Dask IO for sequentially splitting, merging and resplitting multidimensional arrays

Timothée Guédon, Valérie Hayot-Sasson, Tristan Glatard
Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada

***Abstract*—Todo.**

***Index Terms*—multidimensional, array, split, merge, resplit, IO, processing, Dask, Python**

I. Introduction

With the improvement of acquisition methods in several scientific domains such as health sciences, geology and astrophysics, new big data challenges have emerged. Such challenges includes processing big amounts of data and manipulating heavy files like ultra high resolution images. Big Brain, a brain model providing microscopic data (20 micrometers) for modeling and simulation [1] is an example of ultra high resolution images in the neuroscience field.

A. Multidimensional array chunking

On the one hand, scientific applications often model problems as multidimensional arrays that must be stored in chunks for later processing and analysis. Chunking allows for efficient queries and great flexibility in adding new data, among other things. On the other hand, block algorithms are necesary when data cannot be entirely loaded in memory and to parallelize computations.

Multidimensional array chunking raises the need for tools to efficiently split, merge and "resplit" or "rechunk" data files. Previous work in [2] show that naive algorithms to split an array into several files and merge back those files into one output file perform very poorly due to millions of seeks occuring on disk.

B. The occurrence of seeks

For this study we manipulate multidimensional arrays stored in hdf5 files, that is why we assume that files are written in row-major order (a.k.a "C" ordering), where the fastest moving dimension in the file is the last dimension in the array and the slowest moving dimension in the file is the first dimension in the array. 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. As in previous work [2], we assume that the storage layout can be arbitrary as long as it is known to the algorithm.

Retrieving subarrays from an array stored on disk incurs seeks. When accessing a multi-dimensional array, seeks occur in two situations: (1) when an array block is opened for reading or writing, and (2) when the reading or writing process moves within the block to write at different places. While reading the data of interest, one needs to seek to the beginning of each contiguous piece of data constituting the subarray. For a 3D cuboid of shape C, stored in row-major order, each column of data in the k dimension is contiguously stored. They are therefore Cj * Ci data columns for which a seek is required on disk. Although the seek time depends on various parameters such as the distance in bytes between two data columns, we assume all seeks to incur the same time overhead for the sake of simplicity. We further assume that reducing the number of seeks is representative of reducing the processing speed.

C. The multiple and clustered strategies

Two strategies have been introduced in [2] for the split and merge tasks. To illustrate these strategies we shall apply it to the split task (R = I) as it is considered a dual operation of the merge task. The naive strategy to split an array is to repetedly loading one subarray of shape O and writing it down into an output file [2].

The clustered strategy consists in loading contiguous blocks of data of shape O from the input array in order to write it in one seek in each output file which data has been loaded. This strategy loads data blocks one by one, block columns by block columns or blocks slices by block slices. Using this strategy for resplitting would imply reading multiple input files that are contiguous in the reconstructed image or reading the data of contiguous output files by seeking into input files. In either case one can easily find bad cases in which the algorithm seeks in all dimensions.

The multiple strategy aims at not doing any seek while reading and writing. For example, one would read the input file slice by slice and write this slices contiguously in the output files. The tradeoff lies in switching between files at each buffer loading. Using this strategy for the resplit task would imply reading contiguous columns or slices of data from the input files and writing it contiguously in the output files. This would result in even more switches between input and output files and one can easily find worst cases when using small I and O shapes for example where a smarter strategy would be appreciated.

To the best of our knowedge however, no algorithm has been proposed for the resplit task.

D. Problem definition

We focus on 3D arrays for simplicity. Consider a 3D array of shape $R = (R_i, R_j, R_k)$, stored as input blocks of uniform shape $I = (I_i, I_j, I_k)$. Our goal is to resplit the input blocks into output blocks of uniform but different shape $O = (O_i, O_j, O_k)$.

A resplit algorithm (Algorithm 1) takes as input a list of input blocks `inBlocks` of shape I, a list of output blocks `outBlocks` of shape O, the amount of available memory `m`, and a list of buffers `buffersList` defining a partition of array R. The algorithm successively loads the image regions defined by the buffers, and writes them to the output blocks.

For simplicity, we require that all buffers in `buffersList` have the same shape. Our problem is to find the partition `bufferList` that minimizes the number of seeks done by Algorithm 1 given I and O, subject to the amount m of available memory.

Algorithm 1 General resplit algorithm

```
Inputs: inFiles, outFiles
$B \leftarrow I$
cache.initialize()
for buffer in getBuffers($B$) do
    bufferData ← read(inFiles, buffer)
    cache.insert(bufferData)
    for $i$, data in cache do
        write(outFiles[$i$], data)
    end for
end for
```

E. Baseline solution

A naive, baseline algorithm for the resplit task is to load one input file at a time into memory and write it in the appropriate output files. We can compute the amount of seeks produced by this algorithm with the algorithm described in Algorithm 2. As one can see from the algorithm, unless the dimensions of the input and output files matches a considerable amount of seeks will occur. Also, an overlap in the k dimension is more costly than one in the j dimension. Finally, if the output file shape is greater than the input file shape it also incurs the maximum amount of seeks.

F. Considerations on shape mismatch

If $I_x \neq O_x$ in a given dimension x, we call it a shape mismatch. As it has been shown in the paragraph on the baseline algorithm, if $I_x < O_x$ and we read one file at a time, we are condamned to seek in the output files at write time. If, however, $I_x > O_x$, then we get remainders. Finally, if we just read what we need from the input files to write in output files contiguously, we seek while reading and writing. A solution to reduce the number of seeks seems to be to read more than O_x and to elaborate a strategy to keep the remainders in memory instead of writing it down directly until we get enough data to write it with a reduced amount of seeks. Such a strategy is presented in the section keep algorithm.

G. Contributions

This paper makes the following contributions:

- We define the resplit problem

Algorithm 2 Baseline algorithm for the resplit task

```
Inputs: $all_i$, $all_j$, $all_k$
$nb_{seeks} \leftarrow 0$
for $i$ in (len($all_i$)-1) do
    for $j$ in (len($all_j$)-1) do
        for $k$ in (len($all_k$)-1) do
            $L_i = all_i[i+1] - all_i[i]$
            $L_j = all_j[j+1] - all_j[j]$
            $L_k = all_k[k+1] - all_k[k]$
            if $I_k \geq O_k$ then
                if $L_k = O_k$ then
                    if $L_j = O_j$ then
                        $n = 1$
                    else
                        $n = L_i$
                    end if
                else
                    $n = L_i * L_j$
                end if
            else
                $n = L_i * L_j$
            end if
            $nb_{seeks} += n$
        end for
    end for
end for
return $nb_{seeks}$
```

- We propose a first sequential algorithm to efficiently resplit multidimensional arrays
- We give a public implementation of such algorithm in the Python library dask
- We enable the use of the clustered strategy for splitting and merging multidimensional arrays in dask

II. The "keep" algorithm

The keep strategy uses a cache internal to the application to keep remainders in memory. It includes an algorithm to find the best buffer shape in order to always try to have remainders, i.e. $B_x > O_x$. As we are constrained by the amount of memory available, it may be that we cannot keep some remainders. In that case those remainders are directly writen down into the appropriate output files to avoid re-reading input files several times and ensure that all the data has been written at the end of the algorithm. Being able to keep all remainders into memory to write an output file only when all the data has been loaded in memory means doing the minimum number of seeks possible during a resplit task which is equal to $n_i + n_o$, with n_i the number of input files and n_o the number of output files. In the following paragraphs we explain how to find the best buffer shape according to the keep strategy.

A. Input aggregates

As explained, our goal is to find a buffer shape such that $B_x > O_x$ in any direction x to be able to use the keep strategy hence reducing the number of seeks. If $I_x > O_x$, then B_x can be set to I_x. If $I_x < O_x$ however, one needs a buffer that is a multiple of I_x in order to minimize the number of seeks at read time and have remainders ($B_x > O_x$). We define an input aggregate as being the minimum aggregate of input files that covers one output file completely. In particular, it is the input aggregate that covers the first output file (indexed $(0, 0, 0)$) in the storage reconstructed image. The best buffer shape for the keep strategy is therefore the shape of an input aggregate. We call this shape Λ.

B. Stretching the buffer in the storage order

To get a buffer close to Λ we define an algorithm that stretches the buffer shape step by step. This step by step stretching

C. Impact of the buffer order on performance

Using the keep strategy in case of overlaps, one may order the buffer loadings to further reduce the number of seeks. By optimizing the buffer ordering one can reduce the maximum quantity used to store the extra data in memory. For example, if an 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 in memory using the "keep strategy", further reducing the overall number of seeks.

As we will see, the buffer ordering problem is complex and does not seem 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, i.e. 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.

III. A MEMORY ANALYSIS OF THE KEEP STRATEGY

This section covers how we estimate the amount of memory required by the keep strategy to know how much the buffer can be stretched in a given direction while keeping the remainders in memory. In this analysis, we express the quantity of memory used by an array as the number of voxels it contains. It is equivalent to saying that the number of bytes per voxel is 1.

The amount of memory required is the maximum memory consumption reached during the execution of the keep algorithm. To estimate this amount, we first need to define what a remainder volume is. A buffer of shape Λ can be divided in 8 parts or "volumes". 7 out of those 8 volumes are remainder volumes because they represent the overlap between the input aggregate and the output files on its border. These 7 remainder volumes enclose an 8th part that is composed of input files that are either complete or which complementary part have already been loaded by a previous buffer. This means that any complementary part is either in memory (it has been kept in cache according to the keep strategy) or it has been written down previously. Therefore, this 8th volume cannot be kept by the keep strategy and is not considered a remainder. For each buffer, each of volume is indexed following the buffer order (see section on buffer order) G_0 to G_7, with G_0 being the non-remainder volume. If the buffer is smallest than the input aggrgegate or if there is no shape mismatch in a given dimension, it may be that a volume size is set to 0.

Having partitioned the buffers into such volumes, we can see that the maximum memory consumption is computed from two pieces of information: The maximum number of each volumes we must keep during the process and the maximum size of each volume. The maximum number of each volumes that we must keep during the process is (see computation details in Appendix A):

$$\sigma = G_1 + n(G_2 + G_3) + N(G_4 + G_5 + G_6 + G_7) \qquad (1)$$

With n the number of buffers in the first direction of the buffer order and N is the number of buffers in a buffer slice, i.e. in the second direction of the buffer order. Given that at each step of the resplit algorithm one must load a buffer, we should also add the size of a buffer to the equation:

$$\Sigma = (G_1 + n(G_2 + G_3) + N(G_4 + G_5 + G_6 + G_7) + B_i B_j B_k) \qquad (2)$$

Finally, to find the maximum size of each volume one must know the maximum overlap length between the buffer and the output files on its border. One can either compute all possibilities and take the maximum in each dimension if it is not too costly or use the following upper bound: By definition of an input aggregate, the overlap between any buffer and its bordering output files it at maximum Ox in dimension x.

Given equation 2 and the maximum size of the volumes, we can now stretch the buffer shape one dimension at a time using simple equations to ensure that the maximum amount of memory consumed during the process wil stay below the amount of memory available. Such computations and the associated buffer stretching algorithm are given in Appendix A.

IV. METHODS

V. IMPLEMENTATION DETAILS

VI. RESULTS

VII. DISCUSSION

- stretching beyond input aggregate
- the split and merge tasks are special cases of the resplit task, which gives an opportunity to solve the three problems at once. show that equivalent to multiple reads/writes when used for split/merge

A. *Future work*

- explore tradeoff multiple and keep strategies: more or less buffers

VIII. Conclusion

IX. Acknowledgments

References

[1] K. Amunts, C. Lepage, L. Borgeat, H. Mohlberg, T. Dickscheid, M.-É. Rousseau, S. Bludau, P.-L. Bazin, L. B. Lewis, A.-M. Oros-Peusquens, N. J. Shah, T. Lippert, K. Zilles, and A. C. Evans, "Bigbrain: An ultrahigh-resolution 3d human brain model," *Science*, vol. 340, no. 6139, pp. 1472–1475, 2013.

[2] 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.

Appendix A
Buffer extension algorithm

TODO