2015

Easy Distributed Grid Architecture for Research: Easy Access to Supercomputing

Brent Wilson

George Fox University, bwilson@georgefox.edu

EASY DISTRIBUTED GRID ARCHITECTURE FOR RESEARCH: EASY ACCESS TO SUPERCOMPUTING*

Brent Wilson
George Fox University
Newberg, OR 97132
bwilson@georgefox.edu

Michael Vandeberg
Virginia Tech University
Blacksburg, VA 24061
mvandeberg@vt.edu

ABSTRACT

Current distributed systems present many challenges for students who may not be very skilled at programming parallel applications for use on such systems. Grid computing is a cost effective means of providing supercomputing computation for both scientists and students of computing. Easy Distributed Grid Architecture for Research (EDGAR) is a grid computing solution that meets two critical constraints, namely ease of application programming for users and platform independence in implementation. Satisfying these two constraints makes EDGAR one of the only time and cost effective grid computing solutions. EDGAR creates an easy of use high performance solution for scientists and students in solving computationally intense problems. EDGAR provides easy access to supercomputing. The work presented in this paper represents a working implementation of a conceptual grid architecture presented in concept in 2009 [11].

KEYWORDS

grid computing, parallel programming, supercomputing, scientific problem solving

1. BARRIERS TO SUPERCOMPUTING

Distributed computing has been a field of study for many years and has led to the implementation of many different types of distributed systems, architectures, and frameworks. These systems tend to fall into three main areas: massively parallel super

computers, clusters of computing systems, or distributed grids. The foundational aspect of all of these systems continues to be the ability to distribute the processing of data across multiple CPUs. Each type of system has significant barriers that continue to prevent mainstream scientists and educational institutions from accessing and utilizing the computing resources offered by these types of parallel/distributed super computers [3,5].

The single super computer with hundreds to thousands of computing cores is not only cost prohibitive for the vast majority of the scientific community, it is also extremely difficult to program for individuals without parallel programming experience. While similar barriers are also in place for a cluster of computing systems, implementing a much smaller number of systems within the cluster, i.e. 5-10 computers, can significantly reduce the cost. The reduction of computers also reduces the computing power of the cluster and its effectiveness in problem solving large-scale problems. The difficulty of programming a computing cluster however is not reduced with the reduction of computing systems within the cluster. Computing clusters typically use similar parallel programming environments to that of a multi-core super computer. A distributed grid is a uniquely different type of system in which each computer (node) is a volunteer computer that may or may not be owned by the computing institution, thus significantly reducing the hardware cost. The development of an application for a distributed grid increases in complexity with the required use of a third-party grid architecture, such as BOINC [1]. Third-party grid architectures also require the use of C/C++, which again is not typically in the toolset for a traditional scientist.

The barriers to these distributed systems always come down to either significant hardware costs and/or the complexity of developing problem solutions utilizing these systems. The authors have developed a grid architecture that removes these barriers.

2. THE POWER OF GRID COMPUTING

A distributed grid is a client-server architecture made up of volunteer non-dedicated computing systems (clients) that donate their un-used CPU cycles along with a small amount of data storage to the collective grid. Each computing system on the grid is tied together with software called grid “middleware” or “framework”. The middleware is responsible for communications between the clients and the grid server. The grid server uses the middleware to send tasks to the clients and to receive the computational results from those same clients. The grid server is responsible for the decomposition of the main task into smaller sub-tasks that are distributed to individual client nodes for processing along with the aggregation of the clients results into a final result.

Distributed grid clients are typically owned by either the organization responsible for the computing, such as a university computer laboratory, or individual people, such as a student laptop. Every volunteer node within a desktop grid becomes responsible for their own resources: hardware, software, and maintenance. The cost and maintenance of the grid becomes distributed to the individual nodes. Another attractive feature of a desktop grid is the amount of un-used processing time available in the average desktop computer. The average desktop computer's CPU remains idle upwards of 95% of the time. Together these features provide cost effective computing power that when harnessed together becomes a massively parallel supercomputer [4,10].

Over the past fifteen years one particular desktop grid has become quite visible and had demonstrated the effectiveness of harvesting CPU cycles over the Internet. The SETI@home (Search for Extraterrestrial Intelligence) project, based out of University of California, Berkeley and has over 3 million volunteer clients world-wide and is self-credited with an average of 597 TFLOPS (Tera FLoating point Operations Per Second) as of November, 2012 [8]. Besides SETI@home there are many volunteer desktop grid applications worldwide that individuals can join and donate processing time and power. Another popular distributed computing project is Folding@home [6]. This project is a disease research project that simulates protein folding, computational drug design, and other types of molecular research. The most recent distributed computing project that has made the news is that of BitCoin mining [2]. For projects such as these that are substantially complex and computationally intensive problems, the desktop grid provides a mechanism for capturing the distributed processing power needed to begin solving such problems.

3. EDGAR GRID ARCHITECTURE

3.1 Motivation for EDGAR

The development of EDGAR was driven by the desire of the computer science department to support other scientific research on campus with substantial computing power for computationally intensive problems for which no 'viable' resources were available. The computer science department currently maintains a multi-node Beowulf-type cluster that can be used for computationally intensive problems. However, the cluster tends to sit idle due to the complexity of developing parallel applications using C/C++ and MPI (Message Passing Interface). This resource is not accessible for the scientific research community that could actually use the computational power due to the complexity of programming required.

The department has also had experience in developing a grid application using one of the standard architectures from University of California at Berkeley called BOINC (Berkeley Open Infrastructure for Network Computing). The BOINC grid is extremely complex and requires specialized clients to be developed for each client platform. The major downside to developing a BOINC grid is the amount of time needed to configure and bring up the grid along with a single application. Even the BOINC documentation at University of California at Berkeley states that when preparing to develop a grid application, “Budget at least a few person-months for getting a working system running” [9]. We were able to complete our BOINC grid project in just under 2-person months, however the same issue remained as we saw with the Beowulf cluster, scientists don't know how to parallel program in C/C++. Again, the computing power was available but for all practical purposes it was unusable by the intended scientific researchers on the faculty.

Out of frustration, came the idea… why not design and develop our own grid architecture? Even if our grid architecture was not able to reach the TFLOP benchmark of other architectures, any grid architecture that could be brought up in minutes would have a “few person-months” head start on traditional grid architectures. While the runtime might be slower, the overall time (conception to completion) would be faster for traditional scientific problems. As we began discussions of developing a grid architecture

that could be used by both students and faculty, it was driven by two primary design constraints that became foundational (and distinctive) to this new grid architecture based upon our past experiences: coding simplicity and platform independence.

As opposed to other distributed grid solutions, such as BOINC or MPI, the grid architecture EDGAR tackles distributed computing by divorcing the algorithm implementation from the parallelization implementation. While some speed may be lost due of this level of abstraction, EDGAR serves smaller to mid-sized projects that don't require a large implementation like BOINC or MPI, but could still directly benefit from a parallelized solution.

3.2 Implementation of EDGAR

EDGAR, at its core is a client server system using a traditional star topology. The server is responsible for breaking work into independent chunks for the clients to work on. To satisfy the two primary design constraints of ease of application programming and platform independent clients, the decision was made to separate the problem algorithm from the actual parallelization. This separation required the development of an application layer network protocol that would manage clients, distribute the problem script and work, and manage the assigned work. Due to the reliance of the grid protocol on other network protocols as well as a desire to allow EDGAR to run on heterogeneous systems, Java was chosen as the language to develop the EDGAR grid protocol. Java not only provides excellent network abstraction but also object serialization, which is the key feature within EDGAR. As for the ease of application development, it was important to accommodate a variety of programming skills given the research communities targeted for use of EDGAR, including CS1 students. The common programming skills of our research community were basic control structures used within a procedural programming paradigm. Given this common skill set, the decision was made to use Python as the scripting language for EDGAR applications. This provided the syntax simplicity needed along with a procedural paradigm comfortable for developers [11].

With the grid architecture written in Java and grid applications written in Python, this created a communication problem between the two languages. The solution was to utilize Jython [7] within the grid. Jython provides a Python interpreter that runs within the Java Virtual Machine and eliminates the need for a client installed Python interpreter. EDGAR uses Jython as a Java object, which allows EDGAR to evaluate Python code and interface natively between Python and Java data types. Jython allows EDGAR to bind Java variables directly to Python variables of the same name. This dynamic binding is very important for the language communications used within EDGAR. The ultimate success of EDGAR is actually a combination of the Python and Java dynamic bindings coupled with object serialization [12].

3.3 EDGAR's Object Serialization

Object serialization allows for the state of an object to be converted into a byte stream for either storage or transmission. EDGAR uses serialization to convert several key objects within the architecture's protocol for transmission to and from the client. Two of those objects are from the classes Chunk and Result. The Java objects Chunk

chunk and Result result represent a chunk of data to be worked on by a client and the result of that computation respectively. These objects when serialized are transmitted to the client and are then dynamically bound directly to Python variables chunk and result using Jython. The byte streams can then be converted back into a copy of the original Java objects with the original state as well within Python. Each client Python script now holds the Python variables chunk and result, which are now Python objects and can access all public methods and variables from within their class definitions. The Python objects upon completion are then passed back to the EDGAR server utilizing the reverse of this technique.

3.4 EDGAR's API

Driven by the design criteria of ease of programming, a simple Python script API (Listing 1) was developed as a common format for EDGAR applications. The API is capable of expressing a wide variety of algorithms. The basic premise of the API is that most every algorithm can be broken into three stages: Input generation, computation, and result aggregation. Different problems may combine these stages in different ways including repetition and or recursion, however, they can still be expressed in this manner. The EDGAR server parses the Python script, executes the code relevant to the server, and sends clients the portion of the script relevant to them.

```
#import python mods as needed
#Preamble - used for config
def generate():
   . . .
   return chunk

def aggregate(resultList):
   . . .
   return finalResult

def compute(chunk):
   . . .
   return result
#Additional definitions: classes, functions, etc.
```

Listing 1 - Python API for EDGAR projects

EDGAR follows the following call pattern for script execution:

1. Execute the preamble on the control node.
2. The control node calls generate until a null input is returned. Each call returns a single input value.
3. Each input is sent to a client where compute is called on the input.
4. Each result from compute is sent back to the control node where they are put in a list.
5. Once all results have been collected, the list is passed to aggregate.
6. If aggregate returns a value, the problem has been solved. Otherwise, null is returned and more input is generated. (Go back to 2)

4. SUMMARY

The development of EDGAR has provided an accessible supercomputing solution for scientific researchers needing the ability to solve computationally intensive problems. While the actual FLOPS of EDGAR will typically be less than that of a BOINC project, EDGAR is a better overall solution to that of BOINC for smaller to mid-sized projects. Research scientists along with introductory computer science students (CS1) with a relatively small knowledge of programming can develop and run EDGAR applications very quickly. Unlike its grid counterparts, EDGAR's applications are defined in the API script and not in the server configuration itself. This allows for an EDGAR server to run many applications from many domains without any server reconfiguration or development. Most grid architectures today must be reconfigured and actually redeveloped for each application, since the architecture is the application in other grid architectures. The total time that an EDGAR project takes from concept to implementation and run is directly proportional to the time it takes to write the Python script to model the computation. EDGAR's separation of algorithm from architecture allows supercomputing to accessible for traditional scientific communities within educational institutions. EDGAR is a cost effective and time effective way to provide high performance computing to everyone.

REFERENCES

[1] Anderson, D., BOINC: a system for public-resource computing and storage, *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID '04)*, 4-10, 2004.

[2] Bitcoin Project, Bitcoin - open source P2P money, http://bitcoin.org, 2014.

[3] Casanova, H., Distributed computing research issues in grid computing, *SIGACT News*, 33 (3), 50-70, 2002.

[4] Domingues, P., Marques, P., Silva, L., Resource usage of windows computer laboratories, *Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW '05)*, 469-476, 2005.

[5] Durrani, M., Shamsi, J., Volunteer computing: requirements, challenges, and solutions, Journal of Network and Computer Applications, 39, 369-380, 2014.

[6] Pand, V., Folding@home, http://folding.stanford.com, 2013.

[7] The Jython Project, Python for the Java Platform, http://www.jython.com, 2014.

[8] University of California, SETI@home, http://setiathome.ssl.berkeley.edu, 2014.

[9] University of California, Create a virtual supercomputing center, http://boinc.berkeley.edu/trac/wiki/VirtualCampusSupercomputerCenter, 2014.

[10] Watanabe, K., Fukushi, M., Kameyama, M., Adaptive group-based job scheduling for high performance and reliable volunteer computing, *Journal of Information Processing*, 19, 39-51, 2011.

[11] Wilson, B., Willshire, M., EDGAR: grid computing project, *Journal of Computing Sciences in Colleges*, 25 (2), 52-57, 2009.

[12] Wilson, B., *Distributed Shared Checkpointing In Enterprise Desktop Grids*, Doctoral dissertation, Colorado Springs, CO: Computer Science Department, Colorado Technical University, 2009.