Distributed Image Processing For Use In Brain Visualization

Tan Zong Xuan

under the direction of
Mr. Matt Wimer, Mr. Michael Purcaro, and Prof. Sebastian Seung
Seung Lab
Massachusetts Institute of Technology

Research Science Institute
July 27, 2010

Abstract

The construction of maps of neural connections known as connectomes is a highly computationally intensive process due to extremely large images of neural matter involved. Despite advances in automated neural image processing, computer-generated segmentations of neural data still require human validation. Omni is an application that facilitates this validation by allowing users to edit segmentations in 2D or 3D. In doing so, Omni has to build mipmaps out of image and segmentation data. In the course of this project, the serial mipmapping process was rewritten to run in parallel via multithreading. Performance testing showed that large reductions in build time were realized.

Summary

Omni is a computer application for editing maps of connections between neurons which are generated by computers that analyze microscopic images of brains. These maps come in the form of images which have to be displayed on-screen. Since some of these maps are extremely large, Omni has to generate scaled down versions of these maps beforehand, so that the computer can do less work when displaying them. In this project, the process of down-scaling was rewritten to run in parallel across multiple CPU cores, thereby greatly reducing its duration.

1 Introduction

Connectomics is an emerging field which studies image data of neurons and maps out neural connections. Such a map of neurons and their connections is known as a connectome. Advances in automated microscopy have allowed for the collection of large image datasets of neural matter[6, 7], providing ample data for analysis and mapping. However, analysis of such data has traditionally been primarily manual, requiring much human labor. The first connectome constructed was that of the model organism *C. elegans*, which contained only 302 neurons and about 800 neural connections. Despite its relatively small size, the entire endeavor took at least a decade[8].

There is thus an clear disparity between the rate at which image data is created and the rate at which it is analyzed. Hence, it is necessary to develop technology that will greatly increase the speed of image data analysis. A large amount of recent work has allowed for greater automation of neuronal tracing and segmentation of neuroanatomical structures[9, 10, 11]. The accuracy of such automated processes is often lacking, however. To further increase accuracy, computer vision systems based on artificial neural networks were developed. These systems learn from human-made image segmentations using methods such as Maximin Affinity Learning of Image Segmentation (MALIS) and Boundary Learning by Optimization with Topological Constraints (BLOTC) to achieve even greater degrees of segmentation accuracy[3, 4].

Despite the advances made by these learning algorithms, computer segmentation output is still not comparable to that of humans. As such, human validation needs to be performed on the segmentation data. Omni, a program under active development at the Seung Lab, was conceived for this purpose. By loading up raw image and segmentation data, then displaying the data in 2D and 3D, Omni allows its users to easily visualize and edit images and segmentations of neural structures[1, 5]. Since Omni is still in an early stage of development, many parts of the program are not optimized for performance. Due to extremely large datasets that are or will be involved in neural mapping and brain visualization, Omni needs to be able to draw on a large amount of computational resources, such as that of a supercomputer cluster. This imposes the requirement that a large amount of Omni code be written to run in parallel. However, before the commencement of this project, the reverse was true, i.e. much of Omni ran in serial.

Before visualization and editing can occur, Omni has to first import the image data and the pre-generated segmentation of that data. It then builds 3D meshes from the segmentation data[1]. Part of these operations is the process of mipmapping, which is the pre-building of downsampled image data known as mipmaps in order to reduce the load on the graphics processing unit (GPU) at display time[2]. Rather than having to scale down extremely large source images for display on a screen many times smaller, the GPU only has to scale down the pre-built mipmaps. In this project, the mipmapping process was rewritten in a robust, scalable and parallel fashion. This was done by distributing computations amongst multiple threads (tasks scheduled to run concurrently by a computer). Iterative performance testing was used to assess increases in performance after each significant code modification.

2 Background

Before connectomic data can be generated for validation in Omni, brain image data must first be obtained. This is done by using scanning electron microscopes to image many slices of brain tissue, thus building up a three dimensional image stack of the brain tissue. Such raw image data is referred to as *channel* data, due to variety of ways in which the same tissue slice can be stained or fluorescently marked, thus creating unique channels or versions of image data, in similar vein to how most digital images have separate channels of image data for each primary color[1].

Channel data is then analyzed by artificial neural networks such as MALIS, which use image segmentation techniques to create affinity graphs[3]. Affinity graphs contain the affinity data of each pixel within the channel data, that is, how closely connected each pixel is to its neighboring pixels. By applying a threshold to these affinity graphs, neighboring pixels with affinity below that of the threshold are considered separate, and so the image is split into many different segments, creating the *segmentation* data. This segmentation process is currently carried in the Seung Lab by another in-development program known as Watershed. Both the generation of affinity graphs and the actual image segmentation through thresholding are highly computationally intensive processes that Omni is dependent upon. However, since these algorithms are currently contained in programs separate of Omni, increasing their performance through distributed processing was not considered within the scope of this project.

Once channel data and segmentation data have been obtained, they are ready for use by Omni. The data is first imported into Omni from their native image formats, which can be a large stack of 2D TIFF images for channel data, or a HDF5 file which contains both channel and segmentation data in three dimensions. After the source data has been imported, the *volumes* of image data are then built. For channel data, this building process is simply the generation of a mipmap for every level of downsampling that occurs. For segmentation data, building a volume also entails reading and saving the identification number (ID) of the segment that each voxel (volumetric pixel) belongs in. This is done for every level of downsampling.

Following this, 3D meshes are generated from the segmentation data. Once this is done, users can edit and view the channel and segmentation data through the interactive 2D and 3D views. An example of this would be a user selecting two segments for display within the 3D view, examining them to see whether they look like they are actually part of the same object, and then finally merging them into one larger segment.

3 Algorithms

This section describes how mipmapping was implemented in Omni, the algorithms that were used in the process, as well as the improvements made to the implementation and the algorithms in the course of this project.

3.1 Mipmapping

Mipmaps or MIP maps are scaled down (a.k.a.downsampled or subsampled) versions of an image that accompany it. This is so that when a GPU is requested to render that image onto a display at a small size, less time is taken since the GPU simply renders an already scaled down mipmap of that image.

In general, a mipmap created after one level of downsampling from the original image is half the linear dimensions of the original. Large images will require the generation of multiple mipmaps, with each successive mipmap being half the linear dimensions of the previous one. Each mipmap is said to be at a certain mip level. The original image is at mip level 0, while the mipmap half the linear size of the original is at mip level 1, and so on. Hence the mip level of a mipmap corresponds to the number of times the original image had to subsampled to create that mipmap. The highest mip level for an image is known as the root mip level, and this depends on the original dimensions of the image. It is determined by calculating the number of times the linear dimensions of the image can be divided by two before the dimensions are below a certain threshold value. In Omni, the formula used is $R = \lceil log_2 \frac{D}{T} \rceil$, where $\lceil x \rceil$ is the smallest integer not less than x, R is the root mip level, D is the largest linear dimension of a image volume, and T is the threshold value.

To facilitate scalability and discretization of computations, a volume of image data is broken down into cubic regions of data with equal dimensions known as MipChunks. The volume itself is referred to as a MipVolume. Many of the operations performed in Omni are at a chunk-based level, such as the subsampling algorithm itself. Each MipChunk contains a coordinate known as a MipChunkCoord, which includes its x, y, and z position as well as its mip level. When Omni was originally written, the dimensions of a MipChunk were chosen to be $128 \times 128 \times 128$. This choice was largely arbitrary, apart from the fact that 128 is a power of 2 and that at these dimensions, a MipChunk takes up a reasonable 2–8 megabytes of memory.

3.2 Subsampling Algorithm

Given that subsampling an image volume once means that it is scaled down to half of its linear dimensions, there are multiple ways in which such an algorithm might be implemented. In the case of three dimensions, subsampling means that the 8 voxels in every $2 \times 2 \times 2$ block of voxels is reduced to a single voxel, where the new single voxel is known as the parent voxel, and the 8 original voxels are known as the children voxels. This reduction can be done by setting the value of the parent voxel to either:

- the mean of the values of the 8 children voxels,
- the mode of the values of the 8 children voxels,
- the value of a voxel randomly chosen from among the 8,
- or the value of the primary child voxel, i.e. the top-most, left-most, front-most voxel.

In Omni, the code originally allowed for switching in between the different subsample modes, as does the pseudocode in Codeblock 1.

However, users of Omni within the Seung lab found that the first three subsample modes all resulted in the creation of mipmaps that looked too blurred, and that only the fourth subsample mode produced sharp mipmaps. Hence, a design decision was made to use the subsample mode whereby the parent voxel takes the value of the primary voxel of the $2 \times 2 \times 2$ children block. Despite this, the code was left as it was, even though there was no longer any

Codeblock 1 Old subsampling pseudocode with different modes.

```
VoxelCoordinate NewVxlCoord;
foreach( NewVxlCoord in NewVolume )
{
        VoxelCoordinate OldVxlCoords[8];
        OldVxlCoords[0] = GetPrimaryOldVoxelCoord(NewVxlCoord, OldVolume);
        OldVxlCoords[1] = NewCoord(OldVxlCoords[0].x+1,OldVxlCoords.[0].y,OldVxlCoords.[0].z);
        OldVxlCoords[2] = NewCoord(OldVxlCoords[0].x,OldVxlCoords.[0].y+1,OldVxlCoords.[0].z);
        OldVxlCoords[3] = NewCoord(OldVxlCoords[0].x+1,OldVxlCoords.[0].y+1,OldVxlCoords.[0].z);
        OldVxlCoords[4] = NewCoord(OldVxlCoords[0].x,OldVxlCoords.[0].y,OldVxlCoords.[0].z+1);
        OldVxlCoords[5] = NewCoord(OldVxlCoords[0].x+1,OldVxlCoords.[0].y,OldVxlCoords.[0].z+1);
        OldVxlCoords[6] = NewCoord(OldVxlCoords[0].x,OldVxlCoords.[0].y+1,OldVxlCoords.[0].z+1);
        OldVxlCoords[7] = NewCoord(OldVxlCoords[0].x+1,OldVxlCoords.[0].y+1,OldVxlCoords.[0].z+1);

        switch (mode)
        {
                case MEAN:
                NewVxlCoord.Value = CalculateMean(OldVxlCoords);
                break;

                case MODE:
                NewVxlCoord.Value = CalculateMode(OldVxlCoords);
                break;

                case RANDOM:
                NewVxlCoord.Value = OldVxlCoords[rand()%8].Value;
                break;

                case PRIMARY:
                NewVxlCoord.Value = OldVxlCoords[0].Value;
                break;
        }
}
```

Codeblock 2 New subsampling pseudocode using only primary child voxel.

```
VoxelCoordinate NewVxlCoord;
foreach( NewVxlCoord in NewVolume )
{
        VoxelCoordinate OldVxlCoords = GetPrimaryOldVoxelCoord(NewVxlCoord, OldVolume);
        NewVxlCoord.Value = OldVxlCoords[0].Value;
}
```

need to access the seven other coordinates of the non-primary voxels. As part of this project, this unnecessary code was removed, and the subsampling function was modified to look like the pseudocode in Codeblock 2. This effectively removed 7 out of 9 computer operations from the *foreach* loop, allowing for a theoretical speed increase of close to 8 times in the subsampling process.

Theoretically, using the mode or median voxel values should provide a more accurate subsampled image. Should future users of Omni desire such accuracy, an alternative codepath can easily be added to use these subsample modes via an *if-else* statement, thereby allowing users to choose between accuracy or efficiency gains.

3.3 Serial Builds

Before the commencement of this project, the process of building MipVolumes was completely serial. The subsampling algorithm itself is serial because within the *foreach* loop, each voxel is accessed individually in sequence. Rather than running the subsampling algorithm on the entire MipVolume, the serial building process looped through each MipChunk of each mip level of the MipVolume and filled up the image data of that MipChunk by applying the subsampling algorithm to its child data region, i.e. the the data region 8 times volumetrically larger than it. This process is illustrated in Figure 1.

Using MipChunks has the benefit of always passing a cubic data region to the subsampling function to subsample, since the subsampling function always assumes that about the data it receives. Even when image data does not fit exactly into an integral number of MipChunks, the MipChunks which are only partially filled up by real image data are simply completed by filling with empty data, ensuring that they are always cubical.

However, using MipChunks makes it difficult to tweak the code to run in parallel. Because each mip level is built by MipChunks, and each MipChunk has to be subsampled from its 8 children MipChunks in the previous level, a mip level has to be fully built before any higher

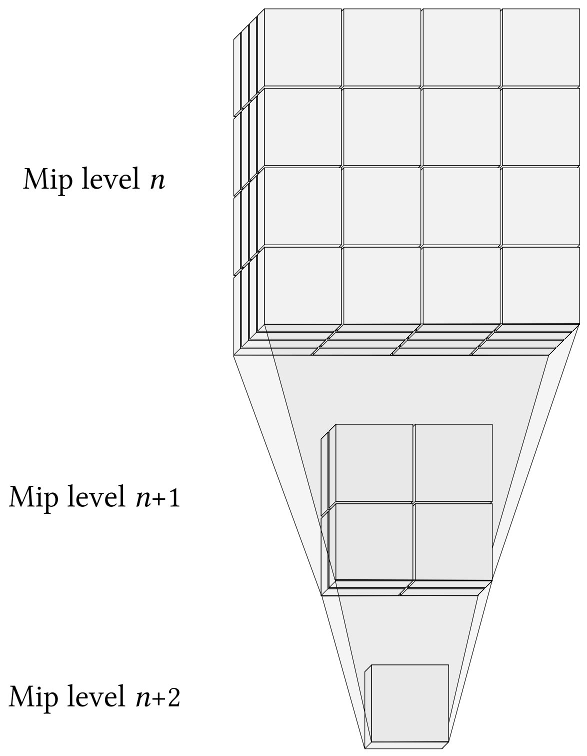

Figure 1: Serial build process. Each MipChunk (cube) is subsampled from its 8 children MipChunks in the previous mip level.

mip levels can be built. Tweaking this process such that each MipChunk is assigned to a thread for subsampling would be suboptimal, since all the threads would have to wait for each other to finish before starting on the next higher mip level.

3.4 Parallel Builds

In many multithreaded processes, there are a multitude of worker threads carrying out similar processes on different sets of data, as well as a manager thread that controls the worker threads and distributes and recollects the data. Ideally, for a parallel mipmapping process, chunks of data should be handed out to each worker thread, and each chunk of data should be continuously downsampled until the root mip level without any thread having to request for more data to complete its task. Figure 2 depicts this.

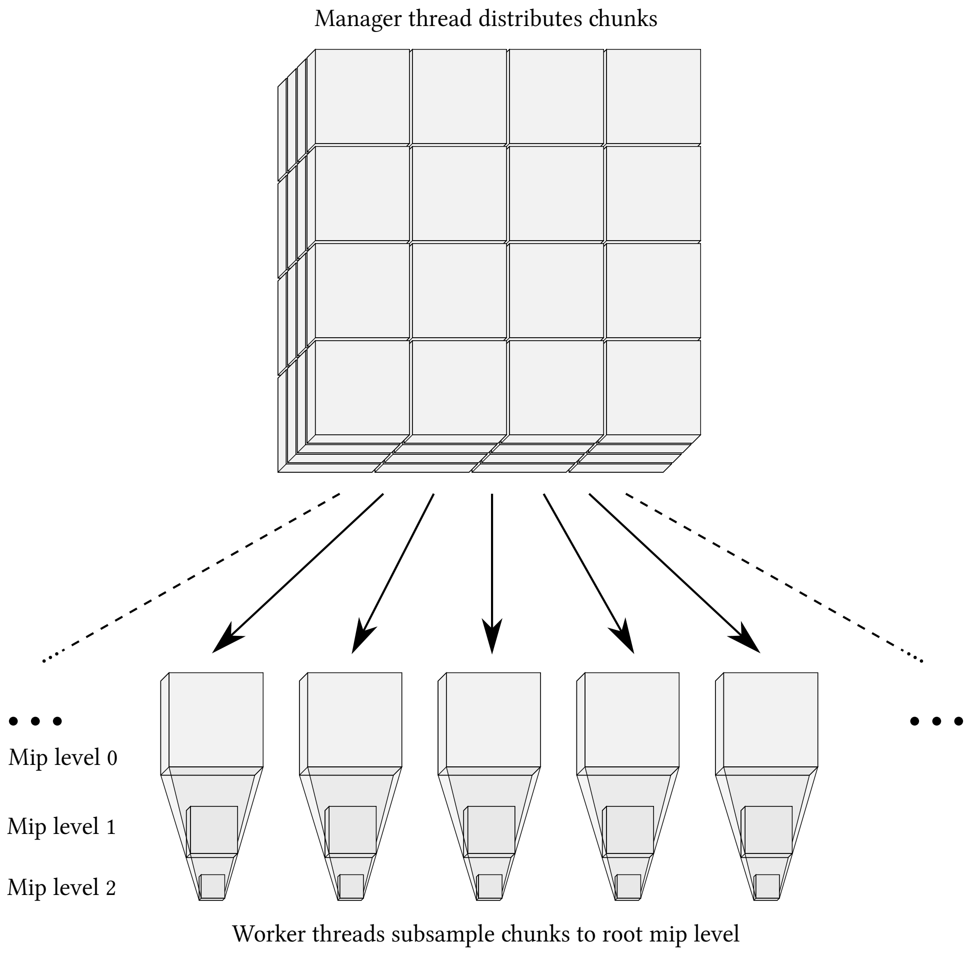
Manager thread distributes chunks
Mip level 0
Mip level 1
Mip level 2
Worker threads subsample chunks to root mip level

Figure 2: Parallel build process.

This parallel build process is described in greater detail below:

1. Manager thread loops through chunk coordinates in mip level 0.
   (a) Concurrently loops through worker threads in thread pool, where total number of worker threads is a user setting.
   (b) Adds current chunk coordinate to current worker thread's queue of assigned chunk coordinates.
2. Worker threads get chunk image data using assigned chunk coordinates
3. Worker threads recursively subsample assigned chunks until root mip level is reached.
4. Once all worker threads are done subsampling all assigned chunks, flush all chunk data to disk.

A problem with this process is that regardless of the mip level, MipChunks always have the same size of 128^3 voxels. Unfortunately, continuously downsampling the same chunk of data means that the chunk gets smaller as the mip level increases. This means these chunks cannot be stored in memory and saved to disk as statically sized MipChunks. Hence, dynamically sized abstractions of the MipChunk, ThreadChunks and ThreadChunkLevels, were created as part of this project.

3.5 Downsampling Chunks

ThreadChunks are the initial chunks of image data that are distributed by the manager thread to the worker threads to downsample. Building a ThreadChunk is actually the recursive building of its associated ThreadChunkLevels, which are the image data at each mip level of the ThreadChunk.

Building ThreadChunks was designed to be much more efficient than building MipChunks by reducing the I/O load. When building a MipChunk, previously saved data has to be read from disk to obtain the image data of the child data region of the MipChunk. In Omni, reading and writing from and to disk cannot be done concurrently. This was not a problem for the serial build, since only one MipChunk was ever being built at any point of time. However, this becomes a significant problem in the parallel build process, since the amount of wait time will increase drastically if all the worker threads are competing to read data from the disk.

This problem is resolved by having the BuildThreadChunk function recursively call itself whilst passing down the newly subsampled image data as an argument, as shown in Codeblock 3. This ensures that all requisite image data is kept in memory, thus evading I/O bottlenecks. The BuildThreadChunkLevel function builds and returns the image data by subsampling it from the source image data passed to it, and then this image data is passed to the next call to BuildThreadChunk, until the root mip level is finally reached.

Codeblock 3 Recursive BuildThreadChunk function.

```
void BuildThreadChunk(ChunkCoord ThreadChunkCoord, ImageData source_data)
{
        ImageData new_data = BuildThreadChunkLevel(ThreadChunkCoord, source_data);

        if (ThreadChunkCoord.Level < RootMipLevel){
                ChunkCoord NewThreadChunkCoord = ThreadChunkCoord;
                NewThreadChunkCoord.Level++;
                BuildThreadChunk(NewThreadChunkCoord,new_data);
        }
}

ImageData BuildThreadChunkLevel(ChunkCoord ThreadChunkLevelCoord, ImageData source_data)
{
        if (0 == ThreadChunkLevelCoord.Level){
                //If mip level is 0, no data to subsample, just read from disk
                ImageData new_data = ReadChunkImageData(ThreadChunkLevelCoord);
        } else {
                ImageData new_data = SubsampleImageData(source_data);
                //Temporarily save chunk level data to memory before flushing, based on coordinates
                SaveChunkImageDataToCache(ThreadChunkLevelCoord, new_data);
        }

        return new_data;
}
```

3.6 Sizing Chunks

In order for the worker threads to continuously downsample their assigned ThreadChunks to the root mip level, the ThreadChunks have to be large enough. Fundamentally, ThreadChunks are sized based on the root mip level. The higher the root mip level, R, the larger the ThreadChunk dimensions, because the ThreadChunk needs to be large enough to undergo subsampling R times. Thus a naive formulation of the ThreadChunk dimension d_t would be 2^R. However, this creates problems when the root mip level is low, as the end result is ThreadChunks with very small dimensions of 2, 4 or 8 pixels on a side, which generates significant input/output (I/O) load when assigning ThreadChunks to threads and when reading and writing data due to the large number of chunks. It therefore seems sensible to set a lower limit to ThreadChunk size, and to take advantage of Omni's existing infrastructure, this lower bound is simply set to the MipChunk dimension.

On the other hand, if ThreadChunk size is allowed to increase indefinitely with root mip level, problems with memory usage arise. If a ThreadChunk dimension grows to just 3 powers of two larger than the minimum, it will have have dimensions of 1024^3 voxels and use up 1–8 gigabytes of memory, quickly depleting the available RAM on most modern computers. It is therefore also sensible to set an upper limit to ThreadChunk size. This was done in Omni by creating a parameter p which sets the maximum powers of 2 that the ThreadChunk dimension can be larger than the MipChunk dimension. Hence the ThreadChunk dimension is calculated as:

$$\begin{aligned} d_t &= \min(\max(2^R, d_m), 2^p d_m) \\ &= \min(\max(2^R, 128), 2^2 \times 128) \\ &= \min(\max(2^R, 128), 512) \end{aligned}$$

with d_m being the dimension of a MipChunk.

An obvious consequence of the upper bound on ThreadChunk size is that the ideal of each worker thread separately downsampling a chunk of image data to the root mip level can no longer be realized. Instead, if the volume is too large (with a root mip level greater than 10 using the settings in the current version of Omni), then a modified version of the parallel build process occurs:

1. Manager thread distributes ThreadChunks in mip level 0
2. Worker threads recursively subsample assigned ThreadChunks until the image data is too small to further subsample
3. When all worker threads are done, flush all chunk data to disk.
4. Repeat process from top, but with highest mip level built as the initial level instead, until root mip level is reached.

This multi-pass build process is illustrated in Figure 3.

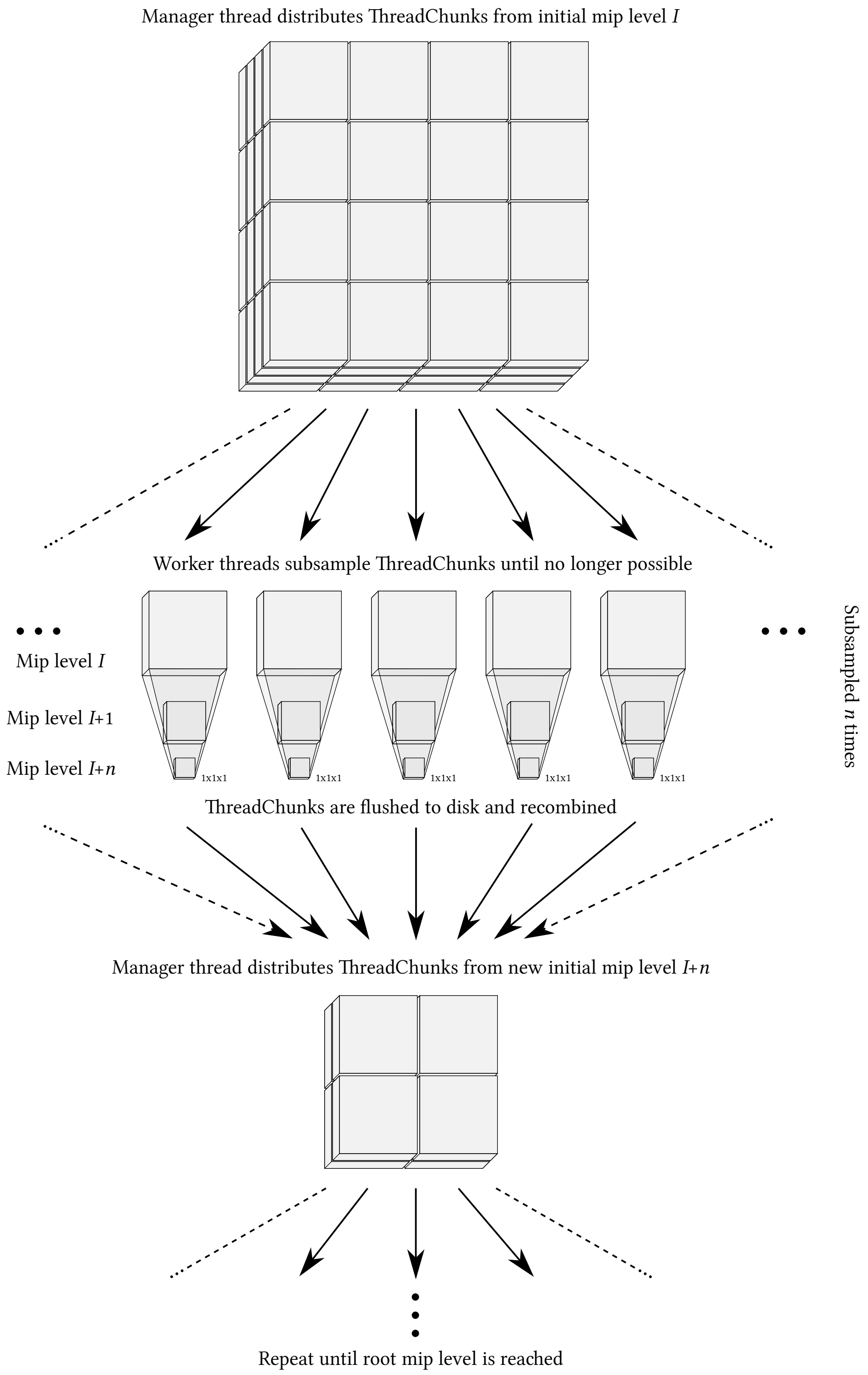
Manager thread distributes ThreadChunks from initial mip level *I*
Worker threads subsample ThreadChunks until no longer possible
Mip level *I*
Mip level *I*+1
Mip level *I*+*n*
1x1x1
Subsampled *n* times
ThreadChunks are flushed to disk and recombined
Manager thread distributes ThreadChunks from new initial mip level *I*+*n*
Repeat until root mip level is reached

Figure 3: Parallel build process with multiple passes

3.7 Segmentation Builds

In the case of segmentation builds, once image data has been subsampled, each MipChunk has the segment IDs contained within it read out and saved as metadata. This has to be done on a per MipChunk basis, because later processes such as mesh generation read the metadata on a per MipChunk basis as well. As a result, dynamically sized ThreadChunks and ThreadChunkLevels cannot be used for this process. However, this does not hinder the segmentation build process from being multithreaded easily. Once the image data is present, segment IDs can be extracted at any later time. For the serial build, this extraction process used to be done immediately after the image data of each MipChunk had been built.

Since segment IDs cannot be extracted on a per ThreadChunkLevel basis and saved as metadata without affecting other parts of Omni, segment ID extraction cannot occur directly after the building of each ThreadChunkLevel. Instead, after all ThreadChunks have been built and mipmapping is complete, the thread manager distributes MipChunks of every level to the worker threads to extract segment ID information concurrently, thus preserving the parallel nature of the build process. This is possible since every MipChunk already contains image data when segment ID extraction starts. However, there is still a slight serial element involved, since the manager thread still has to wait for mipmapping to complete before the next step can occur. Hence, this process has potential for further optimization.

4 Results

This section contains the results of the performance tests conducted on Omni at various points in the project's development. Volume builds were performed on 5 source image datasets of different sizes, and the relevant functions were timed using high-resolution timers inserted into the code. 6 builds were performed for each volume size, with timings for the first build discarded to account for caching.

Performance tests were conducted on Omni at 3 points: at the start of the project, after the modification of the subsampling algorithm, and after the build processes were rewritten to use multithreading.

4.1 System Information

All performance tests were conducted on a computer with the following specifications:

Operating System Ubuntu 9.10 (Karmic Koala)

Kernel Linux 2.6.31-14-generic

CPUs 16

Model Name Intel Xeon CPU E5530 @ 2.40GHz

Total RAM 48393 MiB

Total Swap 66757 MiB

Graphics Card NVIDIA GeForce 9800 GT

No. of Threads (Multithreaded Builds) 30

4.2 Original Performance

This section contains the results of the performance tests conducted on Omni when the project first commenced. No modifications were made to the code other than the addition of high resolution timers into the targeted functions.

The function OmMipVolume::BuildVolume was timed for both the channel build process and the segmentation build process to measure the total build time. The time spent in every function call to the subsampling algorithm (OmMipVolume::SubsampleImageData) during the build process was summed up to reduce small scale noise. This time is independent of whether a channel build or a segmentation build is being performed.

Table 1: Original channel builds, OmMipVolume::BuildVolume

Volume Size (no. of voxels)	No. of MipChunks built	Avg. Time (s)	Std. Dev. (s)
150^3	1	2.54	5.32×10^{-2}
250^3	1	2.56	1.67×10^{-2}
350^3	9	23.16	1.78×10^{-1}
450^3	9	23.10	2.69×10^{-1}
750^3	36	92.64	7.44×10^{-1}

Table 2: Original segmentation builds, OmMipVolume::BuildVolume

Volume Size (no. of voxels)	No. of MipChunks built	Mean Time (s)	Std. Dev. (s)
150^3	1	3.56	6.09×10^{-2}
250^3	1	7.24	1.01×10^{-1}
350^3	9	36.21	2.47×10^{-1}
450^3	9	50.80	5.42×10^{-1}
750^3	36	219.60	6.13×10^{-1}

Table 3: Total time spent originally in OmMipVolume::SubsampleImageData

Volume Size (no. of voxels)	Mean Time (s)	Std. Dev. (s)
150^3	2.51	5.13×10^{-2}
250^3	2.50	2.27×10^{-2}
350^3	22.33	1.69×10^{-1}
450^3	22.41	2.35×10^{-1}
750^3	89.39	5.51×10^{-1}

4.3 Subsampling Performance

This section contains the results of the performance tests conducted on the subsampling algorithm after it was modified.

Table 4: Total time spent in OmMipVolume::SubsampleImageData after modifications

Volume Size (no. of voxels)	Avg. Time (s)	Std. Dev.
150^3	0.35	1.62×10^{-3}
250^3	0.36	1.35×10^{-2}
350^3	3.13	1.32×10^{-2}
450^3	3.13	2.21×10^{-2}
750^3	12.52	8.41×10^{-2}

4.4 Multithreaded Performance

This section contains the results of the performance tests conducted on the multithreaded functions that replaced the original serial functions. For channel builds, OmMipVolume::BuildVolume was replaced by OmMipVolume::BuildThreadedVolume, while for segmentation builds, OmMipVolume::BuildVolume was replaced by OmSegmentation::BuildThreadedVolume.

Table 5: Multithreaded channel builds, OmMipVolume::BuildThreadedVolume

Volume Size (no. of voxels)	No. of ThreadChunks built	Avg. Time (s)	Std. Dev.
150^3	8	0.16	4.39×10^{-2}
250^3	8	0.16	1.44×10^{-2}
350^3	27	0.49	5.24×10^{-2}
450^3	64	0.85	9.26×10^{-2}
750^3	216	2.78	8.11×10^{-2}

Table 6: Multithreaded segmentation builds, OmSegmentation::BuildThreadedVolume

Volume Size (no. of voxels)	No. of ThreadChunks built	Avg. Time (s)	Std. Dev.
150^3	8	0.81	1.57×10^{-2}
250^3	8	1.37	1.79×10^{-2}
350^3	27	3.62	6.88×10^{-2}
450^3	64	7.03	6.06×10^{-2}
750^3	216	29.16	4.02×10^{-1}

5 Discussion

Analysis of the original performance test results revealed that the time taken for channel builds was almost equivalent to the total time spent within the subsampling algorithm, showing that the process was computation-bound rather than I/O-bound. This justified working on reducing time spent in computation as opposed to time spent in I/O.

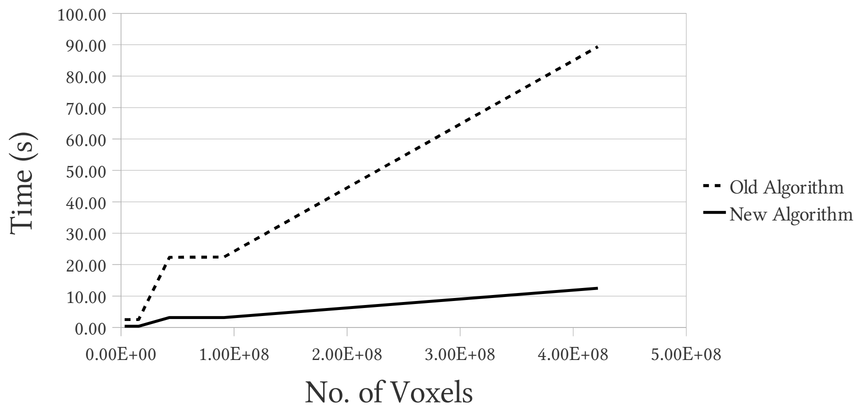

Figure 4: Old and new subsampling algorithm performance

Figure 4 shows the total time spent in both the old and new subsampling algorithms graphed against the number of voxels. It is clearly visible from the graph that the new algorithm takes much less time than the old algorithm. The increase in speed is largely equivalent regardless of volume size. On average, the subsampling algorithm runs 7.13 times faster, with a standard deviation of 4.73×10^{-2}.

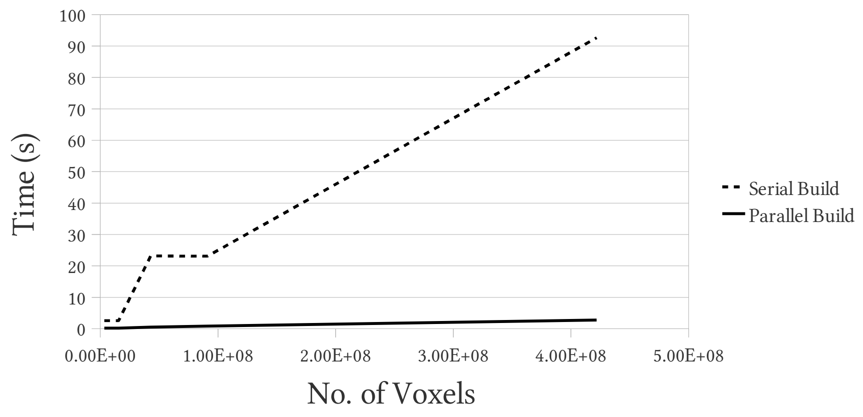

Figure 5: Serial and parallel channel builds

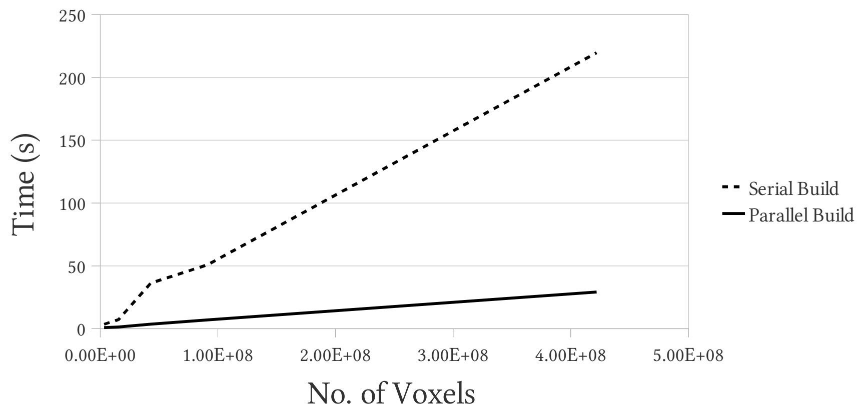

Figure 6: Serial and parallel segmentation builds

Figures 5 and 6 show respectively the time taken for the serial and parallel channel builds and the time taken for the serial and parallel segmentation builds. Again, it can be seen clearly that there are large performance increases, though the increase in speed varies much more depending on root mip level and volume size. Compared to the original channel builds, the multithreaded channel builds are anywhere from 16 to 48 times faster when using

30 threads on the test platform. The multithreaded segmentation builds show more moderate speed increases ranging from 4 to 10 times faster. This disparity is probably due to greater I/O load in segmentation builds, since read requests are going to MipChunks at every mip level, resulting in longer wait times.

Serial channel build time can be observed to correlate linearly with the total number of MipChunks built, as shown in Figure 7. This correlation cannot be observed for the serial segmentation builds however, since the extraction of segment IDs does not depend so much on the number of MipChunks as the number of segments.

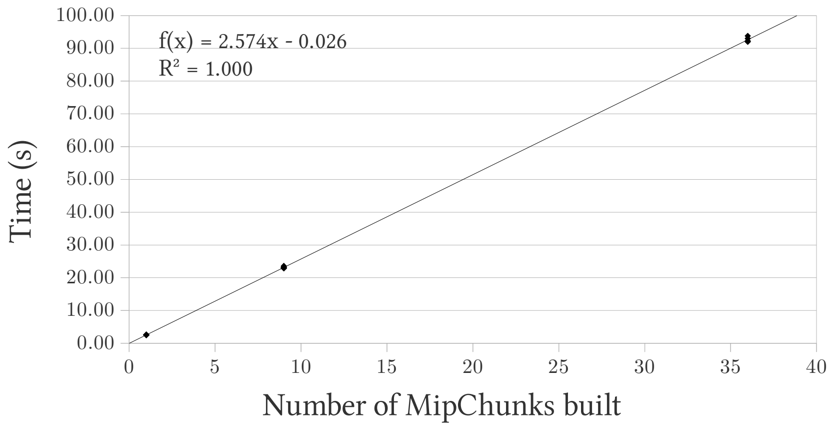

Figure 7: Serial channel build time against number of MipChunks built

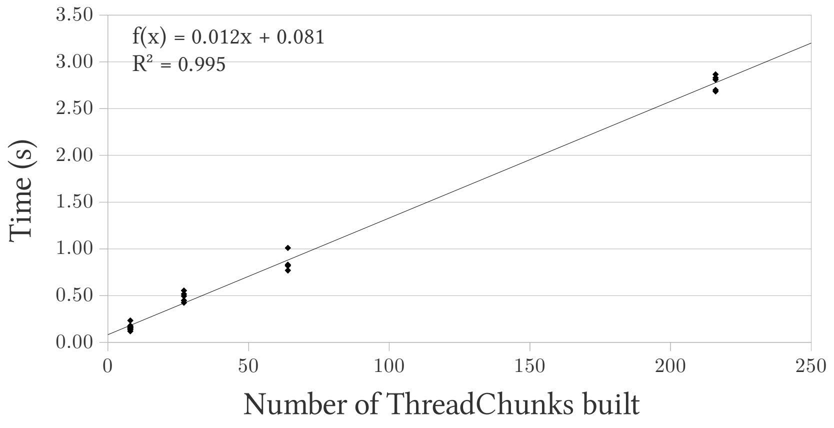

Figure 8: Parallel channel build against number of ThreadChunks built

Similarly, multithreaded build time correlates linearly with the number of chunks built, except that in the multithreaded case, this is the number of ThreadChunks, not MipChunks. This trend is shown in Figure 8. Supposing that both of these linear correlations hold, it can be shown that the multithreaded channel build scales better than the original serial build.

Given a volume that is n number of MipChunks on a side, the number of MipChunks that are built is the sum of all the MipChunks on every mip level except mip level 0. This is given by the formula:

$$\sum_{i=1}^{\lceil \log_2 n \rceil} \left\lceil \frac{n}{2^i} \right\rceil^3$$

This formula is somewhere in between a 3rd-degree and 4th-degree polynomial function of n. Thus, if n is taken as the input size, the worst-case time complexity of the serial build process is $O(n^4)$. For the parallel build process, the total number of ThreadChunks built is exactly n^3, assuming that ThreadChunks have the same size as MipChunks, and that only one pass of ThreadChunk downsampling occurs. This gives a worst-case time complexity of $O(n^3)$. The actual input size should be $N = n^3$, so the serial build process has a time complexity of $O(N^{4/3})$, whereas the parallel build process only has a time complexity of $O(N)$. This means that if the linear relationships between build time and the number of chunks built are correct, then the parallel build process performs considerably better with larger volumes than the serial build process does. Taking into account that ThreadChunks can be larger than MipChunks only affects the build time by a constant multiple. Multiple passes of ThreadChunk downsampling will make the time complexity greater than $O(n^3)$ since the number of ThreadChunks increases with each pass, but the effect is not very significant since there are far less passes than there are mip levels.

5.1 Future Work

Parallel processing on a single computer is still limited, since each computer can only have up to a certain number of processor cores. Future work could enable Omni to build volumes in a distributed manner across a computer cluster. This project has laid the groundwork for that endeavor, since all the algorithms and methods for distributing image data and downsampling image data in parallel have already been implemented.

The segmentation build process can also be further optimized, firstly by reducing the amount of I/O, and secondly by increasing concurrency of operations. The former can be done by reading MipChunks and saving metadata in a more efficient manner, while the latter can be achieved by performing segment ID extraction on a MipChunk immediately after all the image data in that MipChunk has been built, as opposed to waiting until all the subsampled image data in the entire volume is built.

6 Conclusion

By improving subsampling algorithms, and by rewriting mipmapping and segmentation build processes to run in parallel and with minimal I/O load, the time taken to build volumes in Omni was greatly reduced, as shown by results from performance testing. Speed increases of up to 48 times were realized, and the build process was made to be more scalable, thus allowing future end users of Omni to enjoy a considerably more efficient work-flow.

7 Acknowledgments

I would like to thank my mentors Matt Wimer and Michael Purcaro for their invaluable guidance in this project. I would also like to express my gratitude to those who oversaw or complemented my work on Omni, such as Dr. Srinivas Turaga and Aleksandar Zlateski, as well as to Professor Sebastian Seung for overseeing the development of Omni and giving me the opportunity to work at his lab.

This work was made possible by the Research Science Institute and the Center for Excellence in Education, and was sponsored for by the Ministry of Education, Singapore.

References

[1] B. M. Warne. *A System for Scalable 3D Visualization and Editing of Connectomic Data.* M.Sc. Thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Cambridge, MA (2009).

[2] R. W. Shearer. *Omni: Visualizing and Editing Large-Scale Volume Segmentations of Neuronal Tissue.* M.Sc. Thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Cambridge, MA (2009).

[3] S.C. Turaga, K. Briggman, M. Helmstaedter, W. Denk, and H.S. Seung. Maximin affinity learning of image segmentation. Advances in Neural Information Processing Systems 22 (2009).

[4] V. Jain, B. Bollmann, et al. Boundary learning by optimization with topological constraints. Proceedings of the IEEE 23rd Conference on Computer Vision and Pattern Recognition (CVPR 2010).

[5] H. S. Seung. Biography of Sebastian Seung, Ph.D. Available at `http://bcs.mit.edu/people/seung.html` (July 5 2010).

[6] B. A. Wilt, L. D. Burns, E. Tatt W. H., K. K. Ghosh, E. A. Mukamel, and M.J. Schnitzer. Advances in Light Microscopy for Neuroscience. Annual Review of Neuroscience. Vol. 32: 435–506 (June 2009)

[7] N. Ji, H. Shroff, H. Zhong, E. Betzig. Advances in the speed and resolution of light microscopy. Current Opinion in Neurobiology. Volume 18, Issue 6, December 2008, Pages 605–616.

[8] J. G. White, E. Southgate, J. N. Thomson and S. Brenner. The Structure of the Nervous System of the Nematode Caenorhabditis elegans. Philosophical Transactions of the Royal Society B. Vol. 314, No. 1165, 12 November 1986, Pages 1–340.

[9] A. Rodriguez, D. Ehlenberger, K. Kelliher, M. Einstein, S. C. Henderson, J. H. Morrison, P. R. Hof, S. L. Wearne. Automated reconstruction of three-dimensional neuronal morphology from laser scanning microscopy images. Methods. Volume 30, Issue 1, May 2003, Pages 94–105.

[10] A. R. Cohen, B. Roysam, J. N. Turner. Automated tracing and volume measurements of neurons from 3-D confocal fluorescence microscopy data. Journal of Microscopy (1994), No. 2, Vol. 173, Page 103.

[11] A. A. Ali, A. M. Dale, A. Badea, G.A Johnson. Automated segmentation of neuroanatomical structures in multispectral MR microscopy of the mouse brain. NeuroImage. Volume 27, Issue 2, 15 August 2005, Pages 425–435.