

Map algebra on raster datasets represented by compact data structures

Fernando Silva-Coira^{ORCID} | José R. Paramá^{ORCID} | Susana Ladra^{ORCID}

Universidade da Coruña, Centro de Investigación CITIC, Facultade de Informática, A Coruña, Spain

Correspondence

Fernando Silva-Coira, Facultade de Informática, Universidade da Coruña, CITIC, Campus de Elviña, s/n, 15071 A Coruña, Spain.

Email: fernando.silva@udc.es

Funding information

CITIC; Ministerio de Ciencia e Innovación, Grant/Award Numbers: PID2020-114635RB-I00, PDC2021-120917-C21, PDC2021-121239-C31, PID2019-105221RB-C41, TED2021-129245-C21; Xunta de Galicia, Grant/Award Numbers: ED431C 2021/53, IN852D 2021/3 (CO3)

Abstract

The increase in the size of data repositories has forced the design of new computing paradigms to be able to process large volumes of data in a reasonable amount of time. One of them is in-memory computing, which advocates storing all the data in main memory to avoid the disk I/O bottleneck. Compression is one of the key technologies for this approach. For raster data, a compact data structure, called k^2 -raster, have been recently been proposed. It compresses raster maps while still supporting fast retrieval of a given datum or a portion of the data directly from the compressed data. k^2 -raster's original work introduced several queries in which it was superior to competitors. However, to be used as the basis of an in-memory system for raster data, it is mandatory to demonstrate its efficiency when performing more complex operations such as the map algebra operators. In this work, we present the algorithms to run a set of these operators directly on k^2 -raster without a decompression procedure.

KEYWORDS

compression, map algebra, raster maps

1 | INTRODUCTION

In this work, we focus on spatial information represented with a raster model. Raster datasets represent geographic information by dividing space into a finite grid of cells and assigning a value to each cell. This includes images -such as remotely sensed imagery-, engineering, modeling, representations of parameters of the land such as pollution levels, atmospheric pressure, rain precipitations, land elevation, vegetation indices and so forth. The widespread use of different types of devices in many different scenarios causes the amount of information in raster format to increase rapidly. For example, only considering the family of satellites Sentinel, 20 TB of data are being generated each day, and the remote sensing repositories have reached the Peta Byte/Zetta Byte (PB/ZB) scale.¹

Abbreviations: BRWT, Binary Relation Wavelet Tree; DACs, Directly Addressable Codes; DTM, Digital Terrain Model; GIS, Geographical information Systems; LOUDS, Level-Ordered Unary Degree Sequence; NetCDF, Network Common Data Form; OGC, Open Geospatial Consortium; SGI, Spanish Geographic Institute.

This is an open access article under the terms of the [Creative Commons Attribution](https://creativecommons.org/licenses/by/4.0/) License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2023 The Authors. *Software: Practice and Experience* published by John Wiley & Sons Ltd.

The traditional way to process spatial data consists in using special data structures and algorithms to bring data from disk to main memory. These data structures and algorithms exploit the location of data in disk and, together with loading techniques, anticipate the loading of pages or disk blocks (pre-fetching) in a buffer in order to process the typical spatial queries (such as the range query). However, this computing paradigm is reaching its limits, and therefore new approaches are needed. One alternative is distributed computing, such as SpatialHadoop² or SpatialSpark.³ A different approach is in-memory computing,⁴ which implies to load all the data in main memory to avoid the bottleneck of the disk I/O. However, loading all the data into main memory can be difficult and, thus, compression is used to alleviate the space consumption.⁵⁻⁷

Several data compression methods for raster data have been designed.^{8,9} Real systems capable of managing raster data, like Rasdaman, Grass, or even R, as well as raster representation formats such as NetCDF (standard format of the OGC*) and GeoTiff,¹⁰ provide compressed storage based on traditional compression methods such as run-length encoding, Lempel-Ziv-Welch, or Deflate.¹¹ However, classical compression methods have an important drawback: if one wants to access a given datum or portion of the data, the complete dataset, or a large portion of it, must be decompressed.

A recent research field, called *compact data structures*,¹² searches for data structures that make it possible to decompress only the requested data, and therefore they are very suitable for in-memory computing. Moreover, most compact data structures include indexes within the same compressed space, and thus access times over the compressed data are comparable or better than the classical methods over uncompressed data.

Compact data structures have been used for representing several data types. As an illustration, compact data structures have been used to represent graphs of the World Wide Web,¹³⁻¹⁵ represent documents in the context of information retrieval,¹⁶⁻¹⁹ and also to improve query efficiency in geographical information systems (GIS).²⁰⁻²⁴

The k^2 -raster^{23,25} has been shown to be superior to other compact data structures for raster data and a serious competitor of NetCDF. In empirical experiments, k^2 -raster reached improvements up to 60% in space consumption with respect to other compact data structures, whereas it obtains compression power comparable to that of NetCDF. k^2 -raster is up to 9 times faster than other compact data structures to obtain a cell value or 40 times faster when retrieving the cells in a region with values in a range. Compared to NetCDF, k^2 -raster is orders of magnitude faster when running range queries. These results back the hypothesis of using k^2 -raster in an in-memory computing scenario. However, k^2 -raster's original work only showed four queries: get the value of an individual cell, get the values of the cells in a window region, get the cell positions in a window region with values in a range, and get the cell positions and values in a window region with values in a range. Later, it was shown that complex operations like a spatial join are also more efficient using k^2 -raster.²⁶

However, there are other operations that must be studied. *Map Algebra*^{27,28} has become the *de facto* standard operations on raster maps in real GIS. The map algebra defines a set of primitive operations and functions that have rasters as input. They make it possible to perform analysis on the data. Examples of real systems including map algebra can be the ArcGis Pro of ESRI²⁹ and GRASS.³⁰ Map algebra opens a wide variety of data analysis, such as, from a digital elevation terrain raster, "highlight the land surface below 30 meters," "obtain the residential areas where the terrain has a slope greater than 15%," or, in a city council GIS, "show the total number of crimes in each neighborhood."

Therefore, to demonstrate the feasibility of using the k^2 -raster as the basis of an in-memory system, we have to show the efficiency of k^2 -raster when performing map algebra operations. Algorithms for executing map algebra operations over compressed data using another compact data structure for rasters, called k^2 -acc,^{21,22} have been previously presented.³¹ However, as explained, k^2 -raster has been shown to be superior in all aspects to k^2 -acc. Moreover, k^2 -acc has serious difficulties in datasets with already a moderate number of different values in the raster map, as shown in k^2 -raster's original work, where the k^2 -acc was not able to run in several experiments.

In this article, we propose and evaluate a map algebra implementation where the raster data is represented using k^2 -raster.

2 | MAP ALGEBRA ON RASTER DATASETS

Map algebra defines a set of operators and functions to be applied to one or two input raster datasets to obtain a new output raster dataset.^{27,28} Together with variables, which in map algebra represent entire raster datasets, it allows the construction of expressions or equations.

*<https://www.opengeospatial.org/standards/netcdf>

Using the sequential application of these operators, it is possible to perform complex data analysis on raster datasets.³² The analysis procedures are categorized as those applied to an entire raster dataset (e.g., spatial autocorrelation) and those concentrated on specific locations: cell, neighborhood, or zones.

Depending on the neighborhood cells involved in computing the operator or function, there are four types of operators and functions³³⁻³⁵:

- (a) *Local*. The value in each cell of the output raster only depends on the values of the cells at the same location of the input datasets. Local operators are arithmetic, logical, and statistical functions (e.g., minimum, maximum, or mean).
- (b) *Focal*. In this case, the value of each output cell is computed taking into account the values of the cells at the same location of the input rasters and their neighboring cells in a specific range. Examples of focal functions include the computation of typical statistical functions (e.g., mean, mode, minimum, and maximum) corresponding to the neighborhood of a cell. Other processing such as convolution, kernel and moving windows also uses focal operations.
- (c) *Zonal*. In this type of operation, two input rasters are needed. The first one is used to divide the space into zones; then, the values of the output cells within a zone are computed processing the cells of the second input raster within that zone.
- (d) *Global*. This is the application of a bulk change to all cells. For example, adding a scalar value to all cells.

In the present study, we present the following operations. As a local operation, we show the *point-wise* arithmetic operation, see Figure 1A. In the case of global operations, we present two operations: *thresholding* obtains a binary raster where each cell of the output raster has a 1 if the value of the cell at the same position of the input raster is greater or equal than a given scalar value and a 0 otherwise (see Figure 1B), and *scalar operation*, where the output raster is the result of applying the same operation over all cells, such as adding a scalar value to all cells of the input raster (see Figure 1C). Finally, we present one zonal operation, *zonal sum*, which requires two input rasters, the *zonal matrix* and *input matrix*. The zonal matrix defines zones, which are formed by all cells having the same value. For each zone, the algorithm sums the cells of the input matrix, and then, the output is the zonal matrix, changing the original value by the sum, see Figure 1D.

3 | RELATED WORK

3.1 | Storage methods for rasters

A variety of formats and tools are available for the storage of raster data. Observe that a raster dataset can be easily stored as a matrix of values in generic matrix-processing tools, or even using raster-based image file formats. For example, GeoTIFF, a file format specifically designed for storing rasters, relies on the classical image file format TIFF enhanced with metadata to store geographical attributes of the raster.

Next, we will introduce NetCDF, possibly the most widely-used file format for the representation of raster data.

3.1.1 | NetCDF

Network Common Data Form (NetCDF)³⁶ is an Open Geospatial Consortium standard for creating, accessing, and sharing array-oriented scientific data, that is, rasters datasets.

The data format is coupled with a set of software libraries that provide the ability to query the data. It is supported by most GIS software tools that handle raster data.

The file format arranges the cells of the raster in simple N-dimensional arrays. NetCDF provides a compressed or uncompressed representation of the data. Compression is based on Deflate,³⁷ which provides ten compression levels, ranging from level 0 (no compression) to level 9 (maximum compression). The main novelty of NetCDF is that even when compression is used, the data can be transparently accessed without performing an explicit decompression, that is, the access procedure is the same whether the data is compressed or not. Therefore, a trade-off arises between time and space depending on the used compression level; the more compression is chosen, the slower the querying process and vice versa.

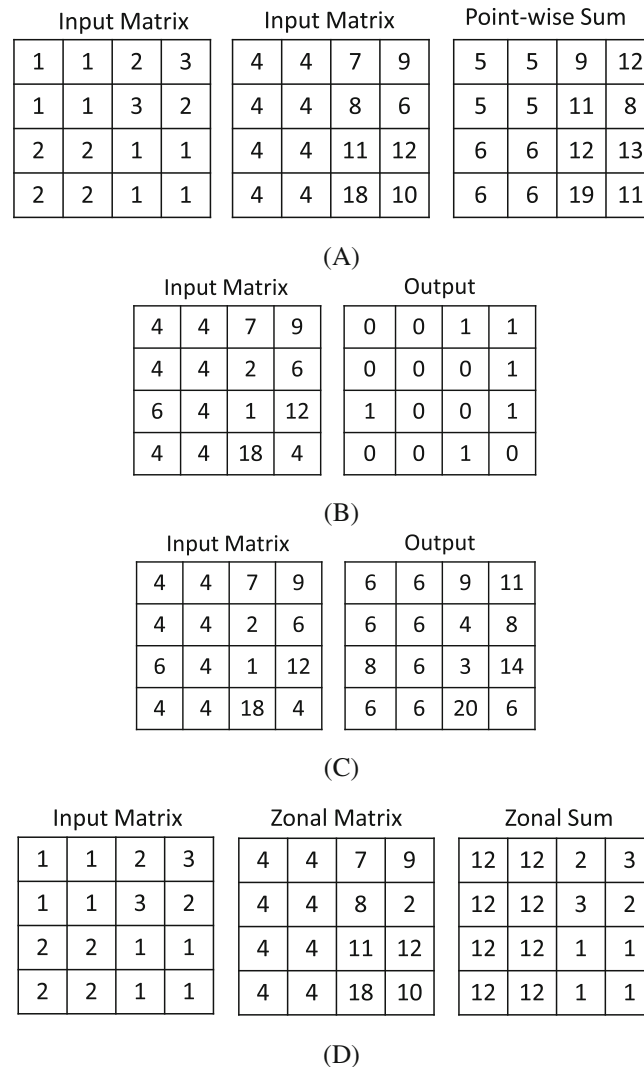


FIGURE 1 Map algebra operations. (A) Point-wise sum; (B) thresholding with threshold = 5; (C) scalar sum, adding 2; (D) zonal sum.

Moreover, queries over compressed data are efficient due to the use of a technique called *chunking*: data is compressed in blocks so that when a specific region of the data is demanded, only the relevant chunks need to be decompressed.

3.2 | Compact data structures for raster data

3.2.1 | Rank and select

Most of the compact data structures use two basic operations over bitmaps: $rank_b(B, p)$ counts the number of occurrences of bit b in bitmap B until position p and $select_b(B, n)$ returns the position of the n th occurrence of bit b in B .

These operations can be efficiently solved in constant time³⁸ using $n + o(n)$ bits of total space. In practice, only rank is solved in constant time whereas select is solved in $O(\log \log n)$ time³⁹ with approximately 5% extra space over the original bitmap.

3.2.2 | Representing rasters with k^2 -trees: k^2 -acc and k^3 -tree

The k^2 -tree was initially designed to store and query web graphs.¹⁵ However, actually, it is a region quadtree⁴⁰ for binary matrices built with the latest developments in the field of compact data structures.

Given a binary matrix M of size $n \times n$, and a parameter k , M is represented with a tree. To build that tree, M is divided into k^2 submatrices of size $n/k \times n/k$. A child node is added to the root node for each of those submatrices. Each node is labeled with one bit. A 0 indicates that the submatrix is full of 0-bits and a 1 means that the submatrix contains at least one 1-bit. Next, for each of the nodes labeled with a 1-bit, the process continues recursively.

To save space, the resulting tree is stored with two bitmaps without pointers, by following a level-wise traversal of the tree nodes: L is a bitmap formed by the bits corresponding to the last level of the tree, and T is a bitmap that contains the rest.

The k^2 -tree is limited to binary matrices. Thus, one way to represent rasters having values in the range $v_1 < v_2 < \dots < v_t$ is to use a k^2 -tree for each value. The representation is formed by t k^2 -trees K_1, K_2, \dots, K_t , where each K_i has a value 1 in the cells whose value is $v \leq v_i$ in the original raster. This approach is the k^2 -acc.^{21,22}

Another way to represent a raster is to add a third dimension to the k^2 -tree. The k^3 -tree^{21,22} stores points $\langle x, y, z \rangle$, where the first two values represent the position in the 2D space, and the third component is the value stored in that cell.

3.2.3 | k^2 -raster

While the k^2 -acc and the k^3 -tree obtained better results both in space and search performance than other ways of representing rasters, such as those based on compressing GeoTIFF images, the k^2 -raster^{23,25} has been proven superior to them, obtaining better results both in space and query times.

The k^2 -raster follows the same quadtree-like building process of the k^2 -tree. Given a matrix M of integers of size $n \times n$,[†] it builds a tree as follows. Given a parameter k , it divides M in k^2 submatrices of size $n/k \times n/k$. Each submatrix adds a child to the root node of the tree storing the maximum and minimum values in the corresponding submatrix. If the maximum and the minimum values are equal, then the node becomes a leaf; otherwise, the process continues recursively on that submatrix.

Again, this tree is compactly represented using binary bitmaps. In addition, we also have sequences of integers corresponding to the maximum and minimum values, which will be also compressed.

We show in Figure 2 an example of k^2 -raster. In the upper part, we can see the matrix and its recursive subdivision. Just below, we can see the corresponding conceptual tree. The root represents the complete 8×8 matrix, and, as seen, the root contains the maximum and minimum values in that matrix. Under the label “Step 2,” we can see the first division into four matrices of size 4×4 , each producing a child of the root node. We highlight the maximum and minimum values in each submatrix, which are stored at the children of the root of the tree. The process continues until finding a submatrix filled with the same value or until we reach individual cells, which produce nodes with only the value of the cell (last level of the tree).

To reduce the magnitude of the numbers stored at the nodes, and thus open the possibility of representing them with fewer bits, the values at the nodes are stored as the difference with the same value at the parent. This is illustrated in the tree at center-bottom of Figure 2.

The topology of the tree is represented with a bitmap. Each node with children is represented with a 1-bit, whereas the leaves in levels before the last one are represented with a 0-bit. Thus, the bitmap is the simple level-wise traversal of the conceptual tree. Indeed, this is a simplified variant of LOUDS (*level-ordered unary degree sequence*) tree representation,⁴¹ which is a compact representation for trees. In Figure 2, we can see this bitmap labeled as “Tree (T).”

Using T , we can simulate the tree navigation with *rank* and *select* operations. More precisely, given a 1-bit at position p in T , that is, a node with children, these children are sequentially located from position $children(p) = rank_1(T, p) \times k^2$ of T , unless the children are individual cells, which are not represented in T . The parent of a node at position p of T is computed as $parent(p) = select_1(T, \lfloor p/k^2 \rfloor)$. Due to the fast response time of *rank* and *select*, it is possible to efficiently access the value of a single cell, retrieve entire row/columns, or solve spatial range queries.

The values at the nodes are stored as arrays, following the same level-wise order of the tree. We need two arrays L_{max} for maximum values and L_{min} for the minimum. These arrays are compressed with *Directly Addressable Codes* (DACs),¹⁹ a compression scheme for integer sequences that provides the ability to direct access to any given position. We use the

[†]If the matrix is not a square, we can fill it with “empty cells” until it becomes a square. This does not impact the space requirements of the final representation, as the k^2 -raster is very efficient representing regions with the same value.

DACs version that calculates the optimal number of bits at each level. DACs exploit the fact that encoded differences tend to be small. This is shown at the bottom of Figure 2, as part of the final representation.

As explained, the k^2 -raster obtains a very compressed representation of the raster matrix and, at the same time, it indexes the data, enabling fast queries over the raster matrix. It has proven to be superior to the rest of the state-of-the-art techniques both in compression power and query performance, except in the case of NetCDF's compression power, since they are almost on par.²³

Compared to k^2 -acc, the technique used in the previous proposal of map algebra using compact data structures,³¹ the k^2 -raster not only obtains less space consumption and query times, but it also scales better when increasing the size of the input data or the number of different values. In fact, when the number of different values is moderate, the current implementation of k^2 -acc does not work, and any possible implementation will have problems to deal with that situation since it requires a complete k^2 -tree for each different cell value present in the raster. Moreover, except in rasters with a

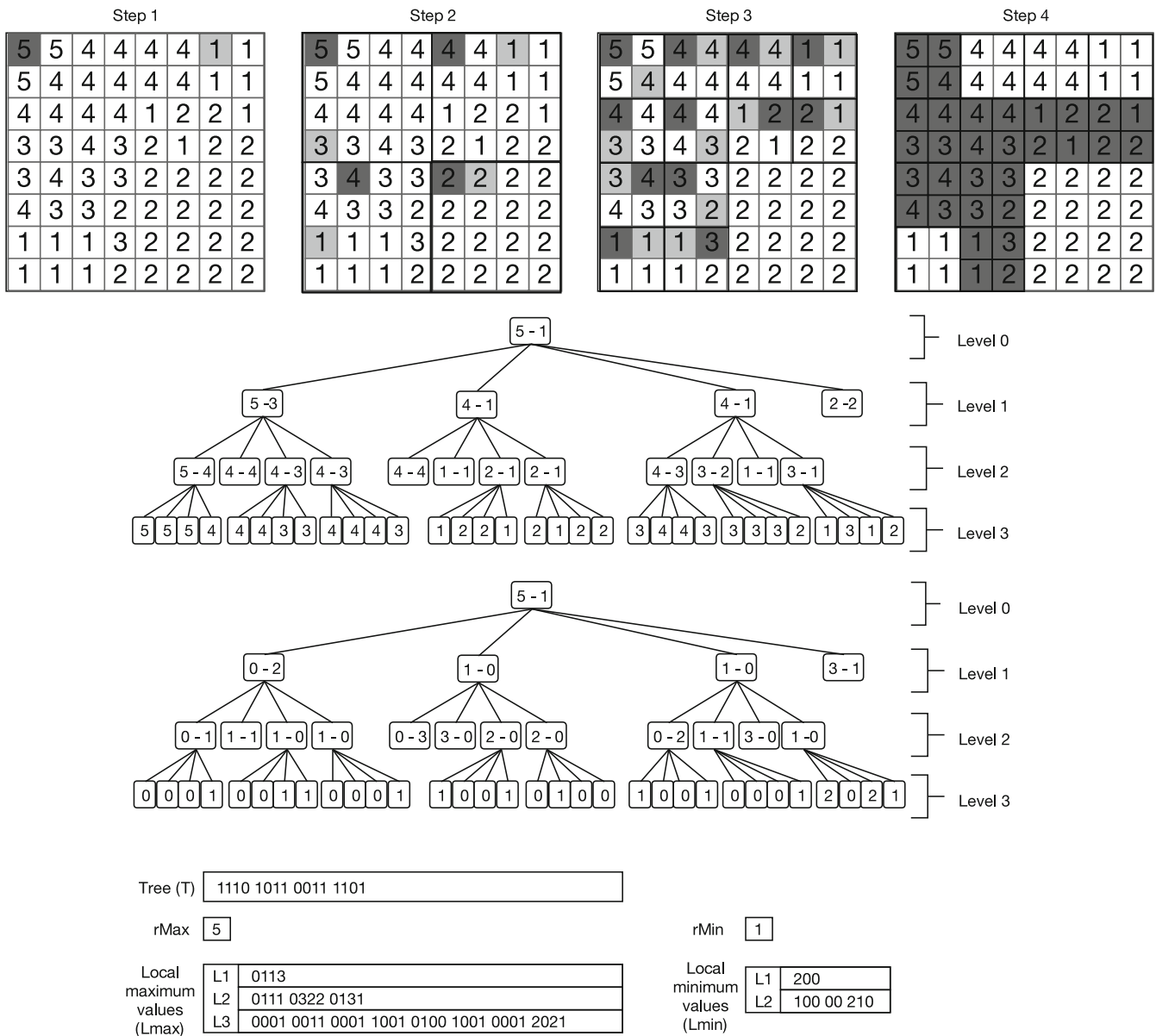


FIGURE 2 k^2 -raster example. Example of integer raster matrix (top), conceptual tree of the k^2 -raster (center-top), conceptual tree using differential encoding (center-bottom), and final representation of the raster matrix using compact data structures (bottom). $rMax$ and $rMin$ denote the maximum and minimum values of the root node. $Lmax$ and $Lmin$ contain the maximum and minimum values of each node, following a level-wise order and using differential encoding. This example uses $k = 2$.

very low number of different values in the raster, k^2 -acc obtains between 2.5 and 8.75 times worse compression. In query times, in most queries, k^2 -raster is faster than k^2 -acc, up to one order of magnitude.

3.3 | Quadrees

k^2 -acc, k^3 -tree, and k^2 -raster use a quadtree as the basic data structure (octree in the case of k^3 -tree), which is then enhanced with several other components. Quadtrees were originally introduced to compress images.^{42,43} However, since then, many different variants have appeared. We can roughly classify them in those focused on indexing data⁴⁴⁻⁴⁶ and those focused on compressing images or 3D meshes.⁴⁷⁻⁵³

As indexes, typically, data are stored in an array-like data structure and a quadtree is used as an auxiliary index,^{44,54} whereas, in the case of compression, no indexation is applied. However, k^2 -acc, k^3 -tree, and k^2 -raster combine in the same data structure two indexes and the storage of the data. A similar fusion has been proposed more recently,⁵⁵ however this system only indexes the space, whereas k^2 -acc, k^3 -tree, and k^2 -raster, in addition to the spatial index (quadtree), are equipped with an index on the values at cells.

Focusing on compression, the basic idea of the quadtree is to take advantage of the intrinsic data smoothness, by representing parts of the image with only one value or, at least, represent parts of image with less information than the original by taking advantage of the redundancy between close parts of the image. k^2 -raster uses the same idea, but adding some other compression mechanisms.

3.4 | Operations over rasters using compact data structures

Brisaboa et al.⁵⁶ proposed algorithms to efficiently perform Boolean operations over binary raster datasets represented using k^2 -trees. They included algorithms for computing the union, intersection, difference, and complement of binary rasters using two different variants of the k^2 -tree representation for binary relations. Thus, it is related to our work as they propose algorithms for computing some local operations over raster datasets, but it differs from us in the sense that their work is focused on binary rasters, whereas we extend the applicability of these types of map algebra operators for rasters of any nature. Later, Quijada-Fuentes et al.⁵⁷ presented algorithms for solving set operations over k^2 -tree and over another compact data structure called binary relation wavelet tree (BRWT); however, again, this work only considers binary rasters.

The closest work to ours is the development of map algebra operations on rasters represented using k^2 -acc.³¹ They presented algorithms for the following operations: thresholding, sum and multiplication by a scalar, point-wise sum, and zonal sum. However, as explained, k^2 -acc has poor performance in comparison with k^2 -raster, especially when the raster has a moderate or high number of different values, which is the most common situation.

4 | OUR PROPOSAL

First of all, let us introduce the following notation. Let p_{M_1} be a pointer to a node of the k^2 -raster M_1 . With $*$ (p_{M_1}), we access the pointed node. That is, $*$ (p_{M_1}).*min* and $*$ (p_{M_1}).*max* return the minimum and maximum values, respectively, stored at the pointed node.

4.1 | Global operations: Arithmetic operation by a scalar value

This operation performs the addition, subtraction, division, or multiplication of a scalar value to all cells of the matrix. It is performed differently depending on whether the operation is addition/subtraction or division/multiplication. However, in both cases, it is simple.

In the case of addition or subtraction, we only have to sum or subtract the input scalar to the values *maximum* and *minimum* at the root node, that is, the fields *rMax* and *rMin* of the Figure 2. Observe that, independently of the size of the raster, this implies just two operations.

In the case of division and multiplication, the algorithm requires applying the operation to all nodes of the tree. This implies applying the operation to $rMax$ and $rMin$ and to all the values in $Lmax$ and $Lmin$. However, the structure of the tree does not change; thus, the operations over $Lmax$ and $Lmin$ can be done sequentially without traversing the tree.

4.2 | Local operations: Point-wise

In this case, the value of each cell of the output matrix $M_o(i,j)$ is computed as the operation over the cells at the same location of the two input matrices, that is, $M_o(i,j) = M_1(i,j)\Theta M_2(i,j)$, being Θ an arithmetic operation (+, −, *, or /).

Algorithm 1 shows the pseudocode of the operation. At any given step of the algorithm, it manages two pointers. Initially, both pointers point to the root node of the two input k^2 -rasters. The algorithm traverses downwards both input trees until reaching quadrants of size 1×1 , that is, individual cells, or a uniform quadrant. It is invoked as **Point-wise**($\Theta, n, 1, p_{M_1}, p_{M_2}$). The first parameter is the operation, n is the (possibly extended) raster matrix size, the third parameter is the current level, and p_{M_1} and p_{M_2} are pointers to the current node of each input k^2 -raster. In the initial call, both point to the root node of the corresponding k^2 -raster.

$k, T_\ell, Vmax_\ell,$ and $Vmin_\ell$ are global variables. k is the k value of the trees. $T_\ell, Vmax_\ell,$ and $Vmin_\ell$ are lists, initially empty; there are $T_\ell, Vmax_\ell,$ and $Vmin_\ell$ lists for each level ℓ of the output tree. In addition, the global variables $pmax_\ell$ and $pmin_\ell$ indicate the last written position of $Vmax_\ell$ and $Vmin_\ell$, respectively.

After running the algorithm, all T_ℓ sequences must be joined to make up T . The same must be done with $Vmax_\ell$ and $Vmin_\ell$ to obtain $Vmax$ and $Vmin$, which, in turn, must be converted into $Lmax$ and $Lmin$ by computing the differences and then encoded using DACs. Observe that the algorithm returns the maximum and minimum values of the resulting matrix, that is, $rMax$ and $rMin$.

Each call to *Point-wise* processes one quadrant (let us say q_p) of both input matrices, initially the whole raster, to obtain the same quadrant of the result.

If the two input pointers correspond to uniform quadrants, that is, all cells have the same value, then lines 4 and 5 obtain the value that will have the output raster in all cells of the processed quadrant. Line 6 performs the operation and introduces the resulting value in both $maxval$ and $minval$ (the answer of the call). When at least one of the nodes is not a leaf, for each non-leaf node, the *for*s of lines 8 and 9 process its k^2 children, with lines 10–17 obtaining the pointers to those children. In the case of leaf node, the pointer remains pointing to the current processed node q_p . For each subquadrant of the processed node, line 18 performs a recursive call to process it.

After the recursive call, line 19 adds the returned maximum value to $Vmax_\ell$. Line 20 checks if the returned values are different. In that case, it appends the minimum value to $Vmin_\ell$ and sets up a 1 in the T_ℓ of that level. If the maximum and minimum values are equal, it sets up a 0 in T_ℓ , provided that we are not returning from a recursive call that processed an individual cell. Lines 28–30 keep track of the minimum and maximum values found so far during the computation of q_p of the target raster.

After processing the k^2 children ($q_{p1}, q_{p2}, \dots, q_{pk^2}$), line 34 checks whether the maximum and minimum values of all children are equal, which indicates that all the children contain the same value. Thus, the algorithm must undo the last operations, as these nodes will not have a representation in the data structure. This can be easily done by removing the last k^2 positions of T_ℓ and $Vmax_\ell$, or just moving the pointer that indicates their last written position, k^2 positions backwards (line 35). Finally, the algorithm returns the maximum and minimum values to its parent call.

Figure 3 shows two rasters, the output of their point-wise sum and their conceptual k^2 -rasters. We are going to use them to illustrate how Algorithm 1 works. The first level division of the rasters is depicted with thick lines. The second level division is drawn with thin lines. First level division produces four quadrants labeled as $q_1, q_2, q_3,$ and q_4 .

In order to distinguish between the two input rasters, we will use $q_i(M_1)$ to refer to the quadrant q_i of the first input raster and $q_i(M_2)$ for the other. To refer the corresponding node of quadrant $q_i(M_j)$, we will use $n_i(M_j)$. Equally, where a pointer is required, by $n_i(M_j)$ also stands as the pointer to that node in the k_2 -raster.

The process begins with the call **Point-wise**(+, 4, 1, $p_{root}(M_1), p_{root}(M_2)$), where $p_{root}(M_1)$ and $p_{root}(M_2)$ are pointers to the root of M_1 and M_2 , respectively. Since $n > 1$, the loops of lines 8 and 9 start with $i = 0$ and $j = 0$, which means that this iteration processes the quadrant q_1 of both input rasters. Given that $n_1(M_1)$ and $n_1(M_2)$ have different min-max values, then lines 11 and 15 obtain the new pointers $p_{M_1}^\ell$, pointing to $n_1(M_1)$, and $p_{M_2}^\ell$, pointing to $n_1(M_2)$.

Algorithm 1. **Point-wise**($\Theta, n, \ell, p_{M_1}, p_{M_2}$) computes T , $Vmax$ and $Vmin$ and returns ($rMax$, $rMin$) of the k^2 -raster representation resulting from performing the operation Θ having as input two rasters, where p_{M_1} and p_{M_2} are pointers to the root node of those k^2 -rasters

```

1:  $minval \leftarrow \infty$ ;
2:  $maxval \leftarrow 0$ ;
3: if IsLeaf( $p_{M_1}$ ) and IsLeaf( $p_{M_2}$ ) then /*Both input quadrants are uniform, that is, leaves*/
4:    $M1 \leftarrow * (p_{M_1}).max$ 
5:    $M2 \leftarrow * (p_{M_2}).max$ 
6:    $maxval, minval \leftarrow M1 \Theta M2$ 
7: else
8:   for  $i \leftarrow 0 \dots k - 1$  do
9:     for  $j \leftarrow 0 \dots k - 1$  do
10:      if  $* (p_{M_1}).min <> * (p_{M_1}).max$  then /*Check whether the node has children or not*/
11:         $p_{M_1}^{\ell} \leftarrow \mathbf{Child}(p_{M_1}, i \cdot k + j)$  /*Obtains the child*/
12:      else  $p_{M_1}^{\ell} \leftarrow p_{M_1}$  /*The pointer remains in the current node since it is a
leaf*/
13:      end if
14:      if  $* (p_{M_2}).min <> * (p_{M_2}).max$  then /*Check whether the node has children or not*/
15:         $p_{M_2}^{\ell} \leftarrow \mathbf{Child}(p_{M_2}, i \cdot k + j)$  /*Obtains the child*/
16:      else  $p_{M_2}^{\ell} \leftarrow p_{M_2}$  /*The pointer remains in the current node since it is a
leaf*/
17:      end if
18:      ( $childmax, childmin$ )  $\leftarrow \mathbf{Point - wise}(\Theta, n/k, \ell + 1, p_{M_1}^{\ell}, p_{M_2}^{\ell})$ ;
19:       $Vmax_{\ell}[pmax_{\ell}] \leftarrow childmax$ ;
20:      if  $childmax <> childmin$  then /*The created node has children*/
21:         $Vmin_{\ell}[pmin_{\ell}] \leftarrow childmin$ 
22:         $pmin_{\ell} \leftarrow pmin_{\ell} + 1$ 
23:         $T_{\ell}[pmax_{\ell}] \leftarrow 1$ 
24:      else if  $n > k$  then /*If it is not the last level*/
25:         $T_{\ell}[pmax_{\ell}] \leftarrow 0$  /*The created node does not have children*/
26:      end if
27:       $pmax_{\ell} \leftarrow pmax_{\ell} + 1$ 
28:      if  $minval > childmin$  then  $minval \leftarrow childmin$ 
29:      end if
30:      if  $maxval < childmax$  then  $maxval \leftarrow childmax$ 
31:      end if
32:    end for
33:  end for
34:  if  $minval = maxval$  then /*All children have the same value in all cells*/
35:     $pmax_{\ell} \leftarrow pmax_{\ell} - k^2$  /*Remove all the children nodes*/
36:  end if
37: end if
38: return( $maxval, minval$ )

```

Then a recursive call **Point-wise**($+$, $2, 2, n_1(M_1), n_1(M_2)$) is issued (see step 2 of Table 1). Observe that $n_1(M_1)$ is the node containing 1-1 in M_1 and $n_1(M_2)$ is the one containing 4-4 in M_2 . Therefore, both are leaves, their values are 1 and 4 respectively, and then lines 4–6 store the result of their sum, 5, in the $maxval$ and $minval$ of the output for this quadrant.

At the returning of the call, $Vmax_1[1]$ receives the 5 and line 25 sets $T_1[1]$ to 0 indicating that it is a leaf node (see step 3 of Table 1). Lines 28 and 30 set $minval$ and $maxval$ to 5, since that is the minimum and maximum values found so far in the children of the root.

We returned to the call of step 1, and now the *for* of line 10 sets j to 1, and this implies that the subquadrant 2 must be processed, and thus, the recursive call **Point-wise**($+$, $2, 2, n_2(M_1), n_2(M_2)$) is issued (see step 4 of Table 1). Since q_2 is not uniform in both input trees, the flow reaches line 8, and then the k^2 subquadrants of q_2 must be processed with recursive calls. Steps 5–12 of Table 1 show the k^2 recursive calls for processing the subquadrants.

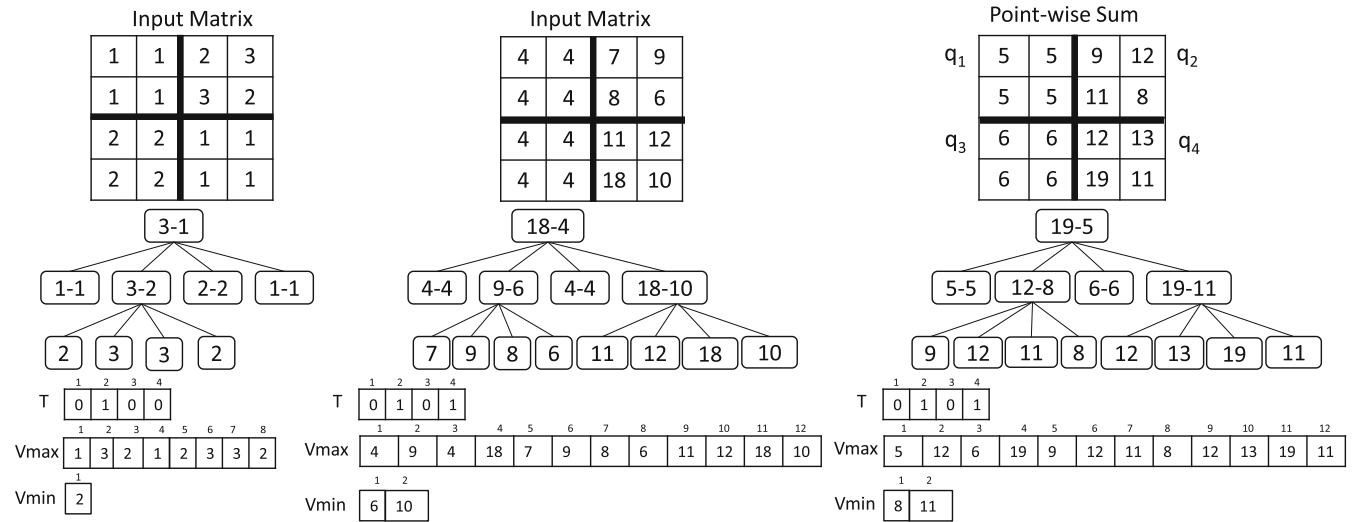


FIGURE 3 The k^2 -rasters involved in the point-wise sum operation.

TABLE 1 Trace of the point-wise sum of the example of Figure 3.

Step	Call	Actions	Rec call	Actions	Rec call
1	$4,1,p_{root}(M_1),p_{root}(M_2)$				
2			$2,2,n_1(M_1),n_1(M_2)$		
3		$Vmax_1(5) T_1(0)$			
4			$2,2,n_2(M_1),n_2(M_2)$		
5					$1,3,n_{11}(M_1),n_{11}(M_2)$
6				$Vmax_2(9)$	
7					$1,3,n_{12}(M_1),n_{12}(M_2)$
8				$Vmax_2(9,12)$	
9					$1,3,n_{13}(M_1),n_{13}(M_2)$
10				$Vmax_2(9,12,11)$	
11					$1,3,n_{14}(M_1),n_{14}(M_2)$
12				$Vmax_2(9,12,11,8)$	
13		$Vmax_1(5,12) Vmin_1(8) T_1(0,1)$			
14			$2,2,n_3(M_1),n_3(M_2)$		
15		$Vmax_1(5,12,6) T_1(0,1,0)$			
16			$2,2,n_4(M_1),n_4(M_2)$		
17					$1,3,n_4(M_1),n_{41}(M_2)$
18				$Vmax_2(9,12,11,8,12)$	
19					$1,3,n_4(M_1),n_{42}(M_2)$
20				$Vmax_2(9,12,11,8,12,13)$	
21					$1,3,n_4(M_1),n_{43}(M_2)$
22				$Vmax_2(9,12,11,8,12,13,19)$	
23					$1,3,n_4(M_1),n_{44}(M_2)$
24				$Vmax_2(9,12,11,8,12,13,19,11)$	
25		$Vmax_1(5,12,6,19) Vmin_1(8,11) T_1(0,1,0,1)$			

Step 5 shows the call **Point-wise**(+, 1, 3, $n_{11}(M_1)$, $n_{11}(M_2)$). n_{11} is an individual cell, with value 2 in M_1 and 7 in M_2 , therefore lines 4–6 set *maxval* and *minval* to 9. After the recursive call, when the flow returns to the call of step 4, line 19 adds the 9 to $Vmax_2$ (see step 6). Neither $Vmin_2$ nor T_2 are updated since we are at the last level (individual cells). Until step 12, the same procedure is applied to the rest of the children of n_2 .

After processing all children of n_2 , the flow returns to the call of step 1 in line 19. The call **Point-wise**(+, 2, 2, $n_2(M_1)$, $n_2(M_2)$) returns (12,8), and thus 12 is attached to $Vmax_1$ and 8 to $Vmin_1$. Finally, a 1 is attached to T_1 since n_2 in the output tree has children.

Next, the call of step 1 processes q_3 issuing **Point-wise**(+, 2, 2, $n_3(M_1)$, $n_3(M_2)$), but since n_3 is a leaf in both input trees, that call returns (6, 6), resulting in the updates of step 15.

Finally, the process of q_4 is similar to the process of q_2 since, although in M_1 that node is a leaf, it is not in the case of M_2 (see steps 16–25).

4.3 | Global operations: Thresholding

Algorithm 2 shows the thresholding operation. It is also a recursive procedure that starts at the root but, in this case, it only traverses one tree. It is invoked as **Thresholdin**(n , ℓ , p_M , t). The first parameter is n the raster matrix size, the second parameter is the current level, p_M is a pointer to the current node of the input k^2 -raster, and t is the threshold. In the initial call, it points to the root node. The global variables are the same as in the case of the Algorithm 1.

The main difference is the recursion stop condition. As in the case of the point-wise operation, when we reach a leaf, recursion stops, but now returning 0 or 1 in both *maxval* and *minval* depending on whether the value at the node is greater or lower than the threshold.

There are two additional cases where recursion stops. If the maximum value at the processed node is smaller than the threshold, then it is sure that all values of that subquadrant are smaller than the threshold, so the recursion stops returning a 0 in both *maxval* and *minval*. Observe that, since we store the same value in both *maxval* and *minval*, then at the returning point of the recursion lines 18–35 will create a leaf node. The other case is when the minimum value of the processed node is greater than the threshold, which results in returning a 1 in both *maxval* and *minval*. Otherwise, recursion continues performing a call for each of the k^2 children of the processed node (lines 14–17), and after returning from that call, the output tree is built (lines 18–34).

Note that this operation takes advantage of the k^2 -raster index capability provided by the min–max values at nodes, which avoids processing tree branches that can be solved at the upper levels of the tree.

4.4 | Zonal operations: Zonal sum

Observe that the point-wise and thresholding operations use recursive algorithms that start at the root of the input trees, and process them downwards until they reach the leaves. Then, when returning from the recursive calls, they build the output tree from the leaves until the root. Therefore, basically, the process involves a downward and an upward traversal.

In this operation, this approach cannot be used, because we have to sum all values in the input matrix overlapping any given value of the zonal matrix. Therefore, until processing completely both input matrices, we cannot start the construction of the output tree.

Zonal sum will be solved with two traversals as well, the first one traverses the input trees downwards and the second one builds the output tree upwards. However, the first traversal must finish completely before the second one starts. This prevents the use of a recursive procedure that mixes the two traversals. The first traversal to obtain the zonal sum of each value of the zonal matrix uses a recursive algorithm. The second traversal is a sequential process of the levels of the zonal matrix from the leaves until the root to build the output tree.

Note that the output tree is basically the zonal tree, except that some non-uniform submatrices (non-leaf nodes) of the zonal tree may become uniform, and thus leaves, in the output tree. Therefore, during the first downward traversal, in addition to obtaining the zones, the algorithm records the structure of the zonal tree using node lists. Each list contains the nodes of one level from left to right. Then, the upward traversal of the zonal matrix is performed using those lists.

Algorithm 2. **Thresholding**(n, ℓ, p_M, t) computes T , $Vmax$ and $Vmin$ and returns ($rMax, rMin$) of the k^2 -raster representation resulting from performing the thresholding having as input one raster and a threshold value

```

1:  $minval \leftarrow \infty$ ;
2:  $maxval \leftarrow 0$ ;
3: if IsLeaf( $p_M$ ) then /*The input quadrant is uniform, that is, leaf*/
4:   if * ( $p_M$ ). $max < t$  then
5:      $maxval, minval \leftarrow 0$ 
6:   else
7:      $maxval, minval \leftarrow 1$ 
8:   end if
9: else if * ( $p_M$ ). $max < t$  then
10:   $maxval, minval \leftarrow 0$  /*All values in the quadrant are below  $t$ */
11: else if * ( $p_M$ ). $min > t$  then
12:   $maxval, minval \leftarrow 1$  /*All values in the quadrant are above  $t$ */
13: else
14:  for  $i \leftarrow 0 \dots k - 1$  do
15:    for  $j \leftarrow 0 \dots k - 1$  do
16:       $p_{\ell_M} \leftarrow \mathbf{Child}(p_M, i \cdot k + j)$  /*Obtains the child*/
17:      ( $childmax, childmin$ )  $\leftarrow \mathbf{Thresholding}(n/k, \ell + 1, p_{\ell_M}, t)$ ;
18:       $Vmax_{\ell}[pmax_{\ell}] \leftarrow childmax$ ;
19:      if  $childmax <> childmin$  then /*The created node has children*/
20:         $Vmin_{\ell}[pmin_{\ell}] \leftarrow childmin$ 
21:         $pmin_{\ell} \leftarrow pmin_{\ell} + 1$ 
22:         $T_{\ell}[pmax_{\ell}] \leftarrow 1$ 
23:      else if  $n > k$  then /*If it is not the last level*/
24:         $T_{\ell}[pmax_{\ell}] \leftarrow 0$  /*The created node does not have children*/
25:      end if
26:       $pmax_{\ell} \leftarrow pmax_{\ell} + 1$ 
27:      if  $minval > childmin$  then  $minval \leftarrow childmin$ 
28:      end if
29:      if  $maxval < childmax$  then  $maxval \leftarrow childmax$ 
30:      end if
31:    end for
32:  end for
33:  if  $minval = maxval$  then /*All children have the same value in all cells*/
34:     $pmax_{\ell} \leftarrow pmax_{\ell} - k^2$  /*Remove all the children nodes*/
35:  end if
36: end if
37: return( $maxval, minval$ )

```

As explained, the upward traversal replicates the zonal tree, and only when the algorithm finds that a non-uniform quadrant of the zonal matrix becomes uniform in the output, then it changes the output accordingly. To detect those situations, the lists, in addition to informing about the structure of the zonal tree, are also used as an auxiliary structure during the computation of the output to store the min-max values of the nodes of the output tree.

Algorithm 3 shows the pseudocode. The hash table $Hash$ has all its entries initialized to 0. $Vmax_2$ is the $Vmax$ array of the zonal matrix M_2 , and it is a global variable. The lists L_1, L_2, \dots , and P_1, P_2, \dots are also global variables. As in the previous algorithm, T_{ℓ} , $Vmax_{\ell}$, and $Vmin_{\ell}$ are lists, initially empty; there are T_{ℓ} , $Vmax_{\ell}$, and $Vmin_{\ell}$ lists for each level ℓ of the output tree. Again, after running the algorithm, all T_{ℓ} sequences must be joined to make up T and $Vmax_{\ell}$ and $Vmin_{\ell}$ must be processed to obtain $Lmax$ and $Lmin$.

Algorithm 3. ZonalSum($n, \#l, p_{M_1}, p_{M_2}$)

```

1: Hash  $\leftarrow$  Sum( $n, 1, p_{M_1}, p_{M_2}$ ) /*Obtains for each value in  $M_2$  its sum*/
2: Add( $L_1, \langle 0, -1, -1 \rangle$ );
3:  $l \leftarrow \#l$ 
4: while  $l > 1$  do /*Processes all levels from the leaves upwards*/
5:    $p_{max_l} \leftarrow p_{min_l} \leftarrow i \leftarrow 0$ 
6:    $minval \leftarrow \infty$ 
7:    $maxval \leftarrow 0$ 
8:   while  $L_l[i]$  is not null do /*Traverses  $L_l$ */
9:     if  $L_l[i].max = -1$  then /* It was a leaf already in  $M_2$ */
10:        $V_{max_l}[p_{max_l}] \leftarrow min \leftarrow Hash[V_{max_2}[L_l[i].ptr]]$ 
11:     else /* It is a parent, the max and min values were updated when its children
were processed*/
12:        $V_{max_l}[p_{max_l}] \leftarrow min \leftarrow L_l[i].max$ 
13:       if  $L_l[i].max <> L_l[i].min$  then
14:          $V_{min_l}[p_{min_l}] \leftarrow min \leftarrow L_l[i].min$ 
15:          $p_{min_l} \leftarrow p_{min_l} + 1$ 
16:       end if
17:     end if
18:     if  $l < \#l$  then /* The nodes in the last level are not represented in  $T$ */
19:       if  $L_l[i].max = L_l[i].min$  then  $T_l[p_{max_l}] \leftarrow 0$  /*The submatrix is uniform, then it is
represented by a leaf node*/
20:       else  $T_l[p_{max_l}] \leftarrow 1$  /*The submatrix is not uniform, then its node has
children*/
21:     end if
22:     end if
23:     if  $maxval < V_{max_l}[p_{max_l}]$  then  $maxval \leftarrow V_{max_l}[p_{max_l}]$ 
24:     end if
25:     if  $minval > min$  then  $minval \leftarrow min$ 
26:     end if
27:     if  $(i + 1) \bmod k^2 = 0$  then /*The parent must be updated*/
28:       if  $minval = maxval$  then /*All children have the same value in all cells*/
29:          $p_{max_l} \leftarrow p_{max_l} - k^2$  /*Remove all the children nodes*/
30:       end if
31:        $\#parent \leftarrow (i/k^2)$  /*We are processing the children of the  $\#parent^{th}$  of level
 $l - 1$ */
32:        $pp_{l-1} \leftarrow P_{l-1}[\#parent]$  /*Obtains the position in level  $l - 1$  from the array of
parents*/
33:        $L_{l-1}[pp_{l-1}].max \leftarrow maxval$ 
34:        $L_{l-1}[pp_{l-1}].min \leftarrow minval$ 
35:        $minval \leftarrow \infty$ 
36:        $maxval \leftarrow 0$ 
37:     end if
38:      $p_{max_l} \leftarrow p_{max_l} + 1$ 
39:      $i \leftarrow i + 1$ 
40:   end while
41:    $l \leftarrow l - 1$ 
42: end while

```

Algorithm 4. $\text{Sum}(n, \ell, p_{M_1}, p_{M_2})$

```

1: if  $\text{IsLeaf}(p_{M_1})$  and  $\text{IsLeaf}(p_{M_2})$  then /*Both input quadrants are uniform, that is, leaves*/
2:    $M1 \leftarrow (p_{M_1}).max$ 
3:    $M2 \leftarrow (p_{M_2}).max$ 
4:    $\text{Hash}[M2] \leftarrow \text{Hash}[M2] + (M1 \times n^2)$  /*Adds to the value  $M2$  of the zonal matrix the
   values of the input raster matrix*/
5: else
6:   for  $i \leftarrow 0 \dots k - 1$  do
7:     for  $j \leftarrow 0 \dots k - 1$  do
8:       if  $(p_{M_1}).min <> (p_{M_1}).max$  then /*Check whether the node has children or not*/
9:          $p_{\ell+1M_1} \leftarrow \text{Child}(p_{M_1}, i \cdot k + j)$  /*Obtains the child*/
10:      else  $p_{\ell+1M_1} \leftarrow p_{M_1}$  /*The pointer remains in the current node since it is a
leaf*/
11:     end if
12:     if  $(p_{M_2}).min <> (p_{M_2}).max$  then /*Check whether the node has children or not*/
13:        $p_{\ell+1M_2} \leftarrow \text{Child}(p_{M_2}, i \cdot k + j)$  /*Obtains the child*/
14:       if  $(p_{\ell+1M_2}).min <> (p_{\ell+1M_2}).max$  then
15:          $\text{Add}(P_{\ell+1}, \text{sizeOf}(L_{\ell+1}))$ ; /*Stores in the array of parents the position in
level  $\ell + 1$  of this node*/
16:       end if
17:        $\text{Add}(L_{\ell+1}, \langle p_{\ell+1M_2}, -1, -1 \rangle)$ ; /*Adds the node to  $L_{\ell+1}$ */
18:     else  $p_{\ell+1M_2} \leftarrow p_{M_2}$  /*The pointer remains in the current node since it is a
leaf*/
19:     end if
20:      $\text{Sum}(n/k, \ell + 1, p_{\ell+1M_1}, p_{\ell+1M_2})$ ;
21:   end for
22: end for
23: end if
24: return  $\text{Hash}$ 

```

Each entry of the lists L_1, L_2, \dots is a triplet $\langle \text{pointer}, \text{min}, \text{max} \rangle$, where *pointer* is a pointer to a position in $V_{\text{max}2}$ (the maximum values of the zonal matrix M_2), *min* and *max* are the minimum and maximum values at the node corresponding to the position pointed to by *pointer*, but at the output tree.

The algorithm is called as $\text{ZonalSum}(n, \# \ell, p_{M_1}, p_{M_2})$. The first parameter is the raster matrix size, $\# \ell$ is the number of levels of the zonal tree (M_2). p_{M_1} and p_{M_2} are pointers to the root node of both input k^2 -rasters.

Line 1 performs the call to the recursive process that adds the values of the input matrix for each different value in the zonal matrix. The returned hash table *Hash* contains the sum for each value of the zonal matrix.

Algorithm 4 shows the pseudocode of this process. It is basically the same recursive process of the point-wise operation, excluding the construction of the answer. The differences are two. First, in line 4, when the recursive process reaches a leaf in both trees, the hash entry of the value of the zonal matrix accumulates the values of the input matrix. Second, when a node of the zonal matrix is accessed, line 17 adds it as an entry of the list of nodes $L_{\ell+1}$. The entry contains a pointer to that node and minimum and maximum values set to -1, and lines 14 and 15 add the position of the node within its level (from left to right), if it is a parent node. This list of parent positions is required during the construction process of the output.

After that call, returning to Algorithm 3, the *while* of line 4 processes the levels of the zonal matrix from the last level until level 2 (that of the children of the root). Variables $p_{\text{max}\ell}$ and $p_{\text{min}\ell}$ are the positions of $V_{\text{max}\ell}$ and $V_{\text{min}\ell}$ where a new element must be written. The *while* of line 8 processes sequentially the entries of L_ℓ .

If the processed entry of L_ℓ has a -1 in the *max* field, then that means that it is a leaf in the zonal matrix; otherwise, its children would have been processed during the level $\ell + 1$ traversal, and this would update the *max* and *min* fields of the processed entry. Therefore, in line 10, the corresponding position in $V_{\text{max}\ell}$ is filled with the sum associated with the

value of the zonal matrix. Otherwise, the maximum and minimum values are already at the entry, and thus, $Vmax_\ell$ and $Vmin_\ell$ are updated accordingly (lines 12–16).

Next, in lines 18–22, T is updated. Lines 23–26 keep track of the maximum and minimum values of the children of the current parent node.

Line 27 checks when the traversal reaches the last child of a parent node. In that case, we have to update the minimum and maximum values at the entry of the parent node in the upper level. Before that, if we find that the maximum and minimum values of the children are the same, line 29 moves the pointer $pmax_\ell$ k^2 positions backwards to erase the children of the output tree, since the parent node will be a leaf.

Lines 31 and 32 obtain the position in level $\ell - 1$ of the parent of the processed children using the list where we stored the position in each level of the parent nodes. Lines 33 and 34 update the maximum and minimum values at the parent node.

Next, we are going to illustrate the Algorithm 3 using the rasters of Figure 4. The algorithm is invoked as **ZonalSum**(4, 3, $p_{root}(M_1)$, $p_{root}(M_2)$). Line 1 issues **Sum**(4, 1, $p_{root}(M_1)$, $p_{root}(M_2)$). As seen, this is a recursive process that obtains the hash table with the sum for each value of the zonal matrix, the lists L_ℓ having the structure of the zonal tree by levels, and the lists P_ℓ of positions in each level having parent nodes. Figure 5 shows these elements for our example. Observe that the lists L_ℓ store the positions in $Vmax2$ corresponding to the nodes of the level ℓ and that the *min* and *max* values are set to -1. The list of parents is our example is only available for level 2, and, as seen, it marks only positions 1 and 3 (starting at position 0), since in the k^2 -tree of the zonal matrix, those positions of level 2 contain parent nodes (the nodes labeled 9-6 and 18-10).

After that call, returning to Algorithm 3, the *while* of line 4 processes the levels of the zonal tree starting at the last one (3). The *while* of line 8 processes L_3 from left to right. The first entry is $\langle 5, -1, -1 \rangle$; since it has a -1 in *max*, the algorithm obtains from the hash table the sum associated with the zonal value. For this, first, the zonal value is obtained using the pointer, so the zonal value is $Vmax2[5] = 7$. Next, the sum associated with 7 is obtained from the hash table, which in our case is a 2 (see the second entry from the top of table *Hash* of Figure 5). Finally, that sum is added to $Vmax_3$. Lines 18–22 are not executed since we are at the last level and it is not represented in T (see row with $i = 0$ and $\ell = 3$ of Table 2). Next, *maxval* and *minval*, which keep track of the maximum and minimum values of the parent node of the currently processed node, are set to 2. The next entry is $\langle 6, -1, -1 \rangle$, which adds a 3 to $Vmax_3$, and updates *maxval* to 3. The same process is carried out with the following two entries of L_3 , then when reaching $i = 3$, this results in $Vmax_3 = (2, 3, 3, 2)$ (see rows with $i = 1, 2, 3$ and $\ell = 3$ of Table 2). However, at that point, that is, when processing the entry $\langle 8, -1, -1 \rangle$, we are processing the last child of the parent node labeled 9-6 in the zonal tree. Therefore, the *if* of line 27 becomes true. Lines 31 and 32 compute the position in level 2 of the node 9-6 of the zonal tree. Line 31 computes the number of parent (from left to

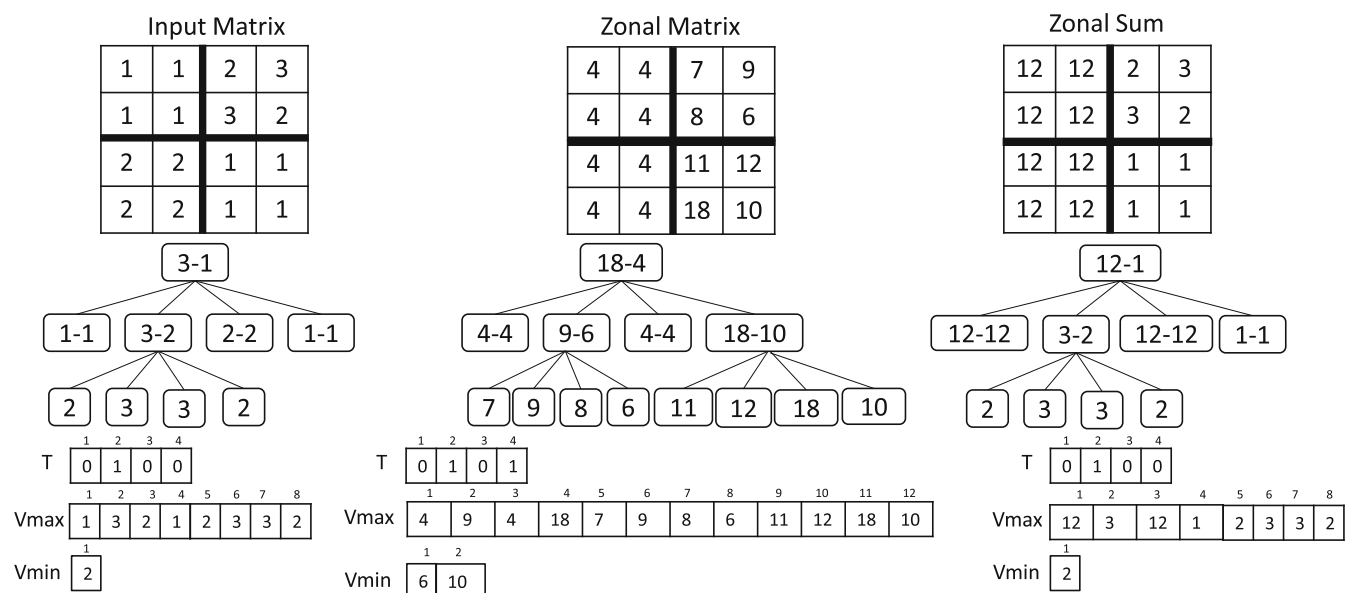


FIGURE 4 The k^2 -rasters involved in the zonal sum operation.

0	1	2	3
L_2	<1,-1,-1>	<2,-1,-1>	<3,-1,-1>

0	1	2	3	4	5	6	7
L_3	<5,-1,-1>	<6,-1,-1>	<7,-1,-1>	<8,-1,-1>	<9,-1,-1>	<10,-1,-1>	<11,-1,-1>

0	1
P_2	1 3

Hash	
Value	Sum
4	12
7	2
9	3
8	3
6	2
11	1
12	1
18	1
10	1

FIGURE 5 The data structures obtained after the call to *Sum* function.

TABLE 2 Trace of the zonal sum of the example of Figure 4.

ℓ	i	$Vmax$	Additional actions
3	0	2	
	1	2,3	
	2	2,3,3	
	3	2,3,3,2	$L_2\langle 1, -1 - 1 \rangle, \langle 2, 3, 2 \rangle, \langle 3, -1, -1 \rangle, \langle 4, -1, -1 \rangle$
	4	2,3,3,2,1	
	5	2,3,3,2,1,1	
	6	2,3,3,2,1,1,1	
2	7	2,3,3,2,1,1,1,1	$Vmax_3(2, 3, 3, 2) \quad L_2\langle 1, -1 - 1 \rangle, \langle 2, 3, 2 \rangle, \langle 3, -1, -1 \rangle, \langle 4, 1, 1 \rangle$
	0	12	$T_2(0)$
	1	12,3	$Vmin_2(2) \quad T_2(0, 1)$
	2	12,3,12	$T_2(0, 1, 0)$
	3	12,3,12,1	$T_2(0, 1, 0, 0) \quad L_1\langle 0, 12, 1 \rangle$

right) in level 2, $\#parent = 3/4 = 0$, and line 32 computes its global position in level 2, $pp_2 = P_2[0] = 1$. Then, finally, the entry at position 1 of L_2 is updated with the maximum value of the last k^2 processed values (3) and the minimum (2), see row with $i = 3$ and $\ell = 3$ of Table 2.

See in Table 2 that from $i = 4$ to $i = 7$ for $\ell = 3$, the k^2 leaves of the node 18-10 of the zonal tree (which are the values 11, 12, 18, 10) have a zonal sum of 1, and thus the four 1s are added to $Vmax_3$. When processing the fourth child, that is when $i = 7$, then the *if* of line 27 becomes true; $minval = maxval = 1$, and thus, $pmax_\ell$ is moved k^2 positions backwards, thus removing the last four added 1s. In addition, lines 31-34 update the minimum and maximum values of the last entry of L_2 , setting both of them to 1.

Next, the flow returns to line 4 to process level 2. The first entry of L_2 ($\langle 1, -1, -1 \rangle$) has a -1 in *max*, so it is a leaf. Using the pointer, we obtain the associated sum (12) from the hash table (see entry with $\ell = 2$ and $i = 0$ of Table 2). However, now we are not at the last level; thus, lines 18-22 are executed, adding in this case a 0 to T_2 , which indicates that this node is a leaf. The next entry is $\langle 2, 3, 2 \rangle$. Given that *max* is not -1, the node is not a leaf, and then $Vmax_2$ and $Vmin_2$ are updated with the values stored at the entry, as it can be seen in the row with $i = 1$ and $\ell = 2$. In addition, a 1 is added to T_2 indicating that it is not a leaf. The third entry is like the first one. The fourth entry ($\langle 4, 1, 1 \rangle$) has *max* different from -1. In this case, both values are the same (1) and thus only $Vmax_2$ and T_2 are updated (see row with $i = 3$ and $\ell = 2$ of Table 2).

4.5 | A note on complexity

In this section, we will analyze the space and time complexity of the proposed algorithms. In the case of the space complexity, we do not include the input/output sizes, but the additional space required to compute the map algebra operations and build the output.

4.5.1 | Global operations: Arithmetic operation by a scalar value

In the case of adding or subtracting a scalar value, both space and time complexities are $O(1)$, as only the fields $rMax$ and $rMin$ need to be modified.

In the case of multiplication by a scalar value, not only $rMax$ and $rMin$ are modified, by all the values of L_{max} and L_{min} arrays. Thus, time complexity is linear to the time required to build these arrays, that is, $O(|L_{max}| + |L_{min}|)$, where L_{max} and L_{min} are the arrays storing the maximum and minimum values of the input k^2 -raster. Regarding space, it requires an additional space for building the DACs arrays of the output, as they cannot be created in compact space. Thus, the space complexity is $O(|L_{max}^{out}| + |L_{min}^{out}|)$, being L_{max}^{out} and L_{min}^{out} the arrays storing the maximum and minimum values of the output k^2 -raster.

4.5.2 | Local operations: Point-wise

The time complexity of the algorithm is linear to the input size, as it just process once each node of the input k^2 -rasters. Thus, it is $O(|M_1| + |M_2|)$, being M_1 and M_2 the input k^2 -rasters.

Regarding space complexity, it just requires the additional space for building the DACs arrays of the output, that is, $O(|L_{max}^{out}| + |L_{min}^{out}|)$, being L_{max}^{out} and L_{min}^{out} the arrays storing the maximum and minimum values of the output k^2 -raster.

4.5.3 | Global operations: Thresholding

The time complexity of the algorithm is linear to the input size, as it just process once each node of the input. Thus, it is $O(|M|)$, being M the input k^2 -raster.

Regarding space complexity, it just requires an additional space $O(|L_{max}^{out}| + |L_{min}^{out}|)$ for building the DACs arrays of the output, being L_{max}^{out} and L_{min}^{out} the arrays storing the maximum and minimum values of the output k^2 -raster.

4.5.4 | Zonal operations: Zonal sum

Again, the time complexity of the algorithm is linear to the input size, as it just process once each node of the input. Thus, it is $O(|M_1| + |M_2|)$, being M_1 the input k^2 -raster and M_2 the zonal matrix. Assuming M_2 is smaller than M_1 , time complexity is $O(|M_1|)$.

Regarding space complexity, again it just requires an additional space $O(|L_{max}^{out}| + |L_{min}^{out}|)$ for building the DACs arrays of the output, being L_{max}^{out} and L_{min}^{out} the arrays storing the maximum and minimum values of the output k^2 -raster.

5 | EXPERIMENTAL EVALUATION

We include in this section to three different experiments. In the first one, we compare our algorithms with two baselines: NetCDF and a naive solution that completely decompresses the input rasters, runs the query over the decompressed data, and finally compresses the result to obtain a k^2 -raster. The second experiment compares our method with the algorithms proposed for the k^2 -acc.³¹ The reason of this division is that we use the authors' implementation of the algorithms on k^2 -acc, which do not write the output, that is, they only compute the result in main memory, but they do not write that result back to disk. Since NetCDF does that, in order to be fair, we decided to run a second experiment with our algorithms

modified to behave in that way. Finally, the third experiment compares the memory consumption of all methods jointly, since the process of writing to disk does not affect this parameter.

In the case of netCDF, we used the netCDF library[‡] (v.4.7.4) with the recommended level 2 of compression, which obtains a good trade-off between space and access times. For the second experiment, the source code for the algorithms of the map algebra operations over the k^2 -acc was provided by their authors.³¹ It uses $k = 2$.

Our algorithms were implemented in C++ using the SDSL[§] library.⁵⁸ All the experiments were run on a dedicated Intel® Core™ i7-3820 CPU @ 3.60GHz (4 cores) with cache sizes 32KB (L1), 256KB (L2), and 10MB (L3), and 64GB of RAM. The operating system was Debian 9.12 with kernel 4.9.0-9-amd64. The code was compiled with gcc version 6.3.0 with `-O3`. We measured elapsed time (in seconds) and, in our experiments, all data were initially on disk. For all experiments, we repeat the map operation 30 times and report average times. Input datasets and values for scalar and thresholding operators were chosen randomly.

5.1 | Datasets

In our experiments, we used real datasets from two sources:

- WorldClim[¶] dataset⁵⁹ provides several layers of global climate information. The surface of the world is divided into equal-sized tiles. In turn, each tile is a raster with cells covering about 1 square kilometer. In our case, we used the layer with mean temperature. Whereas in the original dataset it is represented by a real number with one decimal, in our experiments we transformed it into an integer by multiplying the value by 10. The collections obtained from this source are denoted `clim`.
- Spanish Geographic Institute[#] (SGI) provides a DTM (Digital Terrain Model) of Spain, that is, spatial elevation data of the terrain. The surface is divided into rectangular equal-spaced tiles with 5 m of spatial resolution. Each cell of a tile stores a real number of at most three decimal digits.

Since the presence of uniform areas, that is, areas with the same value, may benefit the k^2 -raster, we considered two different levels of precision for the datasets from the second source by using different numbers of decimal digits. We generate collection `dfm0` by truncating the decimal digits of the raster values, and collection `dfm3` by using the original numbers, but represented as integers, that is, using three decimal digits and multiplying the original value by 1000.

We have created several collections of rasters. In each collection, all rasters have the same size. We decided to use collections of several rasters in order to eliminate any bias due to the use of a single matrix. Our collections are of different sizes to analyze the scalability of the algorithms. In collection labeled as `1x1`, each raster is a matrix built using just 1 tile of the original dataset, `2x2` collections contain matrices built using `2x2` adjacent tiles, and so on. Tables 3 and 4 show the characteristics of our collections of rasters. The data shown in the tables represent the mean values of all rasters in each collection. We also include in these tables the space consumption (in MBs) required by each of the three methods used, that is, k^2 -raster, NetCDF and k^2 -acc when representing these collections. We do not include the results of k^2 -acc for `dfm3` collection since it has too many different values and this technique is not able to run over these datasets.

5.2 | NetCDF experiment

Figure 6A shows the times of the sum by a scalar. This operation benefits clearly to k^2 -raster since the algorithm only implies adding (or subtracting) the scalar to the maximum and minimum values of the root node, which is a constant time algorithm. However, observe that the slope of the line corresponding to k^2 -raster slightly increases as the size of the dataset also increases. This is because our algorithm loads the complete k^2 -raster into main memory, and thus as the size of the k^2 -raster increases, the time for loading and writing from/to disk also increases.

[‡]<https://www.unidata.ucar.edu/software/netcdf/>

[§]<https://github.com/simongog/sdsl-lite>

[¶]<https://www.worldclim.org/>

[#]<https://www.ign.es>

TABLE 3 Properties of dataset `clim`, obtained from WorldClim datasets. It includes collections of raster matrices of different sizes.

Name	Size (MB)	#rows	#cols	# different values	Space consumption		
					k^2 -raster	NetCDF	k^2 -acc
<code>clim-1×1</code>	49.44	3600	3600	252	2.20	1.96	4.80
<code>clim-2×2</code>	197.75	7200	7200	413	10.49	9.20	21.67
<code>clim-3×3</code>	444.95	10,800	10,800	474	25.61	22.46	50.69
<code>clim-4×4</code>	791.02	14,400	14,400	498	42.03	36.88	86.79

Note: The column #different values shows the average number of different values of the input matrices of each size. The last three columns show the space consumption (in MBs) of the three methods used in the experiment.

TABLE 4 Properties of datasets `dfm0` and `dfm3`, obtained from DTM datasets. They include raster matrices of different sizes.

Name	Size (MB)	#rows	#cols	# different values	Space consumption		
					k^2 -raster	NetCDF	k^2 -acc
<code>dfm₀-1×1</code>	91.49	4100	5849	868	12.19	9.66	19.12
<code>dfm₀-2×2</code>	369.03	8242	11,737	1201	48.91	37.67	72.82
<code>dfm₀-3×3</code>	834.76	12,403	17,643	1503	108.65	83.51	174.30
<code>dfm₀-4×4</code>	1488.94	16,564	23,564	1761	197.08	150.79	295.34
<code>dfm₃-1×1</code>	91.49	4100	5849	779,405	53.88	39.00	-
<code>dfm₃-2×2</code>	369.03	8242	11,737	1,066,043	221.81	154.86	-
<code>dfm₃-3×3</code>	834.76	12,403	17,643	1,304,704	502.46	346.96	-
<code>dfm₃-4×4</code>	1488.94	16,564	23,564	1,545,248	897.94	619.66	-

Note: The column #different values shows the average number of different values of the input matrices of each size. The last three columns show the space consumption (in MBs) of the three methods used in the experiments.

Observe as the uniformity of the raster benefits in any case to the k^2 -raster. The most uniform collection is `clim`; thus, our algorithm is between 2.9 and 17.6 times faster than the naive approach and between 2.9 and 6.9 times faster than NetCDF. Something similar happens with `dfm0`, whereas with the least uniform, `dfm3`, improvements range between 4.1 and 6 times in the case of the naive approach and between 2.5 and 2.7 times in the case of NetCDF. Again, in this operation, the changes are due to the effect of uniformity in the compression power of k^2 -raster, as the more uniformity is found in the original raster, the shorter is the tree and thus more compression is achieved, and, as explained, in this operation, the main cost for k^2 -raster is to load and write from/to disk.

Figure 6B shows the results of product by a scalar. In this case, all values in the leaves of the tree must be accessed and operated. Although the tree is not accessed, and only arrays L_{max} and L_{min} are sequentially accessed, in this case, k^2 -raster is not able to outperform NetCDF in most cases. On the other hand, it achieves better performance and scales better compared with the naive approach. Our algorithm is between 1.38 and 2.3 times faster than the naive approach, whereas compared with NetCDF, except in the smallest collection of `clim` where they are on a par, in the rest, it is between 1.05 and 1.38 times slower.

In general, when a sequential scan of the whole raster is needed, NetCDF is capable of matching the speed of the k^2 -raster and even surpassing it. Several factors explain this:

- k^2 -raster is an index also carrying the data. However, when all data must be accessed, the advantages provided by the indexes disappear.
- Once the indexes are useless, we have to pay attention to the speed of the underlying compression method. NetCDF uses Deflate, which is a fast decompressor, whereas the DACs of the k^2 -raster are slower. This is expected, as Deflate does not worry about direct access, since it is an archival method, and NetCDF has to compress by blocks in order to

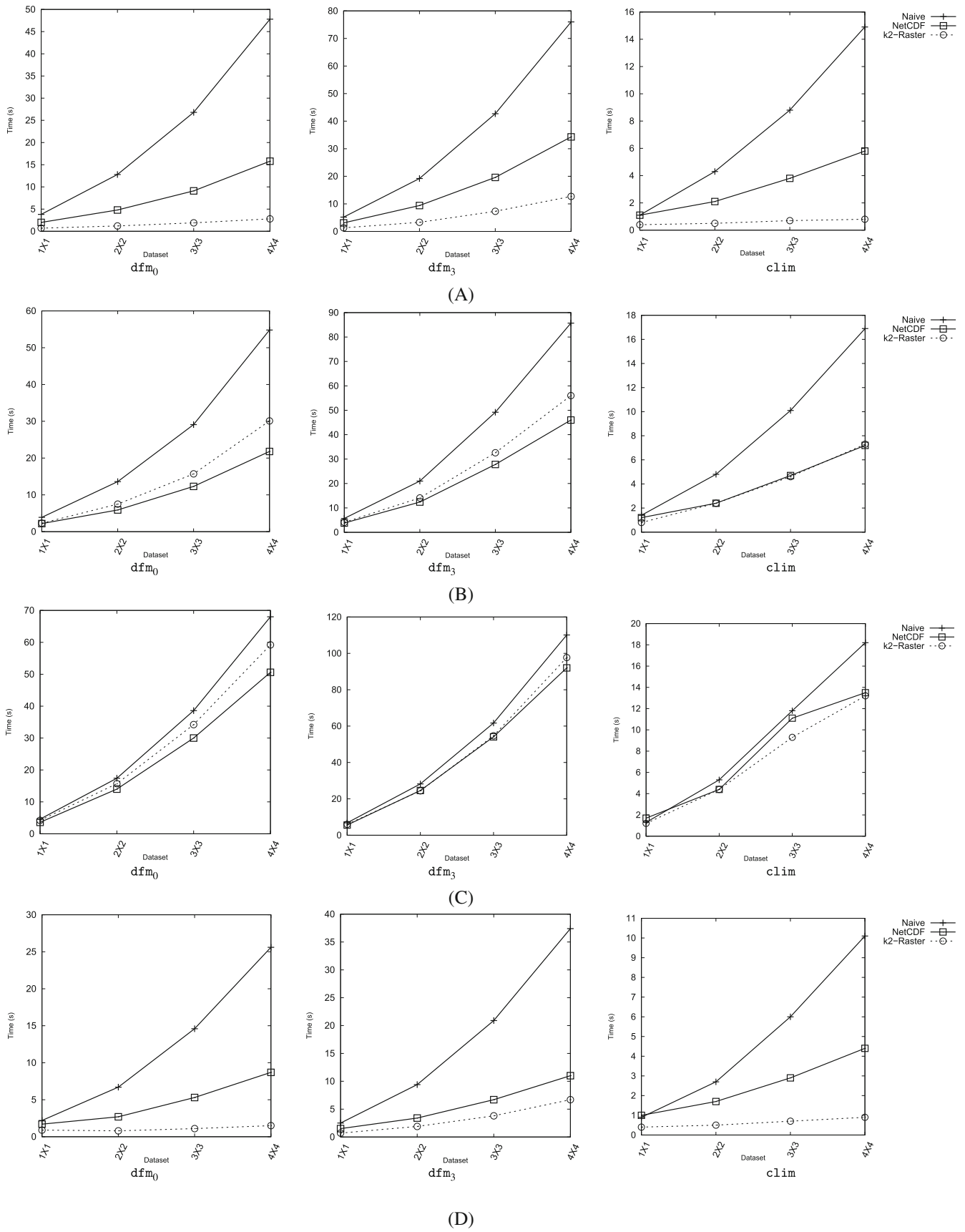


FIGURE 6 Time performance (in seconds) of (A) scalar sum, (B) scalar product, (C) point-wise sum, and (D) thresholding.

obtain some sort of direct access. However, DACs are precisely *Directly Addressable Codes* and any individual value can be decompressed separately.

- The third element that affects speed is that the k^2 -raster in an in-memory method, which favors k^2 -raster compared to NetCDF, a disk-based method.

Figure 6C shows the results of the point-wise sum operation. This is precisely the best example of the previous explanation. Here, the indexes of the k^2 -raster are not helpful since the two input rasters must be completely accessed. However, the in-memory approach of k^2 -raster balances the comparison, and, at the end, there is no clear winner.

Thresholding is precisely the other side of the coin, as shown in Figure 6D. Here, the indexes are capable of processing big parts of the input raster in the upper parts of the tree with just one step. That is, if we know that all cells in a subtree are below the given threshold, the algorithm simply sets them all to zero, without accessing them. Therefore, our algorithm is between 2.1 and 17.6 times faster than the naive approach and between 1.63 and 6 times faster than NetCDF, since both have to sequentially process cell by cell the complete input raster.

In zonal sum, the uniformity has even more impact. Recall that, by construction, as the raster is more uniform, the corresponding k^2 -raster is smaller. Thus, since operations require a top-down traversal, the time performance improves. However, in this operation, the uniformity of the zonal raster is even more critical since, in addition to the aforementioned effect, the algorithm produces an output close to that raster, and the second part of the algorithm takes time proportional to the number of zones.

This can be seen in Figure 7A,B. Figure 7A has a zonal raster with only 10 zones; thus, in the most uniform collections, `dfm0` and `clim`, our algorithm is able to outperform NetCDF. More concretely, it is between 1.11 and 1.34 times faster in `dfm0` and between 1.87 and 2 times faster in `clim`. However, in `dfm3`, our algorithm is between 1.07 and 1.37 times slower. We can see the impact of increasing the number of zones to 500 in Figure 7B. Now, our algorithm is slower in `dfm` collections, between 1.31 and 1.51 times slower, and it is on a par in `clim`. The effect of the number of zones of the zonal raster can be seen in Figure 7C, where we show the time versus number of zones with the collections of size 4×4 . In all cases, our algorithm outperforms the naive approach. With respect to NetCDF, again the uniformity is the key factor, and depending on it, one is better than the other.

5.3 | k^2 -acc experiment

k^2 -acc implementation has serious limitations to deal with datasets of moderate size, not only in the number of cells, but also in the number of different values in the dataset. Therefore, we only present results with the `dfm0` and `clim` datasets, as with `dfm3`, the k^2 -acc is not able to run or requires extremely long execution times.

Figure 8A,B show the scalar operations. In these operations the k^2 -acc can compete with k^2 -raster. The k^2 -acc is based on having one k^2 -tree per value present in the raster, which marks the cells where that value is present. Therefore, for each different value, that value and a k^2 -tree are stored. To solve these operations, only the values are modified, while the k^2 -trees remain unchanged. Namely, the arithmetic operation is applied to the values, and thus the cost of the operation is limited to an arithmetic operation for each different value.

However, in the case of the sum, k^2 -raster only has to perform two sums, one for the minimum value and another one for the maximum. This is constant independently of the number of different values in the raster; thus, as seen, k^2 -raster is around two-three times faster, except in the smallest dataset.

Nevertheless, in the case of the product, k^2 -raster has to traverse all the leaves, and thus, k^2 -acc is between 1.74 and 8.71 times faster.

However, as soon as the k^2 -trees must be accessed, k^2 -acc becomes extremely slow. In the case of the point-wise operations, Table 5 shows the results only for the datasets of size 1×1 , as with bigger datasets, k^2 -acc did not run. As seen, differences are of at least two orders of magnitude.

Thresholding is the other operation where the k^2 -acc can compete. In fact, it is very easy to solve it, as it only requires to recover the k^2 -tree representing the input threshold. Still, k^2 -raster is faster, as shown in Figure 8C, with the exception of the smallest datasets. Observe that k^2 -tree traverses the input tree with the help of the minimum and maximum values in the nodes, which allow the fill of large zones of the output tree with 0s or 1s in the upper levels of the tree; thus, the output tree will probably be a small tree. Still, this implies more processing than a simple copy of the k^2 -tree corresponding

to the threshold value. However, the k^2 -acc is harmed by its poor compression ratios, as seen in Tables 3 and 4, which implies slower load times in main memory than in the case of k^2 -raster.

In the case of the zonal sum, we only include experiments with 10 and 100 zones, as the k^2 -acc requires large running times. Again, since the values of the cells must be accessed, k^2 -acc has serious problems. Figure 9A,B show values where k^2 -raster can reach improvements of up to two orders of magnitude.

5.4 | Memory consumption

In this experiment, we considered resident memory. For memory consumption, the disk-based classical approach of loading one block at a time is the best scenario for NetCDF. In fact, the in-memory approaches try to avoid this traditional

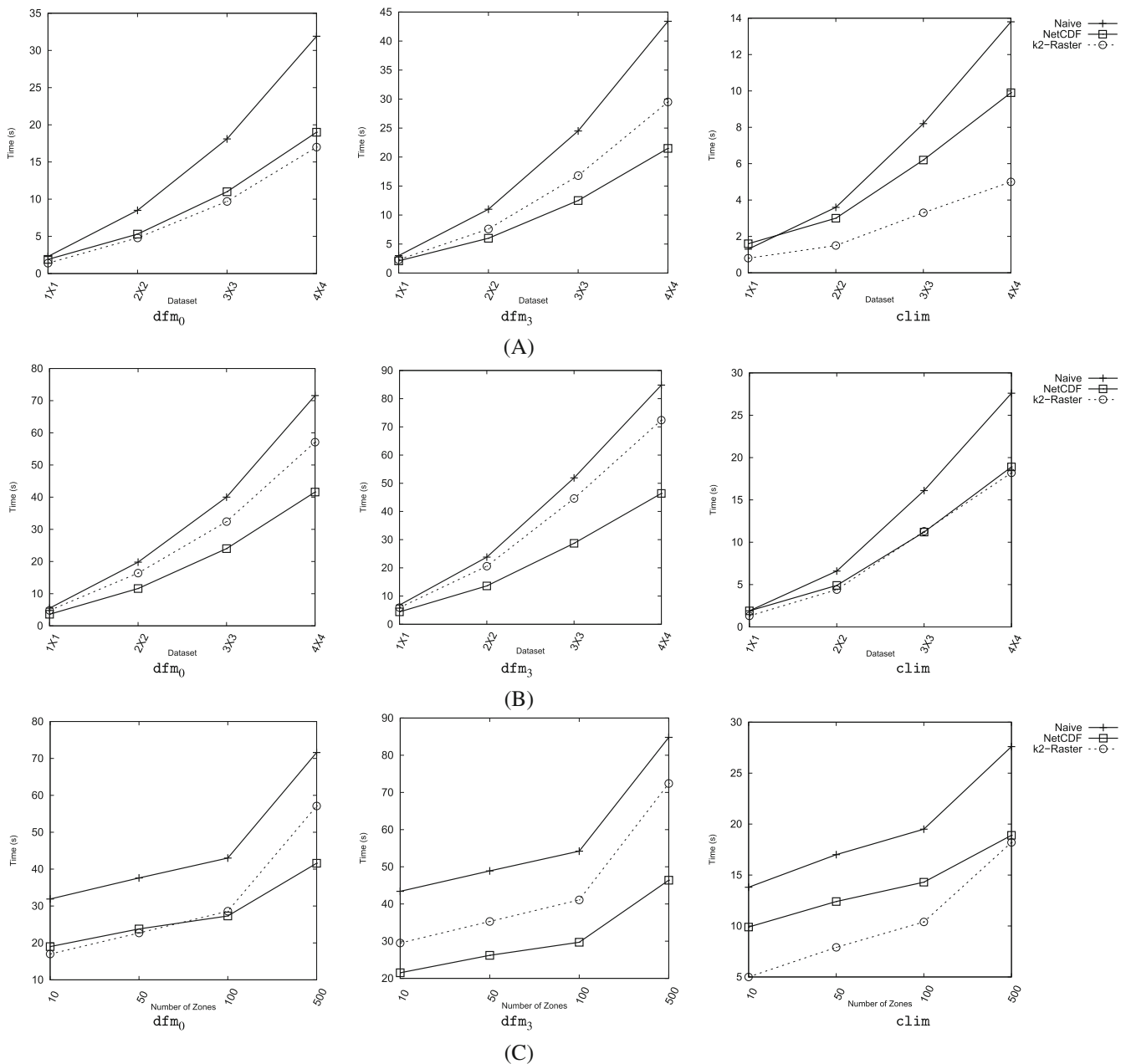


FIGURE 7 Time performance (in seconds) when performing zonal sum with different configurations. (A) Zonal sum with 10 zones; (B) zonal sum with 500 zones; (C) varying the number of zones over collections of size 4×4 .

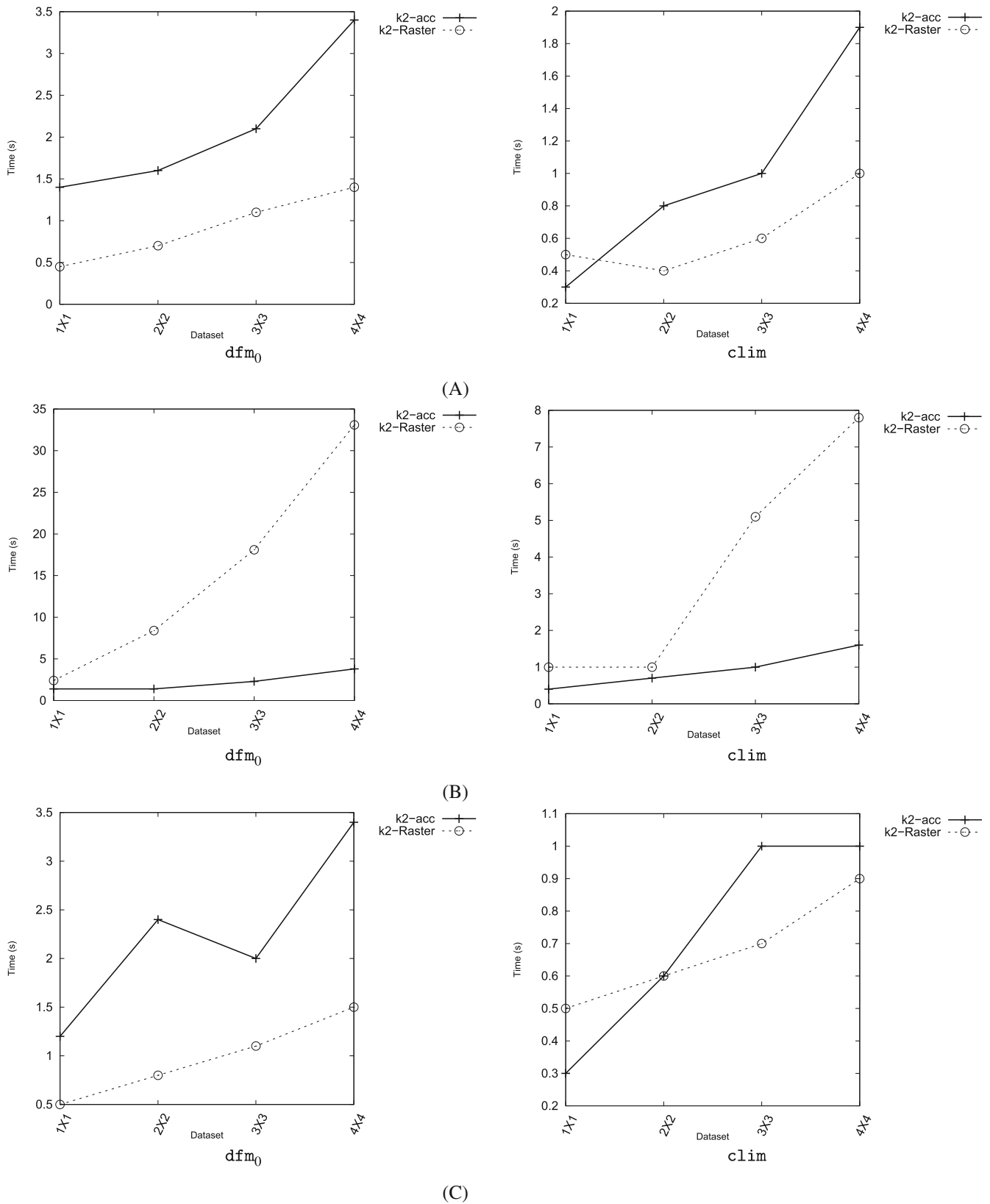


FIGURE 8 Time performance (in seconds) of (A) scalar sum, (B) scalar product, and (C) thresholding.

TABLE 5 Time performance (in seconds) for point-wise operations over two different collections of size 1×1 when using k^2 -acc and k^2 -raster.

Dataset	k^2 -acc	k^2 -raster
dfm ₀ 1x1	3246.6	4.3
clim 1x1	635.8	1.3

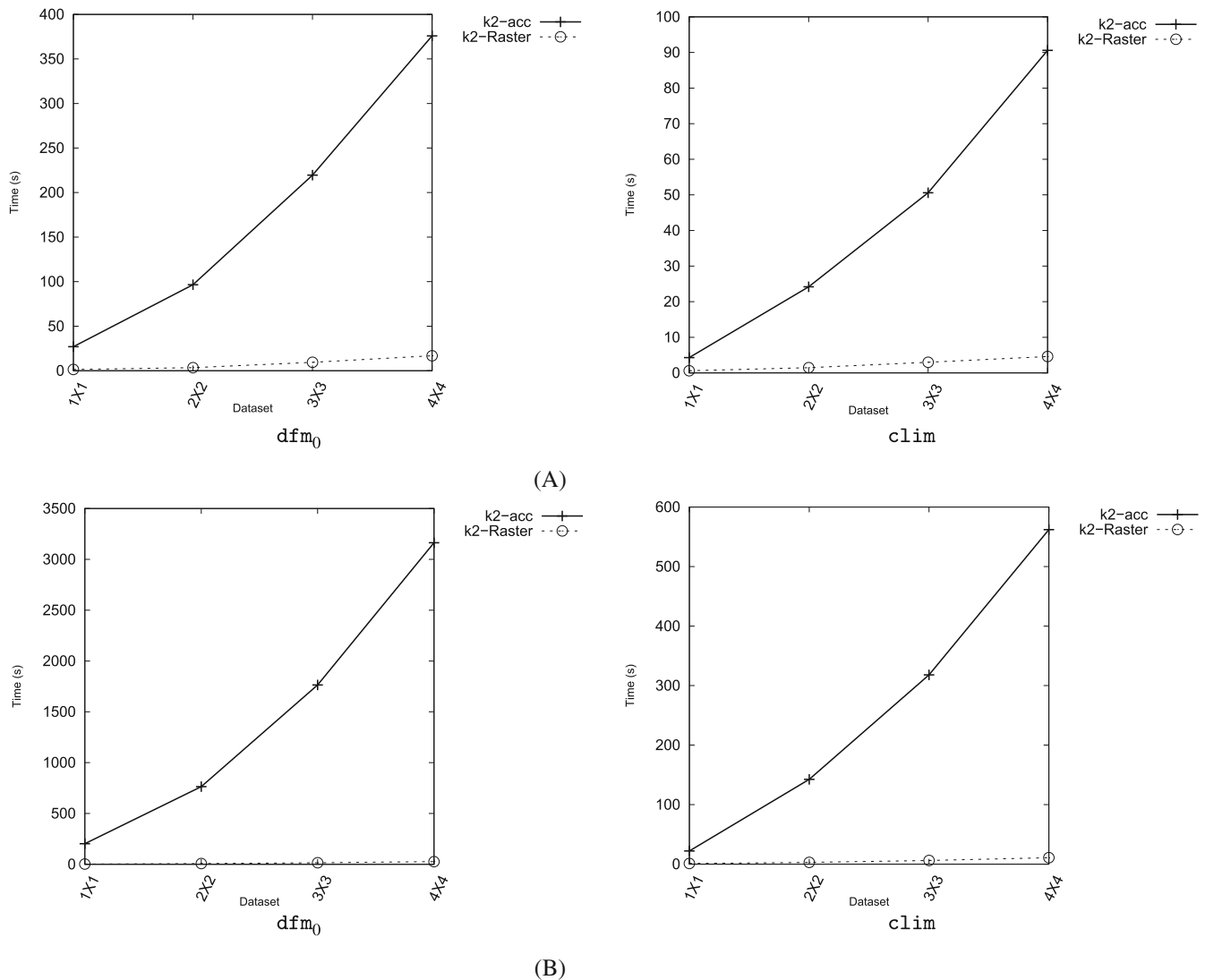


FIGURE 9 Time performance (in seconds) when performing the map operation zonal sum with different configurations. (A) Zonal sum with 10 zones; (B) zonal sum with 100 zones.

method derived from the low memory capacity of old computers. The decreasing in price of memories opens the opportunity of keeping all data in main memory all the time, thus taking advantage of the faster memory times, around 6 orders of magnitude faster. Nevertheless, compression is needed to be able to fit all data in main memory, which lowers that gap.

The k^2 -raster's original paper shows astonishing improvements in query times when its indexes are useful, whereas in this work, it shows variable results when all data must be sequentially processed and the indexes are useless. However, as a consequence of the change of approach, regarding memory consumption, see in Figures 10 and 11 that NetCDF is unbeatable. Still, we show that the algorithms presented here are noticeable better than the naive approach, therefore if

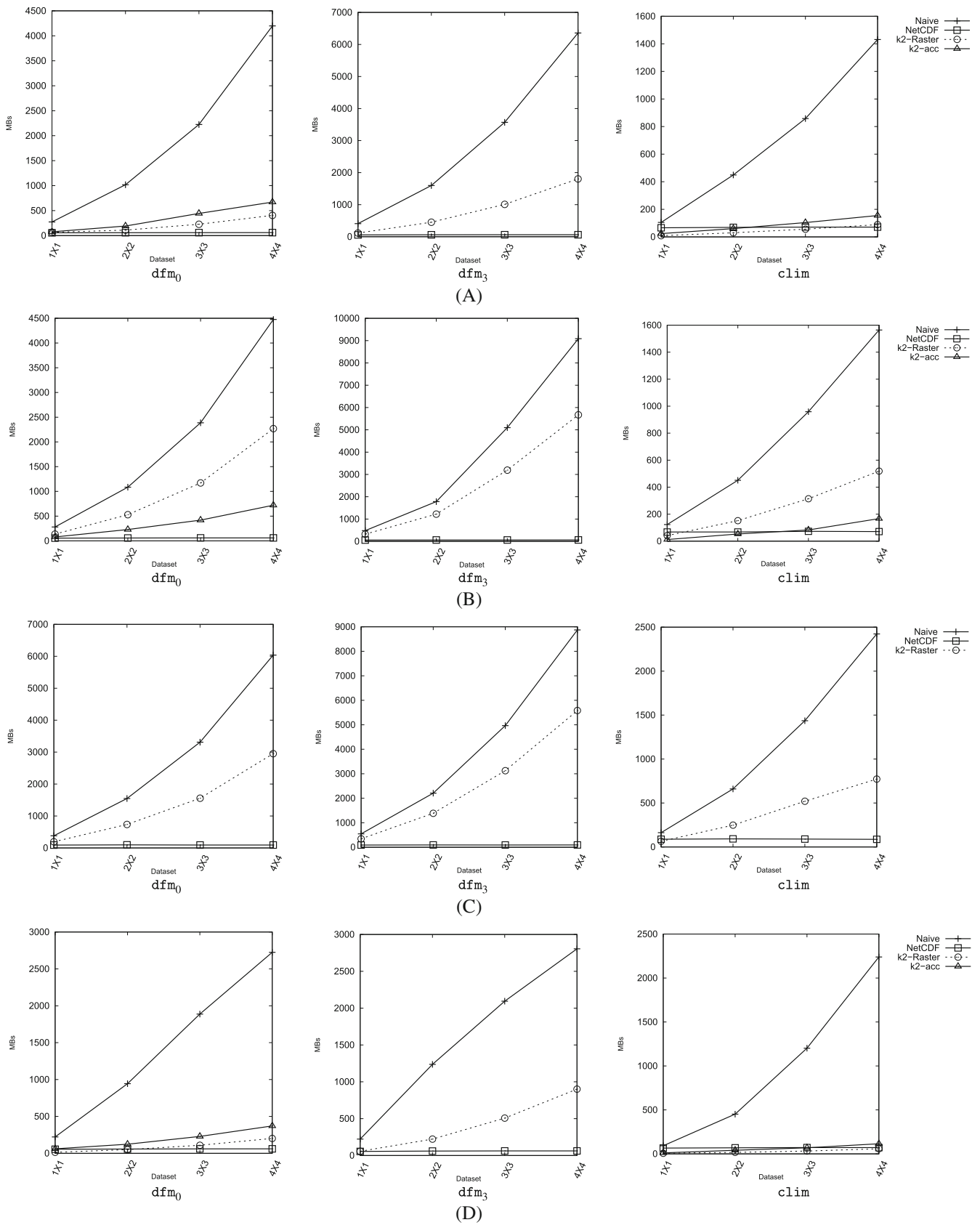


FIGURE 10 Memory consumption (in megabytes) of (A) scalar sum, (B) scalar product, (C) point-wise sum, and (D) thresholding.

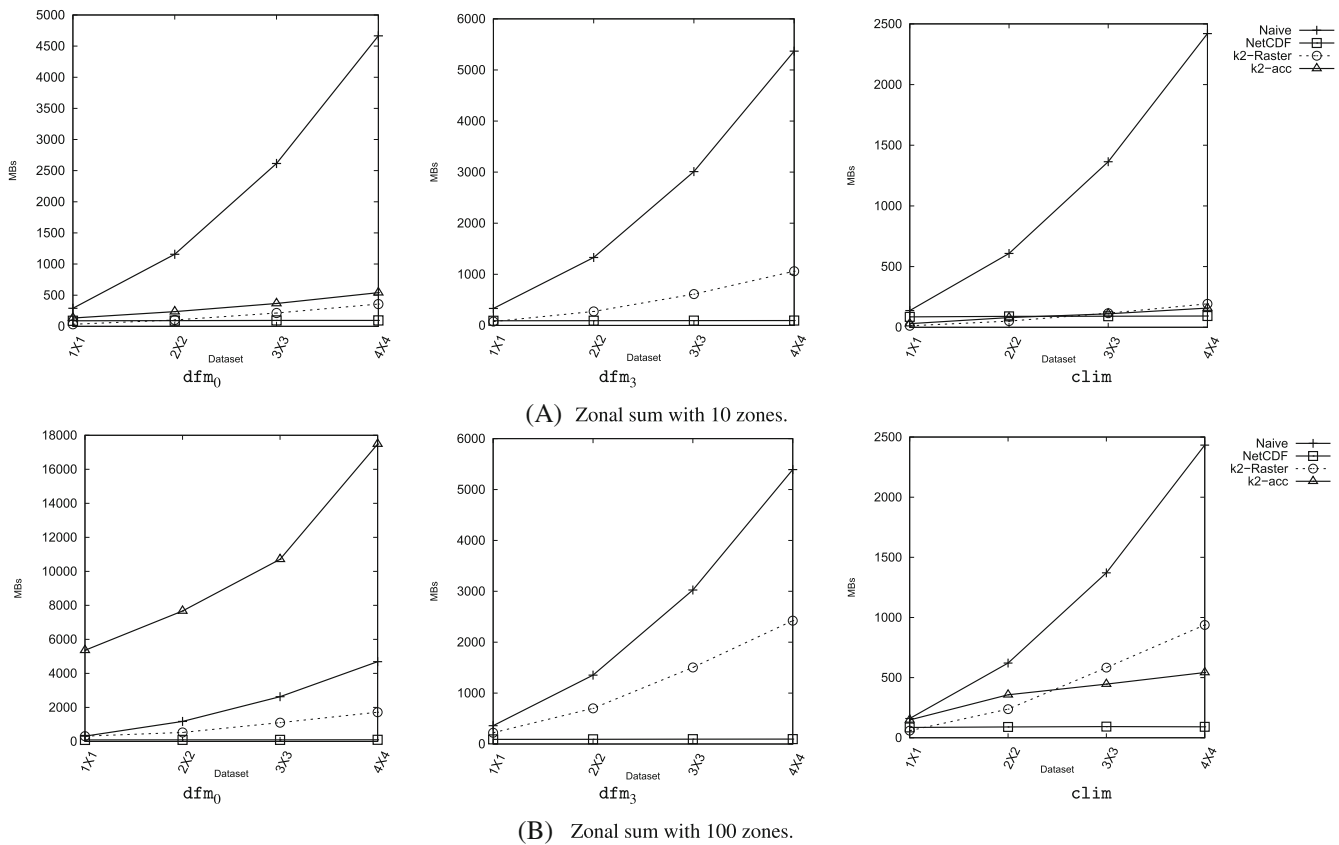


FIGURE 11 Memory consumption (in megabytes) when performing zonal sum with different configurations. (A) Zonal sum with 10 zones; (B) zonal sum with 100 zones.

one chooses the k^2 -raster as its storage method, our new algorithms obtain big memory savings when comparing to the naive ones.

With respect to the k^2 -acc, we do not include data from the point wise operation and, in the rest of operations, we do not include data of the dfm_3 dataset due to the problems of k^2 -acc running that operation and datasets. Unlike time, where the k^2 -raster is a landslide winner, in this parameter there is no clear winner.

6 | CONCLUSIONS AND FUTURE WORK

With compression power similar to that of NetCDF, k^2 -raster has been shown clearly superior when issuing the most common queries, namely, retrieving a region of the space and retrieving the cells in a region of the space with values in a given range. However, although less common, the capability for efficiently solving map algebra operations must be investigated to back the hypothesis of using the k^2 -raster as the method for representing rasters all the time, and more precisely in in-memory environments.

In this work, we have proposed efficient algorithms for performing map algebra operations over the compressed representation of the input rasters using k^2 -raster, and proved, in the experimental evaluation, that our algorithms are preferable to the naive approach of completely decompressing the rasters, running the operation with the uncompressed rasters, and then compressing the result, which is the key of an in-memory system. In addition, to complete the experimental evaluation, we have also included the classical method to store rasters, that is, NetCDF and a previous compact data structure for storing rasters, k^2 -acc.

In the case of NetCDF, there is no overall winner, as there are operations where k^2 -raster obtains the best performance, and there are other operations for which NetCDF outperforms the proposed algorithms using k^2 -rasters.

The proposed algorithms using k^2 -rasters obtain the best performance for summing/subtracting by a scalar and thresholding. In the first case, the k^2 -raster benefits from its structural properties and how it compactly represents the data, solving that operation in constant time. In the second case, the k^2 -raster takes advantages of its indexes. However, the proposed algorithms are not capable of clearly overcome NetCDF when the rasters must be processed sequentially, such as for the point-wise operation of zonal sum. In those cases, the algorithms do not use the indexes of the k^2 -rasters; thus, traversing their tree shape is less efficient than processing the array-like structure of the NetCDF, and only the in-memory approach of k^2 -raster is able to balance the times.

In the case of k^2 -acc, k^2 -raster is the clear winner in time. With much better compression power, it only loses in one operation, the product by a scalar, but in those operations where the k^2 -acc has to determine the value of the cells, k^2 -raster is orders of magnitude faster.

We did not include focal operations, as these operations can be solved with the window query shown in the k^2 -raster's original paper. When the focal operation only affects to one cell, such as computing the slope of the terrain around a cell, that operation will be very efficient, since the k^2 -raster is very fast extracting a portion of the data (window query) given that it is in essence a quadtree, that is, a spatial index. However, when the operation affects to all cells of the raster, such as computing the maximum, minimum, average and so forth of all cells within an area around each cell, the best alternative is to apply the window query to the complete raster, as it does not make any sense to issue a different window query for each cell. This is precisely the naive baseline presented in our experiments, which can be used in any operation, as those cases where a native k^2 -raster algorithm would be inefficient.

As a summary, putting in a balance the pros and the cons, k^2 -raster remains as a better choice: it obtains almost the same compression performance as NetCDF and it is much better when issuing common queries,^{23,25} other complex operations,²⁶ and, as proved in this work, also some of the map algebra operations, and when it is not the clear winner, the performance is similar to that of NetCDF. As future work, we plan to analyze the parallelization of the proposed map algebra implementation, as many of these algorithms are clearly parallelizable.

ACKNOWLEDGMENTS

This work was partially supported by CITIC, CITIC is funded by the Xunta de Galicia through the collaboration agreement between the Department of Culture, Education, Vocational Training and Universities and the Galician universities for the reinforcement of the research centers of the Galician University System (CIGUS). IN852D 2021/3(CO3): partially funded by UE, (ERDF), GAIN, convocatoria Conecta COVID. GRC: ED431C 2021/53: partially funded by GAIN/Xunta de Galicia. TED2021-129245B-C21; PDC2021-121239-C31; PDC2021-120917-C21: partially funded by MCIN/AEI/10.13039/501100011033 and "NextGenerationEU"/PRTR. PID2020-114635RB-I00; PID2019-105221RB-C41: partially funded by MCIN/AEI/10.13039/501100011033. Funding for open access charge: Universidad de Coruña/CISUG.

AUTHOR CONTRIBUTIONS


Fernando Silva-Coira: Conceptualization, Investigation, Software, Writing - original draft, Writing - review & editing. **José R. Paramá:** Conceptualization, Supervision, Writing - original draft, Writing - review & editing. **Susana Ladra:** Conceptualization, Formal Analysis, Writing - original draft, Writing - review & editing.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available in <https://lbd.udc.es/research/k2-raster/>. These data were derived from the following resources available in the public domain: WorldClim: <https://www.worldclim.org/>. Spanish Geographic Institute: <https://www.ign.es>.

ORCID

Fernando Silva-Coira  <https://orcid.org/0000-0003-1341-3368>

José R. Paramá  <https://orcid.org/0000-0002-8727-0980>

Susana Ladra  <https://orcid.org/0000-0003-4616-0774>

REFERENCES

- Li Y, Ma J, Zhang Y. Image retrieval from remote sensing big data: a survey. *Inf Fusion*. 2021;67:94-115. doi:10.1016/j.inffus.2020.10.008

2. Eldawy A, Mokbel MF. SpatialHadoop: a MapReduce framework for spatial data. Proceedings of the 2015 IEEE 31st International Conference on Data Engineering; 2015: 1352-1363.
3. You S, Zhang J, Gruenwald L. Large-scale spatial join query processing in cloud. Proceedings of the 2015 31st IEEE International Conference on Data Engineering Workshops; 2015: 34-41.
4. Zhang H, Chen G, Ooi BC, Tan KL, Zhang M. In-memory big data management and processing: a survey. *IEEE Trans Knowl Data Eng.* 2015;27(7):1920-1948. doi:10.1109/TKDE.2015.2427795
5. Sikka V, Färber F, Lehner W, Cha SK, Peh T, Bornhövd C. Efficient transaction processing in SAP HANA database: the end of a column store myth. Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. New York: Association for Computing Machinery; 2012: 731-742.
6. Lemke C, Sattler KU, Faerber F, Zeier A. Speeding up queries in column stores. In: Bach Pedersen T, Mohania MK, Tjoa AM, eds. *Data Warehousing and Knowledge Discovery. 6263 of Lecture Notes in Computer Science.* Springer; 2010:117-129.
7. Funke F, Kemper A, Neumann T. Compacting transactional data in hybrid OLTP&OLAP databases. *Proc VLDB Endow.* 2012;5(11):1424-1435. doi:10.14778/2350229.2350258
8. Kou W. *Digital Image Compression: Algorithms and Standards.* Kluwer Pub; 1995.
9. Wallace GK. The JPEG still picture compression standard. *Commun ACM.* 1991;34(4):30-44.
10. Ritter N, Ruth M. The GeoTiff data interchange standard for raster geographic images. *Int J Remote Sens.* 1997;18(7):1637-1647.
11. Salomon D. *Data Compression: The Complete Reference.* Springer; 2006.
12. Navarro G. *Compact Data Structures: A Practical Approach.* Cambridge University Press; 2016.
13. Claude F, Navarro G. A fast and compact web graph representation. In: Ziviani N, Baeza-Yates R, eds. *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE). 4726 of Lecture Notes in Computer Science.* Springer; 2007:105-116.
14. Brisaboa N, Ladra S, Navarro G. K2-trees for compact web graph representation. In: Karlgren J, Tarhio J, Hyvärö H, eds. *Proceedings of the 16th International Symposium on String Processing and Information Retrieval. 5721 of Lecture Notes in Computer Science.* Springer; 2009:18-30.
15. Brisaboa N, Ladra S, Navarro G. Compact representation of web graphs with extended functionality. *Inf Syst.* 2014;39:152-174.
16. Claude F, Navarro G. Practical rank/select queries over arbitrary sequences. In: Amir A, Turpin A, Moffat A, eds. *Proceedings of the 15th International Symposium on String Processing and Information Retrieval. 5280 of Lecture Notes in Computer Science.* Springer; 2009:176-187.
17. Navarro G. Spaces, trees, and colors: the algorithmic landscape of document retrieval on sequences. *ACM Comput Surv.* 2014;46(4):52:1-52:47.
18. Navarro G. Wavelet trees for all. *J Discret Algorithms.* 2014;25:2-20.
19. Brisaboa N, Ladra S, Navarro G. DACs: bringing direct access to variable-length codes. *Inf Process Manag.* 2013;49(1):392-404.
20. Brisaboa N, Luaces M, Navarro G, Seco D. Space-efficient representations of rectangle datasets supporting orthogonal range querying. *Inf Syst.* 2013;38(5):635-655.
21. Bernardo G, Álvarez-García N, Navarro G, Pedreira O. Compact queryable representations of raster data. In: Kurland O, Lewenstein M, Porat E, eds. *Proceedings of the 20th International Symposium on String Processing and Information Retrieval. 8214 of Lecture Notes in Computer Science.* Springer; 2013:96-108.
22. Brisaboa NR, Cerdeira-Pena A, Bernardo G, Navarro G, Pedreira O. Extending general compact queryable representations to GIS applications. *Inf Sci.* 2020;506:196-216.
23. Ladra S, Paramá JR, Silva-Coira F. Scalable and queryable compressed storage structure for raster data. *Inf Syst.* 2017;72:179-204.
24. Pinto A, Seco D, Gutiérrez G. Improved queryable representations of rasters. Proceedings of the 2017 Data Compression Conference (DCC'17); 2017: 320-329.
25. Ladra S, Paramá J, Silva-Coira F. Compact and queryable representation of raster datasets. Proceedings of the 28th International Conference on Scientific and Statistical Database Management; 2016: 15:1-15:12.
26. Silva-Coira F, Paramá JR, Ladra S, López JR, Gutiérrez G. Efficient processing of raster and vector data. *PLoS One.* 2020;15(1):1-35.
27. Tomlin C, Berry J. Mathematical structure for cartographic modeling in environmental analysis. Proceedings of the American Congress on Surveying and Mapping, Falls Church, VA; 1979: 269-283.
28. Tomlin C. A map algebra. Proceedings of the Harvard Computer Graphics Conference; 1983.
29. Esri. Introduction to using Map Algebra in Spatial Analyst. Accessed November 2, 2022. <https://pro.arcgis.com/en/pro-app/latest/help/analysis/spatial-analyst/mapalgebra/a-quick-tour-of-using-map-algebra.htm>
30. Grass. r.mapcalc; 2022. Accessed November 2, 2022. <https://grass.osgeo.org/grass80/manuals/r.mapcalc.html>
31. Caniupán M, Torres-Avilés R, Gutiérrez-Bunster T, Lepe M. Efficient computation of map algebra over raster data stored in the k²-acc compact data structure. *GeoInformatica.* 2021;26:95-123.
32. Berry KJ. A mathematical structure for analyzing maps. *Environ Manag.* 1987;11(3):317-325.
33. Tomlin C. *Geographic Information Systems and Cartographic Modelling.* Prentice Hall; 1990.
34. Bosque J. *Sistemas de Información Geográfica.* RIALP; 1992.
35. Mennis J, Viger R, Tomlin CD. Cubic Map Algebra Functions for Spatio-Temporal Analysis. *Cartogr Geogr Inf Sci.* 2005;32(1):17-32.
36. Lee C, Yang M, Ayd R. NetCDF-4 performance report. Technical report, HDF Group; 2008.
37. Deutsch LP. RFC 1951: DEFLATE compressed data format specification version 1.3; 1996.
38. Munro JI. Tables. In: Chandru V, Vinay V, eds. *Proceedings of the 16th Conference Foundations of Software Technology and Theoretical Computer Science (FSTTCS).* Springer; 1996:37-42.

39. Vigna S. Broadword implementation of rank/select queries. In: CC MG, ed. *WEA 2008: Experimental Algorithms. 5038 of Lecture Notes in Computer Science*. Springer; 2008:154-168.
40. Samet H. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann; 2006.
41. Jacobson G. Space-efficient static trees and graphs. *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (SFCS'89)*; 1989: 549-554.
42. Klinger A. *Pattern and Search Statistics*. Academic Press; 1971.
43. Klinger A, Dyer CR. Experiments on picture representation using regular decomposition. *Comput Graph Image Process*. 1976;5:68-105. doi:[10.1016/S0146-664X\(76\)80006-8](https://doi.org/10.1016/S0146-664X(76)80006-8)
44. Gargantini I. An effective way to represent quadtrees. *Commun ACM*. 1982;25(12):905-910.
45. Zhang J, You S. Supporting web-based visual exploration of large-scale raster geospatial data using binned min-max quadtree. In: Gertz M, Ludäscher B, eds. *Scientific and Statistical Database Management*. Springer; 2010:379-396.
46. Zhang J, You S. High-performance quadtree constructions on large-scale geospatial rasters using GPGPU parallel primitives. *Int J Geogr Inf Sci*. 2013;27:2207-2226. doi:[10.1080/13658816.2013.828840](https://doi.org/10.1080/13658816.2013.828840)
47. Woodruff JR. Compressed quad trees. *Comput J*. 1984;27:225-229. doi:[10.1093/comjnl/27.3.225](https://doi.org/10.1093/comjnl/27.3.225)
48. Lin TW. Compressed quadtree representations for storing similar images. *Image Vis Comput*. 1997;15:833-843. doi:[10.1016/S0262-8856\(97\)00031-0](https://doi.org/10.1016/S0262-8856(97)00031-0)
49. Chan YK. Block image retrieval based on a compressed linear quadtree. *Image Vis Comput*. 2004;22:391-397. doi:[10.1016/j.imavis.2003.12.003](https://doi.org/10.1016/j.imavis.2003.12.003)
50. Chung KL, Liu YW, Yan WM. A hybrid gray image representation using spatial- and DCT-based approach with application to moment computation. *J Vis Commun Image Represent*. 2006;17:1209-1226. doi:[10.1016/j.jvcir.2006.01.002](https://doi.org/10.1016/j.jvcir.2006.01.002)
51. Zhang J, You S, Gruenwald L. Quadtree-based lightweight data compression for large-scale geospatial rasters on multi-core CPUs. *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)*; 2015: 478-484.
52. Graziosi D, Nakagami O, Kuma S, Zaghetto A, Suzuki T, Tabatabai A. An overview of ongoing point cloud compression standardization activities: video-based (V-PCC) and geometry-based (G-PCC). *APSIPA Trans Signal Inf Process*. 2020;9:e13. doi:[10.1017/ATSIP.2020.12](https://doi.org/10.1017/ATSIP.2020.12)
53. Luo H, Wang J, Liu Q, Chen J, Klasky S, Podhorszki N. zMesh: exploring application characteristics to improve lossy compression ratio for adaptive mesh refinement. *Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*; 2021: 402-411.
54. Isenburg M. LASzip: lossless compression of LiDAR data. *Photogramm Eng Remote Sens*. 2013;79(2):209-217. doi:[10.14358/PERS.79.2.209](https://doi.org/10.14358/PERS.79.2.209)
55. Strafella L, Chapon D. LightAMR format standard and lossless compression algorithms for adaptive mesh refinement grids: RAMSES use case. *J Comput Phys*. 2022;470:111577. doi:[10.1016/j.jcp.2022.111577](https://doi.org/10.1016/j.jcp.2022.111577)
56. Brisaboa N, Bernardo G, Gutiérrez G, Ladra S, Penabad M, Troncoso B. Efficient set operations over k2-trees. *Proceedings of the 2015 Data Compression Conference (DCC'15)*; 2015: 373-382.
57. Quijada Fuentes C, Penabad MR, Ladra S, Gutiérrez RG. Compressed data structures for binary relations in practice. *IEEE Access*. 2020;8:25949-25963.
58. Gog S, Beller T, Moffat A, Petri M. From theory to practice: plug and play with succinct data structures. *Proceedings of the International Symposium on Experimental Algorithms (SEA)*; 2014: 326-337.
59. Hijmans RJ, Cameron SE, Parra JL, Jones PG, Jarvis A. Very high resolution interpolated climate surfaces for global land areas. *Int J Climatol*. 2005;25:1965-1978.

How to cite this article: Silva-Coira F, Paramá JR, Ladra S. Map algebra on raster datasets represented by compact data structures. *Softw Pract Exper*. 2023;53(6):1362-1390. doi: [10.1002/spe.3191](https://doi.org/10.1002/spe.3191)