1 Introduction

Over the past thirty years, the ubiquity of the computer has spurred every scientific and technological field to become increasingly data-driven. As processors have become more powerful, the size of the datasets they are expected to deal with has also correspondingly increased. Because of this, computing power has become a major bottleneck for researchers in fields from bioinformatics to cosmology. Projects like Folding@Home and BOINC@Home have sought to address this problem by enlisting volunteers to install software on their personal computers that donates idle cycles to solving highly parallel research problems. Although these projects have tremendous potential, they have suffered from lack of adoption. Cusack et al. analyze the reasons for this unpopularity, and find that chief among them are "lack of awareness, limited demographic, and lack of technical savvy." Knowledge of these projects is largely limited to technically savvy users (many of them involved in the research community themselves), and others who might want to join find the difficulty of installing complex software a high barrier to entry.

We suggest that a Javascript-based distributed computing platform that executes code in users’ browsers solves for many of the adoption problems that have plagued such platforms. 85% of Americans today use the Internet, and visiting a website repre-
sents a far lower barrier to entry than installing and configuring software. Moreover, a Javascript-based platform has the potential to be embedded in other websites, allowing website owners to donate the idle cycles of their users to solving these problems.

Until recently, such a platform would be incredibly impractical. Code written in Javascript has traditionally run at just a few percent of the speed of equivalent code in C or C++ compiled for the native platform. However, the development of asm.js, a compilable subset of Javascript supported by modern browsers 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, without any rewriting necessary.

This presents a tremendous opportunity for the distribution of massively parallel computations. If we can divide one of these computations into $n$ chunks of Emscripten-compiled asm.js code, each individual chunk can be sent to a different browser for execution, with the results returned upon completion – all with performance comparable to running the computation natively on a multicore system with $n$ cores (assuming, of course, that the computation is CPU-bound, not I/O bound). If this division is structured in terms of existing OpenMP pragmas for parallelizing work across multiple cores, like #omp parallel for, the resulting system will be nearly transparent to the programmer.

The purpose of this project was to build a distributed computing framework to extend Emscripten with support for the automatic distribution of arbitrary parallel computations specifiable in the OpenMP programming model.
2 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.

2.a 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_fork_call, 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.

2.a.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 \texttt{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 \texttt{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.

2.a.2 Job Execution

Upon a call to \texttt{__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.

2.b Server-Client Distribution Framework

2.b.1 Server

The server is responsible for distributing jobs to clients, which it does intelligently and adaptively using a machine learning algorithm developed by Jason Brooks. It also collects results upon completion and merges them before sending the final result of the parallel computation to the forking process. The server additionally provides fault tolerance – keeping track of currently running jobs and reassigning jobs given to disconnecting clients.

2.b.2 Client

Upon loading the embedded Javascript in a webpage, the client polls the server to request a batch of jobs, along with the corresponding asm.js and initializer heap from the server. 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 Experimental Results

We benchmarked both the general performance of Atlas and the validity of our intelligent job distribution algorithm.
3.a 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. To benchmark

3.a.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

3.a.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.
Although native code running on multiple cores on a single machine will always be faster than 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 runs twenty times as fast as the pure Javascript implementation, even after accounting for the network transfer and initialization overhead incurred by the former. 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 Conclusion

Over the course of this project, I designed and implemented an extension to Emscripten that allows massively parallel programs written in C or C++ that utilize the OpenMP programming model to transparently distribute work to a network of browsers connected to the server. The resulting system represents a paradigm shift from previous systems to distribute computations over the web in two respects.
1. Speed. The performance of previous systems have been constrained by their use of Javascript, a weakly, dynamically typed language with less scope for optimization than C. Atlas is able to combine the ease of distribution of Javascript with the performance of C by using Emscripten as a front-end – outperforming previously designed web-based distributing systems by a factor of twenty.

2. Ease of use. The amount of high-performance code written today in Javascript is virtually zero. Utilizing previously developed browser-based distributing computing systems requires a complete code rewrite, whereas Atlas is able to distribute programs written using OpenMP, a widely used API for parallel applications, with no code modifications.

However, the limitations of the approach used here must be noted. 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. Furthermore, 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. Finally, there is 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.
5 Acknowledgements

A special thanks to Professor Daniel Abadi for advising me on this project, and to Jason Brooks, who I collaborated with on the design and implementation of the system.

References
