Browser-Based Cluster Computing through WebGL-Accelerated MapReduce

Yuchi Tian Zack Verham
University of Virginia
{yt8mn,zdv8rb}@virginia.edu

1. Abstract

As big data problems continue to grow, computing clusters are also forced to scale in terms of size, power, and efficiency. In light of the inherent difficulty of creating and maintaining clusters at scale, alternative methods for performing distributed computing must be investigated. Internet-connected machines with JavaScript-enabled browsers are a potential source of large-scale computing power with no hardware storage requirements and minimal administrative costs. While similar browser-based distributed computing systems have been explored in the past, many of them have been hamstrung both by the single-threaded nature of JavaScript, and by the fact that network inefficiencies quickly become the bottleneck in large-scale computing jobs running on the Internet.

A (relatively) recent development in web technology is the full implementation of the WebGL API in most common browsers. The WebGL API provides JavaScript programs access to Graphics Processing Unit (GPU) resources on the client machine in order to improve the rendering of 2D and 3D graphics in the browser. We propose that a repurposing of this API which allows for generalized matrix computation, coupled with previous work on web-based MapReduce systems, may allow web-based distributed computing systems to overcome common bottlenecks by running massively parallelized computations on client GPUs.

We developed a browser-based MapReduce system which implements WebGL acceleration, and compared the performance of our system against Hadoop on a diverse set of benchmarks. Our initial results indicate that our system could either match or outperform Hadoop on jobs which are computation-intensive and not bottlenecked by network latency and/or the cost of initializing WebGL data structures. However, more results from running our system at-scale need to be collected before conclusive arguments can be drawn from our results.

2. Introduction

The global rate of data collection and associated storage requirements have increased dramatically over the past few years. An article written by McAfee and Brynjolfsson in 2012 [1] states that, at the time, 2.5 exabytes of data were being generated daily. This rate of data generation was doubling every forty months. Researchers in 2013 predicted that IP traffic alone would generate an annual 1000 exabytes of data by the end of 2016, and 2000 exabytes by 2019 [2]. Even if this rate of increase has not been maintained, the volume of data being generated today is incredible, and it poses tangible difficulties as various actors (industry, government/military, academia, etc.) attempt to make sense of it. The large-scale data problem has contributed to the development of new models for parallel and distributed computing. For example, the MapReduce paradigm proposed by Google [3] allows programmers without any experience in parallel and distributed systems to easily and rapidly utilize the resources of a large distributed cluster. Although this paradigm has obvious use cases within Google's information retrieval architecture, it has also seen widespread adoption in other fields. For example, machine learning and data mining scientists [3], and genome biologists [4] have used MapReduce to process overwhelming amounts of data in an efficient and comprehensible manner.

In 2013, Cushing et. al. originally proposed running distributed computation through the browser [5]. The authors of this paper recognized that the Internet has essentially created a very large cluster of machines, each of which is inherently equipped to run browser-based JavaScript. Their proposal was to crowdsource distributed computation through the Internet, and they showed that intensive browser-based computation is more feasible because of novel web technologies (such as Web Workers, Web Sockets, HTML5, WebGL, and WebCL). However, most of these features have not seen widespread adoption in-industry, and the adoption of these technologies in browsers has been inconsistent. Similarly, Langhans et. al. [6] propose a MapReduce framework which allows everyday users to connect to a MapReduce cluster that runs computations in the background of their browser. These background computations can be performed without negatively impacting the browsing experience of users.

In this paper, we build on previous work in recognizing that the Internet has effectively created a giant cluster of networked machines which remains mostly untapped for general computation. However, both the Internet [6] and the single-threaded nature of JavaScript [5] [7] impose bottlenecks which limit the efficacy of Internet-based distributed computing.

The proposal of this paper is as follows: At a high level, we hope to be able to generate massively distributed computation on an Internet-based cluster, removing the need to own and maintain commodity hardware in order to run large-scale distributed computing jobs. Below this, we propose a library which repurposes WebGL in order to run massively parallelized matrix computations in each browser "node." We propose that this additional acceleration can overcome the bottlenecks found in previous JavaScript-based MapReduce systems [5] [6] [7] in a robust and effective manner. Based on our proposal's explicit emphasis on computational efficiency, the fact that our system relies on the web for data transmission rather than direct machine-to-machine connections, and on the results from previous browser-based MapReduce implementations [6] [7], we hypothesize that our system will excel when it is computation-bound, but will not perform well when constrained by network latencies and data transmission.

3. Related Work

Work on distributed computing models has carried on for quite some time. Although MapReduce [3] is not the only distributed computing model available [8], it has seen widespread adoption and remains one of the more popular paradigms for understanding and writing distributed programs. In broad strokes, MapReduce defines two operations: map (which takes in a set of key/value pairs and generates a set of intermediate key/value pairs), and reduce (which consolidates intermediate key/value pairs with the same key into a single output).

The intent of this paradigm is to hide the complex components of parallel computing from the user, and to allow users without much experience in distributed computing to still make use of clustered parallelism [3].

Similarly, there are several examples of distributed systems designed to run in JavaScript-enabled web browsers [5] [6] [7]. However, previous implementations have been inherently limited by a lack of provided source code and strict adherence to the HTML5 Web Worker model (which remains CPU-bound) [9]. Our hope is that our system will be easily adoptable by others, while also performing competitively by maximizing parallel computation through the use of the GPU, instead of relying on HTML5 Web Workers.

Literature exists on the use of WebCL for GPU-based computation in a browser-based cluster [5], but WebCL has no native support in any browser (it is only available through specialized browser releases). Using WebCL will prevent commonplace devices from being included in Internet-based clusters for the foreseeable future. Instead, we propose the use of WebGL, a more widely-adopted and implemented technology which allows browsers to access the client GPU for 2D and 3D graphics rendering.

There has also been movement towards repurposing Graphics Processing Units (GPUs) to run massively parallelized computations on general datasets. Although GPUs are designed to run parallel operations on sets of pixels, they can be repurposed to run on general data, taking the brunt of computation off of the CPU and placing it on hardware which is specifically designed to run in parallel [10]. This idea has been extended to the investigation of running several GPUs in parallel, specifically within the context of MapReduce programs [11].

Although the idea of repurposing the GPU for generalized computation is not new [10], to the best of our knowledge, there has been very little investigation into repurposing the WebGL API for general computation through the browser (outside of difficult-to-find online tutorials [13]).

4. System Design and Implementation

Our system implements two concurrent levels of parallelism. At one level, we utilize the parallelism provided by a large number of browsers which act as nodes in MapReduce computations. Within each of these nodes, we attempt to generate further parallelism by moving work onto client GPUs though a JavaScript library which repurposes WebGL in order to run generalized matrix computations.

4.1. Browser-Based MapReduce System

The following section describes our implementation of high-level parallelism through a browser-based MapReduce framework.

4.1.1. Web-Based MapReduce Framework

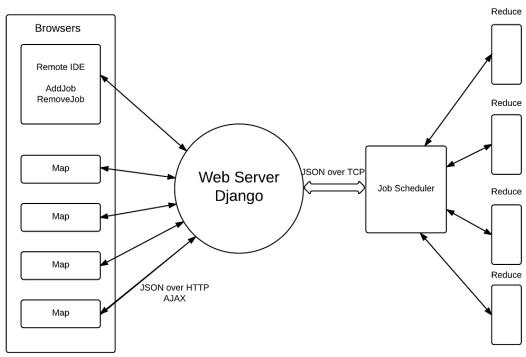
MapReduce is a programming model proposed by Google for large-scale data processing [3]. Hadoop MapReduce is an open-source implementation of Google's MapReduce programming model. Figure 1 provides an architectural overview of the browser-based MapReduce framework we designed and implemented. There are three major components which define our browser-based MapReduce framework: the browsers, the web server and the job scheduler.

Client browsers accept tasks (a map function and input data) from the web server, execute the tasks provided to them, and post the results back to the web server continuously through Asynchronous JavaScript and XML (AJAX).

The web server acts as the communication medium between browsers and the job scheduler by providing three services. First, it provides a remote development and job management platform to users. Developers can create browser-based MapReduce programs and add jobs to the job scheduler remotely through the remote development and job management page. Second, the web server pulls tasks from the job scheduler and sends them to the browsers. After a browser sends a get_task request to the web server, the web server will pull one task from the job scheduler and send the task to the browser. Finally, the web server receives completed results from the browsers and sends them back to the job scheduler.

The job scheduler is the most important and complicated component in our browser-based MapReduce framework. Our job scheduler design emphasizes multithreading. Every time the job scheduler receives a TCP connection request from the web server, the job scheduler will create an Agent to handle this TCP session. If the TCP connection from the web server is for adding a job, the Agent will create a new job and add it to the job queue. If the TCP connection is for getting a task, the Agent will choose a job from the job queue according to our scheduling algorithms and then identify the task of the job, modify relevant task queues, and update the web server. If the TCP connection is for returning a result, the Agent will execute the reduce procedure with this result and record the final outcome.

Figure 1: Browser-Based MapReduce Framework



4.1.2. Word Count Example

A description of our word count implementation follows. This program counts the number of occurrences of each word in a large text corpus (more details are provided in Section 5.1.1).

The map function is written in JavaScript. We chose JavaScript as the programming language for map functions because browsers can natively execute JavaScript code without modification. Importantly, it is also significantly more convenient to integrate WebGL into the MapReduce framework if the API for interacting with WebGL is written in JavaScript. Map functions take a String as input and return an Object which contains a set of key/value pairs.

The reduce function is written in Python. We choose Python as the programming language for reduce functions because Python provides better performance than JavaScript in asychronous call / multithreading programming and socket programming. The reduce function takes two Python dict type variables as input and returns a dict type variable as output. The first parameter of the reduce function is the final_result of the job context and the second

parameter is the result of a task returned by a browser. The result returned by the reduce function will be stored to the final_result of the job context again. Example code is provided below:

```
function __map_function(input)
{
    var split = input.split(" ");
    obj = {};
    for (var x = 0; x < split.length; x++){
        if(obj[split[x]] === undefined){
            obj[split[x]] = 1;
        }
        else{
            obj[split[x]]++;
        }
    }
    return obj;
}
```

```
def __reduce_function(final_result, task_output):
    for i in task_output:
        if i in final_result:
            final_result[i] += task_output[i]
        else:
            final_result[i] = task_output[i]
    return final_result
```

4.1.3. Implementation Details

In the following section, we describe some of our considerations while building our MapReduce system.

Fault Tolerance

Because results from browsers cannot be absolutely trusted [6], we create three task queues for each job to ensure the eventual execution of all tasks.

The first task queue stores the taskids of all the tasks which have not been assigned to any browser. When a task is assigned to a browser, its taskid will be removed from the first queue and appended to the second queue. Finally, when a task is finished (meaning that a browser has posted back the result of that task), its taskid will be removed from the second queue and added to the third queue. Therefore, the second queue has the taskids of all the tasks which have been assigned but not finished, and the third queue has the taskids of all the tasks which have finished.

If a browser sends request to get a task but does not execute or return the result, this task remains in the second queue and will eventually be assigned to another browser.

Asynchronous Return in Map Function

When integrating WebGL acceleration into map functions, an asynchronous return function should be called instead of a common return statement. When the map function returns in a WebGL-accelerated program, the GPU may still be performing computations. In order to make WebGL programs return correct results, the WebGL program should call an asynchronous return function explicitly as soon as the GPU completes its computation.

Reduce Function executed in Server

We decided to execute the reduce function on the server instead of browsers for three reasons. First, letting the reduce function be executed in browsers would result in more complicated system design and imply the need for additional task synchronization. Second, letting the reduce function be executed in browsers will lead to larger overhead caused by additional network transmission of reduce results and/or frequent task synchronization. Finally, letting the reduce function be executed server-side enables the map function and reduce function to be executed simultaneously; otherwise, the reduce procedure has to be executed after all mappers complete.

JSON-based Communication

AJAX and JavaScript Object Notation (JSON) are used for communication between browsers and the web server. JSON over TCP is used for communication between the web server and the job scheduler.

4.2. WebGL Client Computations

Within each client machine, we hope to be able to both speed up computation and handle larger workloads by offloading work onto client GPUs through the WebGL API. This ultimately requires repurposing the WebGL API, which is designed for graphics computations, so that it can operate on generalized matrices. This repurposing is fundamentally dependent on a reinterpretation of how the WebGL API can operate, and what kind of data it can accept and return.

4.2.1. WebGL Basics

Before describing how we repurposed WebGL to operate on generic matrices, it may be helpful to provide a basic overview of how WebGL operates. Fundamentally, WebGL is meant to simplify browser-based access to the client GPU in order to run intensive graphics computations within web applications. The API is modeled heavily off of OpenGL [12], and maintains many of the same naming conventions and design paradigms. For our purposes, there are three primary WebGL API elements which will ultimately allow us to run matrix computations on client GPUs:

1. An HTML canvas which encapsulates every WebGL computation
2. Some geometric object(s), the coordinates of which are defined by fragment shaders (and will ultimately be mapped onto graphics textures and/or frame buffers)
3. Vertex shaders, which define the RGBA color values for geometries defined by fragment shaders

Shaders themselves are written in a C-like language called the OpenGL Shading Language (GLSL). At runtime, the WebGL API accepts GLSL shader programs as strings, compiles them, and runs them on the client GPU in order to generate graphics elements to display on the HTML canvas.

4.2.2. Repurposing WebGL

We perform three primary reinterpretations and conversions on the WebGL API in order to allow it to perform generalized matrix operations:

1. The HTML canvas is created in JavaScript, but never displayed on the screen
2. Our fragment shaders define geometries which exist in a one-to-one correspondence with individual pixel values
3. Our vertex shaders operate on pixel channels as if they were matrix elements.

In this way, textures become matrices and vertex shaders define available matrix operations. Other WebGL API elements, such as frame buffer objects, allow us to store intermediate computations without losing the current state of a given computation, but the core

of the library conversion is based on redefining these three essential components.

4.2.3. WebGL Library Architecture

Our repurposed WebGL library is comprised of five fundamental elements: the WebGLUtility class, the SourceDictionary class, the MatrixUtility class, the MatrixOp class, and the Matrix class. An overview of the library is provided below. For more details, the codebase can be viewed on GitHub.

1. **SourceDictionary:** The SourceDictionary class is used to make requests to the server for shader files. Upon receiving a file, it is parsed into a string and stored within a large JSON object, allowing other utility classes to easily access shader programs for compilation. The rest of the WebGL computation will not begin until the SourceDictionary has downloaded the required shader files. In our implementation, we have provided pre-written basic shader operations, such as matrix addition, subtraction, and multiplication.
2. **WebGLUtility:** This is the class which operates closest to underlying WebGL operations, and it provides a simplified interface to the higher-level classes.
3. **Matrix:** This class provides intermediary methods which connect the WebGLUtility and MatrixUtility classes, allowing MatrixUtility objects to create textures which can operate as matrices.
4. **MatrixOp:** This class creates and contains the shader programs which can be used to run computations on initialized WebGL matrices.
5. **MatrixUtility:** This class provides helper methods which simplify matrix initialization by making calls to the Matrix class API. It also provides helpful toString methods for aid in debugging and visualization.

A significant number of design and implementation decisions made in this library came from an extremely helpful tutorial [13]. The source code in the WebGLUtility, MatrixOp, and Matrix classes in particular were heavily informed by this tutorial.

Overall, the underlying force guiding most design decisions for this library came out of a desire to hide as much of the WebGL API from users as possible. A graphical depiction of the relationships between each of the above classes and how they relate to the larger MapReduce computation can be seen in Figure 2. Classes closer to the bottom of the stack in Figure 2 make most of the direct WebGL API calls, while users will interact with classes further up the heirarchy.

Figure 2: Relationships Between WebGL Library Classes

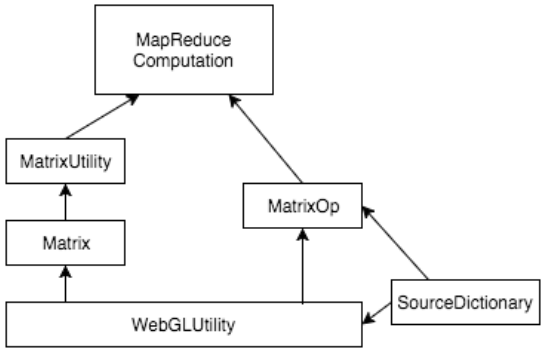


Table 1: Summary of (Expected) Benchmark Characteristics

Benchmark	Network Usage	Computation	WebGL Acceleration
Word Count	High	Low	No
Word Mean	High	Low	No
Pi Estimation	Low	High	Yes
All Pairs Shortest Paths	Low	High	Yes

5. Evaluation

In the following section we describe the process by which we compared our implementation against Hadoop.

5.1. Benchmark Description

In order to evaluate our system, we developed a suite of benchmarks which we implemented both for our system and within the Hadoop ecosystem. Hadoop is an open-source implementation of MapReduce as it was proposed in the original Google paper [3], and it has seen widespread use throughout industry as a tool for running large-scale distributed computation.

Two primary goals guided our thought process in designing and implementing our benchmark suite. First and foremost, we wanted our benchmarks to cover a diverse subset of the potential workloads which could be placed on a distributed computing system. More specifically, because we have proposed that our system will perform better on workloads which mitigate network communications and emphasize computation-per-node, we wanted to capture jobs with various levels of network communication and computational workloads. Our initial expected characteristics for our benchmark suite as collected from our initial design process are summarized in Table 1. Note that, ultimately, we were surprised by some of our results and the characteristics of our benchmarks were not what we expected. This will be described in detail later in the paper. Also note that some benchmarks (WordCount and WordMean) had no direct-mapping into a WebGL-accelerated form.

5.1.1. Word Count

The word count benchmark accepts some corpus of input text files and returns an aggregated word count for each unique word found in the corpus. This is one of the fundamental MapReduce programs used to introduce new Hadoop users to the MapReduce paradigm.

5.1.2. Word Mean

Similar to the word count benchmark, the word mean benchmark will accept some corpus of textual information and return the mean length of each unique word found in the corpus.

5.1.3. Pi Estimation

Our pi estimation benchmark implements a Monte Carlo simulation which estimates the value of pi. Essentially, the algorithm generates a large set of random two-dimensional coordinates which are situated within the unit square. The ratio of random coordinates which fall within a circle (of radius 0.5) circumscribed within the unit square compared to those which fall outside of the circle provides us with a value which can be used to estimate pi.

5.1.4. All Pairs Shortest Paths

The all pairs shortest paths benchmark accepts some number of adjacency matrices as input, and attempts to find the adjacency matrices which represent the distance of the shortest paths between all node pairs in the represented graph. The applicability of such a program is perhaps less intuitive than the previous examples;

Table 2: Summary of Textual Dataset

Dataset Name	Line Count	Word Count	1000-Line Splits
Pride and Prejudice	13426	124588	14
Ulysses	33057	267978	35

however, we propose that one could imagine some large graph which is easily broken up into independent (or nearly independent) subgraphs. For example, one could imagine a structural model of a city which can cleanly be broken up into city blocks. In this case, being able to compute all pairs shortest paths for each block or district concurrently is equivalent to computing the all pairs shortest paths of the entire city graph, which may be quite large.

The Hadoop implementation makes use of the Floyd-Warshall algorithm [14] for computing all pairs shortest paths of each subgraph. However, because our system is designed for matrix computation, we make use of Seidel's algorithm [15] in our implementation of all pairs shortest paths. Seidel's algorithm gives the same result as Floyd-Warshall, but can be distilled into a series of recursive matrix computations. We propose that computing all pairs shortest paths in this manner will better illustrate the performance benefit our system provides.

5.2. Datasets

Each of our benchmarks requires large input datasets in order to run realistic simulations. The parameters for these benchmarks are provided below.

5.2.1. Textual

Our textual datasets (used in word count and word mean) were taken from Project Gutenberg [16]. Project Gutenberg is an online resource which provides public access to works of literature that are out of copyright. Each work can be downloaded as a large text file.

For our text data, we collected text from two books: Jane Austen's "Pride and Prejudice" and James Joyce's "Ulysses". Each of these books was also split into 1000-line sub-files. Metadata information for each dataset is provided in Table 2.

5.2.2. Adjacency Matrices

The all pairs shortest paths benchmark requires a large sequence of adjacency matrices. In order to avoid graphs which contained cycles (for which Seidel's algorithm falls into an infinite recursion), we handcrafted a set of five acyclic graphs and randomly distributed copies of their adjacency matrices within a large text file. This text file contains 100 adjacency matrices, each of which represents an acyclic graph consisting of ten nodes.

6. Results and Analyses

We ran each benchmark in two environments: our implementation of a browser-based MapReduce system (running in a single browser), and a local installation of Hadoop (running on a single node). Ten trials were run for each benchmark in both environments. Future work would involve running the same set of benchmarks on a larger cluster; unfortunately, during the course of the semester in which this project was undertaken, the department-owned Hadoop cluster was out of commission. This prevented us from running the large-scale tests we hoped to perform. Figures which clearly illustrate our results can be found in the appendix at the end of this paper.

6.1. Word Count

As expected, our system performed poorly on the word count benchmark when to compared to Hadoop. We attribute this to the fact that the computations performed in word count are trivial: the majority of the running time is spent in network latencies which our system has not been designed to optimize against.

6.2. Word Mean

Our results for word mean were better than we had initially expected. We attribute our improvements over Hadoop to the fact that both implementations minimize the number of files which need to be passed over the network for this benchmark (by sending the entire text file rather than the 1000-line splits), allowing our system to emphasize computation over network latency. Note that this did not align with our expected benchmark characteristics. The amount of data being passed from the mappers to the reducer across the network is also trivial for this application (two integers).

6.3. Pi Estimation

As expected, our system outperformed Hadoop in estimating pi through a Monte Carlo method. We attribute this to the fact that we were able to offload an overwhelming majority of this computation into WebGL code, which ultimately simplified down to a few matrix operations between two very large matrices which represented random coordinates. Every coordinate computation was pushed through WebGL simultaneously, while Hadoop was forced to rely on a set of operations which are inherently more linear in nature.

6.4. All Pairs Shortest Paths

Our results for this benchmark were actually worse than we had anticipated. We discovered that, in the browser, running a single all pairs shortest paths computation which only requires a single recursive call runs anywhere in the ballpark of 140-160 milliseconds. This equates to tens of seconds to complete computation when running on a larger test set of 100 graphs. Hadoop ran the same computation in 1.563 seconds, on average. The problem, we discovered, is that on a single recursive call, approximately 30-40 milliseconds were devoted to the initialization of matrices necessary to run the computation. This initialization occurs on every recursive call, and it equates to approximately 40 percent of the computation time for a given run of all-pairs shortest paths in our system. These numbers are (very) approximate, but they give a sense of the inefficiency inherent in WebGL initialization.

These results are actually quite insightful when paired with the results from the pi estimation benchmark. It appears that our system's performance on matrix computations is perhaps more nuanced than we had originally anticipated. Our system fares poorly when running algorithms which require many independent matrices to be initialized before running computations. This is especially true if the matrices are not packed full with data. In contrast, pi estimation requires a large number of initial matrix initializations, but each of those matrices are packed full of data, and are then pushed through the computation pipeline without the algorithmic backtracking implicit in the recursive nature of Seidel's algorithm. This seems to have hamstrung the speed of our system in computing all pairs shortest paths, and is helpful when considering what *kind* of matrix computations perform well when coupled with our system.

7. Conclusions

Most importantly, our initial results indicate that our system can compete with more conventional MapReduce systems for certain families of computational problems. This result, coupled with the fact that our system runs without administrative ownership of cluster hardware, indicates that our system may provide novel benefit to the distributed computing community. More concretely, our results show that our system performs well on workloads which are:

1. Computation-bound, rather than network-bound
2. Able to be mapped into matrix computations
3. Minimize matrix initialization time as much as possible

Of course, our benchmarks need to be run at scale, with many additional nodes, before any decisive conclusions can be drawn regarding the efficacy of our system. After all, distribution of work over a large number of nodes is the entire point of running large-scale computing on a cluster. The difficulties we faced in attempting to access hardware to run our conventional Hadoop benchmarks were very unfortunate. However, we propose that our results validate the fact that further evaluations should be run, since our initial evaluation lines up well with our expectations regarding overall system performance.

8. Shortcomings and Lessons Learned

In the following section, we analyze some shortcomings (both expected and unexpected) with our work, and we discuss how these shortcomings inform both our understanding of our current system and our proposals for future work on the project.

8.1. Browser-Based MapReduce

In our browser-based MapReduce system, tasks (defined as map functions and input data) and returned results will frequently be exchanged between browsers and the web server. The data transmission between browsers and the web server may become a major factor which impacts the performance of our system. Programs which minimize I/O operations will probably see improved performance.

In our current implementation of our MapReduce system, all of our intermediate results are stored in memory. This provides efficiency improvements when compared with the traditional Hadoop MapReduce implementation, which stores all intermediate results in files. However, as the number of jobs increases, this implementation might use up all of the memory in the web server and job scheduler. Therefore, a future improvement would dynamically store intermediate results into files when necessary.

8.2. WebGL

One significant problem we faced as we implemented our WebGL library is the unfortunate fact that much of our shader code needed to be specialized for specific applications. We had originally believed that we would simply need to implement basic matrix operations in GLSL, which could then be chained together for repeated use in many other applications. This ended up being infeasible for all but the simplest applications (for example, test programs which only performed simple matrix additions and subtractions). The impact of this problem became obvious as we implemented Seidel's algorithm; the shader code for this particular example has been included in the appendix (Listing 1). Although this code will not be explained in detail, the point of including it is that, simply, in no way will this snippet of GLSL ever be usable in any other application aside from our implementation of Seidel's Algorithm. A hope of this project was to develop a flexible WebGL library for arbitrary matrix computation, but actually achieving this goal properly seems to be beyond the scope of a single semester.

9. Future Work

The most significant implementation-oriented region for future work would involve maximizing the parallelism generated thorough WebGL. We believe that we are not making full use of the parallelism available to us through the WebGL interface, because we are not packing each individual pixel channel with data. In our prototype implementation, we only make use of the alpha channel when running matrix computations; the red, green, and blue channels remain dormant. It should be possible to improve running times even further by placing data in the other three channels as well; we simply did not have enough time to design, test, and implement algorithms for dynamically allocating data at a sub-pixel level. Although the following claim is purely speculative and requires more vigorous validation before it can be accepted as fact, it seems like the WebGL interface makes the implicit assumption that the smallest granularity at which data will be passed to into the shaders stops at the pixel level. Writing routines to pack sub-pixel channels would likely require a significant amount of work.

Similarly, there are other optimizations that could be applied to our WebGL library which we were not able to implement within the timeframe of a single semester. The most immediate is the fact that shader code is not inlined within our JavaScript files. Rather, it is collected from the server through independent XMLHttpRequests after the initial page has loaded. This was done for debugging purposes while writing the library (debugging inlined shader code was extremely difficult). However, we believe that inlining the code could reduce network latency and introduce additional improvements in running time. As an example of the benefit that can be potentially provided through optimizing our code, we performed some trivial optimizations on our all pairs shortest paths application, and re-ran the optimized version of this benchmark. The results of this run can be seen in the second graph of Figure 6 (Appendix). While we still cannot compete with Hadoop on this benchmark, our average running time decreased by approximately 37.8 percent. This indicates that there are probably other significant improvements which we can create purely through optimizing our codebase.

The single most important area for future work, across the board, would be to collect more benchmark data. Within the context of this course project, we were limited by the fact that we only had access to single-node environments. We believe that our results would both be more robust and perhaps more nuanced if we were able to run our benchmarks on a larger portfolio of machine profiles. This would include running our benchmarks on larger Hadoop clusters, and running the benchmarks for our system on a diverse set of client machines with different GPUs.

Finally, we would love to be able to extend our work into other distributed computing paradigms (for example, Apache Spark [8]).

10. Appendix

Listing 1: Portion of shader code used in Seidel's Algorithm. The specificity of shader code required makes it very difficult to generalize use for other applications.

```
precision highp float;

uniform sampler2D X;
uniform sampler2D T;
uniform sampler2D ADJ;
uniform float dim;

varying vec2 vTextureCoord;

void main() {
    // Variable initialization
    float  i, j,
            delta,
            x_val,
            t_val,
            adj_val,
            degree;

    // Capture texture/matrix coord mapping
    i = vTextureCoord.s;
    j = vTextureCoord.t;

    // Initialize delta (allowed error in
    // floating point calculations)
    delta = 0.00001;

    // Cast matrix dimension to int
    int dim_i = int(dim);

    // Capture matrix values at (i, j)
    x_val = texture2D(X, vec2(i, j)).a;
    t_val = texture2D(T, vec2(i, j)).a;
    adj_val = texture2D(ADJ, vec2(i, j)).a;

    float k_counter = 0.0;

    degree = 0.0;

    // Workaround required to use
    // non-constant loop guards
    for(int k = 0; k < 2048; k++) {
      if(k < dim_i) {
        float conn =
          texture2D(ADJ,
          vec2(k_counter/dim, j)).a;
        if(conn > 0.0) {
          degree = degree + 1.0;
        }
        k_counter = k_counter + 1.0;
      }
    }

    // Specify passing of r/g/b chaennels
    // to avoid uninitialized texture
    // values
    gl_FragColor.r = texture2D(X,
            vec2(i, j)).r;
    gl_FragColor.g = texture2D(X,
            vec2(i, j)).g;
    gl_FragColor.b = texture2D(X,
            vec2(i, j)).b;

    // Return conditional-defined result
    float eq = abs(x_val  -
        (t_val * degree));
    if (x_val >= t_val * degree) {
        gl_FragColor.a = 2.0 * t_val;
    }else {
        gl_FragColor.a = 2.0 * t_val
                - 1.0;
    }
}
```

Figure 3: Word Count Results

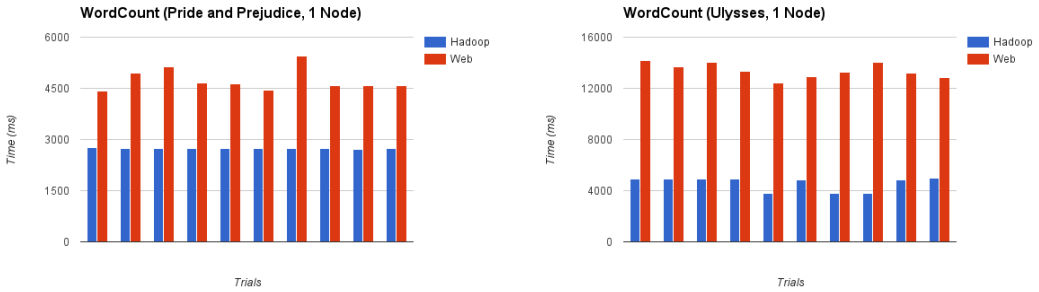


Figure 4: Pi Estimation Results

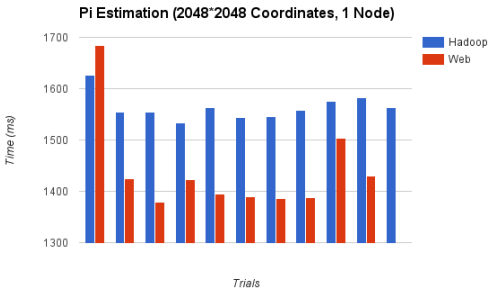


Figure 5: Word Mean Results

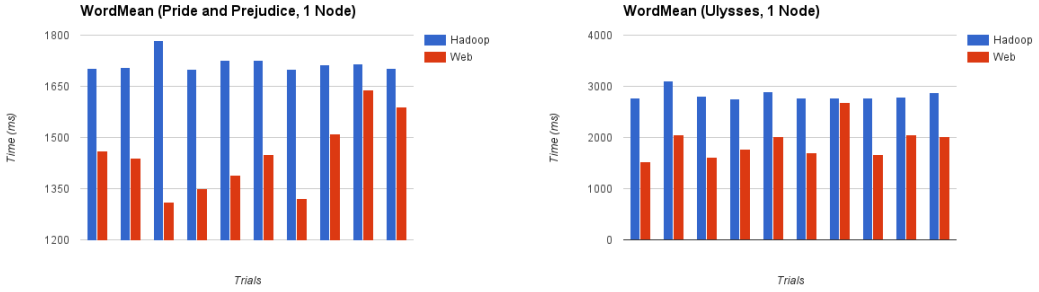
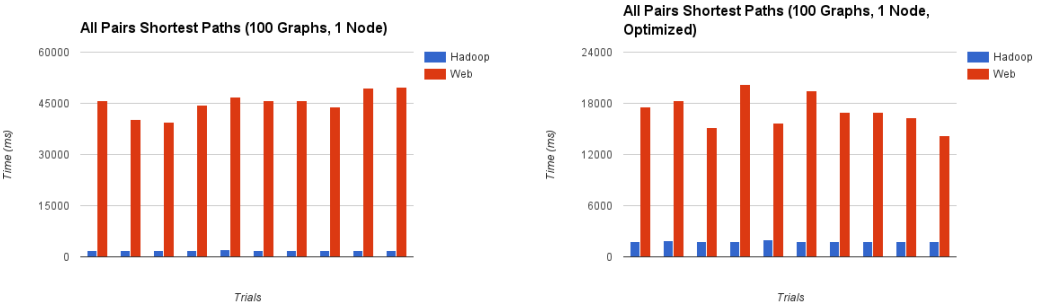


Figure 6: All Pairs Shortest Paths Results



References

[1] Andrew McAfee, Erik Brynjolfsson, Thomas H Davenport, DJ Patil, and Dominic Barton. Big data. *The management revolution. Harvard Bus Rev*, 90(10):61–67, 2012.

[2] Cisco Visual Networking Index. The zettabyte era–trends and analysis. *Cisco white paper*, 2013.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.

[5] Reginald Cushing, Ganeshwara Herawan Hananda Putra, Spiros Koulouzis, Adam Belloum, Marian Bubak, and Cees De Laat. Distributed computing on an ensemble of browsers. *Internet Computing, IEEE*, 17(5):54–61, 2013.

[6] Philipp Langhans, Christoph Wieser, and François Bry. Crowdsourcing mapreduce: Jsmapreduce. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 253–256. International World Wide Web Conferences Steering Committee, 2013.

[7] Sandy Ryza and Tom Wall. Mrjs: A javascript mapreduce framework for web browsers. *URL http://www. cs. brown. edu/courses/csci2950-u/f11/papers/mrjs. pdf*, 2010.

[8] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[9] Hannu Järvinen. Html5 web workers. In *T-111.5502 Seminar on Media Technology BP, Final Report*, page 27, 2011.

[10] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. Gpgpu processing in cuda architecture. *arXiv preprint arXiv:1202.4347*, 2012.

[11] Yi Chen, Zhi Qiao, Hai Jiang, Kuan-Ching Li, and Won Woo Ro. Mgmr: Multi-gpu based mapreduce. In *Grid and Pervasive Computing*, pages 433–442. Springer, 2013.

[12] Dave Shreiner, Bill The Khronos OpenGL ARB Working Group, et al. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education, 2009.

[13] Unleash your inner supercomputer: Your guide to gpgpu with webgl. http://www.vizitsolutions.com/portfolio/webgl/gpgpu/.

[14] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[15] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of computer and system sciences*, 51(3):400–403, 1995.

[16] Michael Hart. *Project gutenberg*. Project Gutenberg, 1971.