# A heuristic to repartition large multi-dimensional arrays with reduced disk seeking

Timothée Guédon

A thesis
in
The Department
of
Department of Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of science in computer science
Concordia University
Montréal, Québec, Canada

December 2020

© Timothée Guédon, 2021

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Timothée Guédon**

Entitled: **A heuristic to repartition large multi-dimensional arrays with reduced disk seeking**

and submitted in partial fulfillment of the requirements for the degree of

**Master of science in computer science**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________________ Chair

*Dr. Essam Mansour*

______________________________ Examiner

*Dr. Essam Mansour*

______________________________ Examiner

*Dr. Hovhannes Harutyunyan*

______________________________ Supervisor

*Dr. Tristan Glatard*

Approved by ______________________________
Dr. Leila Kosseim, Graduate Program Director

December 17, 2020 ______________________________
Mourad DEBBABI, Ph.D., Computer Science, Dean
Faculty of Engineering and Computer Science

# Abstract

A heuristic to repartition large multi-dimensional arrays with reduced disk seeking

Timothée Guédon

Multi-dimensional arrays have become critical scientific data structures, but their manipulation raises performance issues when they exceed memory capacity. In particular, accessing specific array regions can require millions to billions of disk seek operations, with important consequences on I/O performance. While traditional approaches to address this problem focus on file format optimizations, we are searching for algorithmic solutions where applications control I/O to reduce seeking. In this thesis, we propose the keep heuristic to minimize the number of seeks required for the repartitioning of large multi-dimensional arrays. The keep heuristic uses a memory cache to reconstruct contiguous data sections in memory. We evaluate it on arrays of size 85.7 GiB with memory amounts ranging from 4 to 275 GiB. Repartitioning time is reduced by a factor of up to 2.5, and seeking is reduced by four orders of magnitude. Due to the effect of asynchronous writes to memory page cache enabled by the Linux kernel, speed up is only observed for arrays that exceed the size of working memory. The keep heuristic could be applied in platforms that manipulate large data arrays, as is commonly the case in scientific imaging.

# Acknowledgments

I would like to thank Professor Tristan Glatard, my supervisor, for his invaluable support during my master's degree. First of all, I would like to thank him for giving me the opportunity to join his laboratory and do a thesis under his supervision. I would also like to thank him for his help and guidance throughout the thesis. I would then like to thank Valérie Hayot-Sasson for her help as a collaborator on my thesis, but also for her support in academic life at Concordia University. She gave me valuable advice and was always willing to give me a critical and informed opinion on my work. Finally, I wish to thank Christelle Bardelli, my partner in life who gave time and energy to help me reformulate my ideas, practice my speaking skills and get through difficult times.

The computing platform was obtained with funding from the Canada Foundation for Innovation.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Scientific research and applications produce massive amounts of data, generally represented as multidimensional arrays [4]. Some scientific domains manipulating multidimensional arrays include astrophysics, biology, linguistics and economy [10]. For example, images and physical phenomena are two types of data represented as arrays. Since 2012, the amount of stored and treated data has exploded, resulting in a new numeric era called Big Data [21, 22]. Big Data not only refers to the massive amount of data produced but also to the diversity and size of such data [22]. The Big Data era has been enabled by diverse factors such as the increased capacity to store data, new acquisition techniques and new hardware for efficient data processing [21]. Manipulating files that are too big to fit in main memory is also considered Big Data. BigBrain [3], for instance, is a human brain model represented as a big image of size 1TB. Big Brain provides microscopic data at the resolution of 20 micrometres that comes in the form of many 3D blocks.

The predominance of multidimensional arrays in science in the Big Data era has made multidimensional array processing a major challenge nowadays. In particular, the unordered model used by traditional relational databases have become inefficient at processing a large amount of scientific data optimally [21]. Moreover, scientific datasets are often too big in size to fit on a single computer and must be stored and processed on clusters or in the cloud [21]. Although it exists since the '90s, the research on multidimensional array processing remains active and regularly produces new innovations [21, 13, 19].

Multidimensional array chunking is a common practice to split a big array into several files for efficient storage and processing [21]. Previous work showed that it is possible to optimize array splitting (in regular chunks) and merging by reducing the number of seeks using sequential algorithms [11]. In this master thesis, we focused on the optimization of sequential multidimensional array re-chunking/resplitting from chunks of a given shape into chunks of another shape. We call this problem the repartitioning problem. In particular, we present the keep heuristic, an algorithm that reduces the number of seeks created when repartitioning large multi-dimensional arrays in blocks of arbitrary but regular shape. Repartitioning may be triggered either by application requirements or for performance reasons, to improve memory usage and I/O efficiency. In this study we focus on repartitioning 3D arrays, even though our work can be extended to N dimensions. Our motivating use case was to enable faster processing of big 3D or 4D arrays in the neuroscience field.

The keep heuristic leverages a memory cache to read and write data as contiguously as possible, similarly to the "clustered" and "multiple" algorithms described in [11]. The storage order of array elements on disk is assumed to be known to the application, but unconstrained. Implementations based on our algorithm could therefore use arbitrary file formats. We focus on sequential algorithms, assuming that arrays are stored on a single device and accessed by single-threaded I/Os. Extensions to parallel environments are part of our future work. Although in practice seek times depend on various factors, such as the distance between the initial and target drive positions, we focus on minimizing seek *numbers*, for simplicity. Likewise, the effects of I/O optimizations such as page caching or readahead will be discussed experimentally but not modelled.

Chapter 2 presents the necessary background to understand the challenges in array storage and processing and the need for repartitioning tools. Chapter 3 then presents our work, the repartitioning problem and our solution, the "keep" heuristic. Finally, the conclusion chapter summarizes our work and main results, together with future work.

# Chapter 2

# Background

In this chapter we present common ways to store and manipulate multidimensional arrays. We then discuss common solutions and how optimizing array partitioning can be useful.

## 2.1 Multidimensional arrays storage

The survey in [21] provides an excellent summary of array storage, which we summarize in this section.

### 2.1.1 Storing arrays in one file

The easiest way to store arrays is in a single file, on disk. When part of the array is needed for computations, the required part is read from the whole array. The reading time depends on how contiguously the data part is stored on the disk together with the type of disk [21]. Indeed, if the data to be retrieved are scattered in the file, the computer will have to seek, i.e. move the reading head between each data part, which will introduce delays.

If the file is stored on a hard disk drive (HDD), this delay in moving the read head will quickly become significant, whereas if the file is stored on a solid state drive (SSD) it will take more seeks to make a significant difference in the array processing. In particular, the delay introduced depends on the way the data is written on disk and on the query shape [21]. Indeed, although the array is multidimensional, the data is still linearly written on disk. For example, the cells indexed $(0, 0, 0)$ and $(0, 0, 1)$ in 3D could be written linearly at index 0 and 1 on the disk.

The two main ways to store data on the disk are the "row-major" and "column-major" orders [21]. The row-major storage order for example consists in writing a matrix on the disk starting with the dimension that evolves the fastest, i.e. in the order $(i, j, k)$ in 3D, with $i$, $j$ and $k$ the first, second and third dimensions of the array, respectively. The column-major order consists in writing a matrix in the reverse order $(k, j, i)$.

When a multidimensional array is too large to fit in memory, one can use "memory mapping" which consists in creating a virtual object representing the data array in main memory but which will only read the data from the disk when these data are required for the calculation [2]. Although memory mapping allows to process arrays that are too large to fit in memory, the program still has to read the data from the disk each time, which makes it an expensive process.

### 2.1.2 Multidimensional array chunking

Multidimensional arrays are rarely stored in a single file, but are rather divided into subparts called chunks, such that all the chunks form a partition of the original array [21]. This storage mode allows many optimizations such as parallel processing, simultaneous reading of several data parts, storage redundancy or faster query response [5].

Several questions arise when one decides to do chunking [21]:

- What size, in terms of the number of cells, should we assign to the chunks?
- What shape should we choose for the chunks, i.e. the size of each chunk in each dimension?
- What function is used to match each cell to its chunk ?
- How do we write data into each chunk?

Although in the early uses of chunking, chunks used to be of the size of a page of the file system (so that a chunk would not be scattered due to disk partitioning), we now find chunks of increasingly larger sizes, nowadays ranging from the megabyte to a hundred megabytes [21, 19]. There are two ways to index the chunks in relation to each other: the implicit way consists in using a formula that will transform the position of the chunk in space into a 1-dimensional position, i.e. into a linear index, while the explicit method makes each position in space correspond to a "hard-coded" linear index. The row-major and column-major orders and their N-dimensional extensions are among the simplest functions. Space-filling curves are a slightly more sophisticated method of indexing in space. A space-filling curve has the property of passing through all the points of a multidimensional space without ever crossing itself [21, 15, 29]. Commonly used space-filing curves are shown on Figure 1. It is a method for indexing data points in space (chunks here) such that data points that are close in space have close indices. In particular, it can be used for faster data reading: Good indexing methods can reduce the number of chunks read, on average, for any given query polyhedral shape.

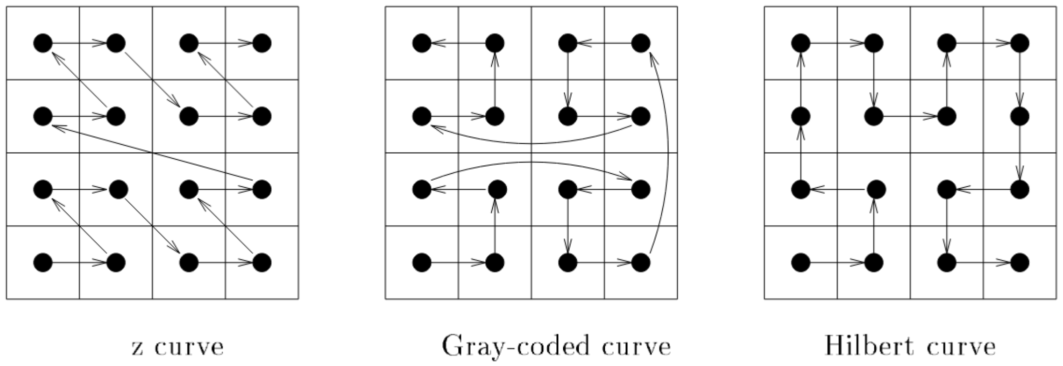


Figure 1: Illustration of commonly used space-filling curves [15]

Apart from arbitrary chunking in which one decides of a unique and uniform shape for the chunks, aligned on the axes of the array, there is a statistical way to find an optimal chunk shape. This method called "workload-based chunking" was developed in response to the observation that access patterns to a chunked array depend on the application, which means that it is not possible to find a configuration of the chunks that would be optimal for all applications. The principle of workload-based chunking is to minimize the number of chunks accessed for the majority of queries [21].

Finally, if the storage medium is a cluster (several machines clustered together whose resources are usually distributed for different incoming tasks and users by a scheduler), once the chunking process is defined, a decision must be made on how to distribute the chunks on the different disks, which is called de-clustering [21]. The optimal way to store chunks seems to be to separate the chunks that are usually required at the same time, in order to optimize bandwidth and improve parallel computing [21]. To do so, several strategies exist, including the round robin method, range-based partitioning and pseudo-randomization. The round robin method assigns a chunk to each disk and then starts again, the slice method works in the same way but distributes blocks of consecutive chunks instead of one chunk at a time, and the pseudo-random method randomly distributes the chunks on the different disks. More advanced methods can be used too, such as methods based on graph theory. The idea is to represent each chunk by a node in a complete graph, i.e. each node is connected to all the others. The edges are weighted by the probability that the nodes they join are requested at the same time. From this graph, we then cut the graph into subgraphs containing chunks that have low probability to be retrieved together. This problem is equivalent to the max-cut/min-cut problem in graph theory. This problem being NP complete, the solutions are therefore heuristics as there is no absolute solution.

As noted by [21], although the minimum part to be recovered from the disk is an entire chunk, when the chunk is in memory the cell(s) of interest still need to be accessed. Even if the main bottleneck is the read/write time from/to disk, repeated searches in RAM can also become significant [21]. An example of optimization for the storage of cells within a chunk is the storage of a sparse array by key/value pairs. We call recursive chunking the application of the same strategies used for chunking within the chunks themselves [21]. For example, one could store data in the file in an indexed way using space-filling curves.

## 2.2 File formats

To store and manipulate arrays, a multitude of file formats have emerged. For example, the NumPy [10] and the HDF5 [9] file formats are specialized file formats for storing multidimensional arrays. Some scientific fields also have their own file formats: NIfTI, MINC and DICOM are some examples of file formats used to store medical images [28]. The multitude of existing file formats can be explained in part by the specific needs of each field, which for example have specific requirements for storing metadata, by the lack of standards, or by attempts to optimize processing times by storing files in an intelligent way [28].

The HDF5 (Hierarchical Data Format version 5) file format is an example of a specialized file format for processing arrays [9]. Storage is simple because it follows the hierarchical structure of a file system. Two types of objects can be created from the root; Datasets or Groups. A Group contains one or more Groups/Datasets, while Datasets contain data [28, 9]. HDF5 allows parallel array processing and array chunking. As HDF5 also supports compression, one can choose to losslessly compress/decompress a target chunk instead of the whole array.

MINC2.0 is an example of a specialized file format for a scientific field [28]. It is an open source file format based on HDF5. It defines the use of HDF5 file format so that researchers follow a standard for storing their data. MINC2.0 was developed for neuroscientists, in the context of human brain mapping, but is now also used in many medical applications. The features sought during the development of MINC include; an extensible header to store all the metadata related to an experiment, the ability to support data in an arbitrary number of dimensions, and the portability of the format. In addition, the first widely used file formats stored metadata in unnamed fields. For example, the patient's name was stored in the $x$ number field. The goal of MINC was to be able to name the fields explicitly so that we did not need a manual to understand the data in the file, as was the case with the proprietary DICOM file format, for example. Each MINC2.0 file must store all its data within the "minc-2.0" group located directly at the root of an HDF5 file. Within this main group are the metadata of the experiment and a folder to store the data at different resolutions (in dedicated subfolders). Among the mandatory metadata are the history of use and creation of the file, the version of MINC used to create the file and a unique identifier [28].

## 2.3 Multidimensional array processing frameworks

As explained, arrays can be processed on a single machine or on a cluster of machines. Especially in the case of large datasets the data cannot fit on a single machine and can be processed on a cluster in order to make the data processing time reasonable. To do this, there are two main data containers that can be used: a file system, or a database [14]. Distributed file systems allow data to be efficiently stored on a cluster and make it accessible to distributed computing environments such as Spark or Dask. Spark [24] and Dask [19] are two distributed computing environments, while SciDB [25] is a database that specializes in processing arrays of data. More than a database, SciDB also contains a toolbox that is optimized for processing and querying arrays. These three environments arose from the observation that traditional relational databases were not well suited for storing and processing multi-dimensional data. Although plugins were designed to make them work, they were complicated to implement and were ultimately sub-optimal compared to systems designed specifically for multi-dimensional arrays. SciDB aims at storing the data efficiently so that it can be analyzed using queries or specialized tools. Spark and Dask are specialized in designing complex algorithms and for batch processing [14, 8].

The distributed processing systems presented above all work with three main components: client nodes, a master node and slave nodes. The principle is as follows: the client nodes send work to the master node, which distributes the tasks in an optimized way to the slave nodes. Once the slave nodes have completed their work, they send the results in the opposite direction: first to the master node, which then sends the results to the client node. The great advantage of these systems is that they generally do not require an expensive infrastructure; the idea is to be able to replace the nodes when they fail, transparently. In addition, distributed storage makes information systems more resilient by replicating data, for example, so that data is not lost if a machine crashes [8, 24].

## 2.4 The Python programming language

The Python programming language has become the reference for data processing [13]. It is widely used in the scientific field because it is easy to use and allows scientists to focus on their algorithms rather than on syntax or code optimization. In particular, it benefits from a rich ecosystem of libraries to manipulate data, create ready-to-use graphs in scientific publications and manipulate specialized file formats. At the heart of this ecosystem are SciPy and Numpy, which are toolboxes respectively specialized in engineering and the manipulation of multidimensional arrays [13, 10]. Another library largely responsible for Python's success in data science is the Pandas library, specialized in tabular data processing. More recently, xarray, built on top of several libraries, has been added to the many libraries that handle multidimensional data tables. It simplifies the use of arrays whose dimensions are indexed by name while allowing to use the power of other specialized libraries such as Dask which allows to make parallel calculations on multidimensional arrays.

With the advent of Deep Learning, which has become one of the most active areas of scientific research today, Python seems to be on its way to maintain its leadership position in the field of data processing as evidenced by the success of the Pytorch and Tensorflow libraries, the two most popular libraries for training neural networks. In addition to being domain-specific, these libraries allow calculations to be easily ported to graphics cards (GPUs) and tensor processing units (TPUs), which can drastically accelerate calculations [10]. Finally, although a high-level Python language is capable of being very fast, it allows many optimizations thanks to compiled languages such as Cython or Numba [10].

### 2.4.1 Dask

Dask is a popular Python package, part of the SciPy ecosystem (the scientific ecosystem of Python), enabling parallel and out-of-core computations [19].

Dask represents computations as task graphs that are dynamically executed by one of several schedulers including the single-threaded, the multi-threaded, the multi-process, and the distributed schedulers. Custom schedulers can also be implemented.

Dask graphs can be used out-of-the-box or through built-in APIs. A Dask graph is implemented in plain Python as a dictionary with any hashable as keys and any object as values. More precisely, a “Value” is any object different than a task and a “Task” is a tuple with a callable as first element. Examples of APIs/collections include `dask.array`, a parallel and out-of-core Numpy clone, and dask.dataframe, a Pandas clone. `dask.array` is a data structure designed for multi-dimensional array processing using blocked algorithms, hence leveraging chunking.

Dask was an inspiration for this thesis, as it offers faster IO and array processing, transparently. Moreover, it enables such optimizations on both local computer and distributed infrastructure. Nevertheless, it also suffers from excessive seeking in some cases, which made us wonder if our work could be integrated into Dask to enable a vast user base to benefit our algorithms.

## 2.5 Sequential algorithms for optimizing multidimensional arrays manipulation

Used in all fields of science to represent and process data, multidimensional array processing is an important part of the work of scientists. Array storage involves several levels of complexity: choice of infrastructure, file format, storage type (chunked or not), and layout on disk. In any case, array parts are stored on disk or in memory, therefore, most scientific processing is subject to seeking. As explained in [21], repositioning the read head to read multiple data parts inside a file causes an important overhead. Finally, chunking requires tools to be able to manipulate arrays efficiently. The advantage of optimizing the read/write of files is that it is independent of the application and common non compressed file formats. Finally, [19] reminds us that the use of data clusters and powerful hardware is not always required, as it requires some computer knowledge and that PCs have acquired enough power to handle some Big Data applications. Input/output optimization also makes sense in this context where one wants to manipulate data that is too large to fit into RAM, directly on one’s computer.

Previous work in [11] showed that naive algorithms to split an array into several chunks or merge array chunks into one output file do not perform well due to millions of seeks occurring on disk. Two types of sequential algorithms were introduced: the clustered and the multiple strategies. The clustered strategy aims at reading each chunk only once but seeks into the original image, while the multiple strategy consists in minimizing seeks in the reconstructed image, seeking into the chunks instead. For example, for the split (or "chunking") task, clustered strategies reads as much contiguous chunks as possible into main memory before writing them without seeks into the output files, whereas the multiple strategy reads contiguous columns of data on disk without seeks, but seeks into the output files, writing the different parts of the loaded buffer into different chunks [11]. The "merge" task is shown to be the reverse process. No strategy has proven better than the other overall, but they both have preferred use cases. Experiments showed that such sequential algorithms are 5 to 10 times faster than naive algorithms, which is promising.

The present study is focused on sequential algorithms for repartitioning multidimensional arrays, letting the parallel and distributed cases as future work, although they are relevant, too. To the best of our knowledge, the array repartitioning problem as defined in this study has not been studied.

## 2.6 Example use case of the repartitioning task

One important domain of application of the repartitioning task could be fMRI time series processing. fMRI time series processing is used in functional neuroimaging, for connectomics studies for example. Functional neuroimaging consists in using neuroimaging technology to study how the brain work, how brain regions are related to each other or to understand the relationships between brain regions and mental functions for example. It is used in cognitive neuroscience and neuropsychology [1]. We call connectomics the "Big Data approach for analyzing the massive datasets produced by [...] brain imaging" [26]. Functional neuroimaging includes searching for inter- and intra-subject correlations and finding correlations between aggregated time series [23]. For example, studies like [23] perform inter-subject correlation analysis to find brain activation patterns in people watching movies.

Datasets usually consists in a set of initial images [7]. These images represent brain slices, ordered by time [7]. At first, some preprocessing is required at the image-level to remove noise or perform realignment and slice time correction for example [23]. Then, specific computations are applied on the time-series like Fast Fourier Transforms (FFTs) [18, 23] or Pearson correlations [23]. For these applications however, the data is processed as pixel time series or as voxel regions time series [18, 23]. Therefore, we may need to process the data as 2D slices first, then as 1D or 3D cuboids. Instead of searching for 1D or 3D parts from the initial 2D images of the dataset, it may be faster to repartition the data between the computations.

This example application has been mentioned by Matthew Rocklin, foundator of Dask, who gave a tutorial on how to process a connectomics dataset on his blog [18]. The dataset processed in the tutorial consists in drosophilia brain slices and is preprocessed as such. The author explains that it is then necessary to "rearrange [the] data from being partitioned by time slice, to being partitioned by pixel" to apply FFTs on the time series [18].

## 2.7 The Region Of Interest Extraction Problem

ROI extraction stands for Region Of Interest extraction [17, 27]. One can divide medical images in regions of interest (ROI) and background areas [27]. From a clinical point of vue, medical images can be divided into lesion areas of interest and non lesion areas, which may occupy much of the image. Removing unuseful data by extracting the ROIs can speedup computations and data transfer [27].

As explained in [17], ROI extraction is an important component of fMRI images processing: "A common approach to the analysis of fMRI data involves the extraction of signal from specified regions of interest (or ROI's)". In fMRI analysis in particular, ROIs can be used for data exploration, statistical control and functional specification [17]. For data exploration for example, ROIs can be useful for studying the activity of a specific brain area under different experimental conditions or by selecting only some variables of interest.

Extracting ROIs from an image, either stored into one file or chunked into several files, may produce important disk seeking, hence a big processing time overhead. According to [17], the "most common approach for exploratory ROI analysis is to create small ROIs (**usually spheres**) at the peaks of activation clusters".

Minimizing disk seeking in tasks like the repartioning task could enable to find efficient disk seeking minimization strategies and speedup common tasks like the ROI extraction problem.

# Chapter 3

# The keep heuristic for repartitioning multidimensional arrays

This chapter presents the KEEP heuristic, an algorithm to reduce the number of seeks created when repartitioning large multi-dimensional arrays in blocks of arbitrary shape. Repartitioning may be triggered either by application requirements or for performance reasons, to improve memory usage and I/O efficiency. For instance, the Python HDF5 library `h5py` recommends a chunk size "between 10 KiB and 1 MiB" [6], while the Dask parallel processing framework [19] suggests a chunk size greater than 100 MB [20].

The KEEP heuristic leverages a memory cache to read and write data as contiguously as possible, similarly to the "clustered" and "multiple" algorithms described in [11]. The storage order of array elements on disk is assumed to be known to the application, but unconstrained. Implementations based on our algorithm could therefore use arbitrary file formats. We focus on sequential algorithms, assuming that arrays are stored on a single device and accessed by single-threaded I/Os. Extensions to parallel environments are part of our future work. Although in practice seek times depend on various factors, such as the distance between the initial and target drive positions, we focus on minimizing seek *numbers*, for simplicity. Likewise, the effects of I/O optimizations such as page caching or readahead will be discussed experimentally but not modelled.

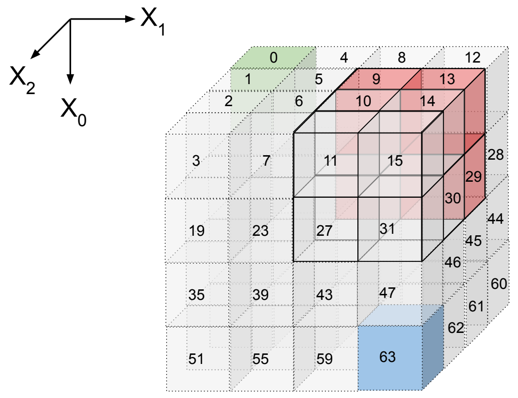


Figure 2: Ordering of 3D array elements in files in C order. The first stored element is colored in green, the last one is in blue. Seeking to the red elements is required when extracting the 3D block with plain borders.

## 3.1 Problem definition and baseline

For the sake of clarity and without loss of generality, we assume that files are written in row-major order (Fig. 2, a.k.a. C order), where the fastest moving dimension in the file is the last dimension of the array, and the slowest moving dimension in the file is the first dimension of the array. This convention is, for instance, the one used in the HDF5 format [9].

Accessing data from an array stored on disk generates seeks in two situations: (1) when an array block is opened for reading or writing, and (2) when the reading or writing process moves to a different address within the block.

### 3.1.1 The re-partitioning problem

We focus on 3D arrays for simplicity. Consider a 3D array `A` of shape $A = (A_i, A_j, A_k)$, partitioned in uniformly-shaped input blocks of shape $I = (I_i, I_j, I_k)$ stored on disk. Our goal is to re-partition the input blocks into uniform output blocks of shape $O = (O_i, O_j, O_k)$, where $O \neq I$. Notations are summarized in Table 1.

Table 1: Notations

| Mono-space fonts are used for array partitions | |
|---|---|
| `A` | 3D array to repartition |
| `inBlocks` | set of input blocks |
| `outBlocks` | set of output blocks |
| `readBlocks` | set of read blocks |
| `writeBlocks` | set of write blocks |

| Square capitals are used for block shapes | |
|---|---|
| $\mathrm{A} = (A_0, A_1, A_2)$ | shape of `A` |
| $\mathrm{I} = (I_0, I_1, I_2)$ | shape of input blocks |
| $\mathrm{O} = (O_0, O_1, O_2)$ | shape of output blocks |
| $\mathrm{R} = (R_0, R_1, R_2)$ | shape of read blocks |
| $\mathrm{W} = (W_0, W_1, W_2)$ | shape of write blocks |

| Round capitals are used for block end coordinates | |
|---|---|
| $\mathcal{I} = (\mathcal{I}_0, \mathcal{I}_1, \mathcal{I}_2), \quad \mathcal{I}_d \subset \mathbb{N}$ | input block end coordinates |
| $\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \mathcal{O}_2), \quad \mathcal{O}_d \subset \mathbb{N}$ | output block end coordinates |
| $\mathcal{R} = (\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2), \quad \mathcal{R}_d \subset \mathbb{N}$ | read block end coordinates |
| $\mathcal{W} = (\mathcal{W}_0, \mathcal{W}_1, \mathcal{W}_2), \quad \mathcal{W}_d \subset \mathbb{N}$ | write block end coordinates |

| $n_X$ are used for block numbers |
|---|
| $n_I$ = number of input blocks |
| $n_O$ = number of output blocks |
| $n_R$ = number of read blocks |
| $n_W$ = number of write blocks |

We formalize the re-partitioning problem as in Algorithm 1. A re-partitioning algorithm takes a list of input blocks `inBlocks`, a list of output blocks `outBlocks`, and the amount of memory `m` available for the algorithm. Subject to `m`, the algorithm determines (1) `readBlocks`, a list of uniformly-shaped blocks of shape $R$ to be read from the input blocks, and (2) `writeBlocks`, a list of blocks, usually not uniformly-shaped, to be written to output blocks (line 10). If read blocks and input blocks have a different shape, then reading read blocks requires more than one seek. Similarly, if write blocks and output blocks have a different shape, then writing write blocks requires more than one seek. Input, output, read and write blocks all form a partition of the 3D array `A`. The main loop of the algorithm (line 12) loads one read block at a time (line 13), inserts it into a cache (line 14), and writes write blocks from the cache when they are complete (lines 15 – 20). Function `write` is assumed to open each output block only once, and to write the required elements of `data` in it.

It should be noted that blocks are passed to the algorithm by reference, not by data. Only variables `data` (lines 13, 17) and `cache` (line 14) hold actual data, contributing to the consumed memory.

Since read and write blocks both represent a partition of $A$, all the elements in `inBlocks` are read exactly once and written exactly once too. Other problem formalizations may allow for input elements to be read and discarded, to reduce seeking. Exploring such a trade-off between seeks and redundant reads could be an interesting extension to this problem.

Our goal is to minimize the number of seeks done by the algorithm, which occurs during reads (line 13) and writes (line 18). The re-partitioning problem is therefore to find `readBlocks` and `writeBlocks` that minimize the number of seeks done by the algorithm, subject to the memory constraint $m$.

Solutions of this problem materialize as implementations of function `getReadWriteBlocks` (Alg. 1, line 10). A lower bound on the number of seeks for the repartitioning problem is $n_I + n_O$, with $n_I$ the number of input blocks and $n_O$ the number of output blocks. Indeed, input and output blocks all have to be opened at least once, which requires a seek.

For simplicity, we require that all blocks in `readBlocks` have the same shape $R$. We also equate the size of an array in memory as its number of elements. To the best of our knowledge, no algorithm has been proposed for the repartitioning problem.

**Algorithm 1** General re-partitioning algorithm

```
1: **Inputs**
2: inBlocks: input blocks of shape $I$, stored on disk
3: outBlocks: output blocks of shape $O$, to be written
4: $m$: memory available
5:
6: **Outputs**
7: none (outBlocks are written)
8:
9: **Algorithm**
10: readBlocks, writeBlocks $\leftarrow$ getReadWriteBlocks($I$, $O$, $m$)
11: initialize(cache)
12: for readBlock in readBlocks do
13:     data $\leftarrow$ read(readBlock, inBlocks)
14:     cache.insert(data)
15:     for writeBlock in writeBlocks do
16:         if readBlock $\cap$ writeBlock $\neq \emptyset$
                and cache.isComplete(writeBlock) then
17:             data $\leftarrow$ cache.pop(writeBlock)
18:             write(data, outBlocks)
19:         end if
20:     end for
21: end for
```

### 3.1.2 Baseline

Our baseline algorithm for the repartitioning problem loads one input block at a time ($R$=$I$), and directly writes it to the appropriate output blocks ($W$=$R$). It assumes that input blocks fit in memory. The number of generated seeks $s_b$ depends on how write blocks overlap with output blocks in each dimension (Fig. 3). To determine $s_b$, we define $\mathcal{W}_d$ and $\mathcal{O}_d$, the sets of write and output block end coordinates in dimension $d$:

$$\forall d \in \{1, 3\},$$

$$\mathcal{W}_d = \Big\{ iW_d, i \in \{1, A_d/W_d\} \Big\} \quad \text{and} \quad \mathcal{O}_d = \Big\{ iO_d, i \in \{1, A_d/O_d\} \Big\} \tag{1}$$

We then define the *cuts* of the $i^{th}$ output block along dimension $d$ as follows:

$$\forall i \in \{1, A_d/O_d\},$$

$$C_{i,d} = C_{i,d}\left(\mathcal{W}_d, \mathcal{O}_d\right) = \Big\{ w \in \mathcal{W}_d, (i-1)O_d < w < iO_d \Big\}$$

The number of output block pieces along dimension $d$ is then:

$$c_d = c_d\left(\mathcal{W}_d, \mathcal{O}_d\right) = \sum_{i=1}^{A_d/O_d} |C_{i,d}\left(\mathcal{W}_d, \mathcal{O}_d\right)| + 1 - \delta_{|C_{i,d}(\mathcal{W}_d, \mathcal{O}_d)|,0}$$

where $|\ .\ |$ is the cardinality operator and $\delta$ is the Kronecker delta. The number of *matches* between block endings is:

$$m_d = m_d\left(\mathcal{W}_d, \mathcal{O}_d\right) = |\mathcal{W}_d \cup \mathcal{O}_d| - c_d$$

Finally, the number of seeks created by the baseline algorithm is:

$$s_b = A_0A_1c_2 + A_0c_1m_2 + c_0m_1m_2 + n_I + (m_0 + c_0)(m_1 + c_1)(m_2 + c_2) \tag{2}$$

with $(m_0 + c_0)(m_1 + c_1)(m_2 + c_2)$ the number of output block openings.

As can be seen from Equation 2, cuts along dimension 2 generate $\mathcal{O}(A_0A_1)$ seeks, which is prohibitive when `A` is large, cuts along dimension 1 generate $\mathcal{O}(A_0)$ seeks and cuts in dimension 0 generate $\mathcal{O}(1)$ seeks. The KEEP heuristic described in the next section aims at avoiding cuts primarily in dimension 2, and if possible in dimension 1 and 0. An experimental verification of the model will be provided in Section 3.3.

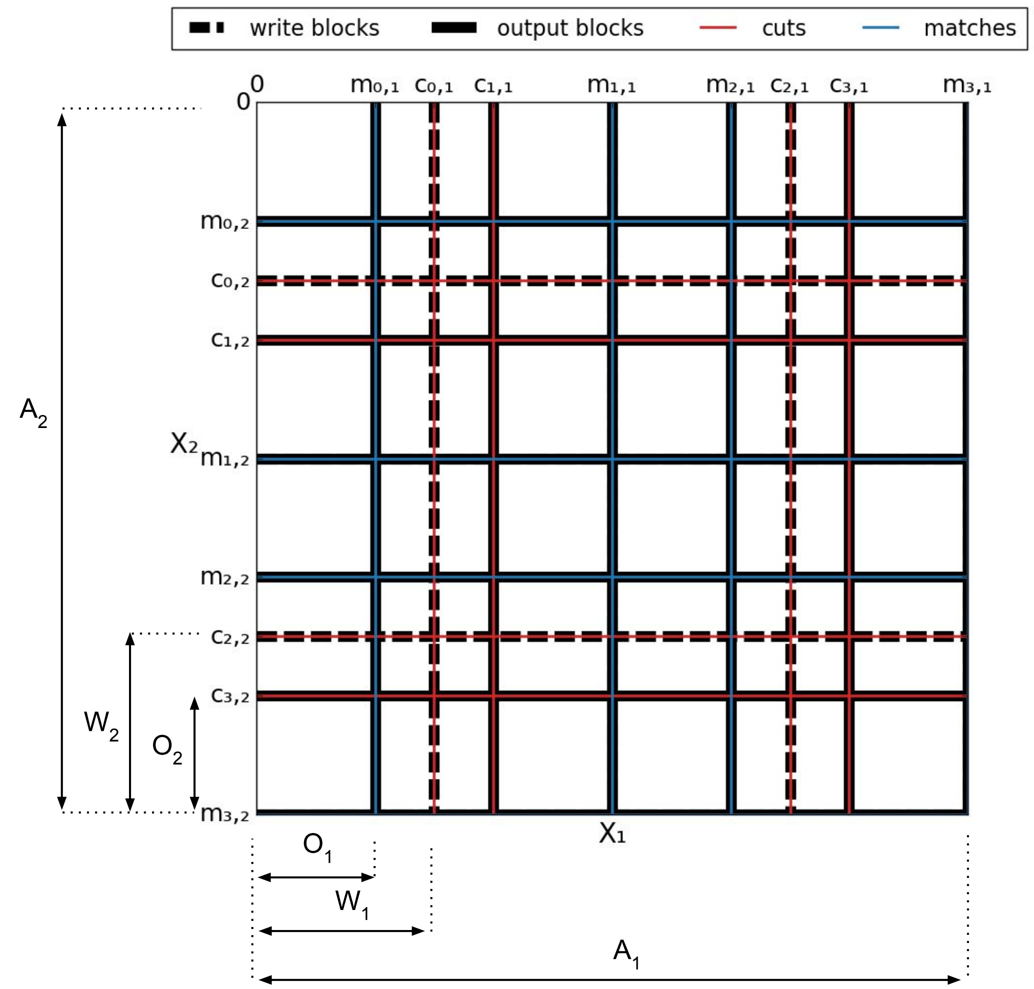


Figure 3: Overlapping of write and output blocks in the plan ($X_1$,$X_2$), showing cuts (red lines) and matches (green lines)

## 3.2 The keep heuristic

As mentioned previously, the baseline strategy empties the cache at each iteration, which generates many seeks when input and output blocks have different shapes. Instead, the proposed KEEP heuristic keeps in cache the array elements that cannot be written contiguously to the output blocks.

### 3.2.1 Overview

The KEEP heuristic (Alg. 2) is a brute-force search of the best read shape among a list of candidates (line 11). For each candidate read shape, it creates a list of write blocks (line 15), determines the memory consumed by this solution (line 16), and provided that the memory constraint is respected, computes the number of seeks created by the solution (line 24). Finally, the algorithm selects the read shape that minimizes the number of seeks (lines 25-27).

The KEEP heuristic uses the following functions, described in the remainder of this section: `candidateReadShapes` (line 11), `blocks` (line 14), `createWriteBlocks` (line 15), `peakMemory` (line 16), and `generatedSeeks` (line 24).

### 3.2.2 Candidate read shapes

Read shape candidates are generated as to avoid cuts along dimension 2 during reads and writes, as they are the most costly. Ideally, the read shape would cover an exact number of input blocks, to avoid seeks during reads, and would include at least one output block, to avoid seeks during writes. In practice, this might not be possible due to memory limitations, since at each iteration the parts of the read block that are not written must remain in cache.

To generate candidate read shapes, we first define a shape $\hat{r}$ such that in each dimension $d$, $\hat{r}_d$ is the smallest multiple of $I_d$ that is greater or equal to $O_d$:

$$\forall d \in \{1,2,3\},$$
$$\hat{r}_d = I_d\left(1 + \left\lfloor \frac{O_d}{I_d} \right\rfloor\right)$$

**Algorithm 2** KEEP heuristic (implements `getReadWriteBlocks`)

```
Inputs
inBlocks: input blocks of shape $I$, stored on disk
outBlocks: output blocks of shape $O$, to be written
$m$: memory available

Outputs
readBlocks: a list of read blocks
writeBlocks: a list of write blocks

Algorithm
readShapes = candidateReadShapes($I$, $O$, $A$)
minSeeks = None
for $R$ in readShapes do
    readBlocks = blocks($R$, $A$)
    writeBlocks = createWriteBlocks(readBlocks, outBlocks)
    mc = peakMemory($R$)
    if mc > m then
        continue {Memory constraint cannot be fulfiled}
    end if
    if $R == \hat{r}$ then
        bestReadBlocks = readBlocks
        break {$\hat{r}$ fulfills memory constraint}
    end if
    s = generatedSeeks($I$, $R$, writeBlocks, $O$)
    if minSeeks == None or s < minSeeks then
        minSeeks, bestReadBlocks = s, readBlocks
    end if
end for
return bestReadBlocks, writeBlocks
```

Using $\hat{r}$ as read shape minimizes seeking in the input blocks, as in this situation each iteration reads input blocks completely, requiring a single seek per input block. If there is enough memory to use write blocks of shape $O$, then this solution also minimizes the seeks in the output blocks, as output blocks can be written in a single seek too.

Function `candidateReadShape` returns the following shapes:

$$\left\{(C_0, C_1, \hat{r}_2) \in \mathbb{N}^3 \;/\; \forall i \in \{0,1\},\;\; \exists k_i \in \mathbb{N},\;\; A_i = k_i C_i \;\text{ and }\; C_i \leq \hat{r}_i\right\}$$

The third coordinate of all read shapes is set to $\hat{r}_2$, to guarantee that no cuts along dimension 2 will be made during reads. This assumes that $m \geq \hat{r}_2$, i.e., a block of shape (1, 1, $\hat{r}_2$) fits in memory, which seems reasonable. It also assumes that $\hat{r}_2$ is a divisor or $A_2$. The other two coordinates are set to divisors of $A_0$ and $A_1$ which are less than or equal to $\hat{r}_0$ and $\hat{r}_1$ respectively.

The candidate shapes are sorted first by decreasing $C_1$ values, then by decreasing $C_0$ values. The first tested read shape is therefore $\hat{r}$. While $\hat{r}$ is the optimal read shape, leading to a single seek per input or output block, there is no obvious relation between the other candidate shapes and the number of seeks that they generate since this number depends on how input, read, write, and output blocks are arranged. For this reason, the KEEP heuristic returns if $\hat{r}$ respects the memory constraint (line 20-23), but it otherwise evaluates all the other read shapes.

### 3.2.3 Block order

Creating uniform blocks from a block shape is straightforward, but the order in which they are read impacts the amount of data stored in cache and therefore the peak memory consumption. For simplicity, function `block` returns read blocks in the order used for array elements on disk (C order in this paper).

### 3.2.4 Creation of write blocks

Function `createWriteBlocks` returns write blocks created from read and output blocks as follows. For each read block, eight write blocks are first created, defined as the $F_i$ blocks in Figure 4. The coordinates of $F_0$ are computed as follows: The upper-left coordinates are $(0, 0, 0)$ in the referential of the read buffer, and the bottom-right coordinates are defined to be the first output block's end coordinates encountered.

The $F_i$ blocks with $i \in [\![1; 7]\!]$ are then divided into smaller write blocks by the end coordinates of the output blocks it crosses i.e. the output blocks which will contain the read block's data. Each smaller write block in the $F_0$ block (Figure 4) is assigned type $F_0$, each write block created from the $F_1$ block is assigned type $F_1$ etc. Finally, the write blocks are merged recursively with sub-blocks of neighboring read blocks, following the rules in Table 2. For example, Table 2 tells the algorithm to fuse write blocks of type $F_2$ in the $X_1$ direction, as the read block cuts an output block in the $X_1$ direction. In fact, the $F_i, i \in [\![1; 7]\!]$ write blocks are fused with $F_0$ write blocks from neighboring read blocks. By fusing neighboring write blocks together, the KEEP heuristic forces the general algorithm (Alg. 1) to write contiguous output block parts at once.

Table 2: Recursive merging of the initial write blocks. The first column identifies a sub-block of a read block (i.e., an initial write block), and the second column identifies the sub-block(s) of the neighboring read block(s) that are merged with the sub-block of the first column. For instance, sub-block $F_1$ is merged with sub-block $F_0$ of the neighboring read block in dimension $X_2$.

| Sub-block | Merged with (sub-block, neighbor direction) |
|---|---|
| $F_0$ | – |
| $F_1$ | $(F_0,\ X_2)$ |
| $F_2$ | $(F_0,\ X_1)$ |
| $F_3$ | $(F_1,\ X_1)$, $(F_2,\ X_2)$ |
| $F_4$ | $(F_0,\ X_0)$ |
| $F_5$ | $(F_1,\ X_0)$, $(F_4,\ X_2)$ |
| $F_6$ | $(F_2,\ X_0)$, $(F_4,\ X_1)$ |
| $F_7$ | $(F_3,\ X_0)$, $(F_5,\ X_1)$, $(F_2,\ X_2)$ |

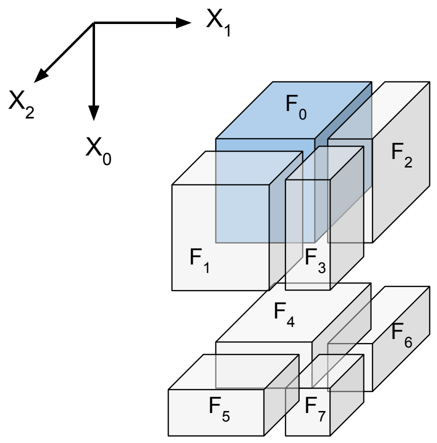


Figure 4: Division of a read block into write blocks. $F_0$, the write block represented in blue, is the output block containing the origin of the read block. $F_1$-$F_7$ are the intersections of the read block with the neighboring output blocks. Following this division, write blocks are merged using the scheme in Table 2.

### 3.2.5 Peak memory

To determine the peak memory used by a particular combination of read and write blocks, one needs to determine the amount of data in the cache at each iteration, which depends on how read and write blocks overlap. Since we were only able to find an analytical upper bound of the peak memory usage, which in fact largely overestimates it, we instead simulate each iteration of the KEEP heuristic to measure the cache size.

The simulation requires |`readBlocks`| iterations. It assumes that (1) read blocks are in the order used for array elements on disk (C order in this paper), consistently with function `blocks` and (2) write blocks are defined as described in the previous section. The cache size is computed for each read buffer and its max across all iterations is returned as the peak memory consumption.

### 3.2.6 Number of generated seeks

Similar to the baseline, seeks generated by the KEEP heuristic happen during reads (line 13 in Alg. 1) and during writes (line 18). To count the seeks generated during reads, we note that the read process is dual to the write process: writing in-memory blocks with end coordinates $\mathcal{M}$ to disk blocks with end coordinates $\mathcal{D}$ generates the same number of seeks as reading disk blocks with end coordinates $\mathcal{D}$ to in-memory blocks with end coordinates $\mathcal{M}$. We can determine this number using Equation 2 without the last term:

$$\begin{aligned}\textsc{seeks}\left(\mathcal{M},\mathcal{D}\right) &= A_0 A_1 c_2\left(\mathcal{M},\mathcal{D}\right) + \\ & A_0 c_1\left(\mathcal{M},\mathcal{D}\right) m_2\left(\mathcal{M},\mathcal{D}\right) + \\ & c_0\left(\mathcal{M},\mathcal{D}\right) m_1\left(\mathcal{M},\mathcal{D}\right) m_2\left(\mathcal{M},\mathcal{D}\right)\end{aligned}$$

Function SEEKS does not assume that $\mathcal{M}$ or $\mathcal{D}$ relate to blocks of uniform shape. Therefore, it can be applied to the KEEP heuristics where write blocks are not necessarily of uniform shape. The number of seeks generated by the KEEP algorithm is given by:

$$s_k = \textsc{seeks}\left(\mathcal{R},\mathcal{I}\right) + \textsc{seeks}\left(\mathcal{W},\mathcal{O}\right) + c_0\left(\mathcal{R},\mathcal{I}\right) c_1\left(\mathcal{R},\mathcal{I}\right) c_2\left(\mathcal{R},\mathcal{I}\right) + n_W \quad (3)$$

The first term represents the number of seeks required to read input blocks of end coordinates $\mathcal{I}$ into read blocks of end coordinates $\mathcal{R}$ whereas the second term represents the number of seeks required to write write blocks of end coordinates $\mathcal{W}$ into output blocks of end coordinates $\mathcal{O}$. The two last terms represent the number of input blocks openings and output blocks openings.

Function `generatedSeeks` is a direct implementation of Equation 3.

### 3.2.7 Implementation

We implemented the baseline and KEEP heuristic in Python 3.7. The project repository is available at https://github.com/big-data-lab-team/repartition_experiments. It includes documentation on how to install the package and its dependencies. The code used for the experiments is also available in the same repository. The implementation supports the HDF5 file format, but the code is modular enough to accommodate other formats in the future.

To reduce the time requirements of brute-force search, we limit the testing of candidate read shapes as follows in the implementation: starting from $\hat{r} = (\hat{r}_0, \hat{r}_1, \hat{r}_2)$, we evaluate candidate read shapes sorted by decreasing $X_1$ values, then by decreasing $X_0$ values. We stop the search if the best number of seeks (`minSeeks` in Alg. 2) does not decrease for 10 iterations.

We could have passed the data from the read buffers to the cache by reference which would have saved us the time of copying the data. However, keeping a reference to the read buffers prevented them to be freed and therefore increased the maximum amount of memory used drastically. Therefore, we found that copying the data into the cache enabled us to stay below the maximum memory consumption that we predicted at the buffer selection time. This is the main limit of our current implementation, preventing us to experiment with very large images, as explained in the results section.

## 3.3 Experiments

### 3.3.1 Seek model validation

We tested our models against lighter implementations of the baseline and KEEP algorithms. We tested each model on 1,000 randomly generated cases. To that aim, we first generate a random original array by multiplying 4 numbers randomly drawn from $2, 3, 5, 7$, putting each number to the power 1 or 2 (again, randomly) and multiplying them: $\hat{r} = (r, r, r)$ with $r = (x_0^{p_0} x_1^{p_1} x_2^{p_2} x_3^{p_3})$. We then compute the divisors of $r$ and take two of them for the shapes of $I$ and $O$. We create 5 cases by picking 5 buffer shapes from the possible buffer shapes given the configuration $(\hat{r}, I, O)$. We do the same after exchanging $I$ with $O$. This gives us 10 configurations for a given randomly created $\hat{r}$. We repeat the process until 1,000 cases have been randomly generated. Then, for each case, we run our model to compute the number of seeks that the algorithm should do, and compare it to the real number of seeks computed by running a light version of the algorithm.

Table 3: Input and output blocks shapes

$A = (3500, 3500, 3500)$

| Config | inBlocks $I$ | $n_I$ | size (MiB) | outBlocks $O$ | $n_O$ | size (MiB) |
|---|---|---|---|---|---|---|
| 0 | (875,875,875) | 64 | 1277 | (875,1750,875) | 32 | 2,555 |
| 1 | | | | (700,875,700) | 100 | 818 |
| 2 | (350,350,350) | 1,000 | 82 | (500,500,500) | 343 | 238 |
| 3 | | | | (250,250,250) | 2,744 | 29 |
| 4 | (175,175,175) | 8,000 | 10 | (250,250,250) | 2,744 | 29 |
| 5 | (350,875,350) | 400 | 204 | (500,875,500) | 196 | 417 |
| 6 | | | | (350,500,350) | 700 | 116 |

$A = (8000, 8000, 8000)$

| Config | inBlocks $I$ | $n_I$ | size (MiB) | outBlocks $O$ | $n_O$ | size (MiB) |
|---|---|---|---|---|---|---|
| 0 | (2000,2000,2000) | 64 | 15,258 | (2000,4000,2000) | 32 | 30,518 |
| 1 | | | | (1600,1600,1600) | 125 | 8,766 |
| 2 | (800,800,800) | 1000 | 977 | (1000,1000,1000) | 512 | 1,907 |
| 3 | | | | (500,500,500) | 4096 | 238 |
| 4 | (200,200,200) | 64000 | 15 | (250,250,250) | 32768 | 30 |
| 5 | | | | (160,160,160) | 125000 | 8 |
| 6 | (400,400,400) | 8000 | 122 | (500,500,500) | 4096 | 238 |
| 7 | | | | (250,250,250) | 32768 | 30 |

### 3.3.2 Experiment conditions

The experiments were run on a DELL-EMC PowerEdge C6420 server with CentOS Linux release 8.1.1911 (Core), kernel release $4.18.0 - 147.5.1.el8\_1.x86\_64$, 250GB of RAM, 6 $\times$ SSDs Intel SSDSC2KG480G8R (480 GB, xfs), 2 $\times$ Intel Xeon Gold 6130 CPUs @ 2.10GHz (16 cores/CPU, 2 threads/core). The server was accessed through the SLURM batch manager.

We repartitioned two arrays of 2-byte elements (float 16): a "small" one (85.75 GiB) of shape $A = (3500, 3500, 3500)$ and a "large" one (1024 GiB) of shape $A = (8000, 8000, 8000)$. We used the input and output block shapes in Table 3. We repartitioned the small array with $m$=275 GiB (256 GB), 8.6 GiB (8 GB) and 4.3 GiB (4 GB), to benchmark the baseline and KEEP algorithms with different memory constraints. For the large array we used $m$=256 GB only. The memory was controlled through SLURM's cgroup-based memory requirement, which means that $m$ was the total amount of RAM allocated to the repartitioning task, including anonymous memory and cache. Swapping was disabled.

A single disk was used when processing the small array. However, for the large array, the 6 disks available on the server were used as the capacity of a single disk was not enough to accommodate the input or the output blocks. We used the disks in round-robin.

The page cache was flushed at the beginning of each run of an algorithm.

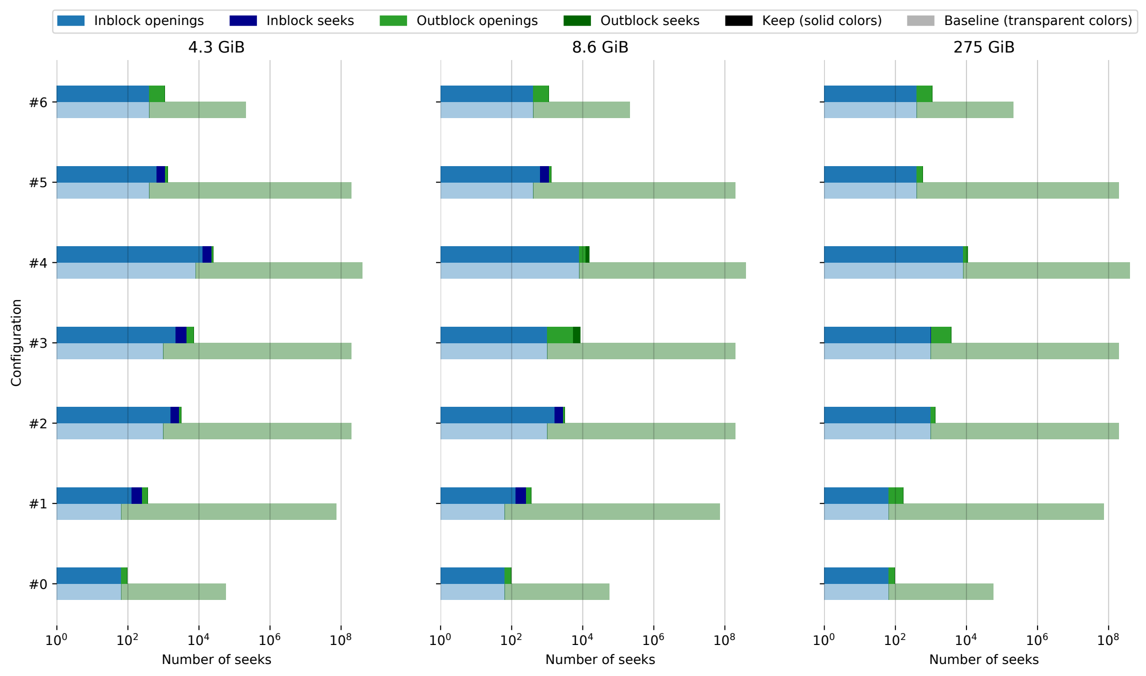
### 3.3.3 Results

#### Small array (85.75 GiB)

Results are shown in Figure 8. By design, the number of seeks is drastically reduced by the KEEP heuristic compared to baseline (Fig. 5a), by a factor of 90,000 on average. In configurations 1 to 5 in particular, baseline requires up to $10^8$ seeks while the KEEP heuristic only requires up to $10^4$.

This reduced seeking translates to important reductions of the repartitioning time

(a) Number of seeks (log scale)



(b) Processing time

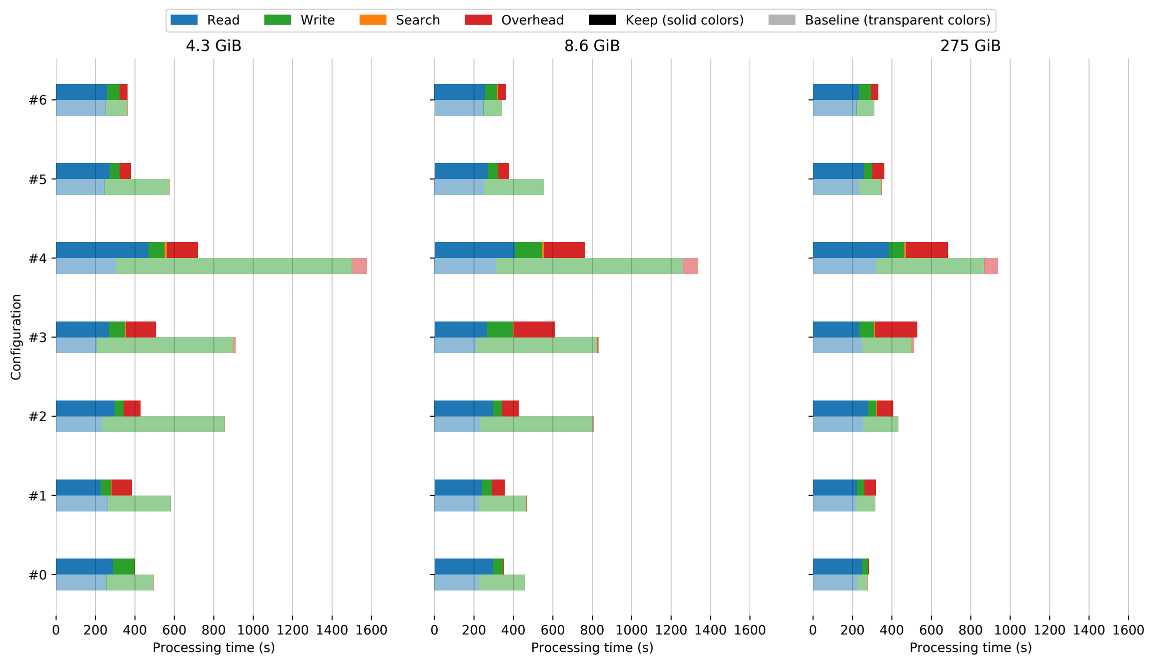


Figure 5: Repartitioning results for the small array (85.75 GiB). Averages on 5 repetitions. **(a)** Compared to baseline, the KEEP heuristic reduces the number of seeks by four orders of magnitude. **(b)** The KEEP heuristic provides speed-up factors of up to 2.5 (configuration 4).

(Fig. 5b). Compared to baseline, the KEEP heuristic provides speed up factors of up to 1.4 for $m$=275 GiB, 1.9 for $m$=8.6 GiB, and 2.2 for $m$=4.3 GiB. Speed-up is only observed when baseline requires $O(10^8)$ seeks, which occurs in configurations 1 to 5: lower amounts of seeks don't seem to have an effect on I/O time. Speed-up is mostly coming from write time reduction, since baseline read complete input blocks without seeking.

The baseline performances improve drastically with the amount of memory allocated to the repartitioning. For $m$=275 GiB, repartitioning times of baseline and KEEP are on par for all configurations except configuration 4. This is most likely due to the use of asynchronous write back to disk through the Linux page cache. In our experiments, the ratio of "dirty" data, the cache data waiting to be written to disk, was set to 40%, a common value for compute nodes, which means that the small array entirely fit in memory for $m$=275 GiB. The KEEP strategy is mostly useful for arrays that do not fit in page cache memory.

The overhead of the KEEP heuristic consists of search time (Alg. 2) and cache management (data copy to/from cache in Alg. 1). While search time is negligible, cache management is substantial except for configuration 0 where the total number of blocks is limited.

Table 4 shows the read block shapes selected by the KEEP heuristic. The optimal shape $\hat{r}$ was used for all configurations with $m$ = 275 GiB.

Table 4: Read block shapes and peak memory estimates selected by the KEEP heuristic to repartition the small image.

| Config | $m$=4.3 GiB | $m$=8.6 GiB | $m$=275 GiB |
|---|---|---|---|
| 0 | (875, 1750, 875)* | (875, 1750, 875)* | (875, 1750, 875)* |
| | 2.8 GiB | 2.8 GiB | 2.8 GiB |
| 1 | (700, 875, 875) | (700, 875, 875) | (875, 875, 875)* |
| | 1.8 GiB | 1.8 GiB | 15.3 GiB |
| 2 | (500, 700, 700) | (500, 700, 700) | (700, 700, 700)* |
| | 2.1 GiB | 2.1 GiB | 12.2 GiB |
| 3 | (250, 350, 350) | (350, 350, 350) | (350, 350, 350)* |
| | 0.45 GiB | 5.5 GiB | 5.5 GiB |
| 4 | (250, 350, 350) | (350, 350, 350)* | (350, 350, 350)* |
| | 0.45 GiB | 5.5 GiB | 5.5 GiB |
| 5 | (500, 875, 700) | (500, 875, 700) | (700, 875, 700)* |
| | 1.0 GiB | 1.0 GiB | 11.5 GiB |
| 6 | (350, 875, 350)* | (350, 875, 350)* | (350, 875, 350)* |
| | 1.2 GiB | 1.2 GiB | 1.2 GiB |

* selected read block shape was $\hat{r}$

### Large array

Given the 250 GB RAM available, the read buffer shapes tested were always equal to the input aggregates' shapes. Globally, one can see on Figure 6 that apart from run 2 the KEEP strategy and baseline are pretty equivalent, and for 5 runs over 8 the KEEP strategy is slightly faster. There is no important improvement, however, which is mainly due to 3 bottlenecks:

- the overhead time
- the preprocessing time
- the read time

It seems logical to get an overhead time that explodes, especially in run 2, as it was already important on the small array. As it is mostly due to cache manipulations, a bigger array implies bigger data transferred during these cache manipulations.

Interestingly, the preprocessing time can be very large. The preprocessing time includes the computation of the write buffers and the computation of a Python dictionary associating each buffer with the output blocks it intersects. The latter operation can be very expensive when there are a lot of buffers.

The experiment on a big array was worth it as it clearly shows a limit of the KEEP strategy: We can see on Figure 6 that for runs 2 to 4, the write time (in green) has been tremendously reduced, but at the cost of an increase in read time.

Finally, one can see on Figure 7 that in this experiment, too, the KEEP strategy succeeded in reducing the number of seeks significantly. Once again, the reduction in processing time is also non-linear with the reduction in seeking.

The outcomes and potential benefits of experiments on large arrays such as 1TB in size are difficult to predict because of the keep algorithm overhead time. The overhead time is mainly due to copying data from read buffers to the cache as explained earlier.

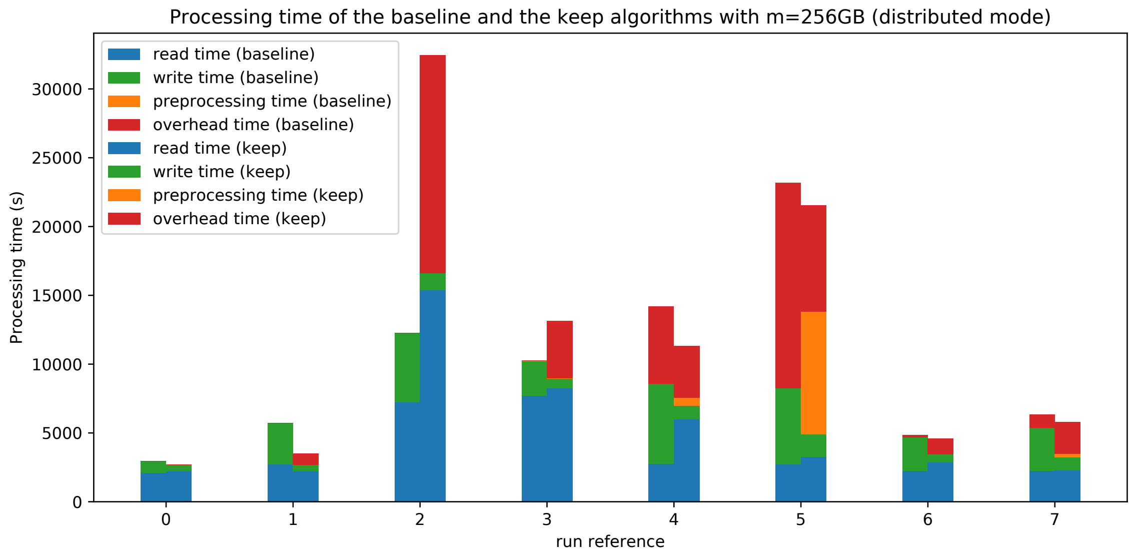


Figure 6: Results in terms of processing time for the keep and baseline algorithms. From left to right, the results are presented for 256, 8 and 4GB of available main memory.

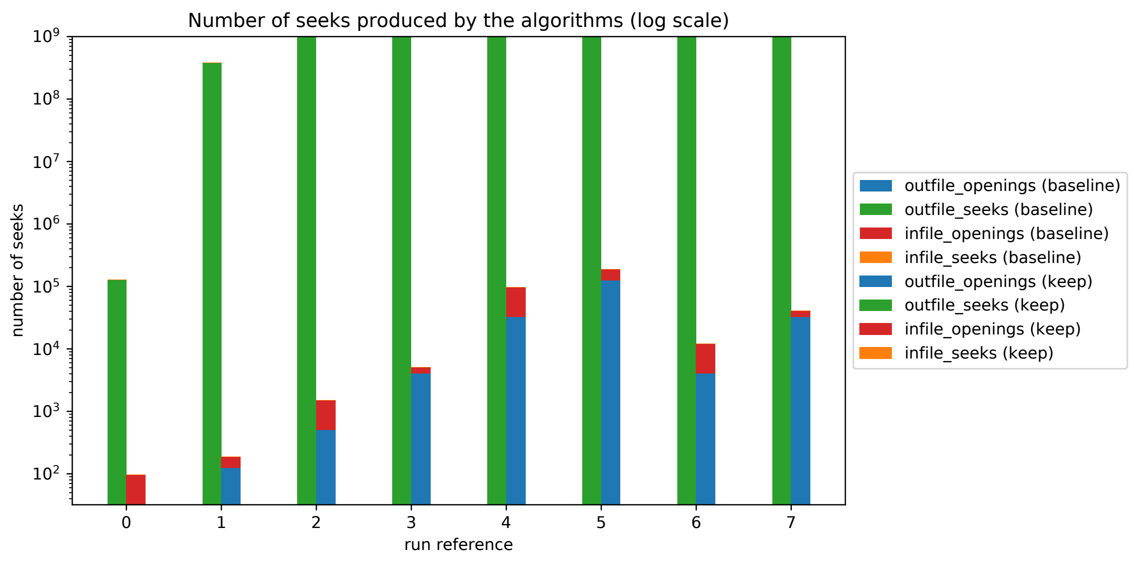


Figure 7: Results in terms of seeks for the keep and baseline algorithms. From left to right, the results are presented for 256, 8 and 4GB of available main memory.

## 3.4 Discussion

The KEEP heuristic can reduce the number of seeks required to repartition an array from millions to thousands compared to the baseline algorithm. As a result, write time is drastically reduced and speed-up factors of up to 2.5 are observed. Speed-up factors are expected to increase with the size of the repartitioned array; in 3D, speed-up is proportional to $A_0 A_1$ when enough memory is available to avoid cuts in dimension 2. Speed-up factors are also expected to increase with the dimension of the repartitioned array; in dimension $d$, speed-up is proportional to the product of the first d-1 $A_i$ when enough memory is available. Therefore, the KEEP heuristic is adapted to the repartitioning of large multi-dimensional arrays in the current context of growing data volumes.

### 3.4.1 Relevant extensions

Allocating only reasonable amounts of working memory, 4.3 GiB for an array of 85.75 GiB in our experiments, seems to be sufficient to remove most of the seeks required for the repartitioning. This is consistent with the fact that the KEEP heuristic only stores partial output blocks in cache, until they can be combined into complete output blocks. In the future, it would be interesting to derive an upper bound of the peak memory consumption, to generalize this observation beyond the particular input and output block shapes tested in our experiments.

The memory page cache provided by the Linux kernel plays a significant role in the repartitioning problem when the amount of memory increases. When repartitioning a 85.75 GiB array with 275 GiB of working memory, all the output blocks are written to memory and asynchronously flushed to disk, bringing the baseline algorithm on par with the KEEP heuristic. This is not surprising given the importance of page cache for I/O intensive applications [12]. The Linux kernel provides other relevant I/O strategies such as readahead and clustered writes that may interfere with the KEEP heuristic. Incorporating awareness of kernel I/O strategies into the repartitioning algorithm may further enhance performance.

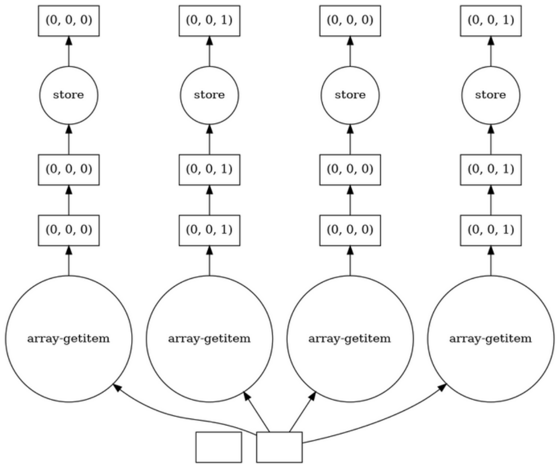
Most applications would process large multi-dimensional arrays in parallel, on a multi-core computer or on an HPC cluster. The KEEP heuristic is directly applicable to multi-core environments, assuming sequential I/Os. The case of HPC clusters, however, it would have to be extended to consider network data transfers. In addition, HPC clusters would most likely store input blocks on a parallel file system such as Lustre where files are striped across multiple disks and nodes. An extended formulation of the repartitioning problem would be required in this context. It would be interesting to integrate such an extension in the Dask engine, a popular parallelization framework of the SciPy ecosystem [19]. In particular, the KEEP heuristic would interface nicely with the `dask.array` API.

### 3.4.2 Implementing sequential optimization algorithms into Dask

As explained in the Background chapter, Dask reaches an important part of the scientific community, enabling users to perform data analysis locally and on distributed environments. In particular, it allows to efficiently process multidimensional arrays using the dask.array API, which represents computations using task graphs. We have done some early work in the implementation of the sequential algorithms presented in this thesis into the dask.array API to see if such algorithms could optimize the IO parts of Dask graphs.

We implemented the clustered algorithm for splitting and merging arrays into Dask. Figure 8b shows the result of applying the clustered algorithm to a task graph representing a “split” task (Figure 8a). At first the task graph is designed to read four subarrays from the original array and split it into four different files (Figure 8a). This task graph has to be read from bottom to top. The read tasks are named “array-getitem” and the write tasks are named “store”. We can see on Figure 8b that after applying the clustered strategy a new task has been added to the graph, forcing Dask to first load a big contiguous buffer into memory, before splitting it for parallel processing. That way, parallel processing is still possible, while reading input data without excessive seeking. Note that although Dask uses different tasks to do parallel processing, reading and writing to disk is still sequential. This work shows promising results and will be continued in the future.

(a) Task graph for splitting an array into several files, generated by Dask.



(b) A first task has been created to force Dask to read a contiguous buffer into memory, and then split it into different tasks.

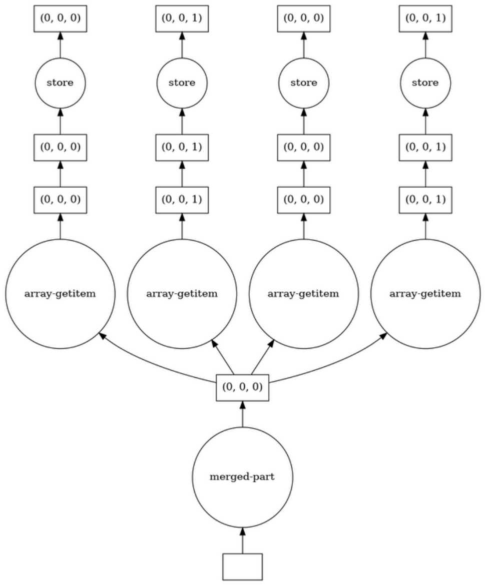


Figure 8: Optimization of a task graph using the clustered strategy.

# Chapter 4

# Conclusion

Multidimensional array chunking is a routine for scientists nowadays. That is why efficiently processing such arrays is of major importance in the big data era.

Previous work showed that splitting and merging arrays can be optimized by reducing disk seeking. We applied a similar strategy to the re-partitioning problem. This problem consists in efficiently re-writing a chunked array to change the chunks' shape. We formally defined the re-partitioning problem together with a baseline algorithm to solve it. We also presented the KEEP heuristic to reduce the number of seeks produced during the re-partitioning and hopefully reduce the processing time of such task. The KEEP heuristic reduces the number of seeks by (1) constraining the read buffers shape to minimize the read time, and (2) leveraging a cache to minimize the write time.

Although the re-partitioning problem is more complex than splitting/merging chunks, we proved that it can be optimized and that it is possible to reduce the processing time significantly. Surprisingly, however, the baseline algorithm has been found to perform pretty well when there is a lot of memory available as it leverages the page cache to speedup computations without reducing the number of seeks. For now, the KEEP algorithm may be more interesting than the baseline algorithm only when the memory constraint is important ($m$ small compared to the array size).

Some cases cannot be solved by our algorithms for now, like repartitioning non-regularly chunked arrays and cases where $\hat{r}_2$ is not a divisor of $A_2$. Also, the baseline algorithm assumes that there is enough main memory available to store one input block in memory.

Please note that due to implementation limitations we decided not to publish our work for the moment. We consider that further experiments with bigger datasets are required before publishing our results.

## 4.1 Future work

Our first goal is to publish our results, which requires us to find a way to remove the processing time overhead of the KEEP heuristic. It would also enable to use the KEEP heuristic for arrays of size 1TB or more.

This study was mainly a proof of concept to show that the repartitioning problem could be optimized by reducing the amount of seeks produced. Many improvements can be imagined for the KEEP heuristic alone, such as removing limitations due to assumptions like using read buffers of uniform shape. Indeed, the removal of this limitation would enable us to apply the KEEP heuristic to irregular chunking. Further processing speed could also be gained by using a different read buffer order than the naive one used in this study.

In the future, the algorithm performances may be improved by finding a variant to the KEEP algorithm that also reduce the read time significantly, identifying and focusing on reducing the most expensive seeks, and finally, finding parallel and/or distributive versions of the KEEP algorithm.

We think that solving the repartition problem could enable us to solve other problems subject to disk seeking. The Region Of Interest (ROI) extraction problem is such a problem and is very common for scientists. A solution using chunking has been introduced in [16]. The authors define an array partitioned into chunks of equal shapes and then define a query as an arbitrary subarray of the input, chunked, array. They define the optimal chunking problem as finding the optimal chunk shape such that the expected number of chunks retrieved to answer the query is minimal. In our opinion, the solution in [16] is limited due to the need of historical or theoretical workload and the necessity to repartition the input array into an “optimal” chunk shape. We would prefer letting the application choose the appropriate chunk shape regarding its needs and not needing to estimate the processing workload.

Finally, we would like to make our algorithms accessible to scientists by implementing it into commonly used Big Data engines like Dask. As discussed previously, we did some promising experiments, that will surely be continued in the future.

# Bibliography

[1] ScienceDaily reference terms - functional neuroimaging. https://www.sciencedaily.com/terms/functional_neuroimaging.htm. Accessed: 2020-12-21.

[2] Memory-mapped files. https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files, Mar 2017. Accessed: 2020-09-27.

[3] Katrin Amunts, Claude Lepage, Louis Borgeat, Hartmut Mohlberg, Timo Dickscheid, Marc-Étienne Rousseau, Sebastian Bludau, Pierre-Louis Bazin, Lindsay B. Lewis, Ana-Maria Oros-Peusquens, Nadim J. Shah, Thomas Lippert, Karl Zilles, and Alan C. Evans. Bigbrain: An ultrahigh-resolution 3d human brain model. *Science*, 340(6139):1472–1475, 2013.

[4] Spyros Blanas, Kesheng Wu, Surendra Byna, Bin Dong, and Arie Shoshani. Parallel data analysis directly on scientific file formats. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 385–396, 2014.

[5] Dhruba Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53(1-13):2, 2008.

[6] Andrew Collette. Datasets. http://docs.h5py.org/en/stable/high/dataset.html, 2014. Accessed: 2020-09-27.

[7] Nivedita Daimiwal and Revati Shriram. Power spectral density analysis of time series of pixel of functional magnetic resonance image for different motor activity. *Biomedical and Pharmacology Journal*, 12(3):1193–1200, 2019.

[8] Khoa Doan, Amidu O Oloso, Kwo-Sen Kuo, Thomas L Clune, Hongfeng Yu, Brian Nelson, and Jian Zhang. Evaluating the impact of data placement to

spark and scidb with an earth science use case. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 341–346. IEEE, 2016.

[9] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47, 2011.

[10] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[11] V. Hayot-Sasson, Y. Gao, Y. Yan, and T. Glatard. Sequential algorithms to split and merge ultra-high resolution 3d images. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 415–424, Dec 2017.

[12] Valérie Hayot-Sasson, Shawn T Brown, and Tristan Glatard. Performance evaluation of big data processing strategies for neuroimaging. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 449–458. IEEE, 2019.

[13] Stephan Hoyer and Joe Hamman. xarray: Nd labeled arrays and datasets in python. *Journal of Open Research Software*, 5(1), 2017.

[14] Fei Hu, Mengchao Xu, Jingchao Yang, Yanshou Liang, Kejin Cui, Michael M Little, Christopher S Lynnes, Daniel Q Duffy, and Chaowei Yang. Evaluating the open source data containers for handling big geospatial raster data. *ISPRS International Journal of Geo-Information*, 7(4):144, 2018.

[15] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, Jan 2001.

[16] E. J. Otoo, Doron Rotem, and Sridhar Seshadri. Optimal chunking of large multidimensional arrays for data warehousing. In *Proceedings of the ACM Tenth International Workshop on Data Warehousing and OLAP*, DOLAP '07, page 25–32, New York, NY, USA, 2007. Association for Computing Machinery.

[17] Russell A Poldrack. Region of interest analysis for fmri. *Social cognitive and affective neuroscience*, 2(1):67–70, 2007.

[18] Matthew Rocklin. Distributed numpy on a cluster with dask arrays. `http://matthewrocklin.com/blog/work/2017/01/17/dask-images`. Accessed: 2020-12-21.

[19] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 130 – 136, 2015.

[20] Matthew Rocklin and James Bourbeau. Best practices. `https://docs.dask.org/en/latest/array-best-practices.html`, May 2019. Accessed: 2020-09-27.

[21] Florin Rusu and Yu Cheng. A survey on array storage, query languages, and systems. *arXiv preprint arXiv:1302.0103*, 2013.

[22] S. Sagiroglu and D. Sinanc. Big data: A review. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pages 42–47, 2013.

[23] Ralf Schmälzle, Martin A Imhof, Clare Grall, Tobias Flaisch, and Harald T Schupp. Reliability of fmri time series: Similarity of neural processing during movie viewing. 2017.

[24] Apache Spark. Apache spark. *Retrieved January*, 17:2018, 2018.

[25] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The architecture of scidb. In *International Conference on Scientific and Statistical Database Management*, pages 1–16. Springer, 2011.

[26] Michael E. Sughrue. Chapter 3 - novel approaches to brain mapping in the era of functional magnetic resonance imaging. In Michael E. Sughrue and Isaac Yang, editors, *New Techniques for Management of 'Inoperable' Gliomas*, pages 11 – 18. Academic Press, 2019.

[27] Shanshan Sun and Runtong Zhang. Region of interest extraction of medical image based on improved region growing algorithm. In *2017 International Conference on Material Science, Energy and Environmental Engineering (MSEEE 2017)*. Atlantis Press, 2017.

[28] Robert D Vincent, Peter Neelin, Najmeh Khalili-Mahani, Andrew L Janke, Vladimir S Fonov, Steven M Robbins, Leila Baghdadi, Jason Lerch, John G Sled, Reza Adalat, et al. Minc 2.0: a flexible format for multi-modal images. *Frontiers in neuroinformatics*, 10:35, 2016.

[29] Pan Xu, Cuong Nguyen, and Srikanta Tirthapura. Onion curve: A space filling curve with near-optimal clustering. *CoRR*, abs/1801.07399, 2018.