Universiteit Leiden
Opleiding Informatica

Deploying Phenotype Analysis On LLSC

Name: David van Es

Date: 26/02/2015

1st supervisor: Fons Verbeek
2nd supervisor: Kristian Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract. The study of living cells is an important area of research in bioinformatics. The results of these studies can reveal new insights which can be applied to a great range of applications. One such experiment utilizes high throughput/high content screening to generate a large amount of data in the form of time lapse images of living cells. At LIACS, two image processing algorithms were devised to extract phenotypical measurements from these images. For this study we adapt these algorithms for use on a computing cluster, and explore the effect of parallel computing on their performance.

1 Introduction

In bioinformatics, the study of living cells is an important area of research. One way to study these cells is by using time-lapse microscopy. Time-lapse microscopy involves using a microscope to acquire a consecutive set of cell images by taking pictures at a certain temporal interval. The interval is of arbitrary length and usually dependant on the specific study and based on previous empirical observations. The resulting images can be studied individually, to observe cell characteristics such as morphology, or played back as a movie, to observe cell migration and motility for example.

At Leiden University, experiments of this nature are being performed using a method called High Throughput/High Content Screening (HT/HC). High Content screening has been defined as "the application of automated microscopy and image analysis to both drug discovery and cell biology"[1]. Furthermore, the experiments being performed are classified as high throughput due to the large range of experimental settings and large amount of images taken. The experiment setup has the following stages:

- Experiment Design
- Experiment Preparation
- Image Acquisition
- Image Analysis
- Data Analysis

Though the purpose of the experiments is largely the same, namely to obtain phenotypic measurements from cells, the cell features to be captured and the method of capturing may vary per experiment. To this end the experiment design takes place. Here is decided what information the cell phenotype contains is essential to the research question, and how to go about capturing this information. One experiment that was conducted was concerned with measuring cell migration when cancer cells were exposed to different growth factors. For this experiment, the experiment design phase included deciding upon the growth factors to use, the temporal interval and the culture plate layouts. The images and results of this study are used as one of the main test sets and control sets for the experiments in this study.

During experiment preparation the cells are treated if necessary and loaded onto the well culture plates. Then, for the Image Acquisition phase the automated microscope will take an image of each well on the plate sequentially, wait for the duration of the sample interval and repeat the process. The result of this phase is a set of images. The images are saved as 8 bit greyscale TIFF files. Each individual well plate has its own set of images, which are grouped together and saved as one file in a TIFF container. This can be opened with image processing software like Fiji[3] and played back like a movie.

These first three stages produce a large volume of data. To all intents and purposes it is physically impossible for a human examiner to go through every image and attempt to extract the required phenotypic measurements. Therefore, the image analysis stage is performed by a computer using image analysis techniques. In his thesis[4], K. Yan describes two robust algorithms for image analysis tailored for HT/HC screening studies. These algorithms were implemented using the Fiji software. This software is designed for the biologist and intended for a single user on a single machine, interfacing through a GUI. For small tasks this is a proven setup, but it can prove impractical for high throughput experiments, with the large data sets often leading to long wait times and delays. A full analysis of one well plate can take two to three hours. Any possibility to speed up computation and decrease wait times is therefore highly desirable.

At LIACS we have built the LLSC: the LIACS Life Sciences Cluster. This cluster consists of a fileserver, one user node and 24 worker nodes. This opens up possibilities to use parallel computing for the final two stages: Image Analysis and Data Analysis. In this project, we explore how the existing algorithms and software can be adapted to run on the LLSC and the performance of the resulting parallelized algorithms is evaluated.

2 Background

The experiments in this study utilize two Image Analysis algorithms, Watershed Masked Clustering and Kernel Density Estimation, which are image segmentation and object tracking methods, respectively.

2.1 Image Segmentation

The images obtained from the image acquisition phase are digital images, and as such they can be processed using digital image processing techniques. The purpose of the experiments is to measure phenotypic properties. Since phenotype is defined as the observable characteristics of some object, having some way to determine the boundaries of that object is critical. The success of both algorithms is highly dependent upon this. One digital image processing technique designed for this purpose is image segmentation.

Segmentation is the process of separating an image into its constituent parts or objects. Usually this means separating the background from the foreground. In our image samples this is also the case. Each pixel belonging to a cell in the image is considered part of the foreground, and all other pixels are background. Segmentation is considered one of the most difficult image processing tasks[2], and also one of the most important since a lot of subsequent processing techniques are dependent on the output of the segmentation phase. Separation of foreground and background is key to further decomposing the foreground into an accurate collection of object masks that represent individual cells.

It is important to note that the segmentation phase does not have to take the raw images directly from image acquisition. Segmentation is usually preceded by an Image enhancement phase. In this phase imperfections in the image such as too much noise or low contrast can be adjusted to make the image look 'better'. Our algorithm makes use of enhancement techniques such as subtracting the background, gaussian blur, noise suppression and contrast enhancement.
There are numerous popular segmentation techniques, and each comes with their own strengths and weaknesses. The choice of which technique to use largely depends on the composition of the target image. The next section introduces the WMC technique, the algorithm developed at Leiden University which has proven robust and effective on the image sets produced by the HT/HC experiments.

2.2 Segmenting with Watershed Masked Clustering

Segmentation methods generally operate on one of two basic properties of intensity values: discontinuity and similarity. The former uses abrupt changes in an image to partition the image. The latter attempts to find regions of the image that are similar (in pixel value) to each other. Abrupt changes are usually edges of objects. For similar regions, the notion similar must first be defined. Put simply, for our input images obtained by fluorescence microscopy, similar pixels have close intensity values to neighbouring pixels above a certain threshold.

WMC is a hybrid algorithm made up of four main steps:

1. Image Enhancement
2. Coarse Region Selection
3. Fuzzy C-Means Clustering
4. Mask Merging

First the raw image is enhanced. To make the distinction between foreground and background clearer a contrast enhancer is applied to the image. Then the background is subtracted using a rolling ball algorithm. A gaussian filter is then applied. This produces an image that produces more accurate masks in the following stages.

Coarse region selection partitions the image into adjacent 'coarse' regions. The regions are named coarse because they are only an approximation of a cell object,

as opposed to the 'refined' objects which closely follow the actual cell contours procured through the next steps. The regions are selected through the (similarity based) maxima-seeded watershed algorithm. In this algorithm, local maxima are found. A local maximum has the highest pixel intensity in comparison to its neighbours. From here, regions are 'grown' by viewing the maxima of the image as water sinks. Each pixel will flow toward a water sink i.e. a local maximum and will be added to the region.

From these regions the more refined object mask must be extracted. This is accomplished using fuzzy c-means clustering. One defining attribute of the WMC approach is that each coarse region can be processed independently. For each region, a thresholding method is used. Thresholding is another widely used similarity based segmentation technique. Thresholding methods assume two classes of pixels in an image. Each pixel is mapped to one of these classes by the thresholding function. The threshold is the value that partitions the image. For example, a greyscale image can be thresholded by simply choosing a value t. Each pixel with intensity i and $i < t$ is mapped to class 1, and each pixel with $i \geq t$ is mapped to class 2, or vice versa. The two classes in each coarse region correspond to background and foreground (cell) pixels. To calculate the threshold the weighted fuzzy c means clustering algorithm is used. The region can then be thresholded to obtain refined object masks of each cell.

Fig. 1. A thresholded apple.

It is possible that the coarse regions split an object into multiple parts. This is referred to as overcut. Overcut objects can be identified by a shared watershed boundary. Sometimes these objects have mistakenly been seperated and are actually one single object. The merging stage attempts to identify these mistakenly split objects and merge them if they fit the criteria. The two criteria are object intensity and orientation. Thus, only if there is enough similarity in intensity, and the difference in orientation is below a certain point the objects are merged.

The end result is a binary image. Each pixel was assigned intensity value 255 if it belongs to a cell, and 0 otherwise. The resulting image is referred to as the mask.

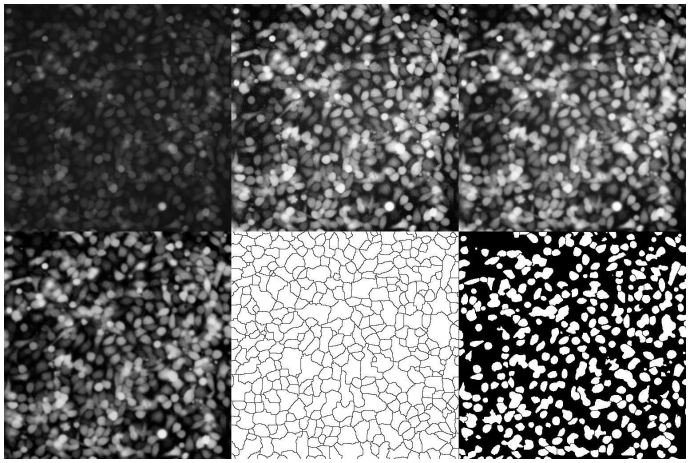

Fig. 2. Segmentation process, from left to right: original image, contrast enhancement, gaussian blur, background subtract, watershed segmented (coarse), final segmented (refined) image.

2.3 Tracking with Kernel Density Estimation

The segmentation step is followed by object tracking. Object tracking algorithms find links between objects. This information can be used to study the movement of these objects over a period of time. In our case, a link must be found between two objects that appear in consecutive images. A successful link found between two objects a and b in images A and B with $a \in A, b \in B$ implies $a = b$, albeit at a different time.

There are several methods and models used to find these linked objects. Methods include particle filter tracking, blob tracking, energy driven linear model tracking and kernel density estimation mean shift tracking. Kernel Density Estimation has been found to work well with our HT/HC experiments, giving a true positive rate of 94.43% and a true negative rate of 92.04%[**?**, yan] For this reason we use KDE in our tracking algorithm.

Tracking results in a collection of trajectories describing the movement of a particular cell. These trajectories, in conjunction with a description of the objects obtained by segmentation, are then used in the Data Analysis step of the experiment.

3 Material and Methods

3.1 LLSC

The LIACS Life Sciences Cluster (LLSC) is a computing cluster recently built at Leiden University. Its intended use is for research related to bioinformatics or the

other life sciences. Initial experiments already conducted include single particle analysis using EMAN[5], and [6]. All experiments conducted in this study have been performed on this cluster.

The cluster is comprised of:

- A single user node, or head node, running the TORQUE scheduler
- 24 worker nodes with varying configurations:
  - 13 nodes with two dual-core Xeon 5150 CPUs and 16 GB main memory
  - 9 nodes with two quad-core Xeon E5430 CPUs and 16 GB main memory
  - 2 nodes with two dual-core Xeon 5150 CPUs and 8 GB main memory
  - All nodes have 400 GB of local storage in a hardware RAID-0 configuration
  - All nodes are interconnected with 100 MBit/s network interfaces
- 2 file servers with 7.5 TB of storage in hardware RAID-5 configuration, 32 GB main memory and connected to the network with a speed of 1 GBit/s

The user node runs the TORQUE Resource Manager. All experiments are run through TORQUE as jobs. TORQUE is responsible for managing resources, the most important of which are the allocation of nodes and scheduling of jobs. TORQUE allows features such as requesting resources, tagging nodes, advanced job logging and statistics, job arrays and easy integration with third party parallel computing solutions such as MPI.

3.2 ImageJ

Both the segmentation and tracking algorithms have been implemented as plugins around the ImageJ framework. ImageJ is an open source image processing program written in Java. It is designed to be used as a standalone application, and is therefore not an image processing library per se, but can be used as such. It natively contains most of the basic image processing algorithms such as contrast enhancement, fourier transforms, applying filters, and more. It recognizes a variety of image formats and includes functionality to save, edit and convert files.

A key feature is the extensibility it provides in the form of plugins. There are hundreds of user made plugins available on the internet. Anyone who wishes to implement an algorithm can do so by implementing the *pluginFilter* or *extendedPluginFilter* interface in Java. By implementing these interfaces and including an ImageJ jar file in a Java project access is granted to essentially all ImageJ features that the standalone version provides.

ImageJ recognizes java class and jar files and automatically installs these as plugins if they contain an underscore in their name and are located in the plugins folder. There is also an option to compile files containing java source code while running the program. Both the segmentation and tracking algorithms were installed as a jar file.

Existing ImageJ Workflow In order to segment an image, the standalone ImageJ application must be run. The user is presented with a menu bar listing all the functions available. The image file must be opened and the option 'WMC Segment' must be chosen from the plugins menu. The currently selected image stack will be processed. Each slice is processed individually and the on screen image is updated slice by slice. The resulting binary masked image can then be saved and the process repeats itself for any subsequent image stacks.

Tracking can be accomplished in a similar way. After the segmented images have been created, these images and the original images are used as input for tracking. The tracking algorithm can be selected from the ImageJ menu. Tracking differs from segmentation in that it does not produce an image as output but raw data. The results of tracking are descriptions of paths and descriptions of the objects in these paths. From both the objects and these paths measurements can be done to measure motility, morphology and other cell phenotype features. This data can be saved as a CSV file and the cell trajectories can either be saved or displayed visually as an image.

Even though ImageJ was designed for one user some automation is possible. Macros can be used to speed up the process for multiple image stacks. ImageJ features its own macro language. A macro can be passed to ImageJ on startup, or selected from the menu. The macro language is interpreted by ImageJ at runtime.

Terminology For clarity we present an overview of the most frequent terminology:

Image Stack A collection of images. These are ordered by the time they were taken. An image stack corresponds to the collection of images belonging to one well plate.

Image Slice A single image taken from a place in the stack.

Partition A subset of slices of a stack, with the stack order preserved. Each nth slice is the direct successor to the n-1th slice. i.e. there are no 'jumps' in a partition.

Overlap Two identical slices in different stacks, where one is the last slice of the first stack and the other is the first slice of the second stack.

Object An object represents a cell in the original image. It contains all the coordinates of the cell, and also meta information such as the slice number it belongs to, median intensity values, and object orientation

Path A path represents a possible trajectory belonging to a cell, across all stacks.

Subpath A subpath is a path across one partition.

3.3 Segmentation

Listing 1.1 gives a high level overview of the procedures involved during the segmentation phase.

```
1  SegmenterPlugin (InputStack image):
2    for ImageProcessor slice in image do
3      segment(slice)
4    save(image)
5
6  segment(ImageProcessor slice):
7    enhanceContrast(slice)
8    gaussianBlur(slice)
9    subtractBackground(slice)
10   watershedSlice := watershedSegmentation(slice)
11   regions := label(watershedSlice)
12   for region in regions do
13     objects := KMeansClustering(region)
14   for object in objects do
15     for x,y in object.mask do
16       pixel(image, slicenr, x, y) := 255 (WHITE)
```

Listing 1. Pseudocode for the segmentation algorithm

From this structure we immediately see two possibilities to utilize concurrency on multiple nodes or processors. Each image stack is processed independently of one another. Furthermore, each slice within a stack is also processed independently. Intuitively this provides two methods to implement concurrency:

1. Stack level concurrency – processing multiple image stacks on different nodes.
2. Slice level concurrency – processing multiple image slices on different cores.

This type of parallelism where multiple processes run concurrently with minimal or no inter-process communication is called *embarrassingly parallel*. Such problems are easy to parallelize and require little to no load balancing. According to Amdahls law, if the slices and stacks are truly independent the speedup should be very close to the number of concurrent processes – i.e. cores∗nodes. This is our hypothesis.

It is also possible to combine the two and use both stack and slice level concurrency. Consider 2 nodes with 4 cores per node and an input set of 8 stacks of 32 slices. Each node can process one stack, and each core can process 8 slices.

In our experiments we test both these methods and examine if the hypothesis is correct.

3.4 Tracking

Object tracking is the more intensive of the two procedures. As we will see, it is also more difficult to parallelize due to dependencies between processes. The pseudocode is presented in listing 1.2.

```
1  TrackerPlugin (InputStack image, InputStack maskedImage) :
2    allObjects := getObjects(image, maskedImage)
3    for n := lastSliceNr to 0 do:
4      foreach object in slice(n) do:
5        if object.visited is true
6          continue
7        else if object.visited is false
8          duplicateCount := 0
9          object.visited := true
10         path := object
11         m = n-1
12         while (m >= 0 and linkObjects(object, getSlice(m)) is found) do:
13           if matchedobject.visited = true
14             duplicateCount := duplicateCount + 1
15           matchedobject.visited := true
16           m--
17           path += matchedobject
18           object := matchedobject
19         if (path >= min_size and duplicateCount/pathlength < 0.5):
20           finalPaths := finalPaths + path
21   save(paths)
```

Listing 2. Pseudocode for the tracking algorithm

The tracker takes as input the original input image stack and the segmented (masked) image stack. Each cell is extracted and stored in allObjects. Each object has information including a slice identifier, object identifier, co-ordinates of the object in the image and measurements such as mean intensity values and orientation. The pair $\{sliceID, objectID\}$ uniquely identifies an object in an image stack, providing the stack has been labeled using the same algorithm.

Recall that the goal of tracking is to find object trajectories. This algorithm works *backwards*, evaluating the slices from the last to the first slice. A slice is evaluated by looping through all objects belonging to that slice. If the object has never been visited during a previous iteration it is added as the first object in a new path. A variable for each object is kept that records when it has been visited. Once the first object has been added an attempt is made to find a link/match in a previous slice using the kernel density estimation mean shift method. This continues until either no match can be found or the first slice has been reached.

When a match can no longer be found the path is evaluated and either accepted as a valid path or discarded. A path with a length less than the minimum length is discarded. The minimum length in all experiments was set to 3. A path with 50% or more of its objects contained in another path is also discarded. All other paths are accepted and added to the final collection of paths.

A property of this algorithm is that there are no two paths with the same initial object and if an object is in a path it is either the initial object or there is no path where that object is the initial object. This is because an initial object is only added if it has not been visited before i.e. does not belong to another path. This also prevents finding paths that are a subset of another path. Consider a path $\{0, 0, 2, 3, 2\}$ from slice 4 to 0. The elements of the path correspond to the object identifier given by the labeler. For object 0 of slice 4, a match is found: object 0 of slice 3. This means that object 0 of slice 3 is visited and is not added as initial object when slice 3 is evaluated. If it was to be added as initial object the path $\{0, 2, 3, 2\}$ would be found. This information would be redundant. A common behaviour of cells is to multiply. When this occurs, two paths are created with a shared ancestor. This is an example of two paths with a large shared subpath that would be excluded from the final collection.
As with segmentation, stack level concurrency can be implemented fairly easily since stacks are again independent. However, it is not immediately apparent how to utilize slice level concurrency for object tracking. Objects in a slice are visited in order, and changing this order can change the output. Slices are not completely independent since at least two slices are needed to link two objects. In the experiments two methods are tested: partitioned concurrent tracking and partitioned object level tracking. The former divides a stack into multiple partitions which are sent to different processors, the latter partitions a stack and further divides each partition into sets of objects which are sent to different processors.

4 Implementation

4.1 Framework

Each experiment is run using one of two methods:

Method 1 The input stacks are stored on one of the fileservers, either in the repository or in a home directory. There are three main components involved. Two scripts written in Python named PA.py and PA-Seg.py, to setup the tracking and segmentation experiments respectively. Two TORQUE/PBS jobscripts using Bash called parallel-tracking.jobscript and parallel-segmentation.jobscript. Finally, a jar file named PA.jar containing the modified ImageJ source and Java implementations for segmentation and tracking. The ImageJ source is modified so that GUI components of certain plugins are no longer instantiated. The call hierarchy is PA.py $\rightarrow$ jobscript $\rightarrow$ PA.jar. PA stands for Phenotype Analysis.

PA.py takes a text file as its argument. This text file itself contains arguments for the rest of the process. PA.py creates a global output directory for the job. Each node has read/write privileges for this directory. It copies a file containing the locations of all input stacks to this directory. It also copies the jobscript, depending on whether it is a tracking or segmentation job. The script also passes

along any relevant arguments to the jobscript. The last statement in both Python scripts executes qsub, which is a TORQUE command and instructs TORQUE to schedule the jobscript.

The jobscript submitted to the cluster contains a header. In this header the output directory can be specified for all job output and logfiles. In our case this is the global output directory. These logfiles can later be used to gather job statistics. Job resources can also be requested. It is possible to request only nodes of a certain type or with a certain amount of memory. The main feature we are interested in is the amount of nodes, and the processors per node. Using these two attributes we can control the total number of resources available per job and measure how total job execution time reacts to different combinations of nodes and processors per node.

The jobscript controls a selected number of nodes and is responsible for calling PA.jar on each node with the correct parameters. For segmentation, each node is given an equal number of stacks to process. If there is a remainder it is again split over the nodes. Once the node 'knows' which stacks it must segment it can run concurrently with no synchronization until it is finished. For tracking, the jobscript first calculates the input arguments which it writes to a file on each node. Again, each node can process its stacks concurrently. When all nodes are finished the joining operation is started.

PA.jar takes different arguments for segmentation and tracking. For segmentation it takes the input stack, the number of cores to use and the location of the output directory. Tracking takes the input stack, the masked input stack, the output directory, the number of cores, and the id's of the slices it must process. An overview of the data flow on the cluster using this method is given in figure 3.

Method 2 The second method is similar to the first but whereas method 1 calls PA.jar using Java method 2 calls Fiji using headless mode and a macro. Enabling headless mode ensures that Fiji does not instantiate GUI objects. It is implemented by using bytecode manipulations at runtime. Since the worker nodes have no GUI any attempt to instantiate GUI objects causes a software crash. Our implementation, in conjunction with this mode and the use of macros caused a bug which would delay Fiji from exiting by approximately 64 seconds. Using this method 2 calls to Fiji are made per job, essentially adding 128 seconds of overhead. This method was only used for out initial segmentation experiments.

4.2 Partitioned Concurrent Tracking

The solution proposed for parallel segmentation falls under the category 'embarrassingly' parallel since the component parts are largely independent of one another. For tracking a different approach is needed. Each object must be linked

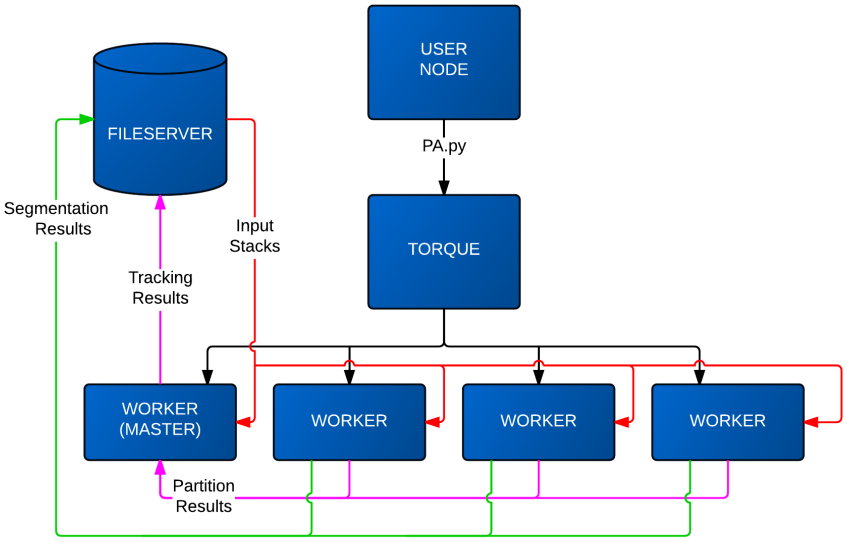

Fig. 3. LLSC data flow overview

to an object in a previous slice and therefore at least two slices must be processed per core. Since the minimum length of a path is 3, a minimum of 3 slices per core is used in the experiments. The principle of partitioned concurrent tracking is to partition each stack into two or more partitions. Each partition has an overlapping slice with the previous and next partition. Then, the algorithm could be run across multiple processors or nodes with different data. This type of parallelism is called data parallelism.

First attempts to implement the partitioned algorithm were implemented using this three step method:

1. Partition an input stack into n partitions $p_1, p_2 \ldots, p_n$.
   Each partition has $stacksize/partitions + 1$ slices, except for possibly the last partition which also processes the remainder.
   For each partition i in partitions $p_1 \ldots, p_{n-1}$, the last slice of p_i is the first slice of p_{i+1}
2. Process each partition according to the pseudocode above.
3. Join the results by
   (a) Finding and adding paths $u_1 \ldots, u_n$ and $v_1 \ldots, v_n$ such that $u_n = v_1$. The resulting path is $u_1 \ldots, u_n, v_2 \ldots, v_n$. Two elements are equal if they have the same object identifier and slice identifier.
   (b) Adding all other paths found to the final results.

Though this approach succeeds in finding paths, there is a major difference with the sequential algorithm. The final paths found using the first version of partitioned tracking were very different from a sequential run. The difference depends on the specific input stack. The lowest similarity measure was 62%. This is due to two reasons:

1. The path properties no longer hold.
2. The objects are being visited in a different order compared to the sequential algorithm.

The two properties were that the path must be longer or equal to the minimum length, and the percentage of already visited objects must be less than 50%. Consider two partitions A, B each containing 4 slices with slice 4 being the overlap. Using the partitioned algorithm, two paths that will be accepted in partition A are $u_A = 1, 3, 5, 7, v_A = 2, 4, 6, 7$. Path u_A will be accepted first, since it starts with a lower object id. Path v_A will then be accepted, since it contains only 25% visited objects. However, if there is a path $w_B = 7, 8, 9, 10$ then the joining algorithm will accept both $join(u_A, w_B)$ and $join(v_A, w_B)$ while the sequential algorithm would not accept $join(v_A, w_B)$. Therefore the percentage of visited objects in the final paths is not always less than 50%. Even though all paths found by partitioned tracking were ≥ 3, not all paths were found. Consider a path in $\{4, 5\}$ in A and $\{5, 6\}$ in B. The sequential algorithm would find $\{4, 5, 6\}$ while the partitioned algorithm would reject both candidate paths.

Since the found paths are different there are three options:

Option 1 A path is in sequential paths and in partitioned paths
Option 2 A path is in sequential paths and not in partitioned paths
Option 3 A path is in partitioned paths and not in sequential paths

We need to eliminate all occurrences of options 2 and 3 so that the paths found in the sequential version are identical to the paths found in the partitioned version. Consider a stack S and two partitions A and B of S. alg_S is the sequential algorithm processing S and alg_A,alg_B the partitioned algorithms.

Because A is the first partition, each object in alg_A is visited in exactly the same order as each object in alg_S. When deciding whether to add an object as the initial object of a path, the decision will be the same for both alg_S and alg_A as the visited boolean will have the same value. The difference in paths of A comes from whether the path will be accepted or not. Option 2 occurs when the a path that ends on the overlap is cut because it is too short, but actually continues in partition B. To include these paths a new rule must be added.

Rule 1 Accept all paths of length > 1 if the path ends on the overlap that would have otherwise been rejected because of length.

Option 3 occurs when the path would be cut by alg_S because of objects not yet visible to alg_A like in a previous example. These paths can be accepted for now

and pruned at a later stage. To recap, with rule 1 applied to alg_A, only option 1 and 3 occur, and option 3 can be pruned later.

As partition B is not the first partition objects that have never been visited in alg_B might have been visited in alg_S. The consequence is that the number of visited objects in a path will differ for the same paths in the two different algorithms, and the choice to accept is no longer the same. Note that this is only relevant for the objects in the first slice (the overlap slice) that is evaluated in alg_B. The reason for this is that all other slices are evaluated in the exact same order as in alg_S. The solution is to accept all candidate paths where the initial object is of the overlap slice regardless of the number of already visited objects. In addition, the paths that were continuations from a previous partition must be kept. Two new rules are added:

Rule 2 Accept all paths of length ≥ 3 if the path starts on the overlap regardless of the number of already visited objects in the path.

Rule 3 Accept all paths where $3 > length > 1$ if the path starts on the overlap.

By combining all 3 rules for alg_B, option 2 is eliminated entirely. Paths corresponding to option 3 must be pruned later and paths corresponding to option 1 are good. If a stack is partitioned into more than 2 partitions, rule 1 2 and 3 must be applied to all partitions excluding the first, which only needs rule 1.

At the end of the alg_A and alg_B, all original paths of alg_S can be obtained from the the separate results. The key is to start with the results of alg_A. Recall that alg_A evaluated each object in the exact same order as alg_S. We need to simulate the workings of alg_S with the results of the other partitioned algorithms. Each path produced by alg_A is again evaluated in order and joined with a path from a subsequent partition. When the path has been joined the two criteria are again tested. Options 2 and 3 can now be completely eliminated. This results in the exact same collection of paths that alg_S found. The pseudocode for this algorithm is given in listing 1.3.

4.3 Partitioned Object Level Tracking

Partitioned concurrent tracking assigns one core to each partition. It is also possible to assign one node to a partition. The processors on each node can then use a different form of multithreading. Each slice in a partition has a number of objects. A subset of objects can be assigned to each processor. Note that this again produces a different set of paths compared to the sequential and partitioned tracking algorithms since the objects are no longer evaluated in the same order. We still consider this in our experiments however. This method is called Partitioned Object Level Tracking.

```
1  for partition P in partitions do:
2    for path in P do:
3      duplicatecount := 0
4      for partition P1 in remainingPartitions do:
5        for path1 in P1 do:
6          if last element of path = first element of path1 then
7            join(path,path1)
8      for each object in path do:
9        if object.visited = true then
10         duplicatecount := duplicatecount +1
11       else
12         object.visited := true
13     if path.length >= minlength and duplicatecount/path.length < 0.5 then
14       results += path
```

Listing 3. Pseudocode for the joining algorithm

4.4 Overview of Concurrency Options

To recap, the concurrency options available for the experiments are:

1. Stack Level Concurrency – Processing different input stacks on multiple nodes
2. Slice Level Concurrency – Processing different slices on multiple cores
3. Sequential Tracking – Processing an input stack sequentially
4. Partitioned Concurrent Tracking – Processing partitions of an input stack on multiple nodes and/or cores
5. Partitioned Object Level Tracking – Processing different sets of objects of an input stack or partition on multiple nodes

Figures 4 and 5 give a graphical representation of the distribution of stacks, slices, and objects using these concurrency options.

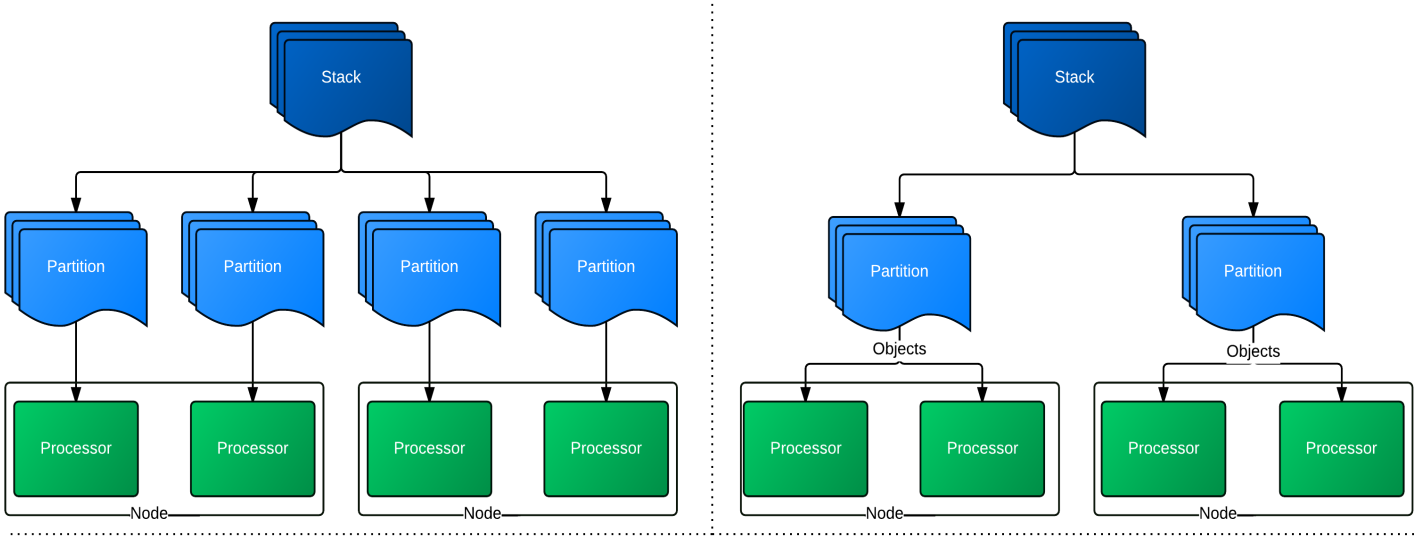

Fig. 4. Partitioned Concurrent Tracking (left) and Partitioned Object Level Tracking (right)

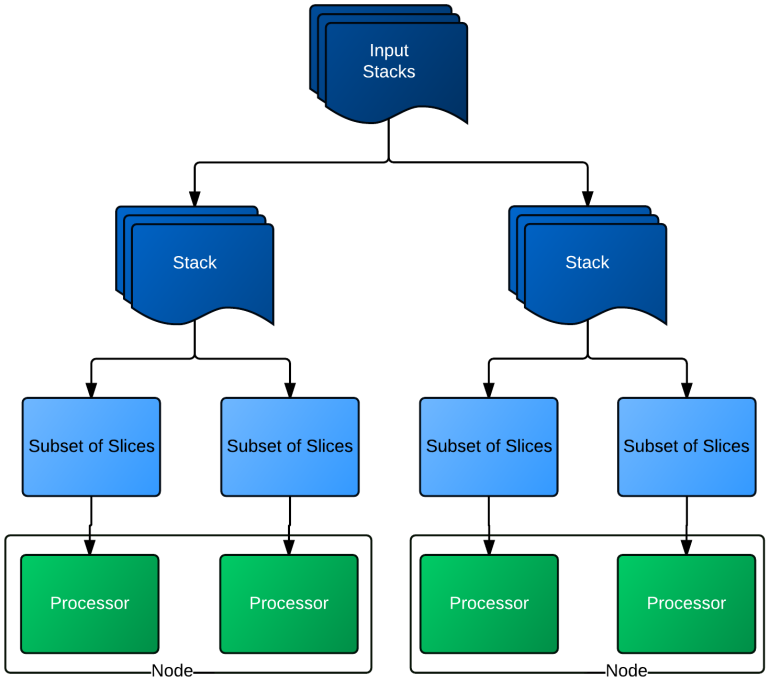

Fig. 5. Stack/slice level concurrency

5 Experiment Setup and Results

5.1 Experiment 1

Setup Experiment 1 uses method 2 as described in Section 4.1 and only performs segmentation. This experiment measures job time for an input set of 512 stacks, processed over 1,2,4 or 8 nodes. The images are MTLn3 cancer cells, cultured in 4 different well plates. The 512 stacks are made up of:

- 144 images from well plate 2, not treated.
- 144 images from well plate 2, treated with EGF.
- 144 images from well plate 3, not treated.
- 88 images from well plate 3, treated with EGF.

Each main job of 512 stacks is split into multiple jobs using TORQUE job arrays. The array is scheduled and run. We vary the number of stacks per job (in the jobarray). The maximum number of nodes tested was 8, effectively giving this configuration of jobs:

Nodes	Jobs	Stacks per Job
1	1	512
2	2	256
4	4	128
4	8	64
4	16	32
4	32	16
4	64	8
4	128	4
4	256	2
4	512	1
8	8	64
8	16	32
8	32	16
8	64	8
8	128	4
8	256	2
8	512	1

Results Figures 6 and 7 show the results of this experiment using up to 4 and 8 nodes respectively. In addition to the actual experiment time the ideal time is shown. The ideal time is the time taken for 1 node divided by the number of nodes. The results show that using 2 nodes with 2 jobs instead of 1 node with 1 job gives a speedup of 1.93. Using 4 nodes with 4 jobs gives a speedup of 3.31. Interestingly, using 4 nodes with 8 jobs still gives a very close speedup of 3.40. After this point the speedup drastically diminishes when using more than 8 jobs

across 4 nodes. On 8 nodes the behaviour is largely analogous. Using 8 nodes with 8 jobs gives a speedup of 6.26. Again, speedup decreases when using more than 8 jobs across 8 nodes.

An explanation for the large increase with for example 512 jobs over 4 nodes is the overhead. TORQUE must schedule 512 jobs instead of 4. However, the real overhead comes from the software bug described in section 4.1. This error causes approximately 128 seconds of overhead to be added per job. With 512 jobs this amounts to an extra 65,536 seconds which is about $4\frac{1}{2}$ hours per node that is wasted.

It is possible to estimate the timings of each job without this bug. Several timings were taken per job: the total time from the first job submission to the last job completion, the average job time, the average segmentation time, and the average tracking time. The estimated total times can be found using the formula:

$$\frac{(average\ job\ time - 128) * nr\ of\ jobs}{nr\ of\ nodes} \tag{1}$$

The results are displayed in figures 8 and 9. The corrected values correspond much more closely with our estimate that there would be a linear scaling as the number of nodes increased. It is again clear that using the same number of jobs as nodes is the best option. With the corrections, using more jobs than nodes still gives less speedup, however it is much closer than before. In theory, the difference in speedup is now solely attributed to the TORQUE overhead and the system calls executed in the jobscripts. This experiment did not copy its local results back to the fileserver therefore no I/O data is available.

5.2 Experiment 2

Setup Experiment 2 is for the most part analogous to experiment 1. However, method 1 is used instead of method 2. This eliminates the overhead caused by the bug. This experiment utilizes all available nodes and was run 3 times. The results reflect the average of these runs. A final difference is that this time the job arrays feature is not used, and there are an equal amount of jobs as nodes.

Results We see that the results verify our corrected results from the previous experiments and that a linear speedup is obtained. Figure 10 shows how segmentation scales for up to 24 nodes.

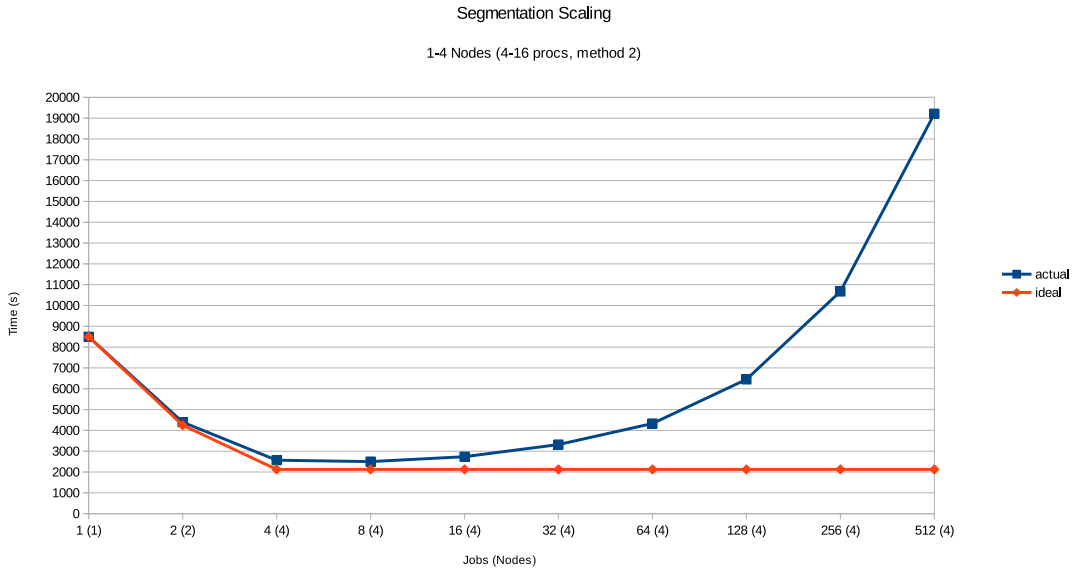

Fig. 6. Segmentation scaling up to 4 nodes using TORQUE job arrays and method 2.

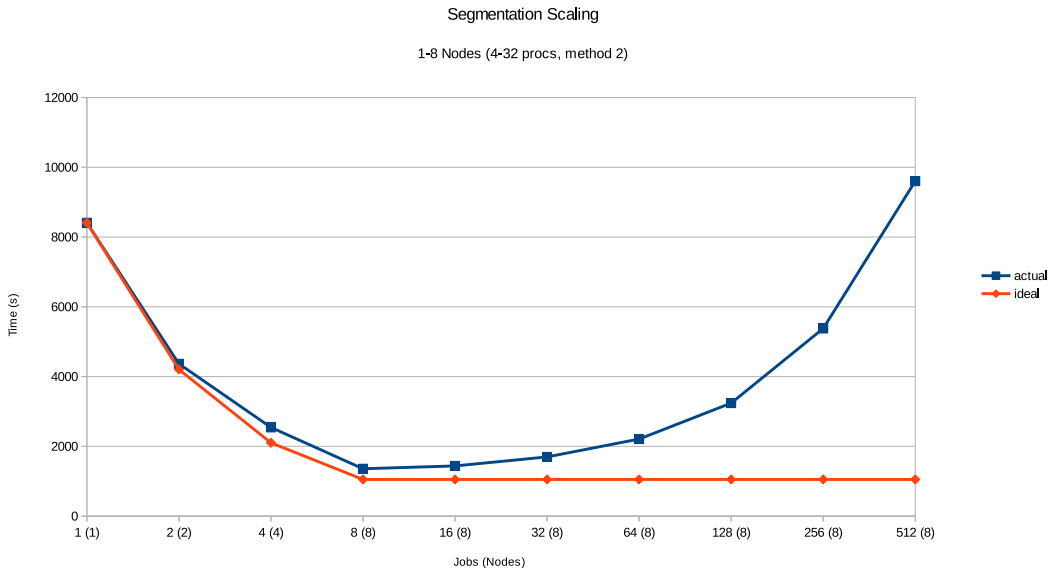

Fig. 7. Segmentation scaling up to 8 nodes using TORQUE job arrays and method 2.

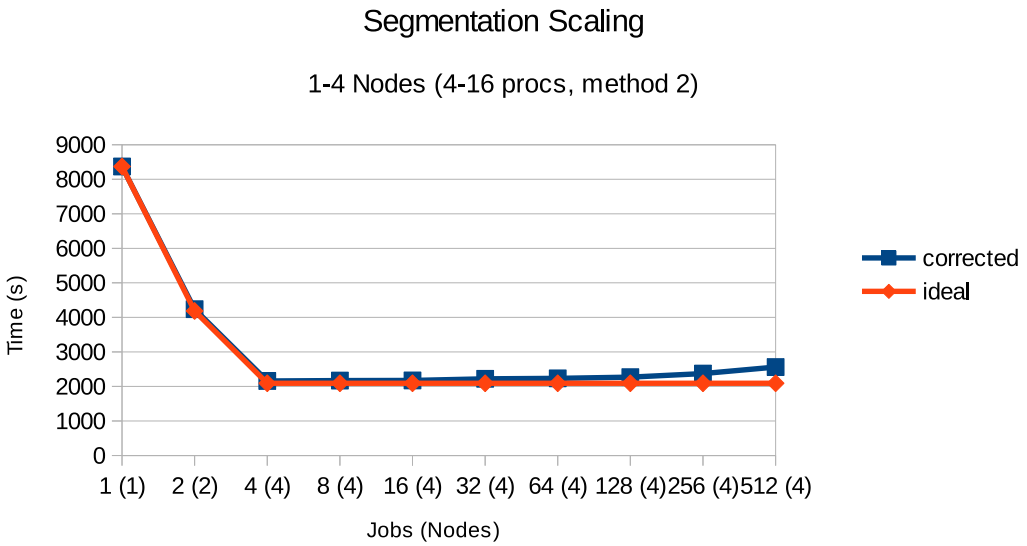

Fig. 8. Segmentation results using up to 4 nodes, with time correction.

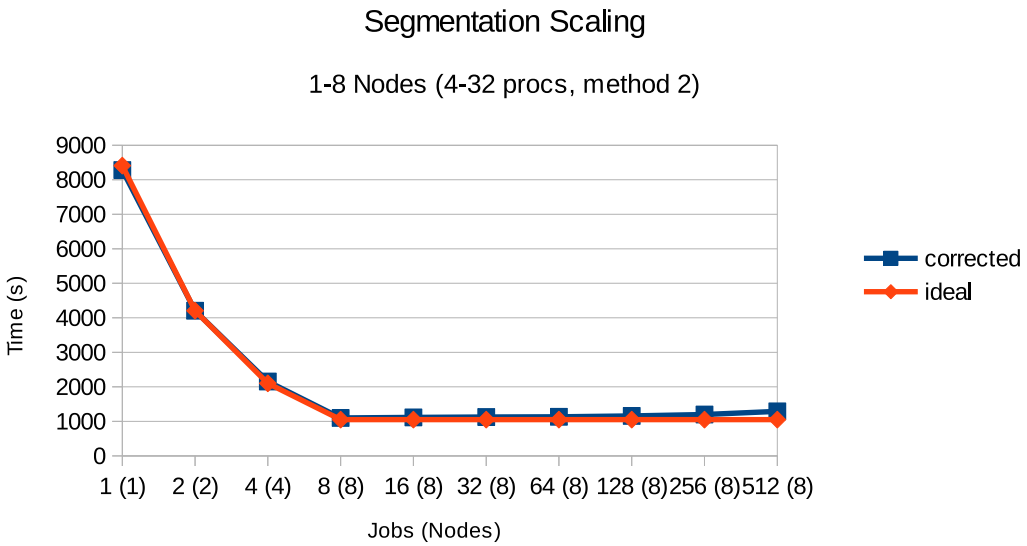

Fig. 9. Segmentation results using up to 8 nodes, with time correction.

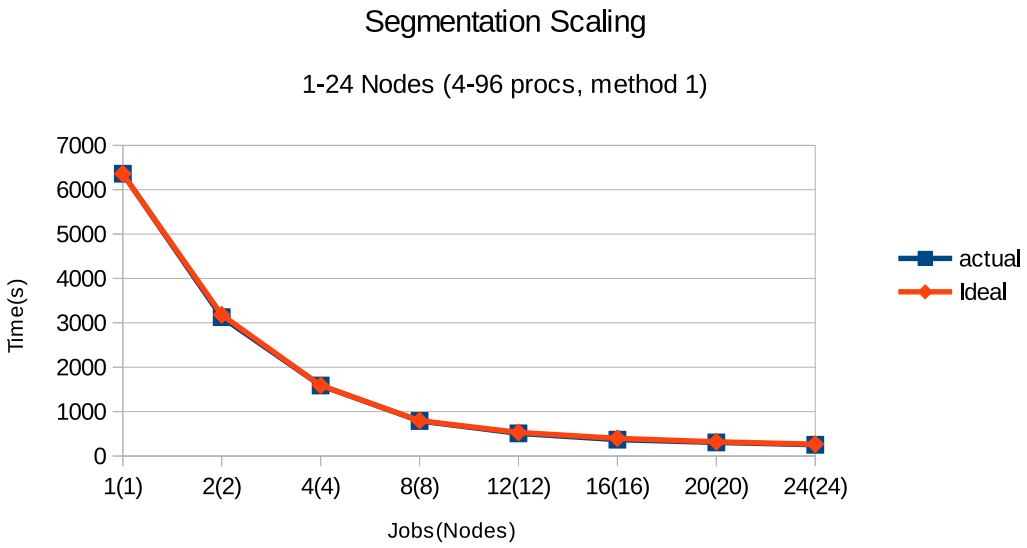

Fig. 10. Segmentation results using up to 24 nodes.

5.3 Experiment 3

Setup Experiment 3 uses method 1 to test different tracking methods. The experiment input is 144 non treated image stacks from well plate 2, and their corresponding masked versions obtained through prior segmentation. Each stack has 31 slices. This experiment processes partitions of a stack concurrently but does not process multiple stacks concurrently. Each image file is read directly from the fileserver and tracked. Once all stacks have been tracked and have written their local results the joining starts. Each node sends its results to a master node where the joining algorithm is run for all stacks. When all algorithms are finished all results are copied back to the fileserver. The tracking, joining and copy operations are all timed separately.

The following combinations are tested:

Algorithm	Cores	Nodes
Sequential tracking	1	1
Partitioned tracking	1	2-10
Partitioned tracking	2	1-6
Partitioned tracking	4	1-3
Partitioned object level tracking	2	1-10
Partitioned object level tracking	4	1-10

Results Each experiment is run 3 times and the averaged results are given in figure 11. The line represents a linear (ideal) speedup. When looking at the total job times the algorithm appears to scale for up to two processors. When 2 or more processors are used the speedup does not increase much further, averaging around the 3.27 mark. An unexpected outcome is that the speedup for 8 or 9 cores is even lower than that for 7 cores.

The total job time includes the tracking time, join time and data copy time. The results of tracking with the copy time excluded are displayed in figure 12. Again, the line represents the linear speedup. This time the chart shows that there is in fact a significant improvement even when more than 2 processors are used. In some cases a superlinear speedup is achieved. The use of 1 core per node consistently gives the best performance. A possible explanation for this is that 1 cpu is used for the algorithm, and in our implementation the entire node (4 processors) has been reserved by TORQUE. No other user intensive user processes were running on the remaining 3 cores. As a result, when 1 core per node is used the entire cpu cache and RAM memory can be used for tracking without interference. When 2 or 4 cores per node are used these resources must be shared, possibly causing increased cache miss rates. Using 4 cores per node is the slowest option (though it still achieves close to linear speedup). Even though it is outperformed, using 4 cores per node is still desirable. If one core per node is used and another user requests the remaining cores for another process performance may drop beyond that of using 4 cores per node. Even if no such request is made the remaining cores are essentially wasted.

It is worth noting that using 7, 8 or 9 processors seems to be worse than using 6. This can be explained by the inefficiency of the slice allocator. Each node is allocated $\frac{size\ of\ stack}{nr\ of\ nodes} + 1$ slices, except the last slice which also processes the remainder. For 6 nodes, each node processes $\frac{31}{6} + 1 = 6$ slices. For 7 nodes, each node processes $\frac{31}{7} + 1 = 5$ slices, except the last slice which processes 8 slices. Similarly, 8 nodes each process 4 slices, except the last which processes 11. Each node must complete before the next stack can be processed. Therefore, when using 7,8 and 9 nodes since there is a node that processes more slices than when using 6 nodes, 6 nodes is faster. This method of slice allocation is the most basic easiest to implement method and should be replaced in a future version.

The difference in speedup between the total job time and tracking + join time is significant. The primary reason is the I/O bottleneck. Each process generates a results file in the form of a serialized object. The size varies depending on the amount of objects in the stack. For busy images in this input set the size was typically 60MB. The reason that the results are written in this form is that the existing implementation also did so. The internal datastructure that is written contains a lot of redundant and duplicate information implicit in the original and masked image stacks. With these stacks and a lightweight vector description of

the paths used in the joining algorithm all information can be reconstructed and therefore the data copy time is less important.
For the partitioned object level tracking displayed in figure 13 anf 14 the results are clearly worse. The same slice allocator is used, but instead of each processor core processing one slice partition, each node processes one partition and assigns a set of objects to each core. Each core accesses a shared datastructure multiple times. This is a possible reason for a slowdown compared to the solely partitioned version.

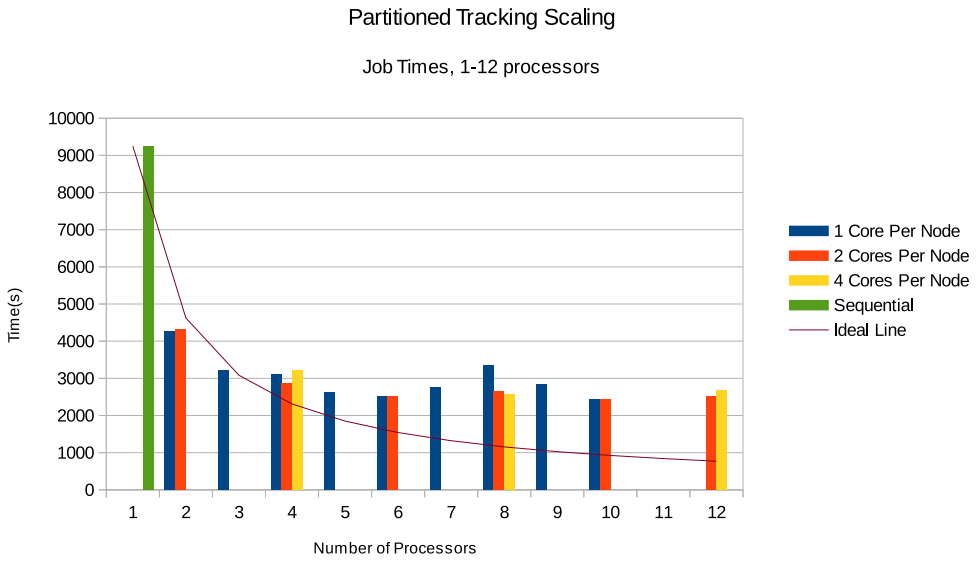

Fig. 11. Partitioned concurrent tracking total job times.

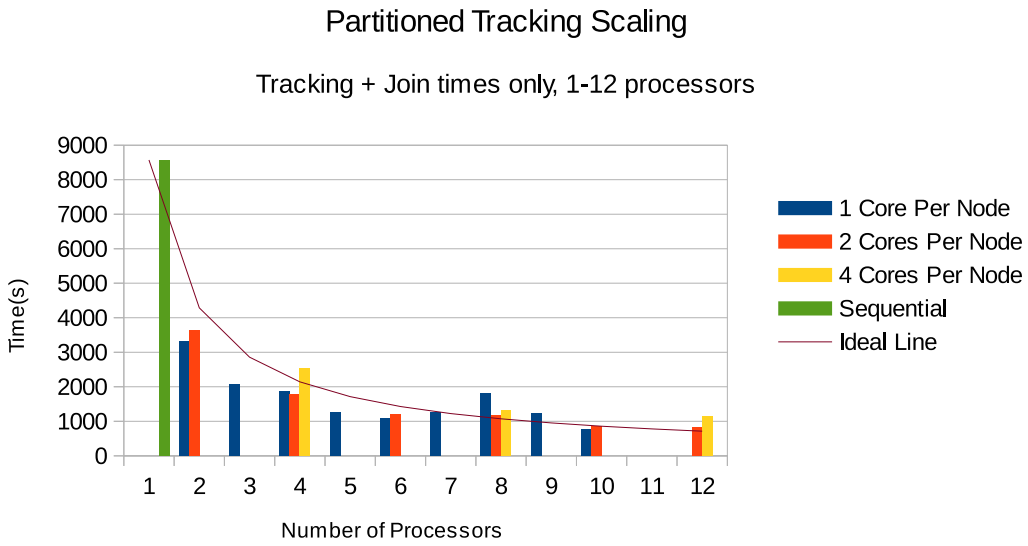

Fig. 12. Partitioned concurrent tracking tracking and join times.

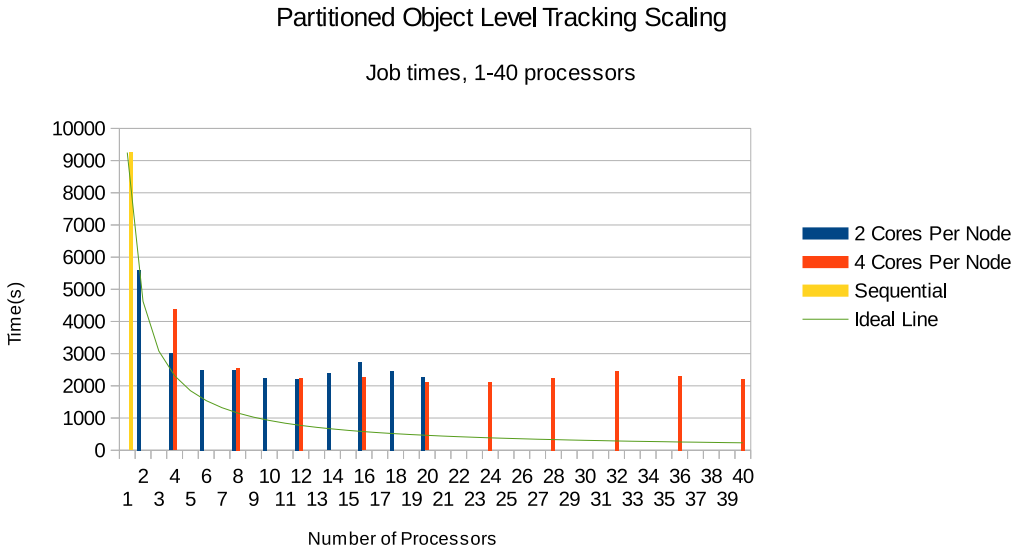

Fig. 13. Partitioned concurrent object level tracking job times.

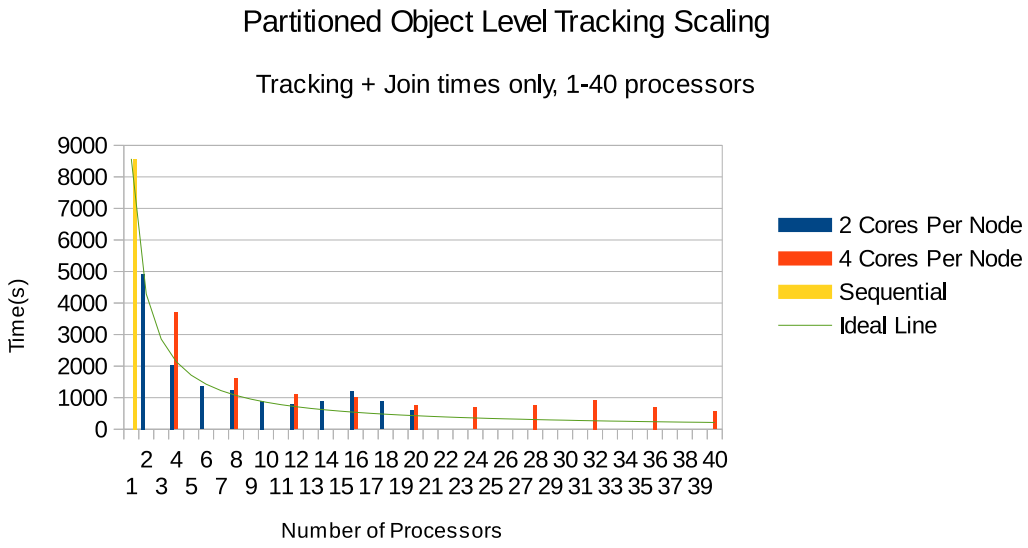

Fig. 14. Partitioned concurrent object level tracking tracking and join times.

6 Discussion and Conclusion

In this study we have adapted image analysis algorithms for use on the LLSC and explored the effect of parallel computing on the performance of these algorithms in HT/HC screening experiments. We have shown that both segmentation and tracking algorithms can be parallelized efficiently, with segmentation scaling linearly up to at least 96 processors using a combination of stack and slice level concurrency. It has been demonstrated that parallelizing object tracking is possible, with the results scaling just below linear time using 4 cores per node and appearing to scale super-linearly when using 1 or 2 cores per node. The results of a sequential run have successfully been reproduced by developing a joining procedure for partitioned concurrent tracking. We have also seen that I/O is a limiting factor and that this problem must be addressed if experiments are to be done on a larger scale.

There are numerous projects that could be done in the future to extend this initial study.

The logical continuation of this study is to merge segmentation and tracking jobs into one job instead of calling them separately. Each stack must be segmented and tracked. There are many different ways to accomplish this. The simplest method is to segment using slice level concurrency on one node, with 4 cores with partitioned tracking on 1 node, 4 cores immediately after segmenta-

tion. This way no extra data needs to be transferred. Another possible solution would be to allocate certain nodes as purely segmentation or tracking nodes respectively. The segmentation nodes could continually segment, and write their output directly to a set of tracking nodes. Since tracking generally has a longer processing time a balance between the number of segmentation/tracking nodes must be found. Further research must be done to find the optimal method.

The simple slice allocator currently implemented should be upgraded to use a more efficient method. Furthermore, the current implementation has no load balancing algorithm in place. Future studies could examine this further as some image stacks are busier than others and will take more time to process. An efficient scheduler could be created to evenly disperse busy stacks.

A related software engineering project could improve the datastructures present in the Java code. It has been shown that I/O is a limiting factor and any effort to decrease this limitation could prove useful. Ideas for projects include efficiently encoding the TIFF stacks and masks, or using a message passing interface to facilitate better inter process communication.

A web interface could be developed to allow easier job submission. At the moment, experiments are prepared and run using a terminal in a Linux environment. The web interface could be used to allow biologists to run their own experiments.

References

1. Zock, J.M.: Applications of high content screening in life science research.
2. Gonzalez, R.C., Woods, R.E: Digital Image Processing (3rd Edition), Prentice-Hall Inc, Upper Saddle River, NJ, 2006
3. Fiji, `http://fiji.sc/Fiji`
4. Yan, K.: Image Analysis and Platform Development for Automated Phenotyping In Cytomics, 2013
5. van Veen, N.: Deploying Single Particle Analysis on the LLSC, 2014
6. Klaver, S.: Deploying and Optimising Electron Tomography with IMOD on the LLSC, 2015