Parallel and Scalable Neural Image Segmentation for Connectome Graph Extraction

by

Quan Nguyen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2015

Certified by . . . .
Nir Shavit
Professor of EECS
Thesis Supervisor

Accepted by . . . .
Albert R. Meyer
Chairman, Department Committee on Graduate Theses

Parallel and Scalable Neural Image Segmentation for Connectome Graph Extraction

by

Quan Nguyen

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Segmentation of images, the process of grouping together pixels of the same object, is one of the major challenges in connectome extraction. Since connectomics data consist of large quantity of digital information generated by the electron microscope, there is a necessity for a highly scalable system that performs segmentation. To date, the state-of-the-art segmentation libraries such as GALA and NeuroProof lack parallel capability to be run on multicore machines in a distributed setting in order to achieve the scalability desired.

Employing many performance engineering techniques, I parallelize a pipeline that uses the existing segmentation algorithms as building blocks to perform segmentation on EM grayscale images. For an input image stack of dimensions 1024 x 1024 x 100, the parallel segmentation program achieves a speedup of 5.3 counting I/O and 9.4 not counting I/O running on an 18-core machine. The program has become I/O bound, which is a better fit to run on a distributed computing framework.

In this thesis, the contribution includes coming up with parallel algorithms for constructing a regional adjacency graph from labeled pixels and agglomerating an over-segmentation to obtain the final segmentation. The agglomeration process in particular is challenging to parallelize because most graph-based segmentation libraries entail very complex dependency. This has led many people to believe that the process is inherently sequential. However, I found a way to get good speedup by sacrificing some segmentation quality. It turns out that one could trade off a negligible amount in quality for a large gain in parallelism.

Thesis Supervisor: Nir Shavit
Title: Professor of EECS

To my family

Thank you for everything.

Acknowledgments

This research was performed under the supervision of Professor Nir Shavit. The project as a whole is joint work with Harvard's Fast Connectomics group.

I would like to thank Nir for being a wonderful advisor. I always find inspiration from his vision, enthusiasm, and humor. Working with Nir has been a fun learning experience.

Many thanks to the Supertech Research Group at MIT for listening to my presentation and providing me with useful feedback. Special thanks go to Prof. Charles E. Leiserson for expressing interest in hearing about my thesis and autographing my laptop.

Thanks to my friends at MIT, especially my fraternity, the Theta Xi Delta Chapter, for the constant support and encouragement.

Thanks to Stephen Plaza at Janelia Farm Research for promptly replying to my emails about NeuroProof.

Last but not least, I would like to thank Tim Kaler, my wonderful mentor. My thesis work would not have been possible without him. He has taught me a whole lot with his extensive knowledge and experience in performance engineering.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Connectomes will mark a turning point in human history ...

- Prof. Sebastian Seung

In recent years, there have been scientific efforts to capture, map, and analyze the neural connections in the brain. The study of nervous systems at the synaptic connections level actually began in the 1970s, but not until the technological advances in the last decade did it receive much general interest. Scientists coined the term "connectome" to refer to the neural connection mapping within the brain, and the science of construction and analysis of connectomes is referred to as "connectomics".

Sebastian Seung, a former MIT professor now working at Princeton, further popularized the term "connectome" with his speech "I Am My Connectome" at a TED conference in 2010 [9]. In his talk, he discusses the theory that maybe aspects of personal identity such as memories, personality, and intellect are stored in the connections between the neurons in the brain. He proposes that it is possible to deconstruct and reconstruct the human brain by obtaining its connectome. According to him, there is no current technology sophisticated enough to map a human connectome in order to test his hypothesis. However, Seung claims that it is enough for the first test of the hypothesis to find partial connectomes of tiny chunks of a mouse brain, a much more modest goal. In fact, many recent technology advances have given scientists a glimpse of the feasibility of constructing a connectome of a small mammal.

In this thesis, we attempt to performance engineer existing software used to perform segmentation on electron microscopy neural images. The state-of-the-art tools currently available lack parallel capability. The contribution of this thesis is introducing parallelism in these tools to make them more scalable when processing very large data sets from the microscope. Technologically, this thesis is part of the first few steps we take towards the theory of connectomics.

1.1 Motivation

Connectomics presents a big data problem due to the large quantity of digital information generated rapidly by the electron microscope. The generation of large data sets is easy, relatively speaking. However, the segmentation process, where pixels in the image stack are labeled according to which object they belong to, presents an interesting performance engineering challenge [7]. The neural objects have unusual shapes. Scientists are not certain about whether or not there is a bound on the number of objects and connections within a volume of brain tissues [7], and that presents a challenge from a scalability point of view.

In this project, we would like to represent the extracted data using a connectivity graph as the final result. This is more complex than many big data correlation problems, where the solution is based on sampling approaches and the output is asymptotically constant. For this problem, the size of the connectivity graph is proportional to the size of the data set because we are representing the entire data set instead of sampling it [7]. This implies that as the data generation rate increases due to improved microscopy techniques, segmentation and graph construction speed needs to improve to prevent bottle-necking.

Fortunately, keeping up with the microscope seems possible with parallel computing. In an article published by Nature Publishing Group, Lichtman, Pfister, and Shavit established some ballpark figures for extracting a connectivity graph from the data set of a small mammal's brain [7]. The brain contains about 20 million neurons and up to 10,000 times as many connections, which would require about 8 terabytes of data to represent the connectivity graph [7]. It is estimated that 500 processors, with 16 gigabytes of memory and 4 cores each, operating at 3.6 GHz would be sufficient to match the electron microscope, which has the generation rate of about 2.5 terabytes per hour. With the resolution of about 5 nm per pixel, this translates to roughly 10,000 machine instructions per pixel.

These calculations demonstrating the feasibility of processing large EM volumes assume, however, that the components of the segmentation pipeline can be efficiently parallelized. Thus, the goal of this thesis is to parallelize the segmentation pipeline so that it is possible to speed up the computation by scaling up the number of CPU's. At the same time, however, we need to keep the number of machine instructions per pixel under 10,000 and maintain the same level of quality.

Based on the number proposed in the paper, we need a tool that could perform segmentation well and at the same time spends no more than 10,000 cycles per pixel. We set up a segmentation pipeline using two state-of-the-art libraries: GALA and NeuroProof. For over-segmentation generation and agglomeration, the total number of cycles reported [1] was 533,299,927,191 and 313,978,336,025, respectively. Dividing the sum by the number of pixels:

$$\frac{313,978,336,025 + 533,299,927,191}{1024 \times 1024 \times 100} \approx 8,080$$

We get about 8,080 cycles per pixel, which is below our bound of 10,000.

Now that we have confirmed the feasibility of doing segmentation using existing software in terms of cycles per pixel, we look for a way to make the computation scalable. For a very large data set, there are two scaling approaches. The first approach is to use multiple machines in a distributed setting, where each machine runs the serial segmentation pipeline. The large data volume from the microscope is split into small volumes that fit in the memory of each machine. There is some

[1] using `perf stat`

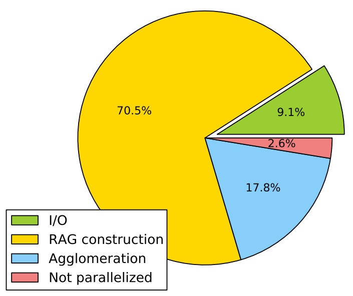
overlap between every two adjacent volumes to allow the results to be stitched together at the end. Although this works well on computations where the runtime is I/O bound, the shortcoming is that it introduces a lot of overhead due to interprocess communication (IPC). Let's take a look at figure 1-1a, which represents the runtime breakdown of the agglomeration step. One can see that I/O is not the bottleneck, so it is more beneficial to utilize more processors to increase the ratio of processors to disks.

Another approach is to parallelize the computation on each machine, where there are multiple processors. Processors in a multicore machine communicate via shared memory, which is much more efficient than IPC. Furthermore, since each multicore machine can operate on a much bigger volume of input data, there is less stitching to be done and thus less overlap. Thus, we make a decision to parallelize the segmentation program until it is I/O bound, at which point we can then further speedup the computation in a distributed fashion like the first approach.

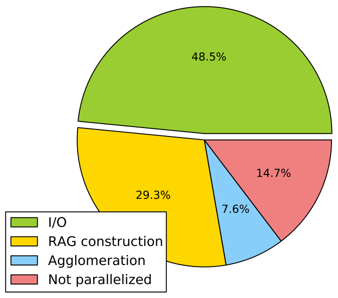
It turns out, however, that parallelizing the segmentation code is quite challenging. Many parts of the program are seemingly unparallelizable. For example, the majority of the agglomeration step involves edges being taken out from a priority queue one by one, and each time an edge is processed it changes the property of other edges that depend on it. This led many people to believe that the code is inherently sequential and that the distributed approach is the only way to gain scalability.

1.2 Contribution

Contrary to what one might expect, it is possible to speed up the program using multiple processors, and in this thesis we show how parallelism is achieved. Nir Shavit's group at MIT works jointly with the Fast Connectomics group at Harvard on connectomics. Shavit is an electrical engineering and computer science professor at MIT. This thesis is part of his group's effort to extract a connectivity graph from a stack of raw images from the electron microscope. The path that turns the data



(a) Serial



(b) Parallelized with 18 cores

Figure 1-1: *Runtime breakdown of agglomeration step in segmentation*

set into a graph is described in Figure 1-2 below. The first step is to perform segmentation and synapse detection. The segmented image stack is then passed to the skeletonization step, which turns it into a graph using the detected synapses to assign connections between the objects.

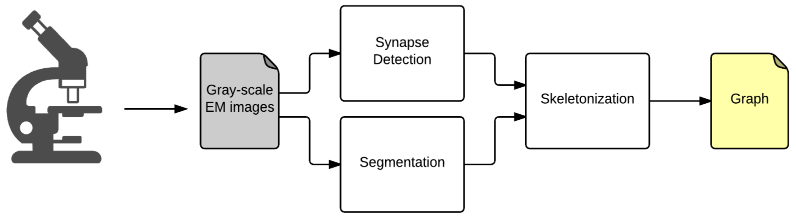


Figure 1-2: *Steps from raw data to connectivity graph.*

The purpose of the thesis is to develop a fully parallel pipeline for segmentation. We are going to achieve that in 2 steps: setting up the pipeline and parallelizing it. In the first step, we assemble together the multiple stages of the pipeline. Here is the list of the stages along with a brief description for each stage:

- The input to the pipeline is a stack of grayscale images from the microscope, as shown in figure 1-4a.
- A probability map is constructed from the input. In a probability map, each pixel is assigned a number representing its likelihood of belonging to a

membrane, or a mitochondrium, etc. Figure 1-4b is a sample of a membrane probability map generated from the image in figure 1-4a.

- Once the probability map is obtained, the over-segmentation is generated. Objects in an over-segmentation are relatively small chunks of pixels clustered together based on the probability map. Figure 1-4c shows an over-segmentation generated from the membrane probability map in figure 1-4b.
- Finally, the small superpixels are merged together if they belong to the same object. The result is the segmentation of the image stack, like in figure 1-4d. The majority of the contribution is in this step, as the others are easily parallelizable.

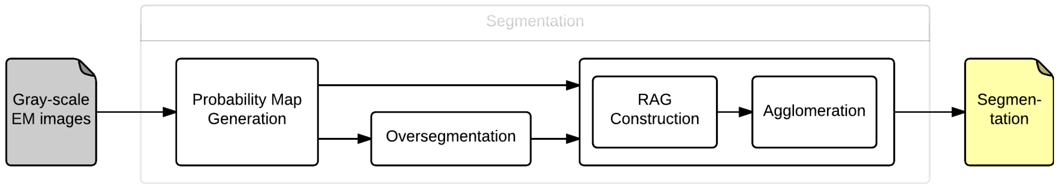
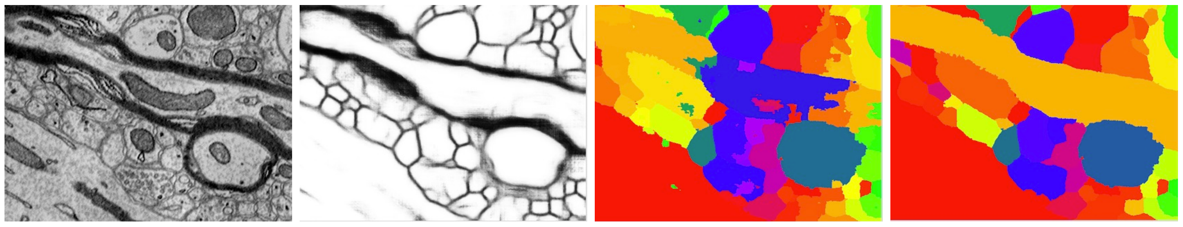


Figure 1-3: *Stages of the segmentation step.*

Once the pipeline is set up, the second step is to unravel the complex dependency inside the segmentation program in an attempt to achieve high scalability by



(a) Raw Image **(b)** Probability Map **(c)** Over-segmentation **(d)** Segmentation

Figure 1-4: *Visualization of segmentation data for all stages: (a) raw image, (b) probability map, (c) over-segmentation, and (d) segmentation.*

parallelizing it. Even though the graphical agglomeration appears to be inherently sequential, it turns out that the agglomeration algorithms used for segmentation can be parallelized by relaxing some sequential constraints. There is a trade-off between parallelism and solution quality inherent in this trade-off, but as we will see in section 1.3 it is possible to achieve usable parallelism without unduly sacrificing solution quality.

1.3 Results

To make the segmentation process more scalable, we parallelized its two main steps: the regional adjacency graph (RAG) construction and the agglomeration. As a result, we were able to achieve very good speedup: 12.5 for RAG construction algorithms and 11.4 for agglomeration algorithms. The overall speedup is 5.3 because it includes I/O. The speedups are measured when running on an 18-core machine with an input size 1024 x 1024 x 100. Unlike the serial program, the parallel program is now I/O bound.

Figure 1-1a shows the breakdown of the total runtime of the original serial code. Within the 104.64 seconds, 7.87 seconds are for reading in the image stack and probability map, and 1.68 for writing the segmented volume. In total, I/O takes about 9.55 seconds. For RAG construction, the first build takes 37.537 seconds and the second 38.180 seconds. For the agglomeration step (merging the edges in the priority queue), the program takes 14.943 seconds.

Figure 1-1b shows the breakdown of the runtime when running with 18 cores on an Amazon Web Service instance. After parallelized, the total runtime of the code is reduced to the point where it is I/O bound. The final runtime is 13.23 seconds, with the RAG construction taking 3.875 seconds and agglomeration process taking 1.004 seconds. This is a large reduction from the initial numbers. More analysis of the speedup and parallel efficiency is done in chapter 7.

As shown in figure 1-1b, the program's runtime has become I/O bound as we desire. Furthermore, the quality of our parallel segmentation result is extremely close to that of the original. The variation of information, a metric we use to measure the quality of the segmentation, reported in chapter 7 is within 0.3% of the initial one, which means we did not give up much in terms of quality. We also managed to stay within the bound in terms of machine cycles per pixel by maintaining the roughly the same number of cycles per pixel.

1.4 Outline

In this thesis, we begin with familiarizing the reader with some frequently used terms and concepts about image segmentation and parallel computing. Chapter 2 contains background information on image segmentation and performance engineering techniques. Chapter 3 talks about two existing state-of-the-art libraries for performing segmentation on electron microscopy neural images. Chapter 4 explain the pipeline from raw images from the microscope to the final segmentation using the building blocks introduced in chapter 3. It also demonstrates that there are known solutions to parallelize all parts of the pipeline except for RAG construction and agglomeration, therefore the remainder of the thesis focus on the parallelization of these algorithms. This sets the stage for the next two chapters, 5 and 6.

In chapters 5 and 6, we describe the parallelization of the two most important components of the pipeline: the construction of regional adjacency graphs and graph-based agglomeration of over-segmented objects. These are the last two stages in the segmentation block in figure 1-3. We choose these steps to apply performance engineering techniques to in this thesis because they are the more challenging parts of the pipeline to parallelize and that currently there is no existing parallel solution. The agglomeration step, for example, uses a priority queue to contain edges that have complex dependency and thus it seems inherently sequential. Thus, parallelizing this code without giving up segmentation quality would be quite an engineering feat.

Chapter 2

Background

The work in this thesis employs many parallel programming concepts and tools. Besides the technical concepts, it is also important to be familiarized with some terminology specific to the segmentation process. This chapter first introduces some terminology used when talking about segmentation, as well as the method to measure the quality of a segmentation. Then, it talks about some concepts that are useful when talking about parallel computing, such as speedup and the work-span model. The performance of a parallel program is described using speedup, and the work-span model is used to predict the runtime of the program.

2.1 Image Segmentation

Segmentation is the process of labeling pixels correctly according to which object they belong to. Each object is assigned a label, and pixels belonging to the same object should be assigned the same label. There are many terms that are useful to know when talking about segmentation.

2.1.1 Terminology

Here are some terms we will encounter in this paper:

Voxel: stands for volume pixel since we are working with 3-dimensional objects.

However, our voxels are flat since we are taking cross-sectional images of the objects, so we will use pixel and voxel interchangeably.

Supervoxel: a group of spatially contiguous voxels with the same label. Sometimes, it is referred to as node or region. Similarly, superpixel and supervoxel will be used interchangeably.

Over-segmentation: the process of initially labeling the raw image stack to create many small supervoxels. The over-segmentation of a data set is often called its watershed, due to the watershed algorithm that is used for this process.

Ground-truth: the segmentation manually done by human. This is considered the correct labeling of objects and is used for training the classifier.

2.1.2 Quality Metrics

For both the training step and the agglomeration step, NeuroProof provides a tool to evaluate the quality of the segmentation: the variation of information (VI) calculator. Variation of information is the metric used to measure how accurate a segmentation is compared to the ground-truth. Let X be a segmentation, Y the ground-truth, and $VI(X,Y)$ the variation of information between X and Y. $VI(X,Y)$ is defined as:

$$VI(X,Y) = H(X|Y) + H(Y|X)$$

Where $H(X|Y)$ is the conditional entropy which quantifies the amount of information required to describe X given Y. If X and Y are identical, the two entropy expressions in the equation are zero, so the variation of information is also zero. Therefore, we would like the entropy expressions to be as small as possible. We can think about $H(X|Y)$ as the amount of over-segmentation, or the split error caused by supervoxels of the same object not being merged together. On the other hand, $H(Y|X)$ can be interpreted as the amount of under-segmentation, or the merge error caused by supervoxels from different objects being merged together [8].

2.2 Parallel Computing

2.2.1 Speedup

In parallel computing, speedup is measured by the ratio of the runtime using 1 CPU to the runtime using n CPU's. Let $S(n)$ be the speedup and $T(n)$ be the runtime, both being functions of the number of CPU's, n.

$$S(n) = \frac{T(1)}{T(n)}$$

$T(n)$ is limited by the amount of code in the program that is strictly sequential, the part that is hard or impossible to parallelize. Let P be the fraction of the program that is parallel, where $P \in [0, 1]$, and $(1 - P)$ the fraction that is sequential. We have what is called Amdahl's Law:

$$S(n) = \frac{1}{(1 - P) + \frac{P}{n}}$$

Note that $S(n) \leq n$. When $S(n) = n$, we call this a linear speedup. When $S(n) < n$, we call this a sub-linear speedup, which is the case for most parallel programs due to some unavoidable inter-process communication that is sequential. When $S(n) > n$, we call this a super-linear speedup. One possible reason for this is the increase in the size of accumulated caches from having many processors, so more data can now fit in the cache.

2.2.2 Work-Span Model

Work-span model is used to estimate the runtime of a program. It gives the times for when running with 1 processor and infinitely many processors as the upper bound and lower bound on runtime. These are T_1 and T_∞ as explained in subsection 2.2.1.

These bounds are given names in this model. T_1 is known as the **work** of the program. It is the time the program would take if it completes all tasks serially.

T_∞ is called the **span** of a program, which is the time it would take a machine with infinitely many processors to finish, executing in parallel. Another way to understand the span is treating the program as a directed acyclic graph, where each node is a task. The span is simply the length of the longest path from the beginning to the end.

Knowing the work and span, one could find the parallelism of the program by dividing the work by the span. This describes the best possible speedup that could be achieved. In other words, a program's parallelism is its capacity to be parallelized. With a greedy randomized scheduler, sufficient parallelism implies good parallel speedup [1].

2.2.3 Cilk Plus

Cilk Plus is an extension to the C++ language that makes utilizing parallel processing and multicore computing power easier for the programmer. Where as manually spawning threads (using *pthreads* for example) forces the program to be multi-threaded, Cilk Plus only requires the programmer to expose the parts of the program that could be executed in parallel. Another advantage is that Cilk Plus is processor-oblivious, meaning the program should run as well as it could on any number of processors [10]. This is due to the scheduler dividing up the work among processors and efficiently performs work-stealing [1]. Thus, the programmer only needs to focus on increasing parallelism in the code without having to consider the number of processors the code is run on.

There are only a handful of Cilk Plus keywords used in this thesis:

`cilk_spawn`: used in front of a function call to tell the runtime system that the function may be run in parallel with the caller. The runtime system is not required to run the function in parallel if there is not enough resources [12].

`cilk_sync`: a stand-alone statement that forces the current function to wait for all its spawned children to finish before continuing.

[1]a scheduling strategy where each processor has a queue of tasks and is allowed to look at others' queues and steal from them in case it runs out of tasks

`cilk_for`: starts a `cilk_for` loop. The syntax is identical to that of a normal C++ `for` loop, but it signals that the iterations in the loop can be run in parallel. The iterations are executed in a divide-and-conquer fashion, which is different than a `for` loop that spawns each iteration [13].

2.2.4 Locks

A lock is a mechanism used to enforce mutual exclusion policies and avoid race conditions when there are many threads executing concurrently. The lock used in this thesis project is `boost::mutex` in the `Boost.Threads` library. This lock class does not have a specific locking strategy due to efficiency purposes, so no assumptions can be made when threads perform uncommon actions such as recursively acquiring a lock or releasing locks that they do not own.

2.2.5 Atomics

There are built-in functions that execute a series of operations atomically supported on x86. Some of them are used in this thesis project:

`__sync_fetch_and_add`: increments a variable and returns its old value. Some other operations other than addition are also supported.

`__sync_bool_compare_and_swap`: performs a compare and swap, meaning if the variable's value is what's expected then overwrite it with a new value. The return value is a boolean indicating whether the swap was successful.

Sometimes, there are inefficiencies with using locks, so it is better to use built-in atomics if the critical section is as simple as incrementing or doing a conditional swap on a shared variable. Contention can happen to both lock-based and lock-free (i.e. using atomics) programs when there are many threads executing. However, consider the case when a thread in a lock-based program acquires a lock, enters the critical section, then is context-switched out by the operating system. All other threads trying to acquire the same lock must wait and experience delays. In a lock-free program using atomics, however, the thread would fail to execute the atomic instruction if context-switched out, but other threads can still go ahead and update the variable [11]. Thus, when there is not much contention, it is more efficient to use atomics instead of locks.

Chapter 3

Related Work

"Finding an entire human connectome is one of the greatest technological challenges of all time. It will take the work of generations to succeed."

- Prof. Sebastian Seung

In this chapter, we talk about two state-of-the-art serial programs for performing image segmentation that we use as a starting point in this thesis. We describe the features they provide and their segmentation approach. The International Symposium on Biomedical Imaging (ISBI) holds a competition on 3-D automatic segmentation of neural electron microscopy images. The challenge is to programmatically produce a segmentation as close as possible to the given ground-truth. At the time when this thesis began, the two teams holding the highest positions on the leader board were GALA and FlyEM, both associated with the Janelia Farm Research group [6]. They achieved excellent segmentation quality using their two open-sourced software: GALA and NeuroProof. We believed these are some of the best software available for image segmentation, so we decided to performance engineer them and use them in our project.

3.1 GALA

GALA stands for "Graph-based Active Learning of Agglomeration", a library for performing segmentation based on the an algorithm described in the paper by Nunez-Iglesias et al [8]. The library performs agglomeration based on a machine learning approach, using features that work for datasets of any size and number of dimensions. For the training process, GALA first runs its watershed algorithm to obtain an over-segmentation of the training input. It builds a regional adjacency graph (RAG) from the over-segmentation, where nodes represent supervoxels and edges represent boundaries. The RAG is modified as regions are merged together. GALA then learns a merge priority function, or a classifier, that determines how likely a boundary between two supervoxels is an actual boundary to decide whether to merge them. Some features used to create this predictive model are the size, the moment, and the histogram of the voxel-level probability within the supervoxels. The information to form these features can be obtained from the probability map, which the GALA team constructs semi-manually using a separate tool called Ilastik. GALA compares the segmentation against the ground-truth as it merges the supervoxels. It does so repeatedly until the segmentation looks close enough to the ground-truth. This process produces a training set, to which GALA fits a classifier.

For the agglomeration after training the classifier, GALA follows the steps similar to those in the training process. It uses Ilastik to construct a probability map, runs its watershed algorithm to obtain the over-segmentation, builds a RAG from the over-segmentation, then agglomerates using the classifier.

Python, being an interpreted language, is not ideal for high performing programs. The Global Interpreter Lock used by the Python interpreter to synchronize multi-threaded executions allows only one thread to run at any time. This presents a major problem when it comes to parallelizing the program. Thus, GALA is not a desirable starting point for our project.

3.2 NeuroProof

NeuroProof is a graph-based segmentation software functionally equivalent to GALA but is implemented in C++. It follows the same machine learning approach as in GALA, as described above. Furthermore, it has a some optimization to improve both the runtime and segmentation quality. One optimization is to expedite the training process by only using less than 20% of the labels instead of the complete ground-truth. This is done using label propagation as described in a paper by Parag, Plaza, and Scheffer on small sample learning [15]. While using less training data, they claim that the classifier trained by this new method is just as accurate as the one learned from the complete ground-truth.

One other optimization in NeuroProof is to improve the segmentation quality. This is done using delayed agglomerative clustering and context-aware segmentation, both described in a paper by Parag, Chakrobarty, and Plaza. Delayed agglomerative clustering postpones the merging of the new boundaries resulted from a recent merge. This approach allows supervoxels to grow larger on average so they generate more discriminative features, which helps to avoid making wrong decisions on merging smaller supervoxels [14]. The other segmentation feature is context-awareness. Assuming there exists a good mitochondria detector, the context-aware segmentation increases the quality even more by performing agglomeration on the non-mitochondria regions first, then merging the mitochondria regions [14]. Most algorithms, like the one used in GALA, are context oblivious, meaning they do not utilize sub-structure distinction (e.g. cytoplasm vs. mitochondria) at the pixel-level when performing agglomeration [14].

NeuroProof, unfortunately, does not have a watershed algorithm implemented in C++ for the over-segmentation generation step. The author suggests using GALA for this step.

Chapter 4

Pipeline

In this chapter, we describe each of the 5 stages of the segmentation pipeline: probability map generation, over-segmentation generation, agglomeration training, agglomeration procedure, and segmentation analysis. The structure of our pipeline mirrors that of NeuroProof and GALA. In section 4.1, we explain in details the tools used for each stage within this pipeline and address the problems they present. In section 4.2, I show the modified pipeline that is used for this thesis.

4.1 Stages of the Pipeline

Here is a brief description for the stages of the pipeline, along with the tools that could be used for each stage:

- Probability Map Generation: the input is the data stack and the output is a probability map, where each pixel is represented by its probability of being a boundary (or other objects). One example of how this could be done is by using the Ilastik tool, which uses a machine learning technique to classify boundaries, in combination with GALA [8].
- Over-segmentation Generation: the input is the probability map and the output is an over-segmentation. Any watershed algorithm can be used for this step, including the serial codes provided by GALA..

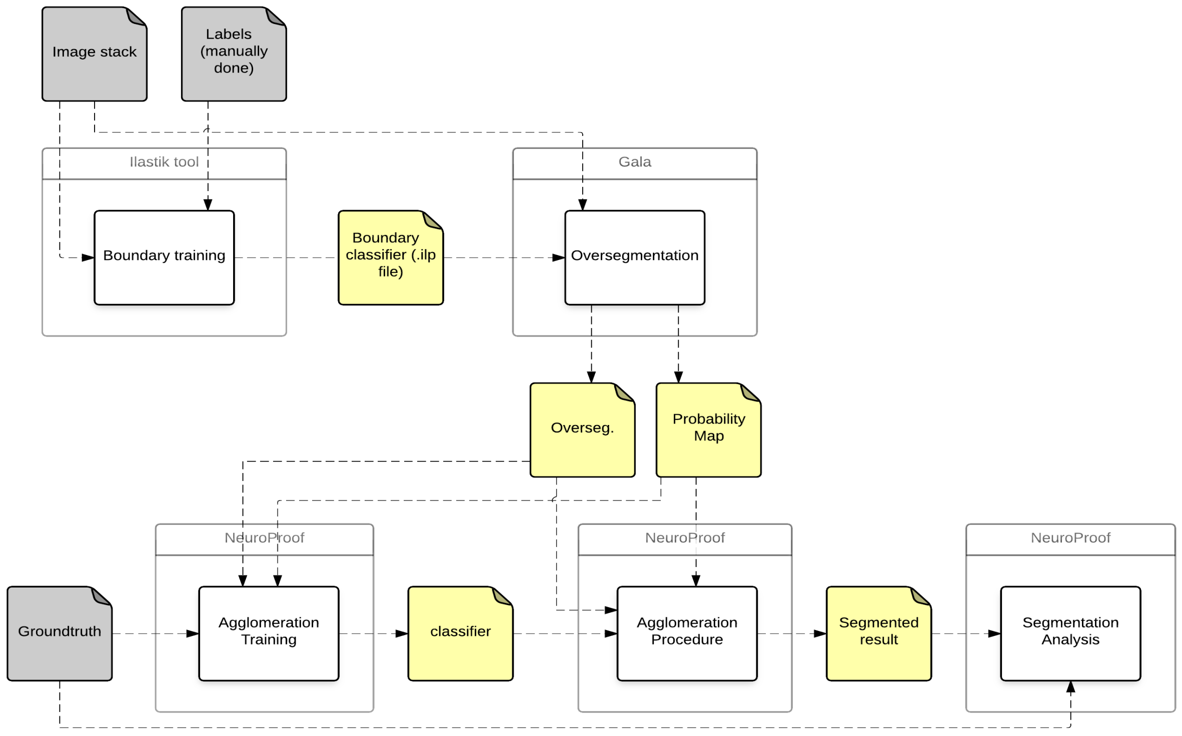


Figure 4-1: *The initial pipeline with gray blocks representing input and yellow blocks intermediate results.*

- Agglomeration Training: the input is the over-segmentation, the probability map, and the ground-truth. The output is a classifier used for agglomeration. Libraries for performing segmentation can be used for this step, including the serial codes provided by NeuroProof and GALA.

- Agglomeration Procedure: the input is the over-segmentation, the probability map, and the classifier from the training stage. The output is the final segmentation. Libraries for performing segmentation can be used for this step, including the serial codes provided by NeuroProof and GALA.

- Segmentation Analysis: the input is the ground-truth and the final segmentation. The output is a pair of numbers representing the errors in merging and splitting objects. This could be done using any variation of information calculator, including the one provided by NeuroProof and GALA.

4.1.1 Probability Map Generation

In the initial pipeline, the probability map is generated using GALA as part of the over-segmentation step. This process requires a boundary classifier which is generated using the Ilastik tool. However, in order for the Ilastik tool to train the boundary classifier, a small section of the raw image set needs to be manually labeled so Ilastik knows the different objects it is classifying. This is a tedious process, which does not fit in with a highly parallel segmentation pipeline. We need another way to automate the probability map generation that is parallelizable and requires no human interaction.

Fortunately, there exists efficient machine learning algorithms to generate the probability map from the EM images without the need for human intervention. Verena Kaynig who works with the Harvard group has provided us with one such algorithm that she worked on [1]. This convolution based algorithm uses a sliding window for membrane classification. We believe it would be trivial to parallelize this algorithm by using multiple windows at once, so we choose not to work on the performance of this code in this thesis.

4.1.2 Over-segmentation

The next step once we have the probability map is to generate an over-segmentation. The over-segmentation is generated by running a watershed algorithm on the probability map. In a watershed algorithm, we use pixel intensity to denote peaks and troughs. Then, a grayscale image can be seen as a topological relief. Metaphorically, labeling pixels in a region of similar intensity is like filling up a trough with water of a certain color. In a typical watershed algorithm, barriers are built to contain the water and prevent different colors of water from merging as the water level rises. This continues until all peaks are under water, and the result is a fine-grained segmentation, or an over-segmentation.

The watershed algorithm is a well-known problem and it should be easy to

[1] `https://github.com/Rhoana/membrane_cnn`

be implemented in C++, so we choose not to parallelize this step. Our group will eventually write our own watershed program. There are known techniques for parallelizing this part of the algorithm by formulating it either as a stencil or data-graph computation [4] [2].

4.1.3 Training and Agglomeration

After an over-segmentation is generated, one can either train a classifier or use an existing classifier to perform agglomeration. For training the classifier, the over-segmentation, probability map, and ground-truth are passed to the agglomeration training step in NeuroProof, which produces a classifier. The ground-truth is provided by the ISBI competition website. For agglomeration, NeuroProof uses the classifier to merge the superpixels in the over-segmentation input. These training and agglomerating procedures are described in section 3.1.

Currently, there is no existing parallel solution to this section of the pipeline, therefore we choose to parallelize the state-of-the-art serial segmentation algorithms mentioned. This is a challenging problem because many of the algorithms used for this step feature a priority queue that stores edges and creates very complex dependencies. We do not spend too much time one the training process because although it would be nice to have a parallel training code, we do not necessarily need a new classifier for every new input.

4.1.4 Segmentation Analysis

The segmentation analysis part of the pipeline computes the quality metric of the segmentation. This code simply reports the variation of information between the ground-truth and the final segmentation. We only use this to test the quality of our segmentation, so there is no need to optimize the performance.

4.2 Modified Pipeline

After making the changes (replacing Ilastik, etc. from the pipeline), the current pipeline is shown in figure 4-2.

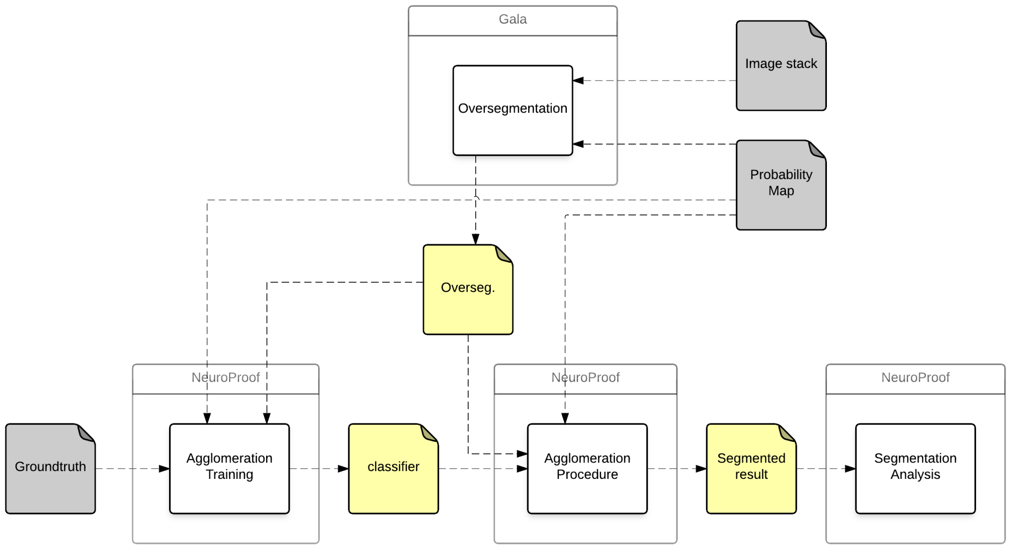


Figure 4-2: *The pipeline as of when this thesis was written. Gray blocks representing input and yellow blocks intermediate results.*

Chapter 5

RAG Construction

In this chapter, we describe and analyze an algorithm for constructing a Regional Adjacency Graph (RAG) from a labeled set of pixels. RAG construction is a key step that precedes the actually merging of superpixels in many graph-based agglomeration algorithms. The nodes of a RAG represent the supervoxels and the edges represent the boundaries between them. Our parallel algorithm has $O(n \log n)$ work and $O(\log^3 n)$ span, where n is the number of pixels. The theoretical parallelism is $\Omega\left(\frac{n}{\log^2 n}\right)$.

In section 5.1, we will talk about how the RAG is constructed serially from an over-segmentation. It turns out that if the over-segmentation is partitioned spatially into smaller volumes, then the serial RAG construction can be applied to the individual volumes to construct a RAG for each of them. We will talk about an algorithm to merge these RAGs together. In section 5.2, we will explain in depth the divide-and-conquer approach used to parallelize the entire process.

Figure 5-1 is an example of a segmentation and its corresponding RAG. For simplicity, let us assume that the data set (left) is 2-dimensional. The numbers are the labels, circles are nodes, and lines are edges representing boundaries. As one can see, objects 1, 2, and 3 share boundaries, and objects 3 and 4 share a boundary. Correspondingly, we see edges between nodes 1, 2, and 3, and one between nodes 3 and 4 in the graph.

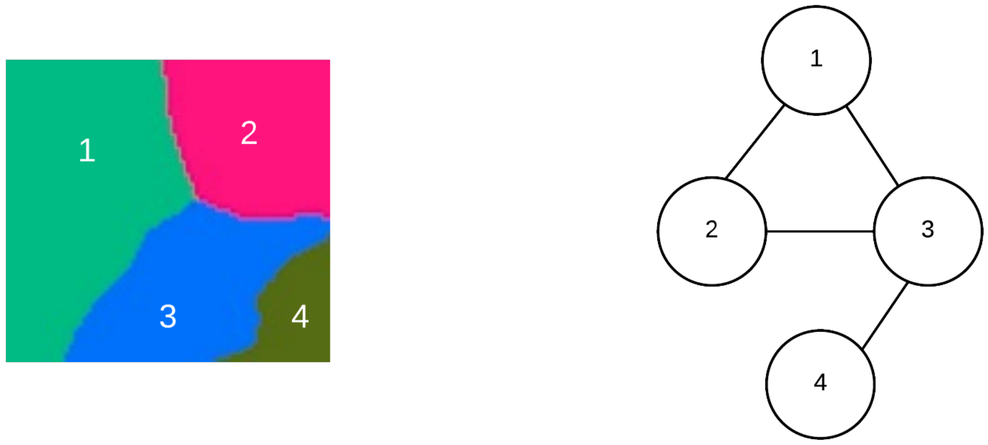


Figure 5-1: *A volume of superpixels and its corresponding RAG.*

5.1 Serial Construction

Let us first see how the RAG is constructed serially in NeuroProof before looking at the parallel construction. The segmentation program in NeuroProof constructs the RAG by iterating over all pixels in the over-segmentation. For every pixel, it checks the pixel's label to see if the object with that label already exists in the RAG. If it does, then it simply increases the size of that object, else it creates a new node in the RAG. Then, the program checks the six adjacent pixels to see if they have different labels from the middle pixel. If they do, the program adds empty nodes for those labels, if they are not already in the RAG, and creates edges connecting them to the node that the middle pixel belongs to.

Lemma 5.1.1 *The complexity of serial RAG construction is $O(n)$, where n is the number of pixels.*

Proof The serial construction iterates over each pixel in the input. For each iteration, insert, update, and lookup elements in the RAG both take constant time, thus the complexity is $O(n)$ ▮

5.2 Parallel Construction

Now that we have described the serial code, let us discuss the parallel implementation of the RAG construction process. A divide and conquer approach would be the right choice for this, since the problem could be broken down into sub-problems of the same type. Each sub-problem could then be processed independently to build a small RAG. In the end, the small RAGs would be merged together to give the right solution to the original problem.

In order to execute this plan, we need a way to merge the small RAGs. In section 5.2.2, we describe an algorithm to merge two RAGs. This algorithm allows us to talk about the divide and conquer approach in section 5.2.3, in which we introduce two recursive algorithms to do the job.

5.2.1 RAG Data Structure

We choose to represent all objects in a RAG using a sorted list. Each edge in the RAG is represented in the list by a pair of numbers representing the labels of the vertices, and each node is represented by a self edge. For example, a RAG with two nodes V_1, V_2 and an edge between them will be represented by this list: $(V_1, V_1), (V_1, V_2), (V_2, V_1), (V_2, V_2)$. Since there are two elements per edge and one element per node in the list, the length of the list is $O(E+V)$. For a connected graph, such as a RAG, this is just $O(E)$.

5.2.2 Merging RAGs

Let us look at an algorithm to merge two RAGs. Figure 5-2 shows the process of merging two RAGs $R1$ and $R2$. The first step is to merge RAG 2's list into RAG 1. Then, when there are two duplicate items, one of them is filtered out. When an item is filtered out, there are two cases. If the two labels are the same, this corresponds to a merge between two nodes. Otherwise, it denotes a merge between two edges. In the figure, the data of node 1, node 2, and the edge between them in RAG 2 are merged with the data of the corresponding objects in RAG 1. Node 4 and its edge are copied over to RAG 1.

Algorithm 1 shows the pseudocode for the merge. First, the sorted lists are merged using a parallel algorithm to merge two sorted lists, such as that employed in parallel merge sort [16, p. 299]. Then, the boolean list $filter$ with the same length as $R1$'s edge list is initialized. This boolean list is used to mark the edges to remove at the end. From line 4 to 11, the algorithm filters out duplicates by marking them in $filter$. Finally, the list of edges in $R1$ is packed so that the elements are contiguous to improve the performance of later accesses.

Algorithm 1 Merging RAG $R2$ into RAG $R1$

```
 1: function MERGE_TWO_RAGS($R1, R2$)
 2:     $R1.edge\_list \leftarrow$ PARALLEL_MERGE($R1.edge\_list, R2.edge\_list$)
 3:     $filter \leftarrow [1] \times length(R1.edge\_list)$
 4:     cilk_for $i$ in range 1 to $length(R1.edge\_list)$ do
 5:         $e_1 \leftarrow R1.edge\_list[i-1]$
 6:         $e_2 \leftarrow R1.edge\_list[i]$
 7:         if $e_1[0] = e_2[0]$ and $e_1[1] = e_2[1]$ then
 8:             MERGE($e_1, e_2$)
 9:             $filter[i] \leftarrow 0$
10:         else
11:
12:         end if
13:     end cilk_for
14:     $R1.edge\_list \leftarrow$ PARALLEL_PACK($R1.edge\_list, filter$)
15: end function
```

Lemma 5.2.1 *The work of algorithm 1 is $O(E)$, where E is the number of edges in* $R1$ *and* $R2$ *combined.*

Proof A parallel merge of two sorted lists has work of $O(E)$ [16, p. 303]. The for-loop has work $O(E)$ since the merging two objects in each iteration takes constant time. The parallel packing algorithm involves a parallel exclusive scan over $filter$ to figure out the positions of the elements in the final list, then the elements are copied over [16, p. 187]. The work of the parallel exclusive scan and copying is

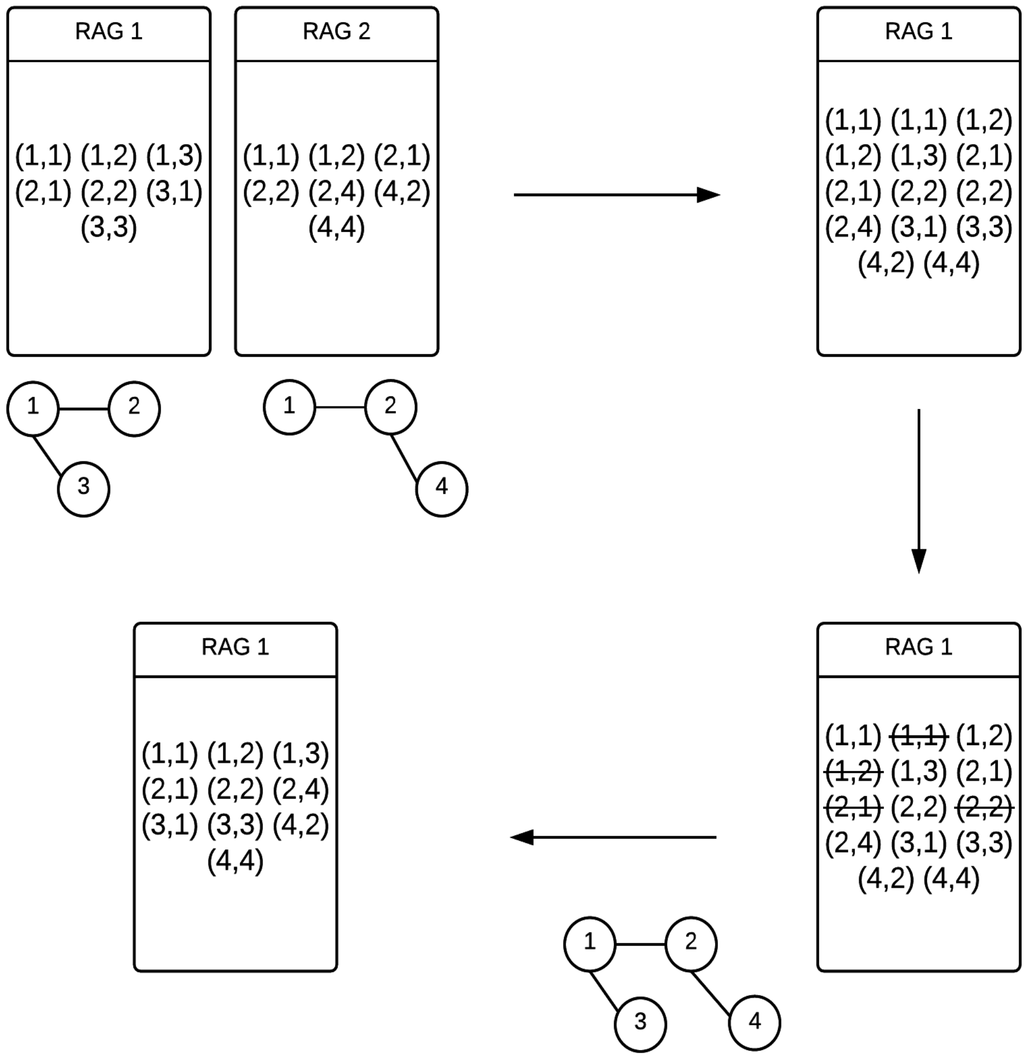


Figure 5-2: *A merge example between two RAGs. RAG 1 would contain all the information after the merge, and RAG 2 is would be deleted.*

$O(E)$ [16, p. 94], so the work of the parallel packing is also $O(E)$. Overall, the work is $O(E)$ ▮

Lemma 5.2.2 *The span of algorithm 1 is* $O(\log^2 E)$*, where* E *is the number of edges in* $R1$ *and* $R2$ *combined.*

Proof A parallel merge of two sorted lists has span of $O(\log^2 E)$ [16, p. 303]. The parallel loop takes $\log E$ to fan out in a divide and conquer fashion. Merging two objects takes constant time. The parallel packing algorithm involves a parallel exclusive scan over $filter$ to figure out the positions of the elements in the final list, then the elements are copied over in parallel [16, p. 187]. Thus, the span of the parallel exclusive scan is also the span of the parallel packing, which is $O(\log E)$ [16, p. 94]. Overall, the span is $O(\log^2 E)$ ▮

According to lemma 5.2.2, the asymptotic span of this algorithm is $O(\log^2 E)$, where E is the number of edges in both $R1$ and $R2$. Since E includes self edges and there is one self each for each node, the merge algorithm can take a long time if there are many nodes and edges in the RAGs.

5.2.3 Divide and Conquer Approaches

The parallel RAG merge algorithm allows us to build RAGs for a labeled volume in parallel using a divide and conquer approach. Two divide and conquer approaches were implemented, both involves spatially dividing the data set. Figure 5-3 demonstrates this approach for two levels of recursion. The pseudocode is described in algorithm 2. For this approach, we keep dividing the volume into two halves along the longest dimension, and keep halving recursively until we have small volumes of 1. Every time a volume is divided, a worker is spawned using `cilk_spawn`. A `cilk_sync` is placed after the two recursive calls. Then, a merge is done at the end of each recursion level to merge the two resulting RAGs.

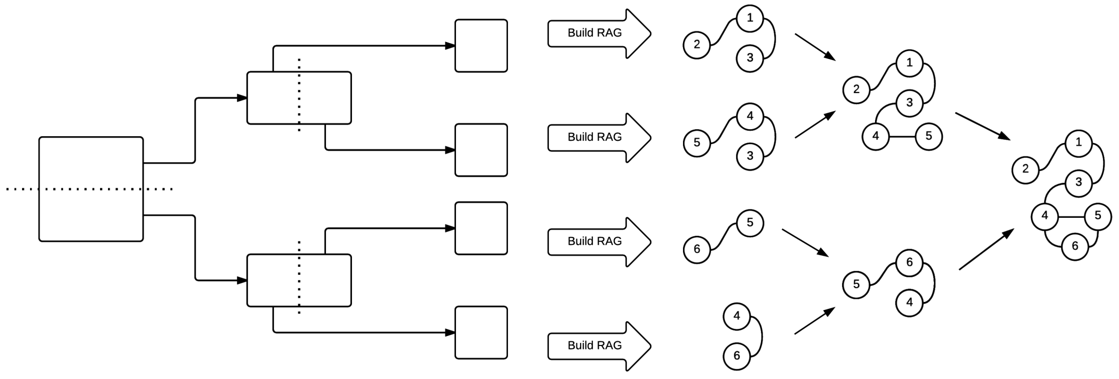


Figure 5-3: *The divide and conquer approach described in 2. For simplicity, assume that the data set has 2 dimensions.*

Algorithm 2 Divide and conquer approach 1

```
function MERGE_RAG_RECURSIVE($vol$)
    if $size(vol) > 1$ then
        $(half\_1, half\_2) \leftarrow$ DIVIDE($vol$)
        $R1 \leftarrow$ cilk_spawn MERGE_RAG_RECURSIVE($half\_1$)
        $R2 \leftarrow$ MERGE_RAG_RECURSIVE($half\_2$)
        cilk_sync
        return MERGE_TWO_RAGS($R1, R2$)
    else
        return BUILD_RAG($vol$)
    end if
end function
```

Theorem 5.2.3 (Work-span analysis of Algorithm 2) *Algorithm 2 has work $O(n \log n)$, span $O(\log^3 n)$, and parallelism $O\left(\frac{n}{\log^2 n}\right)$ where n is the number of pixels.*

Proof The amount of edges, E, in each volume is bounded by the number of pixels. So, $E = O(n)$. The result of dividing a volume is two halves, with each having half the number of pixels of the initial volume. Thus, this algorithm is equivalent to a parallel merge sort, plus the RAG construction from small volumes at the bottom layer. It is equivalent to a parallel merge sort because the merging of two RAGs in algorithm 1 is similar to the parallel merge of two sorted lists. Similar to the parallel merge sort, it has work $O(n \log n)$ and span $O(\log^3 n)$ [16, p. 304]. For RAG construction, the work is $O(n)$ by lemma 5.1.1. Overall, $T_1 = O(n \log n + n) = O(n \log n)$, $T_\infty = O(\log^3 n)$. The parallelism is $\frac{T_1}{T_\infty} = \Omega\left(\frac{n \log n}{\log^3 n}\right) = \Omega\left(\frac{n}{\log^2 n}\right)$ ∎

In practice, the performance of algorithm 2 is not as ideal as suggested: the nodes and boundaries have large dimensions, and thus the small RAGs will have a lot of overlap edges. Lemma 5.2.2 suggests that the the complexity of the merge depends on the number of edges. The overlap would cause the RAG construction to spend more time in the RAG merges and become not as efficient as it could be. Algorithm 2 performs quite many merges, and that hurts the performance.

Thus, we implemented another algorithm aimed to reduce the number of merge function calls, as diagrammed in figure 5-4. Similar to algorithm 2, the data set is divided into small volumes, but this time each worker has its own local RAG. Each worker will get a certain collection of those volumes and build onto its RAG. The RAGs are then merged together at the end.

The pseudocode for this approach is described in algorithm 3. Once again, there is a function that recursively divides the volume, and the small volumes are turned into RAGs at the bottom. However, there is no merging at each level this time. Instead, whichever worker works on a small volume will build it onto its local RAG. Once all the pixels have been processed, the local RAGs are merged into one, just like in algorithm 2. Theorem 5.2.4 shows that algorithm 3 has the same parallelism as the algorithm 2, so it is a suitable replacement in RAG construction.

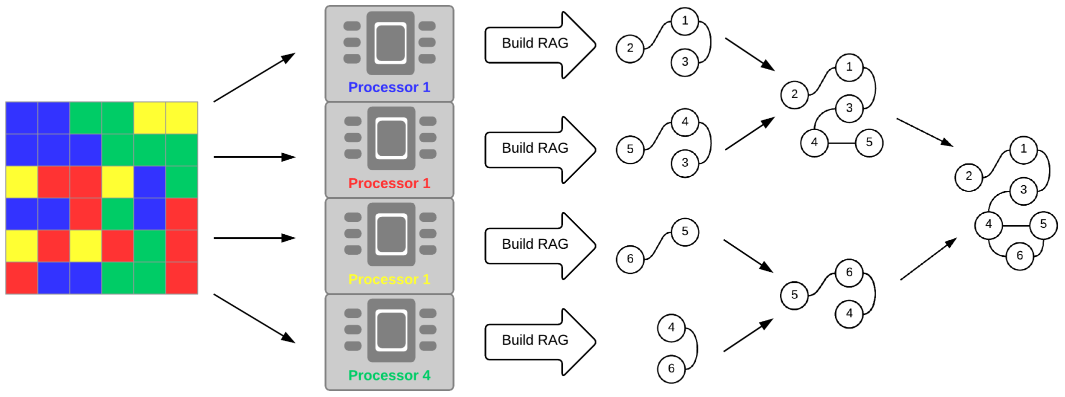


Figure 5-4: *The divide and conquer approach described by 3. The color code tells us which block is processed by which processor.*

Theorem 5.2.4 (Work-span analysis of Algorithm 3) *Algorithm 3 has work $O(n \log P)$, span $O(\log P \log^2 n)$, and parallelism $O\left(\frac{n}{\log^2 n}\right)$ where n is the number of pixels and P is the number of processors.*

Proof The first part of the algorithm is the *build_worker_local_rag_recursive* call that recursively divides the volumes into halves until we have volumes of 1 pixel. This part has work $T_1(n) = 2T_1(n/2) + O(1) = O(n)$ and span $T_\infty(n) = T_\infty(n/2) + O(1) = O(\log n)$.

For the next part, these small volumes are built onto the local RAGs. The work for this is $O(n)$, according to lemma 5.1.1, since we have n pixels. The span is $O(n/P)$ since we have P processors working in parallel.

Algorithm 3 Divide and conquer approach 2

```
Require: $rag\_array$ is the array containing the local RAGs. Its length is equal to the number of workers.
 1: function MERGE_WORKER_LOCAL_RAG($list, start, end$)
 2:     if $end = start$ then
 3:         return $list[start]$
 4:     end if
 5:     if end - start = 1 then
 6:         return MERGE_TWO_RAGS($list[start], list[end]$)
 7:     end if
 8:     $mid \leftarrow (start + end)/2$
 9:     $R1 \leftarrow$ cilk_spawn MERGE_RAG_LIST($list, start, mid$)
10:     $R2 \leftarrow$ MERGE_RAG_LIST($list, mid + 1, end$)
11:     cilk_sync
12:     return MERGE_TWO_RAGS($R1, R2$)
13: end function
14:
15: function BUILD_WORKER_LOCAL_RAG_RECURSIVE($vol$)
16:     $worker\_id \leftarrow$ __CILKRTS_GET_WORKER_NUMBER()
17:     if $size(vol) > 1$ then
18:         $(half\_1, half\_2) \leftarrow$ DIVIDE($vo$l)
19:         cilk_spawn BUILD_WORKER_LOCAL_RAG_RECURSIVE($half\_1$)
20:         BUILD_WORKER_LOCAL_RAG_RECURSIVE($half\_2$)
21:         cilk_sync
22:     else
23:         Construct $rag_array[worker_id]$ using the pixel.
24:     end if
25: end function
26:
27: function BUILD_RAG_WORKER_LOCAL($vol$)
28:     $num\_workers \leftarrow$ __CILKRTS_GET_NWORKERS()
29:     BUILD_WORKER_LOCAL_RAG_RECURSIVE($vol$)
30:     return MERGE_WORKER_LOCAL_RAG($rag\_array, 0, num\_workers - 1$)
31: end function
```

The last part is merging the RAGs together. Since we have P processors, there are $\log P$ levels of these merges. As explained in the proof of theorem 5.2.3, $E = O(n)$ where E is the number of edges. Using lemma 5.2.1 and lemma 5.2.2 to measure the work and span for each level and multiplying with the number of levels, the work is $T_1(n) = O(n \log P)$ and the span is $T_\infty = O(\log P \log^2 n)$

Overall, the work is $O(n + n + n \log P) = O(n \log P)$. The span is $O(\log n + n/P + \log P \log^2 n) = O(\log P \log^2 n)$ since we can have a large P to make n/P constant. The parallelism is $\frac{T_1}{T_\infty} = O\left(\frac{n}{\log^2 n}\right)$ ∎.

In practice, however, the two approaches achieve roughly the same runtime. The reason for this is because, although algorithm 2 has more merges, each merge happens between two adjacent volumes. Two adjacent volumes have many objects in common, and when that happens the merge in the actual implementation simply updates the information about the labels instead of inserting new labels, saving some time. On the other hand, each local RAG in algorithm 3 approach spans the data set more, so they pick up parts of many different objects. As mentioned before, the large number of boundaries and nodes in a RAG can cause the merge to be slow.

Chapter 6

Agglomeration Step

In this chapter, we describe and analyze a parallel agglomeration algorithm used to contract edges in a regional adjacency graph to combine over-segmented supervoxels and produce a final segmentation. The serial agglomeration is described in section 6.1. After building the RAG for the over-segmentation, the trained classifier is used to agglomerate the superpixels until the program believes that all the false boundaries have been removed. The process is broken down into two steps: priority queue initialization and edge processing. In the initialization process, the values of the edges are computed, then the edges are added to the queue. Parallelizing the initialization step involves computing the values concurrently, and it is easily done. This gives good speedup since computing the values is much more time consuming relative to enqueueing the edges.

In section 6.2, we talk about the challenges with parallelizing the edge processing step and present our parallel solution to the problem. The serial dependencies in this algorithm appear to impede parallelization attemps, but we were able to get good speedup by relaxing the dynamism and priority of the edges. Our parallel algorithm achieved bounds $O(\Delta E)$ and $O(E/\log k)$ for work and span, where E is the number of edges in the RAG, Δ is the degree of the RAG, and k is the number of top edges we choose to look at from the priority queue. The parallelism of this algorithm is $O(\Delta \log k)$.

6.1 Serial Agglomeration

Let us understand the serial agglomeration algorithm first before moving on to talking about the parallel agglomeration. Each edge in the RAG has a value associated with it. The value, computed by NeuroProof, represents the probability of the edge being an actual boundary in the data set. Figure 6-1 demonstrates the agglomeration process using a priority queue. First, all edges are inserted into the queue, ordered by their values. Then, the edge with the lowest value is removed from the queue and the nodes that share that edge are agglomerated. In the process of doing so, the RAG is modified. Furthermore, the values of some of the edges in the queue might change, affecting the order. In the actual implementation, the edges that were affected by the merge are marked as "dirty", and their values need to be recomputed when they are dequeued. The program repeats the process, removing one edge at a time in a loop until the queue is empty or the smallest value exceeds a certain threshold.

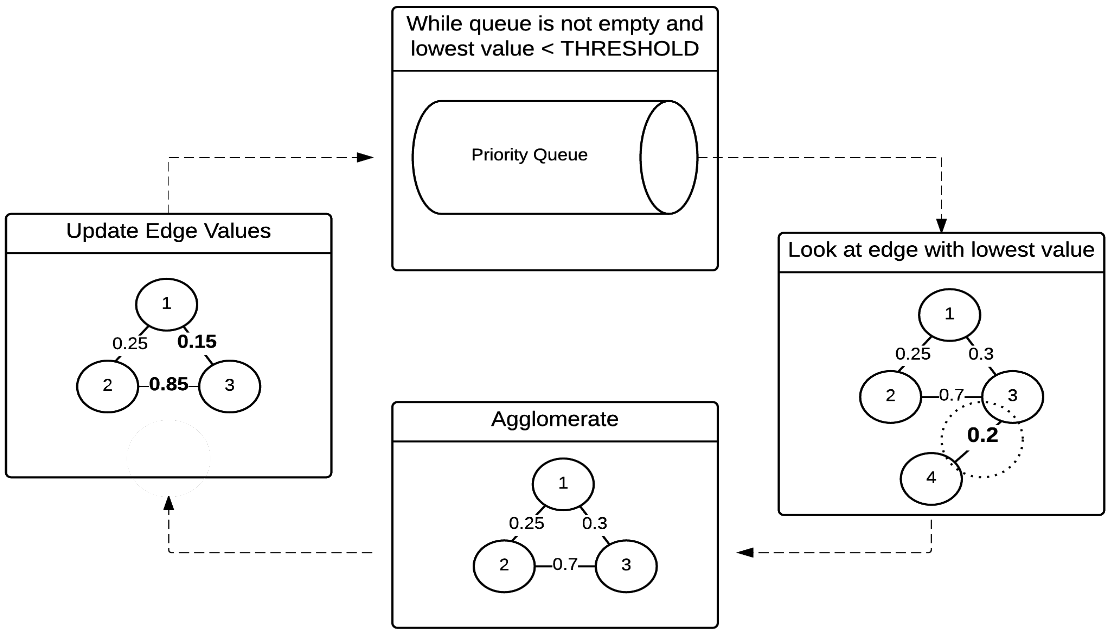


Figure 6-1: Serial agglomeration using a priority queue.

6.2 Parallel Agglomeration

6.2.1 Approach

By inspection, the algorithm for agglomerating over-segmented superpixels might seem to be inherently sequential, meaning we cannot get speedup from running with multiple processors. However, it turns out that we can relax some constraints such as the priority of the edges without sacrificing solution quality.

Relaxing Dynamism

One way to increase parallelism is to process multiple edges during each iteration. Instead of looking at one edge, we look at k edges with the lowest values in the queue. For these edges, we keep their values fixed through out the entire iteration. This means the contracting of an edge [1] cannot affect the values of the edges in this window. Thus, we have relaxed the first constraint of the original code: the dynamism of the probability values.

There is an inherent tradeoff between relaxing dynamism and solution quality. The factor k determines how aggressively we want to trade off dynamism for performance. We demonstrate in section 7.3 in the empirical evaluation how to choose k to maximize speedup without sacrificing too much quality.

Processing the k edges, however, could cause a race condition when two edges share a vertex. We need a way to pick out edges such that we will not have conflicts when relaxing their dynamism and processing them in parallel.

Choosing Independent Edges

Relaxing the dynamism of edge priorities, however, is not alone sufficient to allow for parallelism in an agglomeration algorithm. Within the window of k edges, we need to select only the edges that could easily be processed concurrently with no conflicts. One way to do so is to choose edges that have distance of at least 1 from

[1] contracting an edge is equivalent to agglomerating the two nodes that share that edge.

one another. We call these "D-1 edges" for short. Figure 6-2 shows three different combinations of edges that could be processed in parallel, assuming the window of size *k* contains all edges in the RAG. These are edges that basically do not share a vertex. For the edges that could be processed concurrently, we keep their values fixed through out the entire iteration.

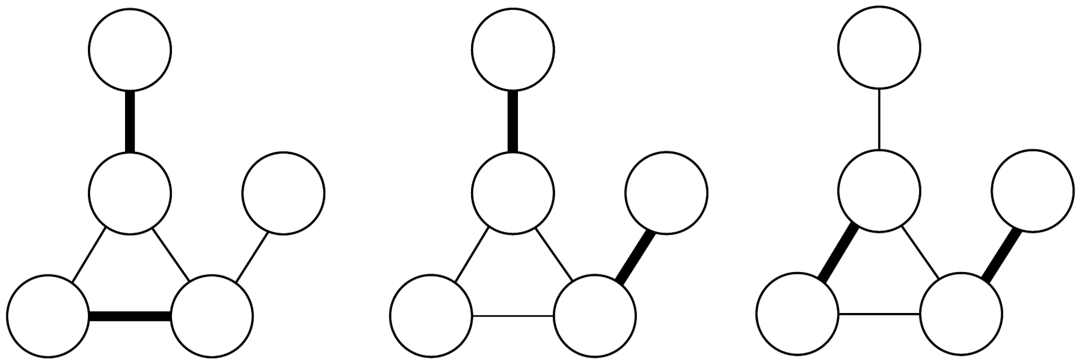


Figure 6-2: *Three possible sets of D-1 edges. D-1 edges are in bold.*

Priority Coarsening

We now have some parallelism and no conflict, but it is not quite enough. The performance could be bad if there are long chains of dependent edges. One way to avoid such chains is by coarsening the priority of the edges in the queue. Another constraint we relaxed was the ordering of the edges within the window of size *k*. The priorities of these edges are randomized so that we could have a provable bound on the number of D-1 edges that could be processed at a time. Since we are using the priorities of the edges to choose the D-1 edges in our implementation, randomization helps avoid long dependency chains and guarantees that there are many D-1 edges to process with high probability. This will be explained more clearly in subsection 6.2.2.

One thing to point out is that these random priorities are only used when determining which edges to process in parallel. If an edge cannot be processed in some iteration, it stays in the queue and retains its actual priority.

6.2.2 Algorithms for Parallel Agglomeration

In this section, we introduce an algorithm for doing agglomeration in parallel. Algorithm 4 shows the pseudocode for the algorithm at a high level. There is a priority Q initialized with all the edges. While it is not empty and the lowest value in the queue is less than a certain threshold, we look at the top k edges. Once the priorities have been randomized, the D-1 edges are selected from the k edges. Then, we build something called the real label lookup table, which we will explain shortly. Finally, using this lookup table we update the RAG concurrently. By theorem 6.2.1, this algorithm achieves work $O(\Delta E)$, span $O(E/\log k)$, and parallelism $O(\Delta \log k)$ where Δ is the degree of the RAG, E is the number of edges in the RAG, and k is the window size.

Algorithm 4 Parallel agglomeration

```
Require: $Q$ is the priority queue containing all edges
Require: $R$ is the RAG
1: INITIALIZE_QUEUE_WITH_EDGES_AND_VALUES($Q, R$)
2: while $Q$ is not empty and lowest value < $THRESHOLD$ do
3:     $top\_k \leftarrow$ GET_TOP_K_EDGES($Q$)
4:     RANDOMIZE_PRIORITY($top\_k$)
5:     $d1\_edges \leftarrow$ GET_D1_EDGES($top\_k$)
6:     $lookup\_table \leftarrow$ BUILD_REAL_LABEL_LOOKUP_TABLE($d1\_edges$)
7:     UPDATE_RAG_FROM_REAL_LABEL_LOOKUP_TABLE($R, lookup\_table$)
8: end while
```

D-1 Edges

To select the D-1 edges from k edges, we use an array of priorities, where each location represents a node. Each edge will use the compare-and-swap atomic function to write its priority to the two nodes it connects. With priorities at the nodes initialized to 0, each edge will attempt to do a Priority-Write-Max, as described in algorithm 5. This is an atomic swap until successful, while the value at the location being written to is less than the priority of the edge. Basically, we overwrite lower value with higher value if they fall in the same location.

Algorithm 5 Priority-Write-Max

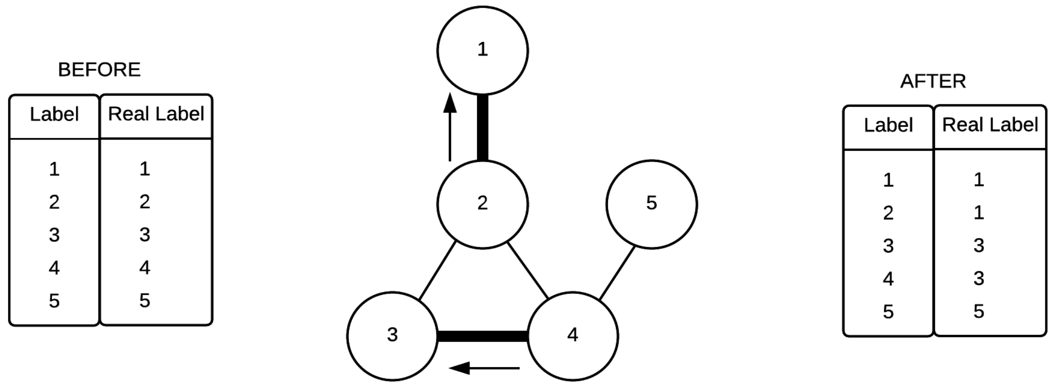
```
1:  function PRIORITY_WRITE_MAX(loc, prior)
2:      old_val ← loc
3:      if old_val < prior then
4:          return
5:      end if
6:      while !__SYNC_BOOL_COMPARE_AND_SWAP(&loc, old_val, prior) do
7:          old_val ← loc
8:          if old_val < prior then
9:              return
10:         end if
11:     end while
12: end function
```

All k edges do this concurrently. At the end, we go through the k edges and allow an edge to be processed if its two corresponding nodes have the same value. While it should be the case that edges with lower values should be processed first, we are randomizing these priorities so the order does not matter. We do this to get a large combination of D-1 edges with high probability. Without randomization, some adversary would be able to cause a pattern when writing priorities to nodes such that the number of D-1 edges found at the end is minimized.

Real Label Lookup Table

One thing we used to implement concurrent modifications to the RAG was the real label lookup table. It allows us to perform lazy updates on the RAG. The real label lookup table is a look-up table that maps a node's label to its real label, specified by the edge associated with the merge. For example, let's look at figure 6-3.

Before the lazy updates, each node's real label is just its own label. Then, agglomerating the top edge changes node 2's real label to 1. Similarly, agglomerating the bottom edge changes node 4's real label to 3. Notice that the real label is not necessarily the label that the node will assume at the very end, since we can have a situation where node 3 is merged into mode 2, and node 2 into node 1. Fortunately,



BEFORE

Label	Real Label
1	1
2	2
3	3
4	4
5	5

AFTER

Label	Real Label
1	1
2	1
3	3
4	3
5	5

Figure 6-3: *How the real label lookup table is used to represent the agglomeration of node 2 into node 1 and node 4 into node 3. The arrows represent the direction of the merges. $X \leftarrow Y$ means Y is absorbed into X. Before the merge, each label belongs to itself. After the merge, a label's real label denotes the object that it now belongs to.*

because we are only using D-1 edges, this situation cannot happen, and it is safe to say that the real labels are the final labels.

RAG Update

The real label lookup table gives us enough information to update the RAG. The pseudocode showing how the RAG modification based on the real label lookup table is described in Algorithm 6. Figure 6-4 shows a simple example to demonstrate how the algorithm works. Figure 6-5 shows a slightly different scenario where the edges are merged into an existing one instead of a new edge.

The pseudocode is broken down into two phases: RAG modification and node deletion. The RAG modification is from line 1 to line 17, which contains a for-loop. In this loop, we go through the real label lookup table and check each pair of key and value. If they are different, we know that this indicates a merge. In lines 5 to 15, we iterate over the neighbors of the node that will be removed. For each neighbor, we check if the RAG has an edge between the node we are keeping and the neighbor's real label. If it does, we add the edge to the RAG R, else we merge

Algorithm 6 Update RAG R from real label lookup table M

```
1:  cilk_for $(k, v)$ in $M$ do
2:      if $k \neq v$ then
3:          $node\_remove \leftarrow k$
4:          $node\_keep \leftarrow v$
5:          cilk_for $n$ in $node\_remove$'s neighbors do
6:              if $n \neq node\_keep$ then
7:                  $node1 \leftarrow \textsc{min}(node\_keep, M[n])$
8:                  $node2 \leftarrow \textsc{max}(node\_keep, M[n])$
9:                  $e \leftarrow edge(node1, node2)$
10:                 if $e$ is not in $R$ then
11:                     $R \hookleftarrow \square$ $\textsc{new}(e)$
12:                 end if
13:                 $\textsc{merge}(R.data(e), data(e))$
14:             end if
15:         end cilk_for
16:     end if
17: end cilk_for
18:
19: cilk_for $(k, v)$ in $M$ do
20:     if $k \neq v$ then
21:         $R.remove(k)$
22:     end if
23: end cilk_for
```

with the existing edge. For the deletion of the nodes, in lines 19 to 23 we iterate through the real label lookup table again to find the pairs of different key and value. For these keys, the label is removed from the RAG by removing the incident edges first, then the node.

The loops are parallelized using `cilk_for`. We use atomic instructions when updating information concurrently. For example, when merging the size feature we use `__sync_fetch_and_add` since it just involves incrementing integers. For features where floating point numbers are involved, our compiler does not support atomic operations for the data type so we use a lock instead.

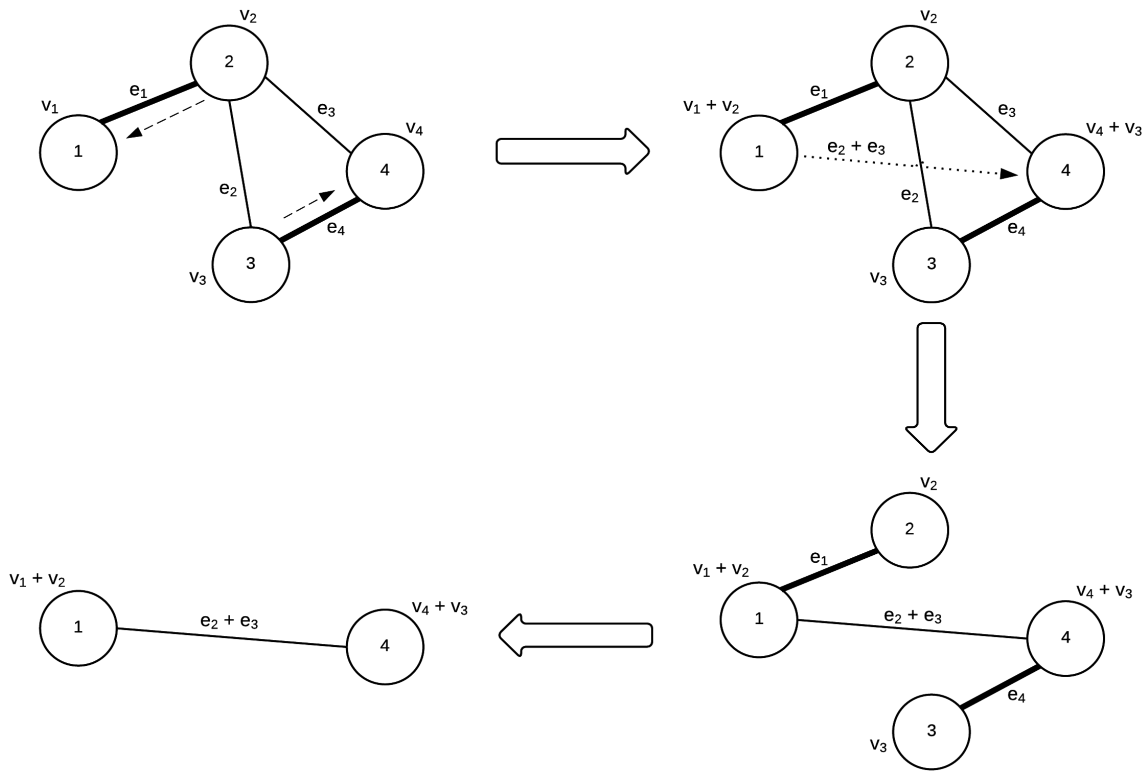


Figure 6-4: *How the RAG changes with parallel merges. Dashed arrows represent merges, dotted arrows represent new edges, and the small variable next to each object represents its data.*

Theorem 6.2.1 (Work-span analysis of Algorithm 4) *Algorithm 4 has work $O(\Delta E)$, span $O(E/\log k)$, and parallelism $O(\Delta \log k)$ where Δ is the degree of the RAG, E is the number of edges in the RAG, and k is the window size.*

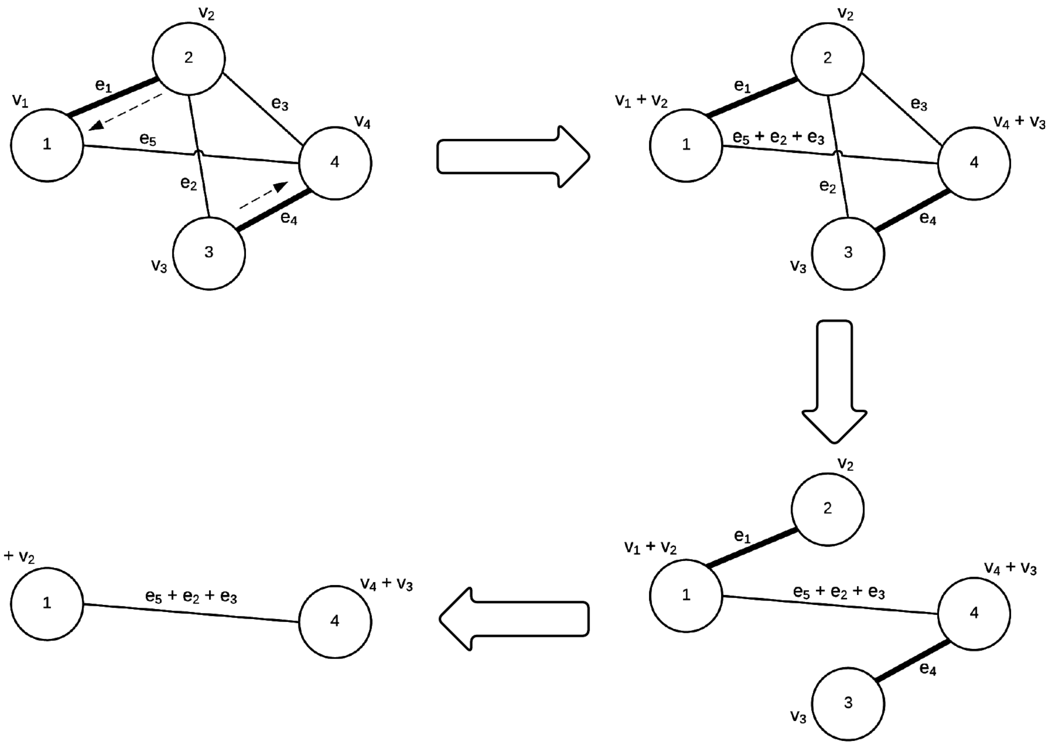


Figure 6-5: *How the RAG changes with parallel merges. Dashed arrows represent merges, dotted arrows represent new edges, and the small variable next to each object represents its data.*

Proof For any k edges, there is a high probability that we get $O(\log k)$ D-1 edges using randomized priority. This is the result of a simple proof that shows with low probability, there is a dependency chain of length $\Theta(\log k)$ in a DAG with k nodes that have randomized priority. We choose to omit this proof, since it is analogous to those bounding dependency chains in other graph algorithms [3].

Initializing the priority queue involves calculating the values of the edges in parallel, which has work $O(E)$ and span $O(1)$ since computing a value takes constant time [2]. Then for the while loop, there are $E/\log k$ iterations total since each iteration processes roughly $\log k$ edges from the queue. In each iteration, getting the top k edges, randomizing the priority, and getting the D-1 edges all have $O(k)$ work and $O(1)$ span. Building the lookup table with $O(\log k)$ D-1 edges takes $O(\log k)$ work

[2]we are working with very basic features

and $O(1)$ span.

For the parallel update, we refer to algorithm 6. The modification loop (first loop) has work $O(\Delta \log n)$ and span $O(1)$ since we are iterating over the D-1 edges and checking the neighbors all in parallel. The deletion loop deletes the edges for the node then the node itself, so similarly it has work $O(\Delta \log n)$ and span $O(1)$.

Overall, the work is $T_1 = O(\Delta E)$. The span is $T_\infty = O(E/\log k)$. The parallelism is $T_1/T_\infty = O(\Delta \log k)$ ∎

Chapter 7

Empirical Evaluation

In this chapter, we present empirical analysis exploring the performance characteristics of the parallel RAG construction and graph agglomeration algorithms described in chapters 5 and 6. In sections 7.2 and 7.3, we give the speedups from the RAG construction and agglomeration step and explain the characteristics the result. In section 7.4, we talk about the performance of the whole program in terms of runtime and quality.

7.1 Experimental Environment

For data collection, we used two machines: the texttcloud2.csail.mit.edu machine that has 12 processors and an AWS c4.8xlarge instance that has 18 processors with hyper-threading disabled. The CSAIL cloud machine is an Intel Xeon X5650 , 2 sockets, 6 CPUs per socket, 12.3M Cache, 2.67GHz. The AWS instance is an Intel Xeon E5-2660 v3, with 2 sockets, 10 CPUs per socket (1 unused), 25M Cache, and 2.60 GHz. For collecting the data showing the effect of priority randomization, we used the CSAIL cloud machine. For the rest, we used the AWS instance. The compiler used was GCC 4.8. The program was compiled with optimization flag -O3.

The data set has dimensions of 1024 x 1024 x 100, which is a stack of 100 images, each being 1024 x 1024 pixels. We used the training data provided on the

ISBI website and the corresponding probability map [6]. The ground-truth used to train the classifier was also provided by ISBI. We used GALA to generate the over-segmentation. The agglomeration type option used is 1, which uses delayed agglomerative clustering.

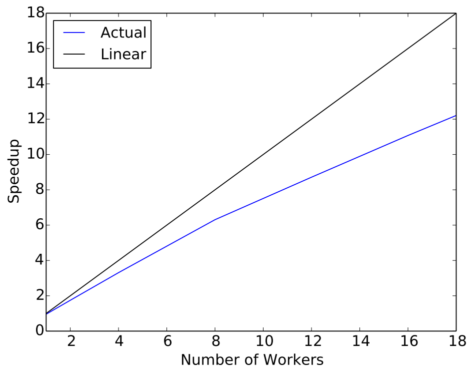
The initial merge error and split error are 0.211234 and 0.774839, respectively. These errors were measured with NeuroProof's variation of information calculator.

7.2 RAG Construction

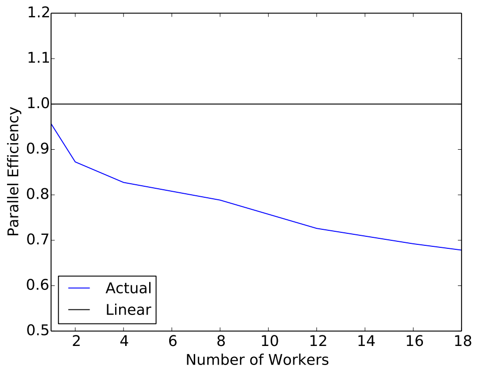
In our implementation, we only divide the input volumes until the small volumes have dimensions that are not too small specified by some coarsening factor. This is to mitigate overhead when the RAG construction is less computationally expensive than the function call, caused my the input being too small. We found that 50 x 50 x 50 cubes work best for the small volumes.

Figure 7-1 shows the speedup and parallel efficiency of the RAG construction algorithm 2 that does not use the worker-local RAGs. The numerical data could be found in the appendix, table A.1. The parallel efficiency is found by dividing the speedup by the number of workers. As one can tell from the parallel efficiency graph in 7-1b, there is an inflection point at 9 workers. This corresponds to the speedup being decent up to 9 workers in figure 7-1a, then it becomes less ideal after that. One possible reason for that is because the CPUs in the machine are split into two groups, each connected to the same socket. Since there are only 9 CPUs we can use on the same socket, the use of more than 9 CPUs introduces additional communication overhead between two different sockets.

Figure 7-2 shows the speedup and parallel efficiency of the RAG construction algorithm 3 that uses the worker-local RAGs. Similar to figure 7-1, there is a bend in the speedup curve and also an inflection point in the parallel efficiency curve at around 9 workers. The explanation is the same as that for figure 7-1. Notice that the data points are more noisy because each worker-local RAG picks up a random collection of small volumes. The size of each worker-local RAGs thus depends on



(a) Speedup



(b) Parallel efficiency

Figure 7-1: *Speedup and parallel efficiency for parallel RAG construction running algorithm 2 (without worker-local RAGs) on Intel Xeon E5-2660 v3, 2 sockets, 10 CPUs per socket (1 unused), 25M Cache, 2.60 GHz. Parallel efficiency is calculated by dividing the speedup by the number of workers.*

the locality of the volumes it picks up.

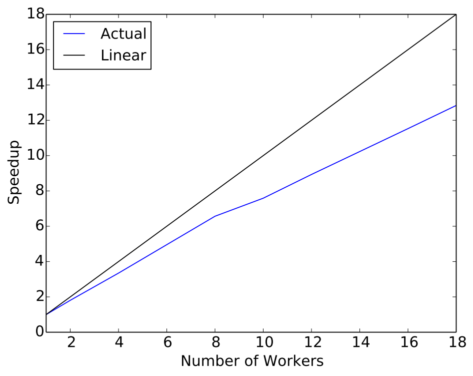
7.3 Agglomeration Step

7.3.1 Effect of Priority Randomization

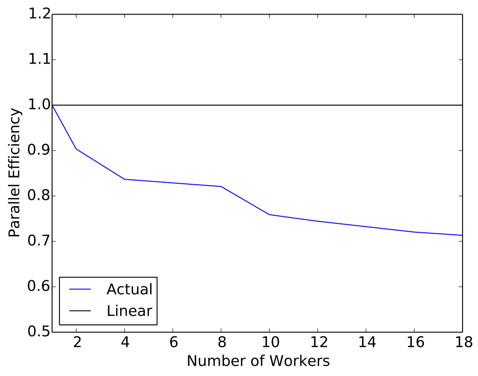
Let us examine the effect of randomization on the runtime. Figure 7-3b shows the runtimes for the while-loop in the agglomeration algorithm, algorithm 4, with and without the priority randomization. For a large range of values for k, the algorithm with randomized priority achieved noticeably better runtime.

One drawback of randomization is the cost to generate the random priorities. Because of this randomizing overhead, one cannot see a clear advantage of using randomized priority in figure 7-3b. Figure 7-4 provides better insight into the effect of randomization. As k becomes larger, the randomization effect becomes more apparent. The number of iterations is much lower for the randomized algorithm, which means more edges are being processed on average per iteration.

Now that we understand the effect of randomized priorities on the performance, let us analyze its impact on the segmentation quality. Figure 7-3a shows the normalized merge and split errors for both randomized and non-randomized versions

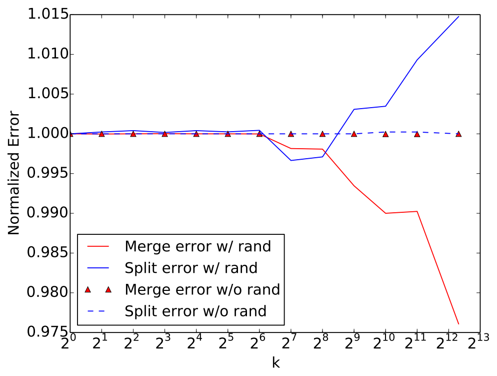


(a) Speedup

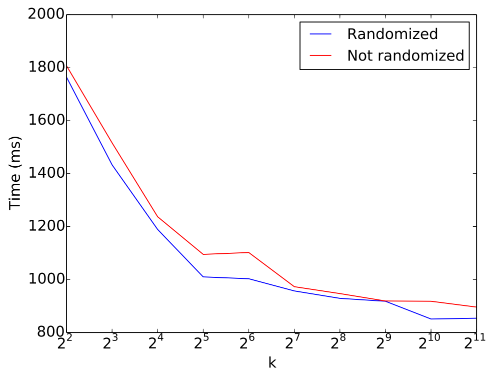


(b) Parallel efficiency

Figure 7-2: *Speedup and parallel efficiency for parallel RAG construction running algorithm 3 (with worker-local RAGs) on Intel Xeon E5-2660 v3, 2 sockets, 10 CPUs per socket (1 unused), 25M Cache, 2.60 GHz. Parallel efficiency is calculated by dividing the speedup by the number of workers.*



(a) Quality vs. *k*



(b) Runtime vs. *k*

Figure 7-3: *Quality and runtime vs. k with 18 workers. The algorithm was run with 12 CPUs on Intel Xeon X5650 , 2 sockets, 6 CPUs per socket, 12.3M Cache, 2.67GHz*

of the algorithm. Normalized error is the ratio of the error obtained from using a certain k to the original error. As one can see, the difference is not noticeable, unless k is very large ($2^9 orabove$). Even then, it is not that much of a difference. At $k = 2^{12}$, the error only differs by 2% from the original.

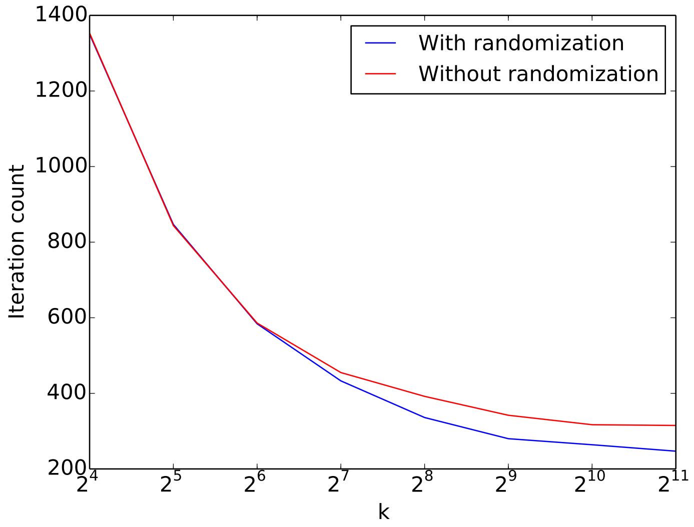


Figure 7-4: *Queue iteration count versus k. The algorithm was run with 12 CPUs on Intel Xeon X5650 , 2 sockets, 6 CPUs per socket, 12.3M Cache, 2.67GHz*

7.3.2 Effect of Window Size k

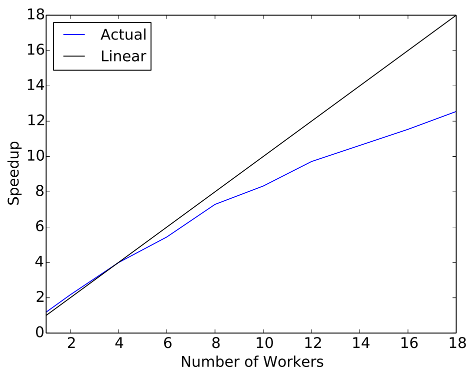
Let us demonstrate how the agglomeration program scales as we increase the window size k, while randomizing the priority of edges within the window. Figure 7-4 shows the reduction in the number of dequeue iterations for this particular data set. Every time k is doubled, the number of iterations is halved due to roughly twice as many edges are processed concurrently per iteration. Notice that once k passes a certain threshold, the rate at which the number of iterations goes down decreases. This might be because there are large objects in the data set, which bottlenecks the number edges can be processed at a time. Each large object is a node with a high degree in the RAG, and thus only one out of these many edges could be processed per iteration.

We need to choose a window size k to optimize the runtime while still maintaining a high quality for the segmentation. Again, we look at the plot of normalized error versus k in figure 7-3a. As we increase k, the merge error tends to go down and split error tends to go up. This is because as we trade off dynamism, many edge values are not being updated, which results in the edges not being merged. When k exceeds $256 = 2^8$, the quality begins to deteriorate significantly. We suspect that 256 is the optimal number to use in terms of quality. For runtime, let's look at figure 7-3b. As one can see, $k = 256$ is in the saturation region of the randomized graph. That means increasing k any further only results in a marginal gain improvement in terms of runtime.

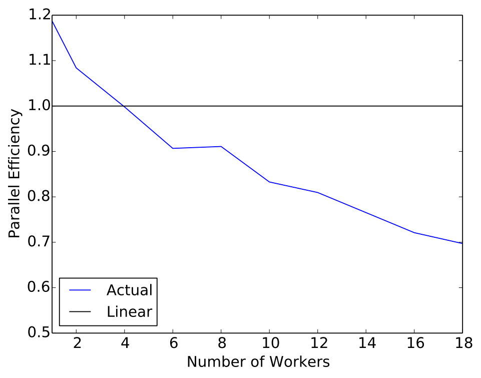
7.3.3 Speedup

Fixing $k = 256$ and using priority randomization, we run the segmentation step (initialization and edge processing) on the AWS instance to obtain the speedup and parallel efficiency plots in figure 7-5. The numerical data can be found in the appendix, table A.3. The speedup is great at first. As the number of workers increases, the speedup becomes less and less ideal. One plausible reason for this is because there are not enough D-1 edges for all workers to work on concurrently if we have many workers. Also, there is a drop in parallel efficiency we can observe at the point of 9 workers. Similar to the curves of the RAG construction, this is due to the increase in communication overhead when we start to have workers from different sockets communicating.

One thing to keep in mind is that we are only working with simple features such as size, moment, and histogram of the object. For more sophisticated features, we will have to take a look to determine whether the constraints could be relaxed. We believe, however, that it should always be possible.



(a) Speedup



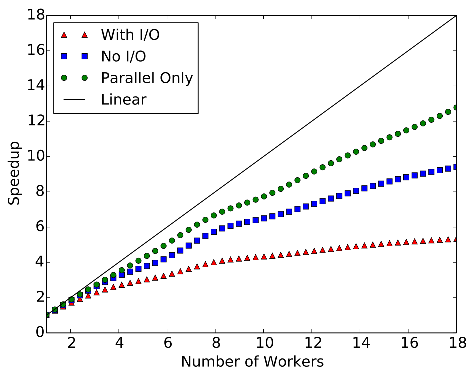
(b) Parallel efficiency

Figure 7-5: *Speedup and parallel efficiency for agglomeration running on Intel Xeon E5-2660 v3, 2 sockets, 10 CPUs per socket (1 unused), 25M Cache, 2.60 GHz. Parallel efficiency is calculated by dividing the speedup by the number of workers.*

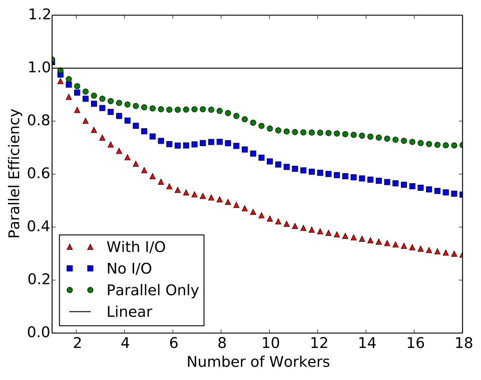
7.4 Overall Performance

To show the performance characteristics of the program, we show the time when it begins reading in the over-segmentation and ends when it outputs a segmented volume, a `.h5` file. We measured time using `/usr/bin/time` and reading the `system` field. The initial time was 124.044 seconds. After inspection, it appears that the serialization of the RAG was only used for analysis purposes, so it was removed the program. This modification reduced the time down to 104.64 seconds, this was used as the initial time to improve on. The RAG construction and agglomeration take up the majority of the time, so they are the targets of our parallelization. I/O time could not be improved since it relies on the I/O performance of the hardware. There are also other serial portions of the program that are hard to parallelize such as initializing the classifier, etc.

The code is at `https://github.mit.edu/qdnguyen/NeuroProof`. With both the RAG construction and agglomeration steps parallelized, the majority of the NeuroProof segmentation code is parallel. Figures 7-6 shows the speedup and parallel efficiency running on the AWS instance. The numerical data can be found in the appendix, table A.4. The curves in the figure represent the speedup counting I/O, not counting I/O, and counting the parallel sections only. Similar to the speedup



(a) Speedup



(b) Parallel efficiency

Figure 7-6: *Overall speedup and parallel efficiency running segmentation on Intel Xeon E5-2660 v3, 2 sockets, 10 CPUs per socket (1 unused), 25M Cache, 2.60 GHz. Parallel efficiency is calculated by dividing the speedup by the number of workers.*

curves of the RAG construction and segmentation steps, the parallel efficiency for the "parallel sections only" decreases after reaching the inflection point at 9 workers.

With $k = 256$, we get 0.211636 and 0.777272 for merge error and split error, respectively. These are still very close to the original numbers, within 0.3%.

Chapter 8

Conclusions

Some day a fleet of microscopes will capture every neuron and every synapse in a vast database of images. And someday, artificially intelligent super computers will analyze the images without human assistants to summarize them in a connectome. I do not know but I hope I will live to see that day.

- Prof. Sebastian Seung

Constructing a connectome is essentially a big-data problem, and thus it is very important to have scalable computing systems for the job. In this thesis, we performance engineered the NeuroProof segmentation code to allow execution of multiple processors in parallel. This gives us the ability to scale up the rate at which raw data are processed by scaling up our resources, in this case the number of processors.

By processing multiple edges from the priority queue at a time in the agglomeration step, we transformed a program that we thought was inherently sequential into a more flexible one, capable of being parallelized. We also demonstrated that it is possible to trade off some quality for performance by choosing the appropriate k value.

One thing our group will keep in mind moving forward is input down-sizing. If we could scale down the resolution of the input without losing important data like membranes, this would reduce the runtime significantly. This is because a large portion of the runtime is spent in RAG construction, and the fewer pixels we have, the faster the construction process. We show in figure 8-1 that the segmentation results from 1024 x 1024 (original) and 512 x 512 (down-sized) data sets are almost identical.

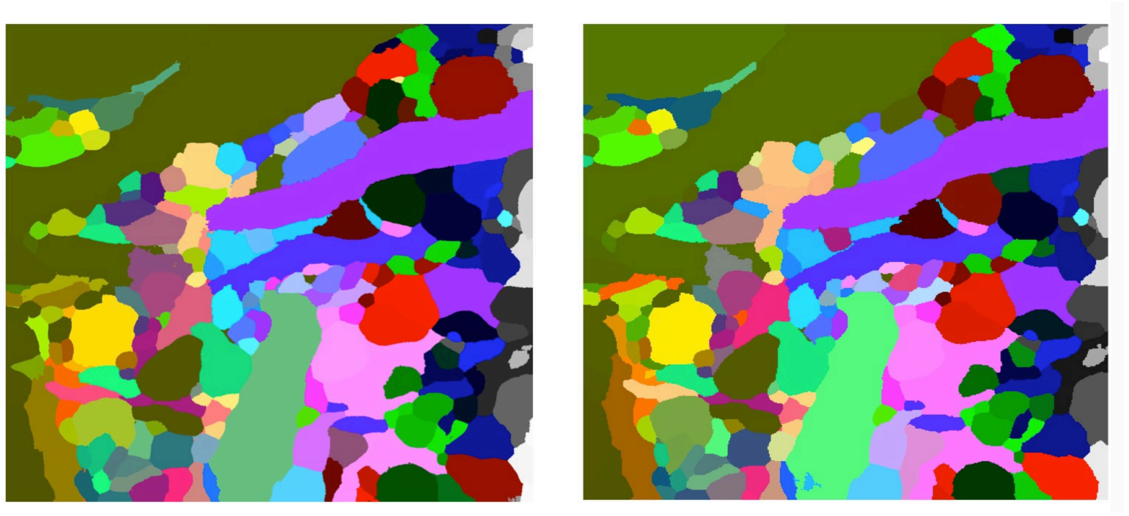


Figure 8-1: *Segmentation results of 1024 x 1024 images (left) and 512 x 512 images (right).*

This thesis is a contribution towards the the modest goal of extracting the connectome of a chunk of a small mammal's brain. Although the progress made was relatively small, I hope it was a big step forward in terms of knowing what is within out reach, and that brings us much closer to our ultimate goal.

Appendix A

Data Tables

Num. workers	Time (ms)
Serial	49755
1	52009
2	28510
4	15034
8	7888
12	5710
16	4498
18	4075

Table A.1: Data for RAG construction without worker-local RAGs.

Num. workers	Time (ms)
Serial	49755
1	49761
2	27538
4	14866
6	10012
8	7577
10	6557
12	5570
16	4316
18	3875

Table A.2: Data for RAG construction with worker-local RAGs.

Num. workers	Time (ms)
Serial	12600
1	10612
2	5812
4	3156
6	2316
8	1729
10	1513
12	1297
16	1092
18	1004

Table A.3: Data for Agglomeration.

Num. workers	Time (s)
Serial	70.59
1	69.01
2	41.63
4	26.25
6	21.46
8	17.53
10	16.30
12	15.23
16	13.66
18	13.23

Table A.4: Data for the entire segmentation process.

k	Merge Error	Split Error	Time (ms)	Iter. count
1	0.211229	0.775018		8088
2	0.211234	0.774839		5921
4	0.211229	0.774706	1764	3915
8	0.211225	0.774885	1433	2430
16	0.211229	0.774706	1189	1350
32	0.211229	0.774828	1010	847
64	0.211232	0.774676	1003	586
128	0.21162	0.777626	957	433
256	0.211636	0.777272	929	336
512	0.212619	0.772636	918	280
1024	0.213362	0.772336	851	264
2048	0.213311	0.767875	854	247

Table A.5: Data for different k values with priority randomization.

k	Merge Error	Split Error	Time (ms)	Iter. count
1	0.211234	0.774839		8088
2	0.211234	0.774839		5921
4	0.211234	0.774839	1807	3912
8	0.211234	0.774839	1516	2337
16	0.211234	0.774839	1237	1353
32	0.211234	0.774839	1095	844
64	0.211234	0.774839	1102	586
128	0.211234	0.774839	973	455
256	0.211234	0.774839	947	392
512	0.211234	0.774839	919	342
1024	0.211229	0.775018	918	317
2048	0.211229	0.775018	896	315

Table A.6: Data for different k values without priority randomization.

Bibliography

[1] R. D. Blumofe and C. E. Leiserson. *Scheduling Multithreaded Computations by Work Stealing*. JACM, 46(5):720-748,1999.

[2] Tim Kaler , William Hasenplaugh , Tao B. Schardl , Charles E. Leiserson, *Executing Dynamic Data-graph Computations Deterministically Using Chromatic Scheduling*, Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures, June 23-25, 2014, Prague, Czech Republic [doi>10.1145/2612669.2612673].

[3] William Hasenplaugh , Tim Kaler , Tao B. Schardl , Charles E. Leiserson, *Ordering Heuristics for Parallel Graph Coloring*, Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures, June 23-25, 2014, Prague, Czech Republic [doi>10.1145/2612669.2612697]

[4] Yuan Tang , Rezaul Alam Chowdhury , Bradley C. Kuszmaul , Chi-Keung Luk , Charles E. Leiserson, *The Pochoir Stencil Compiler*, Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures, June 04-06, 2011, San Jose, California, USA [doi>10.1145/1989493.1989508].

[5] E. H. Norman *Japan's emergence as a modern state* 1940: International Secretariat, Institute of Pacific Relations.

[6] IEEE International Symposium on Biomedical Imaging. *SNEMI3D: 3D Segmentation of Neurites in EM Images*. Leading Groups, n.d. Web. 01 Dec. 2014.

[7] Jeff W. Lichtman, Hanspeter Pfister, and Nir Shavit. *The Big Data Challenges of Connectomics*. Nat Neurosci, Vol. 17, No. 11. (November 2014), pp. 1448-1454, doi:10.1038/nn.3837.

[8] Nunez-Iglesias J, Kennedy R, Plaza SM, Chakraborty A and Katz WT (2014) Graph- based active learning of agglomeration (GALA): a Python library to segment 2D and 3D neuroimages. Front. Neuroinform. 8:34. doi: 10.3389/fninf.2014.00034.

[9] *Sebastian Seung: I Am My Connectome*. YouTube. YouTube, n.d. Web. 12 May 2015. <https://www.youtube.com/watch?v=HA7GwKXfJB0>.

[10] Leiserson, Charles E. *Parallelism and Performance*. Vol. 13. N.p.: n.p., n.d. Open Course Ware. MIT. Web. <http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2010/video-lectures/lecture-13-parallelism-and-performance/MIT6_172F10_lec13.pdf>.

[11] Leiserson, Charles E. *Synchronizing without Locks*. Vol. 16. N.p.: n.p., n.d. Open Course Ware. MIT. Web. <http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2010/video-lectures/lecture-16-synchronizing-without-locks/MIT6_172F10_lec16.pdf>.

[12] *Cilk_spawn*. Intel C++ Compiler XE 13.1 User and Reference Guides. N.p., n.d. Web. 12 May 2015. <https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-6DFD6494-58B4-40ED-88E9-90FEAF5AF8F6.htm>.

[13] *Cilk_for*. Intel C++ Compiler XE 13.1 User and Reference Guides. N.p., n.d. Web. 12 May 2015. <https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-ABF330B0-FEDA-43CD-9393-48CD6A43063C.htm>.

[14] Parag, T., Chakraborty, A., Plaza, S.: *A context-aware delayed agglomeration framework for EM segmentation*. arXiv 1406:1476 (2014).

[15] Toufiq Parag, Stephen Plaza, and Louis Scheffer, *Small Sample Learning of Superpixel Classifiers for EM Segmentation* - Extended Version, MICCAI, 2014.

[16] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured Parallel Programming: Patterns for Efficient Computation* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.