A NEW ALGORITHM TO SPLIT AND MERGE ULTRA-HIGH RESOLUTION 3D IMAGES

Yongping Gao

A thesis
in
The Department
of
Computer Science

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

December 2017

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Yongping Gao**

Entitled: **A new algorithm to split and merge ultra-high resolution 3D images**

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

Master of Computer Science

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

Signed by the final examining commitee:

_____ Chair

Dr. Emad Shihab

_____ Examiner

Dr. Todd Eavis

_____ Examiner

Dr. Joey Paquet

_____ Supervisor

Dr. Yuhong Yan

_____ Co-supervisor

Dr. Tristan Glatard

Approved _____
Chair of Department or Graduate Program Director

_____ 20 _____ _____

Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

A new algorithm to split and merge ultra-high resolution 3D images

Yongping Gao

Splitting and merging ultra-high resolution 3D images is a requirement for parallel or distributed processing operations. Naive algorithms to split and merge 3D blocks from ultra-high resolution images perform very poorly, due to the number of seeks required to reconstruct spatially-adjacent blocks from linear data organizations on disk. The current solution to deal with this problem is to use file formats that preserve spatial proximity on disk, but this comes with additional complexity. We introduce a new algorithm called Multiple reads/writes to split and merge ultra-high resolution 3D images efficiently from simple file formats. Multiple reads/writes only access contiguous bytes in the reconstructed image, which leads to substantial performance improvements compared to existing algorithms. We parallelize our algorithm using multi-threading, which further improves the performance for data stored on a Hadoop cluster. We also show that on-the-fly lossless compression with the lz4 algorithm reduces the split and merge time further.

Acknowledgments

Firstly, I would like to thank my co-supervisor Dr. Tristan Glatard, for his continuous support of my research, for his encouragement, patience, and immense knowledge. I have learnt a lot from him. I would also like to thank Dr. Yuhong Yan for her endless support of my work.

Finally I will thank my colleague Valérie for her tremendous help as well as my friends, my family, who have always been there for me all the time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ultra-high-resolution images that can 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 [3].

The processing of such large images requires extensive amounts of disk, memory, and computing power. Two architectures are commonly identified to aggregate resources: scaling up and scaling out. Scaling up, also called vertical scaling, consists in adding more resources to a single node in a system. Scaling out, also called horizontal scaling, consists in adding more compute nodes to the current system to form a cluster. The nodes in a cluster act like a single system to enable high performance and high availability through load balancing and parallel processing. Scaling out is the most commonly used approach in current Big Data systems, as it results in cheaper, more extensible systems. This thesis focuses on the processing of large 3D images on such clusters.

Big Data clusters currently use specific distributed file systems such as the Hadoop Distributed File system to store the data on multiple nodes. Distributed file systems provide two essential properties: (1) data availability: data is replicated to multiple storage nodes to make the system robust to the failure of a few nodes, (2) data locality: storage nodes are also compute nodes so that computations can be triggered without the need for data transfers in the cluster.

Distributed file systems obviously require the data to be split in multiple chunks,

and eventually merged back to be delivered to the user. This thesis focuses on the specific problem of splitting and merging 3D images efficiently, to enable their efficient processing on Big Data clusters. As detailed later in the thesis, the main issue encountered when splitting or merging large images is that random I/Os need to be done in the large image, to reconstruct spatially-adjacent 3D blocks from non-contiguous bytes stored in a file. This results in important file access and seek times, which can drastically slow down image processing operations. We are looking for algorithms that reduce the number of file accesses required to split and merge a high-resolution 3D image.

We aim at algorithms that can work with arbitrary chunk geometries. Image processing pipelines are extremely diverse and have different requirements on the size and geometry of the chunks they can process. Depending on the pipeline, a large image might be split into slabs or blocks, of different sizes. Our split and merge algorithms should be versatile enough to support arbitrary geometries efficiently.

1.1 Goals and Contributions

The goal of this thesis is:

- To propose a new algorithm to split and merge ultra-high resolution 3D images.
- To compare with the current algorithms.

The contribution of this thesis is as follows:

- We introduce a new algorithm to split and merge ultra-high resolution 3D image efficiently (Chapter 3).
- We parallelize such algorithm (Chapter 4).
- We apply on-the-fly compression to further improve the performance (Chapter 5).
- We implement such algorithms, and build a Python library based on different algorithm parameters.

We also benefit open-science by making the code publicly available [1], and make a new release when done with experiments.

This thesis will be broken into four chapters, in Chapter 2, we will examine the current solution to split and merge ultra-high resolution 3D images of different neuroimaging file formats, and introduce the related tools and libraries we used. Chapter 3 introduces a new sequential algorithm to split and merge ultra-high resolution 3D images. Chapter 4 introduces the parallel version of this new algorithm. In Chapter 5, we apply on-the-fly compression to our new algorithm.

[1] `sam`: https://github.com/big-data-lab-team/sam/releases

Chapter 2

Big Data infrastructure for the processing of ultra-high resolution 3D images

2.1 Introduction

To process ultra-high resolution 3D images by using a parallelized pipeline, it is necessary to split and merge the image efficiently. In this chapter we first investigate different neuroimaging formats and associated libraries that support reading and writing of such file types. Then we address the problem of splitting and merging large images, and investigate the causes that slow down the split and merge process. We review the current solutions to this problem: (1) file formats based on space-filling curves, (2) the clustered reads/writes algorithm. Finally we will review the current big data infrastructure as well as parallel image processing methodology, compare different compression formats and their associated libraries.

2.2 Image formats and library

In this section we introduce several popular image formats and the associated libraries that are going to be used in the thesis.

2.2.1 Neuroimaging Informatics Technology Initiative

Neuroimaging Informatics Technology Initiative (NIfTI), is adapted as an extension of the popular ANALYZE^{TM} 7.5 file format (`.hdr`/`.img` file pairs) [4], which are desirable to functional magnetic resonance imaging (f-MRI) analysis. NIfTI-1 uses the "empty space" in the ANALYZE^{TM} 7.5 header to add several new features [5]. The NIfTI file format remains compatible with non-NIfTI aware ANALYZE^{TM} 7.5 software. NIfTI-1 image can have dual file (`.hdr` & `.img`) or single file (`.nii`) storage. The `.hdr` or header component is described as a single struct, a maximum of 348 bytes in size. The struct can describe this images's metadata like dimension, data type, number of bits per voxel, etc. After the end of the header, the next 4 bytes are a char array field named "extension". In a `.nii` file, these 4 bytes will always be present, and should be set to zero as default. NIfTI-1 image stores the image binary data linearly and in "column-major" order on disk, that is, for an image of dimensions (i, j, k), i is the fastest changing dimension, followed by dimension j, followed by dimension k [1]. All the implementation of the algorithm in this thesis will mainly focus on NIfTI-1 format image.

2.2.2 Medical Imaging NetCDF 2.0

An alternative to NIfTI format, is the HDF5-based Medical Imaging NetCDF (MINC) 2.0 format [6], which is designed for flexible and efficient I/O and for high volume and complex data and supports high-dimensionality and irregularly-shaped dimensions. MINC 2.0 also supports 64-bit data. MINC 2.0 provide more flexibility by allowing data to be partitioned in limited-sized chunks, each chunk being stored in a specific order. Also similar to HDF5, MINC 2.0 file format is hierarchical, consisting of groups and subgroups, similar to a file system. The metadata is stored to the following subgroups: `info` and `dimensions`, and image data is stored within the `image` subgroup. Moreover, the added feature of internal compression is also provided in MINC 2.0, which means we can decompress/compress chunks of an image at a time. NIfTI, on the other hand does not have this built-in feature.

The byte organization in MINC is stored in "row-major" order, that is, for an image of dimensions (i, j, k), k is the fastest changing dimension, followed by dimension j, followed by dimension i [6].

2.2.3 NiBabel

NiBabel is a popular Python library used for the reading and writing of various neurological imaging file formats, including NIfTI-1, NIfTI-2 and MINC 2.0. NiBabel offers both high-level format-independent access to neuroimages, as well as an API with various levels of format-specific access to all available information in a particular file format [7]. A NiBabel image object has three things:

- an N-D array containing the image data;
- a (4, 4) affine matrix mapping array coordinates to coordinates in some RAS+ world coordinate space (Coordinate systems and affines);
- image metadata in the form of a header.

To load image file as an NiBabel image object, get header of the image, and get image data is extremely simple:

```
import nibabel as nib
# img is an nibabel image object
img = nib.load(nifti_file)
# get header of the image
header = img.header
# get data of the image
data = img.get_data()
```

When the image file is really large, just in our case, we cannot use "img.get_data()" to load all the data into memory directly. It will throw an "out of memory" exception. NiBabel gives a solution for this by using array proxy. An array proxy is not the array itself but something that represents the array, and can provide the array when we ask for it. The array proxy allows for the reading of a subset of data into memory without having to read the entire image:

```
import nibabel as nib
# img is an nibabel image object
img = nib.load(nifti_file)
# get array proxy of the image
dataobj = img.dataobj
# get partial data from the image
```

```
data = dataobj[:100]
```

2.3 The problem of splitting and merging large images

In this section, we investigate the naive way of splitting and merging ultra-high resolution 3D images, and explain what is the overhead of the naive methods. This problem is described in [2].

2.3.1 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, 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.

2.3.2 Naive slabs and Naive blocks

We assume that the high-resolution image is split into chunks representing 3D blocks or 3D slabs that fit in memory. A 3D block (as shown in Figure 2) consists of a stack of one or more 2D tiles (incomplete slices or incomplete columns) spanning contiguous

slices, whereas a 3D slab (as shown in Figure 2) 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 0.6 GB. The decision to split an image into slabs or blocks, and the size of the chunks, is up to the application. We can also merge 3D blocks or 3D slabs to a reconstructed image.

To explain the problem, we will take merging as an example. For naive merging 3D slabs or 3D blocks, as shown in Algorithm 1, we load each slab/block at once, and write to the reconstructed file at the corresponding position.

Algorithm 1 Naive merging from 3D blocks/slabs

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

The results shows in Figure 1, we can see naive merging methods for 3D slabs and 3D blocks actually can have very different time even though blocks and slabs have identical sizes (0.6 G).

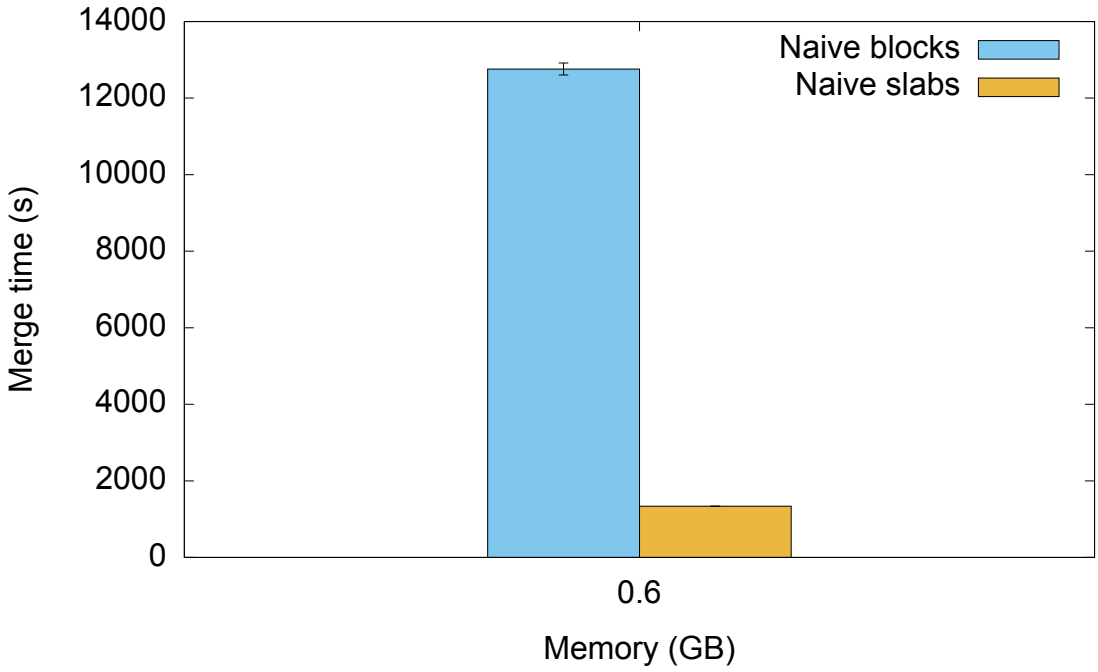

Figure 1: Merge time comparison between Naive slabs and Naive blocks in NIfTI format(data extracted from [2]).

Comparing to naive 3D slabs, naive 3D blocks has to do extra seeks for each row in each slice of each block due to the spatial organization of the bytes on disk (Figure 2). In practice, this difference could lead to a tremendous slowdown.

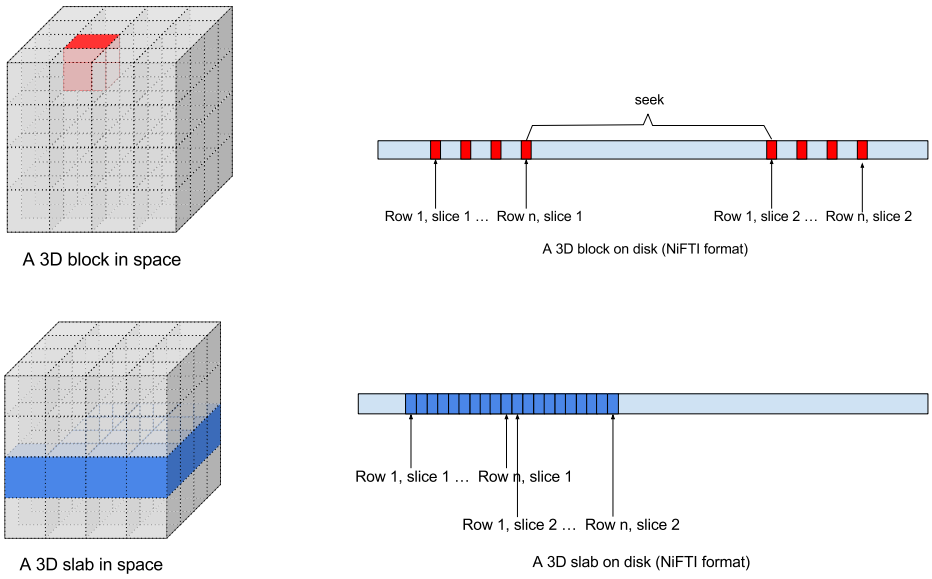

Figure 2: The cost of seeking to read/write 3D blocks and 3D slabs

2.4 Space-filling curves

Space-filling curve is one solution to reduce the seek time. Space-filling curve is using "z-order" or "Morton-order" to map multidimensional data to one dimension while preserving locality of the data points. The z-value of a point in multidimensions is simply calculated by interleaving the binary representations of its coordinate values. Once the data are sorted into this ordering, any one-dimensional data structure can be used [8].

The Open Connectome Project Data Cluster is a scalable database cluster for the spatial analysis and annotation of high-throughput brain imaging data, which is for scalable analysis and vision of high-throughput neuroscience [9]. Image data in the cluster is a dense multi-dimensional spatial array and partitioned into cuboids in all dimensions. Each cuboid gets assigned an index using a "Morton-order" space-filling curve. Space-filling curves minimize the number of discontiguous regions which successfully reduce seek time and preserve the spatial locality on disk.

However, this solution has the following limitations. First, space-filling curves cannot deal with a flexible geometry, for instance it could not be adapted to slabs, as the data in space-filling curves is organized to limit discontiguous segments in blocks.

To adapt it for slab, we essentially need a linear data format. Also it requires that existing pipelines are modified to use a format different than they currently use, which is mainly linear data format.

We need search for algorithms that would reduce the seek time regardless of the data format, and can be better fitting current image processing pipeline so that applications with flexible schemes can be served.

2.5 Sequential split and merge: Clustered reads and Clustered writes [1]

To avoid using complex data formats such as space-filling-curves, Clustered reads in the merging phase and Clustered writes in the splitting phase are two algorithms that can effectively reduce the seek time. The basic idea of Clustered reads is that they load multiple blocks in memory, concatenate them in a buffer and write 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.

Similar to Clustered reads in the sequential merge, Clustered writes in the sequential split will load several blocks from the origin image, split them in the buffer and write the buffer to each blocks.

For Clustered reads and Clustered writes, the number of seeks is actually less than Naive blocks, however, there are still some seeks happening for non-contiguous parts, which may slow down the process.

2.6 Big data infrastructure and parallelization

Big Data technologies have been used in the last decade to analyse enterprise data, internet of things (IoT) data, bio-medical data, etc. [10]. We are trying to apply them to large images to process and analyze the image information.

2.6.1 Parallel image processing

In general, computational platforms for processing large images can be categorized as multi-processor desktop computers, computer clusters, high-performance computing (HPC) resources with big shared memory, grid computing on loosely coupled and networked computers, and computing on novel hardware architectures (e.g., Graphical Processing Units (GPUs), Field Programmable Gate Arrays (FPGA)) [11]. The literature of parallel and distributed image processing has obviously been extensively studied and used in various platforms [12], [13], [14], [15], [16]. However, 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. The literature on this problem is remarkably scarce. Hadoop is the target infrastructure on which image analysis pipelines would run.

2.6.2 Hadoop and Hadoop Distributed File System

Hadoop is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. The Hadoop Distributed File System (HDFS) is the primary storage system used by Hadoop applications [17].

It is useful to have a distributed file system as it could parallelize the read, write process, and the storage can be scalable to several machines. It can also make the data highly available by making the data redundant. HDFS and Hadoop can also allow to have data locality. Data locality is the concept that makes the computation close to where the data resides, which can minimize unnecessary data transfers by moving data to computation node.

NameNode and DataNode of HDFS

HDFS has a master/slave architecture. A HDFS cluster consists of a single NameNode, and multiple DataNodes. The NameNode is the master server that manages the file system namespace and it also saves the information of where all the blocks for a given file are located. The DataNode is where the files are located. The file is usually split into one or more blocks and these blocks are stored in a set of DataNodes. The

DataNodes are responsible for storing and retrieving blocks when they are told to, and they report back to the NameNode periodically. HDFS supports exclusive writes only. When one client make a request to NameNode to open the file for writing, the NameNode will grant a lease to the client. Whenever another client tries to open the same file for writing, the NameNode will see that the lease for the file is already taken, therefore it will reject this request. HDFS cannot be randomly accessed, we always have to read and write one block a time. All HDFS communication protocols are on top of the TCP/IP protocol.

Three different modes in Hadoop

Hadoop can be run in three different modes. They are standalone mode, pseudo-distributed mode and fully distributed mode.

- Standalone mode
  - Default mode of Hadoop.
  - One node is used.
  - HDFS is not used.
  - Local file system is used for read, write.
- Pseudo-distributed
  - HDFS is used.
  - HDFS uses one node for both NameNode and DataNode.
- Fully distributed mode
  - HDFS is used.
  - Mode used in production.
  - Blocks are distributed across many nodes.
  - Different nodes will be used as NameNode and DataNode.

In our case, all the experiments done with HDFS are running in fully distributed mode.

Read and Write in HDFS

When we are trying to read or write file in HDFS, the client will make a request to the NameNode, the NameNode will respond to the client which DataNode(s) the client should be read from or writen to. For read, the client will directly contact the DataNode(s) and start reading data. For write, the DataNode(s) are chosen randomly and we cannot predict it. The client will directly contact the DataNode(s) to make sure they are ready, then the client starts to stream the data to the DataNode(s). When the DataNode(s) received all the data successfully, they will send back acknowledgments of job completeness to the NameNode.

Configuration of HDFS

The default HDFS block size is 64 MB. The blocks of a file are replicated for fault tolerance, and distributed to several DataNodes. The block size and replication factor can both be set in the configuration file. This is the sample configuration file (hdfs-site.xml) on both NameNode and DataNode to set the replication factor and block size.

```
<configuration>
    <property>
        <name>dfs.replication</name>
   <value>1</value>
    </property>
    <property>
        <name>dfs.blocksize</name>
        <value>652190720</value>
    </property>
</configuration>
```

HDFS client: HdfsCLI

HdfsCLI is a Python (2 and 3) library to make interacting with HDFS simpler. It support both secure and insecure clusters [18]. In the experiments, we use HdfsCLI to make interactions with HDFS.

2.7 Compression of large images

File compression has two benefits, it reduces the space needed to store files, and it speeds up data transfer across the network or to or from disk. On the other hand, file compression has an overhead on CPU time.

We focus on lossless file compression where the original data can be perfectly reconstructed from the compressed data, that is, the original and the decompressed data should be identical, which we can keep all the information. There are several lossless compression file formats and libraries available.

2.7.1 Compression algorithms and Python libraries

gzip

gzip is a widely used compression data format. The actual compression is performed by the so-called deflate/inflate-compression [19]. Deflate is an LZ77-based [20] method which is used in several general compression programs. Huffman coding is included in the method to encode the data [21]. Python has its built-in gzip library `gzip`, built on the zlib module [22] . The standard `gzip` class exposes a random access-like interface (via its seek and read methods), but every time we seek to a new point in the uncompressed data stream, the GzipFile instance has to start decompressing from the beginning of the file, until it reaches the requested location, which slows down the performance of a random-access I/Os. There is a library called `indexed_gzip`, built on `zran.c`, which can get around this performance limitation by building an index, which contains seek points, mappings between corresponding locations in the compressed and uncompressed data streams [23]. Additionally, we can load an image directly from a gzip compressed file by using NiBabel (Section 2.2.3).

LZ4

LZ4 [24] is a lossless compression algorithm, and also a compression format which leverages utilization of multiple core CPUs. LZ4 can compress and decompress data in parallel. LZ4 has raw block compression format which is called block format. For compressing an arbitrarily long file or data stream, multiple blocks are required. LZ4 has another format called Frame format. Organizing these blocks and providing a

common header format to handle their content is the purpose of the Frame format [1]. We use Frame format in our cases.

[1] `LZ4`: https://github.com/lz4/lz4

Chapter 3

Multiple reads/writes: a new algorithm to split and merge ultra-high resolution 3D images

Note: the Multiple reads/writes algorithm presented in this chapter was published together with Clustered reads/writes (Section 2.5) in an article accepted to 2017 IEEE International Conference on Big Data [2].

3.1 Introduction

As explained in Section 2.3, the algorithm used to split and merge ultra-high resolution 3D images significantly impacts the performance. Furthermore, to split and merge such data to a computing cluster (HDFS in our case) may bring overhead compared to running on a single machine. Therefore, the goal of this chapter is to find the most efficient algorithms for splitting and merging ultra-high resolution 3D images and investigate whether splitting and merging ultra-high resolution 3D images data on a cluster will bring any overhead.

First, we introduce a new algorithm to improve the performance by reducing seek time. Then we compare the algorithm to Clustered reads/writes in different situations. Finally, we investigate whether splitting and merging to a cluster brings any overhead.

Split and merge relate to the same dual problem in our context. We can always

obtain the splitting algorithm by switching reads and writes from merging algorithm, and vice versa.

All the algorithms presented in this chapter are implemented in Python and available on Github [1].

3.2 Sequential split and merge on single host

3.2.1 Notations

For simplicity, we assume that chunks are non-overlapping cuboids that all have the same dimension. We adopt the following notations (see Figure 3):

- $R = D^3$: number of voxels in the reconstructed image.
- b: number of bytes per voxel (in B).
- n: number of blocks.
- 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 slice/row/column: $\sqrt[3]{n}$.

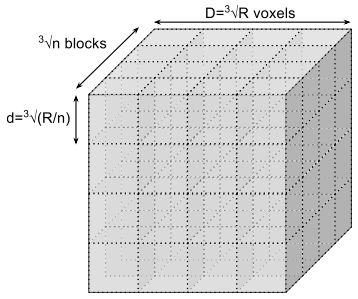

Figure 3: Notations

[1] https://github.com/big-data-lab-team/sam

3.2.2 Problem

For the sake of simplicity and to better understand the algorithms, we will investigate the algorithms on a single machine first, and complexity is measured in number of seeks.

As we mentioned in Section 2.3.2, naive algorithms performs very poorly to split or merge 3D blocks from ultra-high resolution 3D images, due to seek times. For Clustered reads and Clustered writes in Section 2.5, seeking is reduced compared to Naive blocks since contiguous parts of the buffer will be written or read without seeking. A given block is accessed only once during the whole process.

However, the number of seeks performed by Clustered reads and Clustered writes depend 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 rows in the reconstructed image. In the intermediary case, rows are complete but some slices can only be partially reconstructed [1]. We have to investigate a new algorithm which will improve this situation. This algorithm is called Multiple reads and Multiple writes.

3.2.3 Merge: Multiple reads

Multiple reads are shown in Algorithm 2.

Algorithm 2 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 <R*b 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 rows in write_range
10:         insert rows 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
```

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.

In the complexity analysis, we assume that m' represents an integer number k of sub-rows (Case 1, $k_1 < \sqrt[3]{n}$), of complete rows (Case 2, $k_2 < d$), of tile rows (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 4. In each of these 5 cases, we define v_i as follows:

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

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:

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

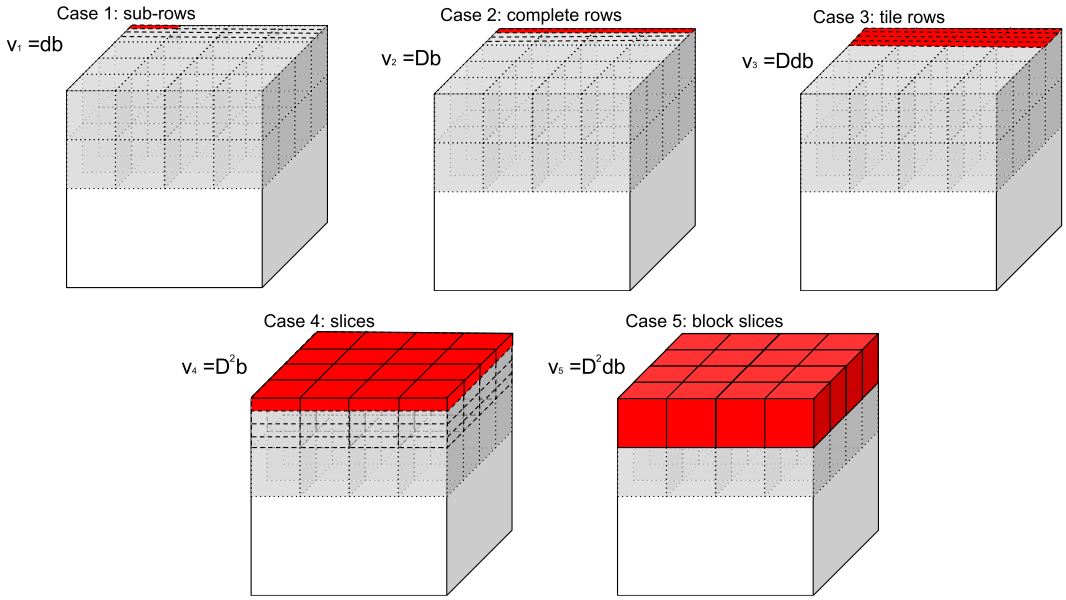

Figure 4: Memory-load configurations in Multiple reads (d=4, D=16, n=64, $k_1 = k_2 = k_3 = k_4 = k_5 = 1$). Red color shows the content of the memory load. Small dots depict block frontiers and long dashes depict rows and slices within blocks.

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:

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

It gives the following expression for N_{MR}:

$$N_{\mathrm{MR}} = \begin{cases} \left\lceil \frac{Rb}{m_1'} \right\rceil (k_1+1) - k_1 & \\ + (\sqrt[3]{n} D^2 \bmod k_1) & \text{if } d \le \frac{m}{b} < D \\ \left\lceil \frac{Rb}{m_2'} \right\rceil (\sqrt[3]{n}+1) & \text{if } D \le \frac{m}{b} < Dd \\ \left\lceil \frac{Rb}{m_3'} \right\rceil (k_3 \sqrt[3]{n}+1) - k_3 \sqrt[3]{n} & \\ + \sqrt[3]{n} \, (\sqrt[3]{n} D \bmod k_3) & \text{if } Dd \le \frac{m}{b} < D^2 \\ \left\lceil \frac{Rb}{m_4'} \right\rceil (\sqrt[3]{n}^2+1) & \text{if } D^2 \le \frac{m}{b} < D^2 d \\ \left\lceil \frac{Rb}{m_5'} \right\rceil (k_5 \sqrt[3]{n}^2+1) - k_5 \sqrt[3]{n}^2 & \\ + \sqrt[3]{n}^2 \, (\sqrt[3]{n} \bmod k_5) & \text{if } D^2 d \le \frac{m}{b} < R \end{cases}$$

And finally, using R and n as main variables:

$$N_{\mathrm{MR}} = \begin{cases} \left\lceil \frac{Rb}{m_1'} \right\rceil \left(\frac{m_1' \sqrt[3]{n}}{\sqrt[3]{Rb}} + 1 \right) - \frac{m_1' \sqrt[3]{n}}{\sqrt[3]{Rb}} & \\ + \left(\sqrt[3]{n} \sqrt[3]{R}^2 \bmod \left\lfloor \frac{m \sqrt[3]{n}}{\sqrt[3]{Rb}} \right\rfloor \right) & \text{if } \sqrt[3]{\frac{R}{n}} \le \frac{m}{b} < \sqrt[3]{R} \\ \left\lceil \frac{Rb}{m_2'} \right\rceil (\sqrt[3]{n}+1) & \text{if } \sqrt[3]{R} \le \frac{m}{b} < \frac{\sqrt[3]{R}^2}{\sqrt[3]{n}} \\ \left\lceil \frac{Rb}{m_3'} \right\rceil \left(\frac{m_3' \sqrt[3]{n}^2}{\sqrt[3]{R}^2 b} + 1 \right) - \frac{m_3' \sqrt[3]{n}^2}{\sqrt[3]{R}^2 b} & \\ + \sqrt[3]{n} \left(\sqrt[3]{nR} \bmod \left\lfloor \frac{m \sqrt[3]{n}}{\sqrt[3]{R}^2 b} \right\rfloor \right) & \text{if } \frac{\sqrt[3]{R}^2}{\sqrt[3]{n}} \le \frac{m}{b} < \sqrt[3]{R}^2 \\ \left\lceil \frac{Rb}{m_4'} \right\rceil \left(\sqrt[3]{n}^2+1 \right) & \text{if } \sqrt[3]{R}^2 \le \frac{m}{b} < \frac{R}{\sqrt[3]{n}} \\ \left\lceil \frac{Rb}{m_5'} \right\rceil \left(\frac{m_5' n}{Rb} + 1 \right) - \frac{m_5' n}{Rb} & \\ + \sqrt[3]{n}^2 \left(\sqrt[3]{n} \bmod \left\lfloor \frac{m \sqrt[3]{n}}{Rb} \right\rfloor \right) & \text{if } \frac{R}{\sqrt[3]{n}} \le \frac{m}{b} < R \end{cases} \tag{1}$$

Implementation

To avoid reading all blocks for each memory load, we pre-process the block headers. We read all the block headers and save their position information in a Python dictionary. When we need to read data, we will go over this dictionary first to check if this split is in this memory load, if not, we just pass it which will make the whole reading process faster. The data buffer used in 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 Multiple reads, dictionary entries are always contiguous in the reconstructed image. We tried to use a single NumPy array instead of Python dictionary 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.

3.2.4 Split: Multiple writes

Similarly to section 3.2.3, as shown in Algorithm 3, the main idea of Multiple writes is that blocks are written partially (line 9) to ensure that the memory buffer only contains contiguous bytes. Therefore, the buffer can be read continuously (line 5) in the original image, without seeking. However, a given block might be accessed multiple times for writing.

Algorithm 3 Buffered splitting of image with Multiple writes

```
1:  block_names = generate blocks name based on the blocks in each dimension
2:  start_index = 0 ; end_index=(m-1)
3:  read_range = (start_index, end_index)
4:  while end_index <R*b do
5:      data_in_range = read_data_from_original_image(read_range)
6:      for block_name in block_names do
7:          index_to_write = calculate_index(block_header)
8:          data_to_write = data_in_range[index_to_write]
9:          write_to_file(data_to_write)
10:     end for
11:     start_index = end_index + 1 ; end_index += m
12: end while
```

Implementation

Similar to Multiple reads, we also create a Python dictionary before split process start to avoid meaningless calculation. The key of the dictionary is the split's filename, the value is the split's header information. The data buffer used in Multiple writes is a

NumPy array, which can be read sequentially from original image by using NiBabel (Section 2.2.3) library in Python. After getting the NumPy array per memory load, we calculate the offset of memory load for each block and start to write the corresponding data.

3.3 Sequential split and merge on a cluster

To process ultra-high resolution images in parallel, images should be distributed in a cluster instead of a single node. We assume that HDFS is used in our experiments. We need to discover if there is any overhead and how much is the overhead to write to HDFS compared to writing data locally (single host). In addition to benchmarking, besides benchmark disk I/O, we also have to measure network I/O between each node in the cluster.

3.3.1 Split an image to HDFS by using Multiple writes

To split an image in the client and write data to HDFS by using Multiple writes, the only difference from running on single host (Section 3.2) is instead of writing data back to the local file system, we write data to a multiple nodes cluster, which is running HDFS.

Implementation

We use HdfsCLI library in Python to implement read/write files in HDFS. The library provides file–like API to make file I/O easier. For Multiple writes, we need to write header files to each split first. Before each write, we read header files from HDFS to get their indexes to filter out the splits not in the writing range, which will avoid meaningless calculation of indexes. However, each time we read header from HDFS, it brings overhead of reading compared to read directly from memory. Therefore, we built a cache in the client memory, saving the header information in memory. This cache is implemented by a Python dictionary. As each header file in our case is only 352 bytes data, they only occupy a little memory.

3.3.2 Split an image to HDFS by using Clustered writes

To split an image in the client and write data to HDFS by using Clustered writes, the only difference from running on single host, is instead of writing data back to the local file system, we write data to a multiple nodes cluster, which is running HDFS.

Implementation

When Clustered writes is on single host mode, we can use NiBabel library to save Nifti image directly by using `nib.save()`. However, NiBabel doesn't support writing to HDFS directly. Therefore, we write binary data directly to the files on the HDFS. We use HdfsCLI library in Python to write binary data to HDFS.

3.4 Experiments

3.4.1 Hardware

Running code on single host

We use a Dell Precision Tower 3620 workstation to run all the programs described in Section 3.2. For convenience, we use its hostname "consider" to refer to this machine from now on. "consider" has 32 GB of RAM and two disks:

- a Hard disk drive (HDD): HGST Travelstar 7K1000, 7200 rpm, 931GiB (1TB), firmware version JB0OA3W0;
- a Solid-state drive (SSD): SanDisk X400 2.5, 238GiB (256GB), firmware version X4130012.

Both drives use 512-byte logical sectors, 4096-byte physical sectors, SATA >3.1.

Running code on a cluster

We use four computers, which are all connected to a local network to run all the programs described in Section 3.3. Also, for convenience, we use hostnames to refer to machines.

- "gao-sparse"

  - CPU: 4 cores, Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz;
  - RAM: 8 GB;
  - Hard disk drive (HDD): Seagate Barracuda 7200.12, 7200 rpm, 250GB;
- "gao-dell"
  - CPU: 8 cores, Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz;
  - RAM: 8 GB;
  - Hard disk drive (HDD): Seagate Samsung SpinPoint M8 (AF), 5400 rpm, 1TB;
- "gao-dope"
  - CPU: 8 cores, Intel(R) Xeon(R) CPU X3450 @ 2.67GHz;
  - RAM: 8 GB;
  - Hard disk drive (HDD): Seagate Barracuda 7200.12, 7200 rpm, 500GB;
- "gao-ahoy"
  - CPU: 4 cores, Intel(R) Core(TM) i5 CPU 760 @ 2.80GHz;
  - RAM: 4 GB;
  - Hard disk drive (HDD): Seagate Barracuda 7200.12, 7200 rpm, 250GB;

We use a Bell Connection Hub as the router for our local network, speed is up to 1000 Mbps. Firmware version FAST2864_v6851E. Hardware version: 2864-000000-002.

3.4.2 Software

Operating system

"consider" runs CentOS Linux release 7.3.1611 with the XFS file system. All the nodes ("gao-sparse", "gao-dell", "gao-dope", "gao-ahoy") in the cluster runs Ubuntu 16.04.3 LTS with ext4 file system.

Hadoop and HDFS

All the nodes in the cluster are installed with Hadoop 2.7.4. Java version: 1.8.0_131.

3.4.3 Monitoring network I/O and disk I/O

Monitoring network I/O and disk I/O is important as we can study whether the nodes in the cluster have similar behaviors in the same condition. We can find out what affect system's performance.

Network I/O

We use `iperf3` [2] utility to measure network I/O. `iperf3` is a speed test tool for TCP, UDP and SCTP. We can use this to measure network I/O between these 4 nodes in the cluster.

```
# on the server side, run:
  iperf3 -s
# on the client side, run:
  iperf3 -c server-ip
```

Disk I/O

We can also use `iperf3` to measure disk I/O. For example, if we want to test node A's read speed (from disk to memory), we can execute the following scripts:

```
  # on the other machine B which is in the same cluster with the node A
  iperf3 -s
  # on the node A
  iperf3 -c B -F toReadFile
```

The sample output of machine A:

```
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 192.168.2.15, port 56262
```

[2] `iperf3`: https://iperf.fr/iperf-download.php

```
[  5] local 192.168.2.17 port 5201 connected to 192.168.2.15 port 56264
[ ID] Interval           Transfer     Bandwidth
[  5]   0.00-1.00  sec   107 MBytes  896 Mbits/sec
...
...
[  5]  10.00-10.04 sec 4.55 MBytes  930 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth     Retr
[  5]   0.00-10.04 sec 1.09 GBytes  932 Mbits/sec  0             sender
[  5]   0.00-10.04 sec 1.09 GBytes  930 Mbits/sec                receiver
```

The sample output of machine B:

```
Connecting to host gao-dope, port 5201
[  4] local 192.168.2.17 port 35584 connected to 192.168.2.15 port 5201
[ ID] Interval           Transfer     Bandwidth     Retr  Cwnd
[  4]   0.00-1.00  sec   113 MBytes  951 Mbits/sec  0    370 KBytes
...
...
[  4]   9.00-10.00 sec  111 MBytes  935 Mbits/sec  0    389 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth     Retr
[  4]   0.00-10.00 sec 1.09 GBytes  936 Mbits/sec  0             sender
[  4]   0.00-10.00 sec 1.09 GBytes  935 Mbits/sec                receiver
```

To test node A's write speed (from memory to disk), we need to run a longer test to factor out network buffering issues. We can execute the following scripts:

```
# on the node A
iperf3 -s -F toWriteFile
# on the other machine B which is in the same cluster with the node A
iperf3 -c A -i 1 -t 40
```

The results of the slowest test will indicate the bottleneck of our whole system, which means we can find our system are disk limited or network limited.

3.4.4 Data

We use the 3850x3025x3500 Big Brain image [3] for running all splitting algorithms. We used 50/125/250/500 non-overlapping chunks with 2 bytes per voxel split by 3850x3025x3500 Big Brain image to run merging algorithm.

3.4.5 Algorithm parameters

Single host mode

For merging blocks by using Multiple reads and splitting image by using Multiple writes on single host mode, we test our program with 3 GB, 6 GB, 9 GB, 12 GB and 16 GB of memory. This corresponds to the case 4 from 3 GB to 12 GB, and case 5 for 16 GB in Figure 4. 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.

Cluster mode

For splitting an image to a cluster by using Multiple writes and Clustered writes, we test our program with 3 GB of memory. For the sake of simplicity, we set the client and NameNode in HDFS to the same machine. We set "gao-sparse" as our NameNode and also we set 2 DataNodes ("gao-dell" and "gao-dope") and 3 DataNodes ("gao-dell", "gao-dope", "gao-ahoy") to test if different number of DataNodes will affect our performance. We did 5 repetitions and also dropped the caches before each run. For the configuration of HDFS, we set replication factor to 1 and also set block size to a proper value (622 M when 125 splits), which should be larger than one split size, to make sure the block will not be automatically split again in HDFS.

[3] 2015 Big Brain release with 40-micrometer isotropic resolution available at ftp://bigbrain.loris.ca/BigBrainRelease.2015/3D_Blocks/40um

3.5 Results

3.5.1 I/O testing

Table 1 shows the average rates of Network I/O, disk to memory, and memory to disk rates. We can clearly see that the bottleneck of our experiments is disk write rates, which means the overhead of the network is negligible compared to the write overhead. So it makes sense to focus on optimizing the writes.

Table 1: I/O testing

	Network I/O rates	**Disk to memory rates**	**Memory to disk rates**
Average	938.5 Mbits/s	935.25 Mbits/s	26.05 Mbits/s

3.5.2 Seeks for Multiple reads on single host mode

The number of seeks of Multiple reads is reported in Figure 5. The baseline corresponds to Clustered reads, and the model is the Equation 1 in Section 3.2.3. The average relative model error is 26.8%, presumably explained by the fact that the model assumes cubic blocks while we used non-cubic ones in the experiment. For Multiple reads, our complexity analysis also assumed that one of the 5 cases in Figure 4 was used while they are usually blended in practice. Overall, the model correctly explains the observations, which indicates that our implementation is correct. Furthermore, Multiple reads has fewer number of seeks compared to the Clustered reads on 3 G, 6 G, 9 G, 12 G memory.

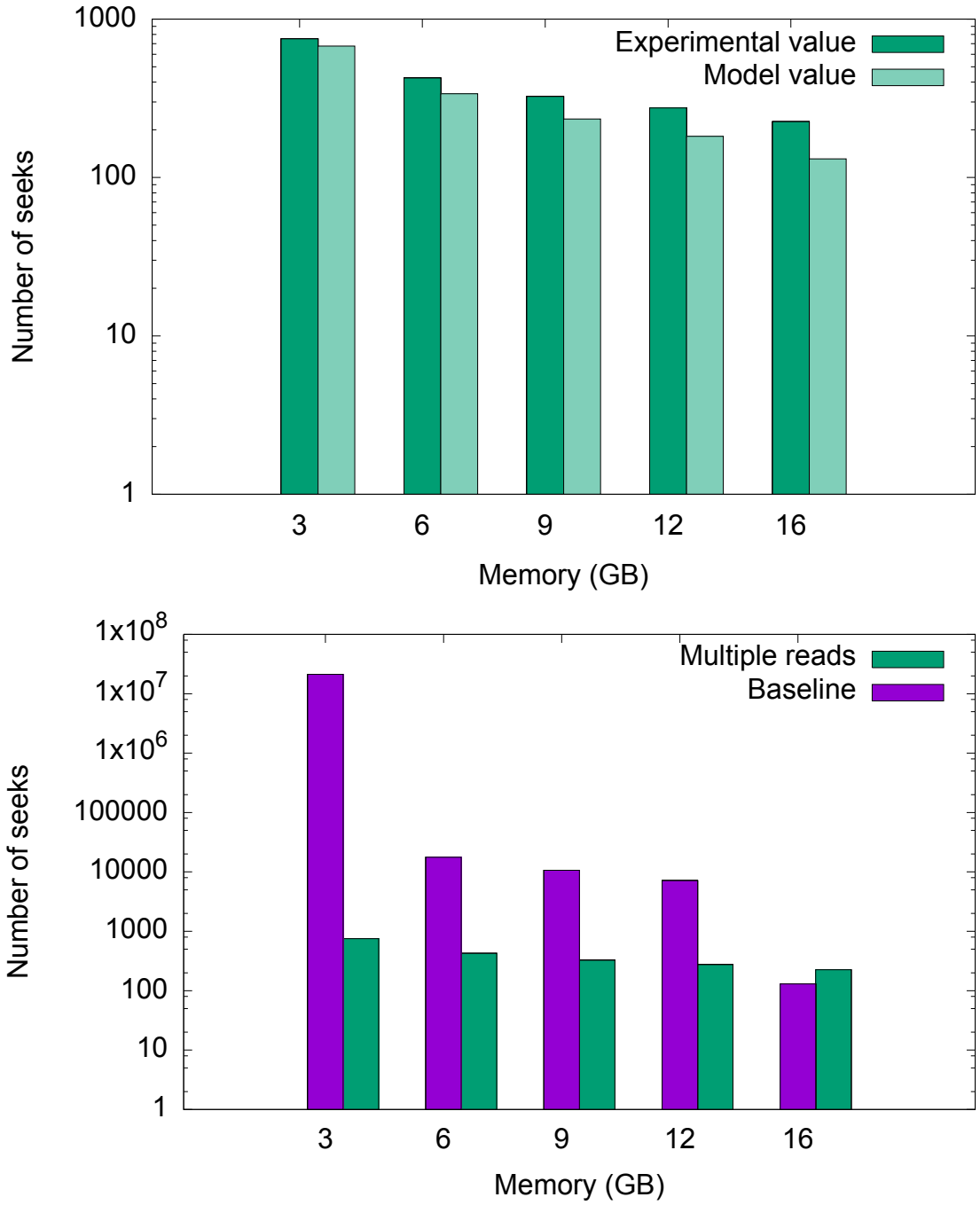

Figure 5: Number of seeks for Multiple reads on 125 splits. Top: experimental values compared with model of Equation 1. Bottom: Multiple reads compared with baseline: Clustered reads.

3.5.3 Merge time for Multiple reads on single host mode

Figure 6 shows that Multiple reads can bring more performance improvement compared to Clustered reads for low memory values, when the number of splits to be merged is 125. Multiple reads are 8.4 times faster than Naive way on HDD and 5.3 times on SSD. As SSD is better on seeking than HDD, we can see the improvements of Multiple reads when running on SSD compared to HDD.

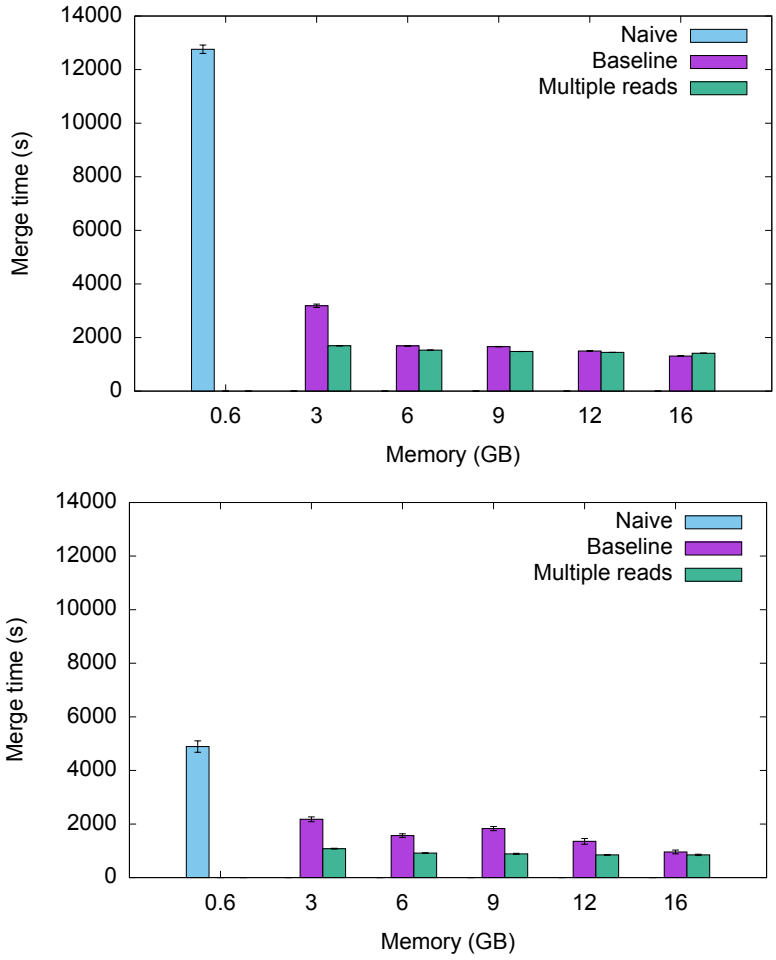

Figure 6: Merge time for 125 splits on Multiple reads. Top: HDD. Bottom: SSD. Averages over 5 repetitions. Error bars show ± 1 standard deviation.

Figure 7 shows how the total merge time on 125 splits breaks down to read, write, seek and overhead time for Multiple reads. Multiple reads almost annihilate the seek time. 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 NiBabel package, which is more efficient than using native Python – as is the case with our writes. The overhead time is small for Multiple reads.

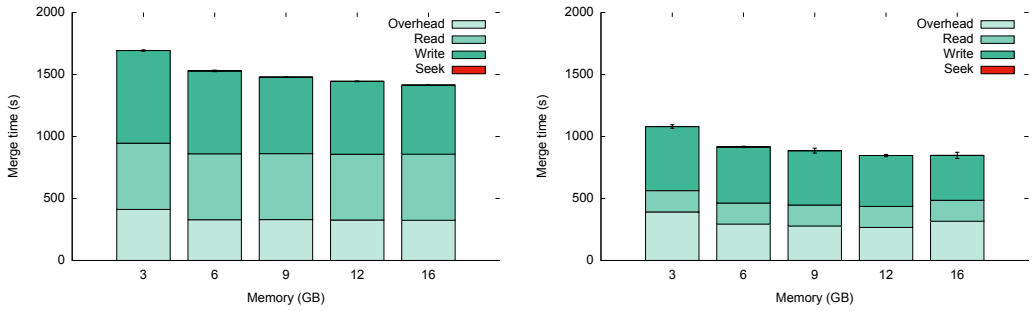

Figure 7: Breakdown of total merge time for 125 splits on Multiple reads. Left column: HDD. Right column: SSD.

3.5.4 Effect of the number of splits

Figure 8 shows that for each memory load, different number of splits can bring different performance. As we can see for 500 splits and 250 splits, Multiple writes is slightly slower than Clustered writes. Especially for 500 splits, Multiple writes are always slower than Clustered writes no matter on which memory configuration. However, when the number of splits is 50, Clustered writes are always slower than Multiple writes.

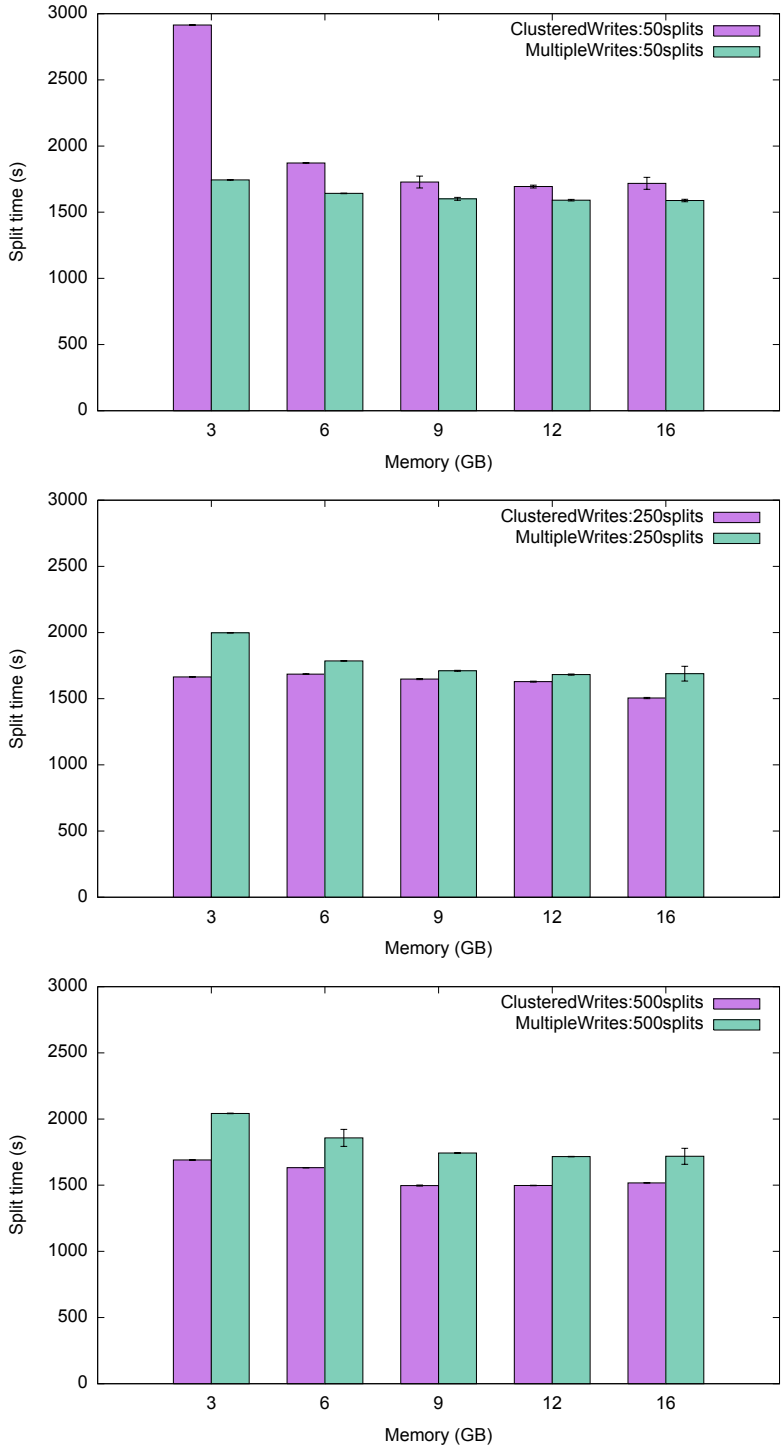

Figure 8: Split time for Multiple writes and Clustered writes based on different number of splits on single host. Top: 50 splits. Middle: 250 splits. Bottom: 500 splits.

Regarding the merge (Figure 9), we can see that for 50 splits, Clustered reads can

be 6.2 times slower than Multiple reads with 3 G of memory. For 500 splits, Multiple reads is a bit slower than Clustered reads.

From the figures, we can see split and merge have the similar behavior for different number of split. We can conclude that for a few large splits, Multiple reads/writes can do better than Clustered reads/writes, and for a lot of small splits, Clustered reads/writes is better than Multiple reads/writes.

The reason is when there are a few large splits, the overhead of seeking in Clustered reads/writes is larger than the overhead of opening each split in Multiple reads/write. On the other hand, if there are lots of small splits, the overhead of opening each split in Multiple reads/write is larger than the overhead of seeking in Clustered reads/writes.

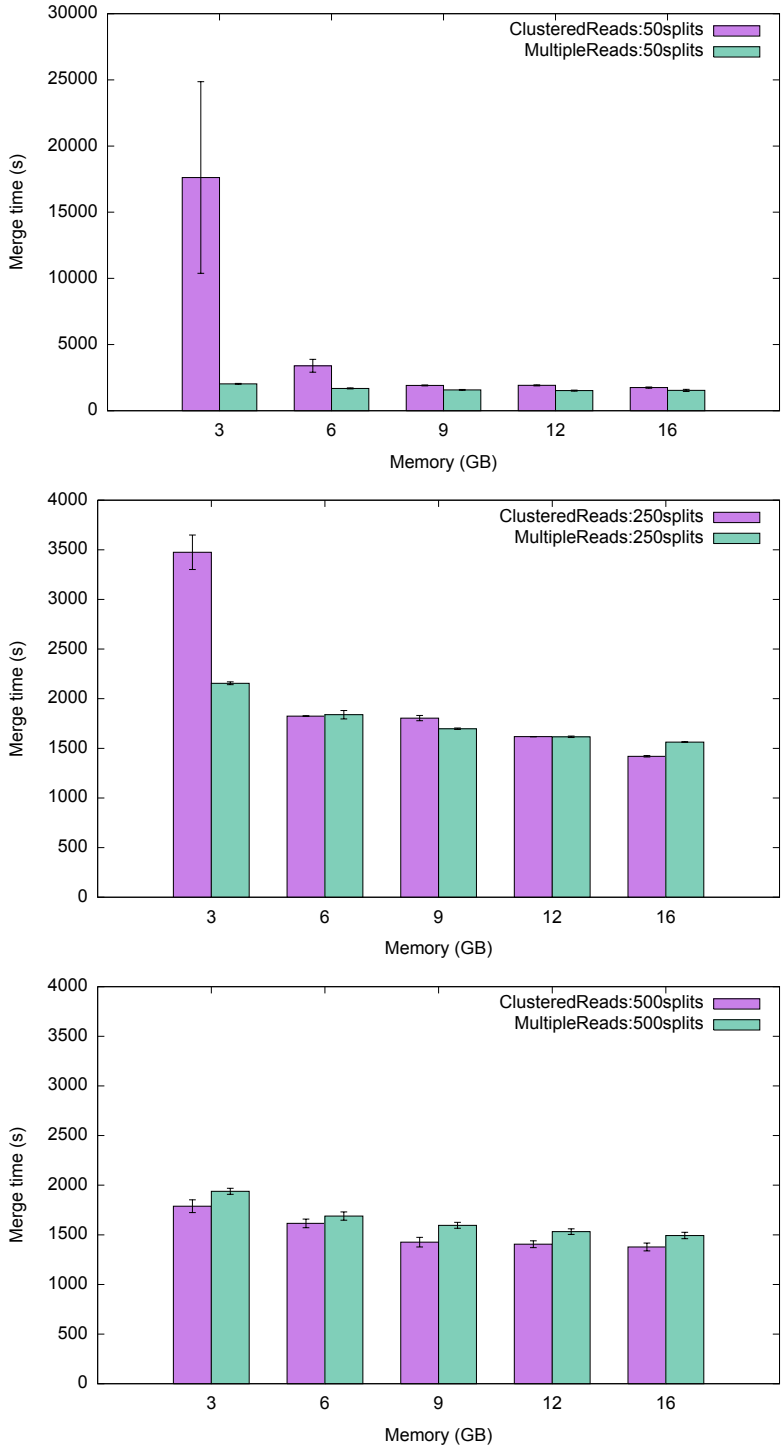

Figure 9: Merge time for Multiple reads and Clustered reads based on different number of splits on single host. Top: 50 splits. Middle: 250 splits. Bottom: 500 splits.

3.5.5 Overhead of cluster mode

From Figure 10, we can see there is little difference between single host mode and cluster mode. No matter the split method, the differences of split time is trivial. We conclude that the overhead of HDFS is negligible.

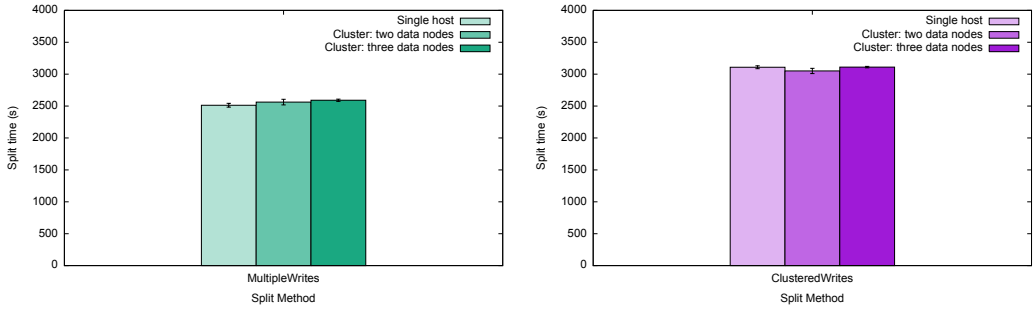

Figure 10: Split time for Multiple writes and Clustered writes based on on single node, two DataNodes and three DataNode. Left: Multiple writes. Right: Clustered writes.

3.6 Conclusion

Multiple reads and writes can both 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.

For splitting, if the number of splits is high, we prefer to use Clustered writes, as for Multiple writes, we have to open every split multiple times and append data to it. In Clustered writes, each split is just accessed once.

In the single host mode, if the memory is low, we prefer use Multiple reads/writes than Clustered reads/writes, as Clustered reads/writes needs more seeking in few memory configuration. If we have enough memory, we can either use Multiple reads/writes or Clustered reads/writes, since their performance is comparable. For the cluster mode, as expected, splitting image to HDFS won't bring any overhead. We can also conclude this based on the I/O benchmark. Clearly, the bottleneck of our program is disk I/O, so using HDFS won't degrade the system's performance.

Chapter 4

Parallelization of the Multiple reads/writes algorithm

4.1 Introduction

As mentioned in Section 3.6, the bottleneck of our program is disk I/O. To further improve our system's performance, we should find solutions to improve our data throughput to/from the disk. In the HDFS cluster, there will be several disks to which we could write.

Since we assume that the images are located on a single disk, parallelizing the reading process seems difficult, although parallel filesystems might be useful here. However, there is only one disk used at one time in our system when doing the write process. If we make all the disks write concurrently, we may improve performance of the sequential algorithm.

As we mentioned in Section 2.6.2, we cannot predict which DataNode is chosen to write data. Therefore, to achieve writing data to disks concurrently, we should make the data always available to each disk. In other words, we should try to send data to each disk in parallel. So in the client side, we decided to leverage multi-core CPU of the client and use multi-threading to make requests to several DataNodes simultaneously. Therefore, the goal of this chapter is to combine multi-threading and our algorithms in the previous chapter, to implement writing imaging data to the different disks in parallel.

Firstly, we introduce how Multiple writes and Clustered writes can be multi-threaded. Then we re-run these algorithms using different parameters and benchmark their performance.

4.2 Parallelizing the splitting algorithms

4.2.1 Multiple writes in parallel

Algorithm 4 shows that to apply multi-threading to the writing process in Multiple writes, the main idea is that, after each read from one memory load (line 6), we will split the data in memory to several pieces (line 7), and trigger several threads to partially write data to image files at the same time (line 12, 13).

Algorithm 4 Splitting of image with Multiple writes in parallel

```
1: block_names = generate blocks name based on the blocks in each dimension
2: start_index = 0 ; end_index=(m-1)
3: nThread = Number of threads
4: read_range = (start_index, end_index)
5: while end_index <R*b do
6:    data_in_range = read_data_from_original_image(read_range)
7:    block_names_pieces = split block_names to nThread pieces
8:    for block_names_piece in block_names_pieces do
9:       for block_name in block_names_piece do
10:         index_to_write = calculate index based on block_name(position, etc.)
11:         data_to_write = data_in_range[index_to_write]
12:         Thread t = create_new_thread(HDFS_client.write(data_to_write))
13:         t.start()
14:      end for
15:   end for
16:   start_index = end_index + 1 ; end_index += m
17: end while
```

Implementation

For the multi-threading part, we use Python's built in threading module to trigger multi-thread. We set write function as the target to create a new Python thread. We start the threads first, then we use `thread.join()` to block the calling thread until all the threads represented by this instance terminates. We use HdfsCLI library in Python to implement read/write files in HDFS.

4.2.2 Clustered writes in parallel

To apply multi-threading to the writing process in Clustered writes, the main idea is that, after we read several splits from the original image on one memory load, we trigger several threads to completely write the splits at the same time.

4.3 Experiments

4.3.1 Hardware, Software

We are still using one NameNode and three DataNodes to run our program. Hardware and software used in this chapter is the same as the cluster mode of the Section 3.4.1 and Section 3.4.2.

4.3.2 Data

As with the previous chapter, the image selected to perform such tests was the 40μm resolution BigBrain image mentioned in as the Section 3.4.4.

4.3.3 Algorithm parameters

For both Multiple writes and Clustered writes in the split algorithm, we set number of splits to 125 and 500. We trigger 1 thread, 8 threads, 16 threads besides the main thread with 2 G memory to each algorithm with different number of splits. We measure their read time, write time and total split time on each configuration.

4.4 Results

4.4.1 Split time for Multiple writes and Clustered writes with 125 splits

As shown in Figure 11, we can see that with 125 splits, Multiple writes are always faster than Clustered writes based on different number of threads. In addition, the performance improvement provided by multi-threading is higher for Multiple writes than for Clustered writes.

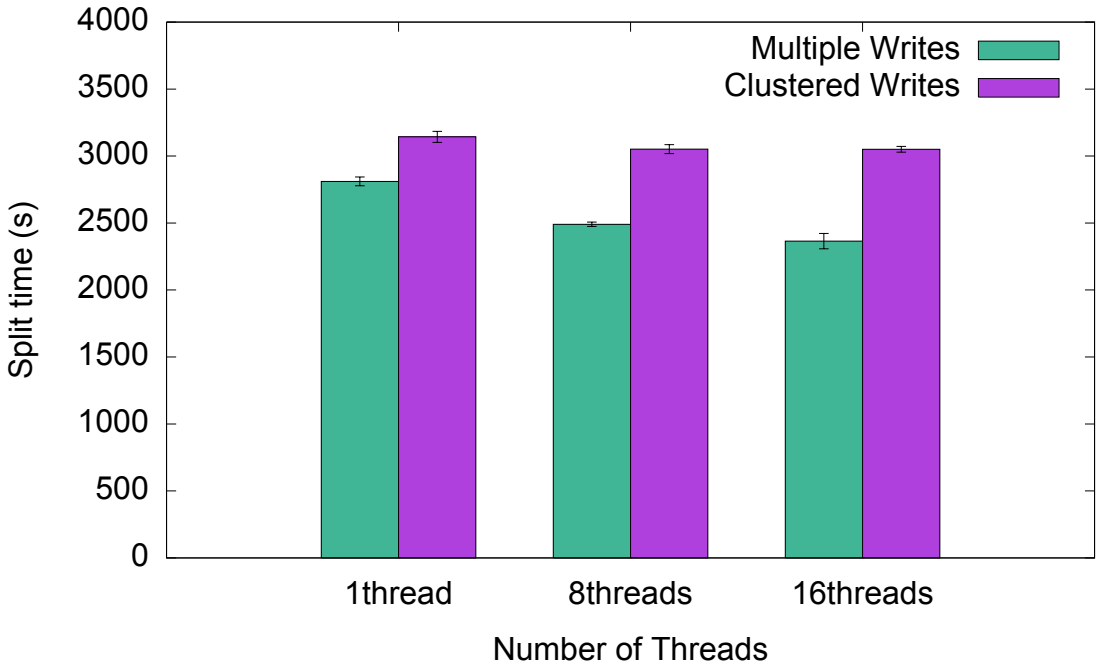

Figure 11: Total split time on multiple threads, 125 splits, with 2 G memory.

We can see from Figure 12, that multi-threading reduces the write time for Multiple writes. However, for Clustered writes, it doesn't have any obvious improvements on write, hence there is no apparent performance speed up of total time.

The reason behind this is to split the image with 125 splits in Clustered writes, the parallel portion of the write process is less than Multiple writes. HDFS will randomly choose DataNode to write and it is unpredictable. As we have three DataNodes, and in Multiple writes we write partial data to each file by using multiple threads at the same

time, we can always make sure that HDFS can write data in fully parallel by using multi-threading. However, as Clustered writes write whole file at one time, we only have 2 G memory and each split is almost 700 MB (125 splits), and 2 $G/700M \leq 3$, thus for three DataNodes, we cannot make sure to get the maximum parallelizing. In other words, due to the lack of concurrent writes in HDFS, parallelisation achieved is limited by the number of splits in the memory load which is larger for Multiple writes than for Clustered writes.

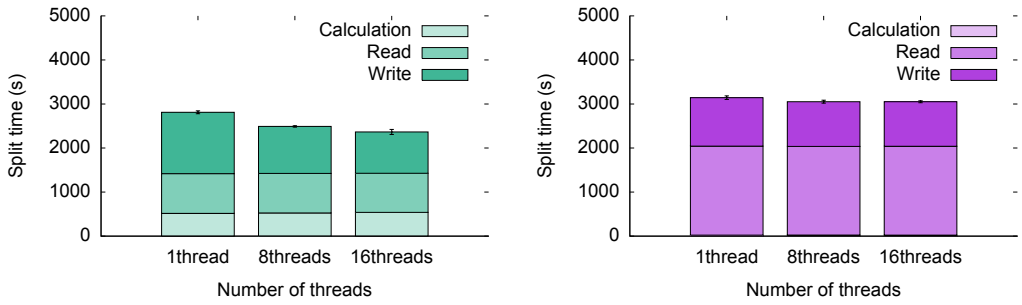

Figure 12: Split time breakdown for Multiple writes and Clustered writes based on different number of threads. Left: Multiple writes on 1 thread, 8 threads, 16 threads, with 2 G memory. Right: Clustered writes on 1 thread, 8 threads, 16 threads, with 2 G memory.

4.4.2 Split time for Multiple writes and Clustered writes with 500 splits

For Figure 13, we can see that with 500 splits, Multiple writes are always slower than Clustered writes for any number of threads.

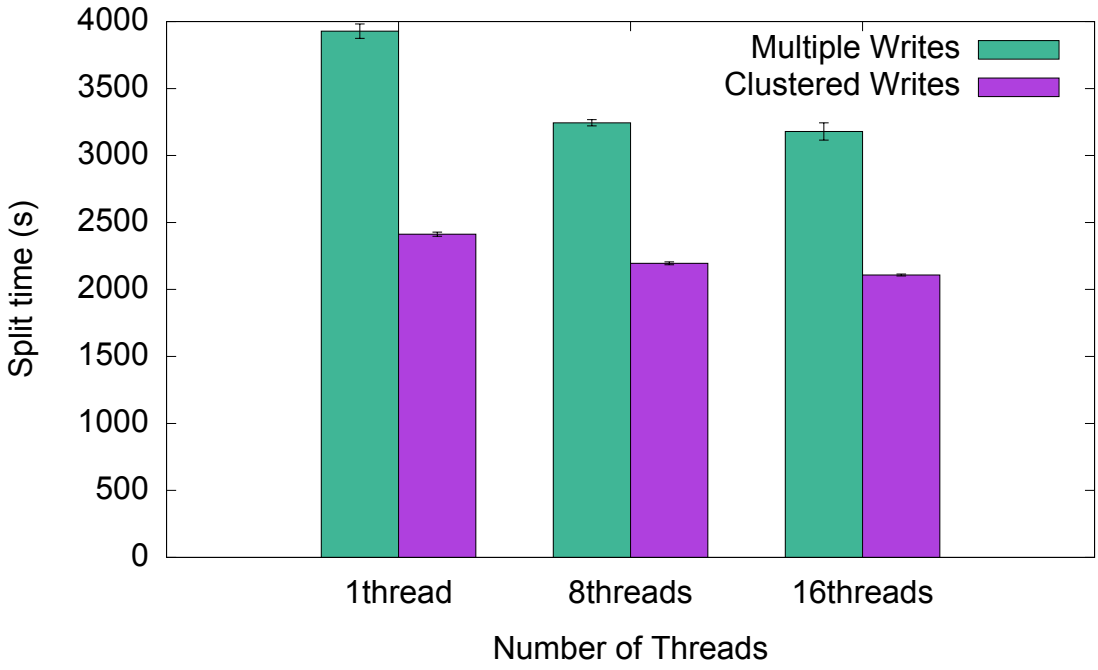

Figure 13: Total split time on multiple threads, 500 splits, with 2 G memory.

With 500 splits, from Figure 14, we can see both Multiple writes and Clustered writes reduce the write time, which lead to performance improvement on total split time.

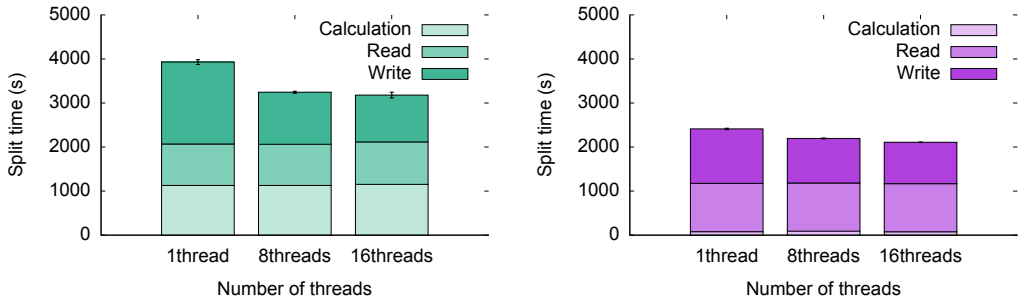

Figure 14: Split time breakdown for Multiple writes and Clustered writes based on different number of threads. Left: Multiple writes on 1 thread, 8 threads, 16 threads, with 2 G memory. Right: Clustered writes on 1 thread, 8 threads, 16 threads, with 2 G memory

When the number of splits is 500, one memory load comes more splits because

splits are smaller. We can say that we can get the higher parallelization than with 125 splits for Clustered writes.

4.4.3 Speed up analysis

To calculate speed up, for each algorithm, we use the execution time obtained with one thread ($T_{1thread}$) as the base value, and we make $T_{1thread}$ divided by $T_{16threads}$, $T_{8threads}$ and $T_{1thread}$ to get the speed up for 16 threads, 8 threads and 1 thread.

We compare each speed up with Amdahl's law [25] formulated as follows:

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{2}$$

$S_{latency}$ is the theoretical speedup of the execution of the whole task; s is the speedup of the part of the task that benefits from improved system resources; p is the proportion of execution time that the part benefiting from improved resources originally occupied.

In our case, s is always equal to 3, since we have 3 DataNodes. We calculate p by equation $p = T_{write}/T_{Total}$, where T_{write} stands for write time for 1 thread, and T_{Total} stands total time for 1 thread. As p and s are constant, based on Amdahl's law, $S_{latency}$ is also constant.

The measurement of speed up and comparison with Amdahl's law are shown in Figure 15. We can see that when the number of splits is 125, Multiple writes has 1.18 times speed up on 16 threads and Clustered writes has 1.03 times speed up on 16 threads. For number of splits is 500, Multiple writes has 1.23 times speed up on 16 threads, and Clustered writes has 1.14 times speed up on 16 threads.

Clearly there is a gap between the value of experiments and Amdahl's law and several factors may cause this. As we have 4 CPU cores in the client (Section 3.4.1), we cannot assure each core runs only one thread, context switch must happen during the process. Second of all, HDFS will randomly choose DataNodes to write, it does not distribute data in the particular order, which means the DataNodes may not be fully parallized sometime.

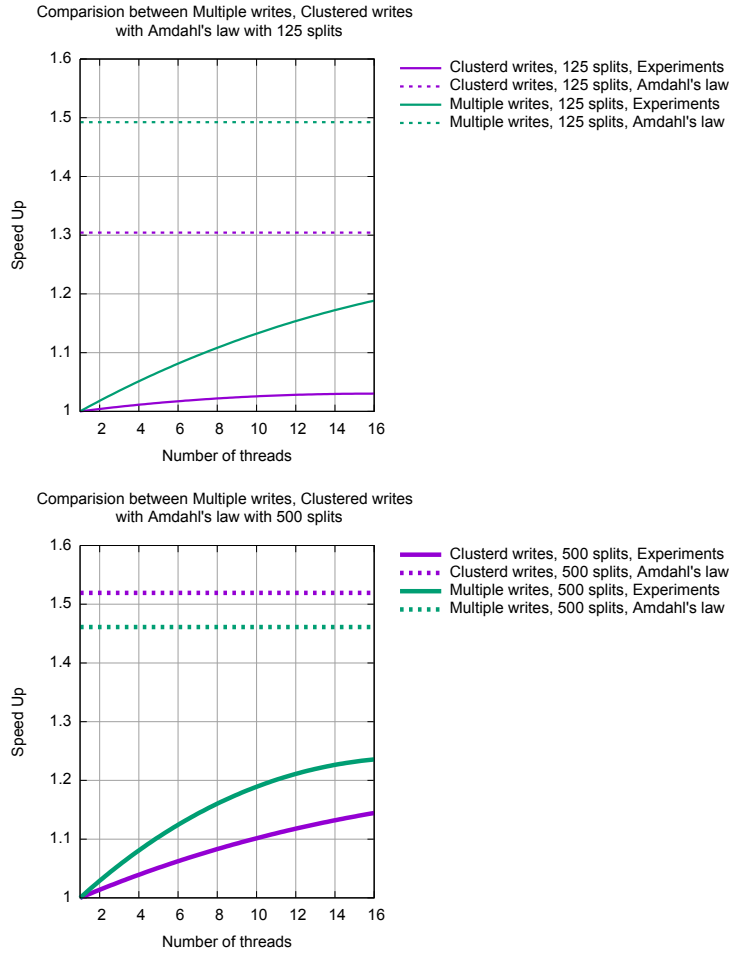

Figure 15: Comparision between Multiple writes, Clustered writes with Amdahl's law with 125, 500 splits. Green line stands for Multiple writes, purple line stands for Clustered writes. Light line: speed up with 125 splits. Bold line: speed up with 500 splits. Dash line: Amdahl's law. All with 2 G Memory.

4.5 Conclusion

Splitting and merging ultra-high resolution 3D images in parallel can improve system performance. We use multi-threading to parallelize the write process.

For the split algorithms, as expected, based on conclusion from Section 3.6, on low memory configuration, to get fewer number of splits, Multiple writes is always faster than Clustered writes no matter of number of threads. However, with more splits, Clustered writes is better than Multiple writes no matter of number of threads. This

is consistent with what we had measured in a sequential environment.

Furthermore, Multiple writes can always gain more performance improvements than Clustered writes because it copes better with lack of the concurrent writes in HDFS. No matter of different number of splits, both algorithms can gain more speed up by splitting to more splits.

Last but not least, it does not mean more threads can bring more speed up. Based on Amdahl's law, when the number of CPU cores achieve some value, there will be little improvement. Also, network I/O bottleneck is another bottleneck for speed up. As we measured the network I/O (Table 1) in our environment is 938.5 Mbit/s. If our data to be transferred per second is larger than this, there will not be any performance improvement.

Chapter 5

On-the-fly compression of the Multiple reads/writes algorithm

5.1 Introduction

So far, we have studied algorithms for splitting and merging the ultra-high resolution 3D images. Processing data in single node or in parallel, we tried to improve performance by reducing seek time of the disk (Chapter 3) and using multi-threading to write in parallel (Chapter 4). All these methods or algorithms are focusing on the disk I/O.

In this chapter, we focus on improving performance by using on-the-fly compression. Using on-the-fly compression has two benefits, it can speed up data transfer across the disk and the network. Therefore, the goal of this chapter is to find a solution to apply on-the-fly compression to our algorithms, and investigate if it can bring any improvement.

We have studied two data compression formats, "gzip" and "LZ4" in Section 2.7.1. At first, we show how we combine on-the-fly compression to our algorithms, and then we make experiments on "gzip" and "LZ4" compression formats. Last but not least, we analyze "gzip" and "LZ4" compression format and compare their performance to the uncompressed case for our algorithms.

5.2 Lossless on-the-fly compression

Lossless compression denotes data compression algorithms where the original data can be perfectly reconstructed from the compressed data, that is, the original and the decompressed data should be identical. We should use lossless compression in our system as we don't want to lose any information of our images.

On-the-fly compression means the data are compressed when they are being read or written. By using on-the-fly compression, especially for the sparse data of the image, we will write fewer data to the disk, which will make our whole write process faster.

However, compression has a computational overhead. For disk read/write rates and network throughput, compression might be useful or not depending on the compression rate.

5.3 Algorithm and implementation

Compression can always be applied to Multiple reads/writes and Clustered writes. However, it cannot be applied to Clustered reads because the write process in Clustered reads need reverse seeking, which is not implementable in the gzip and LZ4 format.

For writing compressed data, the basic idea is before we try to write binary data to HDFS (line 12 in Algorithm 4), we compress data in memory first. For data compression algorithms, we mainly focus on these two data compression formats, "gzip" and "LZ4".

To compress data to gzip format, we use Python's built-in library, "gzip". To compress data to LZ4 format, we use the "lz4" PyPI package in Python. However, in the implementation, Python's built-in "gzip" doesn't support write binary data directly to HDFS, since the implementation of in memory compression are wrapped in Python's built-in "gzip" library, we have to extend "client" class in HdfsCLI, and override its write function to make gzip suitable for HDFS.

For LZ4 compression format, "lz4" package has API to compress the data in memory. Hence, we can compress the data first, then write the compressed data to HDFS directly.

For reading LZ4 compressed blocks, there is no direct solution to random access to

LZ4 file, we have to decompress the whole file first, write to the disk and use NiBabel to load image. For gzip file, we can use NiBabel to load a gzip file directly.

5.4 Experiments

5.4.1 Hardware, software

We are still using one NameNode and three DataNodes to run our program. Hardware and software used in this chapter is the same as the cluster mode of Section 3.4.1 and Section 3.4.2.

5.4.2 Data

As before, the image selected to perform such tests was the 40μm resolution BigBrain image mentioned in Section 3.4.4.

5.4.3 Algorithm parameters

We have done two sets of experiments. All the experiments are splitting to 125 splits with 2 G of memory. First, we read image data in uncompressed format, process and compress them in memory, and write the gzip/LZ4 compressed data to HDFS. We use 1 thread, 8 threads, 16 threads besides the main thread to run the program. Then we read image data in gzip/LZ4 compressed format and after processing, we write gzip/LZ4 compressed data to HDFS. We use 16 threads besides the main thread to run the program. We set compression level of LZ4 and gzip to the highest level, that is, we can get the highest compression rate.

5.5 Results

5.5.1 Read uncompressed, write gzip/LZ4 compressed

Figure 16 shows that when we are using compression formats, using multi-threading can still bring performance improvements.

Furthermore, for both Multiple writes and Clustered writes, when we read uncompressed data and write gzip/LZ4 compressed, LZ4 data format can bring some

performance improvement over write uncompressed data. However, write gzip data format slow down the whole system performance.

The reason is that LZ4 is extremely fast compression algorithm and it will utilize multi-core CPUs to compress data which will bring little overhead. On the other hands, compression in gzip is relative slow. Although we will write less for gzip compressed compared to the uncompressed data, but it takes more time of the CPU to compress the data in memory.

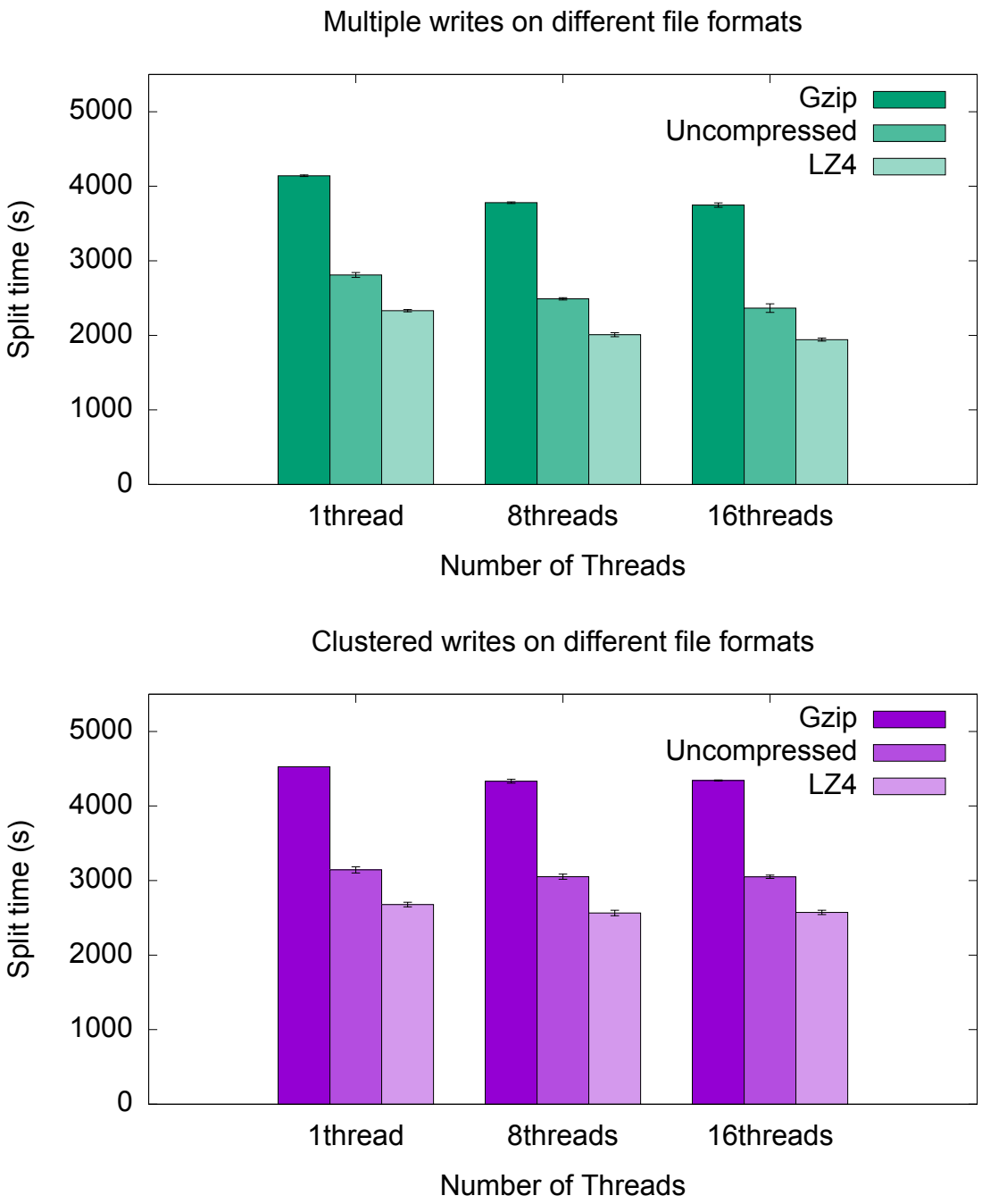

Figure 16: Different compression formats for Multiple writes and Clustered writes. Top: experiments on Multiple writes. Bottom: experiments on Clustered writes.

The breakdown of total time in Figure 17 can still prove that LZ4 reduces write

time and makes the whole process faster. For gzip, we cannot calculate in memory compression time since it is built in Python's library. So the write time includes the compression time, which takes longer than using uncompressed and LZ4 compressed data.

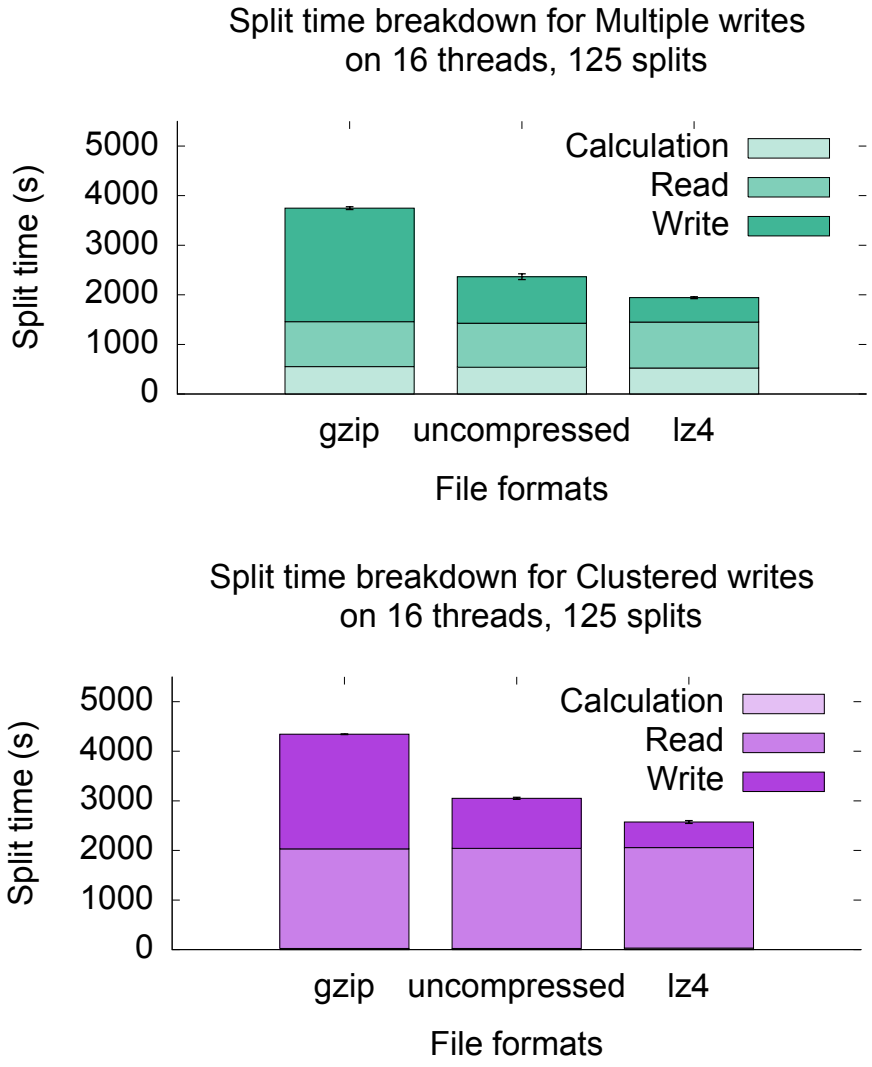

Figure 17: Split time breakdown of different compression formats for Multiple writes and Clustered writes. Top: experiments on Multiple writes. Bottom: experiments on Clustered writes.

From Figure 18, we can see that no matter writing gzip compressed or LZ4 compressed, for fewer memory (Memory = 2 G), Multiple writes is better than Clustered writes on 16 threads, 125 splits, which is the same as before.

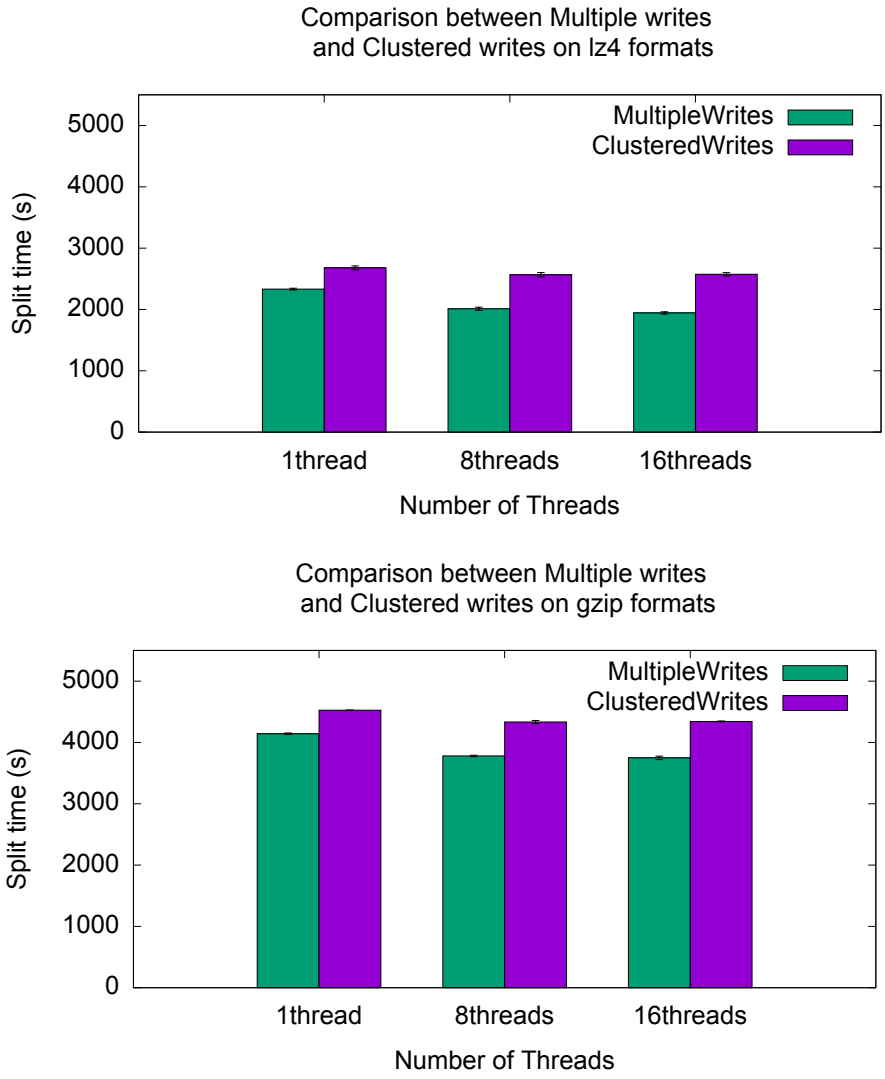

Figure 18: Split time comparison between Multiple writes and Clustered writes on LZ4 and gzip format. Memory=2 G, using 1,8,16 threads. Top: comparison in LZ4 format. Down: comparison in gzip format.

5.5.2 Read compressed, write gzip/LZ4 compressed

When reading compressed, and writing compressed, we can see from Figure 19, gzip compressed takes extremely longer than LZ4 compressed and uncompressed file for either Multiple writes and Clustered writes. Reading and writing LZ4 compressed won't bring performance improvement compared to read and write uncompressed,

since read gzip/LZ4 compressed file can bring more overhead than reading uncompressed files directly.

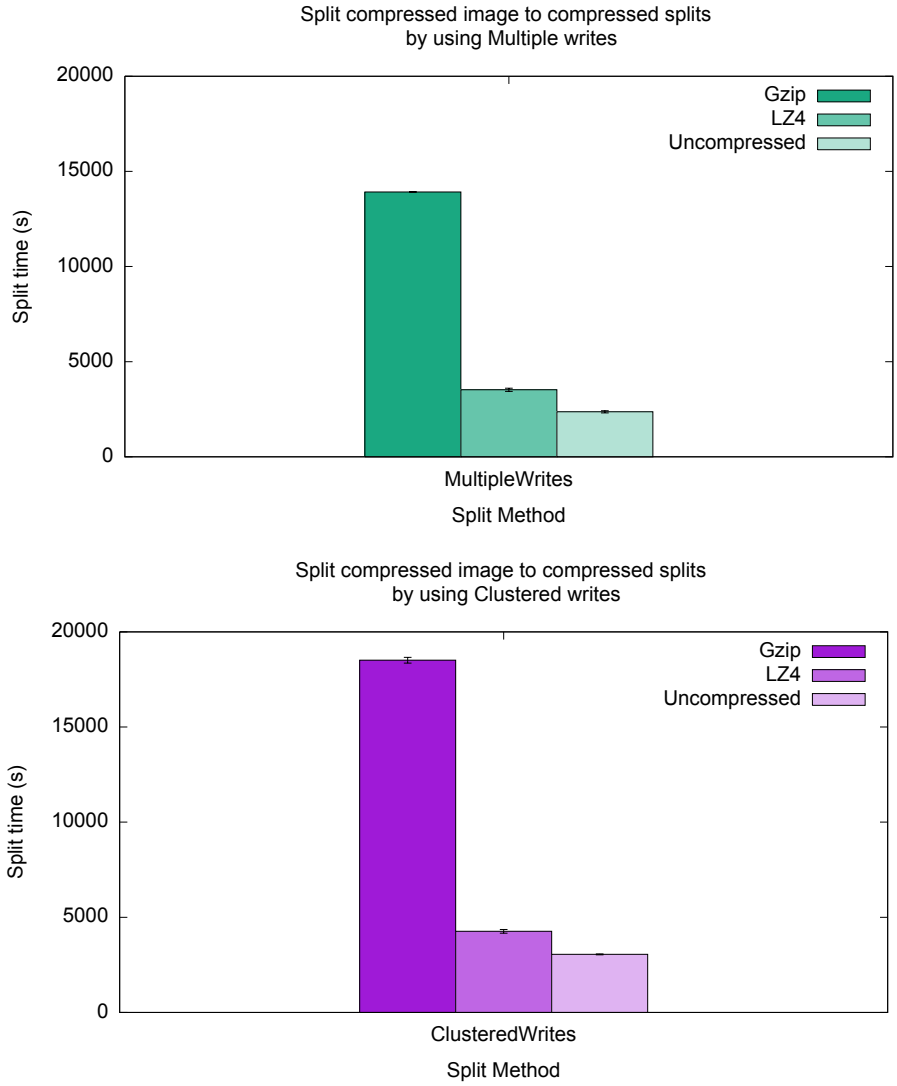

Figure 19: Split compressed image to compressed splits in Multiple writes and Clustered writes. Memory=2 G, using 16 threads. Top: experiments on Multiple writes. Bottom: experiments on Clustered writes.

From Figure 20, we can see that for splitting compressed image to compressed splits, for fewer memory (Memory = 2 G), Multiple writes is always better than Clustered writes on 16 threads, 125 splits no matter of gzip format or LZ4 format. Splitting compressed image to compressed splits, have the same behaviour of both splitting uncompressed image to compressed splits and using uncompressed file for

the whole process, that is for fewer memory, Multiple writes is better than Clustered writes.

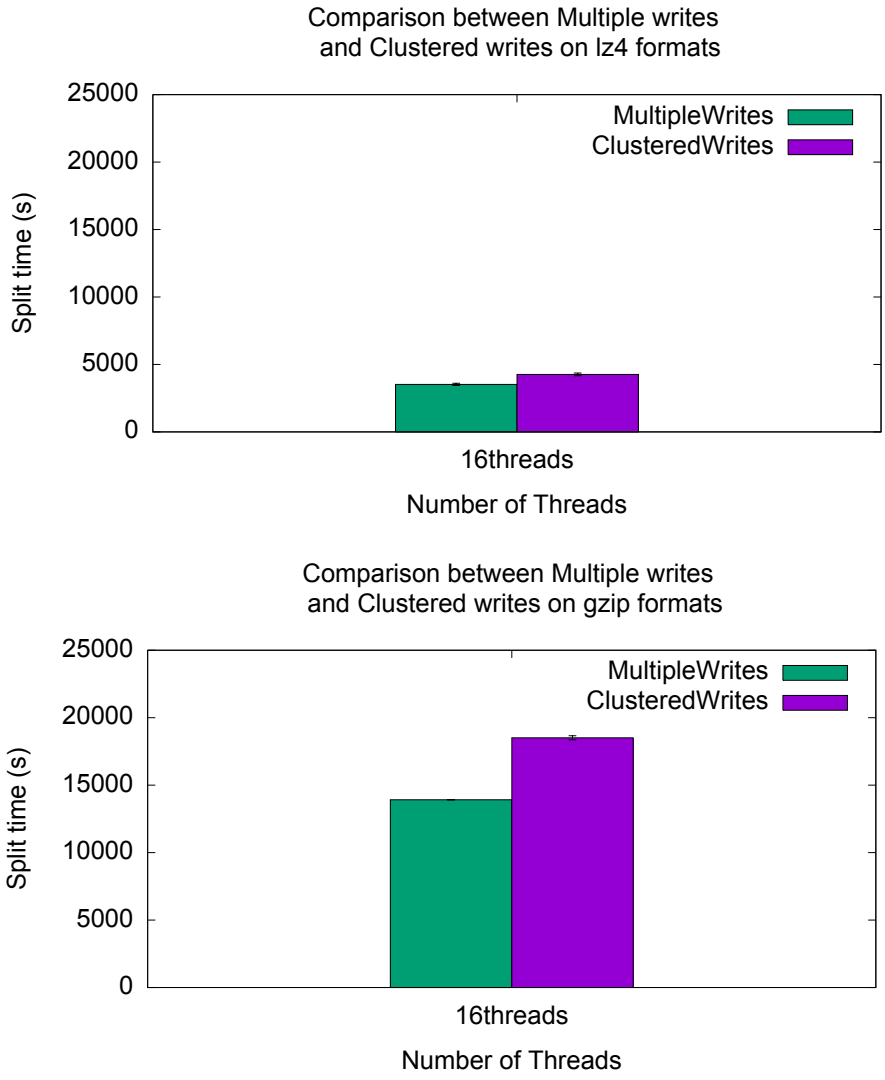

Figure 20: Comparison on splitting compressed image to compressed splits between Multiple writes and Clustered writes. Memory=2 G, using 16 threads. Top: comparison in LZ4 format. Down: comparison in gzip format.

5.6 Conclusion

When we have the input file uncompressed, and we need to write compressed files to the disk, we can use LZ4 file format to compress data on-the-fly, which will improve

our system's performance and save disk space especially for sparse images.

For Multiple writes and Clustered writes, which is better depends on the split size and available memory, compression format will not affect which algorithm is better.

For Clustered reads in merging, we cannot use on-the-fly gzip/LZ4 compression, since Clustered reads in the write phase need to do reverse seeking, and gzip/LZ4 does not support it.

Chapter 6

Conclusion and Future Work

We have proposed a new algorithm for splitting and merging ultra-high resolution 3D images, and compared it to the state-of-the art algorithm in various conditions. We investigated parallelization of the Multiple reads/writes, and made the comparison with Clustered reads/writes. Last but not least, we investigated how on-the-fly compression applies to our algorithms. Table 2 summarizes comparison between the Multiple reads/writes and Clustered reads/writes in all the studied conditions.

We have a clear view on these algorithms and know which one can do better in each circumstance. Next step, we will test our code on the cluster with more nodes and work on converting the outputs to a RDD of Apache Spark [1], which will also be beneficial for the ultra-high resolution 3D images processing pipeline [1]. Furthermore, "re-splitting" 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/writes and Multiple reads/writes will be used as starting points [2]. We will keep studying how the compression works in the MINC file format and compare it with Multiple reads/writes. Last but not least, we will keep working on the library, writing more unit tests and improve the documentation.

[1] `Apache Spark`: https://spark.apache.org/

Table 2: Conclusion

Condition	Best algorithm	Data	Explanation
A few large blocks	Multiple reads/writes	Split: Figure 8 Merge: Figure 9	The overhead of seeking in Clustered reads is larger than overhead of opening each block in Multiple reads
A lot of small blocks	Clustered reads/writes	Split: Figure 8 Merge: Figure 9	The overhead of seeking in Clustered reads is smaller than overhead of opening each block in Multiple reads
Few memory	Multiple reads/writes	Split: Figure 8 Merge: Figure 9	Clustered reads needs more seeking when memory is low.
Lot of memory	Either	Split: Figure 8 Merge: Figure 9	Both boil down to the same number of seeks.
Speed up gained for HDFS cluster	Multiple writes gets more speed up	Split: Figure 15 Merge: N/A	In Clustered writes, the number of files in a memory load is limited, so parallel writes are limited since HDFS does not allow concurrent writes to the same file.
gzip compression in HDFS (only available on splitting)	Multiple writes	Split: Figure 18 Merge: N/A	Clustered reads still do random I/Os in the reconstructed image, which is not supported by LZ4/gzip compression.
LZ4 compression in HDFS (only available on splitting)	Multiple writes	Split: Figure 18 Merge: N/A	

N/A: not tested in this thesis.

Bibliography

[1] V. Hayot-Sasson, "Towards easy and efficient processing of ultra-high resolution brain images," August 2017, master thesis. [Online]. Available: https://spectrum.library.concordia.ca/982970/

[2] V. Hayot-Sasson, Y. Gao, Y. Yan, and T. Glatard, "Sequential algorithms to split and merge ultra-high resolution 3d images," to appear in IEEE Big Data 2017, 10 pages.

[3] 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.

[4] R. W. Cox, J. Ashburner, H. Breman, K. Fissell, C. Haselgrove, C. J. Holmes, J. L. Lancaster, D. E. Rex, S. M. Smith, J. B. Woodward *et al.*, "A (sort of) new image data format standard: Nifti-1," *Neuroimage*, vol. 22, p. e1440, 2004.

[5] "Nifti-1 data format," 2017, [Online; accessed 31-October-2017]. [Online]. Available: https://nifti.nimh.nih.gov/nifti-1

[6] 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 multi-modal images," *Frontiers in neuroinformatics*, vol. 10, 2016.

[7] M. Brett, M. Hanke, B. Cipollini, M.-A. Côté, 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," 8 2016. [Online]. Available: https://doi.org/10.5281/zenodo.60808

[8] Wikipedia, "Z-order curve — wikipedia, the free encyclopedia," 2017, [Online; accessed 22-November-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Z-order_curve&oldid=806618525

[9] 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.

[10] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, 2014.

[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] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] "Apache hadoop," 2017, [Online; accessed 30-October-2017]. [Online]. Available: http://hadoop.apache.org/

[18] "HdfsCLI: API and command line interface for HDFS," 2017, [Online; accessed 31-October-2017]. [Online]. Available: https://hdfscli.readthedocs.io/en/latest/index.html

[19] L. P. Deutsch, "Gzip file format specification version 4.3," 1996.

[20] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[21] J. Kivijärvi, T. Ojala, T. Kaukoranta, A. Kuba, L. Nyúl, and O. Nevalainen, "A comparison of lossless compression methods for medical images," *Computerized Medical Imaging and Graphics*, vol. 22, no. 4, pp. 323–339, 1998.

[22] P. Deutsch and J.-L. Gailly, "Zlib compressed data format specification version 3.3," Tech. Rep., 1996.

[23] 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.

[24] "Lz4," 2017, [Online; accessed 31-October-2017]. [Online]. Available: https://lz4.github.io/lz4/

[25] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.