**Atlas: an intelligent framework for Web-based algorithm distribution**

Jason Brooks  
Yale University  
51 Prospect Street  
New Haven, CT 06520  
jason.brooks@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. Finally, the last goal was to increase the speed of computation over existing Web-based cloud computing methods.

The research space was chosen for the initial version of the software, since many research problems - from k-mer analysis 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[11]. 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].

Existing systems do exist to fill this void. 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.

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. Atlas makes progress on this issue 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. USERS

There are three distinct users of Atlas: researchers, Web site owners, and the general population who browse Web sites that employ Atlas’ technology. Each type of user interacts with the platform differently.

#### 3.1 Researchers

Researchers - the primary users of Atlas - interact with the platform to crowdsource computational power to solve complicated problems. These users begin by creating accounts, which allows them to upload both JavaScript code as well as a set of inputs. Each set of code and inputs is referred to as a single ‘problem’. Once the server has this information, it can split the inputs into jobs, where each job represents a single set of inputs that are sent to a client node for computation.

Researchers then have the ability to select the “reach” of their computation. Essentially, this provides the cap of unique nodes that can simultaneously run a given research problem. Providing an option for researchers to determine their own reach allows them to find the appropriate balance between cost to run and amount of time needed to finish all jobs.

Finally, execution can begin on the problem. From a central dashboard, the number of completed and remaining jobs can be monitored with real-time updates. The reach of each problem can be dynamically changed while being monitored.

Atlas allows for the simultaneous running of multiple problems for a single user.

#### 3.2 Web site Owners

Web site owners are vital to the success of Web-based algorithm distribution. As more and more websites support the Atlas platform, the overall reach and number of unique visitors who can assist with computation increases. The integration process is simple - a single line of JavaScript is embedded on any existing Web page and the Atlas delivery system completes the rest of the work. Owners are paid proportional to the amount of traffic and unique visitors driven from their site.

The simplicity of the delivery system makes it extremely easy for Web site owners to turn on and off Atlas’ functionality at their discretion. The same simplicity also makes it user-friendly to only deploy the delivery system to a subset of site users.

#### 3.3 General Population

In the current iteration, jobs instantly begin running when a page employing Atlas’ technology loads. No additional steps are needed for jobs to compute results and send information back to the server. This is the major benefit of Web-based algorithm distribution - due to the ‘sandboxed’ nature of JavaScript in Web applications, the security risk to the end user is minimized.

### 4. IMPLEMENTATION

The technology powering Atlas is split between server code, which is used to power the learning and code distribution pipeline, and deployment code, which is used to run jobs on Web browsers.

#### 4.1 Server

Server code follows the design of a standard Model-View-Controller (MVC) framework.

##### 4.1.1 Model

One of the main goals of Atlas is to allow for flexibility in its use. In an ideal world, any algorithmic problem should be able to run through the platform with little to no infrastructure changes. As a result, the decision was made to use a document-oriented database for the model. One of the main benefits of a document-oriented database is that stored
objects belonging to the same class of information may have different attributes. MongoDB was ultimately chosen to fit this need due to its native support for dynamic schemas, as well as its ability to store raw JSON objects.

The current version of Atlas utilizes six distinct data models. Every object across all models contains a unique identifier called an ObjectID, which acts as a primary key for a document-oriented database. This ObjectID is set automatically on the creation of each record. Having this unique identifier is very helpful when mapping the execution of each main problem to a set of smaller jobs.

The first of the six models allows for the creation of user accounts for researchers to input and manage their tasks. The data is represented as follows:

```json
User
{
    "id": objectID,
    "name": string,
    "email": string,
    "company": string,
    "password": string
}
```

Four of the remaining five data models are used to aid in the distribution and execution of research problems. Atlas creates a unique object to represent each problem that researchers upload. The object contains metadata regarding the research problem, a reference to the user account of the person who initiated the problem, and the maximum number of nodes that can concurrently run subproblems of the algorithm. Furthermore, embedded in each problem object is a set of remaining jobs as well as a set of already computed solutions from prior iterations on remote nodes. Each embedded job and result provides a reference back to a complete unique object.

```json
Problem
{
    "id": objectID,
    "name": string,
    "numNodes": int,
    "jobs": EmbeddedDocument,
    "results": EmbeddedDocument,
    "author": Reference(user)
}
```

Job and result objects are used to keep track of partial subproblems and results. Each job contains a status flag to maintain its current state (not started, in progress, or completed). This allows the Atlas mapping system to figure out the next set of jobs to send when a new node connects. Consequently, the models for these two objects are as follows.

```json
Job
{
    "id": objectID,
    "data": string,
    "status": int
}
```

```json
Result
{
    "id": objectID,
    "data": string,
}
```

The final data model is to aid in learning the habits of the general population who use the Atlas system. The model stores a unique identifier that is created for a user the first time they connect to the system. Using that uniquely assigned value, users can be referenced across all Web sites that deploy Atlas. The client object also stores a list of problems the user is currently solving in their browser, as well as a date stamp for the last time they solved a job.

```json
Client
{
    "id": objectID,
    "uuid": string,
    "lastUpdate": dateTime,
    "problem": Reference(problem)
}
```

4.1.2 View

Atlas employs a custom HTML, CSS, and JavaScript based front-end. The current version of the system only provides user-facing views for researchers. It provides infrastructure for creating accounts, initializing research problems, and controlling the reach of a given problem. Atlas utilizes sockets to provide instant real-time updates to the current state of a given research problem. Finally, researchers can visualize activity on each of their problems over certain time spans which can be fine tuned for detailed analytics.

![Atlas interface](image)

Content is dynamically rendered using the Jinja 2 templating language.

4.1.3 Controller

Controller logic for Atlas is accomplished in Flask, a lightweight web framework written in Python. The decision to use Flask was after several considerations. First, using a framework written in Python allowed for rapid prototyping, given the breadth and depth of the python package library. Second, it was important to find a controller framework that minimized the distance between all layers of the system. Flask is extremely lean, minimizing the round trip time between sending successive computations. Furthermore, it has built-in support for multithreading, allowing many clients to connect to the same server at the same time.

A basic controller is used for registering and authenticating researchers in the process of uploading and monitoring problems.

The majority of the controller logic, however, is used for the process of connecting to clients, distributing jobs, and...
reducing the results to final solutions. This exists in the form of an API built on top of the main server.

4.1.4 Handshake Process

The controller maps incoming POST requests to an endpoint called /initiate-session to the handshake initialization function, which determines if there are any jobs waiting to be solved. If this is the case, the function grabs a job and its associated parent problem, and sends a response to the client confirming that it is set to run jobs for that problem. In the event that no problems are available, the handshake returns a special token signifying that the client can remain in hibernation for the time being.

4.1.5 Distributing Jobs

A controller endpoint titled /get-job/<problemID> exists, which takes in a variable parameter - the unique ID for a problem. It uses this ID to fetch a new set of jobs for the client. Another endpoint titled /push-histogram/<problemID>/<jobID> is used in order to receive the computed results from jobs. It creates a new result object and modifies the appropriate metadata across the database schema to reflect that new subproblems have been computed. A signal is also sent to the researchers’ dashboard to update them of the progress. Atlas continuously sends and receives sets of jobs until a Web site user navigates off of a page.

One of the unique features of Atlas is that the backing function behind this endpoint aims to intelligently calculate the number of jobs to send in a single iteration. One way in which it does this is to examine the source of the cite a user is visiting from. In its current iteration, Atlas aims to learn how users interact with content-based Web sites, including newspapers, blogs, and other articles. By predicting the amount of time that a user will spend on a particular page as a function of the length of the content and the reading speed of a given person, Atlas computes a predicted number of jobs to send during each network call. This minimizes the number of total network calls, and maximizes the speed of computation. Over time, the server tracks how long users take to read content, and can adjust the number of jobs sent each time accordingly.

4.1.6 Reducing Results

Atlas currently employs a basic approach to reducing the results into meaningful data, by allowing researchers to download the results of each job. In this iteration, researchers are responsible for then combining the results into a meaningful result.

4.2 Deployment

The deployment engine of Atlas is used by Web site owners to integrate the technology into their existing sites. Ease of integration was the primary concern when designing the deployment system, so as a result, the entire process is integrated and initialized with a single line of JavaScript embedded into the given site:

```html
<script src="localhost:5000/static/js/delivery.js"></script>
```

Executing this line of JavaScript in the browser starts the multi-stage deployment process. First, the given site is analyzed to see if several needed libraries, including jQuery, are already present. If any library is not present, the deployment process injects the needed dependencies onto the page. Next, the deployment engine connects with the Atlas server, and begins a handshake process. The process starts by requesting on if there are currently any jobs waiting to be run. If no jobs are in the queue, the platform disconnects from the site.

If there are jobs, waiting to be run, the engine returns the ID of the job. The client JavaScript code then checks the presence of a unique tracking cookie, which is used to reference individual users as they go from site to site. If the cookie does not exist, a new identifier is generator and set before moving forward. Finally, the handshake is complete, and the client signals for jobs to begin streaming for computation. The client requests the JavaScript file associated with the given problem to use for the computation.

Atlas uses special cross domain access control methods to allow for requests to the platform to be made from independent Web sites on different domains.

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.

First, the greatest bottleneck in the pipeline is the amount of time it takes to run computation through the Web browser. This problem inherently comes with JavaScript. The next iteration of Atlas will examine the feasibility of incorporating the ideas from the CrowdCL project, by integrating native support for WebCL. By natively accessing each node’s CPU and GPU, computation can occur significantly faster. However, as no browsers currently support this technology, the results would be experimental and non-scalable.

Another bottleneck that Atlas exhibits is the number of simultaneous connections the Flask server can handle. A proposed solution is to reconstruct the server using the Go programming language, which has much stronger native support for networking and multiprocessing. Furthermore, since Go is a compiled language, it may help speed the overall computation process over an interpreted language such as Python.

Furthermore, the next iteration of the software aims to allow the final step of the process – reducing the jobs into a meaningful result – to occur directly in the same Atlas ecosystem. This decreases the total overhead of using this service over a traditional cloud computer.

Regarding testing, another future goal is to find a large cluster of machines to benchmark the system and check how robust it is when faced with significant traffic.

Finally, one of the most promising features of the system is the machine learning pipeline. The next version of Atlas aims to build out a much more advanced version of this infrastructure to see its effectiveness in speeding up the computation process.

Nevertheless, the current technology behind Atlas seems promising, and this will be verified with more testing and refinement. The goal over the next six 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 me on this project, and to Sachith Gullapalli and Minh Tri Pham, two of the original members on my team for a Programming and Entrepreneurship course, from which this idea was created.

7. REFERENCES


