Open access • Book Chapter • DOI:10.1007/978-3-642-40450-4_6

An Optimal and Practical Cache-Oblivious Algorithm for Computing Multiresolution Rasters — Source link

Lars Arge, Gerth Stølting Brodal, Jakob Truelsen, Constantinos Tsirogiannis

Institutions: Aarhus University

Published on: 02 Sep 2013 - European Symposium on Algorithms

Related papers:

- Communication-Optimal Parallel Standard and Karatsuba Integer Multiplication in the Distributed Memory Model.
- An efficient parallel algorithm for O(N^2) direct summation method and its variations on distributed-memory parallel machines
- Fast Arithmetic in Algorithmic Self-assembly
- Cache-oblivious data structures for orthogonal range searching
- Hierarchical Bin Buffering: Online Local Moments for Dynamic External Memory Arrays

Share this paper:

View more about this paper here: https://typeset.io/papers/an-optimal-and-practical-cache-oblivious-algorithm-for-24uurxgukw

An optimal and practical cache-oblivious algorithm for computing multiresolution rasters

Lars Arge, Gerth Stølting Brodal, Jakob Truelsen, Constantinos Tsirogiannis

MADALGO*, Department of Computer Science, Aarhus University, Denmark
{large, gerth, jakobt, constant}@madalgo.au.dk

Abstract. In many scientific applications it is required to reconstruct a raster dataset many times, each time using a different resolution. This leads to the following problem; let $\mathcal{G}$ be a raster of $\sqrt{N} \times \sqrt{N}$ cells. We want to compute for every integer $2 \leq \mu \leq \sqrt{N}$ a raster $\mathcal{G}_\mu$ of $\lceil \sqrt{N}/\mu \rceil \times \lceil \sqrt{N}/\mu \rceil$ cells where each cell of $\mathcal{G}_\mu$ stores the average of the values of $\mu \times \mu$ cells of $\mathcal{G}$. Here we consider the case where $\mathcal{G}$ is so large that it does not fit in the main memory of the computer.

We present a novel algorithm that solves this problem in $O(\text{scan}(N))$ data block transfers from/to the external memory, and in $\Theta(N)$ CPU operations; here $\text{scan}(N)$ is the number of block transfers that are needed to read the entire dataset from the external memory. Unlike previous results on this problem, our algorithm achieves this optimal performance without making any assumptions on the size of the main memory of the computer. Moreover, this algorithm is cache-oblivious; its performance does not depend on the data block size and the main memory size.

We have implemented the new algorithm and we evaluate its performance on datasets of various sizes; we show that it clearly outperforms previous approaches on this problem. In this way, we provide solid evidence that non-trivial cache-oblivious algorithms can be implemented so that they perform efficiently in practice.

1 Introduction

Rasters are one of the most common formats for modelling spatial data. A raster is a 2-dimensional grid of square cells where each cell is assigned a real value. Among other applications, rasters are used to represent real-world terrains; in this case each cell corresponds to a region of a terrain, and the value of the cell indicates the average height of the terrain in this region. Today, it is possible to acquire massive rasters that represent terrains with very fine resolution; the size of each cell in such a raster can be less than one square meter. Yet, studying a terrain in such a small scale might lead to wrong conclusions. This happens for example when we want to identify landforms on terrains; when we study a terrain at a scale of a few meters, we might identify many small peaks concentrated

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

within a small area. Yet, when looking on a larger scale, these peaks may be a part of another landform; for instance a rough ridge, or a valley.

To tackle this problem, we need to have a method that can analyze the same raster in many different scales. Fisher *et al.* [4] use such a method in their landform classification algorithm; their algorithm constructs multiple rasters $\mathcal{G}_\mu$, where a cell c of $\mathcal{G}_\mu$ covers the same region as $\mu \times \mu$ cells of the original fine-resolution raster. The value assigned to c is equal to the average of the values of the original $\mu \times \mu$ cells. Given the constructed rasters $\mathcal{G}_\mu$, it is then possible to search for landforms at different scales.

Reconstructing a raster in different resolutions is an important tool for many other scientific applications; in remote sensing, Woodcock and Strahler [9] introduced an algorithm to extract the average size of tree canopies in grayscale images of forests. Here, an image is represented by a raster of square pixels, where each pixel is assigned a grayscale value. Their algorithm reconstructs many instances of a given image raster, in exactly the same way as the algorithm of Fisher *et al.* constructs different instances of a terrain raster. For their application, it is critical to construct one instance of the image for every pixel size which is an integer multiple of the pixel size in the original image, until a single pixel covers almost the entire image. This approach has been also used in other image processing algorithms [3].

Therefore, all of the different applications that we described above lead to the same algorithmic problem; let $\mathcal{G}$ be a raster that consists of $\sqrt{N} \times \sqrt{N}$ cells. For every integer $\mu \in \{2, 3, \ldots, \sqrt{N}\}$ we want to compute a raster $\mathcal{G}_\mu$ of $\lceil \sqrt{N}/\mu \rceil \times \lceil \sqrt{N}/\mu \rceil$ cells where each cell of $\mathcal{G}_\mu$ stores the average of the values of the $\mu \times \mu$ cells of $\mathcal{G}$ that cover the same region.

External Memory Algorithms As already mentioned, today many available raster datasets are massive, and may consist of terabytes of data. A raster of this size cannot fit entirely in the main memory of a normal computer; thus, it can only be stored entirely in the hard disk. When we want to process the dataset, we have to transfer blocks of data from the disk to the main memory. We call such a block transfer an I/O-operation, or an I/O for short. Unfortunately, an I/O can take the same time as a million CPU operations. Thus, when designing an algorithm that may process such a large dataset, we want to minimise the number of block transfers that are required to process the full dataset.

For this reason, Aggarwal and Vitter [1] introduced a computational model that takes into account the number of block transfers between the disk and the main memory. This model considers two important parameters: the size of the internal memory M, and the maximum size B of a block of data that we can transfer from/to the disk. The efficiency of an algorithm in this model is equal to the number of I/Os that the algorithm requires during its execution. We call this concept of efficiency the *I/O-efficiency* of the algorithm. The I/O-efficiency of an algorithm is expressed as a function of the input size N, but also of the block size B and memory size M. To scan a set of N records stored in the disk we need $O(\text{scan}(N))$ I/Os, where $\text{scan}(N) = N/B$. To sort a set of N records we need $O(\text{sort}(N))$ I/Os, where $\text{sort}(N) = N/B \log_{M/B} N/B$.

Today computers contain several layers of memory; these include layers of cache that are utilised between the main memory of the computer and the processor. In this context, the values of parameters M and B differ for every pair of consecutive layers of cache that we consider. Then, to minimise the number of block transfers between all layers, the algorithm must be designed so that it achieves an optimal I/O-performance without knowing the parameters M and B. The external memory algorithms that have this property are known as *cache-oblivious* algorithms [5].

Previous Results For the problem of computing multiple resolution instances of a given raster, we consider the case where the raster does not fit in the main memory of the computer. We want to design an external memory algorithm for this problem that has an optimal performance both in terms of I/Os and in terms of CPU operations. In a previous paper, Arge *et al.* [2] proposed two external memory algorithms for this problem; the first algorithm requires $O(\text{sort}(N))$ I/Os and $O(N \log N)$ CPU time, and is very easy to implement. Their second algorithm requires $O(\text{scan}(N))$ I/Os and $O(N)$ CPU time, which is obviously optimal. However, this algorithm assumes that M is at least $\Theta(B^{1+\varepsilon})$ for some selected $\varepsilon > 0$. This algorithm is *cache-aware*, which means that M and B should be known to the algorithm to achieve this performance. Moreover, this algorithm has a strong limitation when it comes to its implementation; it requires that $\Theta(B)$ files are maintained open simultaneously during its execution. Nowadays, B can be in practice as large as a few million units, while most operating systems allow that only a relatively small number of files can be open at the same time (usually around a thousand).

Our Results In this paper we present a new, cache-oblivious algorithm that achieves the optimal performance of $O(\text{scan}(N))$ I/Os and $O(N)$ CPU time, without making any assumptions on the size of the main memory; that is it performs $O(\text{scan}(N))$ I/Os even when $M = O(B)$. The new algorithm is very easy to implement; we have developed a purely cache-oblivious implementation of the algorithm, and we have tested its performance against an implementation of the algorithm of Arge *et al.* that requires $O(\text{sort}(N))$ I/Os. Recall that the $O(\text{scan}(N))$ algorithm of Arge *et al.* is not practically implementable due to limitations of today's operating systems. The new algorithm performs extremely well and, as expected, clearly outperforms the older approach. We consider this to be a solid proof that non-trivial cache-oblivious algorithms can be implemented to perform efficiently in practice, and be used in real-world applications in the place of standard cache-aware implementations.

2 Description of the Algorithm

Preliminaries For a raster $\mathcal{G}$ we denote by $\mathcal{G}[i, j]$ the 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 assigned to this cell. We use $|\mathcal{G}|$ to indicate the number of cells of this raster. We assume that $\mathcal{G}$ is a square; it consists of $\sqrt{N}$ rows and $\sqrt{N}$ columns of cells. Yet, it is easy to show that our analysis holds also for rasters that do not have an equal number of rows and columns. Given a cell $\mathcal{G}[i, j]$ of $\mathcal{G}$, consider the set of cells $\mathcal{G}[k, l]$ for

which it holds that $1 \leq k \leq i$ and $1 \leq l \leq j$. We denote the sum of the values of these cells by $\text{psum}(i, j)$, that is:

$$\text{psum}(i, j) = \sum_{\substack{1 \leq k \leq i \\ 1 \leq l \leq j}} v(k, l) \ .$$

The value $\text{psum}(i, j)$ is the so-called *prefix sum* of cell $\mathcal{G}[i, j]$.

Let $\mathcal{G}$ be a raster 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 raster of dimensions $\lceil \sqrt{N}/\mu \rceil \times \lceil \sqrt{N}/\mu \rceil$ such that for any cell $\mathcal{G}_\mu[i, j]$ the value $v_\mu[i, j]$ associated with this cell is equal to the average value of all cells $\mathcal{G}[k, l]$ for which we have that $(i-1)\mu + 1 \leq k \leq i\mu$ and $(j-1)\mu + 1 \leq l \leq j\mu$. We say that $\mathcal{G}_\mu$ is the *scale instance* of $\mathcal{G}$ at μ, and we call μ the *scale* of this instance.

Considering the size of a scale instance $\mathcal{G}_\mu$, we observe that as we increase μ the number of cells of $\mathcal{G}_\mu$ decreases quadratically. In fact, Arge *et al.* showed that the total size of all scale instances $\mathcal{G}_\mu$ is $\Theta(N)$. We retrieve the following lemma from their paper.

Lemma 1. *Given a raster $\mathcal{G}$ of $\sqrt{N} \times \sqrt{N}$ cells, the total number of cells for all rasters $\mathcal{G}_\mu$ with $2 \leq \mu \leq \sqrt{N}$ is less than $0.65 \cdot N$.*

Proof. The total number of cells for all rasters $\mathcal{G}_\mu$ is:

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

where $\zeta(x)$ is the so-called *Riemann zeta function* [6]. The value of this function is a constant for every $x > 1$. For $x = 2$ we have that $\zeta(2) \leq 1.65$. □

Let M be a 2D matrix whose entries are real numbers. We denote by $M(i, j)$ the value of the entry that appears in the i-th row and j-th column of this matrix. We denote the number of entries of this matrix by $|M|$.

2.1 A Solution Based on Prefix Sums

In the rest of this section we describe our new cache-oblivious approach for computing all scale instances of a raster $\mathcal{G}$. To describe this new approach, we first present some concepts used by Arge *et al.* [2]. For any scale instance $\mathcal{G}_\mu$ of a raster $\mathcal{G}$, Arge *et al.* observed that we can express the value of a cell $\mathcal{G}_\mu[i, j]$ using the prefix sums of the cells of $\mathcal{G}$ as $v_\mu[i, j] = \frac{Sum(i,j,\mu)}{\mu^2}$, where:

$$\begin{aligned} Sum(i, j, \mu) &= \text{psum}(i\mu, j\mu) - \text{psum}(i\mu, (j-1)\mu) \\ &- \text{psum}((i-1)\mu, j\mu) + \text{psum}((i-1)\mu, (j-1)\mu) \ . \end{aligned}$$

Hence, to compute $\mathcal{G}_\mu$ we only need to extract the prefix sums from all cells $\mathcal{G}[i', j']$ of $\mathcal{G}$ such that *both* i' and j' are integer multiples of μ. It is easy to compute all rasters $\mathcal{G}_\mu$ if $\mathcal{G}$ fits in the main memory; first we compute a matrix that has

$\sqrt{N} \times \sqrt{N}$ entries, and which stores the prefix sums for all cells in $\mathcal{G}$. Then we can compute the value of each cell of $\mathcal{G}_\mu$ in constant time, with only four random accesses to the entries of this matrix. Since the total number of cells of all rasters $\mathcal{G}_\mu$ is $\Theta(N)$, this approach leads to an internal memory algorithm that runs in $\Theta(N)$ CPU operations. However, it is not straightforward how to compute the rasters $\mathcal{G}_\mu$ if $\mathcal{G}$ does not fit in the main memory. To solve this problem we provide the following definitions.

Let M_1 denote the 2-dimensional matrix of $\sqrt{N} \times \sqrt{N}$ entries, such that for every entry $M_1(i,j)$ of this matrix we have that $M_1(i,j) = \text{psum}(i,j)$. For any $\mu \in \{2, 3, \ldots, \sqrt{N}\}$, let M_μ be the matrix that has $\lceil \sqrt{N}/\mu \rceil \times \lceil \sqrt{N}/\mu \rceil$ entries, where $M_\mu(i,j) = M_1(i\mu, j\mu)$. Thus, M_μ stores all the prefix sums that are needed for constructing $\mathcal{G}_\mu$; the value of each cell $\mathcal{G}_\mu(i,j)$ is equal to $v_\mu[i,j] = \frac{Sum_\mu(i,j)}{\mu^2}$, where:

$$Sum_\mu(i,j) = M_\mu[i,j] - M_\mu[i,j-1] - M_\mu[i-1,j] + M_\mu[i-1,j-1] \ .$$

Therefore, assume that we already had an efficient algorithm for computing all matrices M_μ. Then, we can extract from these matrices all scale instances $\mathcal{G}_\mu$ I/O-efficiently, in only $O(\text{scan}(N))$ I/Os and $\Theta(N)$ CPU operations by simply scanning each matrix M_μ, and maintaining four pointers to access the prefix sums needed for computing each value $v_\mu(i,j)$.

Hence, we now focus on designing an efficient algorithm for computing matrices M_μ for every $\mu \in \{2, 3, \ldots, \sqrt{N}\}$. It is easy to compute M_1; we can do this by scanning $\mathcal{G}$, starting from $\mathcal{G}[1,1]$ and visiting all cells in increasing order of their row and column indices. To compute a matrix M_μ with $\mu > 1$, we could scan M_1 and extract each entry $M_1(i,j)$ such that both i and j are multiples of μ. However, in this manner we spend $O(\text{scan}(N))$ I/Os to extract each matrix M_μ, leading to $O(\sqrt{N} \cdot \text{scan}(N))$ I/Os for extracting all of these matrices.

To speed up the computation of the matrices M_μ, we can exploit the following property; consider two distinct integers ρ and λ such that $\rho, \lambda \in \{2, 3, \ldots, \sqrt{N}\}$, and $\rho = \nu\lambda$, for some $\nu \in \mathbb{N}$, $\nu > 1$. Then it holds that $M_\rho(i,j) = M_\lambda(i\nu, j\nu)$ for every entry $M_\rho(i,j)$ of matrix M_ρ. In other words, the entries of matrix M_ρ are a subset of the entries of M_λ if ρ is divisible by λ. Thus, we can construct M_ρ by processing a matrix that can be much smaller than M_1. To construct M_ρ faster, we want to use the smallest matrix M_λ for which ρ is a multiple of λ; we must find the largest $\lambda < \rho$ which is a divisor of ρ. We call this number the *largest distinct divisor* of ρ, and we denote it by $\text{ldd}(\rho)$. Consider two matrices M_ρ and M_λ such that $\rho, \lambda \in \{1, 2, \ldots, \sqrt{N}\}$, and $\rho = \text{ldd}(\lambda)$. We say that matrix M_ρ *derives* from matrix M_λ, and that M_ρ is a *derived matrix* of M_λ. In a similar manner, we say that scale instance $\mathcal{G}_\rho$ derives from instance $\mathcal{G}_\lambda$. For a matrix M_μ we denote the set of matrices that derive from M_μ by $\mathcal{D}_\mu$, that is:

$$\mathcal{D}_\mu = \{M_\rho : \rho \in \{2, 3, \ldots, \sqrt{N}\} \text{ and } \rho = \text{ldd}(\mu)\} \ . \tag{1}$$

To compute matrices M_μ, we first scan $\mathcal{G}$ to construct matrix M_1 that stores all prefix sums. Then, we extract all matrices $\mathcal{D}_1$ that derive from M_1; these are

the matrices M_μ such that μ is a prime $\leq \sqrt{N}$. To do this, we use a function *ExtractDerived*(M_μ); the input of this function is a prefix sum matrix M_μ, and the output is the set of the matrices that derive from M_μ. We describe later in more detail how this function works. After constructing matrices $M_\mu \in \mathcal{D}_1$, we apply again function *ExtractDerived* on these matrices to extract all sets of matrices $\mathcal{D}_\mu$. We continue this process recursively, until we have computed all matrices M_μ for the values $\mu \in \{2, 3, \ldots, \sqrt{N}\}$. We call the algorithm that we just described for computing all the scale instances of $\mathcal{G}$ as *MultirasterSpeedUp*.

It is easy to prove that *MultirasterSpeedUp* computes the scale instances of $\mathcal{G}$ correctly, assuming that function *ExtractDerived*(M_μ) computes correctly the derived matrices of any given M_μ. Also, by Lemma 1, excluding the performance of *ExtractDerived*, the rest of the algorithm requires only $O(\text{scan}(N))$ I/Os and $\Theta(N)$ CPU operations. Next we present in detail how we can implement function *ExtractDerived*.

2.2 Extracting the Derived Matrices

To compute the matrices $\mathcal{D}_\mu$ that derive from a given matrix M_μ, we first have to compute all scale values ρ such that M_ρ is a matrix that derives from M_μ. We call these values the *derived indices* of μ. We denote the set of these values by $\mathcal{S}_\mu$. Given μ, we can calculate all derived scales $\mathcal{S}_\mu$ using the following observation; let μ, ρ be two natural numbers such that $\mu = \text{ldd}(\rho)$. Then it holds that $\mu = \rho/\text{spd}(\rho)$, where $\text{spd}(\rho)$ is the *smallest prime divisor* of ρ. Since μ is the largest distinct divisor of ρ we also have that $\text{spd}(\rho) \leq \text{spd}(\mu)$. Based on the above, to compute $\mathcal{S}_\mu$ we first compute $\text{spd}(\mu)$; we go through all integers $\kappa \in \{2, \ldots, \lceil\sqrt{\mu}\rceil\}$ in increasing order, and we stop when we find the first κ that divides μ. Then we compute all prime numbers in the range $[2, \text{spd}(\mu)]$ by trivially trying all possible pairs of integers within this range, and checking if the largest of the two is divided by the smallest. For the special case $\mu = 1$ the smallest prime divisor is undefined, and we consider that $\mathcal{S}_\mu$ consists of all prime numbers smaller than $\sqrt{N}$. Thus, for $\mu > 1$ we can compute scale values $\mathcal{S}_\mu$ in $O(\mu)$ CPU operations. We need at most $O(\text{scan}(\mu))$ I/Os to store these values. For $\mu = 1$ this process requires $O(N)$ CPU operations and $O(\text{scan}(N))$ I/Os.

To extract the derived matrices $\mathcal{D}_\mu$, we will use M_μ to construct an intermediate file F_μ that contains altogether the entries of all matrices in $\mathcal{D}_\mu$. We will then process this file to extract each derived matrix I/O-efficiently. More specifically, file F_μ is organised as follows; for every prime $\rho \in \mathcal{S}_\mu$, and for every entry $M_\rho(i, j) \in M_\rho$, F_μ contains a record of the form: $< i\rho, j\rho, \rho, v_\mu(i\rho, j\rho) >$. The two first fields of the record indicate which is the entry in M_μ that has the same value as $M_\rho(i, j)$. The third field indicates the scale of M_ρ, and the last field carries the value $M_\rho(i, j)$. Most importantly, the records in F_μ appear in lexicographical order of their three first fields.

Thus, F_μ stores a record for each entry of the matrices in $\mathcal{D}_\mu$, including multiples. The number of records in F_μ is $O(|M_\mu|)$; the number of entries of M_μ is $|\mathcal{G}_\mu|$, and due to Lemma 1 the total number of cells of all the scale instances of a raster $\mathcal{G}_\mu$ cannot exceed $|\mathcal{G}_\mu|$. To construct F_μ, we create an indi-

vidual file $F_{\mu,\kappa}$ for each matrix $M_\kappa \in \mathcal{D}_\mu$. File $F_{\mu,\kappa}$ contains only records of the form $\{i\kappa, j\kappa, \kappa, \otimes\}$, where $\otimes$ is a symbolic "no-data" value. Then we merge all those files into F_μ in a bottom-up manner; first we generate F_μ by merging the two files $F_{\mu,\kappa}$ and $F_{\mu,\rho}$ that correspond to the two smallest matrices M_ρ and M_κ in $\mathcal{D}_\mu$; that is ρ, κ are the two largest values in $\mathcal{S}_\mu$. We go on merging F_μ each time with the smallest remaining file $F_{\mu,\lambda}$, until all files are merged into F_μ.

Next we fill in the prefix sum values at the last field of each record in F_μ with a single simultaneous scan of F_μ and M_μ. To extract matrices $\mathcal{D}_\mu$ from F_μ we scan F_μ once per matrix in $\mathcal{D}_\mu$. The matrices are extracted in order of decreasing size; in the first scan of F_μ we extract the largest matrix $M_\rho \in \mathcal{D}_\mu$, and so on and so forth. To extract M_ρ, we pick the records in F_μ whose third field is equal to ρ. We then throw away these records from F_μ, creating a new smaller instance of F_μ. When F_μ becomes empty we will have extracted all derived matrices in $\mathcal{D}_\mu$. The correctness of the algorithm follows from how we handle the prefix sum values in the records of file F_μ. Next we prove the efficiency of this algorithm.

Lemma 2. *Function ExtractDerived computes the set of matrices $\mathcal{D}_\mu$ that derive from M_μ in $O(\mathrm{scan}(|M_\mu| + \mu))$ I/Os and $O(|M_\mu| + \mu)$ CPU operations.*

Proof. We showed that for $\mu \geq 1$ computing the scales $\mathcal{S}_\mu$ takes $O(\mathrm{scan}(\mu))$ I/Os and $O(\mu)$ CPU time. Recall that for the case $\mu = 1$, we can compute $\mathcal{S}_\mu$ in $O(\mathrm{scan}(N))$ I/Os and $O(N)$ CPU operations. Now we prove that for any $\mu > 1$ we can construct all matrices $\mathcal{D}_\mu$ in $O(\mathrm{scan}(|M_\mu|))$ I/Os and $O(|M_\mu|)$ CPU operations. To construct file F_μ, we merge several smaller files $F_{\mu,\rho}$, one merge at a time. As soon as file $F_{\mu,\rho}$ gets merged with F_μ the records of $F_{\mu,\rho}$ become a part of F_μ; from this point and on, these records are scanned once each time we merge F_μ with another file $F_{\mu,\kappa}$. Hence, each record that initially belonged to file $F_{\mu,\rho}$ gets scanned as many times as the number of primes that are smaller or equal to ρ; this is because $\mathcal{S}_\mu$ contains all primes in the range $[2, \mathrm{spd}(\mu)]$, and because we merge files $F_{\mu,\kappa}$ in decreasing order of κ. In the mathematical literature, the number of primes that are smaller or equal to ρ is denoted by $\pi(\rho)$. As each record of $F_{\mu,\rho}$ is scanned $\pi(\rho)$ times, and as $F_{\mu,\rho}$ has $|M_{\mu\rho}|$ records, the total number of records scanned when constructing F_μ is:

$$\sum_{\rho \in \mathcal{S}_\mu} \pi(\rho)|M_{\mu\rho}| = \frac{N}{\mu^2} \sum_{\rho \in \mathcal{S}_\mu} \frac{\pi(\rho)}{\rho^2} \ . \tag{2}$$

The following upper bound is known for $\pi(\rho)$ [7]: $\pi(\rho) < \frac{1.26\rho}{\ln \rho}$. Combining this with (2) we get:

$$\frac{N}{\mu^2} \sum_{\rho \in \mathcal{S}_\mu} \frac{\pi(\rho)}{\rho^2} < 1.26 \frac{N}{\mu^2} \sum_{\rho \in \mathcal{S}_\mu} \frac{1}{\rho \ln \rho} = \frac{1.26}{\log e} \frac{N}{\mu^2} \sum_{\rho \in \mathcal{S}_\mu} \frac{1}{\rho \log \rho} \ , \tag{3}$$

where e is the base of the natural logarithm. We have that:

$$\sum_{\rho \in \mathcal{S}_\mu} \frac{1}{\rho \log \rho} \leq \sum_{i=0}^{\infty} \sum_{\substack{\rho \text{ is prime} \\ 2^{2^i} \leq \rho \leq 2^{2^{i+1}}}} \frac{1}{\rho \log \rho} \leq \sum_{i=0}^{\infty} \sum_{\substack{\rho \text{ is prime} \\ 2^{2^i} \leq \rho \leq 2^{2^{i+1}}}} \frac{1}{2^i \rho} \ . \tag{4}$$

The following bound is known in the mathematical literature [7]:

$$\sum_{\substack{\rho \text{ is prime} \\ \rho \le x}} \frac{1}{\rho} = O(\log\log x) \ . \tag{5}$$

From (4) and (5), we get:

$$\sum_{i=0}^{\infty} \sum_{\substack{\rho \text{ is prime} \\ 2^{2^i} \le \rho \le 2^{2^{i+1}}}} \frac{1}{2^i \rho} = O\left(\sum_{i=0}^{\infty} \frac{i+1}{2^i} \right) = O(1) \ .$$

Combining (3) with (5) we conclude that the total number of records that we need to scan in order to construct F_μ is $O(|M_\mu|)$. This requires $O(\text{scan}(|M_\mu|))$ I/Os. During the merging we do one comparison for every record that we scan, which implies that we do $O(|M_\mu|)$ operations in the CPU in total.

It remains now to show that extracting all matrices of $\mathcal{D}_\mu$ from F_μ requires $O(\text{scan}(|M_\mu|))$ I/Os and $O(|M_\mu|)$ time in the CPU. Recall that we extract the matrices M_ρ in increasing order of ρ, hence, the records of $M_{\mu\sigma}$ will get scanned as many as $\pi(\sigma)$ times each. Therefore the records scanned in this part of the algorithm are as many as the records scanned for constructing F_μ. We showed that this number is equal to $O(|M_\mu|)$, implying $O(\text{scan}(|M_\mu|))$ I/Os and $O(|M_\mu|)$ CPU operations for extracting the matrices for F_μ, and the lemma follows. □

By construction our algorithm does not require knowledge of M and B, hence it is cache-oblivious. Also, its performance does not depend on a lower bound on the size of M. We obtain the following theorem.

Theorem 1. *Given a raster $\mathcal{G}$ of $\sqrt{N} \times \sqrt{N}$ cells, we can compute all scale instances of $\mathcal{G}$ cache-obliviously in $O(\text{scan}(N))$ I/Os and $O(N)$ CPU operations.*

Proof. Function *ExtractDerived* is called only once for each matrix M_μ so, according to Lemma 2, the total number of I/Os and CPU operations required by the entire algorithm is $O(\text{scan}(\sum_\mu(|M_\mu| + \mu)))$ and $O(\sum_\mu(|M_\mu| + \mu))$ respectively. Since M_μ has the same size as $\mathcal{G}_\mu$, then according to Lemma 1 and because $\sum_\mu |M_\mu| = \Theta(N)$, the theorem follows. □

2.3 Ordering the Prefix Sum Matrices

So far, we have described an algorithm that computes efficiently all scale instances of a given raster $\mathcal{G}$. However, this algorithm does not output the scale instances of $\mathcal{G}$ in the right order. More specifically, from the description of algorithm *MultirasterSpeedUp* we can see that there can be pairs of scale instances $\mathcal{G}_\mu$ and $\mathcal{G}_\rho$ with $\mu < \rho$ such that $\mathcal{G}_\rho$ appears in the output before $\mathcal{G}_\mu$. Yet, for most practical applications, it makes sense to have those instances sorted in the output in order of increasing scale value. Fortunately, we can solve this problem while achieving the same performance as with the algorithm *MultirasterSpeedUp*. The proof of the next theorem appears in the full version of the paper.

Theorem 2. *Given a raster $\mathcal{G}$ of $\sqrt{N} \times \sqrt{N}$ cells, we can compute cache-obliviously all scale instances of $\mathcal{G}$, and output these instances in order of increasing scale using $O(\text{scan}(N))$ I/Os and $O(N)$ CPU operations.*

2.4 Improving the practical performance of the algorithm

Earlier in this section, we described how we can extract the prefix sum matrices $\mathcal{D}_\mu$ from a matrix M_μ by building an intermediate file F_μ. This approach of building F_μ requires merging several smaller files, and needs only $O(\text{scan}(|M_\mu|))$ I/Os. Yet we can evade this merging process, and thus improve the practical I/O-performance of the entire algorithm by a constant factor; to build F_μ, we can scan M_μ and stream the records that correspond to the entries of matrices in $\mathcal{D}_\mu$ in the form of queries to M_μ. As soon as we extract the prefix sum value of a queried record, we append this record in F_μ. To do this properly, the records should be streamed to M_μ in lexicographical order of their three first fields. To produce the stream of the ordered records we build a min-heap structure. Each leaf node $\nu[\rho]$ of the heap corresponds to a derived matrix $M_{\mu\rho} \in \mathcal{D}_\mu$ and stores the next record of $M_{\mu\rho}$ that has to be streamed. The root of the heap stores the next record to be queried to M_μ. Figure 1 illustrates the structure of the heap; we can see that the heap is as skewed as it can get in favour of the larger derived matrices. The heap contains one leaf node for each derived matrix of M_μ, so the size of the heap is $O(\text{spd}(\mu))$. Although we do not know M, we can build the heap so that at any point the nodes of the $O(M)$ topmost levels appear in memory. For the rest of the levels, a record will have to pay one I/O for every B levels that it goes up in the heap. Although this method is oblivious of M, we show that we can stream all records to M_μ so that the number of I/Os decreases as M increases. The proof of the following lemma is provided in the full version of the paper.

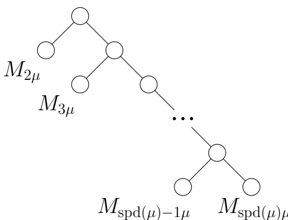

Fig. 1. The structure of the skewed heap that we use to stream the records. With each node we indicate that derived matrix that corresponds to this node.

Lemma 3. *Let M_μ be a prefix sum matrix. We can stream all the records that correspond to the entries of the derived matrices of $\mathcal{D}_\mu$ in lexicographical order in $O(\text{scan}(|M_\mu|/\log M))$ I/Os and $O(|M_\mu|)$ CPU operations.*

3 Implementation & Benchmarks

We have implemented *MultirasterSpeedUp* and evaluated its efficiency on input datasets of various sizes. In the experiments that we conducted, we tried several alternatives in the way that we implemented the most important routines of the algorithm, and we assessed the efficiency of the implementation for each one of these alternatives. We also compared the performance of our implementation with an older implementation of the $O(\text{sort}(N))$ algorithm of Arge *et al.* . Recall that it is not currently possible to implement the $O(\text{scan}(N))$ algorithm of Arge *et al.* due to restrictions in standard operating systems; this algorithm requires that B files are open simultaneously, and while B today is usually in the order of millions of units, standard operating systems allow for about a thousand

files open at the same time. Next we describe in detail the settings that we used for our experiments.

To measure the performance of our algorithm we used massive raster datasets of many sizes. The datasets that we used originate from a massive raster that consists of roughly 26 billion cells, arranged in 146974 rows and 176121 columns. This raster models the terrain surface over the entire region of Denmark. Each cell of the raster represents a square region on the terrain that has dimension of 2 meters. The elevation of each cell is stored as a 4-byte floating point number, and the entire dataset is stored in a geotif file that has 97 gigabytes size. From this dataset, we constructed all scale instances $\mathcal{G}_\mu$ for $\mu \leq 146974$, and we used the largest of these instances as input for the algorithm; we did this to evaluate the performance of the algorithm for a large range of different input sizes.

As already mentioned, we tried different options for implementing the key routines of the algorithm. These are the routines that involve merging or extracting a sequence of files from/to another larger file. For those routines we evaluated how the performance of the algorithm is affected when trying to merge/extract several files simultaneously. More specifically, the routines that we tweaked are the following:

- The part of *ExtractDerived* where, given a prefix sum matrix M_μ, we merge several files to construct an intermediate file F_μ which contains the records that correspond to all the entries of the derived matrices $\mathcal{D}_\mu$.
- The part of *ExtractDerived* where we extract the derived matrices $\mathcal{D}_\mu$ from the intermediate F_μ.

For the above routines we measured how the performance of the algorithm changes if we change the number of files that are merged or extracted together. For the first routine we use f_1 to denote the number of files that we merged simultaneously at each point for constructing F_μ. For the second routine we use f_2 to denote the number of derived matrices that we extracted together each time that we performed a scan of F_μ. In the description of the algorithm, we convey that the value of each of these two parameters is equal to two.

We also implemented a version of the routine that constructs the intermediate file F_μ based on the mechanism of the skewed heap that is described in Section 2.4. Recall that this method does not involve merging any files in order to construct F_μ. All of the variants of our implementation work in a purely cache-oblivious manner.

The algorithms were implemented in C++ using the software library TPIE (the *Templated Portable I/O Environment*) [8]. This library offers I/O-efficient algorithms for scanning and sorting large files in external memory. Our experiments where run on a machine with a 3.2GHz four-core Xeon CPU (W3565). The main memory of the computer is 12GB. This workstation has 20 disks that have a btrfs (raid 0) file system configuration. The operating system on this computer was Linux version 2.6.38. During our experiments, 8GB of memory was managed by our software, and the rest was left to the operating system for disk cache. For each of the versions of our implementation, the maximum amount of disk space used at any time during the execution was 672 GB.

In our first experiment, we ran our implementation of the algorithm on the 97GB dataset for all possible combinations of values of the two parameters $f_1, f_2 \in \{2, 3, 10, 20, 35, 50\}$. We also ran the implementation of the algorithm using the skewed heap approach for all values of parameter $f_2 \in \{2, 3, 10, 20, 40, 50\}$. From all the possible versions that we tried, the best running time was achieved by the version that uses the skewed heap approach, and parameter value $f_2 = 50$; the running time in this case was 2 hours and 15 minutes. The best running time that we got without using the skewed heap approach was for the version with parameter values $f_1 = f_2 = 50$. In this case, the running time was 2 hours and 28 minutes. The worst running time that we got among all versions was from the version that has parameter values $f_1 = 2$, and $f_2 = 2$; the running time for this version was 3 hours and 35 minutes. In general, the running time of each version that behaved like a decreasing function on the values of paramaters f_1 and f_2. Running the implementation of the $O(\text{sort}(N))$ algorithm of Arge *et al.* on the largest dataset yielded a running time of 13 hours and 14 minutes. This running time is a bit less than four times larger than the worst running time that we got for any version of our implementation. For our next experiment, we ran the two best versions of our implementation on the datasets that we got from extracting the 100 largest scale instances of the 97GB raster, including the initial raster itself. We also ran on these datasets the implementation of the $O(\text{sort}(N))$ algorithm of Arge *et al.*, and the naive internal-memory algorithm that uses prefix sums. Figure 2 illustrates the performance of the four implementations. There, we get a good impression on how the performance of our implementation scales with the size of the input. This is a strong indication that the theoretical bounds that we proved for the performance of the algorithm can be reflected in practice. The results of both experiments show evidently the practical efficiency of our algorithm, when also compared to the implementation of the algorithm of Arge *et al.*. Of course, it could be argued here that this result is hardly surprising; in theory, an $O(\text{sort}(N))$ algorithm has obviously worse asymptotical behaviour than an $O(\text{scan}(N))$ algorithm. However, in practice, the performance of an $O(\text{sort}(N))$ algorithm scales linearly in terms of I/Os. Figure 2 provides some evidence on this argument for the algorithm of Arge *et al.*, at least for the range of input sizes that we considered. The explanation behind this phenomenon is that the ratio M/B in most computers has a value close to one thou-

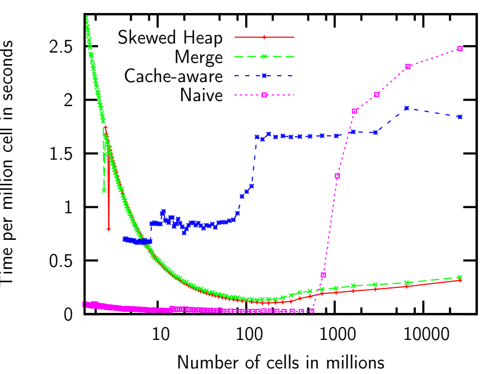

Fig. 2. The performance of the two best versions of our implementation, together with the implementation of the $O(\text{sort}(N))$ algorithm of Arge *et al.* , and the naive internal memory algorithm. The x-axis shows the input sizes using a logarithmic scale with base 10. The y-axis shows running times divided by input size.

sand, and therefore the term $\log_{M/B}(N/B)$ in $\text{sort}(N)$ is not larger than two in all practical cases. Thus, it is not unrealistic to observe $O(\text{sort}(N))$ algorithms performing better in practice than $O(\text{scan}(N))$ algorithms. More than that, in our case, we compare a cache-aware implementation with a cache-oblivious one, and we could expect that this is an advantage for the performance of the cache-aware implementation. Yet, as we see from our experiments, this is clearly not the case; the cache-oblivious algorithm performs much better in practice. This result shows that purely cache-oblivious software can be developed to perform efficiently in real-world applications. It will be interesting to see if similar results can be obtained for other external memory problems as well.

In our last experiment, we ran the best version of our implementation on the largest of our datasets, and at every minute of the execution we measured the rate of the CPU utilisation and the I/O-throughput of this implementation. Figure 3 illustrates the results of this experiment. We see that both the I/O-throughput and CPU utilisation were fairly constant during the run. Also, for the largest part of the execution of the algorithm, the CPU utilisation remained above or close to 40%; hence, the running time of the algorithm was almost equally distributed between the CPU and the I/O-operations.

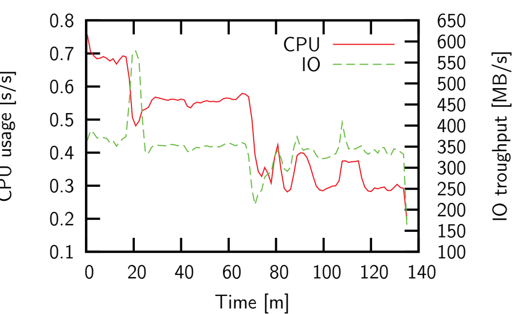

Fig. 3. The CPU utilisation and I/O-throughput of the best version of our implementation.

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, H. Haverkort and C. Tsirogiannis. Fast Generation of Multiple Resolution Instances of Raster Data Sets. In *Proc. 20th ACM SIGSPATIAL Int. Conference on Advances in Geographic Information Systems (ACM GIS)*, pages 52–60, 2012.
3. P.K. Bøcher and K.R. McCloy. The Fundamentals of Average Local Variance - Part I: Detecting Regular Patterns. *IEEE Trans. on Image Processing*, 15:300–310, 2006.
4. P. Fisher, J. Wood, and T. Cheng. Where is Helvellyn? Fuzziness of Multiscale Landscape Morphometry. *Trans. Inst. of British Geographers*, 29(1):106–128, 2004.
5. M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms*, 8(1):1–22, 2012.
6. A.A. Karatsuba and S.M. Voronin. *The Riemann Zeta-Function*. Walter de Gruyter, Berlin, 1992.
7. J.B. Rosser and L. Schoenfeld. Approximate Formulas for Some Functions of Prime Numbers. *Illinois Journal of Mathematics*. 6(1):64–94, 1962.
8. TPIE, the Templated Portable I/O Environment. `http://www.madalgo.au.dk/tpie/`.
9. C.E. Woodcock and A.H. Strahler. The Factor of Scale in Remote Sensing. *Remote Sensing of Environment*, 21:311–332, 1987.