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 [2]. On the other hand, block algorithms are necessary when data cannot be entirely loaded in memory and to increase computation speed using parallelism.

Multidimensional array chunking raises the need for tools to efficiently split, merge and "resplit" or "rechunk" data files. Previous work in [3] 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. Dask

Dask is a Python package enabling parallel and out-of-core computation. It is a specification that encodes computations into a task graph which is then dynamically executed by one of several schedulers offered by dask: the single-threaded scheduler, multi-threaded scheduler, multi-process and distributed schedulers. Custom schedulers can also be implemented. This specification can be used out-of-the-box or by the use of built-in collections or "APIs". For example, dask.array is a "parallel and out-of-core Numpy clone". `dask.dataframe` is a pandas clone, respectively. A dask graph can be implemented in plain Python as a dictionary, with any hashable as a key and Python objects 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.

We focused ourselves on the dask.array collection that is designed for multi-dimensional array processing, using blocked algorithms. The advantage of implementing our work for use with dask is that dask has a big scientific community that is likely to benefit from our work. Moreover, although dask has focused its recent work on distributed systems and machine learning, one of its first goal was to enable big data analytics from an average notebook. That is the reason why we decided to first focus or sequential algorithms for the local schedulers.

C. 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 [3], 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 $C_j * C_i$ 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.

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 files `inFiles` of shape I, a list of empty output files `outFiles` of shape O and the amount of available memory `m`. Based on a given strategy and taking m into account, the algorithm first define an ordered list of buffers

positions `buffersList` defining a partition of array R. The algorithm then successively loads the array regions defined by the buffers, and writes them into the output files. As an intermediary storage, we define a cache that distributes the buffer data to the appropriate output files.

For simplicity, we require that all buffers in `buffersList` have the same shape B. 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.

The minimum number of seeks possible for the resplit task is n_i+n_o, with n_i the number of input files and n_o the number of output files. It means reading each input file and writing each output file in only one seek each.

Algorithm 1 General resplit algorithm

```
Inputs: inFiles, outFiles, m
buffersList ← getBuffers(I,O,m)
initialize(cache)
for buffer in buffersList do
    bufferData ← read(inFiles, buffer)
    cache.insert(bufferData)
    for i, data in cache do
        write(outFiles[i], data)
    end for
end for
```

E. The multiple and clustered strategies

Two strategies have been introduced in [3] 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 load one subarray of shape O and write it down into an output file [3].

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, depending on the amount of main memory available for the buffer. 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 these 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. The algorithm presented in this study can be seen as a smarter adaptation of the multiple strategy to the resplit task.

F. Baseline solution

A naive, baseline algorithm for the resplit task is to load one input file at a time. It is equivalent to replace `getBuffers` by I in Algorithm 1. 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 $O_x > I_x$ in a given dimension x, it also incurs the maximum amount of seeks.

Algorithm 2 Baseline algorithm for the resplit task

```
Inputs: all_i, all_j, all_k
nb_seeks ← 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 ≥ 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
```

G. Contributions

This paper makes the following contributions:

- Definition of the resplit problem
- Proposition of a first sequential algorithm to efficiently resplit multidimensional arrays
- Release of a public implementation of such algorithm using the Python big data library dask in a library called Dask IO.

- Release of the clustered strategy for splitting and merging multidimensional arrays in Dask IO, too.

II. The "keep" algorithm

A. 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 this section.

B. The keep strategy

The keep strategy uses a cache to keep remainders in memory. The strategy requires 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 of the remainders. In this 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 ($n_i + n_o$). In the following paragraphs we explain how to find the best buffer shape for to the keep strategy.

C. 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.

D. 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. At each step, the algorithm estimates how much the buffer can be stretched in one direction while keeping the remainders into memory. Thhe problem of what dimension to increase first still stands. A strategic order seems to be to stretch the buffer in the direction which saves the maximum number of seeks first. Given the analysis of the baseline algorithm, such an order is the storage order as in the case of a row-major ordering such an order would keep remainders in the k dimension first. An analysis of the memory consumption of the keep strategy at each step together with the associated algorithm is given in the next section.

E. 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 remainders. One can reduce the remainders' 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 remainders 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):

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

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:

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

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

Before diving into the implementation of the keep algorithm we first designed an experiment to prove that dask.array was subject to the seek problem. To that aim we created a random array from a uniform distribution and stored it as a hdf5 file for later processing. The array had a shape of x which represents 1/8 of the Big Brain size. The experiment then consisted in splitting and merging the array with dask using two different chunk shapes: one of them doing more seeks than the other. We did this experiment on both HDD and SSD to see if the results were also visible when using an SSD which is suposedly less affected by seeks. The results of this experiment showed that dask was indeed subject to the seek problem. See the "Results" section for more details.

In order to see if a strategy to reduce the number of seeks was effective, we implemented and tested the "clustered" strategy for splitting and merging multidimensional arrays. We chose this strategy first because it seemed simpler to implement and study [3] showed that such a strategy was working. It means that if it failed the problem would be more likely to come from the implementation or dask than from the algorithm itself. We also know from [3] that the difference between the naive and clustered strategies should be large enough to be visible in the processing time. The shapes used for the experiment are shown on the table x. The implementation details of the clustered strategy are discussed in the appropriate section and again, the results are presented in the "Results" section. The code of the experiment is available in a dedicated repository on Github.

After these two preliminary experiment, we implemented a simplified version of the keep algorithm due to the difficulty to implemlent such an algorithm in a task graph (see section "Implementation details"). In order to evaluate the efficiency of the keep algorithm we compare it against vanilla dask on local computer. To be fair we used dask with one thread as required for our implementation but also with the classic multi-threaded scheduler as it is the default when using dask.array. In order to compare both implementations to a naive resplit, we also added a plain Python implementation to the benchmark. For the experiments presented below we built a random 3D array of the size of Big Brain, drawn from a uniform distribution. Once again, the array is initially stored into an hdf5 file.

We did three experiments on the keep algorithm. The first one consisted in comparing the different implementations in the case where the keep algorithm is supposed to be the most effective i.e. when the buffer is equal to the input aggregate $B_x = \Lambda_x$. In particular, three interesting cases are when $B = (1, 1, \Lambda_k)$, $B = (1, \Lambda_j, \Lambda_k)$ and $B = \Lambda$. Those three cases represent ideal cases with different amount of data available. This experiment aimed at (1) showing if the keep algorithm worked, (2) seeing how much it can improve the processing speed and (3) how much gain it represents to be able to stretch the buffer in each dimension.

In the second experiment, a more realistic case has been tested. In this case, $O = I$ which is more likely to happen in real life. Although in a realistic scenario O and I would be relatively small, we also tested bigger shapes to see the impact on performance and if one implementation was more resilient than another.

Finally, the particular case of an important shape mismatch was tested. The cases where $O >> I$ and $O << I$ are equivalent, as in both cases the input aggegate will be quite big. This is supposed to be a bad case for the keep algorithm as extending the buffer to Λ will be more complicated. For this experiment, we fixed m and I and then ran multiple times the algorithms while increasing O.

V. Implementation details

A. Clustered strategy

As explained in the introduction, daks represents computations by a task graph. Implementing the clustered strategy consisted in modifying that graph. To that aim, we implemented an optimization function to create "buffer" tasks in the graph.

The optimization starts by converting the task graph into a mathematical directed graph for use by a BFS (Breadth First Search) algorithm. More precisely, we implement it as adjacency lists. The BFS algorithm allows us to find and store tasks by type for later processing. Assuming that the whole array is being processed, we find the appropriate buffering scheme from I, O and m, according to the algorithm in [3]. The buffering scheme specifies if the algorithm loads buffers block by block, block columns by block columns or block slices by block slices.

We then update the data loading tasks to force dask to load a whole buffer at a time instead of small chunks. Finally, we modify the dependent tasks to make them use our new buffer tasks. According to the clustered algorithm, all buffers do not necessarily have the same shape, that is why we could not use the `dask.array.rechunk` method.

The default scheduler for dask.array is the multi-threaded scheduler. It means that at the beginning of the execution dask dispatches some tasks that have no dependencies into different threads. The problem is that the clustered strategy tries to use all the memory available for each buffer, and such buffer tasks are the one with no initial dependencies. Therefore, we needed to modify dask's scheduler to make it load one buffer only after the previous one and its dependent tasks have ended. This avoid running out of memory during the processing.

B. The keep algorithm

The buffer extension algorithm described in Appendix A has been direclty implemented in plain Python. Implementing the last part of the algorithm consisted in: (1) creating the buffer tasks, (2) dispatching the different data parts loaded by each buffer into the appropriate output files, (3) simulating the cache to keep remainders in memory between buffer loadings.

This time, the buffer task creation was straightforward as it is assumed by the algorithm that the buffers constitute a partition of the whole array. For the second and third part, we used the `dask.array.store` method that requires input arrays as argument, together with the output files in which to store the data and the indices of each data part in the output files. To automatically compute such information, we proceeded as follows. We first compute, for each output file, the list of each constitutive subarrays, with subarrays defined by the intersection of input and output files and buffers. When each output file "knows" which part from which buffer are required, we merge some of them according to the keep strategy to enforce the fact that a given data part of a given output file has to be stored at once. By doing this we create dependencies in the task graph. Such dependency tells dask that it needs to load two buffers before writing a given subarray for example.

C. Dask IO

Both implementations are available in a same Python package called "dask IO". It contains tests with 99% coverage and is available on PyPI (The Python Package Index). Dask IO provides a simple API to use the split, merge and resplit algorithms described in this study (figure x).

VI. Results

VII. Discussion

A. The buffer ordering problem

The buffer ordering problem can be modeled as follows: We can reprensent the problem as a complete bidirectional graph in which each vertex is a buffer. At initialization, we must choose a first buffer as an entry point. Let us define a path in the graph as visitation order of all buffers in the graph. One buffer cannot appear twice in the list. Finally, let us define a cache that contains the data currently in main memory. We define the buffer ordering problem as the problem of finding the optimal path such that the amount of memory used by the cache is kept minimal during the process. For each buffer, we load some data into the cache and free some of it (writing into output files). The amount of data released is different depending of the buffers previously visited and each buffer can only be visited once which can be reprensented by removing all edges pointing to a visited vertex except the one coming from the previously visited vertex. One may solve this problem by using a greedy algorithm but it would incur more infering at runtime and a potentially complex algorithm to run. We decided to keep things simple using a naive buffer order, the storage order, as the goal of this study is primarily to assess if the keep algorithm works.

B. Breaking the buffers

To even reduce the maximum amount of memory used during the resplit process, we could read the input files one by one instead of enforcing dask to read a whole buffer at a time, provided that the input files' parts are read in the right order (following the buffer order). The idea of reading an entire buffer at a time came from the clustered and multiple strategies but we realized afterwards that this could be beneficial to break the buffers into input files' parts. We can see in Equation (2) that it would result in replacing $B_iB_jB_k$ by $I_iI_jI_k$ or less if part of a file is loaded. This idea would only be beneficial in the case where $B_x > I_x$. Indeed, in the other case, the buffer is loaded anyways.

C. ROI extraction problem

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] E. J. Otoo, D. Rotem, and S. 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, (New York, NY, USA), p. 25–32, Association for Computing Machinery, 2007.

[3] 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