Sequential algorithm to split, merge and resplit multidimensional arrays

1st Timothée Guédon
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
t_guedon@encs.concordia.ca

2nd Tristan Glatard
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
tristan.glatard@concordia.ca

3rd Valérie Hayot-Sasson
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
email address or ORCID

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

Introduction

A. Context

With the improvement of aquisition methods in several scientific domains like health sciences, geology and astrophysics, processing a deluge of very high resolution images has become a Big Data challenge. Most pipelines process only specific ROI or process data by block. At the end of the pipeline, merging those blocks into one file may be required as well. Previous work in [1] introduced two types of sequential algorithms to split and merge very high resolution images: the "clustered" and "multiple" strategies. In this paper, we define a new algorithm to resplit data that has already been splitted before into an other shape, and we show that it can also be used to split and merge data with the same behavior of the "multiple" strategy. We also give an implementation of this algorithm, available as an optimization package for the Python big data library "Dask".

B. Problem definition

Consider a multidimensional array of shape $R = (R_i, R_j, R_k)$, stored in some input files with a given shape $I = (I_i, I_j, I_k)$, all input files having the same shape. Our goal is to optimize the process of sequentially resplitting the input files into output files with a different shape $O = (O_i, O_j, O_k)$, all output files having the same shape, too.

The resplit process has two particular cases:

- It becomes a split process if there is one input file and several output files,
- It becomes a merge process if there are several input files and one output file.

For this I/O process to be fast, one needs to minimize the number of seeks that occur on disk while reading and writing. We consider that a seek occurs either when opening a file or seeking into a file.

Let us consider a basic sequential resplit algorithm: One can repeatedly read the maximum amount of data possible from the input files into a buffer stored in main memory, and then write this buffer down into the output files requiring this data, until all output files have been completely written. This resplit algorithm is described in Algorithm 1.

The algorithm described in Algorithm 1 takes a list of input and output files $inFiles$ and $outFiles$ as parameters, as well as the amount of memory m available in RAM for the buffer and the list of the buffers' coordinates. We call m' the amount of memory available in the buffer at a given time during the execution of the algorithm ($m' = m$ at initialization). The list of buffers' coordinates contains the coordinates of each buffer to be loaded in the referential of the reconstructed image.

The algorithm successively loads as much data as it can from the input files into the buffer and write it down to the output files that are supposed to contain this data. Although we could use a naive shape for the buffer, we can use the input and output files shapes to elaborate more efficient strategies as we will see in the next sections. The algorithm ends when all buffers have been read. Therefore, the buffers must cover the whole reconstructed image such that when the algorithm end all the output files have been completely written. For the latter statement to be true, we also need to ensure that all data read from input files are either stored in RAM or directly written, such that all the

output files are completely written at the end of the algorithm.

Given Algorithm 1, the optimization problem that we want to solve can be stated as follows: Given the amount of main memory available m, as well as the shapes of the input and output files I and O, how to select the best buffer shape B which will minimize the number of seeks that take place during reading and writing?

We add two restrictions on the buffers: We shall use only non-overlaping buffers, all buffers having the same size, and each buffer has to be written only once.

Algorithm 1 Basic resplit algorithm

```
Inputs: inFiles, outFiles, m, buffersList
for buffer in buffersList do
    read(inFiles, buffer)
    write(outFiles, buffer)
end for
```

1) Consistency with previous works: For the sake of consistency with previous works [1], we call the original array of shape R stored in the input files the "reconstructed image" (see Figure 1). Of course, the input files' positions in the reconstructed image have to be stored in some way. Also with a view to be consistent with previous works, we assume the files to be written in column-order. In column ordering (also called "F" ordering) the fastest moving dimension is the last dimension of the array and the slowest moving dimension is the first dimension. For example a 3D array with dimensions i, j and k would be written on disk by writing the complete columns in the k dimension first (see Figure 2).

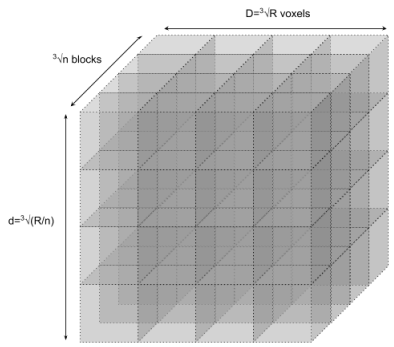

Fig. 1. Illustration of the reconstructed image divided into input files at the initialization of the resplit algorithm.

C. Naive algorithm

As a base case for our problem, let us define a naive algorithm which loads one input file at a time and write it down into the different output files that requires the data. The buffers have the same shape as the input files ($B = I$) and the order is the same order than the storage order i.e. the column order in this study ($[k, j, i]$ order).

D. A particular case

If the input shape is a multiple of the output shape such that one input file covers several output files entirely without falls, then the problem is easily solved: one must read as much input files as one, i.e. the buffer shape B is a multiple of the input shape I. The algorithm will produce one seek per input file and one seek to write each output file, which is the minimum number of seeks possible for a resplit.

If there is a mismatch between the shapes in any dimension however, one needs a strategy to manage with this overlap while minimizing the number of seeks. We will introduce a strategy to keep falls temporarily into memory in the next section, this strategy's efficiency is completely dependent on the amount of main memory available.

Also, the resplit process requires multiple buffers to be read. If there is no overlap between input and output files, then the order in which we load buffers is not important. In case of an overlap however, the order may have an important impact on the number of seeks produced.

THE "KEEP" ALGORITHM

The algorithm presented in this paper is called the "keep" algorithm, as it relies on a so-called "keep strategy" that is presented below.

E. The "keep" strategy

For a given buffer that has been filled in, we call "extra data" the data that will not be used to write from the buffer to any output files (Figure **??**). In other words, the "extra data" is the data contained in the overlap cuboids between the buffer and the output files partially covered by it. Let us define the "keep strategy" the strategy of keeping the "extra data" of a buffer in memory **??**. The main idea is to read more than necessary from the input files and to keep as much extra data in memory as possible. The goal of this strategy is to read input files in one seek, and write as much output files as possible in one seek as well. It may be that we can only keep some of the overlaps in memory as it could take too much buffers until we can write an incomplete output file in one seek without running out of memory. Extra data that cannot be kept in memory is written directly in the output file(s). By doing so we ensure that when the algorithm ends all the data has successfully been written.

F. Reading more than necessary

If we read more than necessary from the input files, we can at least write an output file completely in one seek, and we can try to keep the extra data in memory to write the next output files in the least amount of seeks, too. If we read less than one output file however, then we will have no choice but to write each output file in several seeks. At initialization, we assign a shape of $(0, 0, ..., 0)$ to the buffer. We will then stretch the buffer in each dimension until all the available memory is used

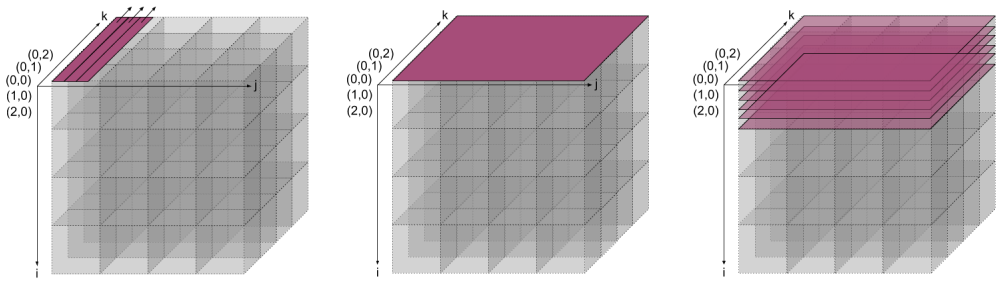

Fig. 2. Illustration of the column-order storage of voxels in a file.

while keeping the number of seeks as small as possible. In case of an overlap between the input and output files' shapes, we assume that an optimal length for each dimension i of the buffer would be the length of the maximum length between the lengths of the input and output files: $B_i = max(I_i, O_i)$.

G. Input aggregates

Let us consider a buffer sitting in the corner of the reconstructed image (see Figure **??**). We define an "input aggregate", Λ, as the rectangle composed of the minimum number of input files needed by the first buffer to cover an output file completely in all dimensions (see Figure **??**). Although it is not always possible to make it happen, the input aggregate has the optimal buffer shape we would like to have to be capable of minimizing the number of seeks as it allows the keep strategy to be used. The mismatch between the input and output shapes and the limited memory available for the buffer may prevent this ideal case to be possible.

By definition, it is possible that the input aggregate covers more than one output file completely (if $I_k > 2O_k$ for example). In this case, if the buffer shape is equal to the input aggregate shape, an overlap between the buffer and the output files will only happen between the output files on the border (see Figure x). We call "output aggregate", Ω, the aggregate of the output files that are being written by the i^{th} buffer. It includes the incomplete output files for which the missing data have been loaded by the i^{th} buffer.

H. Increasing the buffer's dimensions in the order of the fastest moving dimensions

We want the dimensions of the buffer to match the dimensions of the input aggregate as much as possible. The question of which dimension should be increased first remains. We shall now prove that one should increase the buffer's dimensions in the order of the fastest moving dimensions. For example if one is processing a 3D image stored in files following the column-order, one should increase the buffer in the k dimension first, then the j dimension and finally the i dimension.

Algorithm 2 getBufferShape in ND

1: **INPUTS**: m, Λ, B
2: $m' \leftarrow m$
3: **for** d in $dims$ **do**
4: **if** $m' >=$ maxMemoryCost(B.$add(\Lambda_{dim}), dim)$ **then**
5: B.append(Λ_{dim})
6: **end if**
7: **if** maxMemoryCost(...B, 1) $<=$ m' $<$ maxMemoryCost(B.$add(\Lambda_{dim}, dim)$ **then**
8: $maxnbcols = \frac{m'}{\text{cost}}$
9: buffershape.append($maxnbcols$)
10: **end if**
11: **if** $m' <$ maxMemoryCost(B.$add(1, dim)$ **then**
12: nb dims so far = (dim - 1)
13: nb dims left = nbdims - nb dims so far
14: buffershape = buffershape + ones(nbDimsLeft)
15: **end if**
16: **end for**
17: maxIndexes = sortIndexOfMaximumFall(B)
18: **for** $maxIndex$ in $maxIndexes$ **do**
19: **while** $m' >$ maxMemoryCost(B.$add(\Lambda_{maxIndex}), maxIndex$) **do**
20: B[maxIndex]+ = $\Lambda_{maxIndex}$
21: **end while**
22: **end for**

Details:

- line 15: sort the axes in decreasing order of overlap cuboid size
- line 16: for each dimension in this order, try to increase the buffer size
- lines 17, 18: keep adding the length of an input aggregate in that dimension while there is memory left.

ALGORITHM DETAILS

I. Increasing the buffer's dimensions in the order of the fastest moving dimensions

1) Introduction: We want the dimensions of the buffer to match the dimensions of the input aggregate as much as possible. The question of which dimension should be increased first remains. We shall now prove that one should increase the buffer's dimensions in the order of the fastest moving

dimensions. For example if one is processing a 3D image stored in files following the column-order, one should increase the buffer in the k dimension first, then the j dimension and finally the i dimension.

2) Usefulness of the keep strategy in a given dimension: Let us consider a 2D case in which the input and output files overlap in the j dimension only (see Figure **??**). Keeping the extra data in memory would only save 1 seek: Consider the case in which we read the first buffer before the second buffer. Writing the second output file would take one seek only thanks to the extra data kept in memory from the first buffer rather than two seeks, one per buffer, without using the "keep strategy". Notice that the keep strategy is not useful if we cannot store the extra data in one direction completely (see Figure **??**). In case of treating 3D cuboids the overlap would be a volume. Consider that we stretched the buffer in the j direction such that $B_j = \Lambda_j$ but the available buffer memory is too small to keep more than one input aggregate slice: $B = (\Lambda_k, \Lambda_j, 1)$. The resplit process would produce 1 seek per slice, multiplied by each buffer in the i direction. In this situation, the keep strategy would reveal to be worth it if we had enough available memory to store more slices, hence creating a volume (see Figure **??**). The keep strategy saves only 1 seek in 2D, while it saves B_i seeks in 3D. We remark that the keep strategy saves a negligible amount of seeks in the last dimension of the buffer. More generally, one can save more seeks by using the keep strategy on the first dimensions of the buffer than in the last ones.

3) Distinctions between the overlap areas/volumes: Back to the 2D case, let us now consider overlaping in both the k and the j axes: We define 3 overlaping areas, represented in Figure **??**. The a area is the overlap in the k axis. Using the "keep strategy" saves Ω_j seeks in 2D, $\Omega_j * B_i$ seeks in 3D, etc. The b area is the upper right overlap which combines the overlaps in both axis. The keep strategy saves $(\Lambda_j - \Omega_j)$ seeks in 2D, $(\Lambda_j - \Omega_j) * B_i$ in 3D etc. One can see the b area as the continuity of the a area. The c area is the overlap in the j axis only. The keep strategy saves 1 seek in the j direction. It is a negligible gain compared to saving a or b. The a becomes the A volume in 3D, the B volume corresponds to the b area, etc.

4) The order of increasing dimensions: In terms of the number of seeks,

$$\Omega_j > (\Lambda_j - \Omega_j) > 1 \Rightarrow \text{seeksIn(a)} > \text{seeksIn(b)} > \text{seeksIn(c)}$$

in 2D, and this stays true when increasing the number of dimensions. We must therefore use the memory available for stretching the buffer to store a first, then b, and finally c. The same way that it is not useful to save part of the c area as opposed to storing the whole c area, it is not useful to inrease B_j if $B_k < \Lambda_k$. Indeed, the number of seeks will be the same than without using the keep strategy (see Figure x). As the b area is the continuity of the a area in the j direction, if the available buffer memory m' allows to store the b area but not the b and c areas together it seems that increasing B_j from Ω_j to Λ_j while storing only the b area could save a lot of seeks.

As a conclusion, we should increase the fastest moving dimension first to allow the reduction of the number of seeks in the a and b areas in the k direction. Then we should increase B_j as much as possible, while saving only a and then b, even if we cannot cover it entirely. Finally, if we can store the whole c area, we should do it before inreasing the i dimension to prevent the 1 seek to be multiplied by B_i.

J. Special case of non optimality when using the order of increasing dimensions

Consider the 2D case where there is an overlap in the k dimension only, see Figure **??** above. Let us say that the buffer size is big enough to have $B_k = \Lambda_k$ for one row, but is too small to extend B_j to Λ_j. Is the keep strategy efficient in this case ? Keeping the extra data in memory will allow us to write in one seek but it implies using more buffers in the j axis to write the incomplete output file. In the worst case, $B_j = 1$ and we do as much seeks as without using the keep strategy: Λ_j seeks. As soon as $B_j > 1$ however, we divide the number of seeks produced by writing in the incomplete output file per the number of buffers. This observation is true for the other dimensions as well (see Figure increasing i before j complete): as soon as $B_x == \Lambda_x$ in one dimension x, the more we can read in the next dimension $x+1$, the better. This proves that we should not increase a dimension if $B_x < \Lambda_x$ in the preceeding dimensions.

K. Stretching beyond the input aggregate shape

Using small input and output file shapes, it is probable for the buffer to have been stretched to the input aggregate, which means that a good amount of RAM is still available. One have no reason to stop increasing the shape of the buffer from that point. The first idea which comes to mind is to stretch the buffer such that the buffer length becomes a multiple of the output file length in a given dimension x: $B_x = nO_x$. Again, this would create an overlap in the input files however, which would prevent the keep strategy to be applied and create a maximum number of seeks. Therefore, one can only stretch the buffer by adding one input file length to a given dimension of the buffer shape (see Figure **??**). A good idea would be to extends the dimension of the biggest overlap volume s.t. we accumulate extra data from the smallest overlap volumes only.

L. Ordering the buffers

Using the keep strategy in case of overlaps, one may order the buffer loadings to further reduce the number of seeks produced during the resplit process.

M. Ordering impact on performance

By optimizing the buffer ordering one can reduce the maximum quantity used to store the extra data in memory. For example, if the overlap occurs only in the k axis, loading the next buffers in this direction will enable recycling the

extra data kept in memory, resulting in a smallest memory consumption over time. The memory saved thanks to a smart ordering could enable the storage of more overlaps cuboids in memory using the "keep strategy", further reducing the overall number of seeks.

As we will see, the buffer ordering problem is complex and do not seem to be easily solvable. Thanksfully, the impact of the buffer ordering on performance can be mitigated. Indeed, the impact of the buffer ordering depends on the size of the falls, resulting from the overlaps between the buffer and the incomplete output files shapes. One can reduce the falls' sizes by using smallest chunks: Even if the overlap between the input and output files is big with respect to their size, the area/volume of the falls will be kept small. In particular, we remark that the falls tends to be smaller when the buffer shape is bigger than the output file shape, as the overlaps are smallest and concentrated on the borders (see Figure **??**). We can stimulate this property by using small chunks such that we use buffer bigger input aggregates (see Stretching beyond the input aggregate shape), while keeping the overlaps small at the borders.

A MEMORY ANALYSIS

N. Defining a "naive" buffer ordering algorithm

As stated in a preceeding section, there is not any solution of the ordering problem that would be optimal for all possible input and output shapes. For the time being we do not have a good solution of the ordering problem which do not imply a compute-intensive algorithm, therefore let us define a naive algorithm (Algorithm 3) as follows:

Algorithm 3 Naive buffer ordering algorithm

1: **INPUTS**: inputShape, outputShape
2: $overlap \leftarrow dict()$
3: $overlap \leftarrow$ getOverlapUpperBounds(inputShape, outputShape)
4: sort(overlap, by=values)
5: **return** overlap.keys()

O. Algorithm details

The algorithm return an array of dimensions ordered by decreasing overlap size. The idea is to read the buffers in the first direction where the overlap is the most important, then the second etc. For example if the order is j, k, i, we will read buffer rows in direction j, the rows being ordered in increasing 'k' (see figure x). We evaluate the amount of overlap thanks to the function getOverlapUpperBounds. getOverlapUpperBounds returns, for each dimension, the amount of memory needed to keep the extra data in memory if we read a buffer row (a row of buffers in the reconstructed image) in this dimension.

Based on this algorithm, let us try to find the function getOverlapUpperBounds.

P. Computing the overlap size in the 3D case

In this analysis we express the overlap size in terms of the number of voxels that constitutes the overlap because the real quantity of memory used depends on the size of a voxel in memory. For convenience one can avoid this extra parameter by setting the number of bytes per voxel to 1.

When loading the first buffer in a given dimension i, the overlap length $C_x(1)$ (value of C_x when loading the first buffer) is $B_x \mod O_x$ by definition of an output aggregate (see Figure x). When loading the i^{th} buffer, the overlap length is therefore $C_x(i) = i * B_x \mod O_x$ with $i \in [1, b_x]$ (b_x is the number of buffers in direction x). Let us call the index of the buffer in dimension k, j, i: x, y and z respectively.

In two dimensions, there are three overlap areas: a, b and c. When we are loading the $(x, y)^{th}$ buffer, the x^{th} buffer in dimension k which is also the y^{th} buffer in dimension j, the overlap size in voxels is:

- $a = C_k(x) * (B_j - C_j(y))$ ($C_k(x)$ is the length of the overlap in direction k for the k^{th} buffer)
- $b = C_k(x) * C_j(y)$
- $c = C_j(y) * (B_k - C_k(x))$

According to the algorithm, c can be kept in memory by the keep strategy only if there is enough memory to store a and b. The same way, b is kept in memory only if we can keep a before. When stretching the buffer shape, we start by adding as much rows as possible in the first moving direction, then the second one until we can keep b entirely, Keep in mind that c can only be stored completely, or it is not stored, as storing it only saves 1 seek.

In 2D, the overlap area in number of pixels is either (see Figure 3):

- overlap $= C_k(x) * B_j$ if we keep only a in memory.
- overlap $= C_k(x) * B_j$ if we keep a in memory as well as b or a part of b.
- overlap $= a + b + c$ as defined above i.e. we store c completely.

In 3D (see same Figure x), the overlap area is first increased into an overlap volume: overlap $= (a+b+c) * B_i$ if we cannot store d completely. Then, if $B_i = \Lambda_i$ we add the whole d volume directly. The overlap area is therefore:

$$\text{overlap} = [(a + b + c) * (B_i - C_j(z))] + d_{\text{volume}}$$

, with

$$d_{\text{volume}} = C_j(z) * C_j(y) * C_k(x)$$

Q. Computing the maximum amount of memory needed for the keep strategy

Maximum length of the overlap in a given dimension
By definition of the modulo, the maximum overlap length in a given dimension x is O_x. O_x is an upper bound as in some cases like in Figure x we may never reach this point. We could therefore compute $k * B_x \mod O_x$ for all $k \in [1, b_x]$ to find the maximum if it is not too compute intensive in order

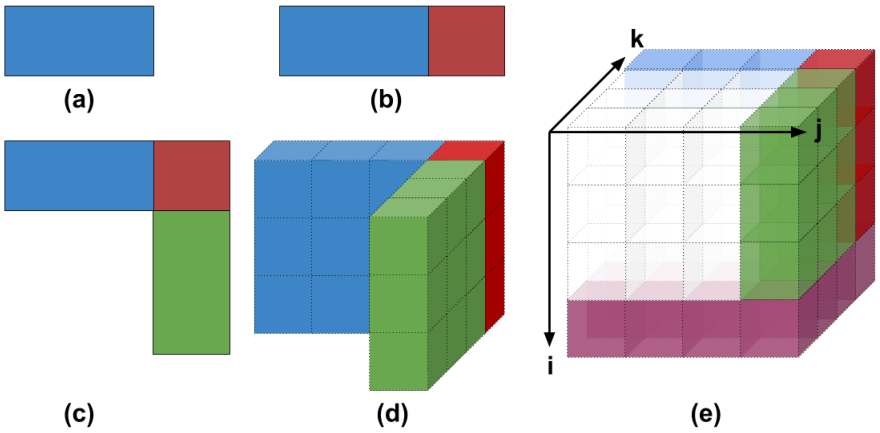

Fig. 3. Possible overlaps when resplitting 3D images. (a): Covering the a area. (b): Covering the b area. (c): Covering the c area. (d): Converting overlap areas into overlap volumes by stretching the buffer in the i direction. (e): Further increasing B_i creates a d overlap volume.

to find the exact maximum length of the overlap in direction x.

Maximum overlap size

Now that we know how to compute the maximum length of the overlap, we can use the overlap sizes found in section "Computing the overlap size in the 3D case" and replace the Cs by their maximum value to get the maximum volumes we can expect for a, b, c and d during the algorithm's execution. Given a buffer ordering, we may have to store several times one or more volumes at the same time in memory.

Total overlap size at a given time

Using the naive ordering, we read buffers in the direction of the biggest overlap volume first. For example, if the buffer shape is the same as the shape of the input aggregate and if $volumes(k) > volumes(j) > volumes(i)$, then we read the buffers in the k direction first. The amount of memory needed to read in the k direction is $m_{\text{needed}} = B.size + volumes(k)$:

- At step $t = 1$, the first buffer is read, and there is no preceeding overlap.
- At step $t > 1$, we have the $C_k(t-1)$ overlap in main memory. We need to read the t^{th} buffer first in order to get rid of the previous overlap and store the new one.
- The last buffer being read in the k direction has no overlap by definition of the resplit process: everything is written down, including the previous overlap $C_k(t-1)$.

During the buffering process in direction k, we must store the overlap volumes in the j and i directions, too, according to the keep strategy in order not to write it down which would incur a lot of seeks. In 3D, when loading the buffer in position (k, j, i) in the referential of the buffers (see Figure 4), the overlap size is:

$$MAX = bufferSize + C_k(x) + [kC_j(y) + (n_k - k) * C_j(j-1)] + [n_s * C_j(z) + (N_s - n_s) * C_j(z)]$$

- N_s is the number of buffers in a slice.

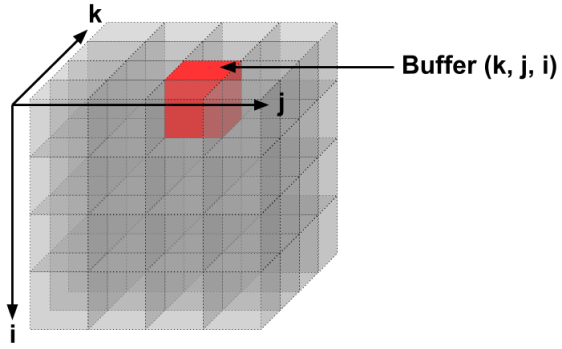

Fig. 4. Buffer in position (k,j,i) in the referential of the buffers.

- n_s is the number of buffers read so far from the current slice.

Maximum amount of memory needed for the process

Defining x_{max} and y_{max} the index at which we are storing the biggest overlap in k axis and the index of the buffer row at which there is the biggest overlap in j, respectively, we know that the case (x_{max}, y_{max}) will happen. The same way, the case $(x_{max}, y_{max}, z_{max})$ will also happen. Therefore, the maximum memory size we will need during the whole process is MAX with all C replaced by their maximum possible value: either the upper bound or the computed value as explained in the preceeding paragraph.

IMPLEMENTATION

EXPERIMENTS

DISCUSSION

R. Solution of the ordering problem

Our goal is to order the buffers so that we use the least memory as possible through the entire resplit process. In other words, we want to minimize the maximum amount of

main memory used during the respit.

1) Memory deltas: One can model the sequential buffer loading problem as a doubly linked directed complete graph where each node is a buffer. Exchanges of data happen in main memory as we walk sequentially through the nodes using the "keep strategy": At each node we get rid of some extra data loaded from previous buffers and we get some extra data from the new buffer. We call this difference between the amount of data in main memory before and after passing through a node a "memory delta".

2) Modeling the problem as a shortest path problem: We will prove in the next paragraph that minimizing the maximum amount of memory during the resplit process is equivalent to finding the shortest path through all nodes of the previously defined graph with the "distance" (weight on the edges between nodes) being the memory deltas. The algorithm must go through each node only once as we want to load all the buffers, only once (due to the non-overlaping buffer constraint). Note that weights can be negative if we give more extra data than we receive at a given node. Also note that weights update each time a buffer has been loaded: All the buffers that will recycle the extra memory from the current buffer see the weights of their incoming edges decrease.

3) Minimizing the maximum amount of memory is equivalent to a shortest path problem: The idea of the proof is that if the solution of one problem is the solution of the other, then they are be equivalent.
To do.
Note to self: An optimal solution of the shortest path problem with memory deltas as distances is a path that minimizes the sum of the memory deltas encountered during the process. Let us take such an optimal path and consider a sequence of the main memory used at each step on the path.

4) Conclusion: The buffer ordering problem is an all-pairs (we can start from any node and and at any node) dynamic (the weights update during the process) shortest-path problem. According to a review by Madkour et al. [2], Demetrescu and Italiano proposed a "Fully dynamic algorithm over directed graphs for all-pairs shortest-paths with real-valued edge weights" [3] to solve such a problem. Their solution seems to have been improved by Thorup in his paper "Fully-Dynamic All-Pairs Shortest Paths: Faster and Allowing Negative Cycles" [4]. The amortized update time is $O(n^2 log^3(n))$ which is not a small overhead (Book of algorithm theory 2004). Proving the benefit of using such an algorithm reveals to be a complex time, that is why we decide to leave it as a future work and use a naive buffer ordering for the moment.

S. Comparison with previous work

1) Split behavior of the algorithm: The split process is equivalent to the resplit process with the special case of having only one input file. In this case, the input aggregate has the same shape than the reconstructed image. The buffer will first be stretched in the k direction until $B_k = \Lambda_k = \mathrm{R}_k$. Then we will stretch in the j dimension until $B_j = \Lambda_j$ and the same behavior applies in the i^{th} dimension. Therefore, we first store complete rows, then block row tiles, then slices, block rows and slices. This algorithm is identical to the "multiple reads" algorithm (see Hayot-Sasson et al.).

2) Merge behavior of the algorithm: The merge process is equivalent to the resplit process with the special case of having only one output file. In this case, the input aggregate has the same shape than the output file shape. The buffer will be stretched in the k axis until $B_k = \Lambda_k = \mathrm{R}_k$. Again, this is the same behavior than multiple reads.

CONCLUSION

ACKNOWLEDGMENT

REFERENCES

[1] 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)*, pp. 415–424, Dec 2017.

[2] A. Madkour, W. Aref, F. Rehman, A. Rahman, and S. Basalamah, "A survey of shortest-path algorithms," 05 2017.

[3] C. Demetrescu and G. F. Italiano, "Fully dynamic all pairs shortest paths with real edge weights," *Journal of Computer and System Sciences*, vol. 72, no. 5, pp. 813 – 837, 2006. Special Issue on FOCS 2001.

[4] M. Thorup, "Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles," in *Algorithm Theory - SWAT 2004* (T. Hagerup and J. Katajainen, eds.), (Berlin, Heidelberg), pp. 384–396, Springer Berlin Heidelberg, 2004.