IET Image Processing

Research Article

IET Journals
The Institution of Engineering and Technology

Distributed ImageJ(Fiji): a framework for parallel image processing

ISSN 1751-9659
Received on 4th February 2019
Revised 14th April 2020
Accepted on 29th May 2020
E-First on 9th September 2020
doi: 10.1049/iet-ipr.2019.0150
www.ietdl.org

Md Amjad Hossain[1] ✉, Preoyati Khan[1], Cheng Chang Lu[1], Robert J. Clements[2]
[1]Department of Computer Science, Kent State University, Kent, OH, USA
[2]Department of Biological Sciences, Kent State University, Kent, OH, USA
✉ *E-mail: mhossai2@kent.edu*

Abstract: ImageJ is an open-source application widely used for image processing. It has developer API that can be used to implement new plugins for specific image processing tasks. However, ImageJ wasn't designed to work on distributed systems. Currently, it is still being used on single machines to process large medical images, which takes several hours to complete. In this article, we present the approaches to make several essential and widely used ImageJ plugins to work in a cluster. As the cluster nodes parallelly run the existing plugins for image processing, they write the results on a shared drive. But one of the main challenges is, merging those results with high accuracy. Several ImageJ plugins were developed to distribute tasks and generate combined results efficiently. The existing 3D-Object-Counter plugin was used for testing the designed system. The experimental results on the test images of 3D objects stored in Tagged Image File Format(TIFF) show faster processing time with high accuracy and similarity compared to the single machine-based results. The image processing time depends on the number of nodes in the cluster. So, we present a mathematical model that determines the cluster size automatically for optimizing the overall image processing time.

1 Introduction

In recent years, advancements in the medical and biological sciences made imaging an increasingly important discipline. Though manufacturers of image acquisition devices include dedicated image processing software, these programmes are not very flexible and do not allow more complex image manipulations. On the other hand, for being easy to use, ImageJ is a widely-used open source JAVA software package for image processing and analysis [1, 2]. ImageJ was originally developed at the Research Services branch of the National Institutes Health (NIH) as a tool to support the analysis of scientific imagery, and formerly known as NIH Image. The software has grown in popularity over the last 30 years, used in many research labs across the world, and driven by a large community of users actively developing extensions (known as plugins) to the software via simple programming interface [3, 4]. Several modifications and application-specific versions of ImageJ have also emerged, including FIJI [5] (a feature-rich ImageJ distribution) with embedded plugins and tools for segmenting and assessing biomedical image samples and beyond. These tools and extensions are indispensable to many microscopists and augment their ability to segment, analyse, and process relevant datasets in the realm of biology and beyond.

While these resources are extremely useful, the rapid development of equipment capable of generating massive datasets and the need to analyse these are hindered by the lack of computing power. In fact, the individual image stacks (containing 3D objects) to be processed are often very large. Processing these image stacks using ImageJ/Fiji on a single machine takes a massive amount of time. Also, ImageJ frequently duplicates the image being processed and requires an individual computer to have a very large main memory that might not be available in most of the cases. However, if we can distribute the processing tasks among several processors or small computers, then the processing time could be reduced significantly, and this would also enable those small machines to process large images satisfying large memory requirements. These processing entities can be located at different sites but must operate in a non-interfering and cooperative manner [6]. Thus, providing the ability to distribute the computation within ImageJ via a simple user interface could provide a significant enhancement in throughput and scientific discovery for thousands of research labs worldwide. However, ImageJ was not designed to work on distributed systems. Rewriting all the available plugins and macros (a series of ImageJ commands) for distributed processing is not a feasible solution.

In this work, we implement a cluster-based system that makes existing Macros and Plugins of ImageJ/Fiji run on distributed nodes of the cluster. Many existing plugins, such as adding noise, filtering, smoothing, sharpening, inverting, finding edges, making binary, etc. process images without generating any statistical information. They also do not need information about the full input image while working on a selected area of the input image. For these plugins, we need to split the input image into multiple tiles(smaller image stacks) and assign them to the processing nodes in a round-robin fashion. When every individual node successfully finishes the processing of the designated tile, the next task is to combine or stitch back all tiles to one large image. However, if a plugin generates relevant data after parallel processing, merely combining these output data from cluster nodes might not give correct results. For example, there is an existing Plugin called $3D\,Object\,Counter$, which counts 3D objects from a stack of images and provides some measurements, including the volume and centre of each object found [7]. A 3D object in the image stack is identified as a group of voxels (pixels in 2D) of the same colour. If we split the input image stack into several tiles for parallel processing, then there is a chance to divide some 3D objects into multiple parts.

Fig. 1 shows the splitting of the Z-projection of image stack containing 3D objects. As we can see, more than one image tiles might have a chunk of the same object. Such a chunk of a fragmented object is referred to as a duplicate object throughout this paper. The $3D\,Object\,Counter$ plugin will count each chunk as an object in individual tile. So, it will lead to a problem of counting an object more than once. We implement multiple ImageJ plugins for detecting these duplicate objects and providing actual object counting results from parallel processing. Thus, the developed system provides a platform to run macros of existing plugins in parallel. We consider the 3D-Object-Counter plugin for testing the system. The experimental results on test images of 3D objects stored in Tagged Image File Format (TIFF) show faster processing

© The Institution of Engineering and Technology 2020

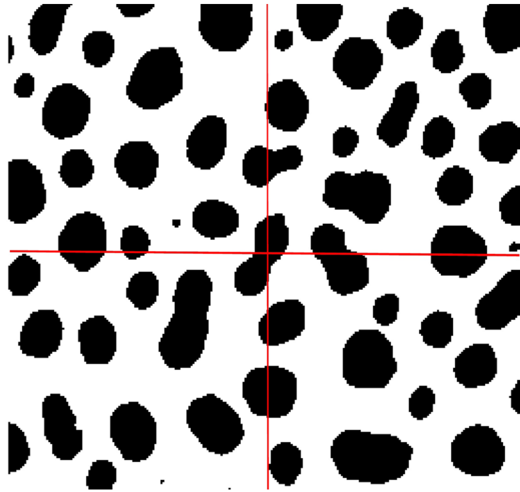

Fig. 1 *Splitting image stack for distributed processing*

time with high accuracy and similarity compared to the single machine-based results. We observe that the number of nodes in the cluster has a significant impact on the image processing time. So, we present a mathematical model that determines the cluster size automatically for optimising the overall image processing time.

The remaining of this paper is organised as follows: In Section 2, we discuss related works. Section 3 presents the system architecture used and the operational overview of the system. In Section 4, we discuss different methods for splitting input images and running the 3D object counter plugin in the cluster. Section 5 presents and discusses the experimental results. In Section 6, we present the mathematical model to determine the cluster size automatically. Section 7 concludes the paper.

2 Related work

Previously, several attempts have been made to run ImageJ in a distributed environment [8–11]. Lindsey developed a plugin called Fiji Archipelago, which is useful for exporting Fiji/ImageJ functionality over a network to several other computers [8]. It also can add new cluster nodes on-the-fly. The root node starts and manages the cluster and the client nodes perform computations exported from the root node. The communication between the root and clients is done over ssh standard IO by default. However, the plugin does not provide any support to other plugins that require additional correction on their results after running on the cluster.

In [9], the image processing has been done on a cluster with ImageJ in headless mode. It uses ImageJ macros for actual processing. A short shell script runs the ImageJ macros after qsub throws the jobs to the cluster. However, the system does not work with macros that require displaying the images to be processed. In [10], Zerbe *et al*. presented a distributed computing system for image analysis using ImageJ and Java Parallel Processing Framework (JPPF) [11]. They mention about data merging from distributed nodes but no experimental data found with acceptable result accuracy.

Other attempts of distributed Image processing are discussed in [12–14]. In [12], Klimeck *et al*. demonstrated the efficient use of cluster computers for the near real-time ground data processing of Mars rover images for large-scale mosaics and left-right stereo image correlation. For this task, they develop a parallel software from existing serial software using Message Passing Interface (MPI). However, the software was developed for Applied Cluster Computing Technologies at JPL and it is not open for others to use. A similar MPI-based parallel implementation is presented in [13], where nearly real-time processing of raw synthetic aperture radar (SAR) data was possible with eight times speedup on nine processors. Liu *et al*. proposed the SEIP in [14], which is built on Hadoop to support various kinds of image processing algorithms on distributed platforms with GPU accelerators. However, none of these works discusses the challenges associated with combining the distributed results into final results.

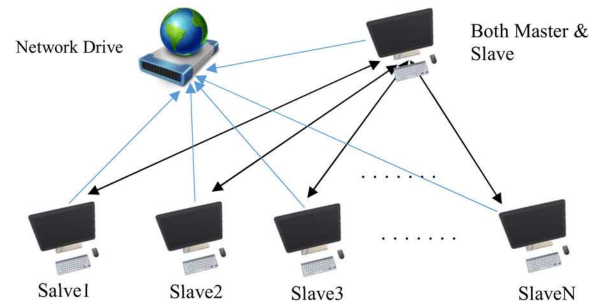

Fig. 2 *System architecture*

In distributed image processing, it is important to use efficient methods for splitting the input images and distributing them to the processing nodes. The methods with high accuracy are also needed to merge the distributed results. Some of these methods are presented in [15–18]. Haralick and Shapiro described the main ideas behind the major image segmentation techniques in their paper, including the classic split-and-merge technique. The split method for segmentation takes the entire image as an initial segment. Then it successively splits each current segment into quarters [15]. Fukada suggests splitting a region successively into quarters until the sample variance is small enough [16]. Browning and Tanimoto described the split and merge technique in their paper, where they first perform the operations on mutually exclusive sub-image blocks then merge the resulting adjacent blocks [17]. Szenasi proposes a new version of the algorithm which is capable of processing large images using the classic split-and-merge technique. The algorithm splits the whole image into smaller images in the first step. Then it runs the region growing on these smaller images, and finally merges the results of the separate region growing [18]. In the split methods discussed above, a single computer must load the whole image in the memory, which can be expensive in terms of memory requirement as well as splitting time. In this work, we use modified split methods based on the user input that helps to improve system accuracy and reduce splitting time requiring small system memory.

3 Cluster architecture and operational overview

3.1 System overview

We use the master–slave approach [19] to build the cluster, as shown in Fig. 2. The master or slave can be any local or remote computer. The master is responsible for distributing the work among all the slave nodes, including itself. For our system implementation, we set up a local cluster virtualising a server computer allocating 6 GB RAM and 12 GB Hard Drive for each cluster node. We preconfigure all the cluster nodes with (i) the same Linux operating system, (ii) the same userID and password, (iii) the IP addresses from the same network, (iv) the ssh enabled, (v) a copy of latest version of Image/Fiji with all necessary plugins, and (vi) the MPI package installed. A network drive is mounted with the cluster nodes as a shared disk that stores the input images, output images, and all other necessary files. All the nodes use the shared disk space to reduce the cost of data communication [20, 21].

The system must start with the master node, which is set up manually. Then, users can add the slave nodes manually or on-the-fly using one of our easily accessible plugins. However, the node to be added must be preconfigured, as mentioned above. Fig. 3 shows the user interface (UI) for node add–delete plugin. It takes the IP address of the node to be added or deleted along with the user ID and the password. The master node maintains a file named *hostfile* that stores the names or the IP addresses of the nodes of the cluster. When the user clicks ‘OK’ to include a new node to the cluster, the plugin automatically adds the name or IP of the new node in the *hostfile*. The plugin also setup passwordless ssh and copies necessary files to the new node. For deleting a node, the plugin just removes the IP address or name of that node from the *hostfile*. Thus, the user can modify their cluster according to the availability of computers.

© The Institution of Engineering and Technology 2020

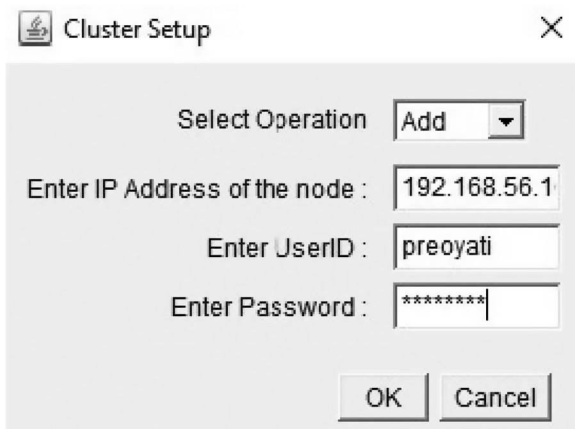

Fig. 3 *User interface for adding and deleting nodes*

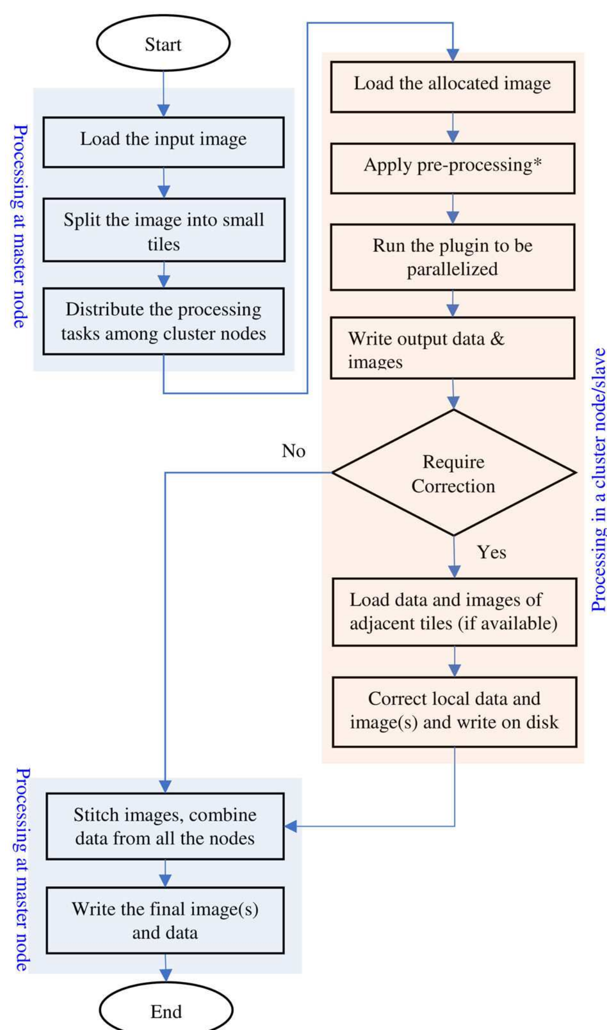

Fig. 4 *Flowchart for distributed image processing using ImageJ*

3.2 Methodology overview

In ImageJ, every image processing method is implemented as a Java plugin. It has thousands of such plugins, including add-noise, smoothing, filtering, sharpening, counting objects in 3D images, segmentation, etc. Users can use a combination of these plugins for a particular image processing task. Some of these plugins would take so long time to process the large images that it would be worthy of running them parallelly. Fig. 4 shows the steps that we can follow for distributed image processing using ImageJ. Every step runs one or more ImageJ plugin or java functions. In some steps, we use existing plugins, such as image splitting, preprocessing, etc. We develop plugins for other steps (task distribution, result correction, merging, etc.) to automate the whole method for single-click processing. The result correction plugin would be different for every plugin to be parallelised.

3.2.1 Methods of splitting input image: For splitting the input image, we can use one of the following methods:

- *Image chopping:* In this method, the master node chops the original image into multiple equal-size tiles and save them in different files. For the task distribution, the master needs to send the file names to the slave nodes. This method requires large memory at the node splitting the input image. The splitting will not be possible if the image size is larger than the available memory space in that node. Moreover, loading the whole input image in memory takes a long time.
- *Region allocation:* Instead of splitting the input image, the master node defines multiple equal-size regions. It calculates the width and height of the regions from the width and height of the original input image. For distributed processing, the master only needs to send the dimension of the regions to the slave nodes. This method is much faster compared to the Image chopping because the width and height of the original image can be obtained without loading the original image. Also, none of the cluster nodes need to load the entire image in the local memory. So, it will allow processing comparatively larger images.

We consider splitting the input images into d^2 equal size tiles or regions where we define d as the split dimension. The value of the split dimension can be decided manually by the user based on the image size, the number of available cluster nodes, etc. However, the system itself can determine the d as well as the cluster size automatically to ensure optimal image processing time. We present a detailed model on this in Section 6.

3.2.2 Task distribution among cluster nodes: For task distribution, the master node creates a job (shell) script for each tile or region. Each job script contains the command for running an ImageJ macro. The macro has the statements for running a series of plugins, including the preprocessing plugins, as well as the main plugins to be parallelised. The user of the system should create the macro manually with a fixed set of plugins. However, it can be modified for a specific image processing task. The command also includes some arguments to be used by the macro. These arguments are the location and dimension of the region, a bit-stream for the plugins, etc. The bit-stream specifies which set of plugins to be executed from the macro.

So, the total number of job scripts is equal to the number of regions or tiles upon splitting, which is d^2. The master assigns these scripts to the cluster nodes to run them in parallel. If the number of cluster nodes is less than d^2, then it assigns the scripts in round-robin fashion. We use qsub [22] to distribute the job scripts among the cluster nodes for image processing in headless mode [9]. However, the headless mode restricts the execution of macros and plugins that require display during the image processing. So, we enable X11 forwarding to each node of the cluster and open Fiji on them so that any plugin can work on the cluster system. We use MPI instead of qsub to distribute the tasks among the cluster nodes for working flexibility because it allows mapping MPI assigned node IDs to job script names.

3.2.3 Parallel processing and result correction: When a cluster node executes the assigned job script, the corresponding macro will run the plugins listed in it, considering the bit-stream argument. The preprocessing (*) plugins can be add-noise, smoothing, filtering, making binary, etc. However, the prepossessing is optional because it depends on the input image as well as the plugin being parallelised. Each plugin has different requirements on the input image to work better. For example, the common preprocessing filters applied for segmentation plugin are 'Deconvolution', 'Subtract Background', 'Gaussian Blur', and 'Find Edges', [23]. For *3D Object Counter*, we use only 'grey-scale conversion' and 'Make Binary' as preprocessing. If the input

© The Institution of Engineering and Technology 2020

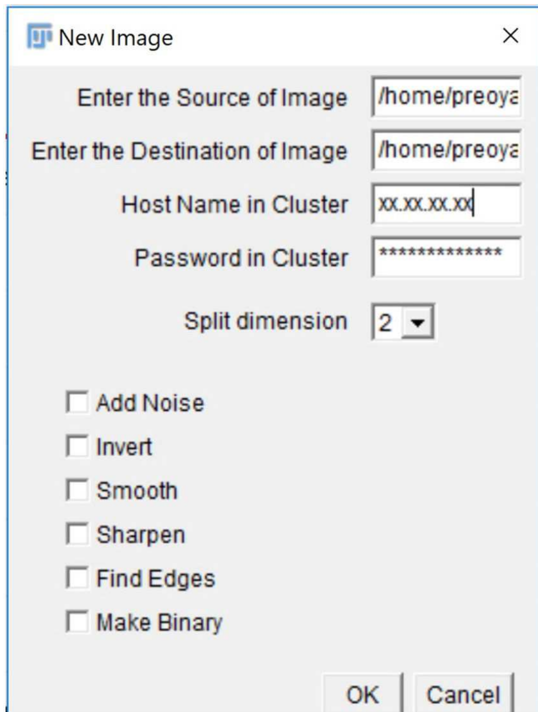

Fig. 5 *UI for sending preprocessing pipeline*

image already satisfies the needs of the plugin, then preprocessing is not required. After running all the plugins, the macro also stores the related data and images on the shared disk.

However, not all the plugins will generate the correct data as the operations executed in parallel may not be independent of each other. For example, the *3D Object Counter* plugin may miscount the number of 3D objects as the input image is split into several tiles, which may cause one object to be divided into two or more. So, the plugin requires the result correction if it is used in parallel processing. We develop supporting plugins for finding the duplicate objects in the output obtained from *3D Object Counter* plugin and correcting the results. The user can instruct the master to create and distribute another script for running the result correction plugins in parallel. Finally, the master can merge the corrected results stored on the shared disk. We discuss detail about the result correction for *3D Object Counter* in Section 4.

3.3 Running the system

As mentioned earlier, ImageJ has thousands of plugins such as add-noise, smoothing, filtering, sharpening, counting objects in 3D images, segmentation, etc. For a particular image processing task, any combination of these plugins can be executed to improve the final result. A macro lists the 'run' statements of all the plugins to be used for the image processing task.

For running the system for the image processing task, the cluster must be set up, as mentioned earlier. ImageJ/Fiji must contain all the necessary plugins, including the plugins for the result correction. For running the distributed image processing job, we provide a user interface (UI) directly accessible from ImageJ, as shown in Fig. 5. From the UI, the user must provide the information for logging in the master node, the split dimension d, and the location of input and output data and images. The user can also send a pipeline of operations/plugins to be executed for a particular image processing task. The pipeline can include different pre-processing operations such as adding noise, filtering, smoothing, sharpening, inverting, finding edges, making binary, etc. It can also include the plugin to be parallelised. Upon click on the 'OK', the master node generates all the necessary shell scripts mentioned earlier. The application converts the selected pipeline to the bit-stream to be used in the macro. It then distributes the scripts among cluster nodes using qsub or MPI. After executing all the operations selected in the pipeline, the cluster nodes save the output images and data on the shared disk drive. If result correction is not required, then the master node stitches the output images and combines the relevant data from all the cluster nodes. For other plugins (such as 3D Object Counter), where data correction is required, the master node creates and distributes another set of job scripts to run the data correction plugins in parallel. After the result correction, the master node combines the results and images.

4 Methods for running 3D object counter in cluster

The distributed execution of pre-processing plugins is simple and straightforward because there is no result correction required when combining the output images. In this section, we discuss how to execute other plugins where complex result correction is required after distributed processing. We pick *3D Object counter* plugin for explaining the whole system. When a user runs this plugin on any image stack of 3D objects, it returns the number of objects in the image stacks, centre voxel of each object, their volumes, etc. The plugin counts only those objects that have the total number of voxels greater than a given threshold say M. The distributed execution of *3D Object counter* plugin should be able to generate all the measurements faster with the lowest error in measurements.

4.1 Image splitting and padding

The original image stack is split into several 3D tiles at the beginning, depending on the given dimension (d). The equation for calculating the number of new tiles (n) is

$$n = d^2 \tag{1}$$

If $d = 2$, then the original image stack is split into four equal-sized 3D tiles or smaller image stacks. Each tile must have the same width and height after the split. For the split operation, we might lose some voxels from the input image. For example, if we have an input image of height 100 and width 100 and if the splitting dimension is $d = 3$, we get 9 images of height 33 and width 33. If we stitch them back, we get an image of height 99 and width 99. Thus, we lose some voxels from the input image. To avoid this problem, we created a plugin for padding the input image with some extra white voxel at the right and bottom side of it. We add one or more extra lines of voxels to the input image as required. For example, in the above case, if we added two rows and columns of white voxels at the right and bottom side of the image. Then after the split, we get 9 images of 34 height and width. Thus we do not lose important voxels from the original image. So, if w and h are the width and height of the input image respectively, we pad input image with p^w lines of white pixels at the right and p^h lines of white pixels at the bottom where

$$\begin{aligned} p^w &= d * \lceil w/d \rceil - w \\ p^h &= d * \lceil h/d \rceil - h \end{aligned} \tag{2}$$

4.2 Parallel steps

The execution time of *3D Object Counter* significantly increases with the image size. So, we can expect significant performance improvement by distributing the execution of *3D Object Counter*. The plugin takes greyscale or binary image stack as input. So, if we have the input image in other formats, we have to apply preprocessing to convert it to greyscale or binary. We can use the available plugins of ImageJ for this purpose by putting them in the preprocessing pipeline or applying explicitly from ImageJ interface. For preprocessing explicitly, run the ImageJ/Fiji, open the input image, then go to (i) Image->Type->8 bit (ii) Process->Binary->Make Binary. Thus, the final input for the plugin is a binary image stack where the black voxels construct the objects, and white voxels are the background. The *3D Object Counter* can be executed on allocated image tile or region in individual cluster node, as explained in Section 3. After running *3D Object Counter* in parallel, the result can be corrected centrally or in parallel. We

© The Institution of Engineering and Technology 2020

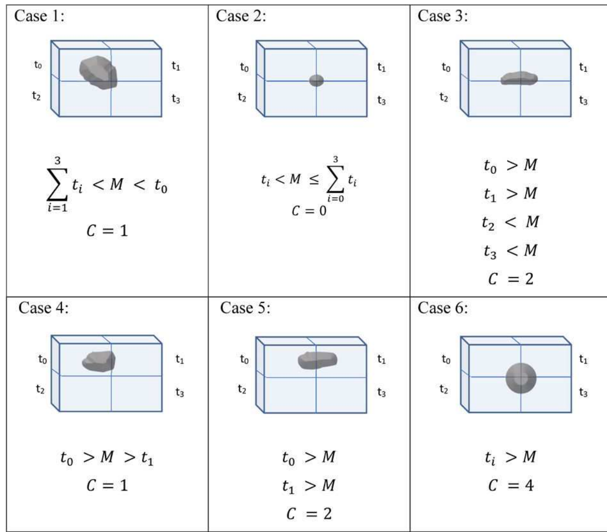

Fig. 6 *Problems of splitting the original image*

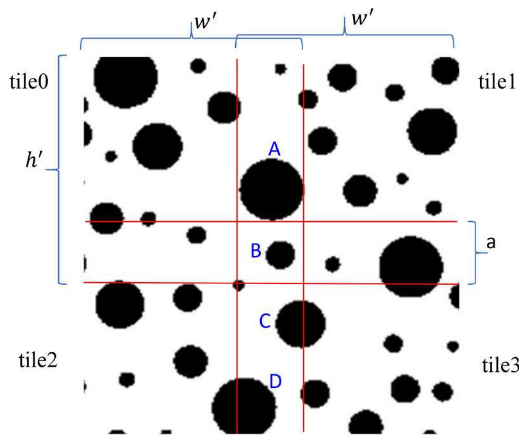

Fig. 7 *Split with overlapped area*

implement our plugins for parallel correction to speed up the overall processing task.

4.3 Necessity of result correction

The result correction is required because of the fragmentation of objects caused by the splitting of the input image. We discuss two methods for splitting image containing 3D objects and their associated reasons for getting incorrect results below:

4.3.1 Splitting without overlaps: In this method, we logically divide the image stack into equal-sized smaller image stacks or tiles, as shown in Fig. 1 and then inform each cluster node to load one of the regions for processing. If w and h are the width and height of the input image after padding, then the dimension of split image stack would be $w' = w/d$ and $h' = h/d$. However, this method can possibly divide one 3D object into at most four parts, where more than one image tiles will have a chunk of the same object. So, distributed execution of *3D Object Counter* plugin would process the same object more than once in different tiles and give incorrect statistics about the objects. Fig. 6 shows a few examples of those critical conditions where an object could be counted 0 to 4 times, depending on its position and the number of voxels in the different tiles.

Here t_i is the number of voxels of an object in $tile_i$, $i = 0, 1, 2$ and 3 and M is the minimum number of voxels together to be counted as an object by *3D Object Counter* plugin. C denotes the total number of times that the plugin counts the object. In the figure, for different cases, we can see the different values of C, which are supposed to be 1. Although the value of C is correct in some cases, Case 1, for example but the volume and the centre of the object, would be incorrect because the number $C = 1$ comes from the first tile and voxels from other tiles are not considered to calculate the volume and the centre. Therefore, a systematic approach is required to correct the **number of objects, their centres, volumes** after parallel execution of *3D Object Counter* plugin. Besides, each node uses local coordinates of the allocated image (each start from 0, 0, 0) and the identified centres of the objects would be in local coordinates. So, the coordinate transformation is also required to get the correct centre voxel of objects in the original image.

4.3.2 Splitting with overlaps: In this method, the input image stack is split based on the assumption that the objects to be counted are close to spherical. Suppose the approximate diameter of the largest object in the image stack is a, i.e. radius $r = a/2$. We split the input image in d^2 new equal-sized images each of width $w' = w/d + r$ and height $h' = h/d + r$. Fig. 7 shows such a split with $d = 2$ on the z-projection of an image stack containing spherical objects. The image splitting with an overlap area of width a makes sure that none of the objects will span from one tile to another crossing the overlap area. This way, some or all voxels of duplicate objects (the object counted at multiple nodes) would always stay in the overlapped area, and at least one node will have full information (centre and volume) of each fragmented object. The objects A, B, C, and D are examples of such objects. The object A is counted twice by adjacent two nodes processing *tile*0, and *tile*1 and both nodes write the same centre and volume information to the shared space. So, for correction, the number of objects needed to be decremented by one and pick other information from any node. Object B is counted four times, and all four nodes write the same information. So, for objects like B, the number of objects needs to be decremented by 3 and pick centre and volume information from any node. For object C, the node processing *tile*2 would have a portion of the object for processing, while another node processing *tile*3 would have full of object C. So, centre and volume generated by two nodes would be different. Therefore, during the correction, we need to keep object information from the later node and decrement the number of objects by one. A similar case for object D but full information would be available at node processing *tile*2. However, to generate the final result, we must perform the coordinate transformation to map local voxel coordinates of tiles to coordinate in the original image.

4.4 Methods of correcting results

After parallel execution of *3D Object Counter* on allocated image tiles, all nodes write their output data to a shared location. However, each image, after the split, starts from the voxel (0, 0, 0). So, the output centres from tiles except for tile0, need to be corrected as their positions are different in the original input image. Suppose the image was split without overlaps with dimension d. The width and height of the smaller tiles are w' and h', respectively. Then a centre (x, y, z) found in tile t can be corrected into (x', y', z') as follows:

$$x' = x + w' * (t \bmod d) \tag{3}$$

$$y' = y + h' * \frac{t}{d} \tag{4}$$

$$z' = z \tag{5}$$

A similar conversion is also possible if the image is split with overlaps. Now we discuss two different methods below based on the two image splitting methods for finding the possible duplicate objects and correcting the output data about the objects. The first method based on splitting the image stack with overlaps and the second method is based on without overlaps.

4.4.1 Method 1: In this method, we discuss the result correction process for image distribution with overlaps. After parallel execution of *3D Object Counter* on allocated image tiles, all nodes

© The Institution of Engineering and Technology 2020

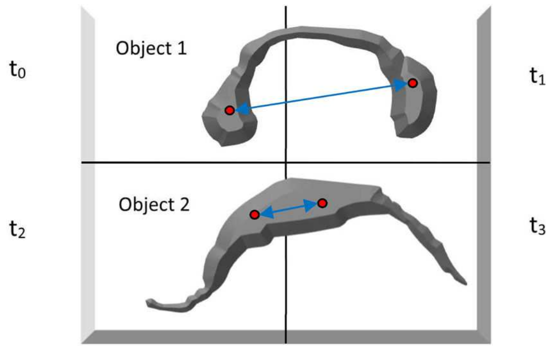

Fig. 8 *Bent objects*

write their output data to a shared location, so the master node can further process these data for a correction. Suppose, the master node accumulates all centres found in all cluster nodes in L_{all} and filter out the distinct centres of the overlapped area in L_{overlap}, therefore, $L_{\text{overlap}} \subset L_{\text{all}}$. We observe that for every centre in L_{overlap}, there are at least two centres of the same object in L_{all}. Those multiple centres in L_{all} can be the same (for objects like A and B) or can be different (for objects like C and D). For result correction, we traverse from each centre $u(x_1, y_1, z_1), u \epsilon L_{\text{overlap}}$ to every centre $v(x_2, y_2, z_2), v \epsilon L_{\text{all}}$ using 3D region growing [24, 25]. In the region-growing, we use Euclidean distance calculated as follow to move from voxel u towards v,

$$e = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \tag{6}$$

In each iteration, we find up to 26 neighbours of the current voxel (8 in the current tile, 9 in each of lower and upper tiles). The region growing moves towards the neighbour who gives the lowest value of e. If both centres u and v are part of the same object, then e becomes 0 eventually. For every such case, we decrement the number of objects by 1, but for every u, we increment 1. For example, L_{overlap} would have a single centre for objects A, C, and D, but L_{all} will have two centres. So, for these objects number of objects would be decremented twice and incremented by one. For object B, L_{all} will have four copies of the same centre. So, the number of objects will be decremented by four and incremented by one. Thus, we correct the number of objects. However, at the same time, we can also correct the volumes and the centres of the objects. From *3D Object Counter*, for every object which is counted, a centre (x, y, z) and associated volume is written to the disk. So, when u reaches other centres v, the v with the largest volume is considered as the corrected centre and the corresponding volume as the corrected volume for that object. This information is correct because, during distributed processing, one of the nodes would have the full object while others get a chunk of it. However, traversing from L_{overlap} to L_{all} is expensive because L_{all} contains the centres of all objects in the input image, and most of them do not have any connection with centres in L_{overlap}. So, we create L_{optimal} that contains all centres from the overlap area (keeping duplicates) and some centres around the overlapped area. Then for result correction, we use L_{optimal} instead of L_{all}. This works because some centres of the duplicate objects might be just outside of the overlapped area. This way, we can correct results for any possible fragmentation of objects.

However, if the estimation of the diameter of the largest object is smaller, then there is a possibility that some objects will be in multiple tiles crossing the overlap area. In this case, the number of objects would be corrected using the same way as discussed above, but for centre and volume correction, we have to perform the additional calculation. On the other hand, a significant overlap area would increase the search space for result correction, and that will increase the overall processing time. So, the correct estimation of the diameter of the largest object is crucial.

This method is ideal for spherical objects and will not work well for complex 3D objects, where the source centre and destination centre of the same object are not traversable from each other by using the shortest path distance method. Fig. 8 shows two bent objects where red dots represent the calculated centres by cluster nodes using *3D Object Counter* plugin and blue connected lines represent the calculated searching line of voxels by the shortest path distance. Tile 0, tile 1, tile 2, and tile 3 are represented by t_0, t_1, t_2, and t_3, respectively. For object 1, centres in t_0 and t_1 are not traversable by the shortest path distance as there is no continuous black voxel in the calculated search line. On the other hand, though object 2 is also a bent object, centres from t_2 and t_3 are traversable from each other by shortest path distance. Thus, the algorithm will detect object 2 as a fragmented object but fail to detect object 1 as a fragmented one. This problem can be solved by using a traverse function that can move back and forth in the image stack to find every voxel of the objects even they are extremely bent.

In practical use of this method, we have a few limitations, as discussed above. In addition, this result correction is done centrally on the master node and it requires the full input image, which would create out of memory issues for large input data and may take huge time to process. For all these reasons, we use method 2 for our system implementation.

4.4.2 Method 2: This is a distributed result correction method unlike method 1 and it is based on the image split without overlap to solve problems discussed in Fig. 6. We have seen that the incorrect results come from the object fragmentation. So, in distributed result correction, each node looks for object fragmentation at the adjacent right and the bottom tiles of the tile it is processing. For that, if a node is allocated to process the tile t, then it also loads the tile $t + 1$ if $((t + 1) \mod d)$ is not 0 (i.e. t has a right tile) and the tile $t + d$ if $\lceil (t + 1)/d \rceil$ is less than d (i.e. t has tiles under it). We refer these three tiles as *Self*, *Right*, and *Bottom*, respectively, in the rest of the paper. In Fig. 8, if the tile t_0 is assigned to a node for processing, then $Self = t_0$; $Right = t_1$ and $Bottom = t_2$.

As we mentioned earlier, the input is a binary image where objects are with black voxels and the background is white. After running *3D Object Counter* we colour the centre to a different colour (say blue) than the object colour. When the image is split, then some voxels (at least one) of fragmented objects would always be on the boundary. Therefore, each cluster node scans all voxels at the rightmost column and the most bottom row of *Self* image stack to find the fragmented objects. During scanning the voxels at the rightmost column, for every black voxel, the node checks for its possible black neighbour in the leftmost column of the *Right* image stack. If it finds such a neighbour, it means both of them are parts of the same object that got separated during the image split. Now the node starts the 3D region growing from both black voxels. When region growing, the black voxels in *Self* tile are changed to grey colour and the number of voxels is counted at both sides. It is also checked if it finds any centre voxel, which is of the colour blue. After the region growing, we will have the following four cases:

- Case 00 – no centres in either sides.
- Case 01 – centre found only in *Right*, no centre in *Self* .
- Case 10 – centre found only in *self* .
- Case 11 – centre found in both side.

If the region-grow finds a group of black voxels but does not find any centre among them, then it means the number of voxels is less than M. Therefore, the plugin did not count this group of voxels as an object. Fig. 9 shows the examples of Case 10 and Case 11, where two objects A and B are divided into two tiles t_1 and t_2. Where object A is divided into a_1 and a_2 and object B is divided into b_1 and b_2. For easy explanation, we show the z-projection of 3D objects with centres coloured in blue. Let the minimum number of voxels for the *3D Object Counter* plugin as input is 5. The number of voxels for a_1, b_1, and b_2 is greater than 5. So, the plugin counts them as objects, i.e. finds their centres. However, as the number of voxels for a_2 is less than 5, the plugin does not count it and we do

© The Institution of Engineering and Technology 2020

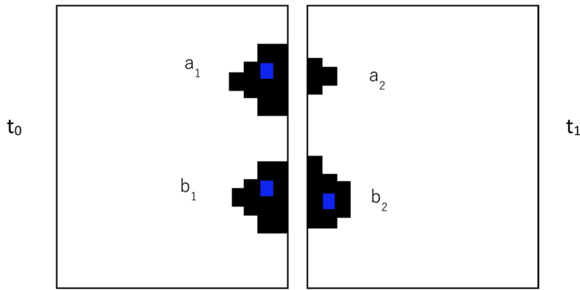

Fig. 9 *Effects of choosing large value of M*

not get any centre for a_2. So, the object A is the example for Case 10 and object B is the example for Case 11. For object A, we only have one centre and one volume, which is from $t_0(Self)$. For volume and centre correction of object A, we simply ignore a_2 as it has only a few voxels. However, during the region growing, we know the number of voxels in a_2 and that can be used to correct the centre and volume. Case 01 is processed in the same way as Case 10. For Case 00, we have voxel-count from the region growing from both *Self* and *Right*. If the total voxel count from both sides is greater than M, then we consider it as a new object and increase the number of objects by one. The volume is estimated from the number of voxels and the centre would be a voxel on the splitting border.

For object B, we have a pair of volumes and centres that are calculated from all voxels of the object, i.e. no missing information. However, when a node is working on *Self* and *Right*, it does not know if there are any parts of these objects (for all cases) in other adjacent tiles. So, it also processes *Self* and *Bottom* as discussed above to find centres of fragmented objects at $Self - Bottom$ splitting line. The only difference is, the objects in *Self* are now of black and grey colours. This way, every cluster node finds all possible pairs of centres that are part of the same object along with their volume information and they are stored as pair as follows for the next step:

$$[(centreX1,\ centreY1,\ centreZ1), Self,\ volume] \\ \rightarrow [(centreX2,\ centreY2,\ centreZ2),\ Right/Bottom,\ volume]$$

where $(centreX1,\ centreY1,\ centreZ1)$ is the centre found in *Self* and $(centreX2,\ centreY2,\ centreZ2)$ found in *Right* or *Bottom* tiles for the duplicate counted objects.

When all pairs of duplicate centres of fragmented objects available from all cluster nodes, then the master node can further analyse these centre pairs to find all the parts of the object because each object can possibly have four parts. If there are two pairs of centres (a,b) and (b,c) then $(a,\ b,\ c)\ \epsilon\ x$, where x denotes an object. Adding volume associated a, b, and c we get an actual volume of an object x. The centre of x is calculated by taking the average of the centres a, b, and c. When combining the results, the number of object counts is decremented accordingly.

5 System implementation and experimental results

5.1 System implementation

For our system implementation, we set up a local cluster virtualising a server computer allocating 6 GB RAM and 12 GB Hard Drive for each cluster node. We implement our system by developing multiple ImageJ plugins based on the image allocation without overlaps and *method*2 of result correction. We do not use *method*1 for developing the final system because the result correction is done centrally on the master node, which takes a long time compared to the time taken by *method*2 where a major correction task is completed in parallel. Moreover, the *method*1 works better only for spherical objects and the user needs to provide the estimated diameter of the largest object in the image.

```
____________________________tile no = 0________________________________
Centers Found From Self Image (0) :
[(141.0,11.0,22.0),0] [(130.0,107.0,31.0),0] [(144.0,148.0,66.0),0]
Centers Found From Right(1) and Bottom(2) :
[(153.0,9.0,18.0),1] [(152.0,101.0,35.0),1] [(145.0,162.0,48.0),2]
[(151.0,149.0,66.0),1]
List of Center Pairs of Duplicate Objects with tile_no and volume:
[{(141.0,11.0,22.0), 0}, 2646.0] -->[{(153.0,9.0,18.0)1}, 652.0]
[{(130.0,107.0,31.0), 0}, 9409.0] -->[{(152.0,101.0,35.0)1}, 829.0]
[{(144.0,148.0,66.0), 0}, 281.0] -->[{(151.0,149.0,66.0)1}, 28.0]

____________________________tile no = 1________________________________
Centers Found From Self Image(1) :
[(194.0,147.0,2.0),1] [(244.0,132.0,24.0),1] [(151.0,149.0,66.0),1]
[(225.0,141.0,77.0),1]
Centers Found From Bottom(3) :
[(196.0,151.0,2.0),3] [(245.0,153.0,33.0),3] [(154.0,165.0,39.0),3]
[(223.0,151.0,71.0),3]
List of Center Pairs of Duplicate Objects with tile_no and volume:
[{(194.0,147.0,2.0), 1}, 152.0] --->[{(196.0,151.0,2.0), 3}, 157.0]
[{(244.0,132.0,24.0), 1}, 8172.0] --->[{(245.0,153.0,33.0), 3}, 1117.0]
[{(151.0,149.0,66.0), 1}, 28.0] --->[{(154.0,165.0,39.0), 3}, 5042.0]
[{(225.0,141.0,77.0), 1}, 4497.0] --->[{(223.0,151.0,71.0), 3}, 167.0]

____________________________tile no = 2_____________________________
Centers Found From Self Image(2) :
[(141.0,210.0,11.0),2] [(145.0,162.0,48.0),2]
Centers Found From Right(3) :
[(150.0,207.0,8.0),3] [(154.0,165.0,39.0),3]
List of Center Pairs of Duplicate Objects with tile_no and volume:
[{(141.0,210.0,11.0), 2}, 3195.0] --->[{(150.0,207.0,8.0), 3}, 45.0]
[{(145.0,162.0,48.0), 2}, 4737.0] --->[{(154.0,165.0,39.0), 3}, 5042.0]
```

Fig. 10 *Duplicate centre pairs written by cluster nodes*

5.2 Input 3D image

The images we use for our experiment are of DAPI (4′,6-diamidino-2-phenylindole) stained mouse nervous tissue slices imaged in 3D using an Olympus FV1000 confocal microscope [26]. DAPI is a fluorescent stain that emits 461 nm (blue) light when bound to DNA and is used to label and detect cell nuclei. It is one of the most widely used stains for many biological applications and an important tool for localising, identifying, and counting cells. DAPI can be combined with other types of stains to probe many different cellular functions in tissues and cell culture systems. Our particular sample is from an animal model of neurodegenerative disease and the ability to count cells provides a measure of the number of infiltrates that are associated with disease severity. The number, centre and volume of the nuclei allow us to probe regional changes in tissue and identify morphological differences in specific sub-types of cells' nuclei. As such, being able to automate counting in large samples provides a method to augment throughput and accuracy by dramatically increasing sample sizes and reducing human error to enhance our ability to screen pharmacological therapies and beyond. Furthermore, the method is applicable to many other studies counting cells (even using different stains) well beyond the scope of neurodegeneration, such as flow cytometry, bacteria colony analysis and cancer drug studies, to name a few.

5.3 Experimental results and discussion

For our experiment, we first run *3D Object Counter* plugin on different sizes of images, 881, 8894, 59402, and 1266073 KB (1.2 GB) directly in the main server machine. Then the same images are processed in the cluster of different sizes (1, 2, and 9 nodes) using our implemented system. The results from the main server are the ground truth that we expect from the parallel processing of the image. The following sections discuss the result correction method (method2) for the image of size 8894 KB processed on the cluster of four nodes. So, we split the input image with $d = 2$. After executing the *3D Object Counter* in parallel, each node starts the result correction method loading *Self*, *Right*, and the *Bottom* image tiles. After scanning and processing, each node generates the pairs of centres for fragmented objects, as shown in Fig. 10. The node processing *tile*3 does not report anything as it does not have any *Right*, or *Bottom* to search for an object fragmentation. The result of tile 0 shows that the number of centres in *Self* is 1 less than the centres found from *Right* and *Bottom*. It means we had a Case 01 from *self* to *Bottom*.

However, as there could be more than two duplicates for an object, we analyse the pairs found from different tiles and search if there is any relationship among them. Fig. 11 shows such relations for this particular input. We can see, an object was divided into

© The Institution of Engineering and Technology 2020

four smaller objects with centres (151, 149, 66), (154, 165, 39), (144, 148, 66), and (145, 162, 48). Taking the average of these duplicate centres, we recalculate the probable centre of the actual object. For this particular object, the probable centre will be (148, 156, 54). And the volume would be 10,088 after adding the volumes of its four duplicates. When merging, the number of objects is decremented by 3 in this case. Similarly, other groups are combined, and the number of objects is decremented by one for each pair. So finally, we report a final list of objects where we replace the fragmented objects by their recalculated centres and volumes. We mark all the objects to show if they are duplicate or original objects with 'duplicate' or 'original' tag. In the output file, those objects are listed in a format of [*Number*. *centreX*, *centreY*, *centreZ*, *volume*, *tag*]. For this example, we report a total of seven duplicate objects as given below:

1. 147, 10, 20, 3298.0, duplicate
2. 141, 104, 33, 10238.0, duplicate
3. 148, 156, 54, 10088.0, duplicate
4. 195, 149, 2,309.0, duplicate
5. 244, 142, 28, 9289.0, duplicate
6. 224, 146, 74, 4664.0, duplicate
7. 145, 208, 9, 3240.0, duplicate

In total, we found 64 objects. Among them, 7 are marked as duplicate and 57 are marked as the original objects. In Fig. 12, we show these seven objects marking their corrected centres with red and the centre of original non-fragmented objects with blue on *Z*-projection of the input image stack.

Table 1 shows the results generated by the different system set up to process the input images of the aforementioned sizes. Analysing the results from this table, we observe that the system found duplicate objects with an error of < 2.5% for very large images. At best, the error rate could be as good as 0%. To achieve that, we can set the minimum voxel count M to a lower value in such a way that $3D\,Object\,Counter$ finds all the split objects. However, small M may increase the overall image processing time. We observe that for the smallest input image (881 KB), the system running on 9 nodes misses out one object giving an error of 6.25%. So, it is not recommended to use the cluster for processing very small images as the overhead of splitting and finding duplicates will degrade the performance as compared with running in a single machine. If we compared the processing time of these images, we find for smaller images (881 KB and 8.68 MB) running in a single machine (main machine and master node) works faster than running in the cluster. However, for a very large image (1.20 GB), the master node could not even run $3D\,Object\,Counter$ due to insufficient memory. For the input image stack of the size 58.01 MB, the master node takes around 5 to 6 times longer compared to cluster-based processing. At the same time, though the main machine has 70 GB RAM, which is more than the total RAM of cluster nodes, it works significantly slower compared to running in the cluster. For the largest image (1.20 GB), it takes around 80 h to run while a cluster of four nodes takes around 18 h and 9-nodes cluster takes only around 5 h to run the plugins and correct the results. Logically, if the input image is divided among nine nodes, we expect the running time of plugin on these images should be around 1/9 times of running time in a single node. As the largest input image (1.20GB) takes 80 h to run on a single machine, we expect from each cluster machine to have a running time of around 80/9 or 8.8 h. In our case, we get ∼5 h, which is much faster than expected. The reason behind this is, the computation is highly dependent on the distribution of 3D objects in the input image. In fact, if we consider the input image with all the 3D objects distributed evenly, we will get a faster running time. Because for the larger images, the $3D\,Object\,Counter$ plugin needs to do much more computations compared to the total computations required when the large image is divided. We also observe that the cluster of four nodes works faster for small images, while the 9-nodes cluster works faster for very large images. Also, running in nine nodes is likely to have more duplicate fragmented objects as the original image is split into smaller tiles. However, the success rate largely depends on the distribution of objects around the split line.

```
[141.0-11.0-22.0-2646.0, 153.0-9.0-18.0-652.0]
[130.0-107.0-31.0-9409.0, 152.0-101.0-35.0-829.0]
[151.0-149.0-66.0-28.0, 154.0-165.0-39.0-5042.0, 144.0-148.0-66.0-281.0, 145.0-162.0-48.0-4737.0]
[196.0-151.0-2.0-157.0, 194.0-147.0-2.0-152.0]
[244.0-132.0-24.0-8172.0, 245.0-153.0-33.0-1117.0]
[225.0-141.0-77.0-4497.0, 223.0-151.0-71.0-167.0]
[141.0-210.0-11.0-3195.0, 150.0-207.0-8.0-45.0]
```

Fig. 11 *Grouping of centres of all duplicate objects*

Fig. 13 shows a logarithmic plot for the processing time in the cluster of different sizes against the size of image stacks. It also shows the processing time of the main server machine. Here, NON stands for the number of cluster nodes. It is clearly pointing that, for smaller input images, the parallel execution does not work well, but as the input size grows, it works significantly faster than running on a single machine. We also observe, as the image size grows, large size clusters work faster. However, splitting the image in more tiles causes more fragmentation of objects, which may reduce the accuracy of the system depending on the distribution of objects. Moreover, because of overhead for task distribution and result correction, only the cluster with a certain number of nodes would give optimum time to process an image of a particular size. We analyse this detail in the next section and provide the formula to automatically calculate the cluster size given the size of the image to be processed.

6 Determining the cluster size

6.1 Determining the split dimension d

Assuming the cluster system has enough cluster nodes so that only one tile is assigned to each processing node, the required number of cluster nodes would be $P = d^2$, where d is the split dimension of the input image. So, the value of the split dimension d is very important in this system because it determines how many processors to be used in the system as well as what will be the processing time of an image. So, given the image size S first, we want to calculate the optimal value of d (i.e. $P = d^2$) so that the overall image processing time is nearly minimum. Processing images in a distributed system have overhead for task distribution and combining the results to obtain final results. Suppose t_c is the time taken to send processing command to one of the cluster nodes and t_m is the time to retrieve and merge results from each process. Each node can process C' bytes of images per second. The total time taken to process the image stack of size S can be estimated from the following equation:

$$T_d = t_c \times P + \frac{S}{P \times C'} + t_m \times P \tag{7}$$

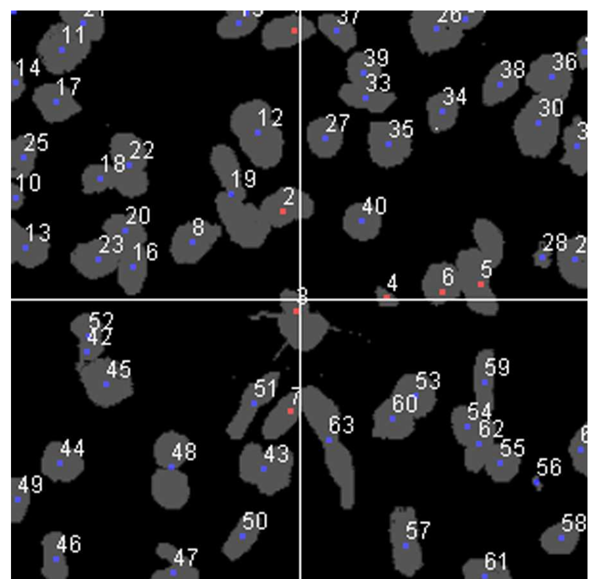

Fig. 12 *Sample output with marked duplicate objects on Z-projection*

© The Institution of Engineering and Technology 2020

Table 1 Output from different size of images

Size, KB	No of nodes	Time	No of total objects	Original object	Duplicate object	Undetected object	% Error
881	1 (main machine)	1 min 27 s	16	16	0	0	0
	1 (master node)	2 min 2 s	16	16	0	0	0
	4	7 min 17 s	16	14	2	0	0
	9	10 min 30 s	15	11	4	1	6.25
8894	1 (main machine)	1 min 33 s	64	64	0	0	0
	1 (master node)	2 min 29 s	64	64	0	0	0
	4	7 min 40 s	64	57	7	0	0
	9	11 min 26 s	63	50	13	1	1.56
59,402	1 (main machine)	1 h 28 min 11 s	218	218	0	0	0
	1 (master node)	2 h 52 min 25 s	218	218	0	0	0
	4	38 min 15 s	213	211	2	5	2.29
	9	29 min 24 s	215	213	2	3	1.38
1,266,073	1 (main machine)	80 h	9606	9606	0	0	0
	1 (master node)	unable to run	—	—	—	—	—
	4	18 h 9 min 27 s	9674	9646	28	68	0.71
	9	5 h 14 min 3 s	9803	9778	25	197	2.05

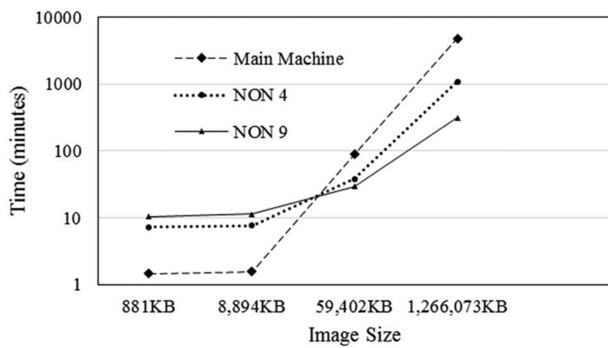

Fig. 13 *Comparison of processing time among cluster systems and main machine for different sizes of images*

Table 2 Sample parameters for (7)

S	t_c	t_m	C'
100	0.2	0.2	5

For a single computer with image processing capacity per second C, the total processing time would be

$$T_s = \frac{S}{C} \tag{8}$$

If we plot the execution time T_d against P for some sample values of other parameters shown in Table 2, we get a quadratic line shown in Fig. 14. The dotted line is for the single computer processing time T_s with $C = 10$.

From the graph we can see, the image processing time is minimum where

$$\begin{aligned} &\frac{\mathrm{d}T_d}{\mathrm{d}P} = 0 \\ &\text{or, } t_c - \frac{S}{P^2C'} + t_m = 0 \\ &\text{or, } C't_cP^2 - S + C't_mP^2 = 0 \\ &\text{so, } P_{\text{optimum}} = \sqrt{\frac{S}{C'(t_c + t_m)}} \end{aligned} \tag{9}$$

Therefore

$$\begin{aligned} d &= \left\lfloor \sqrt{P_{\text{optimum}}} \right\rfloor \\ &= \left\lfloor \sqrt[4]{\frac{S}{C'(t_c + t_m)}} \right\rfloor \end{aligned} \tag{10}$$

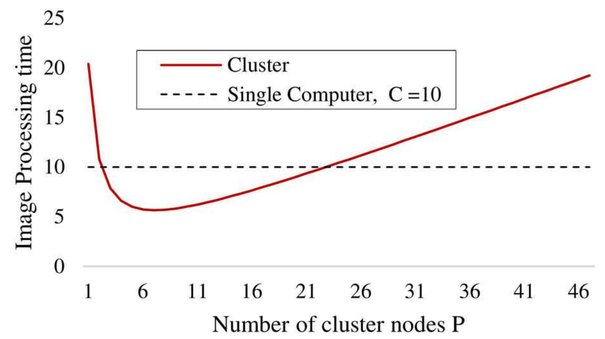

Fig. 14 *Nature of image processing time in cluster*

We use floor function because p_{optimum} might not be a perfect square of an integer number. We could use ceiling function as well, but it would cause more splits in the input image, which leads to more inaccuracy in the combined results.

6.2 Choosing the cluster size from possible values

To choose multi-point distributed image processing over any single computer with image processing capacity per second C, we want the following inequality to be true:

$$t_c \times P + \frac{S}{P \times C'} + t_m \times P < \frac{S}{C} \tag{11}$$

or,

$$CC'(t_c + t_m) \times P^2 - SC' \times P + SC < 0 \tag{12}$$

This equation is in form of quadratic equation $ax^2 + bx + c = 0$, where $a = CC'(t_c + t_m)$, $b = -SC'$, and $c = SC$.

So, if we solve (12) we get

$$P = \frac{SC' \pm \sqrt{(SC')^2 - 4CC'(t_c + t_m)SC}}{2 \times CC'(t_c + t_m)} \tag{13}$$

Now, to get any real value of P, following inequality must be true:

$$\left((SC')^2 - 4CC'(t_c + t_m)SC\right) \geq 0 \tag{14}$$

Therefore

$$S \geq \frac{4C^2(t_c + t_m)}{C'} \tag{15}$$

© The Institution of Engineering and Technology 2020

Table 3 Estimating cluster parameters given image size S

Image size S in KB	$P_{optimum}$	Splitting dimension d	Number of nodes should be used $P = d^2$	P_{max}
881	1.54	1	1	1
8894	4.9	2	4	22
59,402	12.66	3	9	159
1,266,073	58.46	7	49	3416

So, given (15) is true

$$P_{max} = \frac{SC' + \sqrt{(SC')^2 - 4CC'(t_c + t_m)SC}}{2 \times CC'(t_c + t_m)} \quad (16)$$

If (15) is false then there is no real value solution for (11), i.e. single computer will be faster. So, to provide faster image processing by distributed nodes/processors, (15) must be true and the value of P must be between 2 and P_{max}. However, the system can now automatically determine the value of $P = d^2$ as it can easily calculate d from (10). This value will always be between 2 and P_{max}.

6.3 Estimating P from experimental results

From the experimental results in Table 1, we first estimate the value of C' and $t_c + t_m$ using (7), which is possible because several combinations of values for other parameters are available. We found the average $C' = 6.5$ KB and the average $t_c + t_m = 57$ s for our system. Then we calculate $P_{optimum}$, the split dimension d and the actual number of nodes should be used in the cluster, which is P as shown in Table 3 for different sizes of the input images. We also calculate P_{max} considering that we have only cluster computers with all of the same computing power, i.e. $C = C'$. We observe that in the cluster system like we set up, 881 KB is too small to process in parallel. From (15), we get, the image size must be greater than 1485 KB to process it in the cluster. For 1.2 GB image, the $p_{optimum} = 58$ and if we put this value with other parameters in (7), then the image processing time would be 1 h and 51 min. However, according to (10), the split dimension $d = 7$, i.e. $P = 49$ is the best for our system setup and this value of P will take 1 h 52 min to process that image. For lack of resources, we used a maximum of nine nodes, which takes 5 h 14 min and 3 s. The P_{max} can be up to 3416, i.e. we can split the input image in 3364 chunks and the cluster system will still give faster processing time compared to using a single computer. However, as the split dimension increases, the number of fragmented objects also increases significantly. This will reduce the accuracy of the system. So, when splitting the input image, we suggest observing the accuracy of the system. We have to choose d, even far less than the best value if system accuracy drops significantly.

7 Conclusion and future work

ImageJ was not designed to be run in a distributed system. So, running plugins of ImageJ on a distributed system generates highly incorrect results. Simply combining those distributed results would produce very inaccurate overall results. We present different methods for efficiently splitting the input images and correcting results after the distributed execution of ImageJ plugins. Based on these methods, we develop several supporting plugins that allow us to run many ImageJ plugins to run in a cluster system for parallel image processing. For the experiment, we pick *3D Object Counter*, which can process the input image stack of 3D objects and provide the number of 3D objects in the image, their centres as well as volumes as output. We run the plugin on the image stacks of different sizes (maximum 1.2GB) in the cluster of different sizes (here 4 and 9). We observe that for very large images, cluster-based processing works significantly faster compared to processing on a highly configured single node. The methods used for result correction generate combined results from the cluster nodes with very high accuracy. However, because of overhead caused by task distribution and merging the results to/from cluster nodes, smaller images are found to be processed faster on a single computer instead of distributed processing. So, it is important to know whether the input image should be processed in a cluster or not. Therefore, we present a mathematical model that can be used to develop an ImageJ plugin to automatically calculate the cluster size from the size of the input image and other network parameters. We also implement a plugin for cluster management, which provides a user interface for ImageJ to add or remove cluster nodes on-the-fly based on their availability. One main limitation of the system is that it requires a graphical user interface for each Virtual Machine of the cluster. In the future, we would like to make the system work in headless mode so that we can deploy our system in any cluster easily.

8 Acknowledgments

The authors thank Kent State University (KSU) and the Lerner Research Institute (LRI) at Cleveland Clinic (CC) for funding this research project.

9 References

[1] Abramoff, M.D., Magalhaes, P.J., Ram, S.J.: 'Image processing with ImageJ', *Biophoton. Int.*, 2004, **11**, (7), pp. 36–42

[2] Schneider, C.A., Rasband, W.S., Eliceiri, K.W.: 'NIH image to ImageJ: 25 years of image analysis', *Nature Methods*, 2012, **9**, (7), pp. 671–675

[3] 'ImageJ Plugins', https://imagej.nih.gov/ij/plugins/index.html, accessed April 2019

[4] 'ImageJ API', https://imagej.nih.gov/ij/developer/api/index.html, accessed April 2019

[5] Schindelin, J., Arganda-Carreras, I., Frise, E., *et al.*: 'Fiji: an open-source platform for biological-image analysis', *Nature Methods*, 2012, **9**, (7), pp. 676–682

[6] Peleg, D.: '*Distributed computing: a loaclity-sensitive approach*' (Society for Industrial and Applied Mathematics USA, 2000)

[7] Bolte, S., Cordelières, F.P.: 'A guided tour into subcellular colocalization analysis in light microscopy', *J. Microsc.*, 2006, **224**, (3), pp. 213–232

[8] Lindsey, L.: 'Fiji Archipelago', Available at http://imagej.net/FijiArchipelago, accessed January 2019

[9] 'Using Cluster for Image Processing with ImageJ-headless mode', Available at http://wiki.cmci.info/documents/100922imagej_cluster, accessed January 2019

[10] 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', *Diagn. Pathol.*, 2011, **6**, (1), p. S16

[11] 'Java Parallel Processing Framework'. Available at https://www.jppf.org, accessed January 2019

[12] Klimeck, G., Oyafuso, F., McAuley, M., *et al.*: '*Near real-time parallel image processing using cluster computers*' (Space Mission Challenges for Information Technology, USA., 2003)

[13] Cafaro, M., Epicoco, I., Fiore, S., *et al.*: 'Near real-time parallel processing and advanced data management of SAR images in grid environments', *J. Real-Time Image Process.*, 2009, **4**, (3), pp. 219–227

[14] Liu, T., Liu, Y., Li, Q., *et al.*: 'SEIP: system for efficient image processing on distributed platform', *J. Comput. Sci. Technol.*, 2015, **30**, (6), pp. 1215–1232

[15] Haralick, R.M., Shapiro, L.G.: 'Image segmentation techniques', *Comput. Vis. Graph. Image Process.*, 1985, **29**, (1), pp. 100–132

[16] Fukada, Y.: 'Spatial clustering procedures for region analysis', *Pattern Recognit.*, 1980, **12**, (6), pp. 395–403

[17] Browning, J.D., Tanimoto, S.L.: 'Segmentation of pictures into regions with a tile by tile method', *Pattern Recognit.*, 1982, **15**, (1), pp. 1–10

[18] Szenasi, S.: 'Medical image segmentation with split-and-merge method'. Proc. of 5th IEEE Int. Symp. on Logistics and Industrial Informatics, LINDI, 2013, pp. 137–140

[19] 'Master/Slave Model'. Available at https://www.ibm.com/support/knowledgecenter/ssw_aix_71/com.ibm.aix.genprogc/master_slave_model.htm, accessed January 2019

[20] Abandah, G.: 1998 'Reducing Communication Cost in Scalable Shared Memory Systems'. PhD thesis, University of Michigan

[21] Hastings, A.B.: 'Transactional Distributed Shared Memory'. Thesis Summary, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992

© The Institution of Engineering and Technology 2020

[22] ‘Qsub’, http://docs.adaptivecomputing.com/torque/4-0-2/Content/topics/commands/qsub.htm, accessed May 2019
[23] ‘Preprocessing for Segmentation’, https://imagej.net/Segmentation#Preprocessing, accessed April 2020
[24] Revol-Muller, C., Peyrin, F., Carrillon, Y.*, et al.*: ‘Automated 3D region growing algorithm based on an assessment function’, *Pattern Recognit. Lett.*, 2002, **23**, pp. 137–150
[25] Callara, A.L., Magliaro, C., Ahluwalia, A.*, et al.*: ‘*Smart region-growing: a novel algorithm for the segmentation of 3D clarified confocal image stacks*’ (Cold Spring Harbor Laboratory, USA., 2018)
[26] ‘DAPI, or $4^\prime $4′,6-diamidino-2-phenylindole’ https://en.wikipedia.org/wiki/DAPI, accessed January 2019

© The Institution of Engineering and Technology 2020