experiments where we apply these algorithms on large data sets, and we show their efficiency in practice.

In the next section we provide a theoretical description and analysis of the algorithms that solve the FULL GRID SCALE problem and the SUBREGION SCALE problems. In Section 3 we present an experimental evaluation of the efficiency of the algorithms that we have implemented.

2. DESCRIPTION OF ALGORITHMS

Preliminaries.

Given a grid $\mathcal{G}$ we denote by $\mathcal{G}[i,j]$ the grid cell that appears in the i-th row and j-th column of $\mathcal{G}$. We use $v(i,j)$ to denote the value that is associated with cell $\mathcal{G}[i,j]$; for example, if $\mathcal{G}$ represents a grid terrain then $v(i,j)$ is the height value of the terrain for this specific grid cell. It is quite common that grid data sets contain also cells that have "no data" values; those are cells for which no value record exists. For any such cell $\mathcal{G}[i,j]$, we indicate that by specifying $v(i,j) = \otimes$.

For the ease of presentation, we assume that $\mathcal{G}$ is a square, that is it consists of $\sqrt{N}$ rows and $\sqrt{N}$ columns of cells. Yet, the analysis of the algorithms that we present later on in this paper can be easily extended for general rectangular rasters.

Given a cell $\mathcal{G}[i,j]$ of $\mathcal{G}$, we call the *minions* of this cell the set of all cells $\mathcal{G}[k,l]$ such that $1 \leq k \leq i$ and $1 \leq l \leq j$, and $v(k,l) \neq \otimes$ –see Fig. 1. We denote this set by minions(i,j). We indicate the number of the elements in minions(i,j) by $m(i,j)$, that is $m(i,j) = |\text{minions}(i,j)|$. We define that minions$(i,j) = \emptyset$ if $i < 1$ or $j < 1$; we also consider that minions$(i,j) =$ minions$(\sqrt{N}, j)$ if $i > \sqrt{N}$, and minions$(i,j) =$ minions$(i, \sqrt{N})$ if $j > \sqrt{N}$. We denote the sum of the values of all cells in minions(i,j) by aggr(i,j), that is $\text{aggr}(i,j) = \sum_{\mathcal{G}[k,l] \in \text{minions}(i,j)} v(k,l)$. We call aggr$(i,j)$ the *aggregate* of cell $\mathcal{G}[i,j]$.

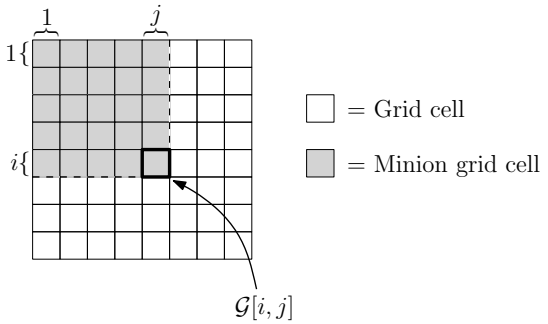

Figure 1: An illustration of the set of minions of grid cell $\mathcal{G}[i,j]$.

Let $\mathcal{G}$ be a grid of dimensions $\sqrt{N} \times \sqrt{N}$, and let μ be an integer such that $1 < \mu \leq \sqrt{N}$. We define $\mathcal{G}_\mu$ as the grid of dimensions $\sqrt{N}/\mu \times \sqrt{N}/\mu$ such that for any cell $\mathcal{G}_\mu[i,j] \in G_\mu$ the value $v_\mu(i,j)$ associated with this cell is equal to the average value of all cells in $\mathcal{G}[k,l]$ for which we have that $(i-1)\mu + 1 \leq k \leq i\mu$ and $(j-1)\mu + 1 \leq l \leq j\mu$. In other words, $\mathcal{G}_\mu$ is a more compact representation of $\mathcal{G}$, where each cell $\mathcal{G}_\mu[i,j]$ covers a region of (at most) $\mu \times \mu$ cells of $\mathcal{G}$, and the value of $v_\mu(i,j)$ is equal to the average value of the at corresponding cells of $\mathcal{G}$. We say that $\mathcal{G}_\mu$ is the *scale instance* of $\mathcal{G}$ at μ, and we call μ the *scale* of this instance.

Given an input grid $\mathcal{G}$ of dimensions $\sqrt{N} \times \sqrt{N}$, in this paper we show how we can efficiently compute grids $\mathcal{G}_\mu$ for all μ such that $2 \leq \mu \leq \sqrt{N}$. At first glance, the size of the output for this problem, that is the total number of cells of all grids $\mathcal{G}_\mu$, would seem to be very large compared to the size of the input grid $\mathcal{G}$. However, this is not the case as we show in the following lemma.

LEMMA 2.1. The total number of cells for all grids $\mathcal{G}_\mu$ with $2 \leq \mu \leq \sqrt{N}$ is $\Theta(N)$, and more specifically it is less than $0.65 \cdot N$.

PROOF. The total number of cells for all grids $\mathcal{G}_\mu$ is equal to:

$$\sum_{\mu=2}^{\sqrt{N}} \frac{N}{\mu^2} < N \sum_{\mu=1}^{\infty} \frac{1}{\mu^2} = N \cdot \zeta(2),$$

where $\zeta(x)$ is the so-called Riemann zeta function [7]. The value of the zeta function converges to a constant for every $x > 1$ and is smaller than 2.62 for all arguments larger or equal to $\frac{3}{2}$. Specifically for $x = 2$ we have that $1.64 \leq z(2) \leq 1.65$. □

2.1 Computing Multiple Instances of an Entire Raster

2.1.1 *Internal Memory Algorithm*

In the FULL GRID SCALE problem, given a raster $\mathcal{G}$ of $\sqrt{N} \times \sqrt{N}$ cells such that $\mathcal{G}$ fits entirely in the main memory of the computer, we want to compute the instances $\mathcal{G}_\mu$ for all scale values $\mu \in \{2, 3, \ldots, \sqrt{N}\}$.

As already mentioned, a simple algorithm that solves this problem is to compute the value of each cell $\mathcal{G}_\mu[i,j]$ by explicitly adding the values of all cells in $\mathcal{G}$ that are covered by $\mathcal{G}_\mu[i,j]$, and then average over the number of these cells. However, this algorithm has $O(N\sqrt{N})$ running time and therefore it is practically infeasible when applied to sufficiently large rasters.

A much more efficient algorithm can be obtained based on the following simple observation; for any cell $\mathcal{G}_\mu[i,j]$ in $\mathcal{G}_\mu$ it holds that:

$$v_\mu[i,j] = \frac{Sum(i,j,\mu)}{Size(i,j,\mu)}, \tag{1}$$

where:

$$Sum(i,j,\mu) = \text{aggr}(i\mu, j\mu) - \text{aggr}(i\mu, (j-1)\mu) \\ -\text{aggr}((i-1)\mu, j\mu) + \text{aggr}((i-1)\mu, (j-1)\mu),$$

and

$$Size(i,j,\mu) = m(i\mu, j\mu) - m(i\mu, (j-1)\mu) \\ -m((i-1)\mu, j\mu) + m((i-1)\mu, (j-1)\mu)$$

Figure 2 provides a schematic description of the expression in (1). From this expression it becomes clear that for computing the value for some cell $\mathcal{G}_\mu[i,j]$ we need only constant amount of data; we need only to know the aggregates and the number of minions of (at most) four cells of $\mathcal{G}$. We call these cells the *defining cells* of $\mathcal{G}_\mu[i,j]$. This approach

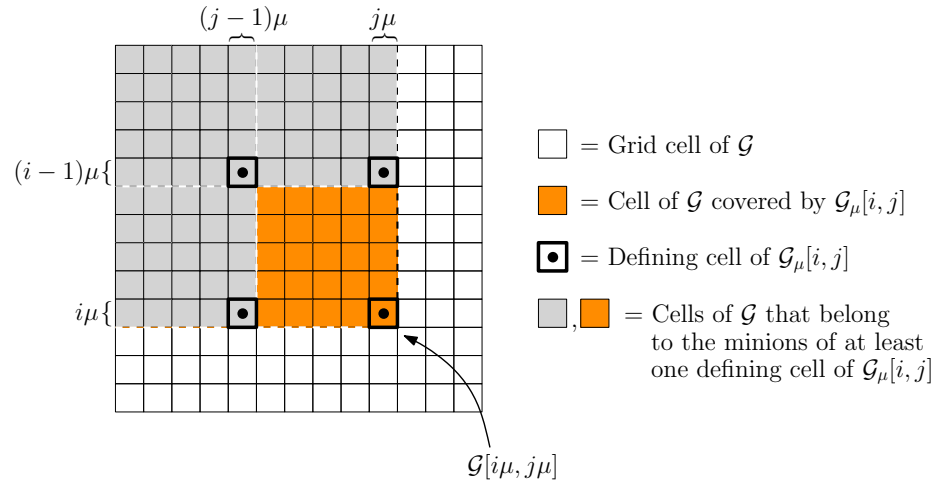

Figure 2: The intuition behind the expression in (1). The value of cell $\mathcal{G}_\mu[i,j]$ can be expressed using the minions of its four defining cells. The sum of the values of the cells of $\mathcal{G}$ that are covered by $\mathcal{G}_\mu[i,j]$ is equal to summing the aggregates of $\mathcal{G}[i\mu, j\mu]$ and $\mathcal{G}[(i-1)\mu, (j-1)\mu]$, and subtracting the aggregates of $\mathcal{G}[(i-1)\mu, j\mu]$ and $\mathcal{G}[i\mu, (j-1)\mu]$. A similar relation applies for the number of the cells covered by $\mathcal{G}_\mu[i,j]$.

is not entirely new, but it follows naturally from well-known methods that deal with problems of geometric searching and dominance [10]. We can derive an efficient algorithm for computing all instances $\mathcal{G}_\mu$ if we precompute efficiently the aggregates and the number of minions for any cell in $\mathcal{G}$. This is what we do in the following simple algorithm.

Algorithm *MultirasterIM*($\mathcal{G}$)
1. Compute $m(i,j)$ for every cell $\mathcal{G}[i,j] \in \mathcal{G}$ and store these values in raster $\mathcal{G}_m$
2. Compute $\text{aggr}(i,j)$ for every cell $\mathcal{G}[i,j] \in \mathcal{G}$ and store these values in raster $\mathcal{G}_{\text{aggr}}$
3. **for** every μ from 2 to $\sqrt{n}$
4. **do for** every cell $\mathcal{G}_\mu[i,j]$ in $\mathcal{G}_\mu$
5. **do** Compute $v_\mu(i,j)$ according to the expression in (1) using the values stored in $\mathcal{G}_m$ and $\mathcal{G}_{\text{aggr}}$

It is easy to show that the running time of *MultirasterIM* is $\Theta(N)$. The values of the cells in rasters $\mathcal{G}_m$ and $\mathcal{G}_{\text{aggr}}$ can be computed in $O(N)$ time with a single scan of $\mathcal{G}$, starting from $\mathcal{G}[1,1]$ and visiting all cells in $\mathcal{G}$ in lexicographical order of their row and column indices. Each repetition of step 5 takes constant time and this step is executed once for each cell of the output rasters $\mathcal{G}_\mu$. In Lemma 2.3 we showed that the total number of cells in the output is $\Theta(N)$, and therefore the running time of the entire algorithm is $\Theta(N)$.

2.1.2 External Memory Algorithms

The problem that we examined in Section 2.1.1 becomes much more challenging if we try to solve it in the external memory model, that is when the input raster $\mathcal{G}$ is so large that it does not fit in the main memory of the computer. In this setting, if we try to apply directly algorithm *MultirasterIM* then, for larger values of μ, we will need for each cell $G_\mu[i,j]$ at least one I/O for fetching its defining cells. In this section we describe two I/O-efficient algorithms for computing the different scale instances of a complete grid $\mathcal{G}$ in the external memory model. The first algorithm is quite simple to implement and to analyse and takes $O(\text{sort}(N))$ I/Os, and the second algorithm takes only $O(\text{scan}(N))$ I/Os. For both of the algorithms we consider that we have precomputed $\mathcal{G}_{\text{aggr}}$ and $\mathcal{G}_m$, this computation can be easily done in $O(\text{scan}(N))$ I/Os by just scanning the cells of the input raster $\mathcal{G}$ in natural order. Next we provide a description for both of the algorithms.

An $O(\text{sort}(N))$ Algorithm .

The main drawback with the I/O-efficiency of algorithm *MultirasterIM* is that the order in which this algorithm handles the required values $\text{aggr}(i,j)$ and $m(i,j)$ is different than the way that these values are stored in $\mathcal{G}_{\text{aggr}}$ and $\mathcal{G}_m$. To solve this problem, we create a stream F that stores this data, which we then sort and scan using fewer I/Os.

More specifically for each cell $\mathcal{G}_\mu[i,j]$ we create (at most) four records in the stream, one record for each defining cell that exists in $\mathcal{G}$. Each record is a tuple of values of the following structure: $< i', j', i, j, \mu, \text{aggr}, m >$. Fields i', j' correspond to the indices of the defining cell $\mathcal{G}[i', j']$ in $\mathcal{G}$, while fields i, j and μ indicate the cell $\mathcal{G}_\mu[i,j]$ of instance $\mathcal{G}_\mu$ whose value $v_\mu(i,j)$ we want to compute. Fields aggr and m are used for storing the aggregate and the number of minions of the defining cell $\mathcal{G}[i', j']$; initially the values of these two last fields are set to zero. Stream F has $\Theta(N)$ size, since it contains four records for each output cell $\mathcal{G}_\mu[i,j]$, and since the number of output cells is $\Theta(N)$ according to Lemma 2.3. Stream F created in $O(\text{scan}(N))$ I/Os, by just enumerating all output cells $\mathcal{G}_\mu[i,j]$, and computing the indices of the corresponding defining cells in $\mathcal{G}$.

After creating F, we sort its records in lexicographical order of the two first fields i', j'. Then we scan the sorted stream sequentially together with the two grids $\mathcal{G}_{\text{aggr}}$ and $\mathcal{G}_m$, filling in the corresponding values of fields aggr, m in each record of F. Next, we sort again the records in F but this time in lexicographical order of the next three fields μ, i, j; now F stores the records that represent the defining cells of the same cell $\mathcal{G}_\mu[i,j]$ consecutively, and with the records of all cells $\mathcal{G}_\mu[i,j]$ stored in order of the scale instance, the row, and the column that they appear. In the last part of the algorithm, we scan F sequentially computing the value $v_\mu(i,j)$ of each cell $\mathcal{G}_\mu[i,j]$ using the expression in (1).

During the execution of the algorithm, we sort and scan a constant number of times streams of $\Theta(N)$ size. Therefore, the total number of I/Os required by this algorithm is $O(\text{sort}(N))$.

An $O(\text{scan}(N))$ Algorithm.

Next we describe how we can solve the FULL GRID SCALE problem in external memory in only $O(\text{scan}(N))$ I/Os, using a more involved strategy. Suppose that given a cell $\mathcal{G}[i,j]$ we can efficiently determine which are the specific cells $\mathcal{G}_\mu[i',j']$ for which $\mathcal{G}[i,j]$ is a defining cell. Then, by maintaining an external stream for each scale instance $\mathcal{G}_\mu$, we can scan $\mathcal{G}_{\text{aggr}}$ and $\mathcal{G}_m$ in a row-by-row manner and for each cell $\mathcal{G}[i,j]$ we copy values $\text{aggr}(i,j)$ and $m(i,j)$ to the stream of this instance if $\mathcal{G}[i,j]$ defines some cell in $\mathcal{G}_\mu$. Of course, it is unrealistic to consider that we can maintain at least one block in memory for the stream of each scale instance; in this way we would have to occupy $\sqrt{N}$ blocks in the main memory, which is not a reasonable assumption in the external memory setting. To tackle this issue, we break the algorithm into two parts; in the first part, we maintain a block in memory only for a subset of the scale instances, only for the $B-1$ instances that are the largest in size. In the second part, we compute the rest of instances using an algorithm similar to *MultirasterIM*; this time without maintaining any memory for each scale instance. In this part, for every $\mu > B$ we compute the value for each $\mathcal{G}_\mu[i,j]$ allowing one I/O for each defining cell that is needed, expecting that the total number of cells in those instances is quite small.

Next we describe thoroughly the first part of the algorithm. To determine efficiently for each cell $\mathcal{G}[i,j]$ which are the cells $\mathcal{G}_\mu[i',j']$ that $\mathcal{G}[i,j]$ defines, we use the following observation; $\mathcal{G}[i,j]$ is a defining cell of some element in $\mathcal{G}_\mu$ if and only if (a) μ is a divisor of both i and j, that is $i = 0 \mod \mu$ and $j = 0 \mod \mu$, or (b) at least one of i,j is equal to $\sqrt{N}$ and the other has μ as a divisor. Hence, we focus on designing an efficient strategy for computing for each $\mathcal{G}[i,j]$ the common divisors of i and j (except the last row and the last column of $\mathcal{G}$ where we need to compute the divisors of one of the indices). To do that we scan $\mathcal{G}_{\text{aggr}}$ and $\mathcal{G}_m$ simultaneously in a row-by-row manner while maintaining two arrays, namely Div_{rows} and Div_{both}. More specifically, each element of these arrays is a list; $\text{Div}_{\text{rows}}[i]$ stores a list with (a subset of) the divisors of i, where i is the index that signifies the i-th row of $\mathcal{G}$. $\text{Div}_{\text{both}}[j]$ stores a list with (a subset of) the common divisors between j, the index of the j-th column of $\mathcal{G}$, and the index of the current row that is currently scanned. In this part of the algorithm, we compute the defining cells only for the $B-1$ largest scale instances, that means the instances $\mathcal{G}_\mu$ for which $\mu \in \{2,\ldots,B\}$. In the meantime, we maintain one open stream F_μ for each one of these scale instances $\mathcal{G}_\mu$. In each stream F_μ we copy from $\mathcal{G}_{\text{aggr}}$ and $\mathcal{G}_m$ the aggregates and the number of minions that correspond to the defining cells of the elements of $\mathcal{G}_\mu$. The summary of the complete algorithm follows; for the ease of presentation, we have not included in this summary how we handle the special cases of the last row and the last column of $\mathcal{G}$.

Algorithm *MultirasterExternal*($\mathcal{G}$)

1. ▷ *First part: Compute the first B instances.*
2. Initialise Div_{rows} so that $\text{Div}_{\text{rows}}[\mu] = \mu$ if $\mu \in \{2,\ldots,B\}$, otherwise $\text{Div}_{\text{rows}}[\mu] = \emptyset$.
3. Initialise Div_{both} so that $\forall i$, $\text{Div}_{\text{both}}[i] = \emptyset$.
4. **for** $i \leftarrow 2$ to $\sqrt{N}$
5. **do** ▷ *Flush all elements of* $\text{Div}_{\text{rows}}[i]$ *in* Div_{both} *and in the next slots of* Div_{rows}.
6. **for** every $\mu \in \text{Div}_{\text{rows}}[i]$
7. **do** Insert μ in $\text{Div}_{\text{both}}[\mu]$.
8. Insert μ in $\text{Div}_{\text{rows}}[i+\mu]$.
9. Empty $\text{Div}_{\text{rows}}[i]$.
10. **for** $j \leftarrow 2$ to $\sqrt{N}$
11. **do for** every $\mu \in \text{Div}_{\text{rows}}[j]$
12. **do** Write $< \text{aggr}(i,j), m(i,j) >$ in F_μ.
13. Insert e in $\text{Div}_{\text{both}}[j+\mu]$.
14. Empty $\text{Div}_{\text{both}}[j]$.
15. **for** $\mu \leftarrow 2$ to B
16. **do** Scan stream F_μ and compute the values of the cells of $\mathcal{G}_\mu$
17. ▷ *Second part.*
18. Compute instances $\mathcal{G}_\mu$ with $\mu > B$ using a process similar to *MultirasterIM*.

It is easy to show that in the first part of *MultirasterExternal* each stream F_μ is assigned the aggregates and the number of minions of exactly those cells in $\mathcal{G}$ that define some cell in $\mathcal{G}_\mu$. At the beginning of the loop in step 10 the lists of Div_{both} store only those integers in $[2,B]$ that are divisors of the i-th row of $\mathcal{G}$; at this point each divisor μ is stored in the list $\text{Div}_{\text{both}}[\mu]$. Every time that list $\text{Div}_{\text{both}}[j]$ is processed, this list contains exactly those integers in $[2,B]$ that are the common divisors of indices i and j. Therefore, the aggregate and the number of minions of cell $\mathcal{G}[i,j]$ are copied only to those streams F_μ for which $\mathcal{G}[i,j]$ is a defining cell. Then, each element μ of the list is propagated to the next slot j' of Div_{both} such that $j' = 0 \mod \mu$. The data of the defining cells are written in every stream F_μ in a row-by-row manner, so that it suffices to scan F_μ only once in order to compute the cells of $\mathcal{G}_\mu$.

Regarding the memory requirements of algorithm *MultirasterExternal*, Div_{rows} and Div_{both} can be efficiently represented using only a constant number of memory blocks. Since we use these lists to store divisors only from 2 to B, the last and the first non-empty slot in each of these arrays can be at most $B-2$ slots apart at any step of the algorithm. Thus, with proper indexing we can maintain these structures using only $\Theta(B)$ memory space.

For each of the streams F_μ we need to store one block in memory, which requires that $M = \Omega(B^2)$. As already mentioned, this is the tall-cache assumption. Later in this section we show how we can skip this assumption without affecting substantially the I/O-performance of this algorithm. We continue with the next theorem.

THEOREM 2.2. Given a grid $\mathcal{G}$ of $\sqrt{N} \times \sqrt{N}$ cells, the problem FULL GRID SCALE can be solved in external memory using $O(\text{scan}(N))$ I/Os.

PROOF. For the first part of algorithm *MultirasterExternal*, the proof of the I/O-efficiency bound is trivial; in this part of the process we need to scan once each of the grids $\mathcal{G}_{\text{aggr}}$ and $\mathcal{G}_m$, and we construct and scan sequentially streams F_μ. According to Lemma 2.3 the total size of these streams is $O(N)$ and therefore $O(\text{scan}(N))$ I/Os suffice for executing this part.

For the second part of the algorithm, since we do not maintain a block in memory for each instance of scale $\mu > B$, at worst case we need to spend a constant number of I/Os for computing each cell of these instances. Hence, the number of I/Os that we need for computing the last instances is at most a constant times the total number of cells of these

instances. The total number of these cells is equal to:

$$\sum_{\mu=B+1}^{\sqrt{N}} \frac{N}{\mu^2} \leq \sum_{\alpha=1}^{\lceil \sqrt{N}/B \rceil} \sum_{\beta=1}^{B} \frac{N}{(\alpha B+\beta)^2} \leq \sum_{\alpha=1}^{\lceil \sqrt{N}/B \rceil} \sum_{\beta=1}^{B} \frac{N}{(\alpha^2 B^2)}$$

$$= \frac{N}{B} \sum_{\alpha=1}^{\lceil \sqrt{N}/B \rceil} \frac{1}{\alpha^2} \leq \frac{N}{B} \sum_{\alpha=1}^{\infty} \frac{1}{\alpha^2} = \frac{N}{B} \cdot \zeta(2) = \Theta(\text{scan}(N)),$$

where ζ is the Riemann zeta function. □

Relaxing the tall-cache assumption.

To decrease the
amount of memory that is required for executing algorithm *MultirasterExternal* we use the following trick; instead of storing one full block of memory for every stream F_μ, we use a buffer whose size is a function of the size of the scale instance $\mathcal{G}_\mu$.

More specifically, suppose that for the memory size M it holds that $M = \Omega(B^{1+\epsilon})$ for some well defined constant ϵ where $0 < \epsilon < 1$. Then for every stream F_μ we store in memory a buffer of size $B/\mu^{1-\epsilon'}$ for some ϵ' chosen such that $0 < \epsilon' < \epsilon$. Each time that a buffer gets full with newly written data, we execute an I/O to flush the data of the buffer in the hard disk.

The total amount of memory that is used to store the described buffers is then:

$$\sum_{\mu=1}^{B} \frac{B}{\mu^{1-\epsilon'}} = B \sum_{\mu=1}^{B} \frac{\mu^{\epsilon'}}{\mu} \leq B^{1+\epsilon'} \sum_{\mu=1}^{B} \frac{1}{\mu} = O(B^{1+\epsilon'} \log B),$$

which is $O(B^{1+\epsilon})$ and fits the amount of memory that we can provide.

Using a buffer of size $B/\mu^{1-\epsilon'}$ for the stream that corresponds to scale instcance $\mathcal{G}_\mu$ leads to as many as $\frac{N}{\mu^2}/\frac{B}{\mu^{1-\epsilon'}}$ I/Os for creating this stream. Therefore, the total number of I/Os that we have to execute using the $B-1$ buffers is:

$$\sum_{\mu=2}^{B} \frac{\frac{N}{\mu^2}}{\frac{B}{\mu^{1-\epsilon'}}} = \frac{N}{B} \sum_{\mu=2}^{B} \frac{1}{\mu^{1+\epsilon'}} \leq \frac{N}{B} \sum_{\mu=2}^{\infty} \frac{1}{\mu^{1+\epsilon'}} = \frac{N}{B} \cdot \zeta(1+\epsilon')$$

$$= O(\text{scan}(N)).$$

2.2 Computing Multiple Instances of a Subregion in the Raster

In this section we describe algorithms that solve efficiently the SUBREGION SCALE problem, that is the problem where we want to compute for every scale instance of $\mathcal{G}$ only those cells that cover a given connected subregion of $\mathcal{G}$. Next we provide a formal definition of the problem.

Let $\mathcal{G}$ be a raster and let region S be a connected subset of the cells in $\mathcal{G}$; that is for any cell $\mathcal{G}[i,j] \in S$ it is possible to reach any other cell in S by traversing a path in $\mathcal{G}$ that crosses only cells of S, and any two cells that are consequtively visited by the path share a common edge or vertex of their boundary.

We denote the number of cells that constitute S by s. Let $\mathcal{G}_\mu$ be a scale instance of $\mathcal{G}$ for some $\mu \in \{2, \ldots, \sqrt{n}\}$. We say that the cell $\mathcal{G}_\mu[i,j]$ is an *active cell* of this instance if the set of (at most) μ^2 cells in $\mathcal{G}$ that are covered by $\mathcal{G}_\mu[i,j]$ contain at least one cell that is also an element of S. Since in this version of the problem we are interested in a subset of $\mathcal{G}$, instead of handling complete rasters, we consider a different input and output format. More specifically, for this variant of the problem we consider that the input is a connected region of cells S given as a list $\mathcal{L}$ whose elements are triples of the form $< i, j, v >$; fields i and j signify a cell $\mathcal{G}[i,j]$ in $\mathcal{G}$ that belong to S, and v is the associated value of this cell. The triples in $\mathcal{L}$ appear in lexicographical order of their two first fields. We consider that the dimensions of the initial grid $\mathcal{G}$ are also provided in the input. The output of the problem is the set of lists $\mathcal{L}_\mu$ for $\mu \in \{2, \ldots, \sqrt{N}\}$ defined as follows. Each list $\mathcal{L}_\mu$ contains exactly those triples $< i, j, v_\mu >$ such that $\mathcal{G}_\mu[i,j]$ is an active cell in scale instance $\mathcal{G}_\mu$ and v_μ is the associated value of this cell. The triples in $\mathcal{L}_\mu$ appear in lexicographical order over their two first fields. We say that $\mathcal{L}_\mu$ represents the *scale instance* of S at scale μ. We continue with providing an upper bound on the size of the output lists.

LEMMA 2.3. The total number of active cells among all scale instances of S is $O(s \log s + \sqrt{N})$.

PROOF. We distinguish the scale instances into two categories; the scale instances $\mathcal{G}_\mu$ for which μ is at most equal to s, and the scale instances for which $\mu > s$. First, we bound the size of the latter category; when $\mu > s$ then $\mathcal{G}_{\text{box}}(S)$ overlaps with at most four cells of $\mathcal{G}_\mu$ and therefore the total number of active cells for all these instances is at most $O(\sqrt{N})$.

For the rest of the scale instances, we first show that the number of active cells in each instance $\mathcal{L}_\mu$ is $O(s/\mu)$. To do that, we split S into connected subsets of μ cells each, and then argue that we need at most a constant number of cells of $\mathcal{G}_\mu$ to cover each subset. More specifically, consider the plane graph $\Gamma_S(V,E)$ where each vertex in V corresponds to the center of a cell in S, and an edge in E connects two vertices if the corresponding cells share part of their boundary. Consider a spanning tree of Γ, and more specifically the planar embedding of this tree as outlined by the corresponding grid cells that belong to S. Consider an Euler tour of this tree, and consider cutting this tour into s/μ pieces such that each piece has μ edges (except at most one piece which has less than μ edges). Each cell in $\mathcal{G}_\mu$ covers the same area as $\mu \times \mu$ cells of $\mathcal{G}$. Therefore, no matter what is the shape of each piece of the tour, we need at most four cells of $\mathcal{G}_\mu$ to cover one piece. Hence we get that $|\mathcal{L}_\mu|$, the size of the instance of S at scale μ, is at most equal to $4\lceil \frac{s}{\mu} \rceil$. Therefore the total size of the output is:

$$\sum_{\mu=1}^{s} |\mathcal{L}_\mu| \leq \sum_{\mu=1}^{s} 4\lceil \frac{s}{\mu} \rceil = O(s \log s)$$

□

It is easy to prove that this bound is tight. Consider the region S that is depicted in Fig. 3. In this case, S is a meandering curve which consists of $\Theta(\sqrt[4]{N^3})$ cells, and $G_{box}(S)$ covers the entire grid $\mathcal{G}$. For every grid $\mathcal{G}_\mu$ where $i \leq \sqrt[4]{N}$, this meandering curve overlaps with $\Theta(\frac{\sqrt[4]{N^3}}{\mu})$ cells and hence the output consists of $\Omega(\sqrt[4]{N^3} \log N)$ cells in total. From this figure it becomes clear that we can actually fit in $\mathcal{G}$ $\Theta(\sqrt[4]{N})$ regions of similar shape, with every region consisting of $\Theta(\sqrt[4]{N^3})$ cells. Thus, given a single grid $\mathcal{G}$ of dimension $\sqrt{N}$ we can outline a set of regions whose scale instances have $O(N \log N)$ size in total.

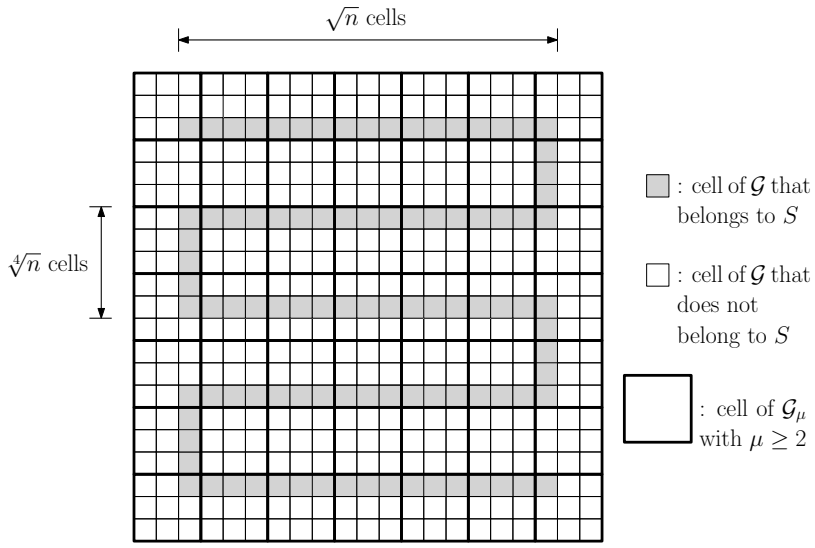

Figure 3: A single connected region of $s = \Theta(\sqrt[4]{N^3})$ cells that produces an output of size $O(s \log s)$. Each maximal horizontal segment of the curve consists of $O(\sqrt{(N)})$ finer resolution cells and each vertical segment consists of $O(\sqrt[4]{N})$ cells.

Of course, this scenario is quite unrealistic: in practice, it is highly improbable that we come across a data set that produces an output of this size, whether we consider the case of a single region, or all the regions outlined by a shape-file. For the rest of this work, we use U to denote the size of the output when the input is a single region S. Next, we present an algorithm for solving the described problem in internal memory in $O(U \log N)$ time. We then show how we can adapt this algorithm to solve the problem efficiently in the I/O-efficient model.

2.2.1 Internal Memory Algorithm

The algorithm that we consider is divided into two parts. In the first part, we compute a superset of the active cells for every scale instance of the input region S. We show that this superset is not very large compared to the actual size of the output, that is it consists of $O(U)$ cells. In this part we compute only the coordinates i, j of cells $\mathcal{G}_\mu[i, j]$ of this superset, but not their associated values $v_\mu(i, j)$. In the second part, we compute the associated values of these cells, discarding those cells that are not active cells.

First Part.

We begin with computing for every instance $\mathcal{G}_\mu$ which cells are potential active cells in this instance. To do that, we first compute the active cells only of those instances $\mathcal{G}_\mu$ for which we have that $\mu = 2^\nu$ for some integer $\nu \leq \log(\sqrt{N})$. We will then use the cells of these instances to compute the potential active cells for the rest of the scales μ.

To compute the active cells of instance $\mathcal{G}_2$, we scan list $\mathcal{L}$ calculating for each element $< i, j, v > \in \mathcal{L}$ the cell $\mathcal{G}_2[i', j']$ that contains cell $\mathcal{G}[i, j]$, and then adding $< i', j', \otimes >$ to $\mathcal{L}_2$. After scanning the entire $\mathcal{L}$, we go through $\mathcal{L}_2$ removing any multiple elements. This can be done by scanning $\mathcal{L}_2$ while keeping track of records that correspond to cells of consecutive grid rows. It is easy to argue that this simple process takes $\Theta(|\mathcal{L}|)$ time. To compute $\mathcal{L}_4$, we apply the same algorithm on $\mathcal{L}_2$, and so on and so forth until we compute $\mathcal{L}_{\lfloor \log(\sqrt{N}) \rfloor}$. We call the lists that we have computed so far the *landmark* lists of S.

Each landmark list $\mathcal{L}_\mu$ contains exactly one record for each active cell of $\mathcal{G}_\mu$; by construction $\mathcal{L}_2$ contains an element $< i', j', \otimes >$ only if $\mathcal{G}_2[i', j']$ covers at least one cell of S, and by induction we can prove that for the rest of the computed lists. To compute a superset of the active cells for the the rest of the scale values we use the following algorithm:

Algorithm *CellLists*($\mathcal{G}$)

1. **for** every μ from 3 to $\sqrt{N}$ such that $\mu \notin \{2, 4, \ldots, 2^{\lfloor \log(\sqrt{N}) \rfloor}\}$
2. **do** Let $\mathcal{L}_\nu$ be the landmark list where ν is the largest possible so that $\nu < \mu$.
3. **do for** every $< i, j, \otimes >$ in $\mathcal{L}_\nu$
4. **do** Compute all cells $\mathcal{G}_\mu[i', j']$ that overlap with cell $\mathcal{G}_\nu[i, j]$ and add these cells to $\mathcal{L}_\mu$.
5. Scan $\mathcal{L}_\mu$ removing multiple elements.

In this algorithm, to compute each one of the remaining lists $\mathcal{L}_\mu$ we use the landmark list $\mathcal{L}_\nu$ that has the largest scale ν such that $\nu < \mu$. We then add one record to $\mathcal{L}_\mu$ for every cell in $\mathcal{G}_\mu$ that overlaps with a cell that appears in $\mathcal{L}_\nu$. We showed that all cells that appear in the landmark lists are active cells. However, this is not always the case for the lists $\mathcal{L}_\mu$ that are computed in algorithm *CellLists*. A cell $\mathcal{G}_\mu[i', j']$ that overlaps with an active cell $\mathcal{G}_\nu[i, j]$ at step 4 may not always be an active cell itself; $\mathcal{G}_\nu[i, j]$ contains at least one of the cells of the input region S, but the intersection of $\mathcal{G}_\nu[i, j]$ with $\mathcal{G}_\mu[i', j']$ may contain none of these cells. Hence, to produce the correct output we need to discard the records in each list $\mathcal{L}_\mu$ that do not represent active cells. This is done in the second part of the algorithm. To this point, we still have to argue that the number of the extra records computed in *CellLists* is not very large compared to the number of the active cells. Indeed, at step 4 given the values of μ and ν there can be at most four cells in $\mathcal{G}_\mu$ that overlap with cell $\mathcal{G}_\nu[i, j]$. Since $\mathcal{G}_\nu[i, j]$ is an active cell, at least one of the overlapping cells in $\mathcal{G}_\mu$ is an active cell as well.

Therefore, after executing *CellLists* each list $\mathcal{L}_\mu$ contains no more records than four times the number of active cells of this scale instance. Thus, the total size of the computed lists is $\Theta(U)$. $\Theta(U)$ is also the running time for this part of the algorithm.

Second Part.

To compute the values of the active cells stored in lists $\mathcal{L}_\mu$, and to get rid of the non-active cells, we consider the following strategy; for every list $\mathcal{L}_\mu$, we "break" each cell $\mathcal{G}_\mu[i,j]$ represented by a record in $\mathcal{L}_\mu$ into its four defining cells in $\mathcal{G}$. In this way we construct a group of $\Theta(u)$ cells of $\mathcal{G}$ in total. Then, for each cell $\mathcal{G}[k,l]$ of this group we calculate values $\text{aggr}(k,l)$ and $m(k,l)$; this calculation is carried out using a data structure that we introduce for this purpose. In the last stage of the algorithm, we compute the associated values for the cells in lists $\mathcal{L}_\mu$ using the expression in (1) and the calculated values aggr, m of their defining cells. At this point we also dump all cells that do not correspond to active cells.

We describe now the second part of the algorithm in more detail. For every list $\mathcal{L}_\mu$, for each record $< i,j,\otimes > \in \mathcal{L}_\mu$ that corresponds to cell $\mathcal{G}_\mu[i,j]$ we compute the defining cells of $\mathcal{G}_\mu[i,j]$. For each of the defining cells $\mathcal{G}[i',j']$ of $\mathcal{G}_\mu[i,j]$, we create a record $< i',j',\mu,i,j,\otimes >$ which we then insert in a min priority queue Q. We call this group of records the *query* records of the queue. Also, we insert in Q another group of records; for every element $< i,j,v >$ of the input list L, the list that represents the input region S, we insert in Q a record of the form $< i,j,1,i,j,v >$. We call these records the *region* records of the queue. The records in Q are ordered lexicographically according to the values of their two first fields. Obviously, we can determine whether a record is a query record or a region record from the values of its third and sixth field.

Our goal is that for every query record $< i',j',\mu,i,j,\otimes >$ in Q we want to compute values $\text{aggr}(i,j)$ and $m(i,j)$. In order to do that we use the following simple tree data structure T_S. This tree is an augmented balanced binary tree, where every node ρ stores two fields $sum[\rho]$ and $size[\rho]$. More specifically, the tree consists of as many leaf nodes as the columns of $\mathcal{G}_{\text{box}}(S)$, the subset of $\mathcal{G}$ that is contained in the bounding box of S. Each leaf node v corresponds to a column of $\mathcal{G}_{\text{box}}(S)$, where $size[\rho]$ stores the number of cells $\mathcal{G}[i,j]$ that have been currently "charged" to this column, and $sum[\rho]$ stores the sum of the associated values $v(i,j)$ of these cells. For each internal node v, $size[\rho]$ stores the sum of the fields $size[\lambda]$ of all leaf nodes $\lambda \in T_S$ that belong to the subtree of ρ, and similarly $sum[\rho]$ stores the sum of the fields $sum[\lambda]$ of all leaf nodes λ that belong to this subtree. After the construction of T_S, the combinatorial structure of the tree remains the same and only the values of the fields $sum[\rho]$ and $size[\rho]$ of the tree nodes can change. Initially, for every node $\rho \in T_S$ the values $sum[\rho]$ and $size[\rho]$ are set to zero.

We consider two kinds of procedures on T_S: an *addition* operation $Add(v,j)$ and a *query* procedure $Query(j)$. In an addition operation $Add(v,j)$, we symbolically "add" a new cell to the j-th column of $\mathcal{G}_{\text{box}}(S)$; the value v of the cell is added to to the field $sum[\rho]$ of the leaf node ρ that corresponds to the j-th column of $\mathcal{G}_{\text{box}}(S)$, and the value $size[\rho]$ is increased by one. The respective fields of the ancestor nodes of ρ are updated accordingly, and hence the whole operation can be executed in $O(\log s)$ time. A query $Query(j)$ returns a pair of values (a,b) where a is equal to the sum of the values $sum[v]$ of all leaf nodes v which correspond to the first j columns of $\mathcal{G}_{\text{box}}(S)$, and b equals the sum of values $size[v]$ of these leaf nodes. It is straightforward to show that $Query(j)$ can be implemented so that it runs on T_S in $O(\log s)$ time.

We use T_S to compute efficiently the values $\text{aggr}(i,j)$ and $m(i,j)$ of the cells that are represented by the query records in Q. While Q is not empty, we extract the record that is currently the front element of the queue. If this record is a region record $< i,j,1,i,j,v >$ then we execute an addition operation $Add(v,j)$ in T_S. If the extracted record is a query record $< i',j',\mu,i,j,\otimes >$ then we execute a query $Query(j')$ in T_S. The region records in Q are extracted and fed to T_S in such an order so that, at any point, $Query(j')$ that is executed in favor of query record $< i',j',\mu,i,j,\otimes >$ returns the aggregate $\text{aggr}(i',j')$ and the number of minions $m(i',j')$ of cell $\mathcal{G}[i',j']$. Since there are $O(U)$ records stored in Q initially, the total time taken for the operations executed on T_S is $O(U \log s)$. The total time taken for the operations in Q is $O(U \log U)$, which is $O(U \log N)$ according to Lemma 2.3.

For each query $Query(j')$ executed on T_S for a record $< i',j',\mu,i,j,\otimes >$, we create a tuple $< \mu,i,j,i',j',\text{aggr}(i',j'),m(i',j') >$, where $\text{aggr}(i',j')$ and $m(i',j')$ are the values computed in this query. We store all tuples that are computed in this manner in a list L_{tuples}, and after processing all queries on T_S we sort the elements of L_{tuples} in lexicographical order of their three first fields. At this point, every four consecutive elements in the list correspond to defining cells of the same cell $\mathcal{G}_\mu[i,j]$. In the last stage of the algorithm, we scan the sorted list and calculate the associated value of each cell $\mathcal{G}_\mu[i,j]$ using the data from the tuples of its defining cells and expression (1). In the meantime, we fill in the calculated associated values in the respective records of lists $\mathcal{L}_\mu$. If for some cell $\mathcal{G}_\mu[i,j]$ the denominator in expression (1) is evaluated as zero, then $\mathcal{G}_\mu[i,j]$ is not an active cell and we remove the corresponding record from $\mathcal{L}_\mu$. The time taken for sorting list L_{tuples} and updating lists $\mathcal{L}_\mu$ is $O(U \log N)$, which is the also the running time for the entire algorithm.

2.2.2 Extending the Algorithm to the I/O-Model

The algorithm presented in the Section 2.2.1 can be easily adapted to compute efficiently the scale instances of a region S in the I/O-model as well. The first part of the algorithm can be extended trivially to work in external memory by handling lists $\mathcal{L}_\mu$ in a single external stream in $O(\text{scan}(U))$ I/Os.

For the second part, we only need to substitute the data structures that we use with I/O-efficient ones. For storing the region/query records we can use an I/O-efficient priority queue that handles each operation in $O(1/B \log_{M/B} U/B)$ I/Os amortized [3]. Instead of the binary tree T_S we can use an augmented version of the *buffer tree* structure [2]. Based on the standard analysis that applies for buffer-trees, and by extending the buffer tree with the same satellite data as the internal memory tree structure of the previous section, we get an external data structure that can perform $\Theta(U)$ addition operations and queries in $O(U/B \log_{M/B}(N/B))$ I/Os in total.

After handling all operations on the buffer tree, we still have to spend $O(\text{sort}(U))$ I/Os for sorting the output of the queries, so as to compute the values of the cells in S.

Hence, the total number of I/Os that take place during the execution of the entire algorithm is $O(\text{sort}(U))$.

3. EXPERIMENTAL RESULTS

We have implemented two of the algorithms that are described in the previous section in order to evaluate their efficiency in practice. In particular, we have implemented the external memory algorithm that solves the FULL GRID SCALE problem in $O(\text{sort}(N))$ I/Os, and the internal memory algorithm that solves the SUBREGION SCALE problem.

The algorithms were implemented in C++, and we used the GNU g++ compiler, version 4.4.3. Our experiments were conducted on an Intel Core 2 Duo CPU, which consists of two 3.16 GHz processors. No parallelization was used in the implemented algorithms, hence at any point during the execution the computations where handled by a single processor. The main memory of the system that we used is 3.8 Gigabytes. However, the maximum amount of memory that was used during each experiment was 1.5 Gigabytes. We ran our implementation on a Linux Ubuntu operating system version 10.04.

For the implementation of the external memory algorithm we used the software library TPIE (the *Templated Portable I/O Environment*) [11]. This library offers I/O-efficient algorithms for scanning and sorting large files in external memory.

In our first experiment, we tested the efficiency of our $O(\text{sort}(N))$ external memory algorithm that solves the FULL GRID SCALE problem. To do this, we ran our implementation of the algorithm on a raster data set of approximately 625 million cells. This raster consists of 25,000 rows and as many columns, and the value of each cell is a 4-byte `float`. Therefore, the total size of the raster file is approximately 2.53 Gigabytes, and thus did not entirely fit in the 1.5 Gigabyte of working memory. The external memory algorithm took 2 hours and 53 minutes to complete the computation of all the scale instances of the input raster.

In our second experiment, we tested the implementation of the internal memory algorithm that solves the SUBREGION SCALE problem. For this experiment we used a raster of $12{,}500 \times 12{,}500$ cells, that is approximately 156 million cells in total. The values of the cells were 4-byte `floats`, and the total size of the raster was roughly 600 Megabytes. On this raster we imposed a shape-file that outlined 160 connected polygonal regions, most of them being non-convex. The largest of these regions consisted of 23 million cells approximately, while the average size of each region was roughly one million cells. After extracting the cells of all the regions using a sweep-line algorithm, we ran our algorithm separately for each region. The running time of the algorithm for all 160 regions was 62 minutes and 20 seconds, while the maximum time taken for a single region was 8 minutes and 20 seconds.

4. CONCLUSIONS

In the present paper, we studied the problem of computing efficiently several instances of a raster, or of a connected subregion of a raster, so that each instance represents the original entity in a coarser resolution. We described efficient algorithms for solving each of these problems in two different settings; when the entire input fits in the main memory, and when the input data resides in external memory. We also implemented two of the presented algorithms and demonstrated their efficiency in practice. An interesting topic for future research would be to design algorithms that compute efficiently the different scale instances of a connected region by taking advantage of the geometric properties of its shape, such as fatness.

5. ACKNOWLEDGMENTS

We would like to thank Peder Klith Bøcher for valuable discussions and advice that he provided during all the stages of the presented research.

6. REFERENCES

[1] A. Aggarwal and J.S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[3] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In*Proc. 34th Annual ACM Symposium on Theory of Computing*, pages 268–276, 2002.

[4] P.K. Bøcher and K.R. McCloy. The fundamentals of average local variance - Part I: Detecting regular patterns. *IEEE Transactions on Image Processing*, 15:300–310, 2006.

[5] P.K. Bøcher and K.R. McCloy. The fundamentals of average local variance - Part II: Sampling simple regular patterns with optical imagery. *IEEE Transactions on Image Processing*, 15:311–318, 2006.

[6] P. Fisher, J. Wood, and T. Cheng. Where is Helvellyn? Fuzziness of multiscale landscape morphometry. *Transactions of the Institute of British Geographers*, 29(1):106–128, 2004.

[7] A.A. Karatsuba and S.M. Voronin. *The Riemann Zeta-Function*. Walter de Gruyter, Berlin, 1992.

[8] K.R. McCloy and P.K. Bøcher. Optimizing image resolution to maximize the accuracy of hard classification. *Photogrammetric Engineering and Remote Sensing*, 73:893–903, 2007.

[9] D.P. Ming, J.Y. Yang, L.X. Li, and Z.Q. Song. Modified ALV for selecting the optimal spatial resolution and its scale effect on image classification accuracy. *Mathematical and Computer Modelling*, 54:1061–1068, 2011.

[10] F.P. Preparata and M.I. Shamos. *Computatioanl Geometry: An Introduction*. Springer-Verlag, 1985.

[11] TPIE, the Templated Portable I/O Environment. `http://www.madalgo.au.dk/tpie/`.

[12] C.E. Woodcock and A.H. Strahler. The factor of scale in remote sensing. *Remote Sensing of Environment*, 21:311–332, 1987.