Image Alignment and Dynamic Graph Analytics

Two Case Studies of How Managing Data Movement Can Make (Parallel) Code Run Fast

by Brian Wheatman

[Previous/Other Information: i.e.: S.B, C.S. M.I.T., 2017]

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2019

© Brian Wheatman

The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created

Author: ____
Department of Electrical Engineering and Computer Science
May 24th, 2019

Certified by: ____
Charles E. Leiserson, Professor, Thesis Supervisor
May 24th, 2019

Accepted by: ____
Katrina LaCurts, Chair, Master of Engineering Thesis Committee

Image Alignment and Dynamic Graph Analytics Two Case Studies of How Managing Data Movement Can Make (Parallel) Code Run Fast

by

Brian Wheatman

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2019, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

High performance applications are becoming increasingly resource hungry. We want to solve more complex problems and use more data to get higher quality results. However, the more data we store, the slower it is to access any piece. This effect is seen directly in the memory hierarchy. We can access our caches faster than our memory, which is faster than reading our disk, which is still faster than going across the network. This means that when processing large data sets, we can spend a large portion of our time simply in data movement.

However, there is much we can do to optimize our programs to exploit our memory systems, so that we do not incur performance degradation as our datasets grow. I show how the careful design of data structures and algorithms allow us to scale to much larger datasets without impacting performance due to the cost of data movement.

I demonstrate the impact of these designs with two case studies. The first examines large-scale image alignment, where I describe how to align a petabyte scale set of images in memory on a single machine and match the performance of current cluster solutions. I achieve .6 - .8 TB/hr on a medium-sized multicore and linear scalability on hundreds of nodes in a shared supercomputing cluster. The second case study explores dynamic graph analytics, where I describe the design of a new data structure for storing dynamic graphs that matches the performance of standard, static formats and enables high performance, dynamic operations achieving millions of updates per second.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

I am grateful to my advisor Charles Leiserson for his help and guidance throughout my master's program.

I would like to thank my coauthors Tim Kaler, Sarah Wooders, and Helen Xu. The work on image alignment would not have been possible without the contributions of Tim Kaler and Sarah Wooders. I thank Helen Xu for help with the design and implementation of PCSR.

Finally, I am grateful to my parents, who have helped and supported me throughout my entire educational journey.

Contents

Introduction **15**

I High Throughput Image Alignment for Connectomics 20

1 Image Alignment **21**

2 Quilter Algorithm **27**

3 `Stacker` Algorithm **37**

4 Frugal Snap Judgments **43**

5 System Evaluation **47**

II Dynamic Graph Analytics 53

6 Graphs **55**

7 Graph Storage Formats **59**

8 Packed Compressed Sparse Row **63**

9 Empirical Evaluation **67**

10 Parallel Packed Memory Array **73**

11 Parallel Packed Compressed Sparse Rows **85**

12 Parallel PCSR Evaluation **91**

Conclusion **97**

A Connectomics **99**

List of Figures

1-1 Two images before and after alignment 22

1-2 Diagram of alignment pipeline for Connectomics composed of `Quilter` and `Stacker`. 25

2-1 Steps of the 2D stitching algorithm 28

2-2 Memory and I/O Characteristics of 2D alignment algorithms on a $10cm^2$ section. 30

2-3 Values for the different constants needed to calculate memory usage of a 2D alignment algorithm 31

2-4 An illustration of a `Quilter` execution. The red strip is the working set and the blue strips are the neighbors. We see that as the red strip moves up, we reuse all the calculated data. 32

2-5 Method for executing `Quilter` with extremely limited memory 33

2-6 Memory high-water mark of Quilter vs. number of tiles aligned 34

2-7 Impact of perturbations on relative tile alignment when using `Quilter`. A section from the *mouse50* and *perturbed-mouse50* dataset were both aligned using `Quilter`, and the relative alignment of each overlapping tile pair was computed. For each pair of overlapping tiles the two relative tile positions produced by aligning *mouse50* and *perturbed-mouse50* were compared to compute δ, the difference (in pixels) of the two relative alignments. The data illustrates that the alignment computed by `Quilter` is invariant to perturbations in the original data. 35

3-1 Illustration of the elastic mesh used for performing 3D elastic alignment. 38

3-2 Memory high-water mark of `Stacker` vs. number of tiles aligned . . . 40

4-1 Error with and without FSJ on human dataset. 44

4-2 Relative importance of variables used by FSJ on *human100* dataset. Variables 1-3 and 12-14 are features describing the area and dimensions of the overlapping region between the pair of tiles in their original position and when aligned relative to each other using the result of the fast path. Variables 5,15 are ratios before and after keypoint filtering stages; 5 is the matches to keypoint ratio; and 15 is the RANSAC-filtered matches to keypoint ratio. Variables 6-11 measure aggregate statistics of keypoint properties in the overlapping region: 6-7 are the average "strength" of keypoints, 9-10 are the average size, and 11-12 are the average octave. 45

5-1 Dataset descriptions. The Z column provides the number of $2D$ sections within the dataset. All datasets were obtained from the Lichtman Lab using the Zeiss multiSEM microscope. 49

5-2 End-to-end performance results for whole alignment pipeline executing both `Quilter` and `Stacker`. In the FSJ column, FSJ30 and FSJ20 indicate that the fast path employed by FSJ used 30% and 20% resolution images respectively. 49

5-3 Vertical scalability. Reports speedup obtained when executing 4 sections of *mouse200* and 4 sections of *human100* on the *LargeMulticore* platform. The *mouse200* scalability is given relative to a 1-core executing using the left-axis of the plot. The *human100* scalability is given relative to a 1-socket execution using the right-axis of the plot. The mapping of cores to sockets is provided via the $N0, N1, N2, N3$ labels. 50

5-4 Weak scalability. Illustrates reported runtimes on the *Cluster* platform on the *mouse200* dataset when the stack input-size scales with the number of nodes used to execute `Quilter` and `Stacker`. 51

5-5 Performance comparison of FijiBento and Quilter for 2D Alignment. . 52

7-1 An example of a graph stored in an adjacency list. Each entry in the vertex array points to a linked list of edges. The vertex ID in the vertices array implicitly stores the source. For weighted graphs, I store a tuple of destination vertex and edge value for each edge. 61

7-2 An example of an unweighted graph stored in compressed sparse row. The values stored in the edges array represent the destination. The vertex ID in the offset array implicitly stores the source. For weighted graphs, there is an additional values array. 62

8-1 An example of the implicit binary tree on the PMA's intervals. If we insert a new element in a leaf and the corresponding interval becomes too dense (shown in light grey), we walk up the tree until we find an interval with a density in the allowed range (shown in dark grey). In the worst case, we walk up to the root and rebalance the entire PMA. This figure is from [24]. 64

8-2 An example of a graph stored in PCSR. S denotes the sentinels. The ranges (start, end) in the vertex array denote the start and end of the corresponding edges in the edge array. 65

9-1 Size per 100,000 edges of each data structure with 100,000 nodes and a variable number of edges. The x-axis represents the number of edges, while the y-axis represents the size per 100,000 elements. 68

9-2 Time to insert 100,000 edges with a fixed number of nodes. We used 100,000 nodes and added a variable number of edges. 69

9-3 Time to insert or update 100,000 edges with a fixed number of nodes. We used 100,000 nodes and added a variable number of edges. 70

9-4 Time with 100,000 vertices and a variable number of edges. The x-axis represents the number of edges, while the y-axis represents the time. . 71

9-5 Sizes of social network graphs used in our tests. 71

10-1 Pseudocode for `get_density` in PPCSR. 75

10-2 Pseudocode for `redistribute(s, t)`. 77

10-3 Pseudocode for `double_pma`. 77

10-4 Pseudocode for `search(lo, hi, v)`. 78

10-5 Inserting into a PMA. 80

10-6 Pseudocode for grabbing locks according to `lock_order`. We use s, t to denote start and end leaf indices in the region we are trying to grab locks for. 81

10-7 The indices of leaves in a PMA and the associated priorities. 83

11-1 An example of a graph stored in PCSR format. "S" denotes a sentinel at the beginning of a vertex's region in the edge PMA. The tall lines denote leaf boundaries and elements are packed left in leaves. 86

11-2 An example of the edge PMA in PCSR with locks on vertices. The boxes represent the leaf boundaries of the PMA and the lines under the PMA represent regions associated with vertices in the graph (with their corresponding locks). 86

11-3 Pseudocode for find-neighbors in parallel PCSR. 87

11-4 Pseudocode for get-all-edges in PPCSR. 88

12-1 Time to dynamically create a graph with 256, 1024, and 4096 updates per vertex on average (from left to right) as a function of number of threads. Each graph has 100,000 vertex. 93

12-2 Time to compute sparse matrix-vector multiplication on a dense vector and sparse matrix store in each of the storage formats. The graph has 256, 1024, and 4096 updates per vertex on average (from left to right) and 100,000 vertices. 94

12-3 Time to compute one Pagerank iteration on each of the storage formats. The graph has 256, 1024, and 4096 updates per vertex on average (from left to right) and 100,000 vertices. 94

12-4 Time to construct various real-world graphs using the dynamic graph storage formats. The sizes of graphs can be found in Figure 9-5. 95

12-5 Time to compute a sparse matrix, dense vector multiplication using the dynamic graph storage formats. The sizes of graphs can be found in Figure 9-5. 95

12-6 Time to run an iteration of Pagerank using the dynamic graph storage formats. The sizes of graphs can be found in Figure 9-5. 95

List of Tables

7.1 Cache behavior of various sparse graph and matrix operations. $n = |V|$, $m = |E|$. The table lists various graph representations and the algorithmic runtime of common graph operations in the external memory model by Aggarwal and Vitter, [2] where B is the cache line (or disk block) size. The RAM model (without cache analysis) is the special case where B or $\lg(B)$ is 1. We analyze PCSR in the right-most column. * We use a `C++` vector for our implementation of CSR, so we do not need to rebuild the vertex list every time we add a vertex. 60

9.1 Real-world graphs. I tested on Slashdot, Pokec, and LiveJournal. All times are normalized against PCSR. 72

Introduction

Data movement causes many of the inefficiencies of modern computing. During processing, data must move from disk into memory and from memory into cache. With large datasets, the data moves between these systems repeatedly causing performance degradation as datasets grow.

This challenge is exacerbated by the fact that the larger a memory device is, the slower it is. Much has been done at the hardware level to combat this challenge, including caches, perfecting and branch-prediction. We have been relying on Moore's Law to continue solving this problem with new hardware, but can no longer do so as Moore's Law ends.

Instead, we must eliminate this cost at the software level by designing systems that take better advantage of memory systems. A different technique for developers to accelerate programs is to use GPUs or specialized hardware, but this hardware often has limited memory. Clusters with scalable amounts of memory gain in performance when communication is limited. CPUs, even those with terabytes of memory, have small caches that are far faster to access than memory. As such, many forms of computing benefit from careful management of memory.

I break the problem of data management into two cases:

- When the data set is too large to fit into memory on a single machine
- When the data set fits into memory on a single machine

We shall see how both situations can be optimized using the same principles, despite their differences. In a basic sense, we can think of a memory system as a small,

fast cache, and a larger slow memory. Users strive to minimize the number of times data must move from the large memory to the small one. Theoreticians model these problems using the External Memory Model, and more details can be found here [29]. This model maps well onto cluster computations, since local memory on the machine is much faster than accessing the data over the network from other machines. However, this model does not fully describe what happens inside a modern computer. Instead of containing a single cache, modern machines have a hierarchy of caches that get progressively bigger and slower. For this situation, theoreticians use the Ideal Cache Model, which makes the parameters of the cache unknown. Thus, for an algorithm to perform efficiently in the Ideal Cache Model, the algorithm must execute efficiently with any cache, and as such will perform well on all sizes of caches in the memory system. For more information on the Ideal Cache Model, see [53].

This thesis is organized into two parts, each of which examines a case study of one of the above situations.

High Throughput Image Alignment for Connectomics

Part I looks into image alignment for Connectomics, a problem where the data is too large to fit into the memory of a single system and clusters are often used. Connectomics is the creation and study of graphs of the connections in an organism's nervous system. For more information on Connectomics, see Appendix A and [41, 42, 57, 63, 66].

The process of image alignment creates and later uses an enormous amount of data. A single slice of a human brain requires about a petabyte of data, and a human brain is comprised of more than 100,000 slices.

I show how by using a schedule designed with the memory system in mind, the alignment for these petabyte-sized datasets can be computed in memory on a single commodity machine. In addition, I show how this transformation is not only work efficient, but outperforms systems running on clusters, by taking advantage of shared memory parallelism. An advantage of shared memory parallel systems is that they can require fewer total resources than cluster approaches do since the data does not need to be loaded or stored on multiple machines. Interestingly, the shared memory

model means that the memory requirement is not dependent on whether the code is run sequentially or in parallel. Lastly, I show how to extend the system I create to take full advantage of a cluster, without accruing any of the aforementioned costs of distributed computing.

Contributions

Overall, I present a system for high performance image alignment that can independently scale to both larger data sets and larger computing environments without sacrificing performance.

The contributions of this work are as follows:

- I introduce the `Quilter` algorithm that performs 2D alignment on very large mosaics without requiring unnecessary file I/O or recomputation. I show that, to first-order, the memory requirements of `Quilter` scale proportionately to the square-root of the area being aligned. The second-order memory requirements of `Quilter` are shown to be sufficiently small to permit in-memory execution on multicores with modest memory resources.

- I introduce the `Stacker` algorithm that performs 3D alignment algorithm with both affine and elastic transformations with local recomputation and allows for horizontal scaling.

- I describe the *frugal snap judgments* (FSJ) optimization technique that can be applied to obtain advantageous performance–accuracy trade-offs for image alignment.

- I provide a system evaluation of `Quilter` and `Stacker` on three datasets ranging in size from 550GB to 38TB and on three computing platforms that illustrate performance on 18-core workstation-style commodity multicores, large 112-core multicores, and on a cluster of 5440 AMD Opteron CPUs composed of 170 nodes. The evaluation illustrates end-to-end performance, vertical scalability, and weak scalability.

Part I is organized as follows: Chapter 2 describes the `Quilter` algorithm for performing 2D alignment. Chapter 3 describes the `Stacker` algorithm for performing 3D alignment. Chapter 4 describes how I employ FSJ to significantly improve the accuracy–performance trade-offs of the 2D image stitching problem. Chapter 5 presents a system evaluation for the image alignment pipeline.

Dynamic Graph Analytics

Part II looks at graph processing. In addition to Connectomics, many other fields use graph processing such as web analysis, networking, and social networks. Graph processing is often inefficient for the memory system due to the data access patterns. Choosing a format for a particular use case is a matter of weighing the trade-offs between space and speed of different operations, such as insertions, updates, look-ups, and traversals.

I introduce a new, dynamic, sparse graph representation called *Packed Compressed Sparse Row* (PCSR), based on an array-based dynamic data structure. PCSR is similar to the commonly used Compressed Sparse Rows (CSR), but gives asymptotically faster insertions and deletions in exchange for a constant factor slowdown in traversals and a constant factor increase in space overhead. It achieves this behavior by placing its data in ways that make it easier for the memory system to complete these operations with few cache misses. This data structure allows graph users to choose a new location on the trade-off curve to best suit their application.

I then develop and prove an efficient Parallel PMA, which I use the parallelize PCSR. Overall, the parallel version of PCSR is able to achieve updates speeds of over 10 million updates per second on an 8-core machine.

Contributions

My contributions are as follows:

- I describe a modification to compressed sparse row format (PCSR). PCSR enables fast searches and traversals with efficient cache usage, while supporting

fast inserts. Table 7.1 lists the cache behavior and space/time guarantees of basic operations for popular graph storage formats and PCSR.

- I implemented PCSR, as well as other standard graph storage formats, and complete an empirical evaluation. I find that PCSR supports inserts orders of magnitude faster than Compressed Sparse Rows and is about a factor of two slower for traversal benchmarks such as sparse matrix-vector multiplication.
- I give the design of a parallel PMA, which has pollylogarithmic span for all of its operations.
- I create and evaluate a parallel version of PCSR.

Part II is organized as follows: Chapter 7 reviews graph storage formats, Chapter 8 reviews the theoretical guarantees of the PCSR data structure, and Chapter 9 reports the results of a variety of benchmarks using the different data structures. Chapter 10 describes how to parallelize a PMA. Chapter 11 uses the parallel pma to construct a parallel version of PCSR. Lastly, Chapter 12, evaluates the parallel version of PCSR as compared to other parallel graph storage formats.

Part I

High Throughput Image Alignment for Connectomics

Chapter 1

Image Alignment

Large-scale image datasets are often composed of sections of a greater whole that must be stitched together like patches of a quilt prior to analysis. Examples of such datasets include satellite imagery [72], agricultural land surveys by camera-equipped aircraft [72], undersea tomography [70], and electron microscopy. An example of two images being aligned can be seen in Figure 1-1.

Image alignment started as a problem in the photography community where people would manually align images by selecting key points in multiple images to tie them together [65]. Starting in the 1980s, algorithms were used to improve the process. These early algorithms worked by minimizing pixel to pixel dissimilarities [65]. In the late 1990s, a new approach of using extracted features started being used [73] [17]. These feature-based methods had several advantages over pixel-based methods, including the ability for more transforms between the images including rotations, skew and stretch. They can also be used to find the relationships between unordered images [65]. Another benefit is that only the features need to be worked with and not the pixels and these features are often orders of magnitude less data.

Comparing alignment algorithms is inherently difficult, since most comparisons are done by eye and there is not a standard approach to measure the accuracy of an alignment [32]. Quantitative approaches are only used when the true alignment is known [59].

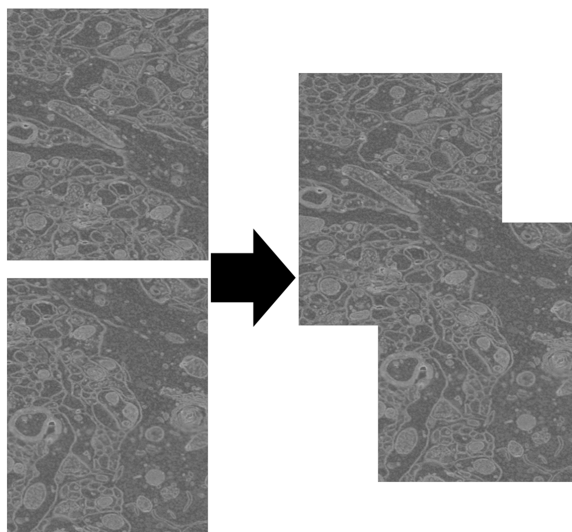


Figure 1-1: Two images before and after alignment

Image Alignment for Connectomics

Image alignment is one step in the Connectomics pipeline; for more information on the entire pipeline, see Appendix A. When aligning images for Connectomics each image comes with metadata, including which 2D slice it came from and an approximate location. The image is normally compressed, often by as much as 10x using JPEG2000 compression. The metadata is accurate enough to determine which pairs of images overlap. However, it is not accurate enough alone to align the images[1]. The alignment process outputs new metadata for each image. This new metadata includes transformations for each tile from its present location to its correct global location, a simple translation $(\delta x, \delta y)$, as well as a full affine and elastic transformation, which allows for both skew and rotation.

High-Throughput Image Alignment

Image alignment is regarded as one of the most compute-intensive steps of the Connectomics pipeline [19], and numerous systems have been described in prior work to align images produced from electron microscopy [16, 57, 66]. Prior work in 2010 [66] has reported throughputs of 24GB/hr. falling far short of the throughput required for

[1]The metadata alone is not precise enough since the motor in the microscope is not accurate to the nanometer scale, so instead it insures overlap between neighboring pictures and assumes the images will be aligned afterwards.

Connectomics, which is on the order of terabytes per hour.

Part of why image alignment is so computationally intensive is because it operates directly upon full-resolution pixel data. Designing an image alignment system in a task-based distributed computing model such as MapReduce [23] or Spark [71] might seem like an appealing option. Frameworks like these alleviate the programming burden required to design scalable big-data systems and can scale "horizontally" through the addition of machines to a computing cluster.

The convenience of MapReduce-style distributed computing simply makes it easier to satisfy the application‘s computing needs without changing its code. However, these frameworks do not reduce the resource needs of an application. Scaling horizontally can, in fact, often reduce efficiency due to the added overheads of fault-tolerance and out-of-core communication for intermediate results.

An alternative to horizontal scaling that can often be more cost-effective is vertical scaling by designing software for a single, large multicore [5, 46, 49]. Instead of improving the performance of an application through the addition of computing nodes, one instead improves performance by increasing the capabilities of individual nodes in the cluster. However, while vertical scaling can be more cost-effective for a particular problem size, it is often more difficult to predict how it scales to much larger inputs.

Existing solutions Implementations of keypoint-based image alignment algorithms exist as part of TrakEM and were distributed as part of the Fiji project and report performing rigid and elastic alignment of EM volumes at a rate of 2GB/hr using a single 4-core 2.66GHz machine [57, 58]. The implementation of TrakEM has been widely used and cited in recent Connectomics work and support for distributing work across nodes in a computing cluster added in FijiBento [55].

A high performance method for performing image stitching of large 2D time-lapse mosaics via Fourier-based methods was described in [18]. This approach is not directly applicable to the alignment of EM volumes for Connectomics, but their impressive performance results contextualize the performance shown in this work. Using a 20-core machine, their CPU-only implementation achieves a throughput of 150 GB/hr, and

their GPU implementation achieves a throughput of 640 GB/hr with 20-cores and two NVIDIA Tesla K40 GPUs.

Memory-Efficient Alignment Algorithms

I describe an efficient approach to the image alignment problem that minimizes the total work and data movement so that it is able to scale to both more hardware and larger datasets efficiently.

The alignment system is composed of two shared-memory multicore algorithms called `Quilter` and `Stacker` that perform 2D and 3D alignment respectively. Both `Quilter` and `Stacker` are carefully performance-engineered codes that rely on the ability to perform fast shared-memory communication within a single multicore. The design of `Quilter` and `Stacker` ensures that both can scale to the extremely large datasets on the horizon in the field of Connectomics.

Scaling to larger datasets. Careful memory management in `Quilter` and `Stacker` permit both algorithms to scale to the largest conceivable dataset on the horizon in Connectomics. To reconstruct a complete human-brain (zettabytes of data) both `Quilter` and `Stacker` would require the ability to process 1000TB sections on a single multicore. On this hypothetical dataset, `Quilter` requires just 1TB of memory and `Stacker` requires just 4.4TB of memory. To reconstruct a complete mouse brain (exabytes of data) `Quilter` would require just 32GB of memory and `Stacker` would need 40GB of memory.

Distributing across multiple machines. Both `Quilter` and `Stacker` operate on coarse units of work that may be distributed across multiple machines: `Quilter` operates on complete 2D sections of data and `Stacker` operates on pairs of 2D sections. A diagram of the alignment pipeline illustrating this coarse-grained parallelism is illustrated in Figure 1-2. The design of `Quilter` and `Stacker` does not require low latency communication of intermediate results, and the intermediate results which are communicated are small (approximately 0.4% of the dataset size).

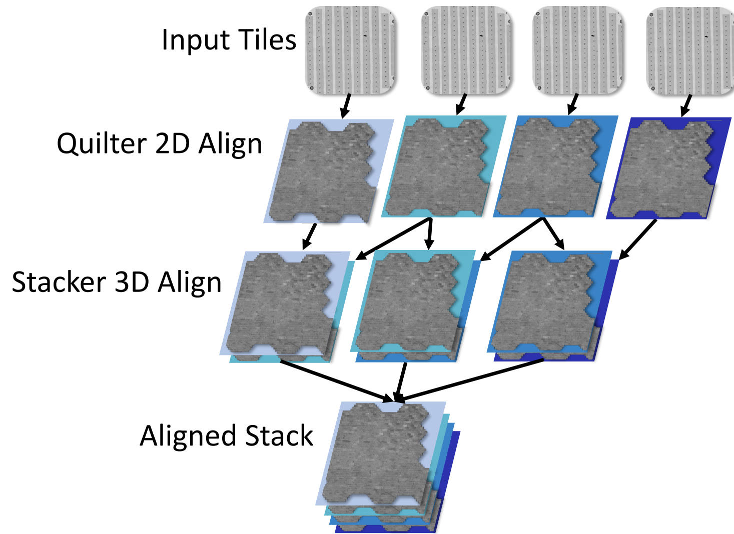


Figure 1-2: Diagram of alignment pipeline for Connectomics composed of `Quilter` and `Stacker`.

These optimizations enable our system to perform image alignment on medium-sized multicores (18 cores) with throughputs of $0.6 - 0.8$ TB/hr, on a large 112-core multicore (Intel Xeon Platinum 8180) with throughputs of 1.4-1.5 TB/hr, and on a shared supercomputing cluster with 5,440 AMD Opteron CPUs with throughputs of 7.5 TB/hr.

Chapter 2

Quilter Algorithm

This chapter describes `Quilter`: a 2D alignment algorithm for stitching very-large mosaics in-memory. `Quilter` employs careful task ordering to bound its memory use while avoiding unnecessary file-I/O and recomputation. This enables `Quilter` to reap the performance advantages of in-memory computing, even for very-large mosaics. `Quilter` can process a 2D cross section of an entire mouse brain with under 50GB of memory and process a 2D cross section of a human brain with less than a 1TB of memory.

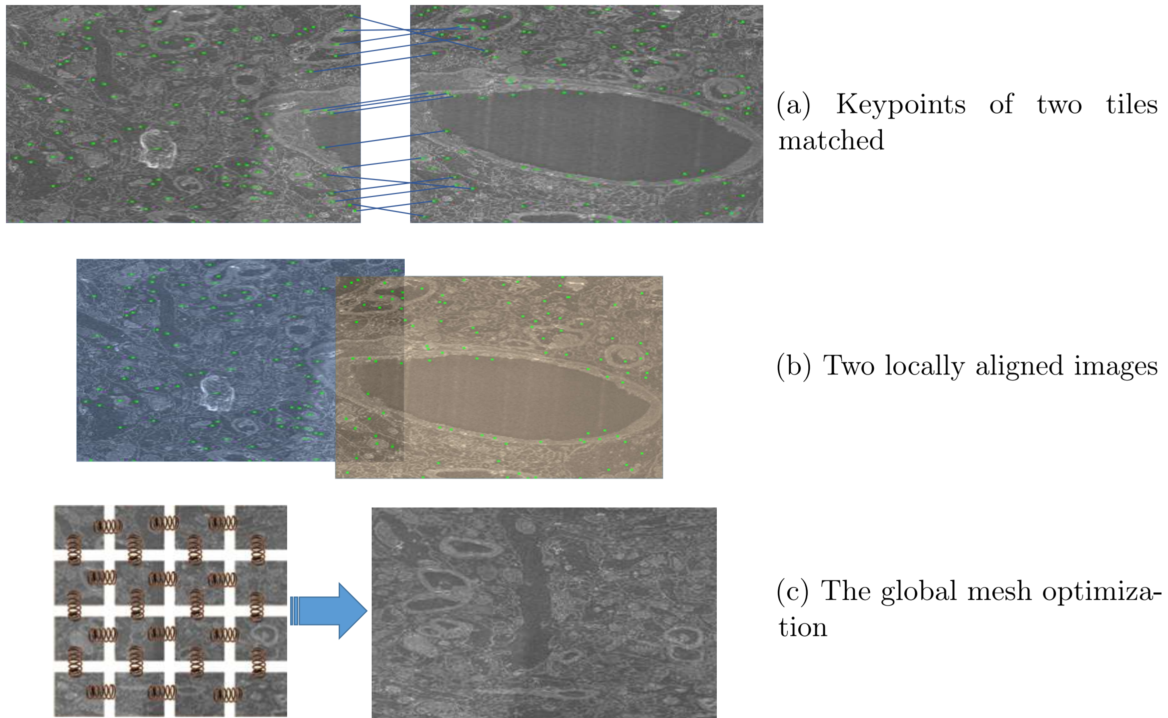
Stitching Algorithm

The basic steps of an image alignment algorithm are:

1. `GenerateKeypoints`:
   Generate keypoints for the overlapping region of each pair of images

2. `MatchTilePair`:
   Determine the relative offset for each pair of tiles by matching corresponding keypoints

3. `OptimizeMesh`:
   Assemble the mesh of images using the relative offsets


(a) Keypoints of two tiles matched

(b) Two locally aligned images

(c) The global mesh optimization

Figure 2-1: Steps of the 2D stitching algorithm

`GenerateKeypoints` uses scale-invariant feature transform (SIFT) to generate the keypoints. SIFT is an algorithm for finding keypoints in an image. Each keypoint has a scale, rotation, and illumination independent descriptor [45].

`MatchTilePair` matches pairs of keypoints within the overlapping region using a nearest neighbor search on the descriptors. A random sampling and consensus algorithm (RANSAC) [27] is then used to determine the best relative offset between the pair of images. Two images with their keypoints matched is seen in Figure 2-1a. Figure 2-1b shows their local alignment.

`OptimizeMesh` models each relative offset as a spring, as seen in Figure 2-1c. The total energy of the system is then lowered using gradient descent.

Design Considerations

Before I introduce `Quilter`, let us discuss a few straw-man algorithms for performing $2D$ alignment. This discussion will help illustrate the memory and I/O demands of a $2D$ alignment algorithm for Connectomics and motivate the design decisions in `Quilter`.

First, let us consider the two extreme approaches: `All-Mem` which loads the entire dataset into memory to perform alignment and `All-I/O` that only stores a single pair of tiles in-memory at any given time and writes all intermediate results to disk.

An `All-Mem` algorithm could store all the image data in-memory, perform the `GenerateKeypoints`, `MatchTilePair` for all tile-pairs, and then perform `OptimizeMesh`. `All-Mem` requires memory proportional to the size of area being aligned. A square, centimeter area imaged at 3×3 nm resolution would require 1PB of memory. The `All-Mem` algorithm could reduce this overhead by a factor of 10 (when using JPEG2K compression) by performing on-demand compression and decompression of images, at the expense of additional computational overhead. An `All-Mem` algorithm needs to read the image data exactly once, and so its file I/O requirement is proportional to the size of the compressed images on-disk.

An `All-I/O` algorithm could read each pair of image tiles that overlap from disk, compute `GenerateKeypoints` and `MatchTilePair` for that pair, and then write the relative alignment of those tiles to disk. After this process, `All-I/O` performs `OptimizeMesh` either in-memory or using an external-memory graph-processing framework (e.g. GraphChi [37]). The memory requirements of `All-I/O` do not depend on the total size of the area being aligned, but could depend on the total number of image tiles in the dataset due to the `OptimizeMesh` computation. The average number of overlapping neighbors for a tile in a Connectomics dataset is approximately 8. An `All-I/O` algorithm, therefore, would read each image approximately 4 times. For a square centimeter section at 3×3nm resolution, 400 TB of I/O would be required to read the images (compressed by 10x) from disk.

There is a continuum of algorithms with different memory and I/O requirements between `All-Mem` and `All-I/O`. Let us consider two such algorithms, `Inter-Mem` and `Inter-I/O`, that both read the image data only once by precomputing the result of `GenerateKeypoints` for all tiles prior to performing `MatchTilePair`.

An `Inter-Mem` algorithm reads each image, executes `GenerateKeypoints` on that image, and stores the intermediate result in-memory. Then `Inter-Mem` performs `MatchTilePair` on all pairs of tiles using the intermediate results of `GenerateKeypoints`.

Method	Memory	Total I/O	I/O Ops
`All-Mem`	1000 TB	100 TB	130 million
`All-I/O`	0.5 TB	400 TB	520 million
`Inter-Mem`	160 - 400 TB	100 TB	130 million
`Inter-I/O`	0.5 TB	640 - 1600 TB	780 million
`Quilter`	0.8 TB	100 TB	130 million

Figure 2-2: Memory and I/O Characteristics of 2D alignment algorithms on a $10cm^2$ section.

The memory requirements of `Inter-Mem` are proportional to the size and number of keypoints generated for each tile. For 8.5MB images, one typically requires at least 1,000 keypoints for each overlapping region to perform alignment, a total of 8,000 keypoints per tile. In practice, typically 20,000 keypoints are precomputed when using 8.5MB images. A SIFT keypoint descriptor requires 156 bytes of memory, and thus the memory requirements of `Inter-Mem` on a square centimeter area range between 160TB and 400TB.

Instead of storing the intermediate results of `GenerateKeypoints` in-memory, one could write these results to disk using an `Inter-I/O` approach. Unfortunately, the size of the intermediate results are bigger than the compressed images. As such, an `Inter-I/O` algorithm would generally have a higher I/O requirement than the `All-I/O` approach.

A summary of this discussion is provided in Figure 2-2 and Figure 2-3.

Design of `Quilter`

Let us now describe the design of `Quilter`.

The input to `Quilter` is a metadata file for a section that contains for each tile a path to its image on disk and an approximate bounding box of the tile's location within the section.

`Quilter` begins by computing the set of overlapping neighbors for each tile. Then `Quilter` sorts all tiles by the y-coordinate of their bounding box's bottom-left corner.

The initial set of tiles that `Quilter` processes is chosen by finding all tiles whose

Constant	Value
pixel resolution	3 nm
image height	2724 pixels
image width	3128 pixels
compressed image size	832 KB
uncompressed image size	8.5 MB
tile metadata size	4 KB
keypoint size	156 bytes
keypoints per image	8,000 - 20,000
images per mm row	107
images per cm row	1,066
images per 10 cm row	10,656
images per mm^2 *section*	13,040
images per cm^2 *section*	1,304,000
images per 10 cm^2 *section*	130,400,000

Figure 2-3: Values for the different constants needed to calculate memory usage of a 2D alignment algorithm

bounding boxes overlap with a horizontal slab extending along the bottom of the section. For each tile selected, all of its neighbors not already in-memory are read from disk. `Quilter` then executes `GenerateKeypoints` and `MatchTilePair` to compute pairwise alignments between selected tiles and all of their overlapping neighbors. Then, `Quilter` progresses by increasing the position of its horizontal slab to select a new set of tiles to process. This new slab contains the neighbors of the previous slab, which are already in memory. Before `GenerateKeypoints` and `MatchTilePair` run on the new slab, its neighbors are brought into memory. Some of the neighbors are already in memory from the previous slab. Once `MatchTilePair` is finished, the first slab's data can be released, the slab moved up, and the process repeated.

Figure 2-4 illustrates an in-progress execution of `Quilter`. We can see the outline of each image. The tiles in red are currently being processed. Their neighbors, in blue, are in memory to allow the red tiles to be processed. The dark grey tiles are already finished and have had their image data released. The light grey tiles have yet to be loaded.

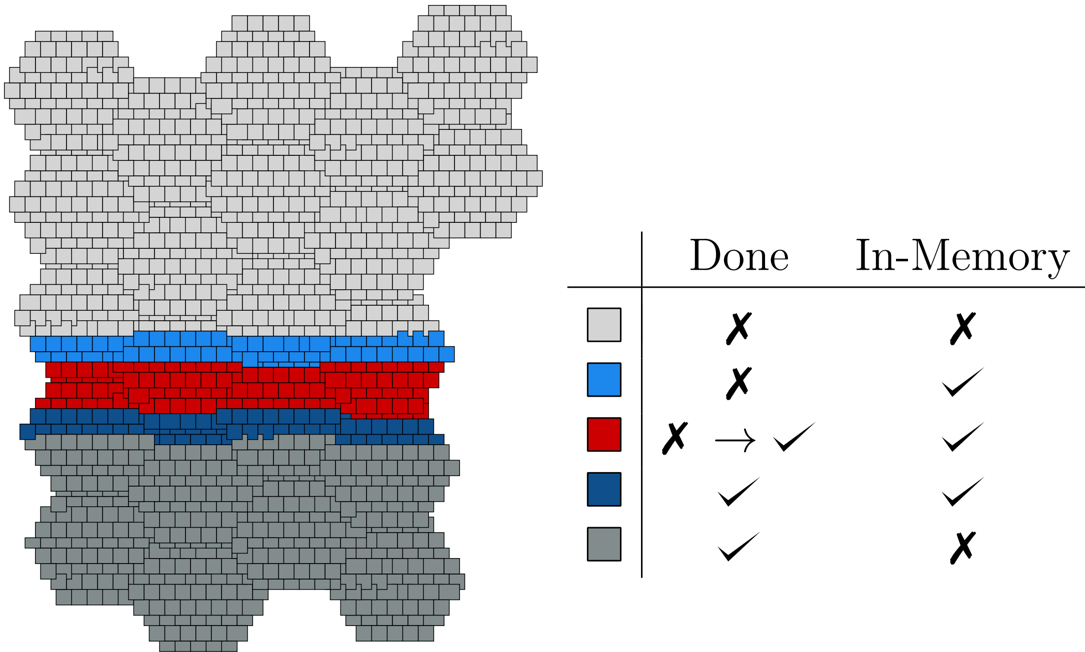


Figure 2-4: An illustration of a `Quilter` execution. The red strip is the working set and the blue strips are the neighbors. We see that as the red strip moves up, we reuse all the calculated data.

Providing Sufficient Parallelism The technique used to bound memory reduces the total parallelism of the 2D alignment algorithm. In order to mitigate this issue, `Quilter` provides a knob that can increase the amount of parallelism in exchange for an increase in memory requirements by allowing more tiles to be processed during each step. Instead of having a single row, each set contains multiple rows of data[1]. This knob gives a linear trade-off between memory and theoretical parallelism. If a section is big enough, there is enough work in three lines to not need to increase the width of the working set. Thus, the only use case is when the total memory requirement is already relatively low.

[1]We always only need to hold onto a single row from the last step.

Extremely Limited Memory Environments

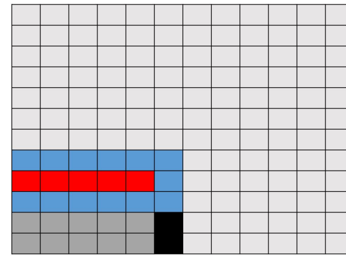


Figure 2-5: Method for executing `Quilter` with extremely limited memory

In an environment where even storing three rows of data requires too much memory, `Quilter` is able to further decrease the memory requirement at the cost of either work or I/O. Instead of the strip going all the way across, it only partially crosses the section. Then the same system as before is run, but `Quilter` is not able to complete the extreme edge of the sweeping line since some of the data is not in memory. These tiles are completed afterwards with another pass through the dataset. However, the tiles on the edge of the sweeps must either be processed multiple times or written to disk. We see in Figure 2-5 how this execution is run. Since only the edge is wasted work we can reduce the work overhead to a small fraction depending on the length of the sweeping line. Having an active set of hundreds of tiles causes an overhead of less than 5%. However, in this case external graph processing libraries must be used for the final step.

Analysis of `Quilter`

We shall now analyze the memory and I/O required by `Quilter` to align a section using the data from Figure 2-3.

Analytic Estimate of `Quilter`'s I/O Usage

`Quilter` reads in the dataset in compressed form. We can calculate the amount of I/O needed for a section using the compressed image size and the number of images in a section. This ends up being about a TB for a mouse brain and 100 TB for a human brain. `Quilter` only writes the tile ids and offsets back to disk, which ends up being neligible in size. See Chapter 3 for how this output can be used in a full alignment pipeline.

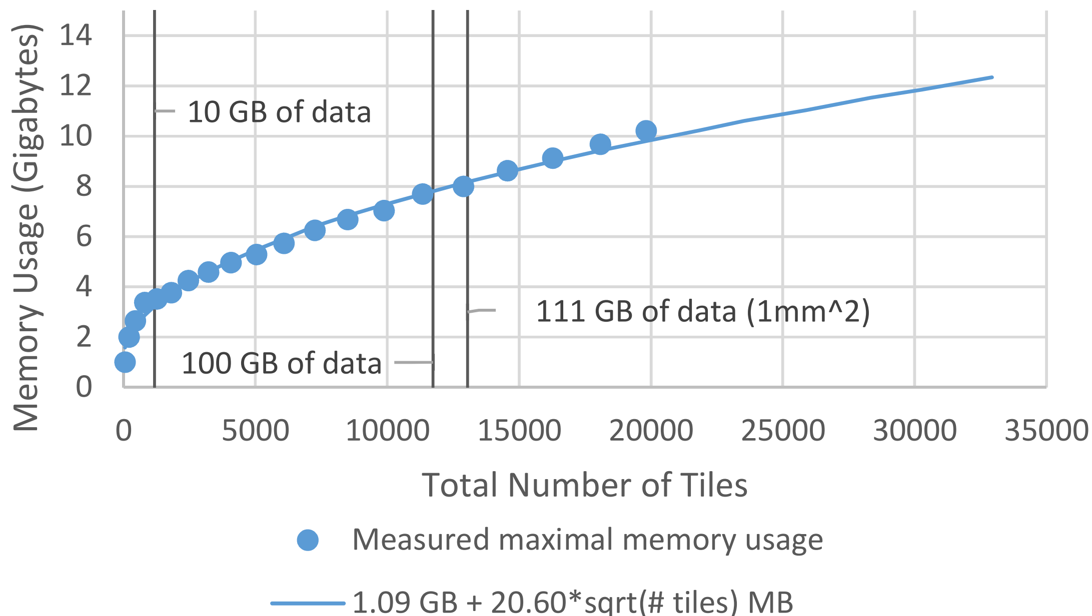


Figure 2-6: Memory high-water mark of Quilter vs. number of tiles aligned

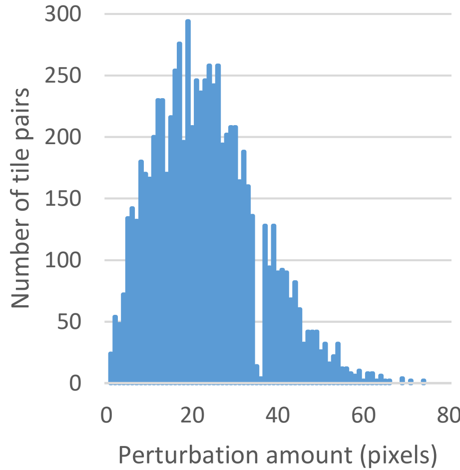
Analytic Estimate of `Quilter`'s Memory Usage

`Quilter` minimizes the number of tiles processed at once to a constant number of rows of tiles. At worst, each tile will have both their image data and keypoint data held in memory. To calculate the memory usage we use the metadata, which needs to be stored for every tile, as well as the uncompressed data size, which is needed only for the three rows of tiles `Quilter` is currently working on.

Overall, `Quilter` only needs about 32 GB for a mouse brain and about 800 GB for a human brain.

Experimental Analysis of `Quilter`'s Memory Usage

I conducted experiments to illustrate the memory requirements of `Quilter`. I took a large dataset of 40,000 tiles from a human biopsy and created smaller datasets, by cutting out a circle from the middle. Then, I aligned regions of varied size and measured the maximum resident set size needed while performing the alignment. From Figure 2-6, we see the memory requirement grows with the square root of the number of tiles. My experiments show that my implementation of `Quilter` has memory



(a) Distribution of Perturbations.

Alignment δ	# Tile Pairs
$0 \leq \delta < 1$	3837 (99.80%)
$1 \leq \delta < 2$	7 (0.18%)
$2 \leq \delta < 3$	1 (0.02%)
$3 \leq \delta < \infty$	0

(b) Post-alignment comparison.

Figure 2-7: Impact of perturbations on relative tile alignment when using `Quilter`. A section from the *mouse50* and *perturbed-mouse50* dataset were both aligned using `Quilter`, and the relative alignment of each overlapping tile pair was computed. For each pair of overlapping tiles the two relative tile positions produced by aligning *mouse50* and *perturbed-mouse50* were compared to compute δ, the difference (in pixels) of the two relative alignments. The data illustrates that the alignment computed by `Quilter` is invariant to perturbations in the original data.

requirements that indeed scale with the square root of the number of tiles in the section. The constant memory-per-tile in my implementation is approximately 20 MB per tile, which is about 2x larger than my analytic estimates of an ideal implementation. The extra memory consumption in my implementation can be attributed, primarily, to the storage of two copies of each image, one with and one without background subtraction.

Accuracy analysis

As stated in Chapter 1, determining if a set of images are properly aligned is only possible when a true alignment is known. Instead, to check accuracy, I used a measure of stability by aligning two datasets of the same images with different metadata.

The original dataset was composed of 1,342 tiles, which overlapped by between 1% and 20% of their area. Alignment errors were introduced by randomly, repeatedly perturbing the location of tiles while maintaining the property that any two tiles that overlapped in the original dataset still overlapped by at least 1% after its perturbation.

Figure 2-7 shows the impact on the data from this perturbation process. We see every pair of tiles now has a different vector between them.

After aligning both the original and the perturbed data, only 8 out of the 3845 pairs of overlapping tiles differ at all, with only 1 pair differing more than one pixel.

Chapter 3

Stacker Algorithm

This section describes the Stacker 3D image alignment algorithm. Stacker operates upon a pair of adjacent sections in a stack that have each been 2D-aligned using Quilter. Stacker supports both affine transformations and non-affine "elastic" transformations, which enable it to compute high-quality 3D alignments. These alignments can correct for distortions introduced by errors during sample preparation or imaging.

Stacker is designed to run in-memory on commodity multicore hardware. Given a stack of sections, Stacker computes the relative alignment between all pairs of adjacent sections. These relative elastic alignments are then combined using an associative operation. This design permits Stacker to align large stacks of images with on multicores with modest memory sizes and to scale horizontally across many machines to align different pairs of sections in parallel.

Design of Stacker

Stacker computes a coordinate transformation mapping points in a section S_i to points in a neighboring section S_{i-1} in two steps: a coarse affine transformation followed by a more precise "elastic" transformation.

Elastic mesh Stacker represents the transformation of section S_i to S_{i-1} as a mesh of triangles formed by triangulated evenly spaced points arranged in a hexagonal grid

of fixed size (3000 in our experiments). Figure 3-1 illustrates the structure of the elastic mesh used by `Stacker` to represent affine and elastic transformations. For each vertex in the elastic mesh, `Stacker` maintains two coordinates: the original placement of the vertex prior to any transformations and a transformed placement that maps the vertex into alignment with the section below. A point inside of a triangle is transformed by computing its barycentric coordinates [34] relative to the original placement of the triangle vertices, and then using the transformed placement of the vertices to invert the barycentric transform.

Coarse Affine Transform. `Stacker` first computes a coarse affine transform using keypoints computed on low-resolution ((1/8)th resolution) image tiles. An approximate nearest neighbor search is performed to find each keypoint's two nearest neighbors, and matching pairs that pass a ratio of distance (RoD) test are saved. Finally, a random sampling and consensus (RANSAC) [27] algorithm is employed to find an affine transform that successfully brings the most matching points into alignment (within a 100 pixel tolerance) across the sections. The computed affine transformation is applied to the section by applying the transform to each of the vertices of the elastic mesh.

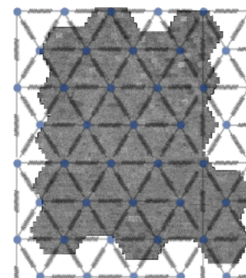


Figure 3-1: Illustration of the elastic mesh used for performing 3D elastic alignment.

Computing Fine-Grained Corresponding Points After a coarse affine transform has been computed that approximately aligns section S_i to S_{i-1} it is possible for `Stacker` to inspect small overlapping areas of S_i and S_{i-1}. `Stacker` iterates over $12k \times 12k$ neighborhoods of S_i and finds corresponding points between that region and section S_{i-1}. These matches are used to compute high-quality local affine transforms between the two sections that are associated with each triangle in the elastic mesh.

Elastic optimization Elastic optimization attempts to minimize the distance between corresponding points in section S_i and S_{i-1} while penalizing length and area

deformations within the section. The mesh is treated as an elastic sheet by penalizing changes to triangle area or triangle edge-length while rewarding deformations that bring matched points closer together.

Combining Elastic Transformations Both the affine and elastic transformations are performed through barycentric coordinate transformations via the elastic mesh. These elastic transformations can be combined. For example, consider the stack of three sections S_1, S_2, S_3. For this stack, S_3 can map any of its points into the pre-transformed coordinate space of S_2. To map a point in S_3 to the post-transformed coordinate space of S_2 `Stacker` uses the elastic transform of S_2 relative to S_1. For moderately sized stacks, this process can be performed serially and for larger stacks parallelized with a logarithmic multiplicative overhead in total work.

Analysis of `Stacker`

I shall now analyze the memory requirements of `Stacker` in order to gauge its ability to scale to larger data sizes. Unlike `Quilter`, `Stacker` performs operations globally over entire sections. Fortunately, these operations require a relatively small amount of data per-pixel, which allows `Stacker` to scale using commodity multicores to align volumes as large as $10cm^3$.

Memory for 3D keypoints `Stacker` requires space to represent the 3D keypoints used to compute a coarse affine transform between pairs of sections. Approximately 100 keypoints-per-tile is sufficient to compute these coarse transforms and each keypoints requires approximately 156 bytes. The human brain is composed of approximately $125,000,000$ tiles and thus `Stacker` requires approximately 2TB of memory per section to store these keypoints in-memory.

Memory for Elastic Mesh A hexagonal grid of width 3000 would cover each tile with at most 2 hexagons and 12 triangles. Each triangle requires approximately 128

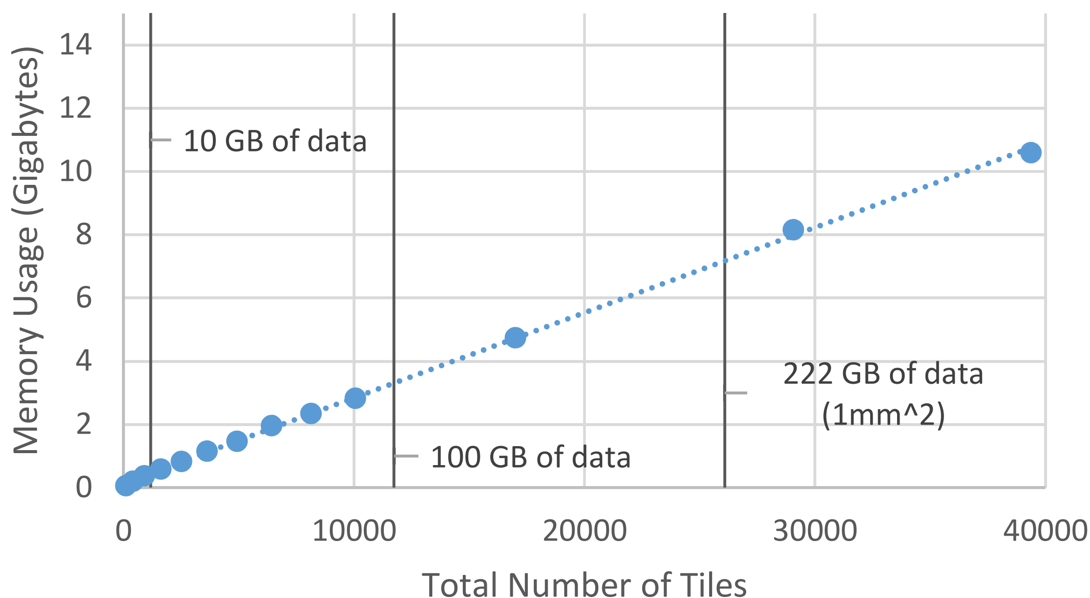


Figure 3-2: Memory high-water mark of `Stacker` vs. number of tiles aligned

bytes of space. For the human brain composed of $125,000,000$ tiles `Stacker` would require approximately 200GB of space per section.

Total Memory Required by `Stacker` `Stacker` must store the data associated with two sections in-memory at a time. For the human brain, this would require the use of a multicore with approximately 4.4TB of memory. Although this would constitute quite a large multicore by present standards, it is still in the realm of readily obtainable hardware. Furthermore, the human brain is 100x larger than the "grand challenge" of processing an entire mouse brain. The memory requirements of `Stacker` on such a dataset is less than 50GB.

Experimental Analysis of `Stacker`'s Memory Usage

I ran an experiment to empirically measure the memory requirements of `Stacker`. I followed a similar methodology to that used in Section 2 to analyze `Quilter`'s memory requirements. Regions of varied size were extracted from a section of the *human100* dataset, and then this section was duplicated to construct a stack of two identical sections. `Stacker` was then executed to align this dataset. Figure 3-2 shows the

results of my empirical experiments to measure `Stacker`'s memory requirements. I found, empirically, that `Stacker` uses 300 KB of data per tile. This is about 20x more than our analytic estimates. There are two reasons for this difference: (a) presently our implementation of `Stacker` uses 4-byte floating points to represent keypoint descriptors where 1-byte is sufficient (since values are discretized into 255 bins); and (b) when aligning a section to its identical copy all keypoints match which results in `Stacker` creating an "edge" between sections for each keypoint. I believe resolving these issues is straightforward, but even with `Stacker`'s current memory consumption all datasets but the human brain remain easily in reach.

Chapter 4

Frugal Snap Judgments

This chapter describes a technique called *frugal snap judgments* (FSJ) that is used for image alignment to significantly improve performance ($\approx$ 5x) without having an appreciable impact on accuracy. It can also help reduce the memory requirement of image alignment.

Motivation

Downsampling is a common technique for reducing the computational cost of image processing pipelines. Indeed, other image alignment pipelines employ downsampling to reduce the computational burden of identifying useful keypoints in the image.

Discussions with practitioners performing image alignment on Connectomics datasets, however, revealed a wariness towards downsampling due to its potential to introduce errors. Their rationale is twofold: (a) the alignment errors introduced in 2D alignment, while small, can accumulate when performing stitching of a large section resulting in unnatural deformations that are difficult to correct; and (b) even small misalignments (e.g. 8 pixels) are sufficient to alter fine details in certain neuronal structures such as the spine necks of dentrites.

Indeed, as shown in Figure 4-1, operating on 30% scale images results in significant relative alignment errors between tile pairs. In fact, 0.02% of tile pairs had a 50 pixel or greater relative alignment error. While 0.02% sounds small, that error rate

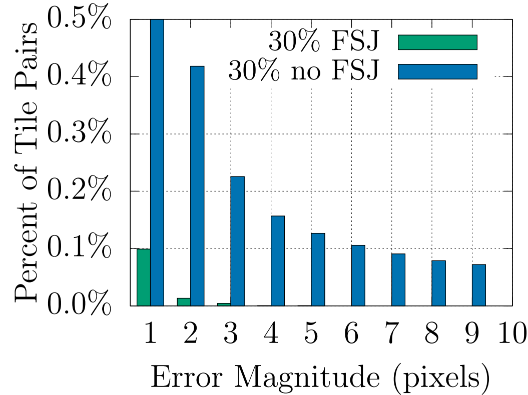


Figure 4-1: Error with and without FSJ on human dataset.

implies the existence of tens of serious alignment errors in every $2mm^2$ section of the *human100* dataset.

The potential performance advantages of downsampling, however, were too alluring to pass up. Thus, I developed the technique of frugal snap judgments to allow us to obtain more advantageous performance–accuracy trade-offs.

Using frugal snap judgments, our system learns to identify when the result of the fast alignment algorithm is likely to match the result of the more-accurate code. The learned criteria is not based upon an objective notion of correctness (which would be costly to compute). Instead, the criteria is based upon intermediate results generated by the fast alignment code that are analyzed to extract a measure of reliability.

Design of FSJ in `Quilter`

Let us now describe how to apply frugal snap judgments to improve the performance of `Quilter`. Frugal snap judgments are specifically used to optimize the algorithm used to compute corresponding keypoints between two overlapping tiles.

Fast-path Algorithm The fast-path algorithm for computing pairwise tile alignments performs in-memory downsampling of the tile images by a factor of 30%. For approximately 2% of tiles and 0.4% of all tile pairs, these modifications result in a less accurate algorithm relative to the original code path operating on full-resolution images. Yet, for the remaining tile pairs the fast-path computes a result that matches

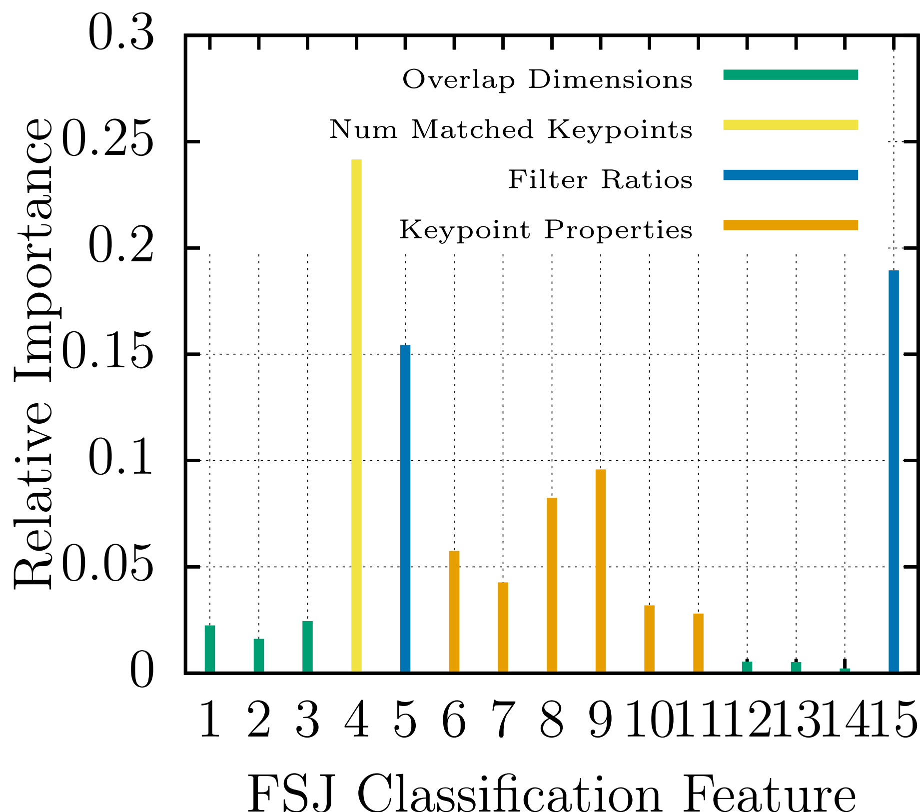


Figure 4-2: Relative importance of variables used by FSJ on *human100* dataset. Variables 1-3 and 12-14 are features describing the area and dimensions of the overlapping region between the pair of tiles in their original position and when aligned relative to each other using the result of the fast path. Variables 5,15 are ratios before and after keypoint filtering stages; 5 is the matches to keypoint ratio; and 15 is the RANSAC-filtered matches to keypoint ratio. Variables 6-11 measure aggregate statistics of keypoint properties in the overlapping region: 6-7 are the average "strength" of keypoints, 9-10 are the average size, and 11-12 are the average octave.

the slow-path within a tolerance of 1-2 pixels.

Detecting Unreliable Fast-Path Results To detect when the result of the fast-path is unreliable, I employ random-forest classification over feature vectors that summarize the intermediate results of the fast-path algorithm.

I employ feature vectors of dimension 15, which include the following information: the area, width, and height of the overlapping region between a pair of tiles, the fraction of keypoints matched, the fraction of matched keypoints that are filtered by RANSAC, the total number of filtered keypoints, and aggregate statistics from the

filtered keypoints. Additional details are provided in Figure 4-2. Note that no direct information about the pixel data within the tiles is used to build the feature vector used by FSJ.

Training the Fast-Path Detector I train our detector for determining when the fast-path is unreliable by randomly selecting 10,000 pairs of tiles from the stack and running both the fast and slow path algorithms on the same input. I compare the computed relative offsets between the tiles and if the fast and slow path differ by more than 1 pixel, I add the fast-path feature vector to our training set as a negative example. For positive examples, I require that the fast and slow path agree within 0.5 pixels. I use an implementation of random forests within OpenCV for training.

Training on $10,000$ tile pairs consistently proved sufficient to develop high-quality classifiers for each dataset. On each dataset, the training process resulted in a classifier with a false-positive rate of less than $\approx 0.5\%$ and a false-negative rate of $\approx 4-10\%$, depending on the resolution of the fast path and the dataset. Training time did not depend on the dataset and took approximately 10-20 minutes on an 18-core Intel Xeon CPU (E5-2666 v3, 2.9GHz).

After training on the *human100* dataset using 30% resolution images in the fast path, I achieved an out-of-bag error of $6.7e^{-2}$, a false-positive rate of 0.4% and a false negative rate of 6%. The relative variable importance scores for the random forest classifier are provided in Figure 4-2.

The accuracy of `Quilter` when using FSJ is substantially more accurate than one would predict from the classifier's false-positive rate of 0.4%. Figure 4-1 shows the 2D alignment errors of `Quilter` when using FSJ on a set of four sections that were not used during training. When using FSJ, there are nearly no errors greater than one pixel, and none greater than five pixels.

Chapter 5

System Evaluation

This section provides end-to-end performance results for the alignment pipeline composed of `Quilter` and `Stacker` on three datasets and across three computing platforms. I illustrate that `Quilter` and `Stacker` have good weak-scalability when scaled horizontally across multiple nodes in a cluster and can scale vertically within larger multicores.

Experimental Setup

A summary of the software, hardware, and datasets employed in our evaluation are as follows.

Software. The alignment pipeline composed of Quilter, Stacker, and Frugal Snap Judgments was implemented as a C++ software library parallelized using Cilk Plus [12, 40] and the Tapir [61] branch of the LLVM [38, 39] compiler (version 6).[1] The following software libraries were used: OpenCV v3.2.0 [14], OpenJPEG v3.2.0 [25], and Google protocol buffers [30].

Our evaluations employ three different computing platforms to evaluate runtime performance: *Common Multicore*, *Large Multicore*, and *Cluster*.

Common Multicore is an 18-core, 2-way hyperthreaded Intel Xeon CPU (E5-2666 v3, 2.9GHz) available as a 4th-generation compute-optimized machine from Amazon

[1] Available from `http://cilk.mit.edu`.

Web Services, which has 64GB of memory and runs Ubuntu 14.04 on Linux Kernel 3.13.0-106. Amazon EFS [3] (elastic filesystem) is used to store image data files. Amazon EFS provides performance tiered to the size of the mounted volume. Based on these tiers and our mounted volume size of 2.8TiB, our maximum burst I/O throughput was 100 MiB/sec during our experiments on this platform.

Large Multicore is a 112-core, 2-way hyperthreaded Intel Xeon Platinum 8180 CPU 2.5GHz with 1.5TB of memory running Centos7 on Linux Kernel 3.10.0-862. This machine was part of the Odyssey Cluster and retrieves stored image data from Lustre mounted storage connected via 56 Gb/s FDR InfiniBand network.

Cluster is a shared supercomputing center LLSC (Lincoln Lab Supercomputing Center) [54] using the AMD Opteron partition of the TX-Green system, consisting of 274 compute nodes, each containing two 16-core AMD Opteron(TM) Processor 6274, running at 2.2GHz, for a total of 32 cores per node, with 128 GB per node. The nodes employ a shared Lustre filesystem and are connected with a 10 GigE Arista switch. A total of 170 Opteron compute nodes were available for our experiments on this platform. Experiments run on more than four nodes were performed by a third party with guidance. Due to uncertainty regarding the type of compute nodes that would be available on *Cluster*, I cross-compiled the software and packaged dynamic libraries conservatively using the `-march=sandybridge` compiler directive.

Datasets Three different datasets were employed in our evaluations: *mouse50*, *mouse200*, and *human200*. *Mouse50* is 550GB dataset composed of 50 sections and 65,000 image tiles. *Mouse200* is a 2TB dataset composed of 200 sections and 200,000 image tiles. *Human100* is a 100-section 38TB dataset.

End-to-End System Experiments

Figure 5-2 illustrates the the absolute runtime obtained on the *mouse200*, *human100*, and *mouse50* datasets on the *Common Multicore*, *Large Multicore*, and *Cluster* systems. Due to the difficulty of moving the full *human100* dataset, I only ran a full end-to-end

Dataset	Z	Size	Description
mouse200	200	2 TB	Subset of 100umSept2017 dataset from $100\mu^3$ volume of mouse brain, stored in J2K compressed format.
mouse50	50	0.55 TB	Subset of 100umIARPASep14 dataset from $100\mu^3$ volume of mouse brain, stored in JPEG compressed format.
human100	100	38 TB	ROI2w05 which is a subset of a 600 TB dataset obtained from human brain biopsy, stored in J2K compressed format.

Figure 5-1: Dataset descriptions. The Z column provides the number of $2D$ sections within the dataset. All datasets were obtained from the Lichtman Lab using the Zeiss multiSEM microscope.

Dataset	FSJ	Hardware	Runtime	Throughput
mouse200	FSJ30	170 Opteron nodes (5440 cores)	16 minutes	7.5 TB/hr
mouse200	FSJ30	112-Core Intel Xeon Platinum 8180	80 minutes	1.5 TB/hr
human100	FSJ30	112-Core Intel Xeon Platinum 8180	26.7 hours	1.4 TB/hr
mouse50	FSJ30	18-Core AWS C4 Instance	49 minutes	0.67 TB/hr
mouse50	FSJ20	18-Core AWS C4 Instance	40 minutes	0.82 TB/hr

Figure 5-2: End-to-end performance results for whole alignment pipeline executing both `Quilter` and `Stacker`. In the FSJ column, FSJ30 and FSJ20 indicate that the fast path employed by FSJ used 30% and 20% resolution images respectively.

test for this dataset on the *Large Multicore* platform.

Vertical Scalability

I ran experiments to evaluate the vertical scalability of `Quilter` and `Stacker` on the *Large Multicore* platform. For the *mouse200* dataset, I ran the alignment pipeline on four sections using 1-112 cores. For executions on 28 cores or fewer, I executed the entire alignment pipeline as a single, shared-memory process. For 56 core, I aligned the first two sections on socket 0 and the second two sections on socket 1, and then I ran `Stacker` on the boundary sections to complete the $3D$ alignment. For 112 cores, I

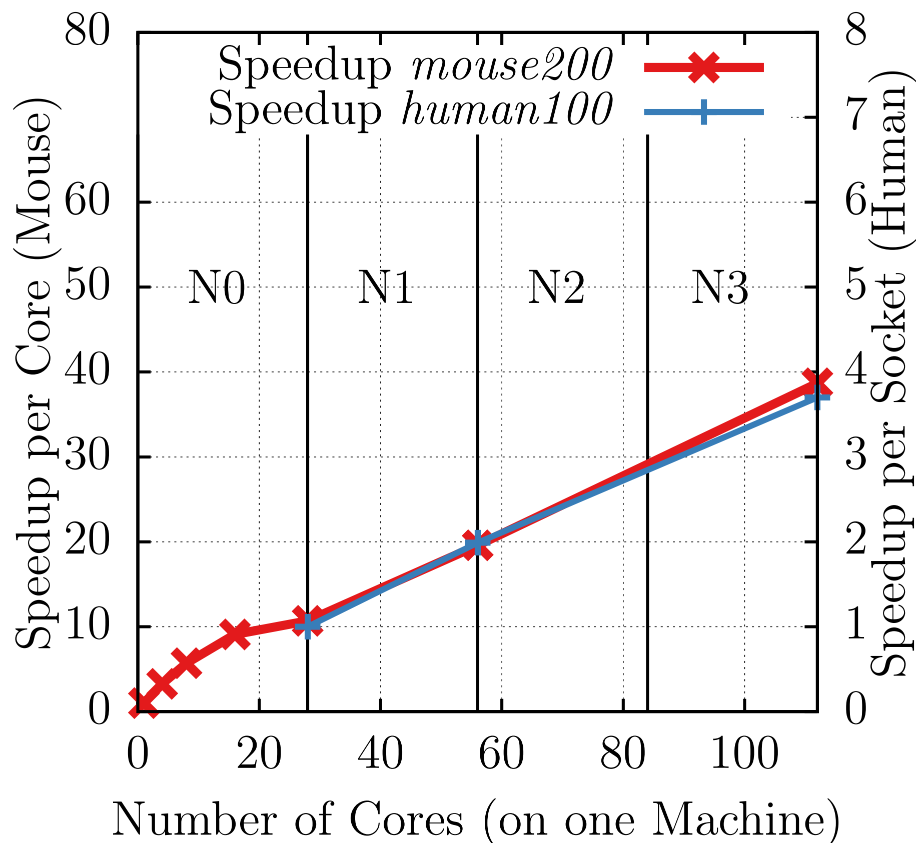


Figure 5-3: Vertical scalability. Reports speedup obtained when executing 4 sections of *mouse200* and 4 sections of *human100* on the *LargeMulticore* platform. The *mouse200* scalability is given relative to a 1-core executing using the left-axis of the plot. The *human100* scalability is given relative to a 1-socket execution using the right-axis of the plot. The mapping of cores to sockets is provided via the $N0, N1, N2, N3$ labels.

ran `Quilter` on each section on separate sockets, and then I ran `Stacker` on 3 sockets to compute the $3D$ alignment of the stack. A similar experiment was run on 4 sections of the *human100* dataset, but I did not perform runs that used fewer than 28-cores, because such executions are time-consuming.

Figure 5-3 illustrates the results of our vertical scalability experiment on four sections from *mouse200* and *human100*. Note that, since I did not run *human100* on less than a full socket, that the scalability needs to be evaluated using separate scales. The left y-axis provides the scalability of the *mouse200* experiment relative to a 1-core serial execution. The right y-axis provides the scalability of the *human100* experiment relative to a 1-socket execution. On the *mouse200* dataset, approximately 10x speedup is achieved on 28-cores relative to a serial 1-core execution. Executions of

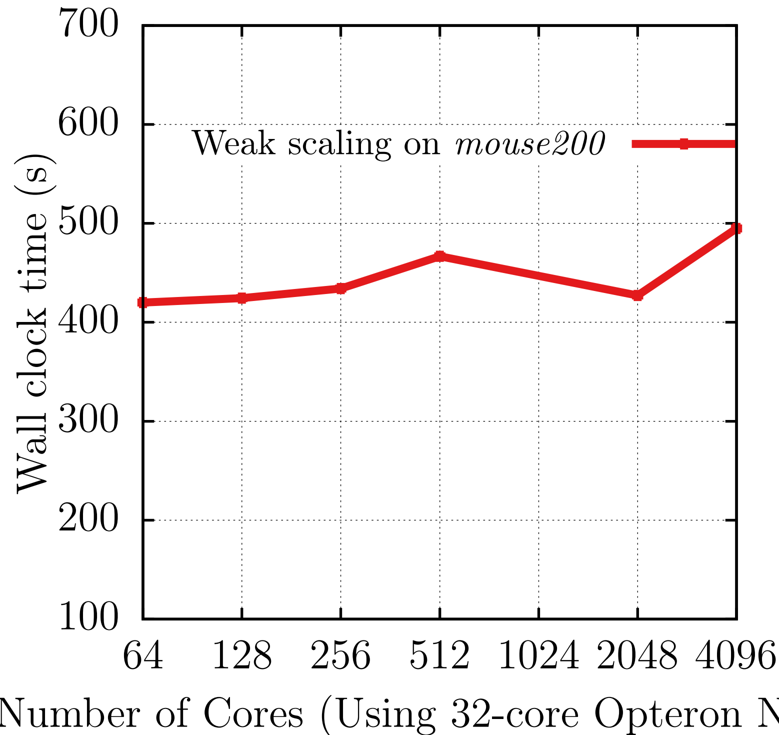


Figure 5-4: Weak scalability. Illustrates reported runtimes on the *Cluster* platform on the *mouse200* dataset when the stack input-size scales with the number of nodes used to execute `Quilter` and `Stacker`.

`Quilter` and `Stacker` achieve near-perfect linear scalability when parallelized using a separate process-per-socket on both *mouse200* and *human100*.

Weak Scalability

I evaluated the weak scalability of `Quilter` and `Stacker` using the *Cluster* platform. Since the work-per-section in a real-world dataset can vary, we constructed a synthetic dataset to evaluate weak-scaling. The synthetic dataset was constructed by taking a section from the *mouse200* dataset and creating a new stack of N images where that section was repeated N times. Then, when running on N compute nodes, I performed alignment on the synthetically generated dataset of N sections.

The results of the weak-scalability experiment are presented in Figure 5-4. On the synthetic *mouse200* dataset, I observed no appreciable decrease in efficiency-per-node when scaling from 64 to 4096 Opteron CPUs. Preliminary results from an analogous

2D Alignment Method	Runtime	Throughput
FijiBento Original	1050 minutes	0.032 TB/hr
FijiBento Improved	362 minutes	0.091 TB/hr
Quilter Full Resolution	180 minutes	0.18 TB/hr
Quilter FSJ(20,100)	32 minutes	1.03 TB/hr

Figure 5-5: Performance comparison of FijiBento and Quilter for 2D Alignment.

experiment using the *human100* dataset show similar weak scalability, but I only have data for 128 and 256-core executions on *human100* and am awaiting receipt of more data.

Performance Comparison with FijiBento

I compared the performance of `Quilter` with the 2D alignment algorithm in FijiBento [55][2].

Figure 5-5 shows the performance of two versions of FijiBento on the *mouse50* dataset compared to Quilter. I discovered a bug in FijiBento's implementation of RANSAC that did not impact correctness, but had a severe impact on performance. I report the performance of FijiBento before my fix as *FijiBento Original*, and report the performance of FijiBento after my fix as *FijiBento Improved*. The performance of Quilter when operating on full-resolution image data is reported as *Quilter Full Resolution*, and the performance of Quilter when using frugal snap judgments with 20% resolution image data in the fast path is reported as *Quilter FSJ(20,100)*.

As shown in Figure 5-5, `Quilter` is approximately 2x faster than *FijiBento Improved* when operating on full resolution images without using frugal snap judgments, and is 11x faster than *FijiBento Improved* when using frugal snap judgments with 20% resolution image data used for its fast path.

For a run of *FijiBento Improved*, approximately 55% of the runtime was spent creating SIFT keypoints, and 40% spent performing keypoint filtering and matching.

[2]The times for fijibento do not include the time to do the mesh optimization since fijibento does this step separately.

Part II

Dynamic Graph Analytics

Chapter 6

Graphs

Graphs store information about objects and the relationships between them. The objects are the vertices, and the relationships are the edges. For example, the web can be studied as a graph, where each web page is a vertex, and each link is an edge. Using this graph model, we are able to learn information about the web with nothing other than this structure itself. For example, we can determine the most trusted web pages.

Formally, I define a graph as follows:

Definition 6.1 (Graph) *A graph*[1] $G = (V, E, w)$ *is a set of vertices*[2] V*, a set of edges* E*, and an edge weight function* w*. I denote the number of vertices* $|V| = n$*, the number of edges* $|E| = m$*, and the degree*[3] *of a vertex* $v \in V$ *is* $deg(v)$*. Each vertex* $v \in V$ *is represented by a unique non-negative integer less than* n *(i.e.* $v \in \{0, 1, \ldots, n-1\}$*). Each edge is a 2-tuple* (u, v) *where* $u, v \in V$*. Finally, the weight function* w *maps each edge* $e \in E$ *to a real value (*$w(e) \in \mathbb{R}$*).*

Graphs can be either sparse or dense. A dense graph has a sizable fraction of the total possible edges, that is to say that m is close to n^2. A sparse graph, in contrast, has relatively few edges, $m << n^2$. Different data structures are best to store graphs,

[1] I focus on weighted graphs. An unweighted graph would not have a weight function w (i.e. $G = (V, E)$).

[2] In other works, vertices are sometimes called nodes. For clarity, in this work, I will always call these graph elements vertices.

[3] I focus only on directed graphs and use degree to mean out-degree.

depending on each graph's density as well as how you want to access and modify the graphs data.

Most graph algorithms are memory bound more so than compute bound [7] due to unpredictable memory accesses and low compute per memory access. Historically, as graphs grew, distributed graph processing frameworks were created to help deal with the compute necessary to run analytics [28, 44, 48]. However, it was found that most graphs are not that big and shared or external memory systems were able to outperform distributed systems due to the lack of data movement necessary [1, 50]. Many of these systems were designed for static graphs, where the set of vertices and edges do not change over the course of the processing.

Static sparse graphs are often stored in compressed sparse row (CSR) format, which packs edges into an array and takes space proportional to the number of vertices and edges. Sparse storage formats pay for these space savings with the cost of updates. CSR format supports fast queries such as membership or finding all neighbors of a vertex, but may require changing the entire data structure to add or remove an edge.

However, many real-world graphs [67] are dynamic and sparse. For example, social networks such as Twitter and Facebook are highly dynamic graphs since new users and connections are added constantly. Twitter averages about 500 million tweets a day [60] and Facebook has about 41,000 posts (2.5Mb of data) per second [68].

Internet graphs are also highly dynamic: large Internet Service Providers (ISPs) field around 10^9 packets/router/hour [33]. Dynamic graphs have wide applications from recommendation systems to cellular networks and require efficient updates to graph storage formats.

A graph can also store a |V| by |V| matrix, and many graph operations can be thought of as matrix ones and vice versus. As such, another important operation for graph storage formats to be able to compute is sparse-matrix vector multiplication (SpMV), a widely-used kernel in numerical and scientific computing, which requires a scan over all nonzero edges. For example, iterative computations such as conjugate gradient are staples of numerical simulations and require repeated SpMV operations [56]. One application of dynamic sparse graph representations is control-flow analysis, which

involves successively extending a graph (adding vertices and edges) until it reaches a fixed point [64].

Related Work

I present a dynamic data structure called packed compressed sparse row (PCSR) independent of any framework. PCSR is a graph representation, rather than a dynamic analytics framework, and can supplement existing graph analytics solutions.

Existing dynamic graph analytics solutions such as GraphChi [37], LLAMA [47], and STINGER [6] [26] provide data structures for graph storage. These frameworks, however, often lack theoretical guarantees on performance. PCSR's performance guarantees may mitigate worst-case behavior in these graph frameworks.

Sha *et al.* [62] introduced GPMA, a GPU-based dynamic graph storage format based on the packed memory array (PMA). GPMA handles concurrent inserts and is optimized for parallel batch updates. In this work, I focus on sequential random updates for CPUs, rather than batched updates.

The most relevant related work is King *et al.*'s in-place dynamic CSR-based data structure (DCSR) for GPUs [36]. DCSR lacks theoretical guarantees on its runtime or space usage, however. Finally, it is only implemented for GPUs and not for CPUs.

Chapter 7

Graph Storage Formats

In this chapter, I describe the following graph storage formats: adjacency matrix, adjacency list, blocked adjacency list, and CSR. I detail their respective space/time trade-offs in Table 7.1.

Adjacency Matrix

An ***adjacency matrix*** is the most basic graph storage format. It stores an $n \times n$ matrix for a graph of n vertices. The entry at $[u, v]$ corresponds to the value of the edge (u, v) or has 0 if the edge does not exist. It excels in storing dense graphs because it does not store any pointers and therefore minimizes overhead for dense graphs, but is poor for sparse graphs since it explicitly holds all the 0's.

The adjacency matrix takes space $O(n^2)$ regardless of the density, which while optimal for dense graphs is very poor for sparse ones. Furthermore, adding a new vertex requires rebuilding the entire data structure. Finally, sparse graph traversals on adjacency matrices require iterating over the entire matrix of size n^2. Since the number of edges is $m \ll n^2$ for many sparse graphs, a graph traversal using an adjacency matrix is not work efficient.

Adjacency List

Another common sparse graph storage format is the ***adjacency list*** (AL). Adjacency lists keep an array of vertices, where each entry stores a pointer to a linked list of

	Adjacency Matrix	AL	BAL	CSR	**PCSR** (amortized)
Storage cost / scanning whole graph	$O(n^2/B)$	$O(n+m)$	$O((m+n)/B)$	$O((m+n)/B)$	$O((m+n)/B)$
Add new edge	$O(1)$	$O(1)$	$O(1)$	$O((m+n)/B)$	$O(\lg^2(m+n)/B)$
Update or delete edge from vertex v	$O(1)$	$O(deg(v))$	$O(deg(v)/B)$	$O((m+n)/B)$	$O(\lg^2(m+n)/B)$
Add node	$O(n^2/B)$	$O(1)$	$O(1)$	$O(1)^*$	$O(\lg^2(m+n)/B)$
Finding all neighbors of a vertex v	$O(n/B)$	$O(deg(v))$	$O(deg(v)/B)$	$O(deg(v)/B)$	$O(deg(v)/B)$
Finding if w is a neighbor of v	$O(1)$	$O(deg(v))$	$O(deg(v)/B)$	$O(\lg_B(deg(v))$	$O(\lg_B(deg(v)))$
Sparse matrix-vector multiplication	$O(n^2/B)$	$O(n/B+m+n)$	$O((m+n)/B)$	$O((m+n)/B)$	$O((m+n)/B)$

Table 7.1: Cache behavior of various sparse graph and matrix operations. $n = |V|$, $m = |E|$. The table lists various graph representations and the algorithmic runtime of common graph operations in the external memory model by Aggarwal and Vitter, [2] where B is the cache line (or disk block) size. The RAM model (without cache analysis) is the special case where B or $\lg(B)$ is 1. We analyze PCSR in the right-most column.
* We use a `C++` vector for our implementation of CSR, so we do not need to rebuild the vertex list every time we add a vertex.

edges. The pointer at index u in the vertex list points to a linked list, where each element v in the linked list is an outgoing edge (u, v).

Adjacency lists support fast inserts but have high space overhead and slow searches because the edges are unsorted. Adjacency lists also exhibit poor cache behavior because they lack locality. A variant of adjacency lists called ***blocked adjacency lists*** (BAL) uses blocks to store edges. BAL's exhibit faster traversals because of improved locality but require extra space for extremely sparse graphs. Blandford, Blelloch, and Kash [10] introduced a dynamic graph data structure based on BAL with many constant-factor improvements, but they stop short of giving theoretical guarantees. For simplicity, I compare PCSR with standard adjacency lists of various block sizes.

Figure 7-1 shows an example of a graph stored in adjacency list format.

Compressed Sparse Row

Compressed sparse row (CSR) is a popular format for storing sparse graphs and matrices. It efficiently packs all the entries together in arrays, allowing for quick

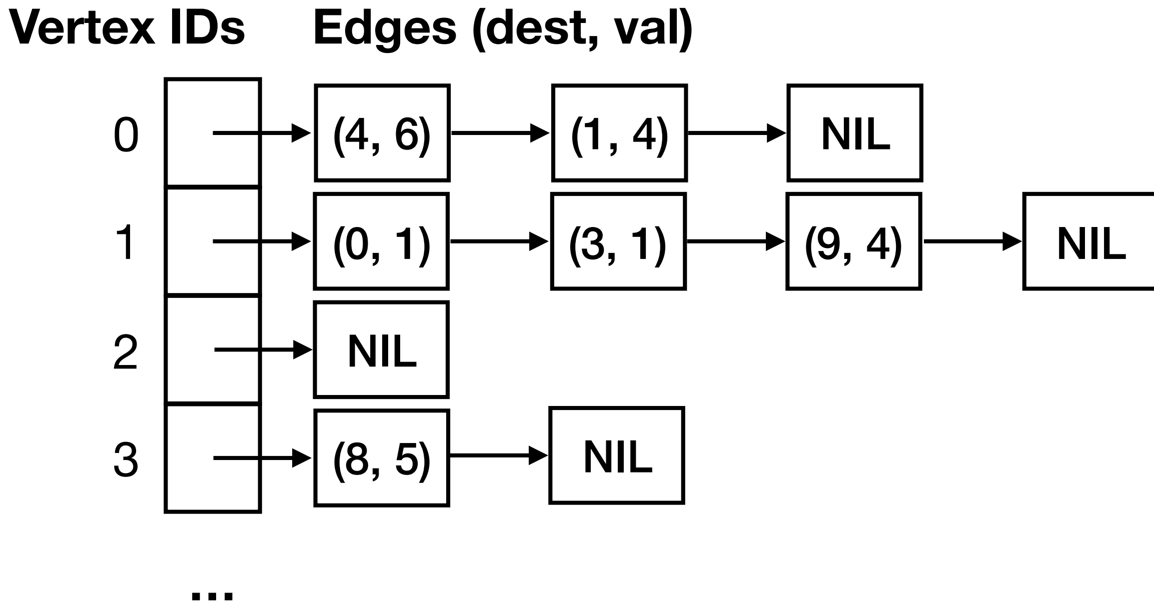


Figure 7-1: An example of a graph stored in an adjacency list. Each entry in the vertex array points to a linked list of edges. The vertex ID in the vertices array implicitly stores the source. For weighted graphs, I store a tuple of destination vertex and edge value for each edge.

traversals of the data structure.

CSR uses three arrays to store a sparse graph: a vertex array, an edge array, and a value array[1]. Each entry in the vertex array contains the starting index in the edge array, where the edges from that vertex are stored in sorted order by destination. The edge array stores the destination vertices of each edge. CSR stores a graph $G = (V, E)$ in size $O(|V| + |E|)$ but needs to be rebuilt upon any changes. Figure 7-2 contains an example of a graph stored in CSR format.

Inserting an edge into a graph in CSR format takes time linear in the size of the graph in the worst case. To insert an edge (u, v) into a graph in CSR format, CSR first must search all edges with source vertex u to find the edge with the smallest destination larger than v. Then, CSR inserts (u, v) into the edge list and slides all elements after that element over by one to make room. CSR then increments the elements in the vertex array for all vertices greater than u by one. The entire edge array may need to be resized and copied into a larger block of memory if there are

[1]not needed in the unweighted case

too many elements in the structure.

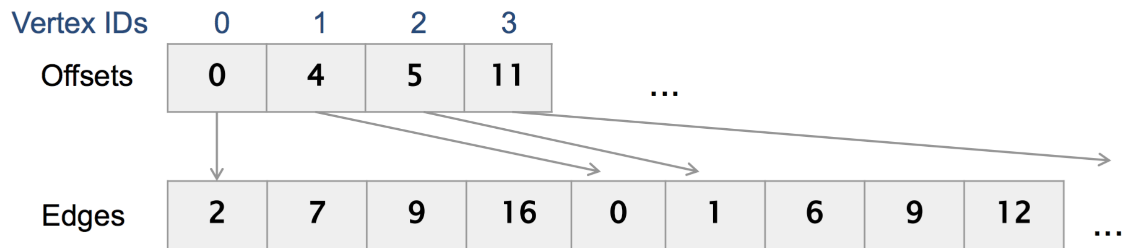


Figure 7-2: An example of an unweighted graph stored in compressed sparse row. The values stored in the edges array represent the destination. The vertex ID in the offset array implicitly stores the source. For weighted graphs, there is an additional values array.

Pinar and Heath [52] introduced a variant of CSR called Blocked Compressed Sparse Row (BCSR), where the locations of nonzero blocks are stored in CSR format. For our experiments, I compare to CSR for simplicity. Practitioners often use CSR for storing static social networks and random graphs.

Chapter 8

Packed Compressed Sparse Row

In this chapter, I discuss the design and guarantees of PCSR. However, first, we need to review the structure and theoretical properties of the packed memory array (PMA) [9].

Packed Memory Array

At a high level, a PMA is an array with spaces between its elements that uses a constant factor more space with polylogarithmic amortized inserts. More formally, a PMA holds n elements in $N = O(n)$ space and maintains order among its elements. It supports cache-efficient scans, i.e. reading S sequential elements takes $1 + S/B$ cache misses. Finally, it supports inserts in amortized $\mathrm{O}(log^2(n))$ work. For more details and proofs of time and space, see [8]. It does this by maintaining an array of size S with empty cells which enable inserting into any cell with a small number of shifts.

The PMA defines an implicit binary tree with leaves of size $\log N$ elements and height $\log(N/\log N)$. The internal nodes encompass the region defined by all of their descendants.

Each node of the tree has a density bound[1] proportional to the height of that node to maintain spaces between elements. The ***density*** of a node is the fraction of non-empty cells in its region. As the level of the node increases, the density bound

[1] In this paper, I use the density bound to upper bound the number of elements in any node. For the case of deletes, a PMA would enforce both an upper and lower density bound at each level.

decreases (lower bounded by 0.5).

The PMA enforces density bounds by redistributing elements to neighbor nodes whenever a node violates its density bound so that the densities of both sibling are equal.

When describing PMA operations and properties, I will refer to the array of elements as the pma and the space (in terms of the number of cells) as N.

Figure 8-1 shows an example of the implicit binary tree on the PMA intervals. If an interval becomes too dense, we walk up the tree and redistribute when an interval is appropriately dense.

Figure 8-1: An example of the implicit binary tree on the PMA's intervals. If we insert a new element in a leaf and the corresponding interval becomes too dense (shown in light grey), we walk up the tree until we find an interval with a density in the allowed range (shown in dark grey). In the worst case, we walk up to the root and rebalance the entire PMA. This figure is from [24].

Description of Structure

PCSR uses the same vertex and edge lists as CSR but uses a PMA instead of an array for the edge list. Adding both edges and vertices to the graph requires updates to both the vertex and edge lists. I use a `C++` vector with doubling and halving for the vertex list. Each element in the vertex list stores start and end pointers into the edge list for its range. Each non-empty entry in the edge list contains the destination

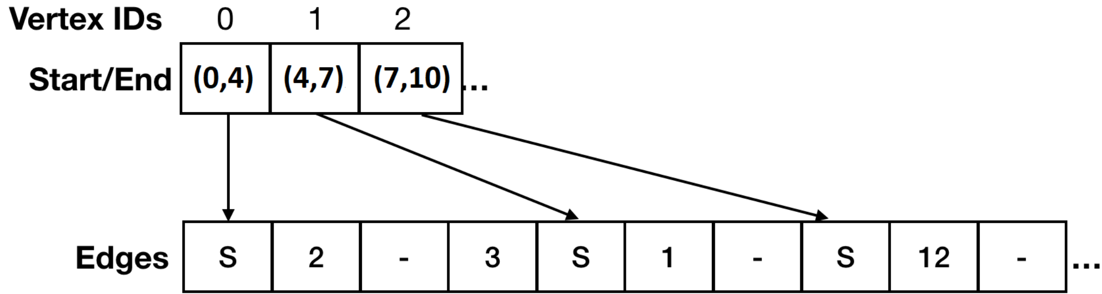


Figure 8-2: An example of a graph stored in PCSR. S denotes the sentinels. The ranges (start, end) in the vertex array denote the start and end of the corresponding edges in the edge array.

vertex and the edge value. Each vertex's range in the edge list has a corresponding sentinel entry in the edge list, which points back to the source in the vertex list for updating the vertex pointers.

I present an example of a graph stored in PCSR format in Figure 8-2.

The size of the vertex list is $O(n)$ since it stores two pointers for each vertex. The size of the edge PMA is $O(n + m)$ since it stores an entry for each edge and vertex. The size of a PMA is $O(N)$, where N is the number of elements in the PMA. Therefore, the total space usage of PCSR is $O(n + m)$, the same as in standard CSR.

Operations

Adding a vertex. PCSR adds vertices by extending the length of the vertex array by one, with a pointer to the end of the edge structure. Then PCSR adds the sentinel edge into the edge structure.

Adding an element to the end of the vertex structure is $O(1)$ and adding an element to the edge structure is $O(\lg^2(n + m))$, so the overall time is $O(\lg^2(n + m))$.

Adding an Edge. Adding an edge first requires finding the vertex in the vertex array, then requires a binary search on the relevant section of the edge array to insert the edge in sorted order, indexed by its destination. If a rebalance is triggered, PCSR checks every moved edge to see if it is a sentinel. If so, PCSR updates the vertex array with its new location.

Finding the location in the vertex structure is $O(1)$, binary searching the relevant

section of the edge array is $O(\lg(deg(v)))$, and inserting is $O(\lg^2(n+m))$, giving us $O(\lg^2(n+m))$ for the overall time.

Removing an Edge. Removing an edge is symmetric to adding an edge. PCSR finds the edge with a binary search, removes it from the PMA, and rebalances if necessary. Therefore, the runtime is the same as adding an edge: $(O(\lg^2(n+m)))$.

Removing a vertex. First, PCSR sets the start and end pointers into the edge array to null. PCSR also keeps track of the number of removed vertices and rebuilds the entire structure when the number of non-removed vertices equals the number of removed vertices. Vertices can only be removed after all their edges have been removed[2]. PCSR needs to mark the vertex in the vertex structure and remove the sentinels from the edge structure. This takes time $O(\lg^2(n+m))$.

To maintain the vertex list with $O(n)$ entries, PCSR simply rebuilds the structure every time the number of removed vertices exceeds half the number of vertices before vertex deletions.

I have not implemented removing edges and vertices, but their asymptotic performance is symmetric to adding edges and vertices.

[2]It would be possible to implement a faster bulk edge removal by deleting all the edges at once and rebalancing at the end.

Chapter 9

Empirical Evaluation

I evaluated PCSR against CSR, adjacency list (AL), and blocked adjacency lists (BAL). Adjacency matrix is not included due to its inability to scale to large sparse graphs. Each structure is evaluated on its performance and space usage, focusing on the sparse case since the adjacency matrix outperforms all other graph representations if the graph is dense. The test graphs are created by randomly generating variable numbers of edges in a graph with a constant number of nodes.

System

I ran our experiments on an Amazon Web Services (AWS) instance with 18 cores, with hyper threading, and a 2.9GHz clock speed. The machine had 64GB of RAM, 32K of L1 cache, 256K of L2 cache, and 25600K of L3 cache. Programs were written in C++ and compiled with GCC 4.8.5 with -O3. All programs were run sequentially.

Memory Footprint

I measured the memory footprint of each data structure for a fixed number of vertices and variable number of edges. Figure 9-1 shows the relative growth of the memory footprint of each graph representation.

The BALs use much more size than necessary when the average degree is small because most of the space in the blocks is empty.

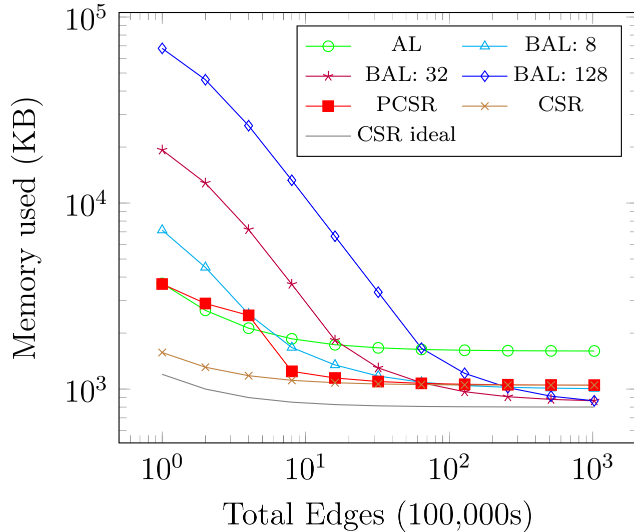


Figure 9-1: Size per 100,000 edges of each data structure with 100,000 nodes and a variable number of edges. The x-axis represents the number of edges, while the y-axis represents the size per 100,000 elements.

The C++ vector for the edge list in CSR doubled the speed of inserts since our implementation of CSR (on average) only needs to copy half of the elements on each insert. Therefore, I also compare to the ideal CSR size without extra space.

I found there is about a factor of two between the size of an ideal CSR, without extra padding, and the worst AL, while PCSR only has a space overhead of between 20% and 30%.

Inserts

I benchmarked the time to insert unique new edges in all the data structures. I generated edges uniformly at random without replacement. Figure 9-2 shows the time to insert $100,000$ edges with a fixed number of vertices and a variable number of edges. AL-based representations supported fast inserts, while CSR was the slowest. CSR starts about three orders of magnitude slower than all other representations and also scales much worse. Therefore, I was unable to run it for large numbers of edges. I found that, in practice, PCSR is about three to four times slower than AL-based representations.

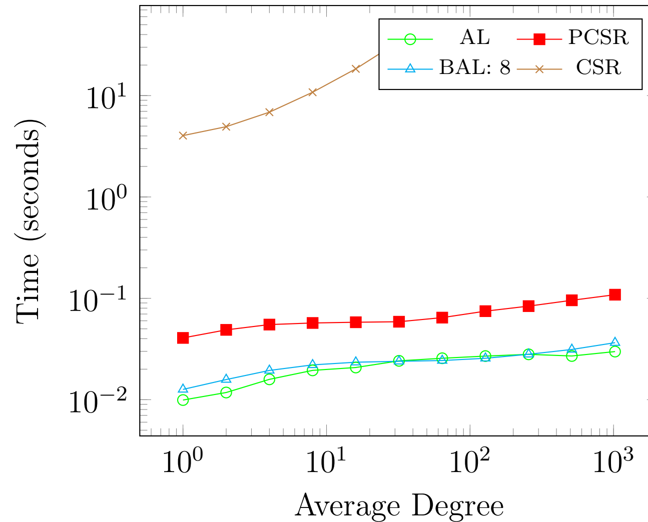


Figure 9-2: Time to insert 100,000 edges with a fixed number of nodes. We used 100,000 nodes and added a variable number of edges.

Updates

I benchmarked update operations on all the data structures. I generated edges uniformly at random with replacement. Figure 9-2 shows the time to insert edges that potentially exist in the edge list with a fixed number of vertices. The difference between update and insert is that update requires a search beforehand to check if the edge is already in the structure. I again show the time for updating (or inserting) $100,000$ edges.

PCSR outperformed all other structures when the average degree grew to reasonable sizes, as expected from Table 7.1. Once again, CSR is several orders of magnitude worse and is too slow to complete on reasonable input sizes. Additionally, the AL-based representations take linear time to search and take much longer than the $O(\lg^2(n))$ search time of PCSR. While the high search cost in AL-based representations can be somewhat offset by increasing the size of the block, larger block sizes increase the size of the AL and slow insertions.

Sparse Matrix-vector Multiplication

Figure 9-4a shows the time to perform a sparse matrix-vector multiplication using the different structures with 100,000 vertices and a variable number of edges.

Although the asymptotic complexity for SpMV is the same for all the structures,

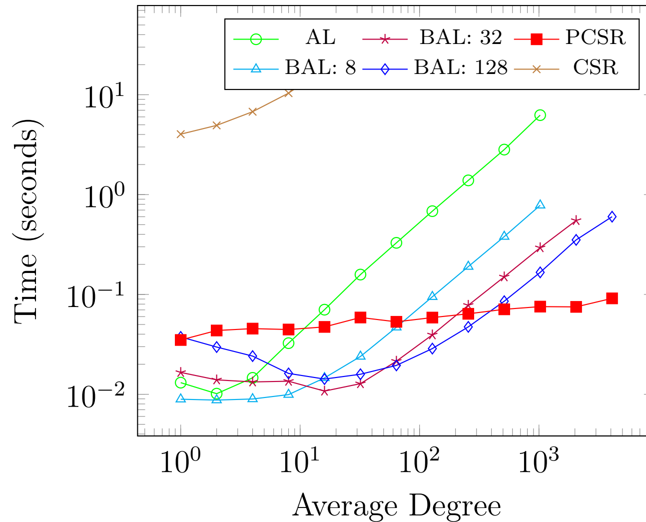


Figure 9-3: Time to insert or update 100,000 edges with a fixed number of nodes. We used 100,000 nodes and added a variable number of edges.

the AL-based structures can suffer from poor cache behavior. Increasing the block size in BALs can improve cache performance. PCSR avoids the problem of cache locality because it stores all its edges in a single array. SpMV takes longer in AL than PCSR because the PCSR has better cache behavior. SpMV in PCSR is within a factor of two and often within 20% of SpMV in CSR.

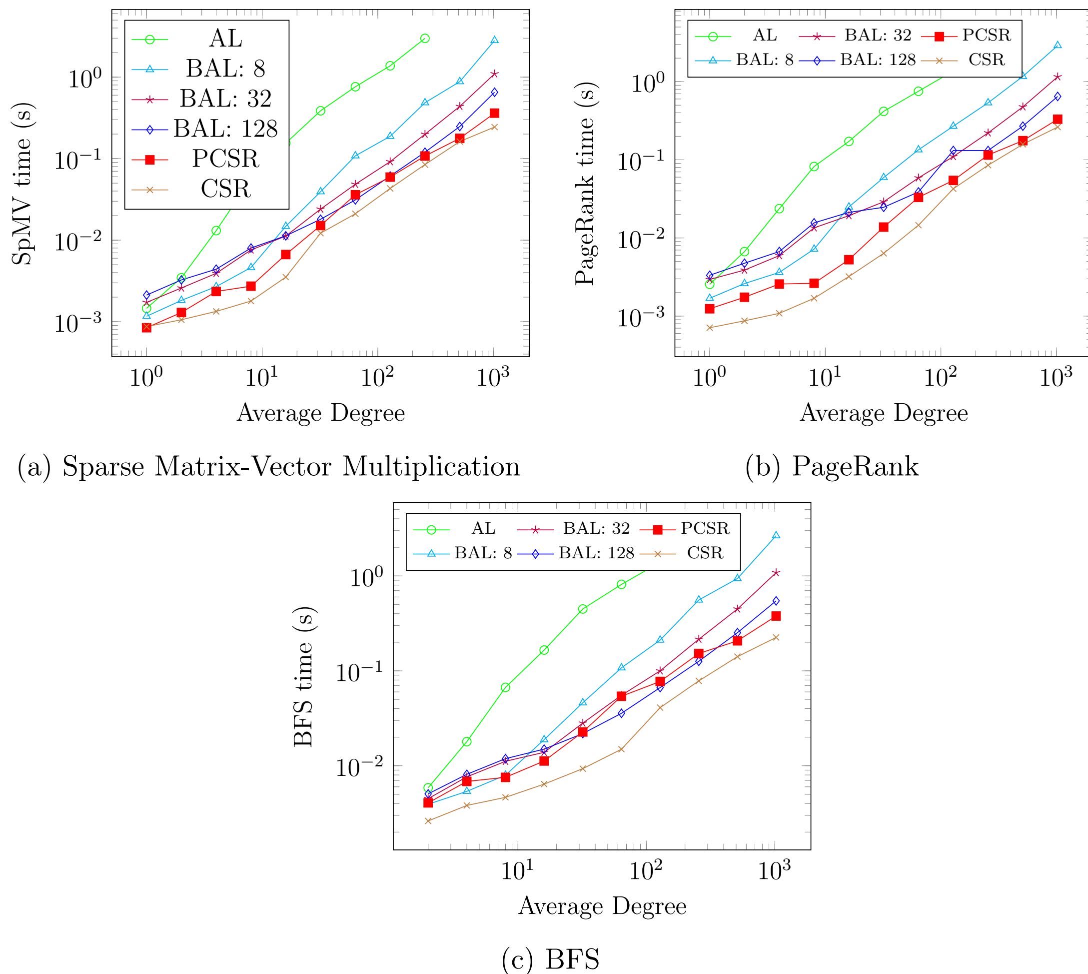
PageRank and BFS

Figure 9-4b shows the time to perform an iteration of PageRank using the different structures with 100,000 vertices and a variable number of edges.

Figure 9-4c shows the time to compute the distance to each vertex from a randomly chosen source vertex using each of the different structures with 100,000 vertices and a variable number of edges.

The time to perform a BFS and an iteration of PageRank scales with the number of edges in the graph in all representations.

PCSR achieved within 25% of CSR's runtime on most input sizes. CSR was the fastest, followed by PCSR and BAL-128. BAL with bigger blocks would perform even better (closer to CSR).



(a) Sparse Matrix-Vector Multiplication

(b) PageRank

(c) BFS

Figure 9-4: Time with 100,000 vertices and a variable number of edges. The x-axis represents the number of edges, while the y-axis represents the time.

Name	vertices	Edges
Slashdot	77,360	905,468
Pokec	1,632,803	30,622,654
LiveJournal	4,847,571	68,993,773

Figure 9-5: Sizes of social network graphs used in our tests.

Real World Graphs

I also tested on three social network graphs of varying sizes from the Stanford Large Network Dataset Collection and report our results in Table 9.1. They were Slashdot, Pokec, and LiveJournal; their sizes are detailed in Figure 9-5. The random graphs generated have a normally distributed degree distribution, while the social networks degree distribution follows a power law distribution [51]. This means that social

Graph Format	*AL*	*BAL 8*	*BAL 32*	*BAL 128*	*CSR*
Slashdot					
Size	0.88	0.82	1.47	4.71	0.41
SpMV	10.87	1.29	1.45	1.32	0.39
BFS	8.86	1.20	1.42	1.17	0.47
PageRank	13.38	1.64	1.85	1.72	0.36
Adding edges	0.25	0.25	0.25	0.25	525.00
Updating edges	10.50	1.25	1.00	0.75	508.75
Pokec					
Size	0.93	0.71	0.98	2.75	0.45
SpMV	15.95	2.43	1.21	1.17	0.51
BFS	7.25	1.64	1.02	1.00	0.48
PageRank	11.77	3.04	1.78	1.72	0.54
Adding edges	0.25	0.50	0.25	0.25	31628.50
Updating edges	9.17	2.50	0.83	0.67	21005.83
LiveJournal					
Size	1.05	0.87	1.36	4.00	0.49
SpMV	20.40	2.77	2.20	2.10	0.59
BFS	9.55	2.30	1.34	1.40	0.53
PageRank	16.20	5.36	2.40	2.73	0.54
Adding edges	0.25	0.25	0.25	0.50	70787.00
Updating edges	13.17	4.00	1.50	1.17	46835.00

Table 9.1: Real-world graphs. I tested on Slashdot, Pokec, and LiveJournal. All times are normalized against PCSR.

network graphs are much more likely to have vertices with high degree.

For adding and updating edges, I added 1000 random edges chosen without replacement with the same distribution as the edges in the original graph.

I found that PCSR was about a factor of two slower than CSR on graph computations but had much faster updates. The AL-based representations are a similar size to PCSR and were between two and ten times slower on graph computations but about four times faster adding edges.

Chapter 10

Parallel Packed Memory Array

I parallelized PCSR to allow for the work of individual operations to happen in parallel as well as to allow multiple operations to occur in parallel. To do this I first needed to create a parallel PMA.

I measure the asymptotic costs of a parallel operation on an input of size N in terms of the work $T_1(N)$ and span $T_\infty(N)$. The work is the total number of machine instructions and the span is the length of the critical path (or the runtime in instructions on an infinite number of processors). The parallel time $T_P(N)$ is bounded by $T_P(N) \leq T_1(N)/P + T_\infty(N)$ [20] assuming a PRAM model [35] with P processors and a greedy scheduler [13, 15, 31]. I use a model of parallel computation where N iterations in a parallel for loop has $O(\log N)$ span [20].

Operations

I will focus on searching and inserting elements into a PMA and define two ***external*** operations:

- `search`: finds an element in the PMA.
- `insert`: inserts an element into the PMA.

To implement and analyze these main operations, I define the following ***internal*** operations:

- `get_density`: returns the number of elements of each PMA leaf in a specified region.

- `redistribute`: redistributes elements in a PMA node.

- `double_pma`: doubles the size of the PMA.

All start and end indices of internal PMA operations must be at the beginning and end of PMA nodes, respectively (i.e. $s, t \mod \log(N) = 0$). Additionally, $(t - s)/\log(N) = 2^x$ for some non-negative integer x.

Parallel modifications

Next, I will discuss modifications to the PMA that I will use in my implementation of PPCSR.

I add an additional density bound to the nodes of the PMA for simplicity. The extra bound ensure that a thread can always insert an edge and will only wait in the redistribution phase of an insert. The new density bound requires that any node in the PMA cannot be full (the upper density bound cannot be 1). That is, the new density bounds at the leaves of the PMA are

$$D_{\text{leaf}} = [\alpha_{\text{leaf}}, \min(\beta_{\text{leaf}}, \frac{\log(N) - 1}{\log(N)})].$$

Since $\lim_{N \to \infty} (\log(N) - 1)/\log(N) = 1$, the additional density requirement does not impact the asymptotic behavior of the PMA.

I enforce a *packed-left* property of the nodes in the PMA so that inserts into one region do not spill over into others. Instead of evenly distributing elements in the PMA leaves, I put them all contiguously towards the beginning of the leaf. The packed-left property along with the non-full density bound ensure that a thread will never shift elements into another node's region.

The packed-left property maintains the cache-efficiency of the original PMA because the original PMA does a redistribute of each leaf after an insert into that leaf [8, 9].

```
1 # s is the start of a node
2 # t is the end of a node
3 # returns the number of elements in each leaf of a pma node
4  def get_density(s, t)
5   counts[((t - s) / log(N)]
6   par_for(i in [0, counts.size()):
7     leaf = i*log(N)
8     counts[i] = count_non_nulls(
9                   PMA[leaf, leaf+log(N))
10  return counts
```

Figure 10-1: Pseudocode for `get_density` in PPCSR.

This redistribute reads and writes each cell in the leaf. An insert with the packed-left property also reads and writes (at worst) each cell in the associated leaf.

The packed-left property also reduces the number of empty cells read in a pass through the PMA. Immediately after doubling a standard PMA half the cells will be empty. Reading all of the edges in such a PMA would read $\Theta(N)$ empty cells. A PMA with the packed-left property would read $\Theta(N/\log(N))$ empty cells because after reading any empty cell, you can jump to the next leaf (i.e. you will read at most one empty cell per leaf).

A PMA with the packed-left property minimizes the number of shifting elements between sibling nodes, which enables an efficient external-memory implementation. If each node at some internal level of the PMA is a separate file on disk, packing left will minimize the number of transfers between files.

Parallel PMA operations

In this section, I describe the PMA operations and show how to parallelize them. I also prove that all the PMA operations have polylogarithmic span.

Internal operations

First, I describe and analyze the internal operations of `get_density`, `redistribute` and `double_pma`.

The `get_density` function counts the elements in each leaf in a region. I specify a

`get_density(s, t)` function that returns the density of each leaf in a region specified by s, t where s is the beginning and t is the end of a region in the PMA.

Lemma 1 `get_density(s, t)` *has work* $O(t - s)$ *and span* $O(\log N)$.

PROOF. The outer loop has work $O((t-s)/log(N))$ and span $\log((t-s)/log(N)) = O(\log(N/\log N))$ because it iterates over all the leaves in the range. We can implement the `count_non_nulls` function with a simple serial for loop with work and span $O(\log N)$. Therefore, the total work is $O(t-s)$ and the span is $O(\log(N/\log N)) + O(\log N) = O(\log N)$. □

The `redistribute` function enforces the density bound of a region in the PMA. Specifically, the `redistribute(s, t)` function guarantees that all nodes in the region defined by s, t respect their density bounds.

Theorem 2 `redistribute(s, t)` *has work* $O(t - s)$ *and span* $O(\log N)$.

PROOF. The pseudocode[1] for `redistribute(s, t)` can be found in Figure 10-2.

By Lemma 1, the call to `get_density(s, t)` has work $O(t-s)$ and span $O(\log N)$. The first parallel for has $(t-s)/\log(N)$ iterations, for $O((t-s)/\log(N))$ work and $O(\log((t-s)/\log(N)))$ span. The inner loop has work and span $O(\log N)$ because it has $O(\log N)$ iterations. Therefore the total work and span of the nested loops is $O(t-s)$ and $O(\log N)$, respectively.

The prefix sum function on an array of size N can be implemented in parallel with $O(\log N)$ span and linear work [11].

The second parallel for iterates over the number of leaves, which is $(t-s)/\log N$, so the span of the second parallel for is $O(\log((t-s)/\log N) = O(\log(N/\log N))$. The `memcpy` will copy at most $O(\log N)$ elements, so it has work and span $O(\log N)$. Therefore, the work and span of this parallel for are $O(t-s)$ and $O(\log N)$, respectively.

The total work and span of redistribute are therefore $O(t-s)$ and $O(\log N)$, respectively. □

Finally, the `double_pma` function doubles the number of cells in the PMA.

[1] Unless otherwise specified, all division in pseudocode is integer division (rounded down).

```
1 # s is the start of a node
2 # t is the end of a node
3 # balances the elements equally among the leaves of the pma in
      the given region
4 def redistribute(s, t):
5   size = t - s
6   counts = get_density(s, t)
7   temp[size]
8   # done in-place
9   parallel_prefix_sum(counts)
10  # copy and pack all edges to temp
11  par_for(k in [s, t); k += log(N)):
12    if(i == s):
13      start = 0
14    else:
15      start = counts[i-1]
16    for(j in [k*log(N), (k+1)*log(N))):
17      if (pma[j] != null):
18        temp[start] = pma[j]
19        start++
20      pma[j] = null
21
22  num_leaves = size / log(N)
23  per_leaf = counts[-1] / num_leaves
24  extra = counts[-1] % count_per_leaf
25
26  par_for(i in [0, num_leaves)):
27    # numbber of items for this leaf
28    for_leaf = per_leaf + (i < extra)
29    # start of leaf's items in temp
30    j = per_leaf*i + min(i, extra)
31    # start of leaf in PMA
32    leaf = s + (i * log(N))
33
34    # copy edges into PMA
35    memcpy(&pma[leaf], &tmp[j], for_leaf)
```

Figure 10-2: Pseudocode for `redistribute(s, t)`.

```
1 def double_pma():
2   new_pma[2*N] = [0]
3   memcpy(&new_pma, &pma, N)
4   pma = new_pma
5   N = 2*N
6   redistribute(0, N)
```

Figure 10-3: Pseudocode for `double_pma`.

```
1 # the region between lo and hi is sorted
2 # returns the index of the element with value at least v
3 def search(lo, hi, v):
4   while (lo < hi):
5   mid = (hi - lo) / 2
6   # null case
7   if pma[mid] is null:
8     # gets beginning of next leaf
9     mid = ((mid / log N) + 1) * log N
10    # do a linear scan
11    # work is O(log N)
12    if mid > hi:
13      for(i in [lo, hi)):
14        if pma[i] >= v:
15          return i
16  # pma[mid] guaranteed to be non-null
17  if (pma[mid] is v):
18    return mid
19  elif (pma[mid] > v):
20    hi = mid
21  else:
22    lo = mid
```

Figure 10-4: Pseudocode for `search(lo, hi, v)`.

Lemma 3 *The* `double_pma` *procedure has work $O(N)$ and span $O(\log N)$.*

PROOF. The `double_pma` pseudocode can be found in Figure 10-3. Initializing the new pma of size $2N$ and copying over the old data has work $O(N)$ and span $O(\log N)$ since these operations take $O(1)$ work per cell. As shown in Theorem 2, redistribute also has work $O(N)$ and span $O(\log N)$. Therefore, `double_pma` has work $O(N)$ and span $O(\log N)$. □

External operations

Next, I show how to implement the external functions using the internal functions and analyze the external functions of `search` and `insert(lo, hi, v)`.

I define a search function `search(lo, hi, v)` which checks a sorted region of the PMA bounded by s, t (the beginning and end of the region, respectively) and returns the location of the smallest element that is at least e.

Lemma 4 `search(lo, hi, v)` *has* $O(\log(hi - lo))$ *work and span.*

PROOF. The pseudocode for the `search` function can be found in Figure 10-4. I modify a traditional binary search to deal with null values. If the midpoint pma[mid] is null, I set the midpoint to the beginning of the next *PMA leaf* in $O(1)$ instructions. Since I enforce the *packed-left* property in PMA leaves, the beginning of each leaf is guaranteed to be non-null. Checking whether a cell is null and computing the beginning of the next leaf take constant time. Suppose that at some level of the binary search hi - lo $= \ell$. The maximum size of the next step is $\ell/2 + \log N$. If $\log N \approx \ell/2$, meaning that I do not decrease the size of the next binary search step by a constant fraction, then I can just look at all the cells serially with work and span $\log N$. Otherwise, $\ell/2 + \log N = O(\ell/2)$ so we decrease the size of the search space by a constant fraction so I expect to take at most $\log(u))$ binary search steps. □

Next, I will describe and analyze the `insert` function. The `insert(lo, hi, v)` function inserts an element e into a sorted region beginning at index s and ending and ending at index t.

Inserting into a PMA takes amortized $O(\log^2 N)$ work [8] since the modifications do not add to the work.

Lemma 5 `insert(lo, hi, v)` *has* $O(\log^2 N)$ *worst-case span.*

PROOF. The pseudocode for the `insert(lo, hi, v)` function can be found in Figure 10-5. By Lemma 4, the `search(lo, hi, v)` function has $O(\log N)$ span. The slide-right function touches at most $O(\log N)$ cells of the PMA, so it also has $O(\log N)$ span. There are at most $O(\log N)$ calls to `get_density(s, t)` and `parallel_sum`, which each have $O(\log N)$ span by Lemma 1. Lastly, there is one call to either `double_pma` or `redistribute(s, t)`, which have $O(\log N)$ span by Lemma 3 and Theorem 2. □

```
1 # the region between lo and hi is sorted
2 # inserts the element v in sorted order of the elements
     between lo and hi
3 def insert(lo, hi, v):
4   level = log(N / log N)
5   height = level
6   index = search(lo, hi, v)
7   #slide elements to the right until a null space is found
8   slide_right(index)
9   pma[index] = v
10  # range of this leaf we inserted into
11  start = (index / log(N)) * log(N)
12  end = start_leaf + log(N)
13  counts = get_density(start, end)
14  # non-integer division
15  density = float(counts[0]) / log(N)
16  while density > density_bound(level):
17    # get start and end of parent nodes
18    start = get_parent_start(start)
19    end = get_parent_end(end)
20    counts = get_density(start, end)
21    # accumulate all non-empty cells in this region
22    count = parallel_sum(counts)
23    density = float(count) /
24      (log(N) >> (height - level))
25    level = level - 1
26    if level < 0:
27      double_pma()
28      return
29  redistribute(start, end)
```

Figure 10-5: Inserting into a PMA.

PMA operations in parallel

I now describe how augment a PMA with locks to support parallel writes. I assign one lock to each leaf of the PMA. Locking each leaf is equivalent to locking nodes at any set depth in the tree, which trades off between locking overhead and parallelism.

Grabbing locks in parallel To avoid linear span to grab the locks for a region, I show how to grab locks in parallel without deadlock and with polylogarithmic span.

There are two cases for grabbing locks: either inserting into a leaf or redistributing some subtree of the PMA. On an insert, a thread only needs to grab the lock of the

```
1 # locks is an array of the leaf locks
2 # the region [s, t) is a node in the pma
3 # grabs the locks from [s to t)
4 def grab_locks(s, t):
5   for(priority in [1, log(n)]):
6     par_for(i in [s, t)):
7       if popcount(i) is priority:
8         locks[i].lock();
```

Figure 10-6: Pseudocode for grabbing locks according to `lock_order`. We use s, t to denote start and end leaf indices in the region we are trying to grab locks for.

leaf it is trying to insert into. Since we enforce a stricter density bound on leaves in the PMA, an insert will never slide elements between subtrees. On a `redistribute`, a thread will grab all the locks in a subtree.

I will now describe a scheme for grabbing contiguous sequences of locks on *leaves* in parallel called `lock_order` according to implicit priorities of each leaf in the PMA. The `lock_order` algorithm first assigns implicit priorities to each leaf in the PMA depending on its index. The priority of a leaf with index i is `popcount`(i). The `popcount` function returns the number of ones in the bit representation of a number. For example, since $5 = 0b101$, `popcount`$(5) = 2$. I provide an example of how to assign priorities to nodes in Figure 10-7.

The `lock_order` algorithm grabs locks in a region from lowest to highest priority and grabs all locks of the same priority in parallel. We provide pseudocode for how to grab locks in a region in Figure 10-6.

Remark 10.1 *The* `popcounts` *in any subtree follow the same pattern as other subtrees at the same height in the tree, but the minimum popcount may differ. For example, consider leaves 0-3 and 4-7 in Figure 10-7, which correspond to two subtrees with roots at the same level. The* `popcounts` *of the leaves follow the same pattern but have different minimums in the different subtree. More generally, the position of the node in the PMA determines the minimum* `popcount` *of any of its leaf indices. For example, consider leaves 4-7 in the second: their minimum* `popcount` *is 1 because the upper bit must be set ($4 = 0b100$). The unique minimum priority of any leaf in a subtree is therefore the priority of the first leaf in that subtree.*

I now prove properties of our parallel locking scheme.

Theorem 6 *Grabbing locks for any two nodes in the PMA using* `lock_order` *is deadlock-free.*

PROOF. I will prove the theorem using case analysis. Suppose two threads are trying to grab locks for two nodes[2] a and b. I denote the set of leaves in the subtree rooted at some node α with leaves(α).

Case 1: leaves(a) $\cap$ leaves(b) $= \emptyset$. Since the regions have no locks in common, grabbing them in parallel will not cause deadlock.

Case 2: leaves(a) $=$ leaves(b). If $a = b$, there will be a unique leaf with lowest priority according to Remark 10.1. The thread that grabs it first will grab the rest of the region while the other one waits for it, avoiding circular wait.

Case 3: leaves(a) $\subset$ leaves(b) (w.l.o.g.). Let left$_a$ be the leftmost leaf in leaves(a). Since left$_a$ has smaller priority than all the other leaves in leaves(a), both threads will attempt to grab it before any other leaf in leaves(a). Therefore, whoever grabs left$_a$ will be able to grab leaves(a) first. There is no circular wait because the thread trying to grab the locks of a need no locks outside of leaves(a).

In all cases, there is no circular wait and therefore no deadlock. □

Theorem 7 *Grabbing all the locks for any node in the PMA according to* `lock_order` *has polylogarithmic span assuming $O(1)$ time to grab a lock.*

PROOF. I will analyze the span of grabbing locks by counting the number of leaves with the same priority. Suppose a thread is trying to grab the locks in some region associated with some node α in the tree with ℓ locks.

As described in Remark 10.1, exactly one of the leaves in the region has some minimum priority and that nodes at the same level of the tree follow the same pattern in their leaf priorities.

[2]These might be the same node.

Leaves	0	1	2	3	4	5	6	7
Priorities	0	1	1	2	1	2	2	3

Figure 10-7: The indices of leaves in a PMA and the associated priorities.

In the remainder of the proof I will assume w.l.o.g. that the first leaf in the region has priority 0 because the same counting argument applies but with a different minimum. The height of the node defines the number of unique priorities in the region because it sets the number of bits that can change among the leaves in the region.

The maximum priority of any index in a region of ℓ leaves is $\log \ell$ because the maximum index takes up $\log \ell$ bits. The number of leaves with some priority k is $\binom{\log \ell}{k}$ because there are k ways to choose which bits are set. I can grab all the locks with priority k in parallel with $\log(\binom{\log \ell}{k}) = O(\log \ell)$ span[3]. There are $\log \ell$ unique priorities with $O(\log \ell)$ span to grab the locks of each priority, for $O(\log^2 \ell)$ total span.

□

Since most operations take locks for a small region of the PMA (e.g. inserts or small redistributes), it is rare to have to wait on another thread with a lock.

[3]By just iterating over all possible ℓ bit-strings in parallel.

Chapter 11

Parallel Packed Compressed Sparse Rows

I will now describe how I use the parallel PMA to parallelize PCSR. For simplicity, I will focus on just a few graph operations, which the rest can be built from.

- `find_value` returns the weight of an edge or 0 if it is not in the graph.
- `find_neighbors` returns the neighbors of a vertex.
- `get_all_edges` returns a list of all the edges in the graph and their weights.
- `add_edge` sets the value of the edge if it is already in the graph, or adds it if it is not yet in the graph.
- `add_vertex` adds a new vertex to the graph.

Figure 11-1 contains an example of a graph stored in PCSR. We can see the differences from the serial version in that we now obey the packed left property and have a lock on each vertex.

Augmenting PCSR with locks

I now describe how we modify the PCSR format to support parallel writes with locks. I added locks to each vertex in the vertex array of the PCSR. Each lock corresponds

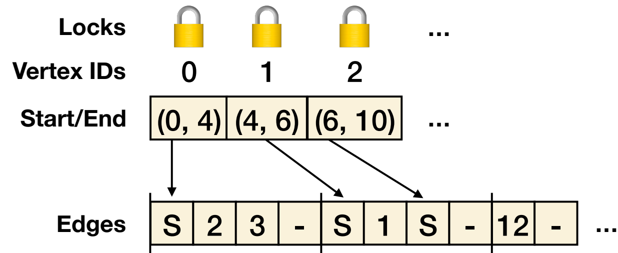


Figure 11-1: An example of a graph stored in PCSR format. “S” denotes a sentinel at the beginning of a vertex’s region in the edge PMA. The tall lines denote leaf boundaries and elements are packed left in leaves.

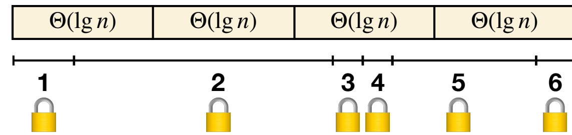


Figure 11-2: An example of the edge PMA in PCSR with locks on vertices. The boxes represent the leaf boundaries of the PMA and the lines under the PMA represent regions associated with vertices in the graph (with their corresponding locks).

to the region of the edge PMA that contains the outgoing edges of that vertex. Every cell in the edge PMA is in the region of exactly one vertex. When reading or writing to any cell, a thread must hold the corresponding lock.

In Chapter 10, I described how to lock a traditional PMA with one lock per node. In PPCSR, where there may be more than one lock per vertex from multiple vertices, grabbing all of the associated vertex locks can be done sequentially. If one lock encompasses multiple leaves, I can just assume the locks are at higher levels of the PMA tree.

I present an example of how the locks are distributed among PMA nodes in Figure 11-2.

Parallel work in graph operations

In this section, I show how each PPCSR operation can be computed with polylogarithmic span. I describe how to implement the PPCSR operations using the parallel PMA operations from Section 10.

```
1 def find-neighbors(u):
2   start = vertices[u].start
3   end = vertices[u].end
4   counts[end - start]
5   # end - start = O(deg(u))
6   par_for(i in [start: end) ):
7     if (edges[i] != null):
8       counts[i - start] = 1
9     else:
10      counts[i - start] = 0
11
12  parallel_prefix_sum(counts)
13  output[counts[end - start - 1]]
14  par_for(i in [start: end) ):
15    if (counts[i] > counts[i-1]):
16      output[counts[i-1]] = edges[i]
17  return output
```

Figure 11-3: Pseudocode for find-neighbors in parallel PCSR.

Read operations

I begin with the read-only operations `find_value`, `get_all_edges`, and `find_neighbors`.

I can implement `find_value(u,v)` directly with `search(lo, hi, v)` in the PMA. From Lemma 4,

Corollary 8 `find_value(u,v)` *has* $O(\log(u))$ *work and span.*

Next, I describe the `find_neighbors` function, which finds neighbors of a particular vertex in the graph. More formally, `find_neighbors(u)` returns a new set V_u such that $v \in V_u$ if and only if $(u, v) \in E$ and cannot return a pointer to somewhere in the data structure. The pseudocode for `find_neighbors(u)` can be found in Figure 11-3.

Lemma 9 `find_neighbors(u)` *in PPCSR has* $O(u))$ *work and* $O(\log(u)))$ *span.*

Proof. Each parallel for loop that iterates over $O(u))$ cells has work $O(u))$ and span $O(\log(u))$ because it iterates through $O(u))$ cells in parallel. The `parallel_prefix_sum` on an array of length N can be implemented with span $O(\log N)$ [11]. All of the other lines the function take $O(1)$ work. □

I use `find_neighbors` to implement `get_all_edges`, which returns a list of all

```
1 def get-all-edges():
2   num_edges[n+1]
3   num_edges[0] = 0
4   par_for(i in [0: n) ):
5     num_edges[i+1] = vertices[i].num_neighbors
6   parallel_prefix_sum(num_edges)
7
8   edges[num_edges[n]]
9   par_for(i in [0, n) ):
10    neighbors = find_neighbors(i)
11    par_for(j in [0: neighbors.size()) ):
12      edges[num_edges[i] + j] = (i, neighbors[j], weight(i, j)
          )
13  return edges
```

Figure 11-4: Pseudocode for get-all-edges in PPCSR.

the edges in the graph and their weights. I outline the `get_all_edges` procedure in Figure 11-4.

Lemma 10 `get_all_edges` *takes work* $O(m+n)$ *and has span* $O(\log n + \log(\Delta(G)))$.

Proof. For each vertex u, find-neighbors has work $O(u))$ and span $O(\log(u)))$ (from Lemma 9). The first parallel-for and the function `parallel_prefix_sum` has work $O(n)$ and span $O(\log n)$. The nested loop has work $O(n+m)$ because it does $O(1)$ work to iterate over each vertex and $O(1)$ work per edge in find-neighbors. The span of the nested loop is just the sum of the spans of outer and inner loops, which are $O(\log n)$ for the outer loop and $O(\log(\Delta(G))$ for the inner loop. Therefore, the work is $O(n+m)$ and the span is $O(\log n + \log(\Delta(G)))$. □

Write operations

I will now describe how to update the PPCSR data structure. PPCSR has two update operations: `add_edge` and `add_vertex`.

I begin by describing `add_edge` and showing how to implement it with parallel PMA operations. `add_edge(u, v, w(u,v))` sets the value of the edge $(u, v) = w(u, v)$. If the edge $(u, v) \notin E$, `add_edge` adds it to the graph with weight $w(u, v)$.

Theorem 11 `add_edge(u, v, w(u,v))` *has amortized* $O(\log^2(m+n))$ *work and* $O(\log^2(m+n))$ *worst-case span.*

PROOF. The `add_edge(u, v, w(u,v))` function updates the edge structure in PCSR. First, I do a search to check if the edge already exists: if so, we update its weight. This takes $O(\log(\deg(u))$ work and span by Lemma 4.

Otherwise, I need to insert a new edge using `insert`. We modify `insert` to handle moving sentinels (in `slide_right` and `redistribute`). This modification takes $O(1)$ work per edge because it checks if each cell contains a sentinel and if so, modifies the pointer to that sentinel in the vertex array. `insert` takes amortized work and worst-case span $O(\log^2(m+n))$ by Lemma 5. □

Next, I describe how to implement `add_vertex` with `add_edge`. The `add_vertex` function adds a new vertex with index n to a graph with n vertices and updates the edge structure with a sentinel.

Lemma 12 `add_vertex` *has amortized* $O(\log^2(m+n))$ *work and* $O(\log^2(m+n))$ *worst-case span.*

PROOF. The `add_vertex` function updates both the vertex and the edge structure in PCSR. First, `add_vertex` appends a new vertex to the end of the vertex array in amortized $O(1)$. If adding a new vertex triggers an $O(n)$ work copy, the copy has $O(\log n)$ span. I then insert the sentinel in the same way we inserted an edge using a call to `add_edge`. □

Chapter 12

Parallel PCSR Evaluation

In this chapter, I describe an empirical evaluation of parallel PCSR (PPCSR) and compare it with other sparse graph storage formats. PPCSR supports fast inserts and searches while still allowing for fast traversals through the entire data structure.

Experimental setup We implemented PPCSR as a C++ library parallelized using Cilk Plus [12, 40][1] and the Tapir [61] branch of LLVM [38, 39] compiler (version 6).

The experiments were run on a 8-core 2-way hyper-threaded Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz with 32GB of Memory from AWS [4].

Alternative data structures In addition to CSR and adjacency List I compare with two new structures in parallel. These are adjacency hash-map [20, 21] and Adjacency vector [22]. Adjacency hash-map is similar to Adjacency List, but supports O(1) searches enabling faster inserting when it is not known known if the edge already exists in the graph. Adjacency vector is the extreme case of Blocked adjacency list, but the block size is equal to the degree of the vertex, using table doubling to amortize the cost of adding to these arrays.

The `add_edge` function requires first checking to see whether the edge is in the structure and then either inserting it or updating its value. Therefore, to support fast implementations of `add_edge`, a graph storage format must support both fast searches

[1] Available at http://cilk.mit.edu.

and fast updates. The adjacency-list-based structures have fast updates since they do not store edges in sorted order, but then must do a linear scan to find any edge. CSR and PCSR support fast searches because they store edges in sorted order, but CSR requires potentially shifting the entire edge structure on an update.

Discussion of random graphs

I test the scalability of the graph operations as a function of the input size and the number of available threads. The test graphs were uniformly randomly generated with 100,000 vertices and a varying number of edges. I measure the size of the graphs in average updates per vertex[2]. I implemented the update function in batches of 10,000 edges at a time.

The adjacency list took too long for all operations on the largest graph I tested (4096 updates per vertex), so I do not include it in the results. The adjacency hash-map took too long to complete scanning operations on the largest graph, so I only include its results for the smaller ones.

First, I compared the performance of the `add_edge` function as shown in Figure 12-1. On very sparse graphs, the adjacency vector structure supports faster updates than PPCSR because it has to search through a small number of edges. Once the graph is sufficiently dense, PPCSR supports faster updates than adjacency vector because PPCSR has sublinear search time through each edge list. CSR took too long to update edges, so I do not include it in the results.

I implemented and tested sparse-matrix vector multiplication (SpMV) [56] and PageRank [69], two common computations on sparse graphs and matrices that require a scan, using the basic graph operations.

Next, I compared the data structures using the scanning operations SpMV and Pagerank. I find that the adjacency list and hash-map have poor performance due to pointer-chasing, which leads to bad cache behavior. CSR supports the fastest scans because all of the edges are contiguous in memory. In the adjacency vector, all edges

[2]Since there may be collisions, the number of updates is not exactly the average degree of each vertex. However, collisions are rare because the graphs are sparse.

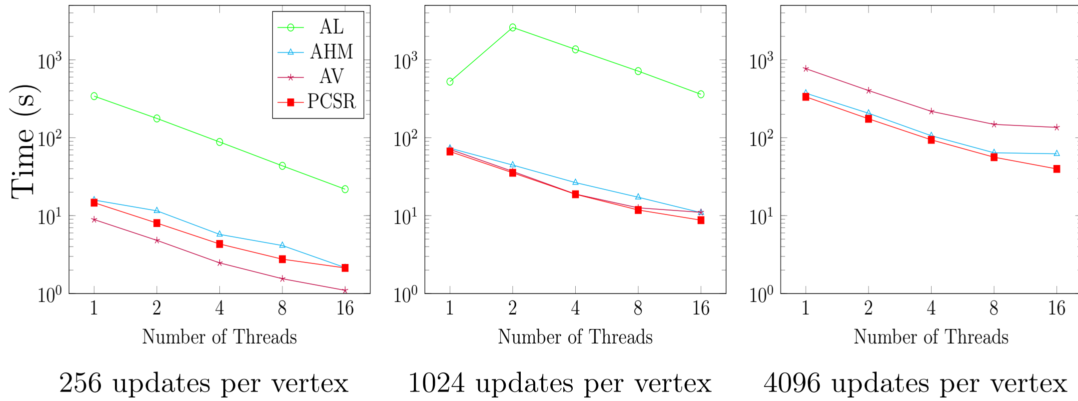


Figure 12-1: Time to dynamically create a graph with 256, 1024, and 4096 updates per vertex on average (from left to right) as a function of number of threads. Each graph has 100,000 vertex.

for each vertex are contiguous in memory, but moving onto the next edge list requires pointer chasing, so it is about a factor of 2 away from CSR. PPCSR has comparable performance to the adjacency vector because it does not require pointer chasing but not all of the edges are contiguous in memory because of the spaces between them. We present SpMV performance in Figure 12-2 and PageRank performance in Figure 12-6.

For inserting edges we find as expected that adjacency list does the worst. However, unexpectedly adjacency hash-map does not do that well and adjacency vector does very well. This is because for graphs with small average degree more time is spent in cache misses in the hash-map then just efficiently scanning through a vector. Adjacency vector stays on par with PPCSR until the average degree is about 1024, and adjacency hash-map is unable to beat PPCSR even at average degree 4096, even though it does asymptotically less work. This shows the real impact of optimizing not for work, but for memory transfers.

For both SpVM and Pagerank we get very expected results CSR being the best, adjacency vector and PPCSR trailing a constant factor behind, and the pointer based structures performing much worse

Discussion of social network graphs

I tested the data structures on the same three social networks from Chapter 9.

We see that with the relative low average degree that adjacency vector does the

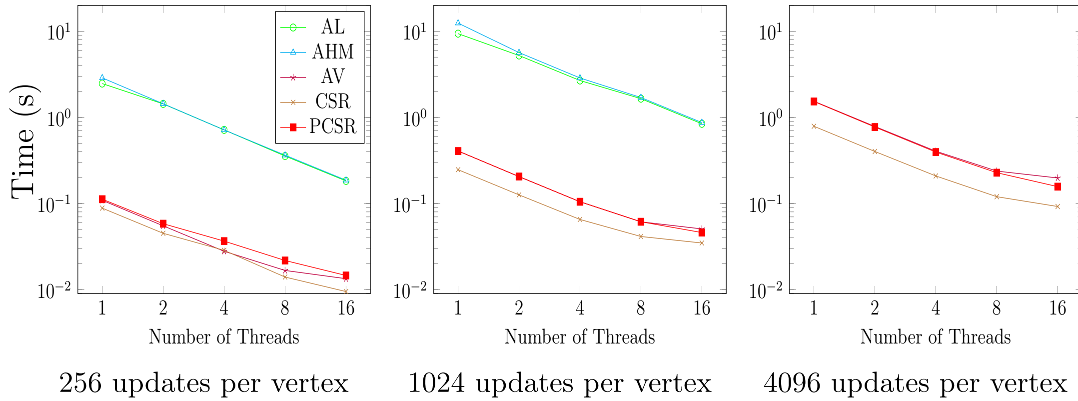


Figure 12-2: Time to compute sparse matrix-vector multiplication on a dense vector and sparse matrix store in each of the storage formats. The graph has 256, 1024, and 4096 updates per vertex on average (from left to right) and 100,000 vertices.

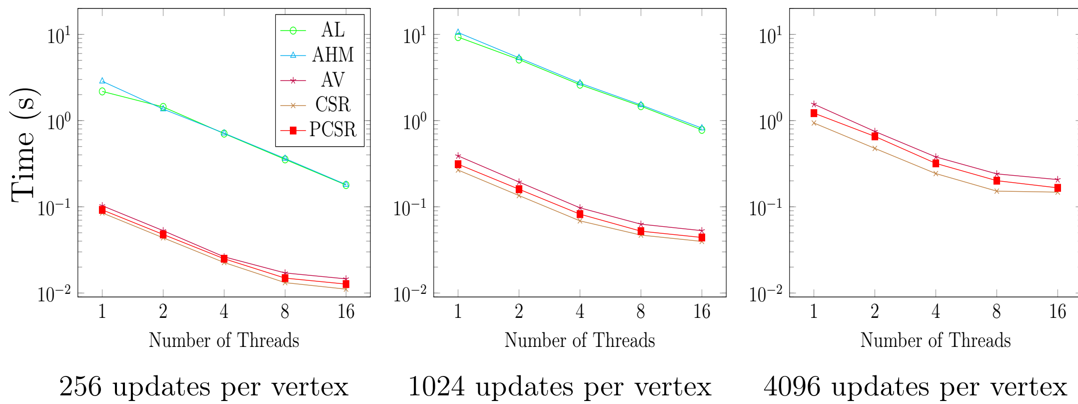


Figure 12-3: Time to compute one Pagerank iteration on each of the storage formats. The graph has 256, 1024, and 4096 updates per vertex on average (from left to right) and 100,000 vertices.

best for adding edges, but does worse with the scanning operations since there are now much more cache misses on the vertices themselves. Overall, we see that PPCSR is good balance that performs reasonably well on many different scenarios for many operations.

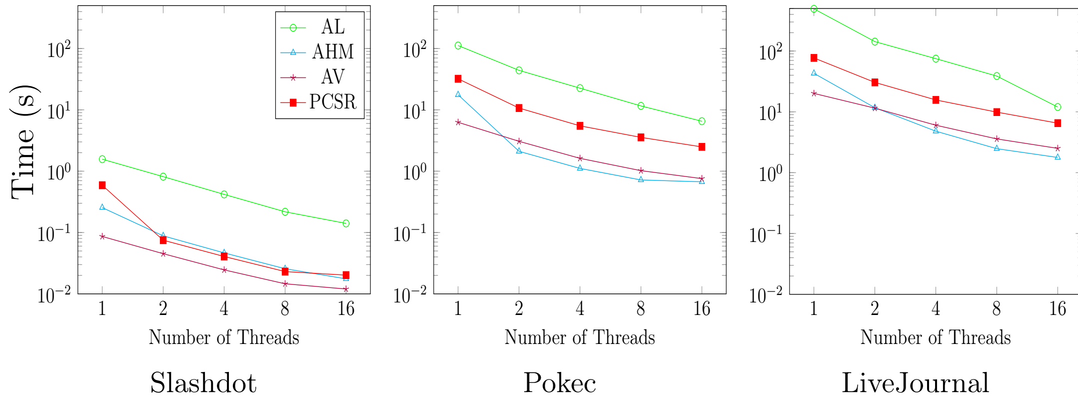


Figure 12-4: Time to construct various real-world graphs using the dynamic graph storage formats. The sizes of graphs can be found in Figure 9-5.

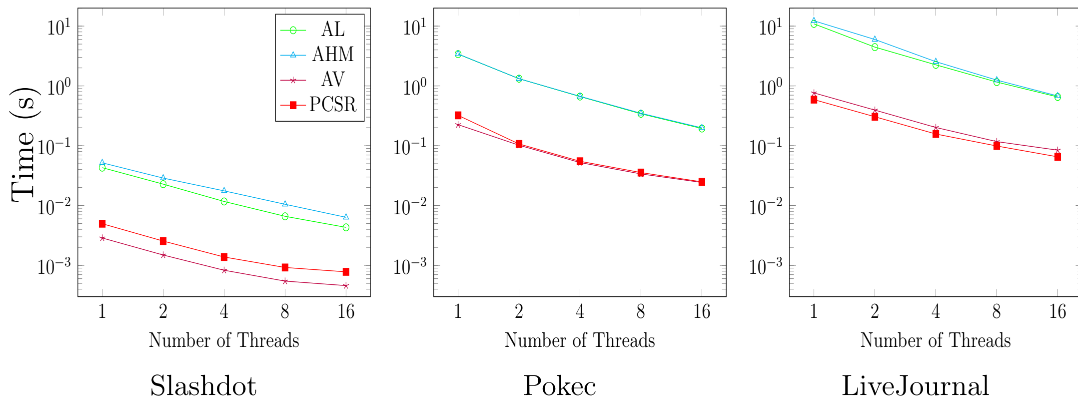


Figure 12-5: Time to compute a sparse matrix, dense vector multiplication using the dynamic graph storage formats. The sizes of graphs can be found in Figure 9-5.

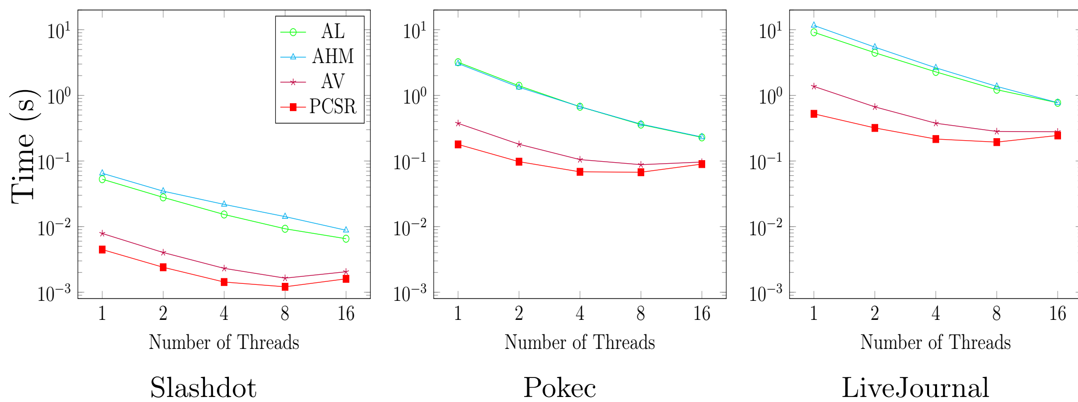


Figure 12-6: Time to run an iteration of Pagerank using the dynamic graph storage formats. The sizes of graphs can be found in Figure 9-5.

Conclusion

I have shown how a seemingly computation-bound application can be solved using a commodity multicore through the combination of careful performance engineering and algorithm design.

The alignment pipeline composed of `Quilter` and `Stacker` described in Part I provides the first fast and accurate multicore alignment pipeline that can align data at TB/hr pace using commodity multicore hardware without compromising alignment accuracy.

The design of `Quilter` and `Stacker` was, to an extent, multicore-centric. The development was principally performed using a single 18-core workstation and the main test dataset was the relatively small 550GB stack from *mouse50*. Through careful algorithmic design and careful consideration of how `Quilter` and `Stacker`'s memory requirements scale with dataset size, these algorithms easily scaled to datasets 100x larger. Furthermore, the standard performance optimizations I employed to improve performance in the shared-memory setting was translatable to the distributed cluster-computing setting through coarse-grained parallelization across sections facilitated by `Stacker`'s use of "associative" elastic transformations.

The performance of the alignment pipeline was achieved, in part, through aggressive downsampling of input images — an "optimization" that is often deemed unsuitable for Connectomics due to its demand for highly accurate alignments. As I have illustrated, however, much of the performance benefits of downsampling can be achieved without sacrificing accuracy by using machine learning techniques to distinguish between highly-reliable and not-so-reliable results produced by a fast, but sometimes inaccurate, code path.

I suspect that there are other instances where techniques such as frugal snap judgments may be applied, with little labor, to achieve more advantageous performance–accuracy trade-offs, especially in other image processing pipelines. Applying it to perform real-time object detection seems like a natural area to explore, especially in restricted settings (e.g. stop sign detection).

I have also described PCSR, a dynamic, graph storage format based on the packed memory array. I find that for slightly more storage and query time, I am able to achieve similar mutability speeds to that of the adjacency list. CSR was unable to handle many inserts in a reasonable amount of time. PCSR was orders of magnitude faster for inserts and updates than CSR, while maintaining similar graph traversal times. Lastly, I show how PCSR can be parallelized and achieve 8x speedups on an 8-core machine.

The growth of social networks and other dynamic graphs necessitates the need for efficient, dynamic graph structures. PCSR is a basic, dynamic graph storage format that can fit into existing graph processing frameworks and support fast insertions with comparable traversal times.

Appendix A

Connectomics

Connectomics is the creation and study of the map of connections in an organism nervous system. The goal is to generate a graph of the neurons that make up the tissue [42]. Neuroscientists hope to use this graph to help understand and treat a variety of psychopathologies.

For a more thorough introduction to the field of Connectomics see [41, 42, 57, 63, 66]

Connectomics Big-Data Challenge

The algorithms and systems designed for Connectomics must meet a grand "Big Data" challenge. [42, 49] Even small volumes of tissue produce data of staggering scale: a tiny $1mm^3$ volume produces petabytes of data; a $1cm^3$ volume (mouse brain) produces exabytes; and a $10cm^3$ volume (human brain) produces zettabytes (1 billion terabytes).

Systems capable of processing large volumes of high-resolution electron microscope imagery are vital to the science of Connectomics. It is not possible to merely operate on lower-resolution images or choose to process smaller volumes: fine-grained structure such as vesicles and dentritic spines are not visible at lower resolutions, and smaller volumes fail to capture whole-neurons whose typical length is greater than $1mm$ [42, 43]. Even if one's goal were only to analyze statistical properties of the human connectome, it would remain necessary to process large data sizes.

Connectomics Pipeline

The Connectomics pipeline is as follows:

1. The brain is embedded in a plastic-like laminar
2. The brain is sliced into 30 nm slices
3. Each slice is imaged with a electron microscope at 3 nanometer resolution[1]
4. The individual images are then globally aligned in both 2D and 3D
5. The images are segmented to identify neurons, synapses, and connections
6. These objects are traced through the 3D volume to create a full neural graph

Previous work has also been done on developing a pipeline for high throughput Connectomics on multicore machines, however, this work took aligned images as input and focused on steps 5 and 6 [63].

[1]These images are often compressed by as much as 10x using JPEG2000, or similar, compression algorithms.

Bibliography

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):20, 2017.

[2] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[3] Amazon. Amazon elastic file system. https://aws.amazon.com/efs/, 2019.

[4] Amazon. Amazon web services. https://aws.amazon.com/, 2019.

[5] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 20. ACM, 2013.

[6] David A Bader, Jonathan Berry, Adam Amos-Binks, Charles Hastings, Kamesh Madduri, and Steven C Poulos. Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. 2009.

[7] David A Bader, Guojing Cong, and John Feo. On the architectural requirements for efficient execution of graph algorithms. In *2005 International Conference on Parallel Processing (ICPP'05)*, pages 547–556. IEEE, 2005.

[8] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 399–409. IEEE, 2000.

[9] Michael A Bender and Haodong Hu. An adaptive packed-memory array. *ACM Transactions on Database Systems (TODS)*, 32(4):26, 2007.

[10] Daniel K Blandford, Guy E Blelloch, and Ian A Kash. An experimental analysis of a compact graph representation. 2004.

[11] Guy E Blelloch. Prefix sums and their applications. 1990.

[12] Robert D. Blumofe, Matteo Frigo, Chrisopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, June 1996.

[13] Robert D Blumofe and Charles E Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[14] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[15] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 1974.

[16] Kevin L Briggman, Moritz Helmstaedter, and Winfried Denk. Wiring specificity in the direction-selectivity circuit of the retina. *Nature*, 471(7337):183, 2011.

[17] David Capel and Andrew Zisserman. Automated mosaicing with super-resolution zoom. In *Computer Vision and Pattern Recognition, 1998. Proceedings. 1998 IEEE Computer Society Conference on*, pages 885–891. IEEE, 1998.

[18] Joe Chalfoun, Michael Majurski, Tim Blattner, Kiran Bhadriraju, Walid Keyrouz, Peter Bajcsy, and Mary Brady. Mist: Accurate and scalable microscopy image stitching tool with stage modeling and error minimization. *Scientific reports*, 7(1):4988, 2017.

[19] Dmitri B Chklovskii, Shiv Vitaladevuni, and Louis K Scheffer. Semi-automated reconstruction of neural circuits using electron microscopy. *Current opinion in neurobiology*, 20(5):667–675, 2010.

[20] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. 2009.

[21] cppreference. std::unordered_map. `https://en.cppreference.com/w/cpp/container/unordered_map`, 2019.

[22] cppreference. std::vector. `https://en.cppreference.com/w/cpp/container/vector`, 2019.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[24] Erik Demaine. Lecture notes in advanced data structures, March 2012.

[25] A Descampe, F Devaux, H Drolon, D Janssens, and Y Verschueren. Openjpeg 2.0. 0, 2012.

[26] David Ediger, Robert McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5. IEEE, 2012.

[27] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[28] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[29] Michael T Goodrich, Jyh-Jong Tsay, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pages 714–723. IEEE, 1993.

[30] Google. Protocol buffers. https://developers.google.com/protocol-buffers/, 2019.

[31] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[32] Douglas N Greve and Bruce Fischl. Accurate and robust brain image alignment using boundary-based registration. *Neuroimage*, 48(1):63–72, 2009.

[33] Sudipto Guha and Andrew McGregor. Graph synopses, sketches, and streams: A survey. *Proceedings of the VLDB Endowment*, 5(12):2030–2031, 2012.

[34] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 19–26, New York, NY, USA, 1993. ACM.

[35] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[36] James King, Thomas Gilray, Robert M Kirby, and Matthew Might. Dynamic sparse-matrix allocation on gpus. In *International Conference on High Performance Computing*, pages 61–80. Springer, 2016.

[37] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. USENIX, 2012.

[38] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, December 2002. *See* `http://llvm.cs.uiuc.edu`.

[39] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, page 75, Palo Alto, California, March 2004.

[40] Charles E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, March 2010.

[41] Jeff W Lichtman and Winfried Denk. The big and the small: challenges of imaging the brains circuits. *Science*, 334(6056):618–623, 2011.

[42] Jeff W Lichtman, Hanspeter Pfister, and Nir Shavit. The big data challenges of connectomics. *Nature neuroscience*, 17(11):1448–1454, 2014.

[43] Jeff W Lichtman, Hanspeter Pfister, and Nir Shavit. The big data challenges of connectomics. *Nature neuroscience*, 17(11):1448, 2014.

[44] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, April 2012.

[45] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[46] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 527–543. ACM, 2017.

[47] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 363–374. IEEE, 2015.

[48] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[49] Alexander Matveev, Yaron Meirovitch, Hayk Saribekyan, Wiktor Jakubiuk, Tim Kaler, Gergely Odor, David Budden, Aleksandar Zlateski, and Nir Shavit. A multicore path to connectomics-on-demand. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 267–281. ACM, 2017.

[50] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what {COST}? In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[51] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 29–42. ACM, 2007.

[52] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Supercomputing, ACM/IEEE 1999 Conference*, page 30, November 1999.

[53] Harald Prokop. *Cache-oblivious algorithms*. PhD thesis, Massachusetts Institute of Technology, 1999.

[54] Albert Reuther, Jeremy Kepner, Chansup Byun, Siddharth Samsi, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, et al. Interactive supercomputing on 40,000 cores for machine learning and data analysis. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2018.

[55] Rhoana. Fijibento. `https://github.com/Rhoana/FijiBento/tree/2d_mbeam`, 2015.

[56] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, Philadelphia, 2nd ed edition, 2003.

[57] Stephan Saalfeld, Albert Cardona, Volker Hartenstein, and Pavel Tomančák. As-rigid-as-possible mosaicking and serial section registration of large sstem datasets. *Bioinformatics*, 26(12):i57–i63, 2010.

[58] Stephan Saalfeld, Richard Fetter, Albert Cardona, and Pavel Tomancak. Elastic volume reconstruction from series of ultra-thin microscopy sections. *Nature methods*, 9(7):717–720, 2012.

[59] Harpreet S Sawhney and Rakesh Kumar. True multi-image alignment and its application to mosaicing and lens distortion correction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(3):235–243, 1999.

[60] David Sayce. 10 billions tweets, number of tweets per day. `http://www.dsayce.com/social-media/10-billions-tweets/`. Accessed: 2018-05-09.

[61] Tao B Schardl, William S Moses, and Charles E Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *ACM SIGPLAN Notices*, volume 52, pages 249–265. ACM, 2017.

[62] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on gpus. *Proceedings of the VLDB Endowment*, 11(1):107–120, 2017.

[63] Nir Shavit. A multicore path to connectomics-on-demand. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 211–211. ACM, 2016.

[64] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, 1991.

[65] Richard Szeliski. Image alignment and stitching: A tutorial. *Foundations and Trends® in Computer Graphics and Vision*, 2(1):1–104, 2006.

[66] Tolga Tasdizen, Pavel Koshevoy, Bradley C Grimm, James R Anderson, Bryan W Jones, Carl B Watt, Ross T Whitaker, and Robert E Marc. Automatic mosaicking and volume assembly for high-throughput serial-section transmission electron microscopy. *Journal of neuroscience methods*, 193(1):132–144, 2010.

[67] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[68] Charith Wickramaarachchi, Alok Kumbhare, Marc Frincu, Charalampos Chelmis, and Viktor K Prasanna. Real-time analytics for fast evolving social graphs. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 829–834. IEEE, 2015.

[69] Wenpu Xing and Ali Ghorbani. Weighted pagerank algorithm. In *Communication Networks and Services Research, 2004. Proceedings. Second Annual Conference on*, pages 305–314. IEEE, 2004.

[70] Xun Xu and Shahriar Negahdaripour. Vision-based motion sensing for underwater navigation and mosaicing of ocean floor images. In *OCEANS'97. MTS/IEEE Conference Proceedings*, volume 2, pages 1412–1417. IEEE, 1997.

[71] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[72] Barbara Zitova and Jan Flusser. Image registration methods: a survey. *Image and vision computing*, 21(11):977–1000, 2003.

[73] Imad Zoghlami, Olivier Faugeras, and Rachid Deriche. Using geometric corners to build a 2d mosaic from a set of images. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 420–425. IEEE, 1997.