Sequential algorithms to split and merge ultra-high resolution 3D images

Valérie Hayot-Sasson*, Yongping Gao*, Yuhong Yan, Tristan Glatard
Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
first.last@concordia.ca
* These authors have contributed equally

***Abstract*—Splitting and merging data is a requirement for many parallel or distributed processing operations. Naive algorithms to split and merge 3D blocks from ultra-high resolution images perform very poorly, as a result of seek times. In contrast, naive algorithms to split and merge 3D slabs perform optimally as seek time is significantly minimized. We introduce and analyze sequential algorithms (Clustered reads, Multiple reads, Clustered writes, and Multiple writes) that leverage memory buffering to address this issue. Clustered reads and Clustered writes, access image chunks only once, but they have to seek in the reconstructed image. Multiple reads and Multiple writes minimize seeks in the reconstructed image, but they access image chunks multiple times. Evaluation on a 3850x3025x3500 brain image shows that our algorithms perform similarly to the optimal configuration provided that enough memory is available. Additionally, Multiple reads supports on-the-fly compression of the merged image transparently but Clustered reads does not, due to its use of negative seeking. We conclude that splitting and merging large 3D images can be done efficiently without relying on complex data formats.**

I. Introduction

Three-dimensional images that exceed typical memory size are increasingly found in a variety of disciplines. Big Brain, for instance, is a 3D histological image of the human brain that represents 1 TB of raw data organized in 3600 planes at full resolution and 76 GB at a 40-micrometer isotropic resolution commonly used in neurosciences [1]. Other examples found in medical imaging, our primary domain of interest, include high-resolution 3D electron microscopy (see, e.g., [2]) or micro- and nano-tomography [3]. As such images would typically be processed on a computing cluster, possibly using locality-aware file systems such as the Hadoop Distributed File System (HDFS [4]), software libraries are needed to split and merge them efficiently, in particular to limit file seeks. In this paper we introduce and compare sequential algorithms to split and merge images with reduced seeking.

We assume that the high-resolution image is split into chunks representing 3D blocks or 3D slabs that fit in memory. A 3D block consists of a stack of one or more 2D tiles (incomplete slices or incomplete columns) spanning contiguous slices, whereas a 3D slab is a series of one of more complete contiguous 2D slices. A dataset such as Big Brain would perhaps be split into 125 chunks of 600 MB. The decision to split an image into slabs or blocks, and the size of the chunks, is up to the application. For instance, spatial filtering would commonly require blocks, whereas slabs might be preferable for acquisition artifact removal. Applications that process voxels individually, for instance histogram computation or k-means clustering, could work on either slabs or blocks. Flexibility is thus required in the splitting scheme.

We also assume that the byte organization in image files is arbitrary but known to the algorithm. Some formats, for instance the file format defined by the Neuroimaging Informatics Technology Initiative (NIfTI – https://nifti.nimh.nih.gov) store the complete image in column-major order, that is, elements belonging to the same column are stored in a contiguous order. Other formats, such as HDF5-based MINC 2.0 [5], provide more flexibility by allowing data to be partitioned in limited-sized chunks, each chunk being stored in a specific order. Byte organization is obviously a critical factor of seek time. The main idea of our algorithms will be to rearrange ordering in memory before or after I/O operations.

The literature on this problem is remarkably scarce. Parallel and distributed image processing has obviously been extensively studied and used in various platforms [6], [7], [8], [9], [10], [11], but methods have focused on geometrical approaches to partition images, and on load-balancing or task scheduling techniques. Instead, we aim at algorithms to efficiently split or merge images regardless of the geometry of the chunks. Although seek times are often identified as an issue, the preferred solution is usually to optimize data storage formats for a certain application. For instance, the Open Connectome Data Cluster [12] is a data warehouse system that allows users to retrieve specific 3D blocks from large image datasets. It reduces seek times through a specific file format based on space-filling curves, which elegantly preserves spatial proximity on disk. On the contrary, we are searching for algorithms that would reduce the seek time regardless of the data format, such that applications with flexible splitting schemes can be served.

To summarize, our paper makes the following contributions. (1) We propose a set of algorithms to split and merge large 3D images from 3D blocks or 3D slabs. (2)

We determine the complexity of those algorithms in terms of numbers of seeks, as a function of the image size, splitting scheme and available memory. (3) We evaluate our algorithms using the Big Brain dataset and two storage drives with different characteristics. Section II presents our algorithms, Section III describes their implementation, Section IV reports experimental results and Section V concludes the paper.

II. Algorithms

Split and merge relate to the same dual problem in our context. We focus here on merging for the sake of concision. Splitting algorithms can be derived from merging ones by swapping reads and writes. Our goal then is to merge a set of n chunks into a single reconstructed 3D image with R voxels of size b. For simplicity, we assume that chunks are non-overlapping cuboids that all have the same dimension.

Although our algorithms could be applied to any byte organization, we consider a file format where voxels are written in column-major order. All voxels in a *slice* have the same k and all voxels in a *column* have the same j.

A. Notations

We adopt the following notations (see Figure 1):

- $R = D^3$: number of voxels in the reconstructed image.
- b: number of bytes per voxel (in B).
- n: number of chunks (blocks or slabs).
- m: amount of available memory (in B).
- $m' < m$: amount of used memory (in B).

We also have the following relations:

- Number of slices/rows/columns in a block: $\sqrt[3]{\frac{R}{n}} = d$.
- Number of blocks in a block column: $\sqrt[3]{n}$.

B. Disk model

A disk is characterized by its read and write rates, its access time and its seek time. For common file sizes, seek time is negligible compared to read or write time as typical seek times range from about 0.1 ms for Solid-State Drives (SSD) to 10 ms for Hard-Disk Drives (HDD). However, as we will shown later, naive algorithms might seek up to 10^7 times to merge a high-resolution image, which renders total seek time comparable to read and write times. In addition, extensive seeking also has an effect on read and write rates, as these are typically increasing with the duration of uninterrupted reads or writes.

In our analysis, we do not distinguish between access time and seek time. We also assume that seeks require a constant amount of time, regardless of the position seeked to. That is, we focus on the average seek time. In practice, large variations would be expected depending on the seek distance, but modeling such variations would inevitably lead to models specific to the hardware, file system or operating system, which we intentionally avoid here. Likewise, in contemporary systems, read and write times are greatly impacted by caches operating at several levels, which we do not model here. Thus, our goal is to find algorithms that minimize the *number* of seek and file access operations, which we denote "number of seeks" in the remainder.

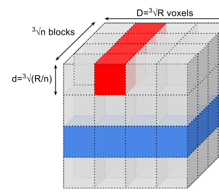


Figure 1: Notations. A *block column* is shown in red. A *block slab* is shown in blue.

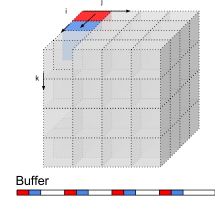


Figure 2: Buffer used in Clustered reads (d=4). White portions in the buffer are not allocated.

C. Slabs vs blocks

Algorithms 1 and 2 show the naive merging methods for slabs and blocks. These algorithms have very different complexities even though blocks and slabs have identical sizes. Since slabs are stored contiguously in the reconstructed image, the number of seeks in Algorithm 1 is only $2n$ as n seeks are required to read the slabs and n seeks are required to write them:

$$N_{\text{slabs}} = 2n \tag{1}$$

However, Algorithm 2 has to do extra seeks for each column in each slice of each block:

$$N_{\text{blocks}} = n + nd^2$$

or, using R and n as main variables:

$$N_{\text{blocks}} = n + n\left(\sqrt[3]{\frac{R}{n}}\right)^2 \tag{2}$$

In practice, this difference could lead to a tremendous slowdown, as we will show later.

Algorithm 1 Naive merging from slabs

```
for each slab do
    read slab
    write slab in reconstructed image
end for
```

Algorithm 2 Naive merging from blocks

```
for each block do
    read block
    write block in reconstructed image
end for
```

D. *Buffered slabs*

Algorithm 1 is a particular case of memory buffering where the amount of available memory equals the maximum size of a chunk. More buffering can be achieved when the amount of available memory increases, as shown in Algorithm 3.

Algorithm 3 Buffered merging from slabs

1: sorted_slabs = sort slabs by increasing k values
2: initialize buffer
3: **for** i = 0 ; i <n ; i+=1 **do**
4: slab = sorted_slabs[i]
5: **if** sizeof(buffer)+sizeof(slab) $\geq$ m **then**
6: write buffer in reconstructed image
7: clear buffer
8: **end if**
9: read slab and append it to buffer
10: **end for**

This algorithm writes in the reconstructed image using a single seek per memory load. Therefore:

$$N_{\text{buff_slabs}} = n + \left\lceil \frac{bR}{m} \right\rceil \quad (3)$$

Buffered slabs is straightforward to implement, however, its extension to block merging is not easy. The remainder of this Section presents our attempts for such an extension.

E. *Buffered blocks: Clustered reads*

Clustered reads is the more direct extension of Buffered slabs to blocks: it loads multiple blocks in memory, concatenates them in a buffer and writes the buffer in the reconstructed image. Seeking is reduced compared to Naive blocks since contiguous parts of the buffer are written without seeking. A given block is accessed only once during the whole merging process.

The buffer, capable of storing multiple disjoint sequences of contiguous bytes without having to allocate memory for the bytes between such sequences, can be represented by an associative array or a Python dictionary. Figure 2 illustrates how the buffer would fill up for the two first blocks in a reconstructed image, assuming, for the sake of this particular example, that blocks are of size 4x4x4.

The number of seeks performed by Clustered reads depends on how blocks loaded in memory arrange in the reconstructed image. In the best case, complete contiguous slabs of the reconstructed image can be assembled in memory and written in a single seek. In the worst case, the memory load only partially covers columns in the reconstructed image: O(d^2) seeks are then required during writing, one for every column of every tile. In the intermediary case, columns are complete but some slices can only be partially reconstructed: O(d) seeks are then required.

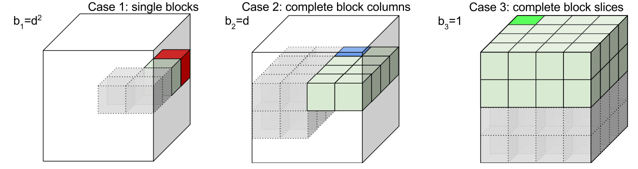


Figure 3: Memory-load configurations in Clustered reads, leading to different number of seeks. Red blocks need seeking before each of their columns (d^2 seeks). Blue blocks need seeking before each of their tiles (d seeks). Green blocks need only a single seek. Grey, dashed, transparent blocks represent the contiguous memory loads and are added for the sake of visualization.

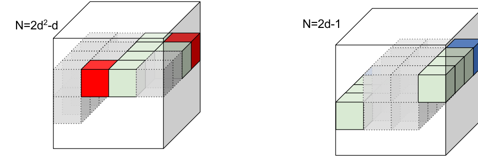


Figure 4: Configurations that increase the number of seeks per memory-load and are thus deliberately avoided by Clustered reads. Left: configuration with incomplete block columns (multiplies the number of seeks by d compared to complete block columns). Right: configuration with block columns that overlap multiple block slices (multiplies the number of seeks by 2 compared to non-overlapping configurations).

Clustered reads focuses on the three memory load configurations represented in Figure 3: the amount of memory m' used by the algorithm is rounded down to the closest number of complete blocks (Case 1), of complete block columns (Case 2) or of complete block slices (Case 3). This is in general reasonable since adding an incomplete columns to a set of complete ones multiplies the number of required seeks by d, as illustrated in Figure 4-Left. In some cases though, rounding m down to m' might increase the number of required memory loads to a point that the overall number of seeks also increases. Such cases are, however, slightly unusual and their complete description requires extensive calculations involving modulo arithmetic, which we felt were unwieldy to report here.

Our algorithm also avoids configurations where the memory load overlaps multiple block slices in Case 2 or multiple block columns in Case 1, as such overlaps multiply the number of required seeks (see Figure 4-Right).

Clustered reads is described in Algorithm 4. Function `switch` (line 3) selects one of the three cases based on the amount of available memory and the number of blocks. It returns `m'` and `case`, the identifier of the selected case. Function `check_overlap` (line 7) determines whether two blocks overlap multiple block slabs (Case 2) or multiple block columns (Case 1). For Case 3 it always returns false.

Function `sizeof` (line 8) returns the actual memory used by its argument, including only its allocated segments in the case that the argument is a buffer.

Algorithm 4 Buffered merging of blocks with Clustered reads

```
1:  sorted_blocks = sort blocks by increasing (k,j,i)
2:  initialize buffer
3:  (m',case)=switch(m,n,R,b)
4:  old_block = sorted_blocks[0]
5:  for i = 0 ; i<n ; i+=1 do
6:     block = sorted_blocks[i]
7:     overlap = check_overlap(block,old_block,case)
8:     if sizeof(buffer)+sizeof(block) ≥ m' or overlap=true
       then
9:        write buffer in reconstructed image
10:       clear buffer
11:       overlap = false
12:    end if
13:    read block and insert it in buffer
14: end for
```

The amount of memory used m' is set as follows in each of the 3 cases:

$$m'_1 = \frac{Rb}{n}\left\lfloor\frac{mn}{Rb}\right\rfloor; m'_2 = \frac{Rb}{\sqrt[3]{n}^2}\left\lfloor\frac{m\sqrt[3]{n}^2}{Rb}\right\rfloor; m'_3 = \frac{Rb}{\sqrt[3]{n}}\left\lfloor\frac{m\sqrt[3]{n}}{Rb}\right\rfloor$$

The number of seeks performed by Clustered reads in each of the three cases is:

$$N^i_{\text{CR}} = n + x_i b_i, \quad i \in [\![1,3]\!],$$

where x_i is the number of required memory loads and b_i is the number of seeks required to write a memory load. The first n seeks in the equation correspond to the reading of all the blocks. According to Figure 3, we have:

$$b_1 = d^2 = \sqrt[3]{\frac{R}{n}}^2 \quad ; \quad b_2 = d = \sqrt[3]{\frac{R}{n}} \quad ; \quad b_3 = 1$$

The numbers of memory loads required to reconstruct the image are:

$$x_1 = \left\lceil\frac{Rb}{\sqrt[3]{n}^2 m'_1}\right\rceil\sqrt[3]{n}^2; x_2 = \left\lceil\frac{Rb}{\sqrt[3]{n}m'_2}\right\rceil\sqrt[3]{n}; x_3 = \left\lceil\frac{Rb}{m'_3}\right\rceil$$

Because our algorithm avoids overlapping configurations, x_1 is proportional to the total number of block columns in the image $\left(\sqrt[3]{n}^2\right)$ and x_2 is proportional to the total number of block slices ($\sqrt[3]{n}$). Finally, the total number of seeks performed by Clustered reads to reconstruct the image is:

$$N_{\text{CR}} = \begin{cases} n + \left\lceil\frac{Rb}{\sqrt[3]{n}^2 m'_1}\right\rceil\sqrt[3]{R}^2 & \text{if } m < \frac{Rb}{\sqrt[3]{n}^2} \\ n + \left\lceil\frac{Rb}{\sqrt[3]{n}m'_2}\right\rceil\sqrt[3]{R} & \text{if } \frac{Rb}{\sqrt[3]{n}^2} \leq m < \frac{Rb}{\sqrt[3]{n}} \\ n + \left\lceil\frac{Rb}{m'_3}\right\rceil & \text{if } \frac{Rb}{\sqrt[3]{n}} \leq m < Rb \end{cases} \quad (4)$$

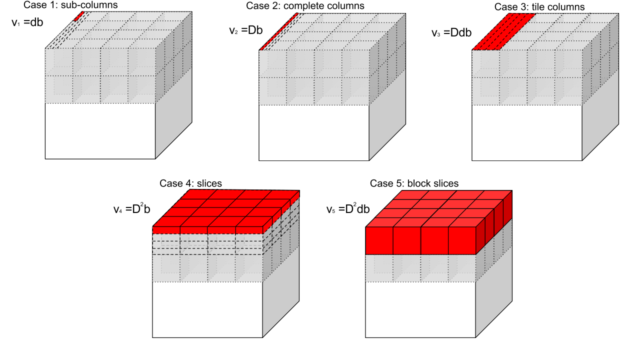


Figure 5: Memory-load configurations in multiple reads (d=4, D=16, n=64, $k_1 = k_2 = k_3 = k_4 = k_5 = 1$). Red color shows the content of the memory load. Small dots depict block frontiers and long dashes depict columns and slices within blocks.

It should be noted that N_{CR} is not a continuous function of m, due to the differences among b_i values.

F. Buffered blocks: Multiple reads

Multiple reads is shown in Algorithm 5. The main idea of this algorithm is that blocks are read partially (line 9) to ensure that the memory buffer only contains contiguous bytes. Therefore, the buffer can be written continuously to the reconstructed image, without seeking (line 13). However, a given block might be read multiple times, in different memory loads.

Algorithm 5 Buffered merging of blocks with multiple reads

```
1:  sorted_blocks = sort blocks by increasing (k,j,i)
2:  start_index = 0 ; end_index=(m-1)
3:  write_range = (start_index, end_index)
4:  while end_index <Rb do
5:     initialize buffer
6:     for block in sorted_blocks do
7:        if block has voxels in write_range then
8:           block_data = read block
9:           in block_data, extract the columns in write_range

10:          insert columns in buffer
11:       end if
12:    end for
13:    write buffer to reconstructed_image
14:    start_index = end_index + 1 ; end_index += m
15: end while
```

In the complexity analysis, we assume that m' represents an integer number k of sub-columns (Case 1, $k_1 < \sqrt[3]{n}$), of complete columns (Case 2, $k_2 < d$), of tile columns (Case 3, $k_3 < \sqrt[3]{n}$), of slices (Case 4, $k_4 < d$) or of block slices (Case 5, $k_5 < \sqrt[3]{n}$), as illustrated in Figure 5. In each of

these 5 cases, we define v_i as follows:

$$v_1 = db \; ; \; v_2 = Db \; ; \; v_3 = Ddb \; ; \; v_4 = D^2 b \; ; \; v_5 = D^2 db$$

so that we have:

$$k_i = \left\lfloor \frac{m}{v_i} \right\rfloor \quad \text{and} \quad m'_i = k_i v_i, \quad i \in [\![1, 5]\!]$$

The total number of seeks performed by multiple reads in Case i is:

$$\begin{aligned} N^i_{\mathrm{MR}} &= x_i + (x_i - 1) b_i + b'_i, \quad i \in [\![1, 5]\!] \\ &= x_i (1 + b_i) - b_i + b'_i \end{aligned}$$

where x_i is the total number of memory loads, b_i is the number of blocks accessed by the first $(x_i - 1)$ memory loads and b'_i is the number of blocks access by the last memory load. The first x_i seeks in the equation correspond to the writing of all memory loads (1 seek per memory load). We have:

$$x_i = \left\lceil \frac{Rb}{m'_i} \right\rceil, \quad i \in [\![1, 5]\!]$$

and:

$$b_1 = k_1; b_2 = \sqrt[3]{n}; b_3 = k_3 \sqrt[3]{n}; b_4 = \sqrt[3]{n}^2; b_5 = k_5 \sqrt[3]{n}^2$$

and:

$$\begin{aligned} b'_1 = \sqrt[3]{n} D^2 \bmod k_1; \quad b'_2 &= b_2; \quad b'_3 = \sqrt[3]{n} \left(\sqrt[3]{n} D \bmod k_3 \right) \\ b'_4 &= b_4; \quad b'_5 = \sqrt[3]{n}^2 \left(\sqrt[3]{n} \bmod k_5 \right) \end{aligned}$$

It gives the following expression for N_{MR}:

$$N_{\mathrm{MR}} = \begin{cases} \left\lceil \frac{Rb}{m'_1} \right\rceil (k_1 + 1) - k_1 & \\ + \left(\sqrt[3]{n} D^2 \bmod k_1 \right) & \text{if } d \leq \frac{m}{b} < D \\ \left\lceil \frac{Rb}{m'_2} \right\rceil (\sqrt[3]{n} + 1) & \text{if } D \leq \frac{m}{b} < Dd \\ \left\lceil \frac{Rb}{m'_3} \right\rceil (k_3 \sqrt[3]{n} + 1) - k_3 \sqrt[3]{n} + & \\ \qquad \sqrt[3]{n} \left(\sqrt[3]{n} D \bmod k_3 \right) & \text{if } Dd \leq \frac{m}{b} < D^2 \\ \left\lceil \frac{Rb}{m'_4} \right\rceil (\sqrt[3]{n}^2 + 1) & \text{if } D^2 \leq \frac{m}{b} < D^2 d \\ \left\lceil \frac{Rb}{m'_5} \right\rceil (k_5 \sqrt[3]{n}^2 + 1) - k_5 \sqrt[3]{n}^2 + & \\ \quad \sqrt[3]{n}^2 \left(\sqrt[3]{n} \bmod k_5 \right) & \text{if } D^2 d \leq \frac{m}{b} < R \end{cases}$$

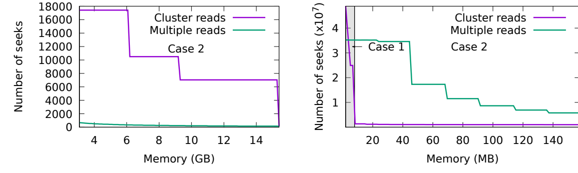


Figure 6: Number of seeks for Clustered reads vs Multiple reads, for D=3458 and b=2. Left: n=125; Right: n=64,000. Case 1 and Case 2 denote Clustered read configurations.

And finally, using R and n as main variables:

$$N_{\mathrm{MR}} = \begin{cases} \left\lceil \frac{Rb}{m'_1} \right\rceil \left(\frac{m'_1 \sqrt[3]{n}}{\sqrt[3]{Rb}} + 1 \right) - \frac{m'_1 \sqrt[3]{n}}{\sqrt[3]{Rb}} & \\ + \left(\sqrt[3]{n} \sqrt[3]{R}^2 \bmod \left\lfloor \frac{m \sqrt[3]{n}}{\sqrt[3]{Rb}} \right\rfloor \right) & \text{if } \sqrt[3]{\frac{R}{n}} \leq \frac{m}{b} < \sqrt[3]{R} \\ \left\lceil \frac{Rb}{m'_2} \right\rceil (\sqrt[3]{n} + 1) & \text{if } \sqrt[3]{R} \leq \frac{m}{b} < \frac{\sqrt[3]{R}^2}{\sqrt[3]{n}} \\ \left\lceil \frac{Rb}{m'_3} \right\rceil \left(\frac{m'_3 \sqrt[3]{n}^2}{\sqrt[3]{R}^2 b} + 1 \right) - \frac{m'_3 \sqrt[3]{n}^2}{\sqrt[3]{R}^2 b} & \\ + \sqrt[3]{n} \left(\sqrt[3]{nR} \bmod \left\lfloor \frac{m \sqrt[3]{n}}{\sqrt[3]{R}^2 b} \right\rfloor \right) & \text{if } \frac{\sqrt[3]{R}^2}{\sqrt[3]{n}} \leq \frac{m}{b} < \sqrt[3]{R}^2 \\ \left\lceil \frac{Rb}{m'_4} \right\rceil \left(\sqrt[3]{n}^2 + 1 \right) & \text{if } \sqrt[3]{R}^2 \leq \frac{m}{b} < \frac{R}{\sqrt[3]{n}} \\ \left\lceil \frac{Rb}{m'_5} \right\rceil \left(\frac{m'_5 n}{Rb} + 1 \right) - \frac{m'_5 n}{Rb} & \\ + \sqrt[3]{n}^2 \left(\sqrt[3]{n} \bmod \left\lfloor \frac{m \sqrt[3]{n}}{Rb} \right\rfloor \right) & \text{if } \frac{R}{\sqrt[3]{n}} \leq \frac{m}{b} < R \end{cases} \tag{5}$$

G. Analysis

Figure 6 plots Equations 4 and 5 for different values of n. When Clustered reads is in Case 1 or 2, it may outperform Multiple reads only for large values of n. When Clustered reads is in Case 3, it is equivalent to Multiple reads: assuming that Rb is an exact multiple of m, Equations 4 and 5 both boil down to $n + \frac{Rb}{m}$.

III. Implementation

We implemented the 5 algorithms presented earlier in a Python library called `sam` (for "split and merge"). It uses Nibabel [13] for image I/O and NumPy for array manipulations.

The data buffer used in Clustered reads and Multiple reads is implemented as a Python dictionary where the keys are offsets in the reconstructed image and the values are NumPy arrays containing the data starting at this offset. When the memory load is complete, dictionary entries are written sequentially to the reconstructed image. In Clustered reads, some seeking might be required between writes. In Multiple reads, dictionary entries are always contiguous in the reconstructed image. We tried to use a single NumPy array as a buffer, but we finally abandoned it as inserting data at a specific position in a NumPy array copies the data in memory, which increases both the execution time and the peak memory consumption. We also implemented a defragmentation procedure for the dictionary that merges

contiguous dictionary entries in a single one, but we abandoned it as it proved more time-consuming than going through all the initial entries, due to the overhead of resizing NumPy arrays to merge entries.

The implementation of splitting algorithms was greatly facilitated by the availability of so-called array proxies in Nibabel, which help reading specific sub-parts of large images. Nibabel's array proxies essentially provided the buffer implementation for splitting algorithms. Unfortunately, they are not available to write data.

In Multiple reads, block headers are read in a first pass where column indices in the reconstructed image are stored in memory. Those indices are then processed in each memory load, to identify the blocks that contribute to it.

We also implemented the splitting algorithms corresponding to Clustered reads and Multiple reads, called Clustered writes and Multiple writes.

A. Lossless compression

We implemented lossless compression for all algorithms using Python's gzip library. Compression is done on the fly, that is, while the data is being read or written during splitting and merging. On-the-fly compression of large datasets is a challenge when extensive seeking is involved as the gzip library has to decompress all the data until the seek position to read from it, making access time a linear function of the seek position. This is potentially an issue for Clustered writes, which could be addressed by indexing techniques such as described in [14] for the NiFTI format and implemented in https://github.com/pauldmccarthy/indexed_gzip. Multiple writes is not impacted since it does not seek in the large image.

In addition, "negative" seeking (seeking to a position that precedes the current one) is not possible while writing a compressed file, which renders Naive blocks unusable with compressed data and Clustered reads usable only in Case 3. Again, Multiple reads is not impacted since it does not seek in the large image.

IV. Experiments

A. Data

We used the 3850x3025x3500 Big Brain image split into 125 non-overlapping chunks of size 770x605x700, with 2 bytes per voxel (total size uncompressed is 75.92 GB). The Big Brain image was also split into 125 non-overlapping slabs of size 3850x3025x28 for our experiments.

Big Brain is a reference brain based on the reconstruction of 7404 histological sections at nearly cellular resolution of 20 micrometers [1]. It is a freely, publicly available tool with numerous applications in neurosciences and neurosurgery.

We used the blocks of the 2015 Big Brain release with 40-micrometer isotropic resolution available at ftp://bigbrain.loris.ca/BigBrainRelease.2015/3D_Blocks/40um. We converted them to NiFTI 1.0 using Nibabel and left them uncompressed. These blocks were then used to reconstruct the Big Brain that was split into our desired block and slab configurations using our naive splitting algorithms.

	3 GB	6 GB	9 GB	12 GB	16 GB
Clustered reads	1	2	2	2	3
Multiple reads	4	4	4	4	5

Table I: Algorithm configurations by memory values.

B. Hardware

We used a Dell Precision Tower 3620 workstation with CentOS Linux release 7.3.1611, 32 GB of RAM and two disks: (1) a Hard Disk Drive (HDD): HGST Travelstar 7K1000, 7200 rpm, 931GiB (1TB), firmware version JB0OA3W0; (2) a Solid-state drive (SSD): SanDisk X400 2.5, 238GiB (256GB), firmware version X4130012. Both drives used 512-byte logical sectors, 4096-byte physical sectors, SATA >3.1 (6.0 Gb/s) and were accessed through the XFS file system v4.5.0. We used `iotop` (http://guichaz.free.fr/iotop) to monitor I/Os on the workstation and make sure that no other process was compromising our measures.

C. Execution conditions

We used Git tag 0.1.1 of our `sam` library. Our experiment scripts are available at https://github.com/big-data-lab-team/paper-sequential-split-merge/blob/master/scripts/experiment. We used them to split and merge using Buffered slabs, Clustered reads and Multiple reads, with 3 GB, 6 GB, 9 GB, 12 GB and 16 GB of memory. Table I shows the configurations of Clustered reads and Multiple reads for each memory value, according to Equations 4 and 5. For instance, at 3 GB, Clustered reads were in Case 1. We also did a run with 0 GB of memory for Buffered slabs and Clustered reads, which triggered Naive slabs and Naive blocks. We did 5 repetitions for each memory value. Memory values were shuffled in each repetition, to avoid potential ordering biases such as caching effects. To ensure equal conditions, we dropped the kernel page, dentry and inode caches before each run (`echo 3 | sudo tee /proc/sys/vm/drop_caches`). We measured the cumulative read, write and seek time in each run, as well as the overhead time defined as the total time minus the sum of all other times.

Compressed blocks and slabs were also merged into a compressed image. On-the-fly gzip compression was used for Naive slabs, Buffered slabs and Multiple reads. As explained before, Clustered reads could only be applied to compressed data while in Case 3, i.e., for 16 GB, and Naive blocks could not be used at all with on-the-fly compression. To give a reference, we used Naive blocks with *offline* compression, that is, we wrote an uncompressed image and compressed it sequentially afterwards.

D. Results

All the experimental data and scripts used to generate the figures in this Section are available at https://github.com/big-data-lab-team/paper-sequential-split-merge under GPLv3 license.

1) Seeks: The number of seeks is reported in Figure 7, for all algorithms and the corresponding models (Equations 1 to 5). Note the logarithmic y scale. Error bars are not reported as numbers of seeks were constant across all repetitions. The average relative model errors are 12.7% (Naive blocks), 0% (Naive slabs), 3.3% (Clustered reads), 26.8% (Multiple reads) and 0.9% (Buffered slabs), explained by the fact that the model assumes cuboid blocks while we used non-cuboid ones in the experiment. For Multiple reads, our complexity analysis also assumed that one of the 5 Cases in Figure 5 was used while they are blended in practice. Overall, the model correctly explains the observations.

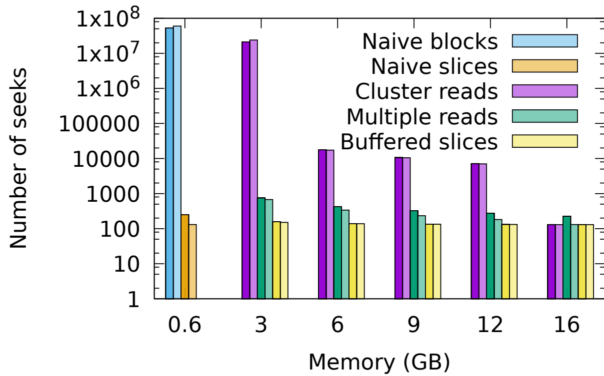


Figure 7: Number of seeks for all algorithms. y scale is logarithmic. Each algorithm is represented with a different color. Dark color is experimental value; bright color is model.

As expected, the difference between Naive blocks and Naive slabs is tremendous, in the order of 50 million seeks. The difference between Clustered reads and Multiple reads is consistent with our analysis. For 3 GB, the number of seeks in Clustered reads is 4 orders of magnitude higher than for Multiple reads, and 5 orders of magnitude higher than Naive slabs. This huge difference comes from the fact that at 3 GB Clustered reads are in Case 1. For 6 GB, 9 GB and 12 GB, Clustered reads are in Case 2 and the difference with Multiple reads and Buffered slabs reduces. At 16 GB, all algorithms perform the same.

2) Merge time: Figure 8 shows the merge time for all algorithms by memory values. Naive blocks are 9.5 times slower than Naive slabs on HDD (6.7 times on SSD), which quantifies the effect of the problem targeted by our algorithms. In the remainder we use Naive blocks and Naive slabs as references to evaluate our algorithms.

Buffered slabs provides a negligible speed-up compared to Naive slabs. In most cases, their memory overhead would not be worth the time gain.

Clustered reads provides substantial speed ups compared to Naive blocks, both on HDD and on SSD. It is 6.8 times faster than Naive blocks on HDD and 5.1 times on SSD (average accross all repetitions, all memory values). Surprisingly, it performs substantially faster than Naive blocks even at 3 GB, while in Case 1. This may be explained by the fact that the seeks required to write incomplete block columns to the reconstructed image are shorter than the ones for Naive blocks.

Multiple reads is even faster than Clustered reads on this dataset. They are 8.4 times faster than Naive blocks on HDD and 5.3 times on SSD (average accross all repetitions, all memory values).

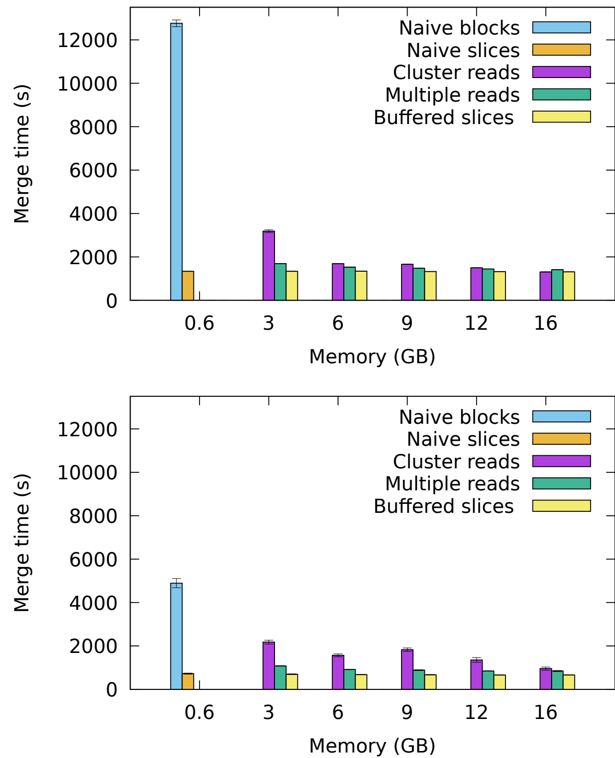
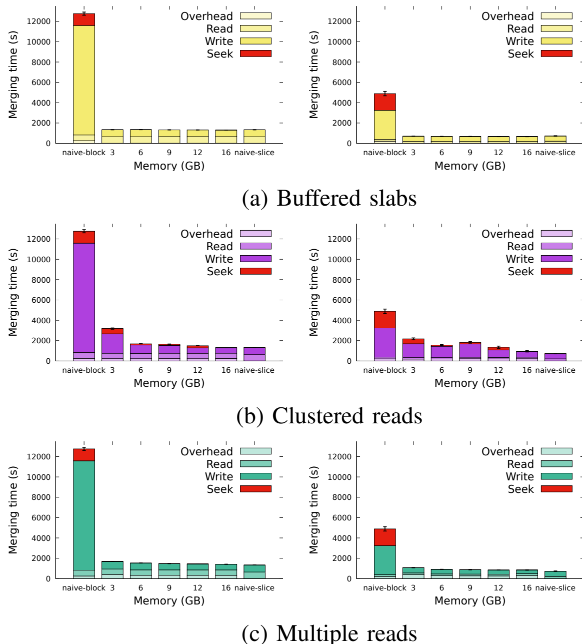


Figure 8: Merge time by algorithm. Top: HDD. Bottom: SSD. Each algorithm is represented with a different color. Averages over 5 repetitions. Error bars show $\pm$ 1 standard deviation.

3) Merge time breakdown: Figure 9 shows how the total merge time breaks down to read, write, seek and overhead time for our algorithms. Naive blocks and Naive slabs are shown as references. The huge difference between Naive blocks and Naive slabs is coming from both the seek time and the write time, which suggests that seeking degrades the



(a) Buffered slabs

(b) Clustered reads

(c) Multiple reads

Figure 9: Breakdown of total merge times. Left column: HDD. Right column: SSD.

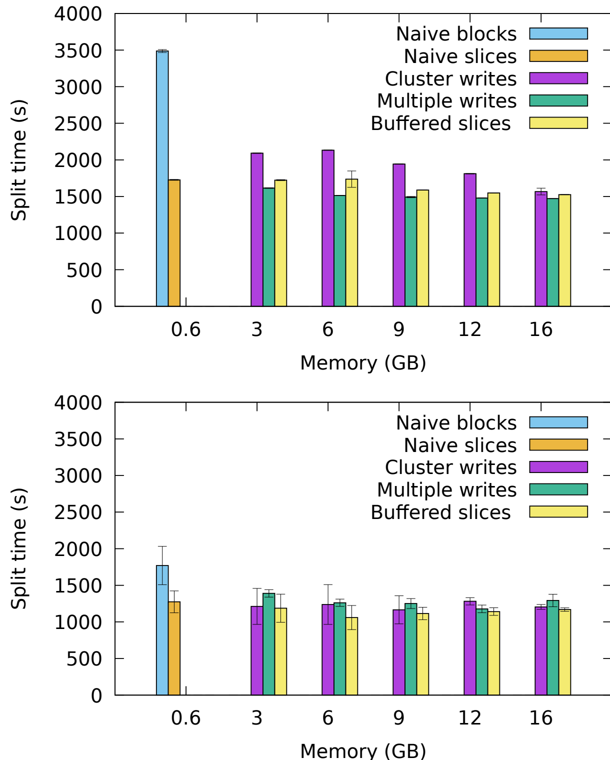


Figure 10: Split time by algorithm. Top: HDD. Bottom: SSD. Each algorithm is represented with a different color. Averages over 5 repetitions. Error bars show ± 1 standard deviation.

write rate in addition to introducing extra delays. Clustered reads reduces the seek time and thus the read time very substantially. Multiple reads almost annihilates the seek time and brings the read time to a value comparable to Naive slabs. The same behavior is observed on HDD and on SSD, although the effect of seeking is slightly smaller on SSD, as expected. Read times are consistently and substantially lower than write times. This may be a result of discrepancies between disk read and writes rates, or of reading data using Python's NumPy package, which is more efficient than using native Python as is the case with our writes. The overhead time is small for both Clustered reads and Multiple reads.

4) Split time: The total split time by algorithm is shown in Figure 10. The difference between Naive blocks and Naive slabs is still significant (average ratio is 1.4 on SSD, 2.0 on HDD) although less than for merging. On SSD, Clustered writes and Multiple writes both perform similarly to Naive slabs. On HDD, Clustered writes is slightly slower than Multiple writes until 16 GB. Buffered slabs provides no speed-up compared to Naive slabs.

The breakdown by read, write and overhead time is shown in Figure 11. We were not able to measure seek time for splitting algorithms as it was blended in read time by the Nibabel library. Naive blocks are strongly penalized by important read times coming from extensive seeking. Buffered slabs, Clustered writes and Multiple writes all reduce the read time compared to Naive blocks, but they also increase the write time, most likely due to caching effects in Naive blocks and slabs.

E. Compression

Figure 12 shows the impact of gzip compression on the merging time, for Multiple reads and Naive slabs. Unsurprisingly, compression dramatically slows down merging time for all algorithms. Multiple reads, however, still provides speed-up compared to Naive blocks, although it does not completely reach the performance of Buffered slabs and Naive slabs.

V. Discussion

Clustered reads and Multiple reads reduce to a negligible amount the overall seek time required to split or merge 3D blocks in a high-resolution image where data is stored linearly. Both algorithms performed equivalently and compared to the reference configuration where slabs are merged or split without seeking. Our initial problem is solved.

Our results demonstrate that large images stored in simple formats may be split and merged without performance loss compared to more complex formats, for instance MINC 2.0 or the format based on space-filling curves mentioned in [12]. This is of major interest in the current open-science context since simpler formats favor data-sharing and

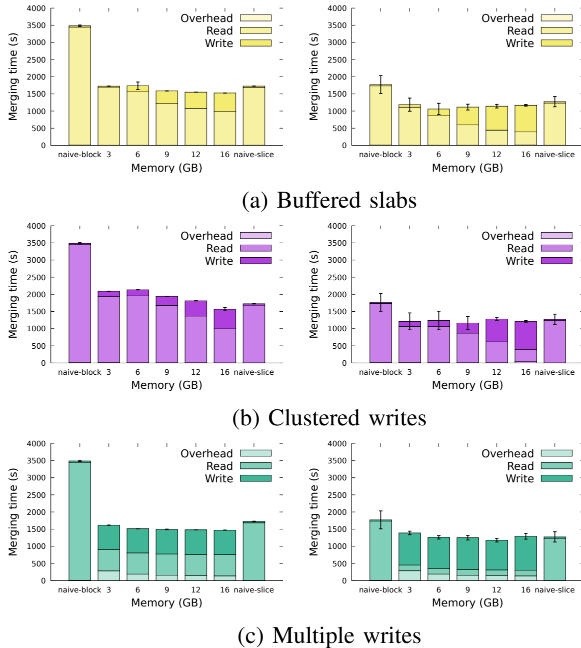


Figure 11: Breakdown of total split times. Left column: HDD. Right column: SSD.

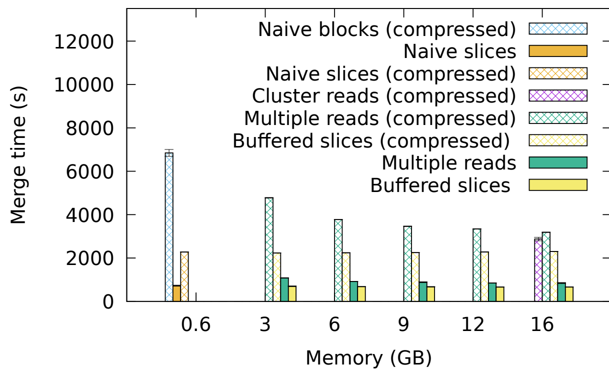


Figure 12: Merging time by algorithm, using compression (on SSD). Each algorithm is represented with a different color. Hatching represents compression. Averages over 5 repetitions. Error bars show ± 1 standard deviation.

interoperability. Moreover, our algorithms could potentially be adapted to any chunk geometry, even though we demonstrated them on slabs and blocks only, while file formats inevitably assume a particular geometry. For instance, the format in [12] is not designed to naively split slabs. However, we aimed to extract all the blocks, whereas [12] aimed to extract a single block from a large image. MINC 2.0 might also help with on-the-fly compression for Clustered reads.

On-the-fly compression, indeed, could not be used with Clustered reads due to its use of negative seeking. This was not a problem for Multiple reads since Multiple reads completely removes seeking in the large image. Likewise, Multiple writes would not benefit from techniques aiming at accelerating random access reads in compressed images since they do not need to seek in it.

I/O optimization is a holistic problem that is in practice highly dependent on the hardware used, firmware, operating system, file system and programming language. Caching occurs at various levels and might always influence performance, potentially differently depending on the split or merge algorithm used. In some disks, seek time greatly varies with the seek distance, which would open the door to additional opportunities for I/O optimization. Interactions between those components might also result in performance differences particular to a specific algorithm. To ensure the portability of our library across systems and configurations, we focused on reducing the overall number of seeks and ignored specific system configurations. We demonstrated the performance of our methods on both an HDD and an SSD disk, using state-of-the-art and widely used versions of Linux (CentOS7) and file systems (XFS v4.5.0).

High-resolution images are likely to be processed on computing clusters, for instance using software from the Hadoop project, in particular the Hadoop Distributed File System [4]. In this context parallel split-and-merge algorithms would be beneficial, since the various blocks of a large image could be uploaded to different disks concurrently. In the same vein, "re-spliting" algorithms would be beneficial in case an image already split needs to be split in a different geometry. Designing such algorithms is part of our future work, in which Clustered reads and Multiple reads will be used as starting points.

Our `sam` library is available at https://github.com/big-data-lab-team/sam under MIT license.

Acknowledgment

We warmly thank Lindsay B. Lewis and Claude Lepage for helping us with Big Brain, Greg Kiar for useful discussion about the Open Connectome Data Cluster and Pierre Bellec for discussions about MINC.

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 *et al.*, "Bigbrain: an ultrahigh-resolution 3d human brain model," *Science*, vol. 340, no. 6139, pp. 1472–1475, 2013.

[2] D. D. Bock, W.-C. A. Lee, A. M. Kerlin, M. L. Andermann, G. Hood, A. W. Wetzel, S. Yurgenson, E. R. Soucy, H. S. Kim, and R. C. Reid, "Network anatomy and in vivo physiology of visual cortical neurons," *Nature*, vol. 471, no. 7337, p. 177, 2011.

[3] M. Langer, A. Pacureanu, H. Suhonen, Q. Grimal, P. Cloetens, and F. Peyrin, "X-ray phase nanotomography resolves the 3d human bone ultrastructure," *PLOS ONE*, vol. 7, no. 8, pp. 1–7, 08 2012. [Online]. Available: https://doi.org/10.1371/journal.pone.0035691

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 2010, pp. 1–10.

[5] R. D. Vincent, P. Neelin, N. Khalili-Mahani, A. L. Janke, V. S. Fonov, S. M. Robbins, L. Baghdadi, J. Lerch, J. G. Sled, R. Adalat *et al.*, "Minc 2.0: A flexible format for multimodal images," *Frontiers in neuroinformatics*, vol. 10, 2016.

[6] S. Miguet and Y. Robert, "Elastic load-balancing for image processing algorithms," in *International Conference of the Austrian Center for Parallel Computation*. Springer, 1991, pp. 438–451.

[7] G. Tang, L. Peng, P. R. Baldwin, D. S. Mann, W. Jiang, I. Rees, and S. J. Ludtke, "Eman2: an extensible image processing suite for electron microscopy," *Journal of structural biology*, vol. 157, no. 1, pp. 38–46, 2007.

[8] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on cuda," in *Computer Science and Software Engineering, 2008 International Conference on*, vol. 3. IEEE, 2008, pp. 198–201.

[9] T. Bräunl, S. Feyrer, W. Rapf, and M. Reinhardt, *Parallel image processing*. Springer Science & Business Media, 2013.

[10] D. Moise, D. Shestakov, G. Gudmundsson, and L. Amsaleg, "Terabyte-scale image similarity search: experience and best practice," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 674–682.

[11] P. Bajcsy, A. Vandecreme, J. Amelot, P. Nguyen, J. Chalfoun, and M. Brady, "Terabyte-sized image computations on hadoop cluster platforms," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 729–737.

[12] R. Burns, K. Lillaney, D. R. Berger, L. Grosenick, K. Deisseroth, R. C. Reid, W. G. Roncal, P. Manavalan, D. D. Bock, N. Kasthuri *et al.*, "The open connectome project data cluster: scalable analysis and vision for high-throughput neuroscience," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013, p. 27.

[13] M. Brett, M. Hanke, B. Cipollini, M.-A. Ct, C. Markiewicz, S. Gerhard, E. Larson, G. R. Lee, Y. Halchenko, E. Kastman, cindeem, F. C. Morency, moloney, J. Millman, A. Rokem, jaeilepp, A. Gramfort, J. J. van den Bosch, K. Subramaniam, N. Nichols, embaker, bpinsard, chaselgrove, N. N. Oosterhof, S. St-Jean, B. Amirbekian, I. Nimmo-Smith, S. Ghosh, G. Varoquaux, and E. Garyfallidis, "nibabel: 2.1.0," Aug. 2016. [Online]. Available: https://doi.org/10.5281/zenodo.60808

[14] Z. Rajna, A. Keskinarkaus, V. Kiviniemi, and T. Seppänen, "Speeding up the file access of large compressed nifti neuroimaging data," in *Engineering in Medicine and Biology Society (EMBC), 2015 37th Annual International Conference of the IEEE*. IEEE, 2015, pp. 654–657.