MRJS: A JavaScript MapReduce Framework for Web Browsers

Sandy Ryza Tom Wall

Final Project - CSCI2950-u - Fall 2010

Abstract

This paper describes MRJS, a JavaScript based MapReduce framework. MRJS runs on the open internet over HTTP rather than in a data center. Instead of using clusters of machines for processing power, it relies on volunteer clients in the form of JavaScript engines in web browsers. Users can specify, execute, and fetch data for arbitrary MapReduce jobs with HTTP requests. Our centralized job server acts as a coordinator. We have implemented MRJS and results show that it performs well for certain types of MapReduce computations.

1 Introduction

As more and more people have become connected to the internet at faster and faster speeds, the practice of offloading computation to volunteer clients over the internet has gained appeal. Numerous projects have proven the viability of such volunteer computing. SETI@Home [4], Folding@Home [5] and similar projects distribute small pieces of massive computational problems over the internet, providing vast computational resources to the scientific community at almost no cost. While some of these projects have had great results, there are some shortcomings. Traditionally, volunteer computing projects have relied on a piece of client software which must be downloaded and installed. We believe this initial cost of entry, while low, discourages many potential users. Another issue is that the problems that these systems solve are not always generalizable. While they may be similar in spirit, each project requires its own specialized software. Creating a job to outsource requires consideration of the complicated details of distributed computation. Partitioning a problem into parallelizable chunks, tolerating client slowness and failure, verifying results, and a host of other challenges emerging from the system's distributed nature are issues which ideally, a programmer for one of these projects should not have to worry about.

MRJS avoids the barriers of entry common to other volunteer computation projects by using a simple programming model on a computation engine that most volunteers already have installed: a web browser. To contribute cycles to a computation, a person needs only to open up a tab in their browser and direct it to the MRJS web page. From there, their browsers will download chunks of work, perform computation in JavaScript and submit the results back to a server. We use MapReduce [9], a framework originally designed by Google to distribute computation across their clusters, as a programming interface. MapReduce simplifies distributed programming by abstracting away the details of partitioning, scheduling and fault tolerance. A developer specifies their program in terms of two functions, a map function and a reduce function. MRJS partitions and schedules the work, sending the map or reduce code and some input data to volunteer client browsers. MRJS processes the outputs of these outsourced computations to produce a final result.

This paper explores the challenges of building such a system and evaluates its effectiveness. We reason about the general viability of the system, identify potential bottlenecks, and classify ideal jobs. Section 2 presents an overview of the technologies and frameworks that MRJS relies upon. Section 3 describes our system's architecture and describes the design choices we made. Section 4 describes our testing methodology, covering our experimental setup, a description of the test jobs, and a description of the timing data collected during the tests. Section 5 presents the results of our experiments and from these, identifies the system's strengths and weaknesses. Section 6 draws some more general conclusions about our work, including potential improvements and ideas for future work. Section 7 goes into more depth on some related work and the similarities and differences of our approach. Finally we conclude with a brief summary of our work in section 8.

2 Background

2.1 MapReduce

The MapReduce programming model provides a simple API for concurrent data processing on a cluster of machines. Input data is organized as a set of `<key, value>` pairs. For each pair, a user-specified `map()` function is applied. The `map()` function processes the `<key, value>` pair and produces some number of intermediate `<key, value>` pairs. Once the `map()` function has been executed on each of the input keys, the collective output of all the `map()` calls (i.e. the set of intermediate `<key, value>` pairs) is sorted. For each unique intermediate key, a list of values is built. The user-specified `reduce()` function processes each intermediate `<key, value list>` pair to produce a final result.

There are a number of optional, job-specific configuration and optimization techniques that a user might take advantage of. In the original MapReduce system, users can specify a data partitioning function and an intermediate key sorting function. Also, a combiner function can be specified to minimize the number of `map()` outputs. The combiner function, if specified, acts similarly to the `reduce()` function. The combiner does a partial `reduce()` of the intermediate `<key, value>` pairs generated by a single mapper. Figure 1b details the word count MapReduce job and its use of a combiner function.

MapReduce is popular because of its simplicity. Users need not worry about the complex coordination details; in most cases they only need to specify map and reduce functions. It is up to the runtime system to efficiently organize and coordinate both the the data and the machines to maximize concurrency. While most well known implementations such as Apache Hadoop [1] rely on a cluster of commodity machines running a distributed file system, we take a different approach.

```
var results;

//main thread calls postMessage()
//on this worker to start the computation
onmessage = function(event) {
   var job = JSON.parse(event.data);
   for(var i = job.startIdx; i < job.endIdx; i+= job.recordsPerRequest) {
      //AJAX get request to the data proxy
      //to retrieve a set of input data records
      var inputData = fetchInput(job.inputURL, i,
                      min(job.endIndex, i + job.rangeRequestSize));
      for(var record in JSON.parse(inputData))
         map(record.key, record.value);
   }

   for(var key in results) {
      var values = results[key];
      results[key] = null;
      combine(key, values);
   }

   //AJAX POST the results to the job server
   sendResults();
}
```

(a) WebWorker thread logic

```
//key: a file name
//value: a chunk of text from the file
function map(key, value) {
   var words = value.split(' ');
   for(var word in words)
        emit(word, 1);
}

//key: a word
//values: a list of 1's
function combine(key, values) {
   var total = 0;
   for(var i in values)
       total += i;

   emit(key, value);
}

function emit(key, value) {
   if(results[word] == null)
      results[word] = new Array();

   results[word].push(1);
}
```

(b) Map and Combiner function

Figure 1: A portion of the JavaScript code for the map task of the word count job.

2.2 JavaScript

As the web has matured, there has been a shift from serving static HTML pages to dynamic, fully featured applications. At the core of this evolution has been the JavaScript programming language. While originally used for manipulating pages in response to client interaction, it has grown to support far more sophisticated uses. A few key advancements have made JavaScript a feasible language for a volunteer computation system.

The AJAX (Asynchronous JavaScript and XML) [10] programming model was a tremendous paradigm shift for web application development. In an AJAX application, the browser can make background HTTP requests in JavaScript without having to request a whole new page. This process creates a more interactive and responsive experience for users and offers greater flexibility for developers. Developers use the `XmlHttpRequest` JavaScript object [16] to send and receive content from web servers in the background. They can then update a page appropriately with data from the response. While originally designed to transmit XML data, it is often used to transmit other serialized data formats, such as the more compact and JavaScript friendly JSON [7].

Until very recently, all of a page's JavaScript code ran in a single thread. Event handling code executed in response to user interaction is executed sequentially. Developers typically avoid long running code blocks (such as a map or

reduce function) because the engine cannot respond to any user interaction until the expensive task is complete. These long running, complex tasks create the appearance of a sluggish or unresponsive web page. This problem is amplified when browsers share a single JavaScript engine amongst all windows or tabs open in a browsing session, because a heavy JavaScript load on one page could affect the responsiveness on another completely unrelated page.

The HTML5 specification introduces a JavaScript threading construct known as WebWorkers [13]. With WebWorkers, computationally expensive code can be run in its own thread, allowing the main user interface thread to continue to respond to events without interruption. To remain simple, there is little support for synchronization. Worker threads cannot modify the HTML DOM and can only communicate through a message passing API.

The main JavaScript thread can spawn a new thread with the `Worker` object. The `Worker` object takes as a parameter a path to a JavaScript file to execute. Threads communicate using the `postMessage()` function. A thread can register itself as a listener of the `onmessage` event for a `Worker` object. To communicate, a thread calls `postMessage()`, passing any string as an argument. Listeners of the `onmessage` event receive notification of the `postMessage()` and can execute an event handling function in response to the event. Figure 1a shows an example of this communication. The main thread (not shown) creates a `Worker` object and calls its `postMessage()` function. In response, the `onmessage` event handler in the figure is executed, and the map computation begins.

Our JavaScript MapReduce implementation uses both AJAX and WebWorkers to do heavy client side processing while not diminishing the user's overall web browsing experience. Map and reduce computation occurs in a background thread to minimize impact on the user interface. AJAX requests for JSON formatted data are made as it is needed to minimize the size of the initial page request. The next section goes into detail on how the system utilizes these concepts to realize our goal.

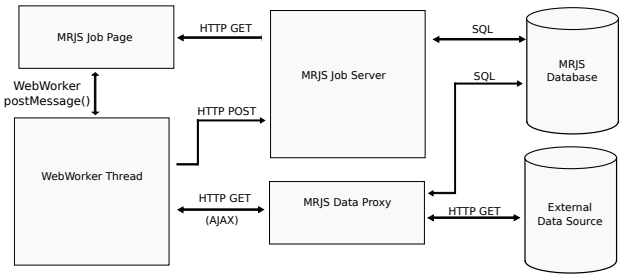


Figure 2: MRJS Architecture

3 MRJS Design

3.1 Design Goals

The reliance on on volunteer clients over the internet forces us to design carefully. We must expect and deal with slow and broken connections. Clients may navigate away from our web page; if this happens, it is essential that we do not lost track of incomplete work. On the other hand, we must also expect results to arrive at the server even after they are no longer needed. Because our system relies on voluntary participation, it must be as noninvasive as possible on the client's browsing experience. It should avoid slowing the browser down and should never crash or adversely affect other windows or tabs. Third, we did not want our systems computational model to be general enough to solve many types of problems. Our system must be able to run any MapReduce job, and the creator of a job should have control over parameters that might affect performance. Finally, efficiency is important, because in practice our system will be most useful when it is faster to formulate a task as a MRJS job than it is to perform the task in some other way.

3.2 Data Model

The input data for a job is partitioned into a set of uniformly sized chunks, the number of which is be specified by the job's creator. We call these *WorkChunks*. A WorkChunk constitutes the basic unit of work that can be given to a client. It includes the appropriate JavaScript code to run, start and ending record indices, a data input URL and a request size parameter.

When a WorkChunk is assigned to a client for execution, a *ClientTask* is created. The ClientTask stores the results of a client's execution of a WorkChunk. It may be the case that multiple clients execute the same WorkChunk, in which case multiple ClientTasks are created. By having multiple copies of the results of a computation, it becomes easy to detect faulty results; a simple majority vote decides which ClientTask represents the correct output.

We originally considered a design that allowed for variable-sized ClientTasks and no notion of WorkChunks. This would provide flexibility in assigning arbitrary regions of the input data to clients. However, this would have required searching through the set of created tasks to find the spaces that had not yet been covered for each request for work, and also would have slowed down and complicated the process of choosing the majority result for a WorkChunk if multiple clients had been assigned to work on it.

3.3 Job Server

At the center of MRJS is the job server. The job server schedules WorkChunks to be executed and organizes the results of client computation. For each job, the job server keeps track of which WorkChunks are incomplete, and assigns ClientTasks for those WorkChunks in response to client requests. In our implementation, the job server runs on a single machine. This simplified our implementation, but it is also a single point of failure. A more sophisticated system might rely on a cluster of machines to collectively perform the duties of the job server.

The job server keeps track of the number ClientTasks created for a particular WorkChunk. When a request comes in for work, the job server simply chooses the WorkChunk (across all jobs) with the fewest ClientTasks and assigns it to the client. As a result of this scheduling policy, when there are more clients asking for work than there are WorkChunks to work on, multiple clients will be assigned to the same task. If a client fails to return results, eventually the same work will be sent to another client. When multiple clients return results for the same WorkChunk, the majority result will be used as a final value. This mitigates the effect slow clients have on the overall performance of the execution; if each WorkChunk is assigned to one client, the computation can only be as fast as the slowest client. Using multiple clients per WorkChunk reduces the probability of a slow client hindering the overall execution. It is also useful to ensure the integrity of client results. If a faulty or malicious client returns an invalid result, it can be detected. The downside to having multiple ClientTasks per WorkChunk is that there is duplicated effort which may have been better spent on different computation. The number of ClientTasks to assign for each WorkChunk is a parameter that can be specified by the job creator and defaults to 1.

With the expectation in mind that jobs may be operating on enormous datasets, it might be too costly to transfer the data set to our job server. Instead, our design allows input data to a job to be stored at an external host. The host must make this data available via a simple HTTP GET interface. The URL for this interface takes a start and end parameter which act as indices into the data set. A response to a properly formed request to this URL returns a JSON string containing the subset of input records specified by the start and end parameters.

A complication we discovered early on was that we were unable to send AJAX requests to a different domain because of JavaScript security limitations imposed by the same-origin policy [16]. The `XmlHttpRequest` object returns a security error when a request is attempted to any domain other than the domain from which the JavaScript originated. Thus, we were forced to modify our design to include a proxy on the job server that forwards these fetch requests to the server hosting the data. The JavaScript code make a request to the MRJS data proxy for data. Upon receiving the request, the data proxy issues the HTTP get request to the data server to fetch the data on the behalf of the JavaScript code. This can add a significant amount of latency to the data fetching process. We discuss the implications of the data proxy in section 5 and potential alternatives to the data proxy in section 6.

3.4 Job Execution

Jobs are submitted to the job server via an HTML form. A job's creator specifies the URL to retrieve the input data from, the range of input indices, the number records to process per WorkChunk and the number of records to request from the data proxy at a time. Upon submission of this form the job server verifies the job and partitions the input range into WorkChunks for the map phase.

Figure 2 describes the various components involved in the execution of a Job and their communication patterns. During the map phase, the server sends map WorkChunks in response to client HTTP GET requests for work. The intermediate `<key, value>` pairs created by a client are stored in the job server's database. When results have been received for all map chunks, the map phase for that job is marked as completed, and the server then performs

the shuffling work that must be completed before the reduce phase begins. In this shuffling phase, described in more detail below, intermediate outputs from the map phase are partitioned into reduce WorkChunks.

The reduce phase proceeds in the same way as the map phase, with the job server sending out reduce ClientTasks to clients and storing the results. When results have been received for all reduce chunks, the reduce phase is marked as completed. The server gathers the output data from the reduce phase and writes it to a file, which the job creator can then download. An extension we have considered but haven't implemented would be to issue an HTTP POST of the results to an external server.

3.5 Client Code

To execute a job, the main JavaScript thread starts a WebWorker. The WebWorker runs a JavaScript file containing the task code, retrieved by a separate HTTP request. Figure 1 describes the Worker's execution. The main thread defines a JSON object, initialized by the job server, encapsulating some WorkChunk specific parameters. These parameters include the range of indices to process and number of indices to request at a time from the data proxy. The main thread passes this object to the WebWorker thread via `postMessage()` to start the computation. By keeping the WorkChunk specific data out of the WebWorker file, it can be cached and reused by the client if it executes another WorkChunk for the same phase and job with a different range of input data indices.

The WebWorker thread sends AJAX HTTP GET requests to retrieve the input data, and then runs the map or reduce function on that data. Finally, when the client has finished computation on all of the input data assigned to it, it issues an HTTP POST of the results back to the job server. After submitting the results, the worker messages the main thread to indicate completion. Upon receipt of this message, the main thread refreshes the page, triggering a new WorkChunk to be issued to the client.

3.6 Interphase Shuffling

Between the map and reduce phase, the server must processes the intermediate data created by the map workers and to prepare inputs for the reduce workers to operate on. We call this the *interphase shuffling*. For each key that is an output from the map phase, all the values from all map outputs tied to that key must be aggregated. Because multiple clients may be assigned to and report back results for the same WorkChunk, this also involves choosing a single client's results for each chunk of work. We choose the most common result amongst all the client outputs for a given WorkChunk. With enough duplicate ClientTasks per WorkChunk, the majority value should protect against clients that return invalid data.

This interphase shuffling can be a bottleneck for a job, because no work can be sent out for that job until shuffling is complete. In our implementation, we run the shuffling in a background thread upon receipt of results for the last incomplete map WorkChunk. Even though it runs as a separate thread, a large enough set of intermediate data can slow the whole job server down. With a more advanced infrastructure, this work could likely be spread out across multiple machines.

Google's MapReduce implementation parallelizes part of the shuffling by having map workers partition their map output into buckets. While copying this approach might have provided some degree of speedup, we chose not to because we wanted finer-grained control over reduce chunk sizes to avoid overloading clients. With partitioning occurring on the server after all map results have been submitted, we can better balance reduce WorkChunk sizes.

4 Methodology

This section describes our testing process. First we describe the jobs that were run in the tests. Next is a description of the hardware and software used on both the job server and on the clients. Finally we explain our metrics and describe how they were recorded during the tests.

4.1 Sample Jobs

To test our system we have crafted two MapReduce jobs that should test the limitations and benefits of our system. Because of the centralized nature of our job server, the latency of the data proxy, and the potentially long round trip times between client and server, the cost of transferring data will be high. We expect jobs with a high computation

per byte of input data to perform well on our system, while jobs with lots of data and relatively small amounts of computation should do poorly.

4.1.1 Traveling Salesman Problem

As a job that was likely to be computation bound instead of I/O bound, we chose a brute force traveling salesman problem (TSP) solver. The solver checks all possible tours of the input cities and chooses the least costly one. The map function operates on a set of permutations of the input cities (defined by a range start and end), calculating the cost of the route represented by each permutation in its range, and outputting the best one. A single reduce task chooses the best route out of all the map outputs.

This job, while somewhat artificial, has an extremely high amount of computation per byte. The size input data, which comprises the distances between the cities is proportional to the square of the number of cities, and the number of paths that need to be checked is proportional to that number's factorial. For our test, which contained 13 cities, 479,001,600 possible paths have to be checked. Each map task contains five input key-value pairs and checks over 5,000,000 paths. With 97 map tasks, the input to the reduce task was 97 tours and their costs.

4.1.2 Word Count

The word count job lies on the other end of the spectrum, requiring very little computation per byte transferred. Figure 1b shows the map and combiner functions for this job. The reduce logic is very similar to to the combiner function's logic. It takes a document as input and outputs a mapping of each word to the number of occurrences of that word in all the documents.

The input is broken into 100 chunks, each chunk containing with 1500 words of Shakespeare's Hamlet. During the map phase, the client retrieves the data with a single request. The key for a given record is the index of that record in the data set. The value is a 1500 word (all lowercase) passage of Hamlet. The reduce data is split into chunks containing 20 keys (unique words in the text) each, which results in 239 chunks.

4.2 Client and Server Setup

In this section we describe the hardware and software for both the job server and on the client machines.

4.2.1 Job Server

The job server is running as a Xen virtual machine on Brown's network. There are benefits and downsides to this. The primary benefit is that we are given total access to the machine. This allowed us to install custom software and do configuration not normally possible on Brown network machines. On the other hand, the virtual hardware is somewhat limited. The server's virtual machine ran Debian Linux 5.0 on a Intel Xeon 2.80 GHz CPU and with 256 megabytes of memory.

We used version 2.2.9 of the Apache web server. Our application was written with the Django web framework version 1.2 and Python version 2.5. Finally, we ran a MySQL version 5.0.51a database using the InnoDB storage engine. The database was used to store both application data and also to store job input data. When a client makes a request to the data proxy, the data proxy issues a request to itself just as it would any other server.

Apache is configured to serve static files automatically and sets caching headers appropriately. After a client requests a WebWorker JavaScript file for the first time, it is cached on the client forever. Clients request two static JavaScript files per job, one for the map phase and another for the reduce phase. A bug in an early version of our system broke the caching and suffered a large performance hit as a result. An improperly configured Apache passed all static file requests to the Django framework. While Django was capable capable of serving static files, it did so in an extremely inefficent manner and did not include proper caching headers.

4.2.2 Clients

Each client was a separate Amazon EC2 machine instance. Each instance ran an official 32 bit Ubuntu 10.10 server image. We used the EC2 small instance size. A small instance contains 1 compute unit and 1.7 gigabytes of memory. A compute unit is an abstraction of a CPU suitable for the virtualized nature of an EC2 cluster. According to Amazon, one compute unit is roughly equivalent to a 1.0 GHz Xeon processor from 2007 [2]. We used the Amazon EC2 API

tools, a set of command line tools for interacting with EC2, to configure and launch client machine instances. Once the instances were running, a simple Java program would connect to each instance and issue commands.

Each client runs Firefox 3.5. In order to run Firefox on an Ubuntu instance with no display, we installed a virtual X server using Xvfb (X Virtual Frame Buffer) [17] on each instance. Firefox did all of its normal rendering and JavaScript execution without the instance actually having to display anything. For each test, Firefox loads the URL for our request work page. The server assigns a task as in response to the client's request. As soon as Firefox loads the page, the WebWorker begins executing and work is started. Upon completion, the page is refreshed. A page refresh is interpreted by the server as a new request for more work. Clients continue to do work and refresh until the job is complete.

4.3 Timing Data

Detailed timing data is necessary to determine the effectiveness of the MRJS system. A breakdown of how time was spent computing a particular job is essential. We have instrumented the system to record timing data at various points in the execution of a job. Below is a description of the timing metrics.

- **Task Total Time:** The time delta between when the server receives a request for work and when the results for that work have been stored in the database.
- **Get Task Time:** The time delta between when the server receives a request for work and when it has chosen a WorkChunk and is ready to respond to the client.
- **Store Results Time:** The time delta between when the server receives an HTTP POST from a client and when the results from this POST have been stored in the database. The overhead of Django's object-relational mapping is included in this time.
- **WebWorker Time:** The total time spent in the WebWorker on the client. The first line of code that a WebWorker executes records the client's system time, and the last line it executes before sending the results back to the server records the ending time. This time includes the task's Computation and Fetching times.
- **Computation Time:** The total time spent in the map, reduce, and combiner functions on the client, measured by noting the start and end time of each of run of these functions and calculating a total.
- **Fetching Time:** The total time spent on the client in the `fetchInput()` function (see Figure 1a), which does an AJAX request to the data proxy to retrieve input data. Measured by noting the start and end time of each of run of these functions and calculating a total.
- **Other Task Time:** The remainder of the Task Total Time after the WebWorker Time, Get Task Time, and Store Results Time have been accounted for. Includes the time spent requesting the WorkChunk's code, thread `postMessage()` communication, and the HTTP POST of the results back to the job server.
- **Other Client Time:** The remainder of the WebWorker Time after Computation Time and Fetching Time have been accounted for.
- **Time Spent In Each Phase:** The map and reduce phase start times are marked when the first task of the phase is served out to a client. The finish times are marked when the results for the last task required to complete the phase are submitted. The interphase shuffling time is calculated as the delta between the last map task finish time and the first reduce task start time. We chose to mark the start time when the first client work chunk is sent out, rather than when the server creates and stores the job (i.e. after it is submitted) to simplify our testing methodology; there is a small gap of time between when the job is created and when the clients begin to request work due to the time spent connecting to each client to tell it to request work. We did not want this gap to unfairly represent the actual time spent doing work.

We believe that this timing data provides an accurate breakdown of how time was spent during a job's execution. The storage and bandwidth overhead of recording this data is minimal. Except for the last metric, all metrics are recorded per ClientTask, and stored in the ClientTask table in the database. The timing data recorded on the client, which includes the WebWorker Time, Computation Time, and Fetching Time, is sent in the HTTP POST data to the job server when the task has finished. In total, there were 7 additional integers fields added to the ClientTask database table and 3 integer fields transferred with each results HTTP POST.

5 Results

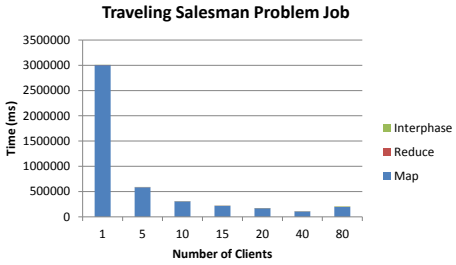
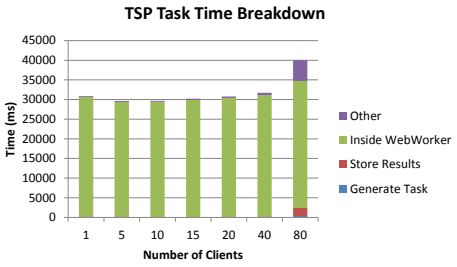


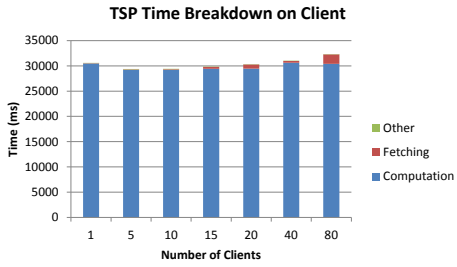
Figure 3: Total time spent executing the TSP job.

5.1 Traveling Salesman

Figure 3 displays the time taken by different numbers of client browsers to complete the TSP job. As we expected, our system performed very well on the TSP job. For up to 20 clients, it scaled almost perfectly, with the job taking less than a 17th of the time to complete when run with 20 clients instead of 1. Unfortunately, we were only able to deploy 20 instances at once due to EC2 restrictions enforced by Amazon. To test beyond 20 clients, we had to use multiple tabs per browser. Despite this, we found that speed continued to improve when scaling beyond 20 clients. By the 80 client mark performance began to degrade.



(a) Average time breakdown of the time spent on an single task in the TSP job.



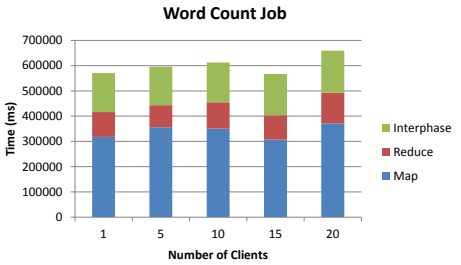
(b) Average time breakdown of the time spent on a client for a single task in the TSP job.

Figure 4: Timing breakdown for the TSP job.

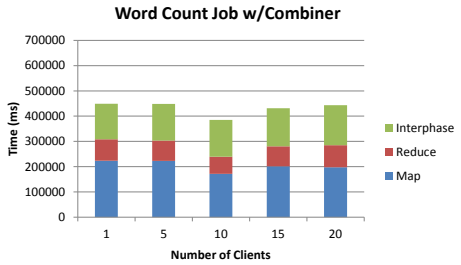
Figure 4a shows the average amount of time spent on the various parts of a single task. Figure 4b gives a more accurate view of the timing on the client, breaking down the time spent within the WebWorker thread. The amount of time spent on the actual computation dominates everything else.

As the number of connections grows, we see that the job server begins to become overwhelmed. The ”Other” time and ”Store Results” time in Figure 4a is grows to its largest with 80 clients. The ”Other” time rises because the server is unable to serve client web requests quickly enough. The CPU can only do so much concurrently; as the number of Apache client processes rises each process gets less processor time. We see a similar problem in the ”Store Results” portion of the 80 client bar. The MySQL database is overburdened, resulting in a slower average response time. Figure 4b also supports this conclusion. The fetching time rises with the number of clients. Since the data for these jobs is stored in the same MySQL database, it takes longer to query the database for the same reasons as the ”Store Results” time. In addition to the database slowness, a client request for data has results in two requests for our already slowed down server: one to invoke the data proxy and another for the data proxy to request the data on the clients behalf.

We ran the TSP tests again with a combiner function that reduced the number of routes reported back per chunk from 5 to 1, and observed a slight decrease in the amount of time spent on the reduce task and on interphase shuffling.



(a) Total time for the word count job without a combiner function



(b) Total time for the word count job with a combiner function

Figure 5: Total time taken to execute the word count job.

5.1.1 Word Count

As we expected, the word count job performed poorly. This makes sense on a general level, as the server is much more likely to be a bottleneck in the case of a job for which large amounts of data need to be transferred to the client for a comparatively tiny amount of computation.

Figure 5 shows the time spent in each phase of the job for different numbers of clients. The scalability of this problem is somewhat startling; even 5 clients took longer than 1. In comparison with the TSP job, a significant amount of time is spent interphase shuffling and in the reduce phase. The amount of time spent shuffling helps to explain some of the lack of scalability, as none of the clients can contribute while it is occurring. We believe this is happening for the same reasons that the 80 client TSP job took longer than the 40 client TSP job, albeit much sooner and much more dramatically. Figure 6 shows that as the number of clients is increased, the time spent on each task increases, driven mostly by increases in the time taken to fetch data and store results.

Including a combiner function with the word count job had a significant positive effect on performance, with the time spent in each phase declining noticeably for all numbers of clients. Figure 5 shows the total execution time for the word count job with and without a combiner function. This makes sense in the context of the word count job, because the combiner function reduces the amount of output from the map phase by a large factor. In the map phase, this translates to a smaller amount of data sent to the server and an accompanying reduced need for server time and resources to store these results. In the interphase shuffling less data needs to be aggregated, and in the reduce phase smaller volumes of input data need to be sent to the workers. Without the combiner function, there were 148,500 values stored in the database, and with the combiner function, only 57,138 values. Despite the shorter execution time, the job still has trouble scaling.

It is also significant to note that in our first attempts at testing the word count job, in which we used about a data set about 100 times the size of the data set used in our final evaluation, we could not get the job to complete in any reasonable amount of time. We checked on the job server and found that it was thrashing severely while during the interphase shuffling. The large amount of data could not all fit in memory at once. To support larger data sets a more sophisticated shuffling algorithm may be necessary.

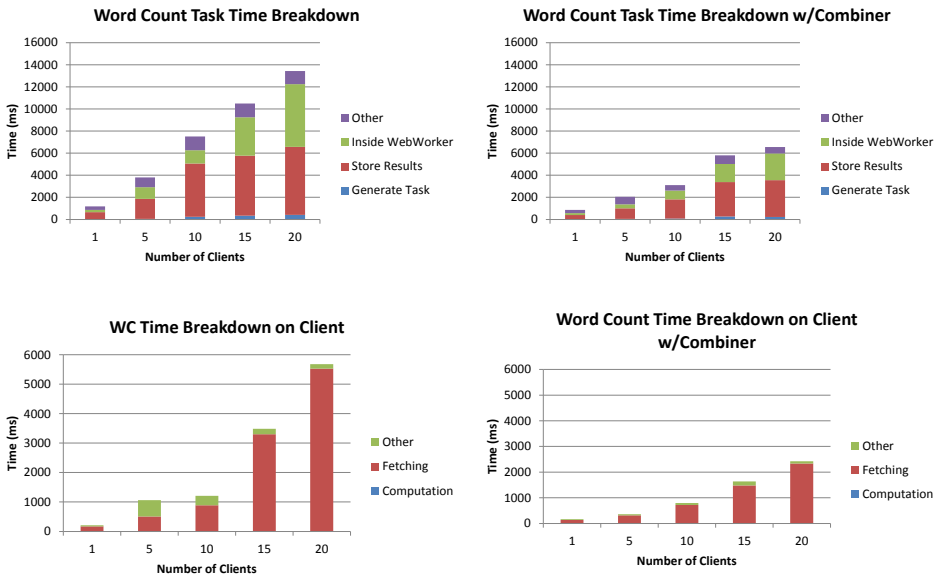


Figure 6: Word Count Timing Breakdown

6 Discussion

6.1 Ideal Jobs

Our system was able to perform quite well on the TSP job. However, it is important to note that a job like this is an atypical MapReduce job. MapReduce was originally proposed to solve problems involving the massive datasets that Google works with, and thus many of the example MapReduce jobs we came across in our research were data-intensive and computationally simple. As we originally hypothesized, our system is not well suited for these types of jobs, and is far more efficient for jobs that require large amounts of computation per byte of input data. This makes sense intuitively, because while each byte of input data imposes stress on the same source, the computation is independent and thus purely parallelizable.

While many traditional MapReduce jobs may be ill-suited for MRJS, a host of more computationally intensive exist that could take advantage of our system. Monte Carlo simulations and algorithms to find large prime numbers fit into this category. Many genetic and protein folding algorithms, if they could be cast to MapReduce, also likely fall into this category, because they rely on iterating over and analyzing many permutations of a small amount of data.

As we have seen, the point at which the server becomes the bottleneck varies highly with the type of job. Some jobs, while perhaps too data-intensive to run with the anemic server we used for our tests, might make sense to run with MRJS using a more powerful server. There are other jobs that would probably never make sense to run on MRJS. Word count, for which the resources required to send a word to a client and receive a response are likely less than the resources required to count the word on the server, is likely an example of such a job.

6.2 Improvements and Future Work

Our results show that the Job server can quickly become a bottleneck. While this bottleneck is dependent on the nature of the job being executed, there are also ways in which the high cost of data transfer can be mitigated.

The MRJS data proxy provided a simple way to bypass the restrictions of the same origin policy. While it is a functional workaround, the cost of additional latency incurred by the input data passing through our server greatly hinders the utility of our system. Had there been a way to make direct contact with the external data source, our system may have been more useful. Frustratingly, during the writing of this section (long after our implementation and testing were complete) we discovered a new W3C draft for cross origin resource sharing (CORS) [15]. The CORS draft calls for changes to the HTTP protocol which allow a user agent to specify optional headers to allow cross origin requests.

These changes have been implemented in the latest versions of Firefox, Safari and Chrome [18]. Thus, if the providers of input data used web servers that followed the CORS standard, there would be no need for a data proxy and workers could retrieve their map input data directly from its source, alleviating load on the job server.

HTML5 introduces a new model for doing local storage on the client [12]. While cookies have provided a limited form of client side storage, they are not suitable for many purposes because they must be transferred to and from the server with every request. The new WebStorage model does not have this problem. WebStorage provides client side storage via a JavaScript associative array. It may be possible to use this storage to improve data locality for some MapReduce algorithms. A caveat is that the task scheduling becomes more complicated; tasks will have to be distributed in a manner which favors clients that already have cached data.

We believe that there is abundant room for improvement in our system. While the job server was a bottleneck relatively early on, it is likely that it can be scaled in many ways. First, the hardware can improve. The CPU and memory of our job server were rather limited; better hardware should increase the total number of concurrent connections we can support. Secondly, there is the possibility of scaling horizontally by adding more machines. Having dedicated web server machines and a cluster of database machines could potentially reduce the burden and increase availability for more concurrent clients. With an improved job server infrastructure, we believe that bandwidth would emerge as a more significant problem.

Finally, a thorough security analysis is necessary before our system can be considered safe for real use. Our system for verifying results democratically, i.e. requiring multiple clients to submit results for the same chunk to protect against submission of incorrect. Due to time and space considerations, however, we decided to leave evaluation of this functionality to future work. Also, a production implementation would most likely require some static analysis of job code to prevent the execution of malicious code. While the JavaScript sandbox protects the end user's operating system from malicious code, there is JavaScript code that can harm both the client's browsing experience and other servers.

7 Related Work

Since Google's original MapReduce publication [9], a number of interesting research projects which aimed to bring the MapReduce paradigm to different computational architectures. There is also a large body of research on volunteer computation systems. Many of these systems have influenced the design and implementation of our system. In this section we describe some of these systems.

7.1 Map Reduce Systems

Hadoop [1] is the open source implementation of MapReduce. It is very similar to the system described by Google. A collection of commodity hardware running as a Hadoop cluster stores its data in the Hadoop distributed file system (HDFS). This fault tolerant file system is designed for large files that are rarely overwritten. Hadoop MapReduce jobs are specified to a master *job tracker*, who distributes the work amongst several *task trackers* to produce a result. Our job server is analogous to the job tracker, while the browser clients act as task trackers.

Several groups have researched the use of MapReduce on shared memory multicore and multiprocessor systems. The Phoenix [14] project is an API and runtime system for MapReduce on multicore architectures. Map and reduce tasks run as different threads, and data is shared amongst them via global shared memory buffers. They find that using shared memory greatly reduces the burden of data transfer, but the overhead of formulating certain algorithms in the MapReduce style is too high. In [6], Chu et al. similarly explore the process of adapting machine learning to shared memory MapReduce architectures.

MapReduce systems on nontraditional shared memory architectures have also seen much success. Mars [11] implements a MapReduce system on graphics processing units (GPUs). Graphics coprocessors were designed with many lightweight cores to take advantage of the data parallel nature of graphics processing. Mars abstracts the graphics oriented programming APIs and instead provides a simple MapReduce interface. Similarly, Kruijf et al. [8] implement a MapReduce system on the cell architecture. The cell architecture is somewhat like a CPU/GPU hybrid. In their system, the master coordinator runs on the CPU-like PPE unit and distributes tasks to a set of GPU-like SIMD processors. Similar to our results, they predict that the performance of their MapReduce implementation varies greatly with the size and type of input data. They note that when there is significant work to be done between phases, the parallel

processing power is wasted. We saw similar a occurrence in our tests; clients had no work to do during the interphase shuffling.

7.2 Volunteer Computation Systems

A number of systems have relied on the computational resources of volunteers to do large scale parallel processing. Two of the most well known volunteer projects are SETI@Home [4] and Folding@Home [5]. Both projects have endless amounts of data that needs processing. By framing the problems as shared-nothing parallel computations, the computation can scale horizontally very nicely. BOINC [3], a successor to the SETI@Home project is a general purpose framework for volunteer computation. With BOINC client software, a client can choose to donate cycles to any number of projects.

There are many complications inherent to volunteer computation. First, a volunteer system is only effective when there are many clients willing to participate. To encourage volunteers to participate BIONC and the @Home projects turn the work into a competition. They rely on a credit system and leaderboards to promote volunteers. We have yet to investigate ways to promote MRSJ. Since computation is performed in a simple web page, it may be possible (with cooperation from popular web sites) to gain processing power by embedding a MRJS work page as a hidden frame on a more popular page. This must be done carefully, most likely with some sort of opt-in system in order to remain faithful to the volunteer premise.

MRJS is immune to many of the security problems that other volunteer computation systems must deal with. In a system such as BOINC, computation is run as an OS level executable. Since anyone can start a computation project, the clients must have a great deal of trust that the project isn't secretly a virus or other malicious program. BOINC uses digital signatures on the executables and MD5 hashing on data to ensure validity. Since JavaScript is sandboxed already by web browsers, there is little serious damage that can be done by a malicious job owner that couldn't be prevented with simple static code analysis. SSL/TLS may be used to prevent tampering with data and JavaScript code. BOINC also does a great deal to prevent clients from cheating on the computation. We have not investigated solutions to these problems in MRJS, but we believe solutions will be similar.

8 Conclusion

In this paper we have discussed the design, implementation and evaluation of a prototype JavaScript MapReduce framework. We saw mixed results on the performance of our system. Performance depended highly on the nature of the job being executed; jobs with a high amount of computation per byte of data did well, while data intensive jobs performed quite poorly. Jobs well suited for our system were able to scale well, and we have noted many ways to enhance the system to further improve scalability. We have learned a great deal from the MRJS prototype, and believe that a large scale volunteer JavaScript computation system can be realized with further research.

References

[1] Apache hadoop project description, 2009. This is an electronic document. Date of publication: [Date unavailable]. Date retrieved: December 14, 2010. Date last modified: September 20, 2009.

[2] Amazon ec2 instance types, 2010.

[3] D.P. Anderson. BOINC: A system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004.

[4] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@Home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[5] A.L. Beberg, D.L. Ensign, G. Jayachandran, S. Khaliq, and V.S. Pande. Folding@Home: Lessons from eight years of volunteer distributed computing. 2009.

[6] C.T. Chu, S.K. Kim, Y.A. Lin, Y.Y. Yu, G. Bradski, A.Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference*, page 281. The MIT Press, 2007.

[7] D. Crockford. Introducing json.

[8] M. De Kruijf and K. Sankaralingam. MapReduce for the cell broadband engine architecture. *IBM Journal of Research and Development*, 53(5):10, 2010.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] J.J. Garrett et al. Ajax: A new approach to web applications. 2005.

[11] B. He, W. Fang, Q. Luo, N.K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[12] Ian Hickson. Web storage. Last call WD, W3C, December 2009. http://www.w3.org/TR/2009/WD-webstorage-20091222/.

[13] Ian Hickson. Web Workers. Technical report, W3C, December 2009. http://www.w3.org/TR/2009/WD-workers-20091222/.

[14] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. Citeseer, 2007.

[15] Anne van Kesteren. Cross-origin resource sharing. W3C working draft, W3C, March 2009. http://www.w3.org/TR/2009/WD-cors-20090317/.

[16] Anne van Kesteren. XMLHttpRequest. Technical report, W3C, November 2009. http://www.w3.org/TR/2009/WD-XMLHttpRequest-20091119/.

[17] David P. Wiggins. Xvfb(1) manual page. This is an electronic document. Date of publication: [Date unavailable]. Date retrieved: December 1, 2010. Date last modified: [Date unavailable].

[18] Nicholas Zakas. Cross-domain ajax with cross-origin resource sharing, 2010.