**Atlas: an intelligent, performant framework for Web-based grid computing**

Jason Brooks  
Yale University  
51 Prospect Street  
New Haven, CT 06520  
jason.brooks@yale.edu  

Sachith Gullapalli  
Yale University  
51 Prospect Street  
New Haven, CT 06520  
sachith.gullapalli@yale.edu  

**ABSTRACT**

This paper presents Atlas, a distributed computing system that allows for collaboration over Internet browsers. It leverages wasted processing power from Internet-connected machines to help leading researchers and companies compute difficult tasks. The platform aims at maintaining similar speeds to available cloud computing services while running these tasks, while doing so at a less expensive cost. In order to do so, Atlas intelligently learns the patterns and habits of its users, and minimizes the amount of time needed per computation. Benchmarks demonstrate that Atlas may be a viable alternative to traditional cloud computing methods.

1. INTRODUCTION AND BACKGROUND

Scientific progress in the past few decades has been phenomenal. Emerging fields such as bioinformatics and computational biology have provided us with deep insight into how the human body works. Companies such as Illumina are now able to sequence the whole human genome for less than $1000 [6]. Most people use the Internet for simple tasks such as reading the news or browsing Facebook. Because most people have very powerful computers, there is a lot of unused computing power.

What if this untapped computing power could be used to solve research problems? In this paper, we discuss Atlas - a distributed computing system that allows for collaboration over Internet browsers. With this service, research problems are executed across all connected nodes in a network, with each node handling a small piece of the total overall computation.

The concept behind Atlas came in the Spring of 2015 as a business idea for a Programming and Entrepreneurship class at Yale University. While that course focused on the entrepreneurship techniques behind building successful software start-ups, it only touched upon implementing the actual technology backing each idea. Due to positive feedback and interest on the idea, the decision was made to build upon the original idea by creating a complete prototype of the system with features to make the software actually effective.

The initial plan for Atlas was conceived with three main goals. The first and main goal was to develop a system capable of taking research problems and input data sets, and develop a way to distribute pieces of the problem across a cluster of nodes connected over the Internet. The second goal was to build a system that could be easily distributed and embedded on Web sites across the world to maximize the number of nodes on the Atlas network. The final goal was to increase the speed of computation over existing Web-based cloud computing methods.

We chose the research space for the initial version of the software since many research problems - from k-means clustering to DNA sequencing to physics simulations - are easily parallelized, yet require significant computing power and resources for results. While heavily funded research universities with their own cloud computing programs (like Yale) do exist, this is not the norm due to significant overheads in price and space[12]. It is not uncommon for researchers to spend significant amounts of grant funding on computing resources to speed up computation time for faster results.

This paper explains how these three goals were approached and met. Specifically, the paper introduces the backing technology behind Atlas, as well as the foundation for a machine learning pipeline to speed up computation.

2. PRIOR ART

Significant research has been completed in the public-resource grid computing space. Two of the earliest implementations were GIMPS (the Great Internet Mersenne Prime Search) and distributed.net [10, 5]. GIMPS used computing power through the web with the goal of searching for Mersenne prime numbers, while distributed.net attempted to use idle CPU and GPU time to solve cryptography problems. These projects led to the creation of BOINC - the Berkeley Open Infrastructure for Network Computing - which provides an infrastructure to easily allow scientists to run large, shared computing projects [1]. The BOINC project started to differentiate Grid computing from public-resource computing. The former relies on machines which are constantly powered, monitored, and managed by professionals while the latter relies on public, decentralized machines. With public-resource computing, a major focus shifts to security, as participants are not controlled by a central source, and can follow malicious behaviors.

BOINC quickly became the backing infrastructure behind many large-scale public-resource computing projects including SETI@home, Folding@home, and Genome@home [2, 7]. These projects aim to use volunteers’ computers for scientific research. These systems are implemented by providing a separate client to install on a PC, which directly accesses the CPU and GPU power to run computations. SETI@home still remains the largest public-resource computing project in existence, with over three million users.
These projects provide the foundation for Atlas, and demonstrate that public-resource computing is a viable method for solving computations. Atlas aims to improve upon these methods by removing the need for a separate client to be installed on each volunteer machine. This limits the scope in which participants can interact and help solve computational problems. In fact, throughout the past five to ten years, a major push has led to solutions that do not require a separate client to function. In a 2007 paper, Boldrin, Taddia, and Mazzini describe a solution for using web browsers as clients for distributed computing [3]. Their solution utilizes Javascript and AJAX technologies to build code computation right into any hosted web page. Single subproblems are sent one at a time from a server to a client, and computation takes place to resolve each subproblem. Finally, a single result is returned back to the server.

More recently, the Capataz framework was developed, which allows for web browsers on multiple devices to contribute to the execution of distributed algorithms [9]. Capataz only allows for the input of algorithms written in Javascript. The system significantly improved upon the Boldrin, Taddia, and Mazzini method by providing a method for bundling different jobs together in order to minimize the amount of time spent computing solutions for similar subproblems. Analysis proved that the framework provided a viable method for distributed computing, and the job-bundling feature reduced execution time compared to previous systems.

The major bottleneck presented by these systems is speed of computation. Atlas significantly improves upon these systems by removing the requirement for input algorithms to be coded using Javascript. Even with simple programs, the physical time a block of Javascript code takes to run is significantly slower than a comparable program in a language like C [4]. With programs using more complex data structures and control flows, this ratio can be as high as 50:1.

People have suggested solutions to this problem of the inefficiency of Javascript. CrowdCL is an open-source framework developed by MacWilliam and Cecka for the purpose of applying OpenCL technologies to volunteer computing [8]. OpenCL allows for the direct access of CPU and GPU power from the web browser, providing the flexibility of a client-less solution with the power of a native application. CrowdCL demonstrates that this method significantly improves run speeds over native Javascript implementations of the same algorithms.

There are two limitations with the CrowdCL approach. First, and most importantly, OpenCL technology is not natively supported in any major browser, and there are no plans to add this support at the current time. Furthermore, while CrowdCL increases the speed of computation, it still does not address a major bottleneck: the amount of time it takes to request and receive new sub-problems, and the amount of time it takes to send partial results back to the server.

More recently, another approach to removing the performance limitations of Javascript was proposed by the Gray Computing framework, developed by Pan and White, although they do not actually implement their proposed changes [11]. Pan and White suggest that a grid computing system utilize asm.js, a compilable subset of Javascript supported by most modern browsers that has brought Javascript execution speed to approximately 65-70% of the speed of native code.

Unlike normal Javascript, asm.js is not intended to be written by hand. Instead, compilers are used to take code written in statically typed languages and compile them to asm.js. One of these compilers is Emscripten, a LLVM backend developed by Mozilla. Because Emscripten can compile code written in C or C++, as a substantial amount of high performance computing code is, it is now possible for this high performance code to be executed in the browser at reasonable speeds.

Atlas makes progress on these issues by both implementing an Emscripten based code translation system (supporting the OpenMP programming model to prevent the need to rewrite code) and by using machine learning to accurately predict the amount of time a user will spend on a given page. Rather than sending a single sub-problem at a time, Atlas will send a proportional number of sub-problems, and limit the number of network requests to a minimum.

3. IMPLEMENTATION

Atlas currently consists of two interdependent extensions to Emscripten: a Javascript implementation of a subset of the OpenMP runtime and a server-client framework that distributes jobs and collects the results.

3.1 Emscripten Extensions

When the -fopenmp flag is passed to Clang, any OpenMP pragmas in source files are parsed and used to transform the AST into the appropriate form – outlining blocks that should be parallelized into their own functions and generating calls to the OpenMP runtime library to handle task allocation and thread creation. Because Emscripten uses Clang as a frontend for compilation, these runtime calls are present in the generated asm.js – they just point to stub functions because there’s no Javascript OpenMP runtime to link. The OpenMP runtime library consists of hundreds of functions, but only two of these are necessary to parallelize a statically scheduled parallelized for loop generated with #pragma omp parallel for schedule(static). The first of these, __kmpc_for_static_init, determines the scheduling pattern, while the second, __kmpc_for_static_schedule, is responsible for spawning “tasks”. Both of these are implemented in Atlas, allowing C or C++ code using the above pragma to be distributed across the Internet with no code modifications. To support a wider variety of execution models, we have additionally implemented support for the reduction clause.

3.1.1 Parallelization Model

In a traditional environment, the objects of parallelization are threads, but in the distributed environment of Atlas we must instead create “jobs”, using the fork-join model. While threads can operate on shared memory using synchronization constructs, the heavy cost of network operations makes these constructs impractical for a performant distributed system. Therefore, we impose the constraint that all jobs must be able to operate independently of each other. In more precise terms, this requires of input programs the following conditions:

1. No job needs to read memory after it is written by another job (note that this still allows the firstprivate clause to be expressed in Atlas).

2. If a memory location is written to by multiple jobs,
the main thread will not attempt to read that location after the completion of the parallel section (or if it does, the last write will be valid (this allows the lastprivate clause to be expressed in Atlas)).

These two conditions ensure the correctness of Atlas’s parallelization model. Because of these constraints, Atlas is best suited to the class of problems that can be considered "embarrassingly parallel", or can at least be broken into several embarrassingly parallel steps. Programs that can be expressed within the MapReduce model are thus good candidates for Atlas.

3.1.2 Job Execution

Upon a call to __kmpc_fork_call, the main process sends the current heap and source code along with the list of jobs to be run to the Atlas server, whose implementation will be discussed below. After performing some initial preprocessing, the server distributes these jobs to clients, who initialize their state to the state of the main process at the time of forking and then jump to the outlined parallel section, using the arguments of the job they have been assigned to execute the correct set of iterations of the for loop. When the job is complete, the client compares its heap to the heap used to initialize it, and sends an object containing all changes back to the server. These changes are merged on the server using the policy of "last write wins" (where order is determined by iteration index). When all jobs have completed and the results have been merged, the final diff is sent back to the main process. The main process applies the diff to its heap and continues execution.

3.2 Server-Client Distribution Framework

Atlas uses Internet-connected Web browsers as nodes for the grid computer. One of the major challenges presented by this setup as compared to a traditional grid computing system is in the availability of nodes. In a traditional system, nodes are static and – to an extent – have guaranteed availability. Conversely, Web browsers are dynamic, and Web pages exposed in those browsers are frequently changed. Since the system only runs on pages that employ the Atlas framework, it is important to maximize the amount of computation that can be completed during a given session.

One of the major bottlenecks of the proposed system is the network time needed to send information back and forth between the client and server. A naive approach to the problem of distribution is to send a single job to a client, wait for computation to occur, send the result back to the server, and wait for the next job. This proves slow as a result of the network bottleneck. Another naive approach to the problem is to send a large quantity of jobs at a time, compute a result for each problem, and send one result to the server. This also proves to be problematic as increasing the amount of time needed on the client also increases the likelihood that a given browser session will change before results are sent back, effectively wasting the computation time.

A major contribution of Atlas lies in its ability to intelligently distribute jobs to nodes. Rather than treating all clients the same, the system makes an intelligent prediction as to the amount of time a browser node will remain open. The number and size of jobs is then modified to maximize the amount of computation that can be completed on that node. This process is described below.

3.3 Intelligent Job Distribution

3.3.1 The Dataset

A YouTube dataset from a recent study at Cornell University was selected to train Atlas. The study provided two unique information logs. The first was a random videos log, which contained public information from the YouTube API on a set of 1,125 randomly selected videos. The second log provided an understanding of how individual users behaved while watching videos. For this, a Google Chrome browser extension was used to collect data from a set of 158 participants who watched 1814 videos. In addition to the standard metadata collected about each watched video from the YouTube API (including view count, favorite count, like count, dislike count, share count, and video duration), the Chrome extension was able to capture a view time stamp, demographic information about the viewer, and dwell time. The dwell time – which is the amount of time the user actually spends on the page after it has loaded – was specifically collected in order to understand how the user behaves on a given video page based on the parameters of the video. Also provided was a sentiment analysis of the comments for each video, which implicitly contained information about the emotions viewers felt after watching.

One of the limitations of the YouTube dataset is its relatively small size. Furthermore, given its specificity of parameters found on only video sharing sites, it is not generalizable to a wider range of sites found on the Internet. Given these limitations, other larger and more generalized datasets were considered for use. Since the YouTube dataset was the only one that contained dwell time information – a factor needed for building the model – it was ultimately chosen.

3.3.2 Dataset Analysis and Model Generation

For the purposes of training Atlas, only the individual log dataset was used. As the proposed contribution of the machine learning pipeline was to provide accurate information as to how long a user spends on a given Web page, the goal was to accurately predict the general_engagement column for each entry. This data point provides a percentage of dwell time on a given YouTube video page as a ratio to the total length of the video.

Extensive testing was performed across all 107 parameters collected for each entry in the log. Half of the log was used as a training set and the other half was used for testing. Different combinations of related variables were tested in an effort to find a model with a strong R-squared value. A PCA analysis was also used to isolate seemingly uncorrelated variables that may have been useful in a regression.

Ultimately, general engagement was predicted using a linear model. The predictors used were a combination of discussion_density (the proportion of comments that are replies to other comments), share_percentage (shares per view), like_percentage (likes per view), and numChild (the number of replies on any comment). This model provided an $R^2$ value of 0.21, which was statistically significant for use on the machine learning model.

Initially, a more complex analysis was attempted using the sentiment data provided in the dataset, but the results were noisy and not useful in the model. The PCA analysis also did not provide better results in practice.

3.3.3 Model Implementation
The generated model allows the initial version of *Atlas* to support an offline machine learning system. It is used by default on pages that contain the parameters used in the model generation.

When a page containing the *Atlas* distribution framework loads, a request is made to the server – via an asynchronous AJAX call – which contains page metadata. That information is used to make a prediction as to how long that browser session will remain alive. Jobs are manually clocked for iteration speed when a given research problem is first uploaded to the system. When jobs are ready to be sent to the client, a set of jobs that totals the predicted time are selected. When the server receives results from the initial set of distributed jobs, it continuously sends single jobs back to the client until the connection dies.

Future versions of *Atlas* will integrate an online machine learning pipeline to continuously refine the model.

### 3.3.4 Client

Upon loading the embedded Javascript in a Web page, the client also polls the server to request the corresponding asm.js and initializer heap from the server for the chosen jobs. Once all necessary data is received, the client launches a web worker in the background (preventing computations from adversely affecting page responsiveness) and executes the parallel section once for each assigned job. While computation is ongoing, the client periodically sends a heartbeat to the server to verify that the client has not left the page.

### 3.4 Server

A lightweight node.js server is used to handle interactions with each distribution client. The server exposes endpoints for sending jobs, receiving results, initializing tasks, and reducing partial results into complete solutions.

The server also allows for execution failure for a given job. Using the heartbeat system, the server has the ability to recognize when nodes are no longer connected in the grid, and can redistribute those failed jobs efficiently to finish computation.

## 4. EXPERIMENTAL RESULTS

We benchmarked both the general performance of *Atlas* and the validity of our intelligent job distribution algorithm.

### 4.1 Performance on K-means Clustering

The highly parallelizable nature of K-means clustering makes it a good candidate for *Atlas*. Andrea Ferretti’s K-means Benchmark Project aims to compare the performance of a naive K-means clustering algorithm in different programming languages, focusing on idiomatic code over extreme optimization. We timed runs of the OpenMP and Javascript code from this project to compare the scalability of *Atlas* to that of a single-machine multicore system running the same code, as well as the raw performance of the compiled asm.js vs the idiomatic Javascript previous projects have sought to distribute.

#### 4.1.1 Experimental Configuration

The Atlas server and main Node.js process were run on an Amazon EC2 c4.xlarge node with the following specifications:

- 4 Intel Xeon E5-2666 v3 CPUs
- 7.5 GiB RAM

To simulate clients visiting a webpage, we used Selenium to run Firefox (version 46) browser instances on different cores, split across 2 Amazon EC2 c4.8xlarge nodes with the following specifications:

- 36 Intel Xeon E5-2666 v3 CPUs
- 60 GiB RAM

### 4.1.2 Data

Unparallelized, the native C version took 242 seconds to run, *Atlas* took 481 seconds to run, and the equivalent pure Javascript code (executed in the latest release of Firefox’s Javascript engine, Spidermonkey 38) took 10,297 seconds to run. This is consistent with previous results that have shown pure Javascript to run up to 50x slower than native code and compiled asm.js to be within a factor of 2.

![Performance on 30k iterations of k-means](image)

<table>
<thead>
<tr>
<th>Multiprogramming Level</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Native</td>
</tr>
<tr>
<td>2</td>
<td><em>Atlas</em></td>
</tr>
</tbody>
</table>

Although native code running on multiple cores on a single machine will always be faster then a distributed system running Javascript on the same number of nodes, these results demonstrate that *Atlas* performs quite respectably – never less than half the speed of the equivalent native system, while demonstrating far greater capacity for scaling. Moreover, *Atlas* outperforms the pure Javascript implementation by a factor of over 20, demonstrating the power of asm.js as a tool for building high performance distributed systems. Had we had time to design and benchmark a distribution system for the pure Javascript code, this means its performance would have been capped at 5% of what we were able to achieve with this initial prototype of *Atlas*.

### 4.2 Performance of Intelligent Job Allocation Algorithm

#### 4.2.1 Experimental Configuration

The intelligent job sizing pipeline was tested using the Computer Language Benchmarks Game Spectral Norm benchmark. This benchmark computes the spectral norm of a matrix using the power method. For the purpose of benchmarking *Atlas*, the matrix was sized to 30,000. The results of this computation were compared against two other implementations, as described below.

First is the Naive Approach as discussed above. Here, we end a single job to the client, wait for computation to finish, send the result back to the server, and request a new job.
The second is the adaptive algorithm proposed in the Gray Computing system [11]. This system utilizes a Weibull distribution to model dwell time on Web sites. Effectively, the probability that a browser session ends becomes smaller as time after page load increases. This system only provides a theoretical approach to the problem with benchmarks. As a result, here we start by sending a single job to the client, and then double the number of jobs sent in each successive call to the server. The first call will send a single job, the second call will send two jobs, the third call will send four jobs, and this pattern continues until the node is removed from the grid.

The same Selenium simulation framework as described in the previous section was utilized. The system was tested with different numbers of jobs remaining in the queue, ranging from 30 to 3000 total jobs.

4.2.2 Data

As expected, all three tests yielded similar results when the total number of jobs needing to be computed equaled the number of browser instances. This computation took between 190 and 200 seconds depending on the test. The machine learning pipeline significantly outperforms both the naive approach and the adaptive algorithm proposed in the Gray Computing system when a greater number of jobs are given. In fact, Atlas outperforms the naive approach with a speed increase of nearly 2x. While the naive approach takes 423 seconds to run when given 3,000 jobs, Atlas takes only 240 seconds.

These results exemplify the importance of sizing the number of jobs sent to the client. The reason that the machine learning pipeline outperforms both the adaptive algorithm as well as the naive approach lies in its approach to minimizing network requests. By intelligently calculating the amount of time it expects the browser to remain open, both inputs and results for multiple jobs can be sent via a single network request. This can be seen in the fact that all jobs require the same amount of computation - the only difference between the three algorithms is the number of calls to and from the server needed to receive inputs and return outputs.

5. CONCLUSIONS AND FUTURE WORK

This paper describes all the components that Atlas requires to effectively operate as a browser based grid computer. Atlas in its current iteration is a rough prototype of the described system. It is clear that there are still several areas for improvement that can increase its overall effectiveness and efficiency, as discussed below.

Currently, the Atlas programming model requires that jobs be completely independent - no synchronization primitives can be used. This is tied to the fact that network requests are currently a bottleneck for Atlas. Even for jobs requiring little data transfer, the high latency of HTTP requests reduces performance dramatically, meaning that Atlas is only suitable for programs where the computation time of a single job is far longer than the latency of the network. Both these problems may be ameliorated by using new technologies like WebSockets and WebRTC, which enable low-latency and peer-to-peer data transfer in the browser.

In addition, the traditional web workers used to run jobs on clients can be replaced with shared web workers. While standard web workers are single scoped, shared web workers can persist through sites run on the same origin. As a result, computation can continue while users browse different pages of the same site.

At the moment Atlas sends the entire heap to every node, while in most cases only a small part of it is ever accessed. Future work will seek to reduce the data transfer requirements by predicting data accesses in advance, and only initially sending the region(s) of the heap to the client in which data is predicted to be accessed.

There is also work to be done in extending the scope of the Atlas programming model. Today, MapReduce is one of the most popular models for parallel, distributed algorithms. Atlas effectively implements the “map” part of this model, but the reduction step (if specified in the OpenMP pragma) is conducted in serial. Further work will extend Atlas to perform reductions in parallel as well.

Finally, a larger and more generalized dataset can be used to train the machine learning model. This will allow for more pages to use the machine learning pipeline over the naive pipeline.

Nevertheless, the current technology behind Atlas seems promising, and this will be verified with more testing and refinement. The goal over the next few months is to incorporate some of the discussed changes. With this, hopefully browser-based grid computing will eventually become an effective alternative to traditional cloud computing methods.

6. ACKNOWLEDGEMENTS

A special thanks to Professor Daniel Abadi of the Yale University Computer Science department for advising us on this project.

7. REFERENCES


