Journal of Big Data

RESEARCH **Open Access**

Performance-efficient distributed transfer and transformation of big spatial histopathology datasets in the cloud

Esma Yildirim*

*Correspondence:
eyildirim@qcc.cuny.edu
Department of Mathematics and Computer Science, Queensborough Community College of CUNY, New York, USA

Abstract
Whole Slide Image (WSI) datasets are giga-pixel resolution, unstructured histopathology datasets that consist of extremely big files (each can be as large as multiple GBs in compressed format). These datasets have utility in a wide range of diagnostic and investigative pathology applications. However, the datasets present unique challenges: The size of the files, propriety data formats, and lack of efficient parallel data access libraries limit the scalability of these applications. Commercial clouds provide dynamic, cost-effective, scalable infrastructure to process these datasets, however, we lack the tools and algorithms that will transfer/transform them onto the cloud seamlessly, providing faster speeds and scalable formats. In this study, we present novel algorithms that transfer these datasets onto the cloud while at the same time transforming them into symmetric scalable formats. Our algorithms use intelligent file size distribution, and pipelining transfer and transformation tasks without introducing extra overhead to the underlying system. The algorithms, tested in the Amazon Web Services (AWS) cloud, outperform the widely used transfer tools and algorithms, and also outperform our previous work. The data access to the transformed datasets provides better performance compared to the related work. The transformed symmetric datasets are fed into three different analytics applications: a distributed implementation of a content-based image retrieval (CBIR) application for prostate carcinoma datasets, a deep convolutional neural network application for classification of breast cancer datasets, and to show that the algorithms can work with any spatial dataset, a Canny Edge Detection application on satellite image datasets. Although different in nature, all of the applications can easily work with our new symmetric data format and performance results show near-linear speed-ups as the number of processors increases.

Keywords: Big data applications, Content-based image retrieval, Cloud networks, Distributed transfer algorithms, Digital microscopy, Whole slide images

Introduction

WSI datasets are very large tissue slide images in multi-giga-pixel resolution, produced by digital scanners and they have utility in a wide range of diagnostic and investigative pathology applications [1]. Running biomedical analytics applications that process big

WSI datasets in the cloud presents unique challenges which can be classified as *dataset challenges*, *cloud architecture challenges*, and *application challenges*.

The first problem with these datasets is that the file size can be quite large. Considering a typical image can be as large as 200,000 × 200,000 pixels, each image file can occupy 30–50 GB space in uncompressed format, which makes it unfeasible to bring into memory as a whole. Bueno et al. [2] used a parallel data access method to bring the images into the memory of a node using an MPI-based approach. In that study, the master process assigns an image id to the worker processes, and the worker processes access the storage system directly. Parallelism is provided for one single WSI image at a time and does not consider large datasets that consist of several WSIs. The data size loaded into the memory depends on the size of the node memory. This limits the scalability of the application to WSI size/memory size without taking into account the characteristics of the dataset and the analytics algorithm. Also, fixing the tile size brought into the memory of the node based on the size of the WSI and memory limit of the node may increase algorithm overhead. Therefore, their performance results show that it can only scale up to 17 cores. Another downside is that the analytics application has to be written in MPI to use this access method.

Another problem with these datasets is that there is no uniform format and each digital scanner brand can produce images in propriety formats. The vast majority of existing commercial scanners utilize different data formats relying on propriety metadata and compression techniques. For example, while Aperio-Leica scanner uses SVS format, Hamamatsu uses VMS or VMU formats [3, 4]. There are only a handful of libraries that can read these images (e.g., Openslide [5], Bio-formats [6]) and they cannot support all the different file formats on their own. Also, they provide limited access capabilities for cloud storage systems (e.g., HDFS [7], AWS S3 [8]). Version 5 of Bio-formats can only work with parallel file systems (e.g., Lustre [9], GPFS [10]) and its pixel data format is not compatible with OpenCV [11] interfaces. Therefore, a generic scalable data format is needed to remove these cross-compatibility issues.

The use of parallel processing algorithms and frameworks with whole slide image datasets is also very limited. Parallel access to the storage system usually is not considered as part of the parallelization process [12, 13]. In Teodoro et al. [12], several image processing algorithms including object segmentation, feature extraction, classification, and tracking of pixel data access patterns are analyzed for CPU, GPU, and MIC processor architectures. However, in this paper, only memory access patterns are considered in their performance analysis, and a master/worker model similar to the work presented in [13] is adopted. Aji et al. [14] provide a Hadoop-based system for spatial image data access. However, the images need to be processed into WKT format which is a text-based format that stores polygon data of the regions of interest. They provide limited scalability results and store processed features rather than raw WSI pixel data, henceforth, limiting the range of analytics applications that can use these datasets. In our previous work [1], we presented a distributed, dynamic asynchronous transfer and transformation algorithm that works on the Hadoop [15] ecosystem and this algorithm was able to scale seamlessly based on the capabilities of the underlying resources and store WSIs in raw format which increases the range of analytics applications that can use this format.

The second set of challenges can result from the architecture of the cloud. Although clouds can provide tremendous opportunities for a plethora of biomedical applications [16–22], black box object storage systems (e.g., AWS S3) and poor cloud networks are the main reasons analytics applications perform poorly [23]. These studies usually deal with smaller size image formats such as MRIs, microarrays, pCTs and do not come across with the problems presented by extremely large WSI datasets. The designed platforms also install their own storage system, thus having full control. In this case, they can easily predict the load of the system and have real-time access to the metadata and logs which are almost impossible with black box storage systems like AWS S3. The third challenge is usually application-specific and also dependent on the data access patterns of the application. Teodoro et al. [12] present that different access patterns of a plethora of image processing operations have a great effect on the performance depending on the architecture used.

In this study, we target these challenges and present novel, distributed algorithms that can pipeline the transfer and transformation tasks of WSI datasets and scale seamlessly. The algorithms transform the unstructured format of raw WSIs to a symmetric binary format providing easy access to a large set of parallel and distributed analytics applications. Previous work could only provide processed feature formats [14, 24] which limit the range of applications that can work with them. The proposed algorithms outperform widely used cloud transfer tools such as *aws s3 cp* and *s3-distcp*, Amazon Elastic Load Balancing service's Round-robin algorithm, and our previous work [1] in an inter-/intra cloud data center setting using AWS S3 storage system and AWS EMR service.

We tested these symmetric binary datasets using three different case studies. The first study is a content-based image retrieval application [25] for searching cancerous patterns in a prostate carcinoma WSI dataset. The application is implemented on the Hadoop MapReduce framework using OpenCV library. The second study is a deep learning classification application that classifies regions as cancerous or not in a breast cancer WSI dataset. The application is implemented using Keras deep learning library on Tensorflow. In the third study, to show that the proposed algorithms will work with any spatial dataset, we targeted big satellite image datasets. We implemented a distributed edge detection algorithm using Spark [26] and ImageJ [27] libraries. The performance results for the applications provided near-linear speedups as the number of vcpus was increased.

Overall, the contribution of this study includes:

- Two novel distributed data transfer and transformation algorithms/tools for converting WSI datasets into symmetric formats in the cloud.
- Access libraries to transformed datasets for implementing highly scalable analytics applications on Hadoop, Spark, and Tensorflow.
- A distributed implementation of a Content-based Image Retrieval application with Hadoop MapReduce using prostate carcinoma WSI datasets
- A distributed implementation of Deep Neural Network application for classification of images as tumor and normal with Tensorflow/Keras using breast cancer WSI datasets.

- A distributed implementation of Canny Edge Detection Algorithm with Spark and ImageJ using satellite image datasets.

In the following subsections, we explain the system and algorithm design (Sect. "Methodology: system and algorithm design"), case study applications (Sect. "Methodology: case studies"), give an extensive experimental study (Sect. "Experimental results"), and finally end with conclusions and future work (Sect. "Conclusion and future work").

Methodology: system and algorithm design

In this section, the architecture of the underlying system is presented and three different algorithms are explained in detail: *Dynamic Asynchronous Scheduler* [1], *Greedy Scheduler*, and *Pipelined Greedy Scheduler*.

Utilization of native YARN applications on Hadoop ecosystem

The algorithms are implemented as a native YARN application. YARN is the scheduler of the Hadoop ecosystem. Figure 1 presents the interactions between the Application Master task and transfer/transformation tasks. The image access library (e.g., Openslide) does not know how to interact with a distributed file system (e.g., HDFS) or an object storage system (e.g., S3). Therefore the transformation task where the WSI files are converted to symmetric binary files happens in the local disk of the transformation nodes. Most of the algorithm logic is implemented to the *Application Master*. The list of URLs of the WSI datasets is given to the *Application Master* in the form of a text file. The *Application Master* distributes these URLs to the transformation nodes each of which communicates with the source storage system to transfer the files into their local disks and starts the transformation process. Once the files are transformed they are transferred back to the destination storage system. Usually, the destination storage system resides in the same region as the processing cluster. The transformation nodes can recognize different types of source and destination storage systems such as web servers, HDFS, and S3 and use the proper protocol to transfer these files from/to the storage system.

Distributed transfer/transformation algorithms

Three algorithms are presented in this section. *Dynamic Asynchronous Scheduler* is the algorithm developed in our previous work [1] and used as a baseline algorithm here. *Greedy Scheduler* and *Pipelined Greedy Scheduler* algorithms are novel algorithms that use intelligent file distribution strategy and pipelining.

Dynamic asynchronous scheduler

This baseline algorithm assigns one transfer/transformation task to each transformation container. The containers are launched by the YARN scheduler and as soon as one container finishes with its job another is launched. The number of parallel containers depends on the capabilities of the computer node and Hadoop configuration parameters. The outline of the algorithm is given in Algorithm 1.

The algorithm takes the input text file which contains the URLs for the WSI images, the resolution level, and tile size as input parameters. WSI images can have multiple

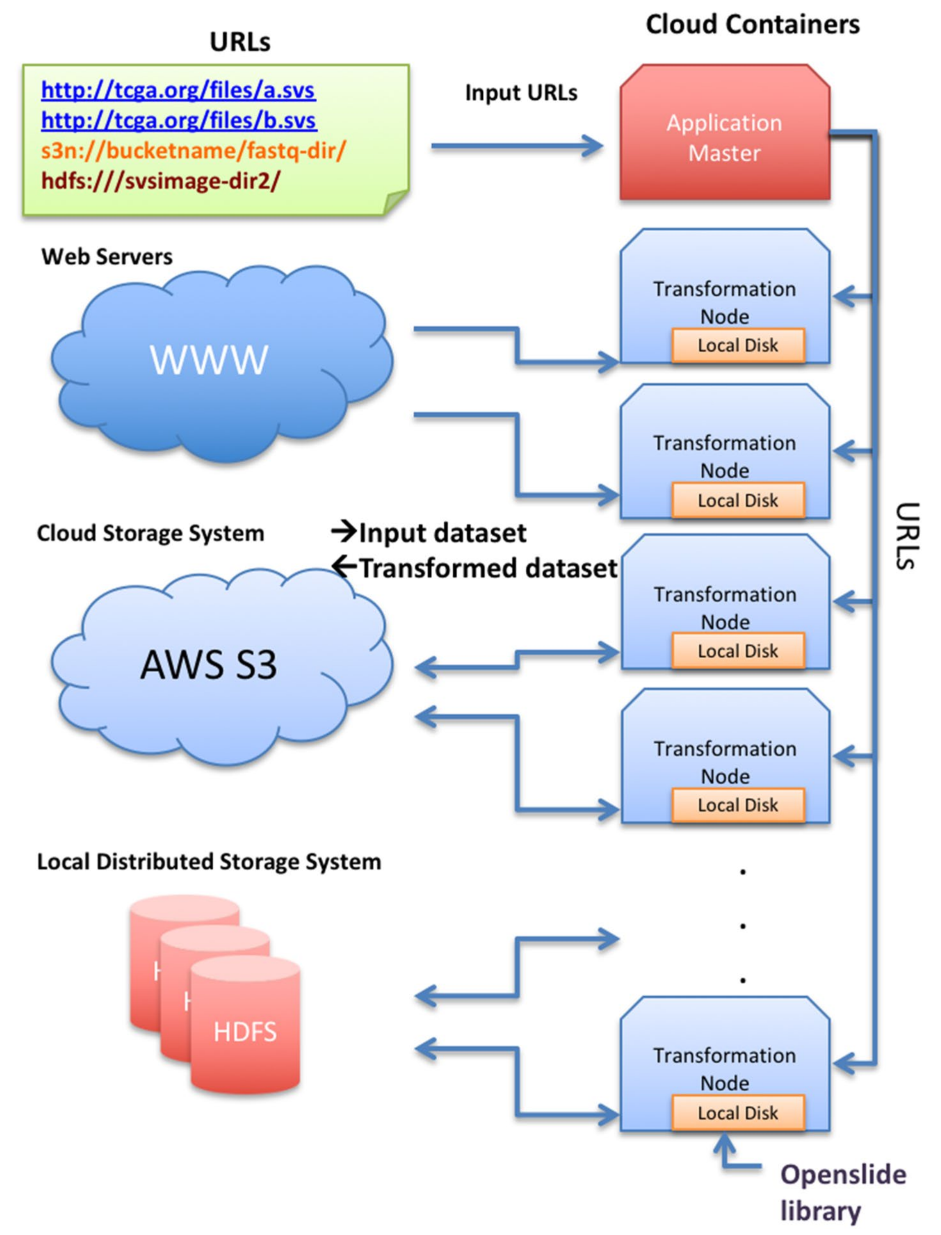


Fig. 1 System architecture

resolution levels. The *Application Master* container goes over the URLs in the text file and recognizes URL types. Then by using the appropriate protocols for each URL type, it interrogates the storage systems to make a global list of image files (Line 2). For each file URL in the list, the application master launches transformation containers that are responsible for the transfer, transformation and storage of the files on the local HDFS system or back on the AWS S3 storage system (Lines 3–5). A simple scheduling algorithm is applied in which each container is assigned a single URL to process.

The application running on the transformation container recognizes the URL type and uses appropriate transfer protocol or API to transfer the file into its local disk (Lines 7–8). It then uses the image access library (Openslide or bio-format) to read the metadata of the file and gathers a list of coordinates based on the resolution level and tile size parameters(Lines 9, 10). For each tile coordinate, it creates a key that consists of the WSI file name, x and y coordinates, and the size of the tile. The metadata about each tile is stored in the key because, in this way, it is easier to implement specific Hadoop *partitioner* and *groupby* classes that can also be used to implement select-where clauses or join objects. This property allows various analytics applications and different access patterns to be possible with sequence files.

The transformation container application then reads the raw partial pixel data in the form of a tile, converts it into a value object, and writes the key-value pair into HDFS or S3 as part of a binary sequence file (Lines 11–15). Sequence files are compressed binary object files that can be split by programming paradigms like Hadoop MapReduce and Spark automatically.

Algorithm 1 Dynamic_Asynchronous_Scheduler

```
Require: $P_{WSI}$ : $Input\ path\ of\ text\ file\ containing\ WSI\ URLs \lor T_w$ : $Tile\ width \lor$
    $T_h$ : $Tile\ height \lor R$ : $Resolution\ level$
 1: $\underline{Application\ Master}$ :
 2: $L_{URL} \leftarrow$ list_of_file_URLs($P_{WSI}$)
 3: for all $URL_i\ in\ L_{URL}$ do
 4:     launch_transformation_container($URL_i, R, T_w, T_h$)
 5: end for
 6: $\underline{Transformation\ Container}$ :
 7: $URL\_type\_x \leftarrow$ recognize_URL_type($URL_i$)
 8: $local\_file\_path \leftarrow$ transfer_to_local_disk($URL\_type\_x, URL_i$)
 9: $Width, Height \leftarrow$ retrieve_size_of_WSI($local\_file\_path$)
10: $List_{tiles} \leftarrow$ retrieve_tile_coords($Width, Height, local\_file\_path$)
11: for all $Tile_i\ in\ List_{tiles}$ do
12:     $Key \leftarrow file\_name, x\_coord_i, y\_coord_i, tile\_width, tile\_height$
13:     $Value \leftarrow$ read_tile_pixels_from_WSI($local\_file\_path, x\_coord_i, y\_coord_i$)
14:     write_to_sequence_file($Key, Value, destination\_path$)
15: end for
```

Greedy scheduler

Dynamic Asynchronous Scheduler creates a separate YARN container for each file to be transferred. However, launching/ tearing down a container is a costly operation, and as the number of files in a dataset increases, this cost increases respectively. The idea behind the *Greedy Scheduler* algorithm is to minimize this cost by launching as many parallel containers as the underlying system allows. For example, if the computer cluster has 12 processors, the algorithm then launches only 12 containers. In this case, the distribution of files among the containers becomes very important as we want all the containers to finish their tasks approximately at the same time. Therefore, *Greedy Scheduler* applies a greedy approach to load balance the distribution of files to the transformation containers. The outline of the algorithm is given in Algorithm 2.

After the list of URLs is constructed (Line 2), the files are sorted in descending order of their sizes (Line 3). Then, the transformation container with the list of files of which total size is minimum is found and in every iteration of the loop, the file in line is assigned to the list of that container (Lines 4–7). In doing so, we assign the largest files first to the minimum size list hence maintaining load balance. Then, a transformation container is launched for each list (Lines 8–10). The only difference in the algorithm of Transformation Containers is that now they have a list of files to transfer/transform instead of a single file (Lines 12–22).

In comparison to AWS Elastic Load Balancing (ELB) algorithm [28], our Greedy Scheduler is designed to work with datasets that consist of very large files and can support different URL types. On the other hand, AWS ELB is designed to work with small size HTTP POST and GET requests. Since ELB algorithm has no idea about how many requests will be queued up at any specific time, it uses a simple round-robin scheduler [28]. A round-robin scheduler might be the best choice for similar-size small requests with unknown sizes, however, our Greedy Scheduler interrogates the file and storage systems to get an idea about the size of each transfer/transformation request for the entire dataset. Therefore, it makes a more intelligent decision, starting up with assigning the largest files first to the container of which queue has the minimum total file size assigned. We present the performance results of ELB's round-robin algorithm in Sect. "Experimental results".

Algorithm 2 Greedy_Scheduler

```
Require: $P_{WSI}$ : $Input\ path\ of\ text\ file\ containing\ WSI\ URLs \vee T_w$ : $Tile\ width \vee$
    $T_h$ : $Tile\ height \vee R$ : $Resolution\ level \vee N$ : $Number\ of\ transformation\ containers$
 1: $\underline{Application\ Master}$ :
 2: $L_{URL} \leftarrow$ list_of_file_URLs($P_{WSI}$)
 3: sort_descending_filesize($L_{URL}$)
 4: for all $URL_i\ in\ L_{URL}$ do
 5:     $L_{URL_j} \leftarrow List\ of\ URLs\ of\ container_j\ with\ minimum\ total\ file\ size$
 6:     $L_{URL_j}.add(URL_i)$
 7: end for
 8: for all $L_{URL_j}$ do
 9:     launch_transformation_container($URL_i, R, T_w, T_h$)
10: end for
11: $\underline{Transformation\ Container_j}$ :
12: for all $URL_i\ in\ L_{URL_j}$ do
13:     $URL\_type\_x \leftarrow$ recognize_URL_type($URL_i$)
14:     $local\_file\_path \leftarrow$ transfer_to_local_disk($URL\_type\_x, URL_i$)
15:     $Width, Height \leftarrow$ retrieve_size_of_WSI($local\_file\_path$)
16:     $List_{tiles} \leftarrow$ retrieve_tile_coords($Width, Height, local\_file\_path$)
17:     for all $Tile_i\ in\ List_{tiles}$ do
18:         $Key \leftarrow file\_name, x\_coord_i, y\_coord_i, tile\_width, tile\_height$
19:         $Value \leftarrow$ read_tile_pixels_from_WSI($local\_file\_path, x\_coord_i, y\_coord_i$)
20:         write_to_sequence_file($Key, Value, destination\_path$)
21:     end for
22: end for
```

Pipelined greedy scheduler

Once the file lists are assigned to the containers there is no reason that a transfer task of a file should wait for the transformation task of the previous file in the list. Therefore, in *Pipelined Greedy Scheduler* algorithm, the transformation containers have a multi-threaded producer/consumer architecture. Each container task launches one transfer and one or more transformation threads where the threads share a common URL buffer. Once the transfer task finishes, the local URL path is inserted into the buffer (Lines 14–18). The transformation task picks up a URL from the buffer and starts the transformation (Lines 20–28). In this case, the transfer and transformation tasks are pipelined. The outline of the algorithm is given in Algorithm 3.

The only difference in the *Application master* compared to *Greedy scheduler* is that once the file lists are constructed for the containers, they are sorted in ascending order of their file sizes (Line 9) . The reason for this is that it has been observed that the transformation takes more time than transfer. Therefore, the sooner the transformation task starts, the better for the pipelining overlap. As a result, the algorithm transfers smaller files first.

Algorithm 3 Pipelined_Greedy_Scheduler

```
Require: $P_{WSI}$ : $Input\ path\ of\ text\ file\ containing\ WSI\ URLs \vee T_w$ : $Tile\ width \vee$
    $T_h$ : $Tile\ height \vee R$ : $Resolution\ level \vee N$ : $Number\ of\ transformation\ containers$
 1: $\underline{Application\ Master}$ :
 2: $L_{URL} \leftarrow$ list_of_file_URLs($P_{WSI}$)
 3: sort_descending_filesize($L_{URL}$)
 4: for all $URL_i\ in\ L_{URL}$ do
 5:     $L_{URL_j} \leftarrow List\ of\ URLs\ of\ container_j\ with\ minimum\ total\ file\ size$
 6:     $L_{URL_j}.add(URL_i)$
 7: end for
 8: for all $L_{URL_j}$ do
 9:     sort_ascending_filesize($L_{URL_j}$)
10:     launch_transformation_container($URL_i, R, T_w, T_h$)
11: end for
12: $\underline{Transformation\ Container_j}$ :
13: $\underline{Transfer\ Thread}$ :
14: for all $URL_i\ in\ L_{URL_j}$ do
15:     $URL\_type\_x \leftarrow$ recognize_URL_type($URL_i$)
16:     $local\_file\_path \leftarrow$ transfer_to_local_disk($URL\_type\_x, URL_i$)
17:     $prodcons\_buffer.add(local\_file\_path)$
18: end for
19: $\underline{Transformation\ Thread/s}$ :
20: for all $local\_file\_path_i\ in\ prodcons\_buffer$ do
21:     $Width, Height \leftarrow$ retrieve_size_of_WSI($local\_file\_path_i$)
22:     $List_{tiles} \leftarrow$ retrieve_tile_coords($Width, Height, local\_file\_path_i$)
23:     for all $Tile_j\ in\ List_{tiles}$ do
24:         $Key \leftarrow file\_name, x\_coord_j, y\_coord_j, tile\_width, tile\_height$
25:         $Value \leftarrow$ read_tile_pixels_from_WSI($local\_file\_path_i, x\_coord_j, y\_coord_j$)
26:         write_to_sequence_file($Key, Value, destination\_path$)
27:     end for
28: end for
```

Methodology: case studies

The transformed symmetric datasets can be fed into a variety of big data analysis frameworks such as Hadoop, Spark, and Tensorflow, increasing their scalability levels. In this section, we present three case studies: A CBIR application that searches for cancerous patterns in a prostate carcinoma dataset from the Cancer Genome Atlas database, a deep learning classification application that classifies breast cancer images from the Camelyon Challenge dataset, and a distributed canny edge detection algorithm using Natural Earth Dataset.

Case 1: distributed content-based image retrieval

If the transfer and transformation of the WSI dataset is the first step of a bioinformatics application pipeline, then the second step is the implementation of a scalable CBIR application that can access these datasets. We implemented a distributed content-based image retrieval application for prostate carcinoma WSI datasets from the Cancer Genome Atlas database using a coarse searching algorithm presented in [25].

In Fig. 2, a query patch of glandular prostate carcinoma structure is searched in a tile cropped from a WSI image. In this case, a tile is a value object in our binary data format. A sliding window approach is used to search Region of Interests (ROIs) against the query patch. The size of the query patch defines the size of the sliding window. Both query patch and ROI are divided into rectangular circles and their color histograms are calculated using OpenCV library. Chi-square distance formula is used to calculate the distance between histograms, which provided the best results with our tests. After the distance of each rectangle is calculated, they are averaged. The mapper task reads the transformed binary files in the form of key-value pairs. The key of the map task is a user-defined key class that consists of the file name, x and y coordinates of the tile, and width and height of the tile. The value is the BGR byte representation of the tile pixels. The query patch is provided to the map task using distributed cache feature of Hadoop ecosystem where the patch is shared among all tasks. The mapper applies the searching algorithm and calculates the distances. The output key is the distance and the output value is a user-defined class that consists of the file name, x and y coordinates of the tile, width and height of the tile, x and y coordinates of the ROI, and width and height of the ROI. Making the distance the output key of the mapper tasks allows the reducer task to automatically sort the results based on the distance. Once the results are sorted we can pick the top best ROIs.

Case 2: distributed classification via deep neural networks

The second application is a deep convolutional neural network for the classification of breast cancer images from the Camelyon Dataset. The dataset consists of tumor WSIs, tumor mask WSIs, and normal WSIs (Fig. 3). Our transfer/transformation algorithms can add a label (0 for normal, 1 for tumor) to the key as these datasets are converted into symmetric binary format. For tumor tiles, if the tile tumor mask

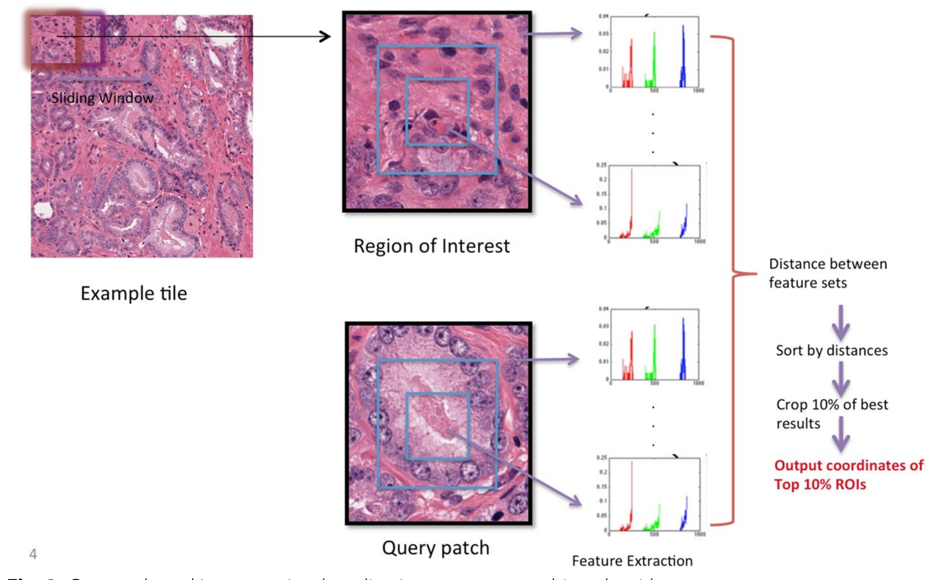


Fig. 2 Content-based image retrieval application—coarse searching algorithm

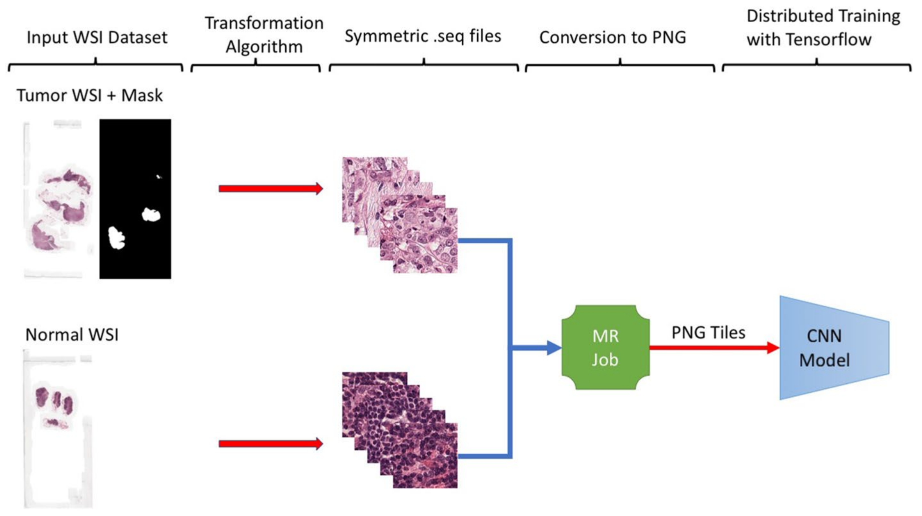


Fig. 3 Classification application pipeline

is more than 50% white, we assign the label as *tumor*. For normal WSI images, a different thresholding method is applied. If the average of the R, G, and B pixels is less than 150 and the percentage of such pixels in the tile is greater than 80%, we assign the label as *normal*. These values can be changed based on the background color of the WSI image.

Although Hadoop and Spark applications can access the transformed datasets directly because of their Java serialization compatibility, Tensorflow applications need an extra step. Therefore, before the training process starts, we add another MapReduce job to the pipeline to transfer the transformed datasets into the local filesystem in the form of PNG patches. Then, our distributed Convolutional Neural Network application starts training. We have not been able to find a TFRecordReader class that can read Java serialized binary sequence files. In our future work, we plan to bypass Java serialization and write our own serializer for Hadoop and a TFRecordReader class in C++ for Tensorflow so the incompatibility issue is solved.

The CancerNet model by Adrian Rosebrock [29] is used for our classification application. The model uses exclusively 3×3 convolution filters, stacks multiple 3×3 convolution filters on top of each other before performing max-pooling, and uses depth-wise separable convolution rather than standard convolution layers. Their model used 48×48 patches from Kaggle Breast Histopathology dataset [30]. Since we can have any size patches through our transformation algorithms on the fly, we used a larger patch size 128×128 while creating the symmetric datasets. We also enabled distributed training by using Tensorflow's MultiWorkerMirroredStrategy. Therefore, we increased the batch size as well from 32 to $32 \times$ #vcpus initially. The dataset was divided into training, validation, and testing datasets. 80% of the dataset was used as the training set while the remaining %20 was used as the testing set. 10% of the 20% test dataset was also set aside as the validation set. The application scaled well as we increased the number of vcpus and provided very high accuracy results (see Sect. "Accuracy and performance results of classication application").

Case 3: distributed Canny edge detection

In the third case study, we tested our algorithms with a different type of spatial dataset. We selected the Natural Earth Dataset [31] which consists of very large (21,589 × 10,762 pixels each) satellite images of the world map. The original image file format is TIFF and because it is not a microscopy dataset, Openslide library cannot be used to transform them. To achieve the transformation of these datasets, we changed the library used in Fig. 1 from Openslide to Java ImageIO. This way, any dataset can be used as input to our algorithms simply by extending the Java class TileReader.java and providing an implementation for the abstract method readTile() which takes x and y coordinates, tile width, and tile height as input parameters. The transformed datasets are then fed into a Spark Job for applying a Canny Edge Detection algorithm using ImageJ library (Fig. 4).

Edge information is essential in many image analysis and computer vision applications. The generic methods that detect edges are usually designed for monochromatic images and focus on the intensity calculations. However, a superior method is needed for color images that will treat color RGB channels as vectors but not scalars. Canny edge detection algorithm [32, 33] designed for color images has proven to provide better results than monochromatic image edge detection techniques. The algorithm starts by smoothing the RGB channels using a Gaussian filter. Then, the color magnitude is calculated as the squared local contrast obtained from the dominant eigenvalue of the structure matrix. From this matrix, the local gradient vector is calculated. That is followed by non-maximum suppression, edge tracing, and thresholding calculations. The complete algorithm can be found at [33]. From the edge tracing results, we write the tracing coordinates as the output of the Spark job. Accuracy and performance results of the algorithm are provided in Sect. "Accuracy and performance results of Canny edge detection application".

Experimental results

In this section, we present an extensive experimental study showing the performance and accuracy of our algorithms and case study applications. AWS S3 storage system and AWS EMR service are used for the experiments.Our algorithms will work on any system with a Hadoop installation and we support different URL types (s3://, hdfs://, http://). So data can reside on AWS S3, a local or remote cluster's HDFS system or it can even be on a web server with a public http:// URL. Also developers extending the class *GenericURL.*

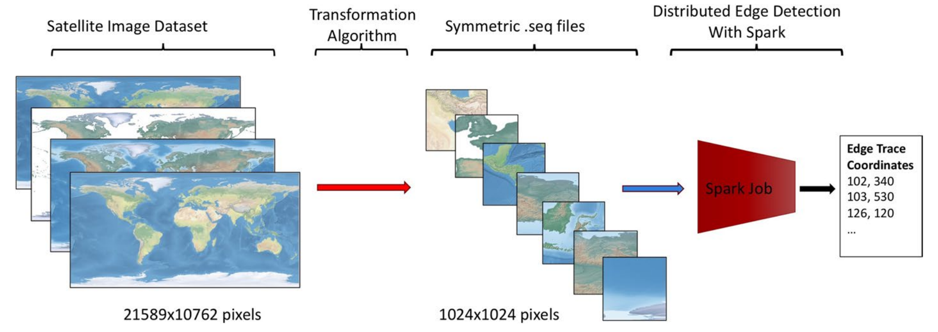


Fig. 4 Distributed Edge Detection Pipeline

java can provide their storage system and transfer protocol. AWS provides a scalable infrastructure and comes with a readily installed Hadoop system. Therefore, we chose AWS as our testing environment.

4 different storage locations are selected for the datasets: Virginia, Oregon, Canada, and Frankfurt. There are two reasons the algorithms were tested in different regions. The first reason is that we wanted to see how they will perform in different RTT and traffic pattern situations. Virginia data center is an example of an intra-data center transfer/transformation case with very small RTT (less than 1 ms) and busy traffic patterns. Oregon data center has busy traffic patterns and a large RTT (50 ms) representing wide-area data transfers. Canada data center represents short RTTs (15 ms) and not busy traffic patterns. And finally, Frankfurt data center represents a transoceanic transfer/transformation with very large RTTs (80 ms) and not-so-busy traffic patterns. The four data centers in general represent 4 different combinations of RTT and traffic patterns. The second benefit to test different data centers is that medical centers may be geographically distributed but can form collaborations. In this case, different centers would want to store their data to the data center closest to them but want to collaborate and share their data. Hence, algorithms were designed to do transfer/transformations from multiple data sources.

AWS EMR comes with a readily installed Hadoop system. For EMR clusters, memory optimized instances are selected since these applications require large memory space. The node types we used range from 4-vcpu (m5.xlarge) to 32-vcpu (m5.8xlarge) instances. However, for the classification application, only c3 type instances were available in AWS EMR with Tensorflow deployment. Therefore, we used c3.2xlarge, c3.4xlarge, and c3.8xlarge instances in the experiments.

Comparison of total transfer time

In this experiment, the transfer time of the dataset from the source S3 storage system to the local file system of the EMR cluster is measured. The transformation feature of the algorithms was disabled just to see the total transfer time. We compared Dynamic Asynchronous Scheduler (DAS) and Greedy Scheduler (GS) algorithms against widely used transfer tools in the cloud: *aws s3 cp* and *s3-dist-cp*. The reason why we only measured the transfer time is that these tools are not capable of transforming the datasets. *aws s3 cp* is the default transfer tool of the AWS client interface and is a multi-threaded application. *s3-dist-cp* is a Mapreduce application that distributes the transfer of files into reducer tasks.

In Fig. 5, we transferred a 100 WSI dataset (approximately 19 GB) using m5.xlarge instance clusters ranging the vcpu number between 4 and 32. The results show similar characteristics for all source regions regardless of their different RTTs. Although *aws s3 cp* outperforms the rest of the tools/algorithms for 4 vcpu results, it cannot scale beyond the limits of a single instance due to its shared memory architecture. On the other hand, a multi-threaded architecture has less overhead compared to a multi-container architecture. We believe its superior performance at 4 vcpus is because of its threaded implementation providing less overhead.

As we mentioned in Sect. "Methodology: system and algorithm design", GS algorithm is optimized compared to DAS algorithm because it uses fewer containers, this

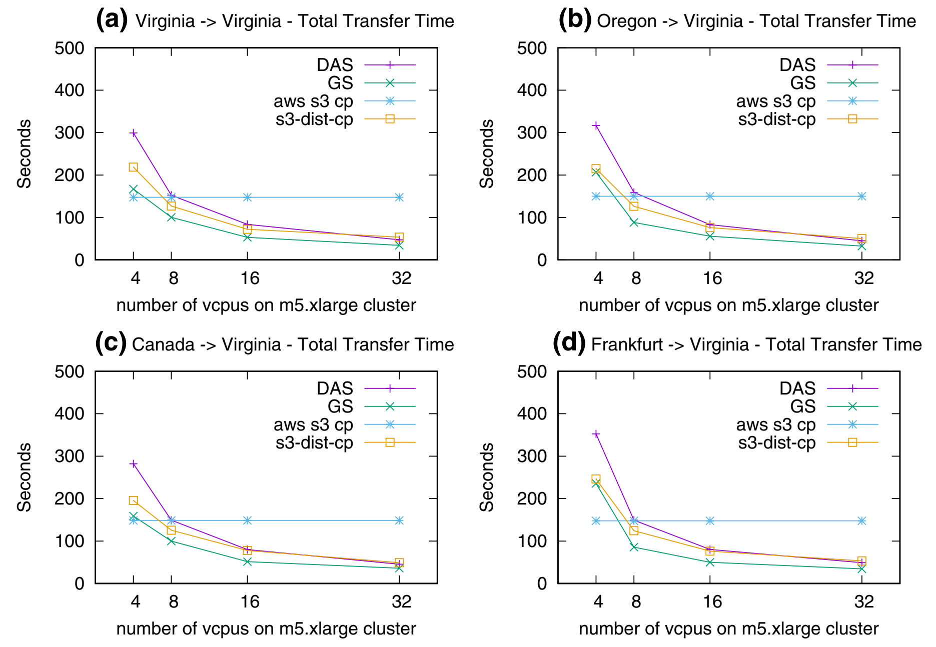


Fig. 5 Performance comparison of total transfer time

statement also applies to *s3-dist-cp* as it uses as many reducer containers as the underlying Hadoop configuration allows. In all of the cases, GS algorithm outperforms *s3-dist-cp* and when the parallelism level is high DAS algorithm slightly outperforms *s3-dist-cp* as well.

When we look at the total transfer time in different regions Canada has the best performance although its RTT ($\approx 15ms$) is higher than Virginia (less than $\approx 1ms$). We believe, the intra-network traffic of the Virginia data center is much higher than the inter network traffic between Canada and Virginia. This indicates that RTT is a less effective factor in transfer time compared to the network traffic. Similarly, the total time of Oregon-Virginia (RTT $\approx 50ms$) transfers is very close to Frankfurt-Virginia (RTT $\approx 80ms$) transfers proving that traffic is a much more definitive factor than RTT when it comes to cloud networks.

PPGS algorithm is not included in the results because it only differs from GS when the transformation is enabled.

Comparison of total transfer/transformation time

In the second experiment, we transferred a 50 WSI dataset (approximately 15 GB in compressed format). The replicas of the dataset are again placed in Virginia, Oregon, Frankfurt, and Canada data centers. An EMR cluster of m5.4xlarge nodes is created in the Virginia data center and the transfer/transformation algorithms were tested on this cluster. The destination is the S3 system of the Virginia data center.

Figure 6 presents the total time it takes to convert WSI files into the symmetric binary format as we increase the number of vcpus in the cluster between the range [8–32]. In the figure, DAS stands for *Dynamic Asynchronous Scheduler*, RR stands for *Round-Robin* Scheduler, GS stands for *Greedy Scheduler* and PPGS stands for *Pipelined Greedy*

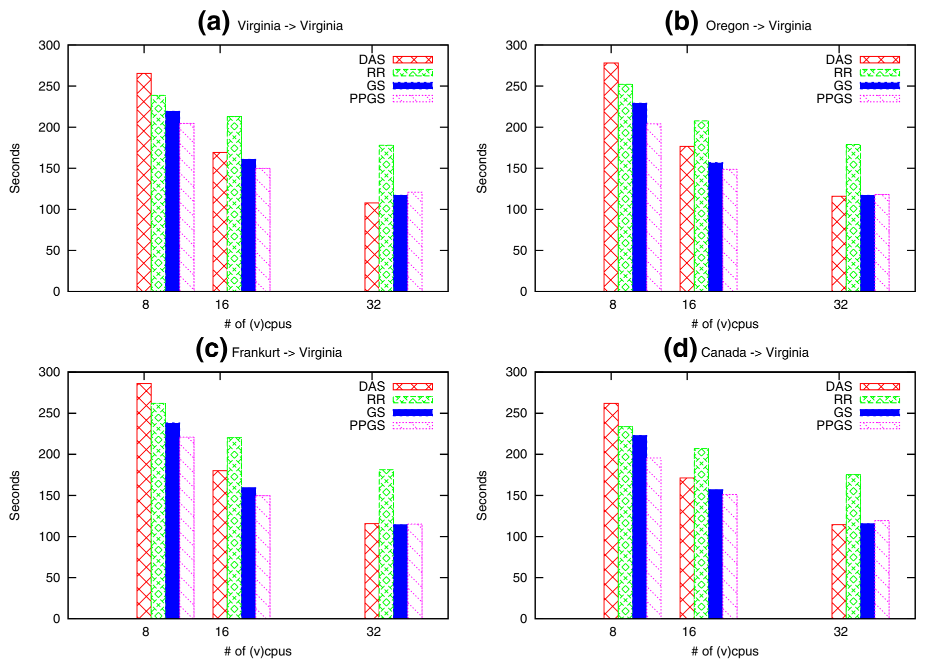


Fig. 6 Performance comparison of transfer/transformation algorithms on AWS S3 and EMR

Scheduler. As the number of vcpus is increased, the number of transformation containers launched increases for RR, GS, and PPGS algorithms as well. As a result, the number of files per container dramatically decreases, hence the optimizations we do in these algorithms lose their effect. That's why for the 32-vcpu case, all the algorithms perform similarly except for the RR algorithm. We observe that RR algorithm performs better than DAS when the number of vpus is 8. However, as we increase the number of vcpus, it does not scale well and falls behind all the other algorithms. For all the other cases, GS performs better than DAS and RR, and PPGS performs better than GS algorithm. Also, as we increase the number of vcpus, the total transfer/transformation time decreases for all cases. The same observation that there was not much difference between the total times for different data centers were made in total transfer/transformation times as well. Then again, the highest total time value belongs to Frankfurt-Virginia transfers as they are transoceanic transfers. On the other hand, Virginia-Virginia and Canada-Virginia transfers almost took the same amount of time. Based on these results, it is clear that our new algorithms (GS and PPGS) outperform the baseline algorithm (DAS) and AWS ELB's round-robin algorithm, and they scale well with the number of vcpus.

Effect of pipelining transfer and transformation tasks

Although PPGS algorithm outperforms both GS and DAS algorithms, we expected its performance to be better. To analyze the results, we took a closer look into four sample transfers between Canada and Virginia data centers on an 8-vcpu setting.

In Fig. 7, we present a time series graphic for DAS algorithm job execution times. Since we did not have any control over which job/task was assigned to which vcpu, the figure presents each file transfer separately through their job ids. It is clear that initially 8

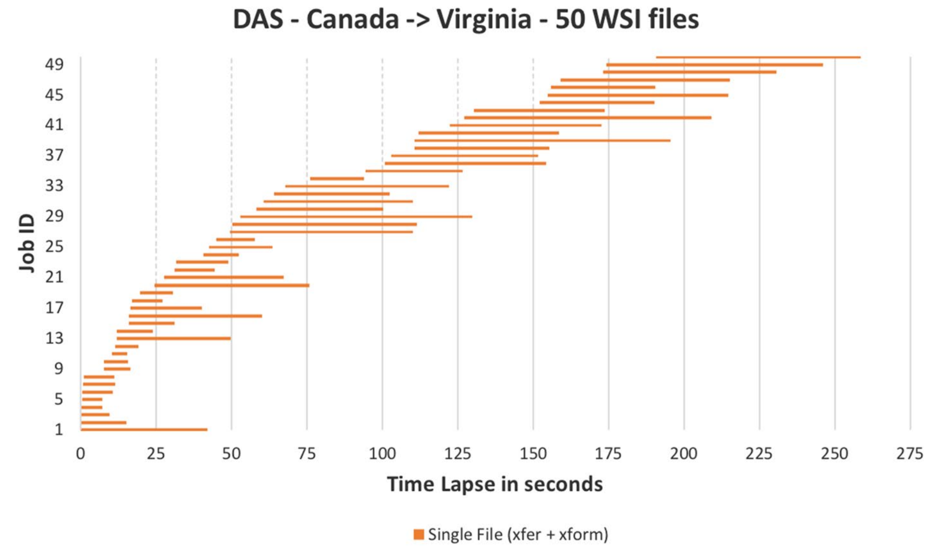


Fig. 7 DAS

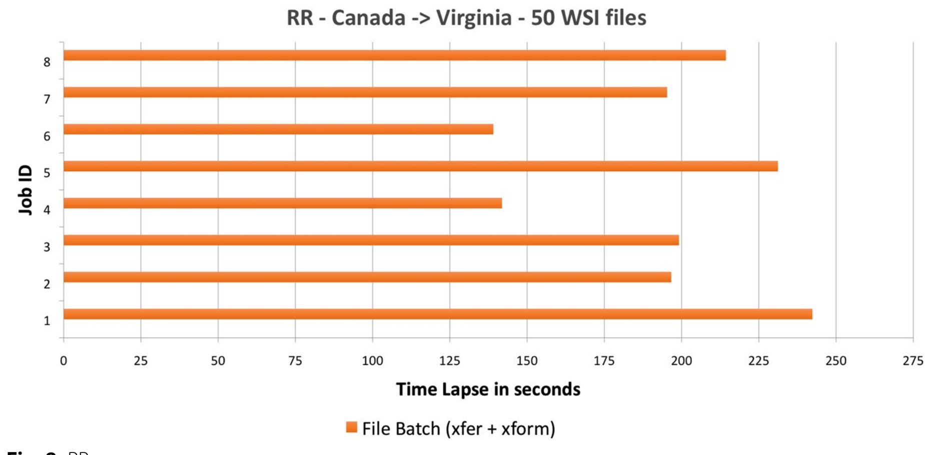


Fig. 8 RR

jobs are started about the same time and as soon as one finishes another one is launched. In total 50 containers (corresponding to jobs) were launched.

In Fig. 8, we present a time series graphic for AWS ELB's RR algorithm job execution times. Eight jobs were run, each of which transferred/transformed a batch of files. By looking at the figure, we can see that the total execution times of the jobs were unbalanced. Although it performs better than DAS algorithm due to the decrease in container create/destroy overhead, it cannot load balance the batch sizes, because each file can be of a different size and the standard deviation of the file sizes could be quite large.

In Fig. 9, we present a time series graphic for GS algorithm job execution times. Eight jobs were run, each one transferring/transforming a batch of files. By looking at the figure, we can see that the total execution times are quite load-balanced due to the intelligent scheduling technique applied based on the file size distribution.

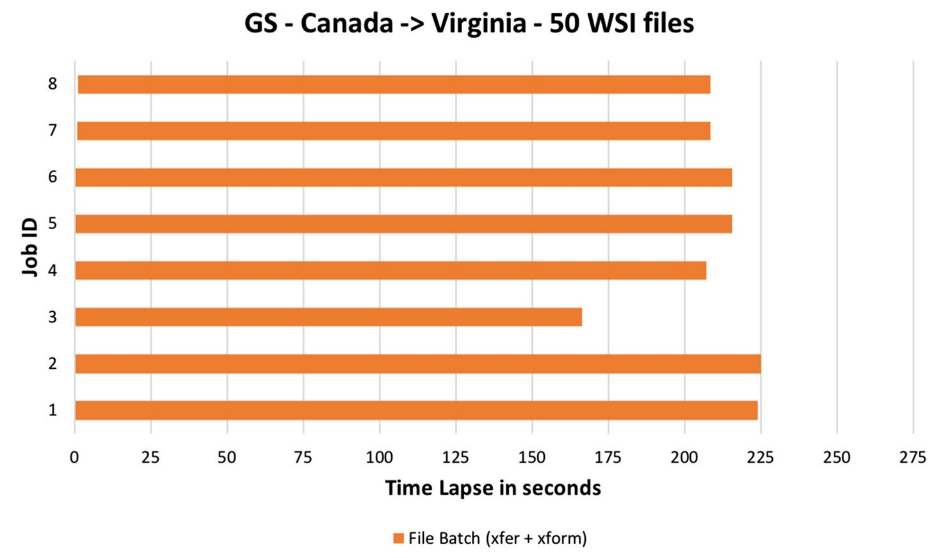


Fig. 9 GS

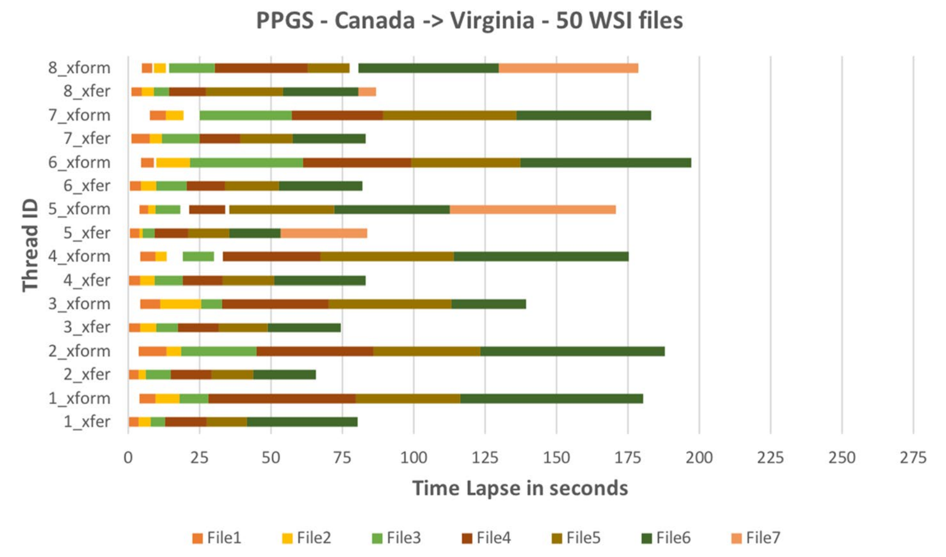


Fig. 10 PPGS

In Fig. 10, we present a time series graphic for PPGS job execution times. Since each job is a 2-thread application the y-axis shows the job id and the thread type. For example, 1_xfer refers to job 1—transfer thread while 1_xform refers to job 1 - transform thread. Between the transfer and transformation threads of a job, the same colors represent the same file transfer and transformation respectively. The first interesting observation was that the transfer and transformation threads overlapped very well. The question is why is NOT total run time much better compared to GS? The second observation was that the only times a transformation thread waited on the transfer thread were the white breaks in the timelines and there were very few of them. So the lack of performance was due to the transformation threads waiting for each other rather than waiting for transfer threads to finish.

With these observations in mind, we decided to do more extensive tests parametrizing the number of parallel transformation threads so that we can create more than one transformation thread per transfer thread.

Figure 11 presents the effect of changing the ratio of #transfer threads to #transformation threads of PPGS algorithm on the total time. We set # job containers to 3 in Fig. 11a and use different instance types, the smallest being m5.xlarge having 4 vpus. The YARN scheduler allocates one container to the application master, therefore for true parallelism, we set the # containers to 3. Similarly, for Fig. 11b parallelism is set to 7 for m5.2xlarge instance (8 vpus) and to 15 for m5.4xlarge instance (16) in Fig. 11c. The first observation made is that as we increase the # of transformation threads total time decreases but eventually becomes stable. The slope of the decrease is higher for more powerful instance types (instances with more vcpus). For example, in Fig. 11a as we increase the # of transformation threads for m5.4x large instance the total time quickly flattens although it improves for PPGS compared to DAS and GS. We looked into CPU utilization and saw that it was maxed out at 1/2 ratio. We increase the ratio only once after this point to prove the flattening of the curve. As the instance type changes, more vcpu time becomes available thus PPGS algorithm performs better. Most of the time, the time flattening occurred due to maxed out CPU utilization except for the case of reaching the network bandwidth limit. This happened only in two cases: first, when #containers = 7 and PPGS-1/3 and then second, when # containers = 15 and PPGS-1/1.

Based on these findings, we decided to design our transfer/ transformation tool to parametrize the #containers and #transformation threads to be set by the user

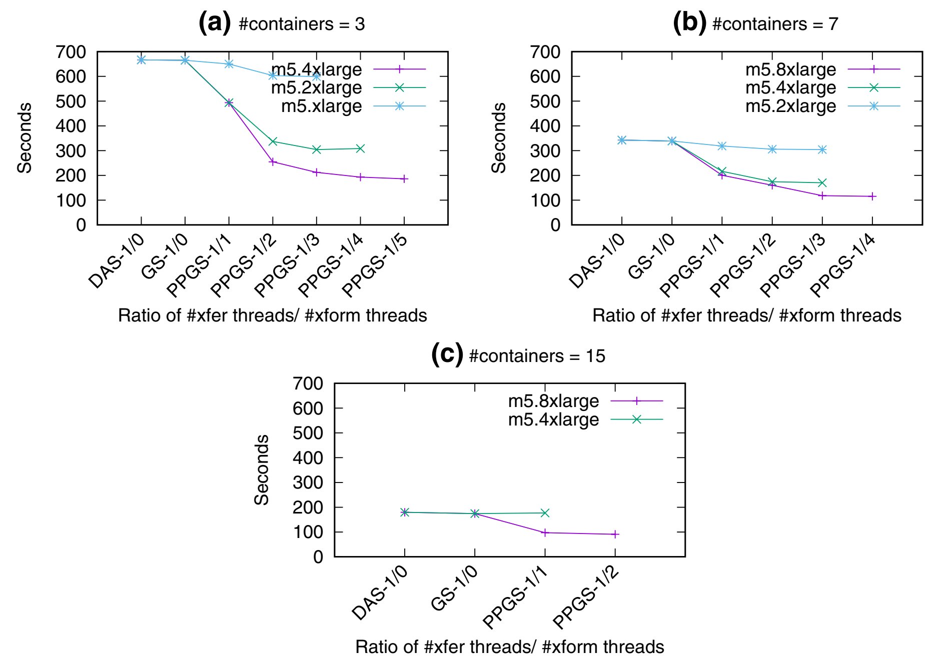


Fig. 11 Effect of ratio of #transfer threads / #transformation threads

considering different instance types and their processing powers. In our future work, we intend to automatically set optimal values for these parameters.

Comparison of data access performances

The goal of our work is similar to the works presented by Bueno et al. [2], Aji et al. [14] and Zerbe et al. [13] which is to provide performance-efficient data access to large spatial datasets. While Aji et al. present distributed data access via Hadoop Framework, their system works with processed polygon data in WKT format. Therefore a fair comparison is not possible. However, Bueno et al. and Zerbe et al. provide performance-efficient algorithms to access pixel data. So we implemented their data access algorithms and compared it against the performance and scalability of data access by a simple MapReduce Job to our transformed sequence file datasets that also contain raw pixel data.

Bueno et al. implement an MPI-based algorithm where each WSI file is divided into equal size blocks and each process accesses the block directly from the file system. If the block size is greater than the memory size of the process, then the process further divides the block into memory-size pieces and retrieves them one at a time. We implemented the same algorithm using MPI.

Zerbe et al. follow a similar Client/Server strategy. Datasets are behind either a WSI streaming server or a file server. They use JPPF framework to launch multiple jobs. Each job retrieves a moderate-size block from the WSI into its memory and then divides it into smaller tiles based on the analytics application's needs. We implemented the same strategy using YARN scheduler instead of JPPF.

We used an AWS EMR cluster that consists of multiple m5.xlarge instances. Then, we created two file systems: a Network File System (NFS) using AWS EFS service and a parallel Lustre File System using AWS FSx Service. We mounted those file systems into the instances of the EMR cluster and tested the data access implementations on the cluster. For our work, we used HDFS and S3 file systems while for Bueno et al. and Zerbe et al.'s work we used the NFS and Lustre File Systems. Their work uses a specific WSI image access library such as Openslide, therefore it is only possible to test their approach on a regular file system. The total read time and speedup results of a 50-WSI dataset are presented in Fig. 12.

We saw that the data access times of Bueno et al. and Zerbe et al. got worse on an NFS file system when we increased the number of vcpus, therefore we did not present it here.

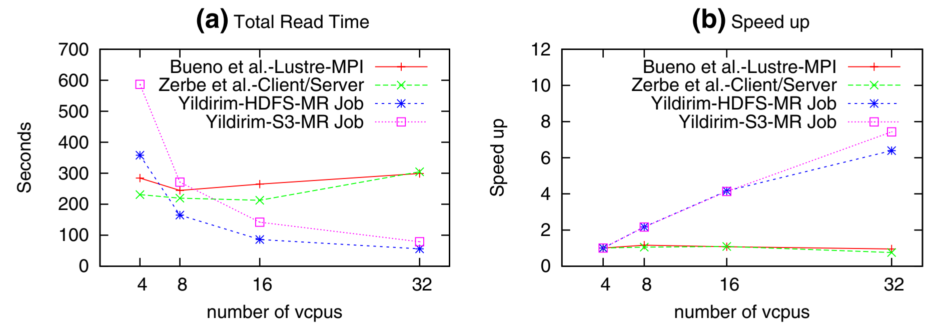


Fig. 12 Performance comparison of data access

Based on their Lustre file system performances, a small decrease in total read time is observed when we increased the number of vcpus to 8 for both Zerbe et al. and Bueno et al.'s algorithms, and a further small decrease is observed when we increased the number of vcpus from 8 to 16 for Zerbe et al.'s algorithm. On the other hand, accessing our transformed symmetric dataset with a MapReduce Job scaled very well. The read performance for 4 vcpus was worse compared to the related work, however as we increased the number of vcpus we saw that total read time decreased dramatically while Bueno et al. and Zerbe et al. results increased. Our method of access provides linear speed-up for the S3 file system and a near-linear speed up for HDFS file system.

We suspect the reason for the worse performances of Bueno et al. and Zerbe et al.'s algorithm is that multiple processes are trying to access the same file at the same time. Even if the file system behind is a parallel file system such as Lustre, it is known that data access does not scale well if you try to access the same file by multiple processes. So the scalability results on their paper actually reflects the scalability of the specific analytics algorithm they implemented but not the data access times. On the other hand, with our approach, the transformed symmetric data files are accessed in a distributed fashion using data locality. Therefore, our approach scales well.

Accuracy and performance results of CBIR application

In this experiment, we tested the CBIR application on a 25-split transformed dataset. A split stands for a block of symmetric data for which a separate mapper/reducer task is created. A 100 × 100 pixel query image that contains a single gland and a 300 × 300 pixel query image that contains clusters of glandular structures were used (Figs. 13 and 14). The application takes two input parameters: *Kbins*, which stands for the number of rectangular circles the ROIs are divided, and *Olap*, which stands for the overlap percentage between the sliding windows. Four different parameter combinations were used in the tests for *Kbins* and *Olap* respectively: 5–0.25, 5–0.5, 10–0.25 and 10–0.5. Figure 13 presents the accuracy results of the top ten ROIs for the specific 100 × 100 pixel query patch based on different parameter settings. In almost all of the cases, similar glandular structures were found. These structures were more similar to the query patch for the top 5 results. It was interesting to see that *Olap* parameter was more effective on the results than *Kbins* parameter. The results (Fig. 14) were better for the 300 × 300 query image where a cluster of glandular structures was searched in the dataset. In all of the cases, the exact query image was found as the top similar result. All the cases presented very

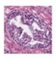
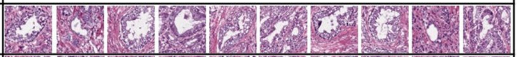
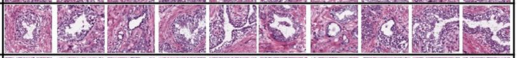
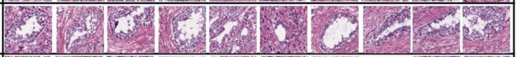
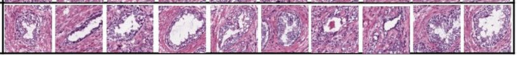
Query Patch (100x100)	Kbins	Olap	Top ROIs
	5	0.25	
	5	0.5	
	10	0.25	
	10	0.5	

Fig. 13 Accuracy results of 100 × 100 query patch—single gland

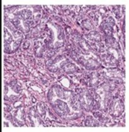
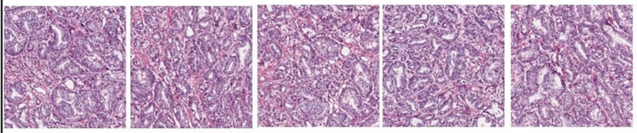
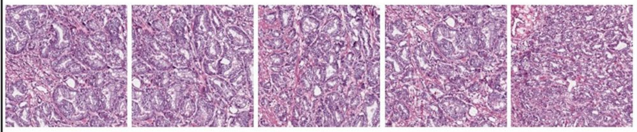
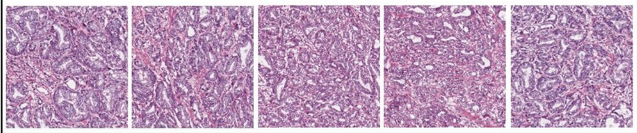
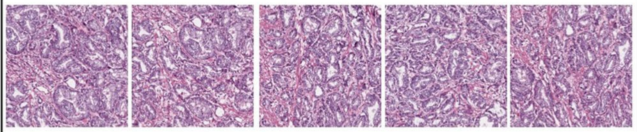
Query Patch (300x300)	Kbins	Olap	Top 5 ROIs
	5	0.25	
	5	0.5	
	10	0.25	
	10	0.5	

Fig. 14 Accuracy results of 300 × 300 query patch—clusters of glands

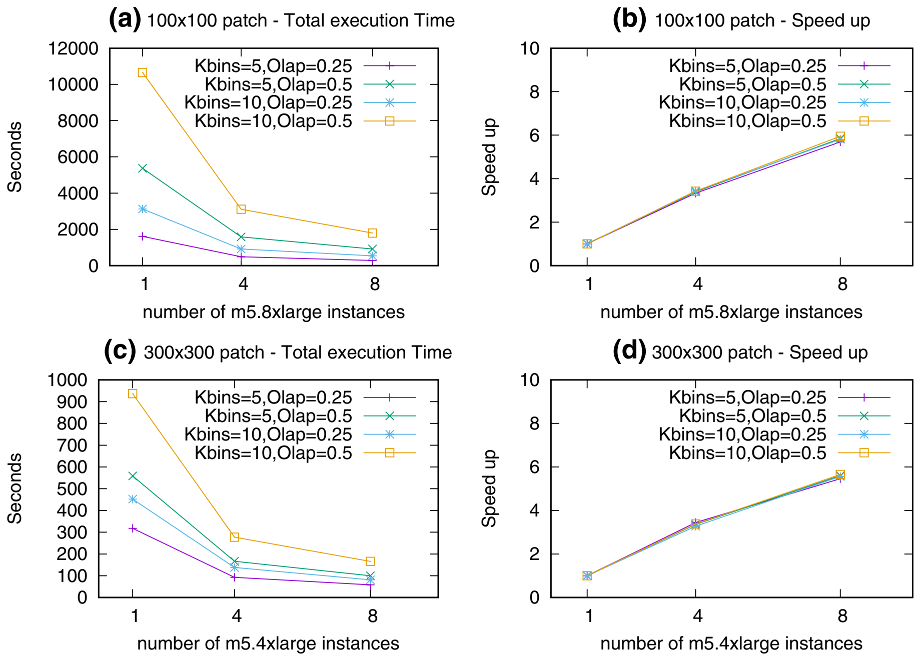


Fig. 15 Performance results of CBIR Application on AWS S3 and EMR

similar, clustered glandular structure images. Again *Olap* parameter was more effective compared to *Kbins*.

Figure 15 presents the performance results of running the CBIR application on an EMR cluster of m5.8xlarge and m5.4xlarge instances as we increase the number of nodes parameter. The CBIR turned out to be a memory-intensive application. For the 100 ×

100 query patch the memory of an m5.8xlarge instance was sufficient to process a split of the dataset without giving errors. For the 300 × 300 query patch, the memory of an m5.4xlarge instance was sufficient. As we increased the number of instances for both cases between the range [1–8], the total execution time decreased dramatically (Fig. 15a, c). Increasing *Kbins* from 5 to 10 also increased the execution time as well as increasing *Olap* from 0.25 to 0.5. It took more time to search for the 100 × 100 query patch than the 300 × 300 query patch. We also calculated the speed-up values by dividing the total execution time with 1 instance to total execution time with n instances. Their corresponding speed-up graphics (Fig. 15b, d) show a near-linear increase as we increase the number of instances indicating that trend might go on for a larger number of instances. These results prove that the CBIR application works well with symmetric binary datasets in terms of accuracy and scalable execution time.

Accuracy and performance results of classification application

We tested the second application with a WSI dataset of 21 (10 tumor + 11 normal) breast cancer images from the Camelyon Dataset. The transformed files consist of 61 symmetric splits. The initial batch size was set to 32 × *#vcpus*. The #vcpus were ranged between 8-32 on c3 types of EMR instances. The #epochs were set to 20. Figure 16 shows the comparison of training accuracy and loss to validation accuracy and loss, as well as testing accuracy, sensitivity, and specificity.

Initially, we increased the total batch size in proportion to the number of vcpus but observed that this approach only lengthened the time it took for the validation accuracy and loss to converge with the training accuracy and loss. Another interesting observation was that if we keep the batch size fixed but increase the number of vcpus, the optimal number of epochs remains the same (6 epochs).

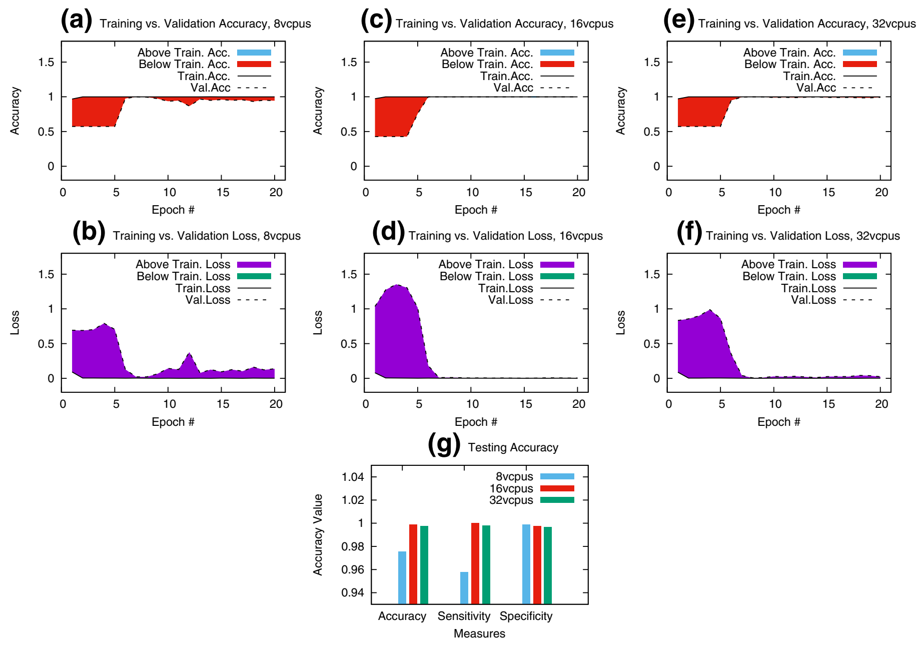


Fig. 16 Accuracy results of the classification application

We also got 97–99% accuracy, sensitivity, and specificity on the testing dataset results for different parallelism levels (Fig. 16). These results are much higher compared to the original CancerNet results on Kaggle dataset (83% accuracy).

The performance results of the experiments are presented in Fig. 17. Per epoch time decreased significantly as we increased the number of vpus (Fig. 17a). Increasing the batch size as we increase the number of vcpus did not affect the per epoch time, however it caused the optimal number of epochs for the validation accuracy and loss to converge into training accuracy and loss to increase. Therefore, we kept the batch size fixed and did not show the results of differentiating batch sizes in this paper.

The conversion time of .seq files to png patches also decreased significantly as we increased the number of vpcus (Fig. 17b).

All of the cases took 6 epochs for the validation accuracy to reach at least 95% accuracy. Therefore, we set this number as the optimal number of epochs and calculated the optimal training time by summing the per epoch time of the first 6 epochs. Although the time decrease slowed between 16 and 32, still there was a significant decrease as we increased the vcpu number (Fig. 17c).

The last measure presented is the speed up, which is calculated as dividing the optimal training time of the base case (8vcpus) by the optimal training time of n vcpus (Fig. 17d). The results showed a near-linear increase as we increased the number of vcpus.

Accuracy and performance results of Canny edge detection application

The third application was tested using the Natural Earth Dataset. We downloaded 4 different versions of the world map of which the total size is 2.8 GBs. After the dataset was transformed it only occupied 937 MB if the tile size is 1024 × 1024 and 1014 MB if the tile size is 4096 × 4096. The transformed datasets occupied less space than the

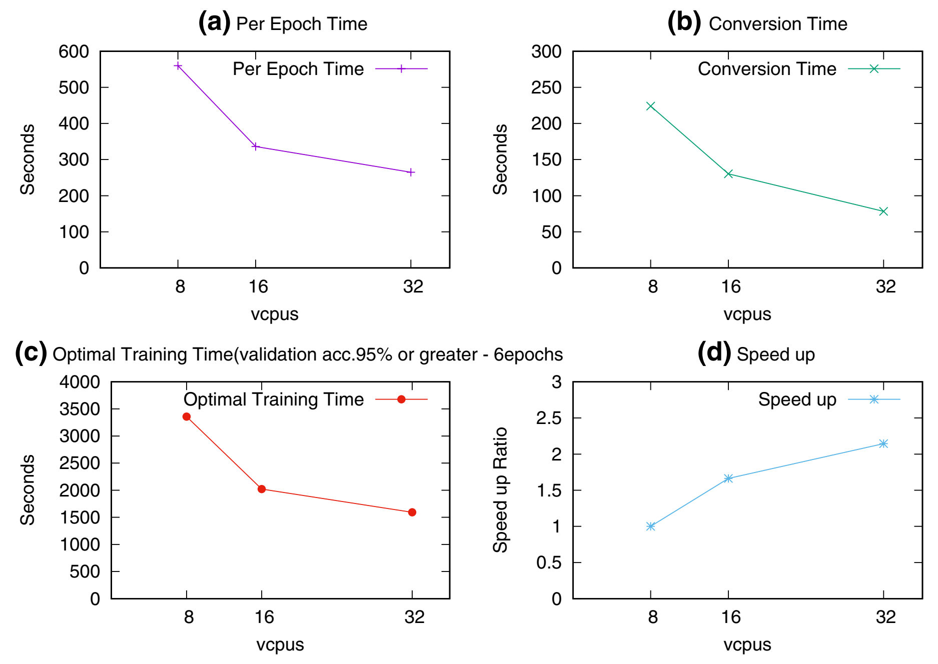


Fig. 17 Performance results of the classification application

original dataset due to the additional use of the Bzip compression algorithm during the transformation process. The Canny Edge Detection algorithm was implemented using Spark and ImageJ libraries. We selected a different distributed processing platform (Hadoop MapReduce, Tensorflow, and Spark) for each case study application to show that the transformed datasets can easily be used with different platforms.

The Canny Edge Detection algorithm calculated the edge magnitude, orientation, and binary map of the source image tiles. We present the results of these output values in Fig. 18. We selected tiles that contained only ocean, land, ice, and a combination of them to see how accurate the edge detection algorithm was. For the *ocean* tile, the edge magnitude results show a wavy behavior of similar strength, indicating the absence of sharp edges. Therefore, only very few edge lines are presented in the binary map. For the case of the *land* tile, clear magnitude differences are observed between the areas of forests and deserts, hence the edges between them are also clear in the binary map. There is a strong and clear separation of ice from the ocean and also different shades of ice in the *ice* tile edge magnitude results and we see this separation in the binary map. And finally, when we have a combination of oceans, mountains/deserts, and forests, the separation is more clear between the ocean and the forests than the mountains/deserts and forests. In the final column, we draw the binary map contours onto the source image tile in black.

To test the scalability of the application we used an EMR Spark Cluster of m3.xlarge instances ranging the vcpu number between 4 and 32. Figure 19 presents total execution time and speed-up results for two different tile sizes: 1024 × 1024 and 4096 × 4096. For both tile sizes the total execution time decreases as we increase the number of vcpus. However, 4096 × 4096 tile size performs better than 1024 × 1024 tile size. The speed-up is calculated again by dividing the execution time of 4vcpus to the other

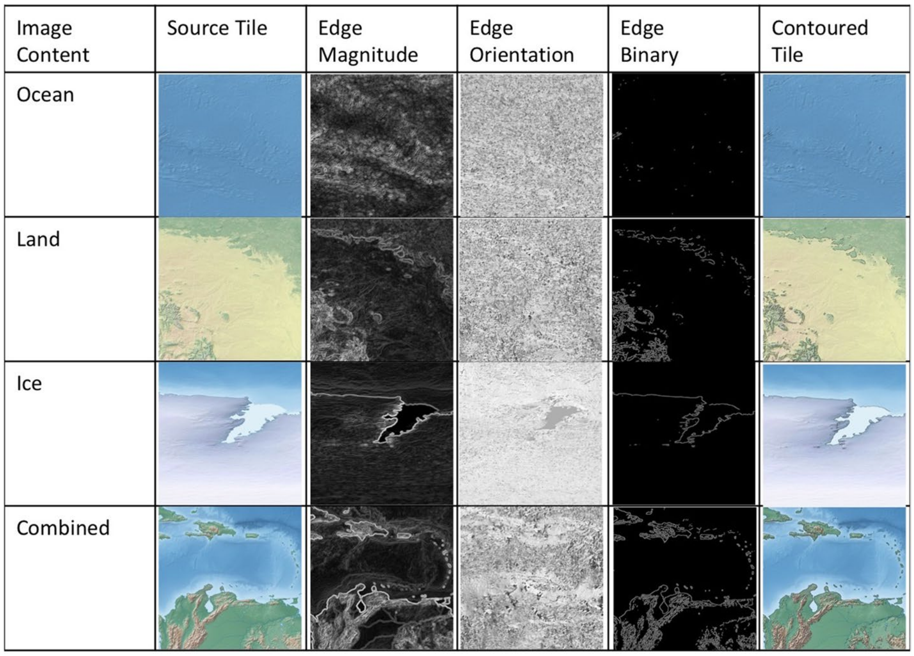


Fig. 18 Accuracy results of Canny edge detection algorithm

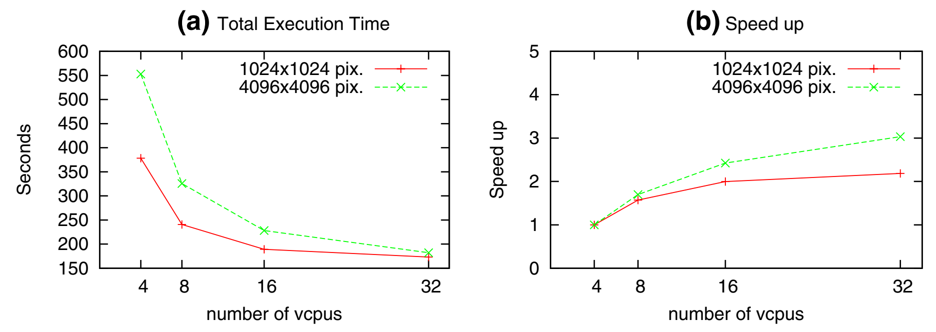


Fig. 19 Performance results of Canny edge detection application

vcpu sizes. A near-linear speed-up is observed for the larger tile size case while a logarithmic speed up is observed for the smaller tile size.

Conclusion and future work

In this study, we presented two novel, distributed transfer/ transformation algorithms that used intelligent file distribution and pipelining as optimization techniques. The algorithms scaled well in an AWS cloud setting and outperformed the algorithm from our previous work, AWS ELB's round-robin algorithm and commonly used transfer algorithms in the cloud such as *aws s3 cp* and *s3-dist-cp*. The transformed datasets were tested with a distributed CBIR application for prostate cancer image datasets, which showed near linear speed up in terms of total execution time; a distributed deep learning classification application of breast cancer image datasets, which also showed near linear speed ups for optimal training time, and finally a distributed Canny Edge Detection application on satellite images which showed near-linear speed-ups for larger tile sizes. As future work, we intend to develop serialization methods for more flexible reader libraries for our transformation formats to target a large range of scalable analytics applications. We also plan to improve our transfer algorithms by setting the optimal parallelism levels automatically.

Abbreviations
WSI: Whole slide image; AWS: Amazon web services; CBIR: Content-based image retrieval; DAS: Dynamic asynchronous scheduler; GS: Greedy scheduler; PPGS: Pipelined greedy scheduler; ROI: Region of interest.

Acknowledgements
Not applicable.

Authors' contributions
The corresponding author is the sole owner and producer of all the research materials and methods. All authors read and approved the final manuscript.

Authors' information
Esma Yildirim graduated from Lousiana State University with a PhD in Computer Science focused on Data-intensive Distributed Computing in 2010. She worked as a Scientific Programmer in University at Buffalo until 2012. She was a Research Associate at Rutgers Cancer Institute of New Jersey and Rutgers Discovery Informatics Institute between the years 2015–2017. She is currently an Assistant Professor of Computer Science at the Department of Mathematics and Computer Science in Queensborough Community College of CUNY. Her research interests include biomedical informatics, high-performance computing, high-speed networks, cloud computing and distributed systems.

Funding
This work was supported by PSC CUNY Grant # 62177-00 50.

Availability of data and materials

The dataset used in the prostate carcinoma case study is available at the cancer genome atlas data portal (https://portal.gdc.cancer.gov). The other dataset for breast cancer case study and the code for the algorithms are available from the corresponding author on reasonable request. A newer version of the breast cancer dataset (without tumor mask images) is available at Camelyon challenge website (https://camelyon17.grand-challenge.org). The Natural Earth Dataset can be found on the website https://www.naturalearthdata.com.

Declarations

Competing interests

The author declares that she has no competing interests.

Received: 13 August 2021 Accepted: 23 November 2021
Published online: 07 December 2021

References

1. Yildirim E, Foran DJ. Parallel versus distributed data access for gigapixel-resolution histology images: challenges and opportunities. IEEE J Biomed Health Inform. 2017;21(4):1049–57.
2. Bueno G, Gonzalez R, Déniz O, García-Rojo M, Gonzalez-Garcia J, Fernández-Carrobles M, et al. A parallel solution for high resolution histological image analysis. Comput Methods Programs Biomed. 2012;108(1):388–401.
3. Farahani N, Parwani AV, Pantanowitz L. Whole slide imaging in pathology: advantages, limitations, and emerging perspectives. Pathol Lab Med Int. 2015;7:23–33.
4. Openslide; 2021. Available from: https://openslide.org.
5. Goode A, Gilbert B, Harkes J, Jukic D, Satyanarayanan M. OpenSlide: a vendor-neutral software foundation for digital pathology. J Pathol Inf. 2013;4.
6. Moore J, Linkert M, Blackburn C, Carroll M, Ferguson RK, Flynn H, et al. OMERO and Bio-Formats 5: flexible access to large bioimaging datasets at scale. In: Medical Imaging 2015: Image Processing. vol. 9413. International Society for Optics and Photonics; 2015. p. 941307.
7. Borthakur D, et al. HDFS architecture guide. Hadoop Apache Project. 2008;53(1–13):2.
8. Amazon Simple Storage System; 2021. Available from: https://aws.amazon.com/s3/.
9. Braam P. The Lustre storage architecture. arXiv preprint arXiv:190301955. 2019.
10. Schmuck FB, Haskin RL. GPFS: a shared-disk file system for large computing clusters. In: FAST. vol. 2; 2002.
11. OpenCV library; 2021. Available from: https://opencv.org.
12. Teodoro G, Kurc T, Kong J, Cooper L, Saltz J. Comparative performance analysis of Intel (R) Xeon Phi (TM), GPU, and CPU: a case study from microscopy image analysis. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. IEEE; 2014. p. 1063–1072.
13. Zerbe N, Hufnagl P, Schlüns K. Distributed computing in image analysis using open source frameworks and application to image sharpness assessment of histological whole slide images. In: Diagnostic pathology. vol. 6. BioMed Central; 2011. p. S16.
14. Aji A, Wang F, Vo H, Lee R, Liu Q, Zhang X, et al. Hadoop-GIS: A high performance spatial data warehousing system over MapReduce. In: Proceedings of the VLDB Endowment International Conference on Very Large Data Bases. vol. 6. NIH Public Access; 2013.
15. Hadoop; 2021. Available from: https://hadoop.apache.org.
16. Chard R, Madduri R, Karonis NT, Chard K, Duffin KL, Ordoñez CE, et al. Scalable pCT image reconstruction delivered as a cloud service. IEEE Trans Cloud Comput. 2015;6(1):182–95.
17. Parsonson L, Grimm S, Bajwa A, Bourn L, Bai L. A cloud computing medical image analysis and collaboration platform. In: International Conference on Cloud Computing and Services Science. Springer; 2011. p. 207–224.
18. Kagadis GC, Kloukinas C, Moore K, Philbin J, Papadimitroulas P, Alexakos C, et al. Cloud computing in medical imaging. Med Phys. 2013;40(7).
19. Madduri RK, Sulakhe D, Lacinski L, Liu B, Rodriguez A, Chard K, et al. Experiences building Globus Genomics: a next-generation sequencing analysis service using Galaxy, Globus, and Amazon Web Services. Concurr Comput: Pract Exp. 2014;26(13):2266–79.
20. Milletari F, Frei J, Aboulatta M, Vivar G, Ahmadi SA. Cloud deployment of high-resolution medical image analysis with TOMAAT. IEEE J Biomed Health Inform. 2018;23(3):969–77.
21. Godinho TM, Viana-Ferreira C, Silva LAB, Costa C. A routing mechanism for cloud outsourcing of medical imaging repositories. IEEE J Biomed Health Inform. 2014;20(1):367–75.
22. Harvey BS, Ji SY. Cloud-scale genomic signals processing for robust large-scale cancer genomic microarray data analysis. IEEE J Biomed Health Inform. 2015;21(1):238–45.
23. Jackson KR, Ramakrishnan L, Muriki K, Canon S, Cholia S, Shalf J, et al. Performance analysis of high performance computing applications on the amazon web services cloud. In: 2010 IEEE second international conference on cloud computing technology and science. IEEE; 2010. p. 159–168.
24. Bremer E, Almeida J, Saltz J. Representing whole slide cancer image features with Hilbert curves. arXiv preprint arXiv:200506469. 2020.
25. Qi X, Wang D, Rodero I, Diaz-Montes J, Gensure RH, Xing F, et al. Content-based histopathology image retrieval using CometCloud. BMC Bioinformatics. 2014;15(1):287.
26. Spark; 2021. Available from: https://spark.apache.org.
27. Image Processing and Analysis in Java; 2021. Available from: https://imagej.nih.gov/ij/index.html.

28. AWS Elastic Load Balancing; 2021. Available from: https://aws.amazon.com/ru/elasticloadbalancing/application-load-balancer/.
29. Breast Cancer Classification with Keras and Deep Learning; 2021. Available from: https://www.pyimagesearch.com/2019/02/18/breast-cancer-classification-with-keras-and-deep-learning/.
30. Breast Histopathology Images; 2021. Available from: https://www.kaggle.com/paultimothymooney/breast-histopathology-images.
31. Natural Earth Dataset; 2021. Available from: https://www.naturalearthdata.com.
32. Koschan A, Abidi M. Detection and classification of edges in color images. IEEE Signal Process Mag. 2005;22(1):64–73.
33. Burger W, Burge MJ. Digital image processing: an algorithmic introduction using Java. Berlin: Springer; 2016.

Publisher's Note