Abstract

Fortune magazine estimates that Apple sold an average of more than 5 million Macs per quarter in 2015 [1]. The PC market is even larger. Most of these devices boast impressive hardware specifications. However, the use cases of an average computer owner's machine (browsing the web, text editing, etcetera) suggest that the vast majority go unused. Given the abundance of untapped computational resources, it seems counterintuitive that developers depend so heavily on cloud giants to run their servers and perform their distributed computations.

BARK presents a tailored application of BOINC (Berkeley Open Infrastructure for Network Computing), a framework for volunteer computing, to Apache Spark, the increasingly popular big-data processing tool. Given a Spark job and a pool of volunteer computers - anything from dedicated Linux workhorses to personal laptops - BARK will set up a Spark cluster, exclusively comprised of volunteer nodes, and execute the job across it. In this way, BARK offers a free alternative to for-charge cloud services for end-users interested in performing MapReduce computations. BARK is an example of a style of application that may become increasingly prevalent as the more fluid flow of computational resources between participants in peer-to-peer networks becomes normalized.

1. Introduction

The vast majority of the world’s potential computing capacity is wasted in idling systems, with powerful personal laptops, desktop machines, and even cellular devices using only a fraction of their valuable CPU and storage capacity. At the same time, the need for and the purchase of scalable, for-rent computational resources has never been more common. We intend to transform the underused personal and professional computers of today into the low-cost computing cloud of the future.

As the networks between these machines become ever more reliable and the world’s computational needs continue to grow, a variety of tools have been developed in an attempt to bring work to the otherwise unused computers sitting in homes, offices, and datacenters across the globe. Each must strive to overcome the unique challenges in persistence, synchronization, and parallelization faced by any program run in the cluster computing environment. But even with a dramatic increase in development over the past decade, distributed systems remain largely
unable to usefully incorporate the variety of operating systems, processors, and user machines that so often go to waste. Significant work and research has already pushed this field to the forefront of computer science but several particularly influential and relevant tools will be our focus.

1.1 Amazon Web Services (AWS)

By far the most popular and arguably the most comprehensive distributed computing suite is provided by AWS, which includes Amazon’s Elastic Cloud Compute, or EC2. Allowing users to easily rent customizable clusters of available servers, stored in a variety of data centers, EC2 offers developers the power of thousands of parallel machines to quickly complete computations that would otherwise be infeasible. A giant in the field of cluster computing, EC2 is the benchmark against which the viability (and pricing) of all other distributed computing platforms will be measured given its estimated 73 percent share [2] and dominant pricing schemes. With five billion [3] in revenue, EC2 demonstrates the sheer size of the market for computational resources and the potential profitability of an enterprise that can provide computational resources as a reliable service. Utilizing scale to reduce cost, Amazon has thus proven that there exists an incredible demand for computer resources. What it does not do however, is make use of the underutilized machines in which we are interested, and for that we must turn to our next tool.

1.2 Berkeley Open Infrastructure for Network Computing (BOINC)

BOINC, or the Berkeley Open Interface for Networked Computing is a distributed computing platform that connects volunteers’ unused CPU time with a variety of computationally intensive academic projects run on independent servers. Signing up with a client program that gives project owners partial access to their computer’s processor and storage, a volunteer earns intangible credit in the form of “cobblestones” (named for the developer of the well known SETI@Home project [4]) but otherwise donates their machine. Developers of BOINC projects are primarily academic and scientific institutions with insufficient computational power to perform their analysis and thus present trusted and reputable recipients for a user looking to donate otherwise unused resources.

With over three million users and boasting a combined average of 190 petaFLOPS [5] over 58 supported projects, BOINC represents the largest foray into the use of widely distributed personal computers to perform calculations normally requiring a datacenter. But with the variety of hardware and operating systems on this volunteer cluster, control over the execution of massive computations is difficult and requires a large degree of system dependent control. With this control comes complexity. Traditionally, BOINC projects take roughly three man months to develop with a recommended one month of work allocated from each a systems administrator, an experienced programmer, and a web developer. This considerable barrier to entry restricts access to the massive (and ever growing) pool of resources amassed on the BOINC network to only the largest and most technically savvy laboratories and organizations. Even those willing to tackle the daunting task of BOINC project development must contend with attracting volunteers to their project, making the framework unworkable for one-off computations that fall outside the bounds of a the organization’s current resources. But to meaningfully improve the development experience when working with parallel computation at this scale, we need only look to our final relevant tool.
1.3 Apache Spark

Apache Spark is a open source cluster computing framework that is in many ways the successor to Hadoop, the popular Apache implementation of the MapReduce programming model. Relying on the consistency of its core data structure, the resilient distributed dataset (RDD), developed in part to ease the restrictions placed on computations using the highly specific MapReduce paradigm [6], Spark allows for fault tolerant distributed computing while requiring relatively little computational and administrative overhead. With map and reduce functions in addition to the more flexible direct data access offered by the RDDS, Spark is a formidable tool when executing massively parallel computations.

Run on a single machine or over a specified cluster of connected computers, Spark can administer tasks and collect computed data from its composite pieces consistently, quickly, and largely independently of the system architectures of the underlying nodes. A cluster manager is used to maintain this distinctive structure, with a master node allocating work to the cluster of connected worker nodes that is the hallmark of Spark computations. With its portability and comparative ease of use, Spark effectively complements the power of BOINC and offers access to its trove of volunteer CPUs when implemented as a BOINC project itself.

1.4 BARK

Finally having had an overview of these existing tools, which dominate the respective landscapes of for-rent-computation, volunteer-computation, and distributed execution, the importance and technical challenges of executing Spark jobs over volunteer nodes can be examined. The ability to submit Spark jobs directly to the volunteer cluster will dramatically decrease the amount of time necessary for a Spark developer interested in running computations on volunteer nodes to do so. Not only this, but it will allow development in Spark, a well documented and growing tool specifically designed for these massively distributed computations. With the ease of use and computational power Spark jobs bring to the BOINC volunteer cluster thus established, attention can be turned to the implementation of BARK itself.

By acting as a cluster manager coupled with the BOINC framework, we can create a cluster (in the master-worker node structure mentioned above) constructed purely of volunteer nodes, across which Spark jobs can be run. A web interface allows a user, for example, a scientist seeking to run a Spark job, to input a spark job, written in python (at this point BARK does not support java or scala jobs), which is then executed across a subset of available volunteer machines. This execution returns a value which is then output through the UI. Although the BOINC project itself, that is the cluster manager, must remain active and is under the direct control of the project developers, all computations are performed by the volunteer nodes they connect. Relying on the persistence and scalability guarantees of Spark, the power of the cluster is limited only by the number of available volunteers. A BOINC project capable of running Spark jobs allows the easy submission of a wide variety of computations and represents a dramatic improvement over native BOINC for many computations.

2. Technical Approach

The central design philosophy of BARK is to, as much as possible, play to the strengths of both BOINC and Spark. Too often, a marriage of technologies results in a product that is significantly less powerful than the sum of its parts. To that end, the BARK implementation
features a decoupled design, separating the roles of Spark and BOINC such that each can do what it does best. BOINC is responsible for connecting the system to volunteer nodes. Once these nodes are connected, Spark is responsible for efficiently computing the desired result. Essentially, BARK acts as a cluster manager for Spark, using the BOINC framework to create and track clusters consisting exclusively of volunteer computers.

2.1 Basic structure

In the ensuing explanation of the BARK implementation, it may be useful to refer to the architecture diagram included below. Although simplified for aesthetic purposes, it provides a broad-strokes visual of what will now be described in words.

BARK serves from a static IP address on an EC2 instance. It runs a Web User Interface, driven by Node.js, as well as the actual BOINC server. Various daemons also run on this instance, each tasked with a different piece of the core BARK functionality. Crontab is used to schedule the execution of BARK daemons. The BOINC framework provides a C++ API that allows BARK daemons to change BOINC state and execute built-in procedures provided by the framework. Volunteers communicate with the BOINC server via shared memory. BARK also adds two MySQL tables, spark_job and spark_node, to the existing BOINC schema. These are instrumental in keeping track of submitted jobs, as well as their associated volunteer clusters.

The basic execution path of the BARK system is as follows: (1) a user submits a Spark job via the BARK user interface (2) BARK tasks itself with creating a valid Spark cluster (depicted below as one master and two worker nodes) (3) BARK relinquishes control to Spark, which proceeds with computation over the cluster it has been presented (4) the result is written to a file and sent back to BARK for post-processing.

2.1 Work Generation

Among the most important daemons in the BARK system is spark_work_generator (abbreviated to work_generator). This code is responsible for creating workunits - a term BOINC uses to refer to jobs ready to be sent to available volunteers - as they become needed. When a user first submits a Spark job, the table, spark_job, becomes updated with relevant information. This includes job status details and the name of the provided Spark job file. Work_generator periodically scans this table for unhandled jobs. Should it find one, it will create a new workunit intended for a new master node.

After a master node has been established, and evidence of such has been received by BARK (see “Message Passing” below), work_generator is also responsible for creating workunits intended for new worker nodes. These workunits are passed relevant information, such as master’s URL, so that they may successfully connect to the correct master node.

2.2 Message Passing

Throughout intermediate steps in the BARK execution process, BARK uses trickle messages to communicate with the various volunteers it is attempting to coordinate a cluster over. Trickle messages are part of the standard BOINC framework. Each type of message requires an independent message handler daemon. BARK has three such message handlers: one for handling master node startup, one for handling worker nodes, and one for handling master node shutdown. For simplicity, the diagram below refers to just a single message_handler; the same simplification will be made in the following description.
Once the master node successfully creates a Spark cluster, it sends the URL for this cluster to BARK via trickle message. Then, `message_handler` uses this information to update the `spark_job` and `spark_node` tables. The former provides `work_generator` the necessary information to create worker workunits, and the latter allows BARK to maintain a record of the newly registered master node. After worker nodes have successfully connected to the master node, they also check in with a trickle message. This is primarily so that `message_handler` can update the `spark_node` table, but it also allows `message_handler` to send a message to the master node, commanding it to begin job execution, once all expected worker nodes have registered. Finally, when a job has finished executing, the master node notifies `message_handler`, which can relay a shutdown signal to all worker nodes.

### 2.3 Result Assimilation

The final major component of the BARK implementation is the means by which it handles results output by the master node. This output is written to an output file, which the master node uploads to the BARK EC2 instance. Then, a daemon called `spark_assimilator` (abbreviated to `assimilator`) is used to read the result and update the appropriate row in `spark_job` to include it. With this table updated, the Web UI should be able to display the result to the user.

### 2.4 Error Handling

Error handling in BARK is incomplete. That being said, the system is robust enough to handle unreachable worker nodes. Every job submission asks the user to specify a desired number of nodes over which to run their job. If not enough qualified volunteer nodes are found, or if one crashes before it can begin running Spark computations, BARK will continue polling the downed node and searching for a replacement for a set time limit (default 10 minutes). Once that limit has passed, it will command the master node to proceed with execution over the existing, smaller-than-expected cluster. If a worker node goes down mid-execution, Spark, as per its documentation, should be able to recover and continue executing over the remaining nodes.
3. Evaluation

BARK brings considerable ease of use and distributed computational power to the BOINC framework. But an analysis of its value to a BOINC developer and to the field of cluster computing more generally would be incomplete without a look at its envisioned use cases when compared with native BOINC and its impact on the robustness and security of the larger network of volunteers.

3.1 Volunteer Computing

When speaking of BARK, it’s easy to lose track of the potential scale at which these computations are being performed. In order to give an indication of the incredible aggregate power of a volunteer cluster, we briefly compare an example Spark cluster against a conventional EC2 instance provided by AWS when running a benchmark test, the factoring of a 512 bit RSA key. The backbone of many modern encryption schemes, RSA keys are easily created from two prime numbers but computationally difficult to factor to the component primes. To complete the factorization process in under 4 hours using EC2 would require an instance of 200 machines, specifically of the c4.8xlarge class in Amazon’s tiered system. To complete the same factorization process within the same time frame when using the average BOINC client (capable of an estimated 3 GFlops) would require a cluster of an estimated 7000 machines. Although this approximation makes large assumptions about everything from the overhead of very large Spark clusters, to the failure rate among volunteer nodes, to the level of utilization of nodes within the EC2 instance, the numbers are encouraging given that, with more than 3 million volunteers, BOINC projects routinely have clusters several orders of magnitude larger than this at their disposal. Furthermore, this calculation cost $75 dollars to run on Amazon’s hardware but was entirely free to run with the BOINC clients. And by exclusively using otherwise wasted resources for an exciting cause, the BOINC based computation has every right to be free. Although far from the streamlined and low latency cluster arrangements utilized by Amazon and the other cloud computing companies, this motley collection of volunteer nodes and their varied system architectures prove more than capable of handling the same type of computations normally reserved for a datacenter.

3.2 BARK

In the first case, our implementation of Spark over a volunteer cluster could be picked up in its current form as viable BOINC project. Supported by a trusted institution, the project could attract volunteers in the same way any other project might but, instead of running only highly specialized calculations written specifically for BOINC, could execute the more general Spark jobs. A single BOINC cluster could then be created for a large institution and Spark jobs from a variety of teams executed across it. No longer would massive organizations (like CERN or university departments) need to create, advertise, and maintain several independent clusters. Instead, a larger and more powerful single cluster could be shared and communally maintained with easily deployed Spark jobs determining the specific calculations being performed at any given time.

Alternatively, with the BARK project running over a volunteer network, Apache Spark jobs can be submitted and run with access only to the online user interface. In addition to the existing BOINC interface which displays information on the health of the volunteer cluster, this interface displays active nodes and allows for the submission of Spark jobs but does not
otherwise give access to the details of the BOINC cluster. In this arrangement, an organization looking to make use of a BARK managed volunteer cluster would not need to develop the network of dedicated volunteers usually necessary to leverage the power of BOINC but could instead simply collude with an existing project to make use of its resources. Opening the BOINC network to one-off scientific and academic computations would dramatically expand its envisioned use case. By allowing access to clusters without a dedicated BOINC project, both rollout strategies provide a measure of abstraction from the details of BOINC that can dramatically speed application deployment.

3.3 Security and Fault Tolerance

In largely bypassing the conventionally linear model of BOINC project development, followed by volunteer collection, and ending with data analysis, BARK revolutionizes the process of volunteer cluster computing but exposes significant but innate security issues with the BOINC framework. Because it allows largely unchecked code to execute on a volunteer machine, BOINC as an entity attempts to mitigate security risks but makes no guarantee of the performance of any of its constituent projects. Executing in a protected, account-based sandbox mode on a donated node, access to a volunteer’s personal files is forbidden. Options even exist for BOINC to run on a portable virtual machines that completely separate BOINC code from volunteer files. It is worth note that in its 14 years of existence, there have never been any native BOINC related security issues.

Although their files and system are secure, when it comes to the actual calculations being performed, volunteers are largely expected to trust in the intentions of the universities and academic organizations to which they donate their time. This guarantee is meaningful when the high overhead cost of project creation ensures that only legitimate institutions can sponsor and maintain projects. But when the calculations performed on the clustered machines become decoupled from the projects to which the volunteers have donated resources, this trust is no longer meaningful. How does a BOINC project owner decide what Spark jobs to approve to run on the cluster? How does a volunteer who dedicated resources to a certain CERN calculation ensure that their computer is used only by this organization and not simply used to execute Spark jobs for the highest bidder. BARK provides no functionality for vetting Spark code. In an effort to create as feature-rich an environment possible, this fact is not likely to change. Although it undermines the existing and largely trust based project-volunteer relationship, the convenience BARK offers a BOINC developer more than justifies a reimagining of the current security equilibrium.

The success of any execution across the BARK platform is also inexorably tied to the health of the cluster. Networks of volunteered personal resources are efficient and low cost but cannot possibly be described as reliable [8]. Already, BARK can handle the failure of worker nodes during the course of execution. This failure results in the work allocated to this node being returned to the master node unfinished until eventually being computed by another, still active volunteer. Volunteers can dynamically join the cluster as the job executes and receive tasks allocated by the master node so that as machines exit the cluster, new ones can continually enter. What BARK lacks in its current stage is the ability to handle permanent master node failures. Execution can be unexpectedly interrupted at the failure of this node, chosen at random from the list of available volunteers. This complete dependence on a single point of failure is an obligatory result of Spark’s cluster model but does not need to sound the death knell of the entire
execution. Apache offers several tools designed to improve the robustness of a cluster and the packaging and deployment of these tools represents an important step in the future development of BARK.

4. Future Directions

Next steps with BARK include the development of sleek client side downloadable that would connect volunteers seamlessly with the existing cluster. By ensuring that all connecting nodes have the requisite software, a decrease in the existing overhead for each calculation would be expected, as it is the delivery of Spark source code to uninitiated nodes during execution time that dominates the creation time of the cluster. Further development in this direction would likely provide functionality to customize this client side downloadable, or offer versions that also include popular cluster computing tools known to integrate well with Spark (like Kafka or Avro). In this way, the already considerable power of Spark could be further bolstered and easily maintained with the addition of external libraries.

A further optimization of the existing system would be the implementation of a Spark job optimizer. Each Spark job’s submit script allows an executor to specify minimum system requirements in worker nodes in an effort to balance node latency overhead with computational power. Because of the wide variety of underlying machines and systems on which this Spark job is being executed, it is likely that certain nodes will not meet these minimum requirements. This extremely simplistic approach to node creation does not take into account the viability of other nodes in the cluster and a more complex optimizer for Spark settings could easily mean the difference between a slow and infeasible calculation. Minimum processing power, RAM, and network settings can all be detected through BOINC and a tool that makes this information available to and optimally integrates it with Spark could constitute a valuable and technically challenging project in its own right.

Once sufficient volunteer resources have been accumulated, calculations executed, and data returned, a new problem emerges—how to validate the collected information. With calculations being performed at this scale, error checking results is already incredibly challenging. And this supposes that each node performs the calculations correctly. What happens when a volunteer node decides to play the vandal and intentionally return incorrect values? BOINC provides some simple result validation checks by redundantly computing certain aspects of the job before awarding credit [9]. However, should multiple nodes collude to produce incorrect results, BOINC’s native protections can easily be overwhelmed [10]. Although research exists on detecting and avoiding attacks of this nature, currently, dishonest return values are simply an expected danger when utilizing individually owned machines, both with basic BOINC framework and with BARK.

For the most effective utilization of the world’s otherwise wasted computing resources, we must look to extending our conception of their usefulness rather than simply extending BARK’s functionality. Already, it has been shown that volunteer clusters can readily match the computational capacity of EC2 instances. One could easily envision a future platform across which an individual can sell their unused CPU and storage to the highest bidding developer. One could even envision a cluster composed of a mixture of volunteer and purchased nodes created dynamically to match the requirements put forth by the current Spark job. Much as with creating an EC2 cluster, this developer could select the hardware, system properties, and number of machines to utilize, paying the owners of the computers for the privilege. Although boasting an
impressive number of users, BOINC’s network contains only a small fraction of the underutilized machines of the world and the incentive of payment would likely allow a massive expansion of the number of volunteer (although that term must now be used loosely) nodes available. A technically limited expansion of the existing products in the field of user-based cluster computing, the expansion into a monetized market is not only possible but highly useful.
5. References


