Neptune: A Domain Specific Language for Deploying HPC Software on Cloud Platforms

UCSB Technical Report #2011-02
January, 2011

Chris Bunch Navraj Chohan
Chandra Krintz
Computer Science Department
University of California, Santa Barbara
{cgb, nchohan, ckrintz} @ cs.ucsb.edu

Khawaja Shams
Jet Propulsion Laboratory
California Institute of Technology
Khawaja.S.Shams@jpl.nasa.gov

ABSTRACT

In this paper, we present the design and implementation of Neptune, a domain specific language (DSL) that automates configuration and deployment of existing HPC software via cloud computing platforms. We integrate Neptune into a popular, open-source cloud platform, and extend the platform with support for user-level and automated placement of cloud services and HPC components. Such platform integration of Neptune facilitates hybrid-cloud application execution as well as portability across disparate cloud fabrics.

We evaluate Neptune using different applications that employ a wide range of popular HPC packages for their implementation including MPI, X10, MapReduce, DFSP, and dwSSA. In addition, we show how Neptune can be extended to support other HPC software and application domains.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Software Engineering - Language Classifications (Extensible Languages); C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks - Distributed Systems (Distributed Applications)

General Terms

Design, Languages, Performance

Keywords

Cloud Platform, Service Placement, Domain Specific Language

1. INTRODUCTION

Cloud computing is a service-oriented methodology that simplifies distributed computing through transparent and adaptive resource (compute, networking, storage) acquisition and management. With traditional systems, developers typically assume a static number of nodes and a fixed deployment style. Cloud computing enables developers to challenge these assumptions and to do so to build applications that can quickly acquire and release resources on-demand.

To date, public cloud providers largely have focused on delivering very low-cost, scalable web service support – at varying levels of abstraction. Amazon Web Services provides a scalable infrastructure from which users acquire access to individual and configurable virtual machines (VM) instances and to application-level services (e.g., persistent and block storage, key-value and relational databases, and queuing). As an alternative to fully customer self-service VM use, Amazon, Google, Microsoft, and others, offer complete runtime stacks (cloud platforms) that facilitate access to similar scalable services (storage, data management, queuing, messaging, etc.) through well-defined APIs. With platform cloud computing, developers implement and test their code locally against a non-scalable version of the platform and then upload their application to a proprietary implementation of platform (typically executing on the provider's resources) that implements scalable versions of the APIs, and that provides automatic scaling of the application front-end (web servers). Other cloud vendors, such as SalesForce, provide remote access to scalable implementations of complete applications, which can be customized by users.

Despite the abundance of offerings, there remain barriers to entry to the use of cloud systems for execution of HPC applications. Most significant is the challenge of configuration and deployment of libraries, services, and technologies that HPC applications employ for execution. Although not specific to cloud computing, this challenge is exacerbated by the cloud execution model since cloud fabrics are either fully customer self-service, or provide support that targets and is optimized for the web services domain. The former precludes the reuse of HPC application infrastructure (tools, services, packages, libraries, etc.) and requires non-trivial installation, configuration, and deployment effort to be repeated. This is in sharp contrast to the computational grid model in which the software infrastructure is configured and maintained by experts and *used* by developers.

Modern virtualization technology alleviates this problem to some degree: an expert developer can customize a single VM

and distribute it to others to automate and share the software configuration. However, three key challenges remain. First, most applications require services to be customized for a particular use or instance (e.g. via writing configuration files) and for service components to be started in the proper order. Incorrect configuration and startup ordering can lead to the inability to use the software at all or to poorly performing programs. The cloud-based configuration and deployment process for a wide variety of popular HPC software imposes a severe learning curve on scientists and tends to make HPC application execution using cloud fabrics inaccessible to all but expert users [14]. Moreover, complex configuration and deployment of HPC applications can prevent scientists from reproducing results, as well as from comparing to, reusing, and extending the work of others, slowing or stifling scientific advance.

The other two primary challenges to HPC application deployment using cloud resources are specific to the cloud computing model: (i) clouds are, by definition, opaque, and (ii) extant cloud systems implement a wide range of APIs (even for access to similar services) that differ across systems and that evolve as their implementers identify new ways of improving their offerings for users. Both of these characteristics add an additional level of complexity to the software stack that developers must master to deploy software efficiently and in a repeatable fashion. Differing APIs poses a greater threat to application developers since targeting a single API (cloud system) leads to lock-in – the inability to easily move from one cloud system to another.

The goal of our work is to address these challenges to enable users to develop HPC applications for execution over cloud systems in a more flexible and portable fashion. To enable this, we present Neptune, a domain-specific language that facilitates configuration and deployment of disparate cloud-based services for use by applications. Neptune provides a single interface and language through which developers configure the cloud resources and services for the execution of an HPC application.

We implement the Neptune language and runtime within the AppScale cloud platform – an open-source implementation of Google App Engine. We extend AppScale with new services that implement and export popular, general-purpose, HPC packages, including MPI, X10, MapReduce, and biological simulations via the Diffusive Finite State Projection algorithm (DFSP) [10] and doubly-weighted Stochastic Simulation Algorithm (dwSSA) [8]. Neptune is extensible in that user can add support for other HPC software packages in a straightforward manner. The Neptune runtime manages and controls these services.

In addition, we extend AppScale with support for dynamic placement of application components and platform services. This support can be employed manually by cloud administrators or automatically by the platform. The latter provides elasticity – the platform spawns only those nodes required by the cloud application and dynamically grow and shrink the resources according to program behavior (and/or other factors, e.g. cost). The platform also reuses virtual machines between computations to amortize their cost over multiple runs. This support enables developers to experiment with their own placement strategies. This ability to experiment is vital - many platforms assume a fully separated system, where each component is run on a dedicated machine, is the optimal system layout, while others assume that colocating components will improve performance. Neptune allows developers to measure and quantify the differences between layouts. Also unique to our work, we target placement within virtual machines, allowing developers to specify which machines run which services without requiring the specific knowledge of how to do so. Finally, since Neptune is integrated at the platform-level and we have ported the platform to different cloud infrastructures, our system facilitates application portability across cloud fabrics. We use this support to investigate placement of HPC applications across private-public and public-public cloud hybrids using popular cloud infrastructures (Eucalyptus and Amazon EC2).

In summary, we contribute:

- The design of a domain specific language that automates configuration and deployment of cloud services for HPC applications.
- An implementation of the language that integrates with an open-source cloud platform to provide support for MPI, X10, and MapReduce for general-purpose computation, and DFSP and dwSSA for computational science.
- Platform-agnostic techniques for user-specified placement of platform services while retaining flexibility for the platform to intelligently schedule placement of other services, including the ability for the cloud platform to reuse virtual machines between computation jobs.
- Hybrid cloud placement techniques that facilitate developer deployment of applications within homogeneous clouds (multi-clouds) (e.g., multi-availability zones within Amazon EC2) and heterogeneous (hybrid) clouds (e.g., a Eucalyptus cloud and an Amazon EC2 cloud).
- An experimental evaluation of the HPC software packages supported by Neptune as well as a cost analysis that harnesses the cloud platform's ability to reuse virtual machines between computation jobs.
- An investigation into what is required to make arbitrary cloud software scale within their respective platforms, including what aids and hampers scaling for cloud applications and a discussion of how to extend Neptune with support for other services and application domains.

In the sections that follow, we describe the design and implementation of Neptune and our extensions to the AppScale cloud platform. We then empirically evaluate our system using HPC applications and different placement strategies. We then present related work and conclude.

2. DESIGN

Neptune is a domain-specific language that gives cloud application developers the ability to easily configure and deploy various HPC software over cloud fabrics. Neptune operates at the cloud platform layer (runtime system level) so that it can control infrastructure-level entities (virtual machines) as well as application components and cloud services.

2.1 Syntax and Semantics

The Neptune language is a metaprogramming extension of the Ruby programming language. As such, it is high-level and familiar, and enables the Neptune runtime to leverage the large set of Ruby libraries with which it can interact with cloud infrastructures and platforms. Moreover, any legal Ruby code is also legal within Neptune programs, enabling users to use Ruby's scripting capabilities to quickly construct functioning programs. The reverse is also true: Neptune can be used within Ruby programs: to which it appears in the way that a library or API would appear to users of a particular programming language.

Neptune uses block objects (denoted throughout this work via the `do` and `end` keywords) to identify and communicate with services within a cloud platform. Legal Neptune code follows the syntax:

```
job ''service-name'' do
 @option-1 = ''setting-1''
 @option-2 = ''setting-2''
end
```

The semantics of the Neptune language are as follows: each valid Neptune program consists of one or more blocks, each of which indicate a job to run in a given cloud. The *service-name* marker indicates the name of the job to run (e.g., MPI, X10) and thus which parameters (prefixed by the `@` symbol) are necessary for the given block. This point is important to emphasize: Neptune jobs are extensible enough to enable each job to require a certain set of keywords be used. This design choice is intentional: not all jobs are created equal, and while some jobs require little information be passed to the runtime, other runtimes can benefit greatly from this added information. As a further step, we leverage Ruby's dynamic typing to enable the types of parameters to be constrained by the user: thus Neptune can dictate exactly what parameters are needed to optimize a job and what data types are required for each. If the user specifies a Neptune job but fails to provide the necessary parameters, the runtime informs them which parameters are required and aborts execution.

The value that the block returns is also extensible: in the cases where a HPC job is being initiated, a common pattern is used: the Ruby symbol (similar to that of a constant string in other programming languages) `:success` is returned when the job is successfully started, and the symbol `:failure` is returned in all other conditions. In the scenario where the block asks for the data access policy for a particular piece of data stored in the underlying cloud platform, the return value for the block is the data access policy itself (and `:failure` in scenarios where the ACL cannot be retrieved).

Finally, when the user wishes to retrieve data via a Neptune job, the block returns the location on the user's filesystem where the output can be found, and `:failure` if the output could not be retrieved. Work is in progress to expand the number of failure symbols to give users more information about why particular operations failed (e.g., if the data storage mechanism was unavailable or had failed, or if the cloud platform itself was unreachable in a reasonable amount of time), to enable Neptune programs written by users to become more robust and more adequately deal with failures at the cloud level. The typical format of a user's Neptune code is thus of the following form:

```
return_value = job ''mpi'' do
 @code = ''powermethod''
 @nodes_to_use = 4
end

if return_value == :success
 puts ''Your MPI job is now in progress.''
else
 puts ''Your MPI job failed to start.''
end
```

2.2 Design Choices

It is important to contrast the decision to design Neptune as a domain specific language with other configuration options that use XML or other markup languages [15]. These languages work well for configuration but, since they are not fully functional programming languages, they are bound to their particular execution model. In contrast, Neptune's strong binding to the Ruby programming language enables users to leverage Neptune and its HPC capabilities to easily incorporate it into their own codes. For example, Ruby is well known for its Rails web programming framework [22], and Neptune's interoperability enables Rails users to easily spawn HPC applications without explicit knowledge of how Neptune or the HPC application operates.

Markup and workflow languages are powerful in the types of computation that they enable. Neptune similarly allows arbitrary computation to be connected and chained to one another. The following example shows how the output of a MapReduce job can be used as the input to a X10 job. Here, the MapReduce job produces a graph representing links between web pages, while the X10 code takes this graph and performs a shortest-path algorithm from all nodes to one another:

```
job ''mapreduce'' do
 @input  = ''/rawdata/webdata''
 @output = ''/output/mrgraph''

 @mapreducejar = ''graph-generator.jar''
 @main = ''main''

 @nodes_to_use = 64
end

job ''mpi'' do
 @input = ''/output/mrgraph''
 @output = ''/output/shortestpath''

 @code = ''ShortestPath''
 @nodes_to_use = 64
end
```

To enable code reuse, we allow certain primitive operations to be used across HPC applications. In particular, setting access control policies (ACLs) for data produced by jobs and the ability to retrieve output from a job are two operations

that occur throughout all the HPC software Neptune supports. Thus, the Neptune runtime enables these operations to share a single code base for the implementation of these functions. This feature is optional: not all software packages may support ACLs and a unified model for data output, so Neptune gives developers the option to implement support for only the features they require, but with the ability to leverage existing support as well.

3. IMPLEMENTATION

To enable deployment of Neptune jobs, the cloud platform must support a number of primitive operations. These operations are similar to those found in computational grid and cluster utilities such as the Portable Batch System [19]. The cloud platform must be able to receive Neptune jobs, acquire computational resources to execute jobs on, run these jobs, and place the output of these jobs in a way that enables users to retrieve them later or share them with other users. Additionally, there must be mechanisms from the user's perspective that enable them to deploy jobs and modify the access permissions on data that has been uploaded to the cloud platform. For this work, we employ the AppScale cloud platform to add these capabilities.

AppScale is an open-source cloud platform that implements the Google App Engine APIs. Users deploy applications using AppScale via either a set of command-line tools or web interface, which then deploys and hosts the application. An AppScale cloud consists of one or more distributed database components, one or more web servers, a monitoring daemon (the AppController) that coordinates services on a single node as well as across nodes in the cloud, and a set of tools to configure, deploy, and manage a cloud. AppScale implements a wide range of distributed/replicated datastore technologies for its database interface (the Google Datastore API) via popular open source technologies, including Cassandra, Voldemort, HBase, Hypertable, MongoDB, MemcacheDB, and others. AppScale runs over virtualized and un-virtualized cluster resources as well as over Amazon EC2 and Eucalyptus [17] cloud infrastructures automatically. The full details of AppScale are described in [6, 3]. In this section, we overview the AppScale components (the AppScale Tools and AppController) that are impacted by our extensions and contributions that enable customized placement, automatic scaling, and Neptune support within AppScale.

3.1 Cloud Support

Our extensions to AppScale facilitate the interoperation of Neptune and AppScale. In particular, we modified AppScale to acquire and release machines used for computation and to enable service placement (statically or automatically). To do so, we modify two components within AppScale: the AppScale Tools and the AppController.

3.1.1 AppScale Tools

The AppScale Tools are a set of command line tools that developers and administrators can use to manage AppScale deployments and applications. In a typical deployment, the user writes a configuration file specifying which node in the system is the "master" node and which nodes are the "slave" nodes. Prior to this work, this meant that the master node always deployed a Database Master (or Database Peer for Peer-to-Peer databases) and AppLoadBalancer to handle and route incoming user requests, while slave nodes always deployed a Database Slave (or Database Peer) and AppServer hosting the user's application.

We extend this configuration model to enable users to specify a new configuration file that identifies which nodes in the system should run which components. For example, users can specify that they want to run each component on a dedicated machine by itself, or alternatively users could specify that they want their database nodes running on the same machines as their AppServers and have all other components running on another machine. Critically, we also allow users to designate certain nodes in the system as "open", which tells the AppController that no services are to run on this node and that it can be utilized for running Neptune jobs.

We extend this support to enable hybrid cloud deployment of AppScale – in which different nodes are managed by different cloud infrastructure over which AppScale runs. Here, users specify which nodes belong to each cloud, and then export a set of environment variables that correspond to the credentials needed for each cloud. This is done by design, to mirror the styles used by Amazon EC2 and Eucalyptus. One potential use case of this hybrid cloud support is for users who have a small, dedicated Eucalyptus deployment and access to Amazon EC2: these users could indicate to use their smaller Eucalyptus deployment to test and optimize the performance of their HPC codes and then deploy to Amazon EC2 when a larger number of nodes are needed. Similarly, Neptune users can use this hybrid cloud support to run jobs in multiple availability-zones simultaneously, enabling them to always run HPC jobs in the availability zone that is physically closest (and thus with the lowest latency) to their data. Furthermore, for scenarios when the application to be deployed is not an HPC application (e.g., in the case of web applications), it may be beneficial to ensure that instances of the application are served in as many availability zones as possible to ensure that the user has access to a nearby instance whenever possible. This deployment strategy enables web users much lower latencies to their applications and some degree of fault-tolerance in the rare cases when an entire availability zone is temporarily inaccessible.

3.1.2 AppController

The AppController is the AppScale monitoring service that runs on every node in the cloud. On each node, the AppController configures and instantiates all necessary services. This typically involves the starting of databases and running Google App Engine applications. AppControllers also monitor the status of the services they run, and periodically sends heartbeat messages to other AppControllers to learn the status of the components running there. Specifically, it queries each node to learn the CPU, memory, and hard drive usage, although it is extensible to other metrics as well.

We extend the AppController component to receive Neptune jobs from users. Our extensions enable the AppController to receive and understand RPC (via SOAP) messages from the Neptune runtime and to coordinate Neptune activities across other AppControllers (and thus other nodes) in the AppScale deployment. Furthermore, upon receiving a configuration file from the AppScale Tools designating that

certain machines are to be allocated for Neptune jobs, the AppController ensures that no other services run on those machines.

If running in hybrid cloud deployments, this also entails spawning machines in each cloud that the user has requested machines in, with the credentials that the user has provided. Additionally, as cloud infrastructures currently meter on a per-hour basis, we have modified the AppController to be cognizant of this and reuse virtual machines between Neptune jobs. Currently, any virtual machine that is not running a Neptune job at the 55-minute mark is terminated; all other machines are renewed for another hour.

Administrators can query the system via either the AppScale Tools and users via the web interface provided by the AppLoadBalancer. These interfaces inform users about the jobs in progress and in hybrid cloud deployments, which clouds are running which jobs.

3.2 Job Data

Clouds that run Neptune jobs must allow for data to be used as input to other Neptune jobs and thus we must store this data and make it available for later use. In Neptune, the output of a job is stored as a three-tuple: a string containing the job's identification number, a string containing the output of the job, and a composite type indicating the access policy of the given data. The access policy used for Neptune is similar to that of the access policy used by Amazon's Simple Storage Service [1]: a particular piece of data can be tagged as either private (only visible to the user that uploaded it) or public (visible to anyone). Data is by default private but can be changed by the user by running a Neptune job. Similarly, data is referenced as though it were on a file-system: paths must begin with a forward-slash ('/') and can be compartmentalized into folders in the familiar manner. Specifically, the Neptune program to set the ACL of a particular piece of data to be public is:

```
job ``set-acl'' do
 @output = ``/mydata/nqueens-output''
 @acl = ``public''
end
```

Just as a Neptune job can be used to set the ACL for a piece of data, a Neptune job is also used to retrieve the current ACL:

```
current_acl = job ``get-acl'' do
 @output = ``/mydata/nqueens-output''
end

puts ``The current ACL is: '' + current_acl
```

Retrieving the output of a given job works in the same fashion as it does for access policies: a Neptune job suffices. By default, it returns a string containing the results of the job, but as many jobs return data that is far too large to efficiently be used in this manner, a special parameter can be used to instead indicate that it should be copied to a file on the local machine. The following Neptune code illustrates both use cases (note that the # character is Ruby's comment character):

```
# for a job with small output
result = job ``get-output'' do
 @output = ``/mydata/boo''
end

puts ``Output is: '' + result

# for a job with much larger output
result = job ``get-output'' do
 @output = ``/mydata/boo-large''
 @save_to_local = ``/shared/boo-large.txt''
end

if result == :success
 puts ``Output copied successfully.''
end
```

3.3 Employing Neptune for HPC Frameworks

To expand the set of cloud services available from extant fabrics in support of HPC applications, we *service-ize* a number of key HPC packages within the platform for use via Neptune. Specifically, we provide support for MPI, X10, and MapReduce, to enable users to run arbitrary codes for different computational models within cloud systems. We also provide two popular stochastic simulation algorithms (the diffusive finite state projection algorithm and the doubly weighted SSA coupled with the cross-entropy method) which are used by a wide range of computational science applications.

3.3.1 MPI

The Message Passing Interface (MPI) is a popular, general-purpose computational framework for distributed scientific computing. The most popular implementation is written in a combination of C, C++, and assembly, however many other implementations exist for other programming languages, such as Java and Python. AppScale employs the C/C++ version, enabling developers to write code in either of these languages to access MPI bindings within an AppScale cloud. The developer uses Neptune to specify the location of the compiled application binary and output data. This information is communicated via Neptune to the AppScale Shadow. The Shadow then starts up NFS (or any distributed file system) on all the nodes in the system, if it has not yet been started. is required by MPI. The job is then run, and once it is completed, Neptune returns the standard output and standard error of the job (the results) to the developer. An example of such a specification is as follows:

```
job ``mpi'' do
 @code = ``powermethod''
 @nodes_to_use = 4
 @output = ``/mpioutput''
end
```

In this example, we designate the name of the output file to create from whatever is printed to standard output and the location of the compiled code. Note that this program does not use any input parameters nor need to write to any files on disk as part of its output, however, we can extend Neptune to do so if necessary. We also can designate which shared file system to use when running MPI. Currently, we support

NFS and are working on support for the Lustre Distributed File System [16]. The second code fragment blocks until the MPI job has finished running and copies the output of the job back to the user's machine.

3.3.2 X10

While MPI is suitable for many types of application domains, one recent demand in computing has been to enable programmers to write fast, scalable code using a high-level programming language. In addition, as many years of research have gone into optimizing virtual machine technologies, it was also desirable for a new technology to be able to leverage this work. In this spirit, IBM introduced the X10 programming language [5], which uses a Java-like syntax and executes transparently over a non-distributed Java backend and a distributed MPI backend.

As X10 code can compile to executables for use by MPI, X10 jobs are reducible to MPI jobs. Thus, the following Neptune code deploys an X10 executable that has been compiled for use with MPI:

```
job ''mpi'' do
 @code = ''NQueensDist''
 @nodes_to_use = 2
 @output = ''/x10output''
end
```

With the combination of MPI and X10 within Neptune, users can trivially write algorithms in all both frameworks and (provided a common output format exists) compare the results of a particular algorithm to ensure correctness across implementations. One example used in this paper is the $n-queens$ algorithm: the following Neptune code illustrates how to verify the results across these implementations:

```
# run mpi version
job ''mpi'' do
 @code = ''MpiQueen''
 @nodes_to_use = 4
 @output = ''/mpi/nqueens''
end

# run x10 version
job ''mpi'' do
 @code = ''NQueensDist''
 @nodes_to_use = 4
 @output = ''/x10/nqueens''
end

mpi_output = job ''output'' do
 @output = ''/mpi/nqueens''
end

x10_output = job ''output'' do
 @output = ''/x10/nqueens''
end

if mpi_output == x10_output
 puts ''Output matched!''
else
 puts ''Output did not match.''
end
```

3.3.3 MapReduce

Popularized by Google in 2004 for its internal data processing [9], the map-reduce programming paradigm (MapReduce) has experienced a resurgence and renewed interest recently. In contrast to the general-purpose message passing embodied in MPI, MapReduce targets embarrassingly parallel problems. Users provide input, which is split to a user-defined Map function. The output of this function is then sorted based on a key provided by the Map function and all the outputs with the same key are given to a user-defined Reduce function, which typically aggregates the data. As no communication can be done by the user in the Map and Reduce phases, these programs are highly amenable to parallelization.

Hadoop provides an open-source implementation that runs over the Hadoop Distributed File System (HDFS) [13]. The standard implementation requires users to write their code in the Java programming language, while the Hadoop Streaming implementation faciliates writing code in any programming language. Neptune has support for both implementations - users specify a Java archive file (JAR) for the standard implementation, or name where the Map and Reduce files are located on their computer.

Neptune copies the required files to AppScale. AppScale runs the job on the Neptune-specified nodes in the system. In particular, the AppScale Shadow contacts the Hadoop JobTracker node with this information, and polls for job completion (indicated by the output location having data). When this occurs, Neptune copies the data back to a user-specified location. From the user's perspective, the necessary Neptune code is:

```
job ''mapreduce'' do
 @input  = ''/input-text.txt''
 @output = ''/mroutput.txt''

 @mapreducejar = ''hadoop-examples.jar''
 @main = ''wordcount''

 @nodes_to_use = 4
end
```

Neptune exposes a number of interesting options specific to MapReduce to the user. As was the case with the MPI code, the user specifies where the input is located, where to write the output to (here this is an HDFS location), as well as the location of the code on their local machine. Users specify either a relative path or full path to their code. Since MapReduce inputs can be extremely large, we allow the user to specify an HDFS input location for the input file instead of copying it over the network on every run. This is useful if a previous Neptune run has already copied the input or if it was already placed there by another service. Similarly to the MPI implementation in Neptune, the second code block waits for the MapReduce job to finish and copies the results back to the user's machine.

3.3.4 SSA

One type of algorithm that computational biologists employ in their research is the Stochastic Simulation Algorithm (SSA), popularized by Gillespie [12]. At the highest level of abstraction, SSA is a Monte Carlo algorithm that simulates chemical or biochemical systems of reactions efficiently and accurately. As these simulations are non-deterministic, a large number of these simulations must be performed to achieve acceptable estimation accuracy. Statistical analysis (such as mean and variance) is then performed on the results of these simulations. Such computations are embarrassingly parallel and like MapReduce, can be characterized as Single Program Multiple Data. Two implementations of the SSA are the Diffusive Finite State Projection algorithm (DFSP) [10] and the doubly weighed SSA coupled with the cross-entropy method (which we hereafter refer to as simply dwSSA) [8], which offer both accurate and efficient simulation compared to other methods.

Currently, scientists at UCSB and elsewhere execute simulations in an ad-hoc manner. Scientists reserve a set of compute nodes and run a large number of simulations over these nodes, with at least 10,000 simulations needed to minimize error to acceptable levels for DFSP and 1,000,000 simulations for dwSSA. To simplify these processes, we implement cloud service support for both DFSP and dwSSA in AppScale and export Neptune support to end-users. With Neptune, scientists only need to specify the number of simulations they wish to run. Neptune then contacts the AppScale Shadow, which splits the work across it and the other nodes in the system. Neptune then merges the results and returns them to the user when completed. From the user's perspective, the code is:

```
job ''dfsp'' do
 @nodes_to_use = 4
 @simulations = 100_000
 @output = ''/dfspoutput''
end

job ''dwssa'' do
 @nodes_to_use = 4
 @simulations = 1_000_000
 @output = ''/dwssaoutput''
end
```

As our implementation uses the same input and code for all runs, the user need not specify such information. They only specify how many simulations to run and the name of the output. The second code block indicates where to copy the results to and blocks until the job finishes.

We have also seen use cases where scientists are interested in running a certain number of simulations and then, if the requested confidence level in their experiment level is not achieved, running more simulations. The Neptune code required to do this is trivial:

```
confidence_needed = 0.95
i = 0
loop {
 job ''dfsp'' do
  @nodes_to_use = 4
  @simulations = 100_000
  @output = ''/mydata/run-#{i}''
 end

 confidence_achieved = job ''get-output'' do
  @output = ''/mydata/run-#{i}''
 end

 if confidence_achieved > confidence_needed
  break
 else
  puts ''Sufficient confidence not reached.''
 end

 i += 1
}
```

3.4 Employing Neptune for Cloud Scaling and Enabling Hybrid Clouds

Our goal with Neptune is to simplify configuration and deployment of HPC applications. However, Neptune is flexible enough to be used with other application domains. Specifically, Neptune can be used to control the scaling and placement of services within the underlying cloud platform. Furthermore, if the platform supports hybrid cloud placement strategies, Neptune can also be used to control how services are placed across cloud infrastructures.

To demonstrate this, we use Neptune to enable users to manually scale up a running AppScale deployment. Users need only specify which component they wish to scale up (e.g., the load balancer, application server, or database server) and how many of them they required. This reduces the typically difficult problem of scaling up a cloud to the following code:

```
job ''appscale'' do
 @nodes_to_use = {:cloud1 => 3, :cloud2 => 6}
 @add_component = ''appengine''
 @time_needed_for = 3600
end
```

In this example, the user has specified that they wish to add six application servers to their AppScale deployment, and that these machines are needed for one hour. Furthermore, three of the servers should be placed in the first cloud that the platform is running over, while six servers should be placed in the second cloud that the platform is running over. This type of scaling is useful for instances where the amount of load in both clouds is known: here, this is useful if both clouds are over-provisioned but the second is either expecting greater traffic in the near future or is sustaining more load than the first cloud.

As was the case with other Neptune-enabled services, scaling and automation is only amenable to the same degree as the underlying services allow for. For example, while the Cassandra database does allow new nodes to be added to the system dynamically, users cannot add more nodes to the system than already exist (e.g., in a system with N nodes, no more than $N-1$ nodes can be added at a time) [4]. Therefore, if more than the allowed for number of nodes are needed, either multiple Neptune jobs must be

submitted or the cloud platform must absorb this complexity into its scaling mechanisms. Similarly, the underlying cloud platform must have the capability to manually add user-specified components: Neptune only directs these capabilities and does not implement them.

3.5 Limitations with Employing Neptune

As previously stated, Neptune enables automatic configuration and deployment of software by a supported cloud platform to the extent that the underlying software allows. It is thus important to make explicit scenarios in which Neptune encounters difficulties, as they are the same scenarios in which the supported software packages are not amenable to being placed in a cloud platform. From the end-users we have designed Neptune to aid, we have experienced three common problems that are not specific to Neptune or to distributed systems (e.g., clouds, grids) in general:

- Programs that require a unique identifier, whether it be an IP address or process name to be used to locate each machine in the computation (e.g. as is required by Erlang systems).
- Programs that are run on machines of a different architecture than the cloud supports, requiring either remote compilation or cross-compilation.
- Programs that have highly specialized libraries for end-users but are not free / open-source, and thus are currently difficult to dynamically acquire and release licenses for (e.g. Matlab).

We are investigating how to mitigate these limitations as part of future work. In particular, since Neptune is extensible in the parameters it supports, we can extend it to enable remote-compilation as-a-service. For unique identifiers, it is possible to have Neptune take a parameter containing a list of process identifiers to use within computation. For licensing issues, we can have the cloud fabric make licenses available on per-use basis. Our platform then can guide developers to cloud fabrics have the appropriate licenses for execution of their application.

4. EVALUATION

We next use Neptune to empirically evaluate how effectively the supported services execute within the AppScale cloud platform. We begin by presenting our experimental methodology and then discuss our results.

4.1 Methodology

To evaluate the software packages supported by Neptune, we use benchmark and example applications provided by each. We measure the cost of running Neptune jobs with and without VM reuse.

To evaluate MPI, we use a Power Method implementation that at its core multiplies a matrix by a vector (the standard `MatVec` operation) to find the absolute value of the largest eigenvalue of the matrix. We choose this code over more standard codes such as the Intel MPI Benchmarks as this code tests a number of the MPI primitives working in tandem, producing a code that should scale with respect to the number of nodes in the system. By contrast, the Intel MPI Benchmarks largely measure interprocess communication time or the time taken for a single primitive operation, which is likely to scale negatively as the number of nodes increase (e.g., barrier operations are likely to take longer when more nodes participate). We use a 6400x6400 matrix and 6400x1 vector to ensure that the size of the matrices evenly divides the number of nodes in the computation.

For X10, we use an NQueens implementation publicly available from the X10 team, that is optimized to run over multiple machines. It is designed to solve the n-queens problem, and to ensure a sufficient amount of computation is available, we set $n = 16$, thus creating a 16x16 chessboard and placing 16 queens on the board. For comparison purposes with MPI, we also include an optimized MPI version publicly made available be the authors of [21]. It is also set to use a 16x16 chessboard, using a single node to distribute work across machines and the others to perform the actual work involved.

To evaluate MapReduce, we use Java WordCount, which takes an input data set and finds the number of occurrences of each word in that set. Each Map task is assigned a line of the input text, and for every word it finds, it reports this with an associated count of one. Each Reduce task then sums the counts for each word and saves the result to the output file. Our input file consists of the works of William Shakespeare appended to itself 500 times, producing an input file roughly 2.5GB in size.

For our SSA codes, DFSP and dwSSA, we run 10,000 and 1,000,000 simulations, respectively, and measure the total execution time. As mentioned earlier, previous work in each of these papers indicate that these numbers of simulations are the minimum numbers of simulations that scientists typically must run to achieve a reasonable accuracy.

We execute these tests over different dynamic AppScale cloud deployments of 1, 4, 8, 16, 32, and 64 nodes. In all cases, each node is a Xen guestVM that executes with 1 virtual processor, 10GB of disk (maximum), and 1GB of memory. We also employ a placement strategy provided by AppScale where one node deploys an AppLoadBalancer (ALB) and Database Peer (DBP), while the other nodes are designated as "open" (that is, they can be claimed for any role by the AppController as needed). Since no Google App Engine applications are deployed, no AppServers run in the system. All values reported here represent the average of five runs.

For these experiments, Neptune employs AppScale 1.5-Beta, MPICH2 1.2.1p1, X10 2.1.0, Hadoop MapReduce 0.20.0, the DFSP implementation graciously made available by the authors of the DFSP paper [10], and the dwSSA implementation graciously made available by the authors of the dwSSA paper [8].

4.2 Experimental Results

We begin by discussing the performance of the MPI and X10 Power Method codes within Neptune. We time the computation (including any necessary communication required for the computation); we exclude the time to start NFS, to write

Table 1: Parallel efficiency for the Power Method code utilizing MPI over varying numbers of nodes.

# of Nodes	MPI Parallel Efficiency
4	0.9285
8	0.4776
16	0.3358
32	0.0488
64	0.0176

MPI configuration files, and to start prerequisite MPI services. Figure 1 presents these results. Table 1 presents the parallel efficiency, given by the standard formula:

$$E = \frac{T_1}{pT_p} \quad (1)$$

where E denotes the parallel efficiency, T_1 denotes the running time of the algorithm running on a single node, p denotes the number of processors used in the computation, and T_p denotes the running time of the algorithm running on p processors.

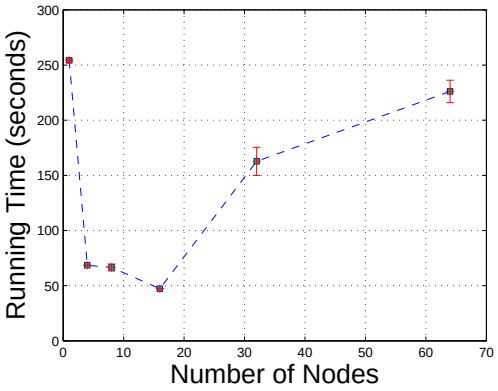

Figure 1: Average running time for the Power Method code utilizing MPI and X10 over varying numbers of nodes. These timings include running time as reported by MPI_Wtime and do not include NFS and MPI startup and shutdown times.

Both Figure 1 and Table 1 show clear trends: speedups are initially achieved as nodes are increased to the system, but the decreasing parallel efficiencies show that this scalability does not extend up through 64 nodes. Furthermore, the running time of the Power Method code increases after the 16 node point. Analysis using VAMPIR [23], a standard tool for MPI program visualization, shows that the collective broadcast calls used are the bottleneck, becoming increasingly so as the number of nodes increase in the system. This is an important point to reiterate: since Neptune simply runs supported codes of varying numbers of nodes, the original code's bottlenecks remain present and are not optimized away in any fashion.

The MPI and X10 n-queens codes encounter a different type of scaling compared to our Power Method code. Figure 2 shows these trends: the MPI code's performance is optimal at 4 nodes, while the X10's code performance is optimal at 16 nodes. The X10's n-queens code suffers substantially at the lower numbers of nodes compared to its MPI counterpart; this is likely due to its relatively new work-stealing algorithm, and is believed to be improved in subsequent versions of X10. This is also the rationale for the larger standard deviation encountered in the X10 code. We omit the discussion of parallel efficiency for this code: this is because the MPI code dedicates the first node to coordinate computation, and thus we cannot compute the time needed to run this code on a single node (required for computing parallel efficiencies).

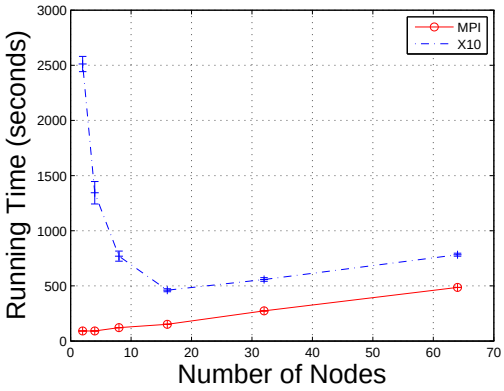

Figure 2: Average running time for the n-queens code utilizing MPI and X10 over varying numbers of nodes. These timings include running time as reported by MPI_Wtime and do not include NFS and MPI startup and shutdown times.

MapReduce WordCount experiences a superior scale-up compared to our MPI and X10 codes. This is largely because this MapReduce code is highly optimized by Hadoop and does not use any communication between nodes aside from that required for communication between the Map and Reduce phases. Figure 3 and Table 2 show the running times of WordCount via Neptune. Like with MPI, we measure computation time and not the time incurred starting and stopping Hadoop on the nodes involved.

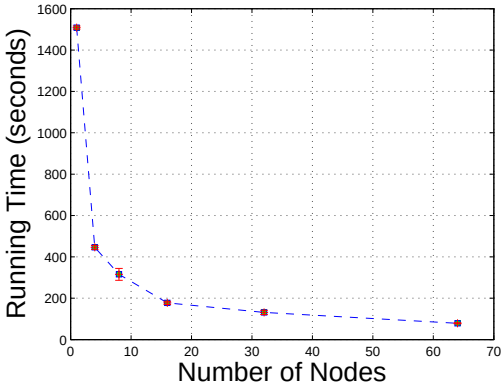

Figure 3: Average running time for WordCount utilizing MapReduce over varying numbers of nodes. These timings include Hadoop MapReduce runtimes and do not include Hadoop startup or shutdown times.

Figure 3 and Table 2 show opposing trends compared to the MPI results. With our MapReduce code, we see consistent speedups as more nodes are added to the system, although

Table 2: Parallel efficiency for WordCount using MapReduce over varying numbers of nodes.

# of Nodes	Parallel Efficiency
4	0.8455
8	0.5978
16	0.5313
32	0.3591
64	0.3000

the impact of this is diminished as we add more nodes to the system. This is clear from the decreasing parallel efficiencies, and as stated before, these speedups are not related to MapReduce or MPI specifically but are due to the programs evaluated here. WordCount sees a superior speedup compared to the Power Method code due to the reduced amount of communication and larger amounts of computation. We also see a much smaller set of standard deviations when compared with the Power Method MPI code, as now the communication is strictly handled by the runtime itself and optimized by the framework.

DFSP also benefits from parallelization and support via Neptune and a cloud platform. This is because the DFSP implementation used has no internode communication during its computation and is embarrassingly parallel. This is in contrast to the MapReduce framework, where communication occurs between the Map and Reduce phases. In the DFSP code, once each node knows how many simulations to run, they work with no communication with other nodes. Figure 4 and Table 3 show the running times for 10,000 simulations via Neptune. Unlike MapReduce and MPI, which provide distributed runtimes, our DFSP code does not, so we time all interactions once AppScale receives the message to begin computation from Neptune until the results have been merged on the master node.

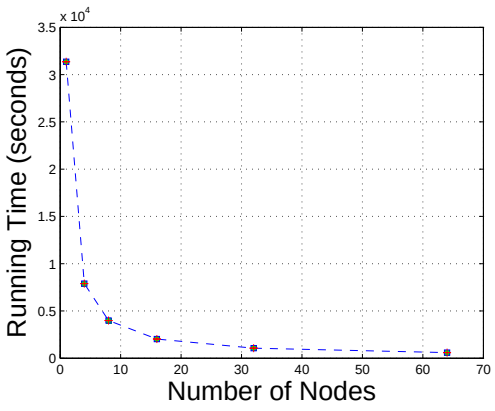

Figure 4: Average running time for the DFSP code over varying numbers of nodes. As the code used here does not have a distributed runtime, timings here include the time that AppScale takes to distribute work to each node and merge the final data.

Figure 4 and Table 3 show similar trends for the DFSP code as seen in MapReduce WordCount. This code also sees a consistent reduction in runtime as the number of nodes increase, but retains a much higher parallel efficiency compared to the MapReduce code. This is due to the lack of

Table 3: Parallel efficiency for the DFSP code over varying numbers of nodes.

# of Nodes	Parallel Efficiency
4	0.9929
8	0.9834
16	0.9650
32	0.9216
64	0.8325

Table 4: Parallel efficiency for the dwSSA code over varying numbers of nodes.

# of Nodes	Parallel Efficiency
4	0.7906
8	0.4739
16	0.3946
32	0.2951
64	0.1468

communication within computation, as now the framework needs only to collect results once the computation is complete, and does not need to sort or shuffle data as is needed in the MapReduce framework. As even less communication is used here compared to WordCount and Power Method MPI codes, the DFSP code exhibits a smaller standard deviation, and a standard deviation that tends to decrease with respect to the number of nodes in the system.

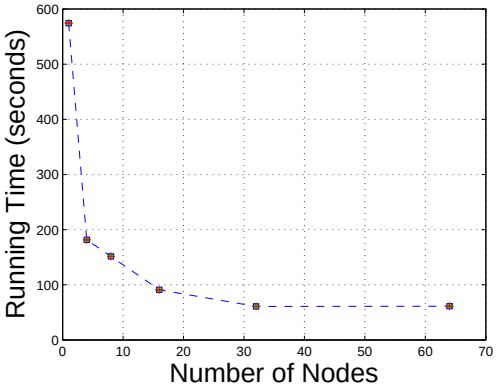

Figure 5: Average running time for the dwSSA code over varying numbers of nodes. As the code used here does not have a distributed runtime, timings here include the time that AppScale takes to distribute work to each node and merge the final data.

One final example that follows similar trends to the DFSP code is the other Stochastic State Algorithm, dwSSA, shown in Figure 5 and Table 4. This code achieves a reduction in runtime with respect to the number of nodes in the system, but does not do so at the same rate as the DFSP code, as can be seen through the lower parallel efficiencies.

4.3 VM Reuse Analysis

Next, we perform a brief examination of the costs of the experiments in the previous section if run over Amazon EC2, with and without the VM reuse. The VMs are configured with 1 virtual CPU, 1 GB of memory, and a 64-bit platform. This is similar to the Amazon EC2 "Small" machine type

Table 5: Cost to run all experiments for each type of Neptune job, with and without reusing virtual machines.

# Type of Job	Cost with VM Reuse	Cost without VM Reuse
PowerMethod	$12.84	$64.18
NQueens(MPI)	$12.92	$64.60
NQueens(X10)	$13.01	$64.60
MapReduce	$13.01	$64.18
DFSP	$35.70	$78.63
dwSSA	$12.84	$64.18
Total	$100.32	$400.37

(1 virtual CPU, 1.7 GB of memory, and a 32-bit platform) which costs $0.085 per hour.

Each PowerMethod, MapReduce, DFSP, and dwSSA experiment is run five times at 1, 4, 8, 16, 32, and 64 nodes to produce the data shown earlier, while each NQueens experiment is run five times at 2, 4, 8, 16, 32, and 64 nodes. We compute the cost of running these experiments without VM reuse (that is, by acquiring the needed number of machines, running the experiments, and then powering them off) compared to the cost with VM reuse (that is, by acquiring the needed number of machines, performing the experiment for all numbers of nodes, and not powering them off until all runs complete). Note that in the reuse case, we do not perform reuse between experiments. For example, the Neptune code used to run the experiments for the X10 NQueens code is:

```
[2, 4, 8, 16, 32, 64].each { |i|
 5.times { |j|
  job ``x10'' do
    @code = ``NQueensDist''
    @nodes_to_use = i
    @output = ``/nqueensx10/nodes#{i}/run#{j}''
  end
 }
}
```

Table 5 shows the expected cost of running these experiments with and without VM reuse. In all experiments, employing VM reuse greatly reduces the cost. This is largely due to inefficient use of nodes without reuse: many scenarios employ large numbers of nodes to run experiments that use only a subset of an hour (VMs are charged for by AWS by the hour).

5. RELATED WORK

The research most related to Neptune from a conceptual point-of-view are the RightScale Gems [20] and `boto` [2], sets of Ruby and Python libraries, respectively, that interact with cloud infrastructures. These libraries allow users to easily access Amazon services such as EC2, S3, and SQS in their Ruby and Python codes. Two major differences between these libraries and Neptune are the scope and extensibility: with the RightScale Gems and `boto`, users can perform the same functions they could with the Amazon-provided Java libraries in Ruby or Python instead. With Neptune, new types of computation are enabled through the included packages, and users can easily add support for their own as needed. This largely leverages the fact that the underlying platform is open-source; the RightScale Gems and `boto` are largely limited since the primary infrastructures they run over are closed-source (although in theory nothing prevents them from being extended with open-source cloud infrastructures such as Eucalyptus).

As MPI and MapReduce are frameworks that have seen widespread popularity in the distributed systems community, many have migrated them into cloud systems. Prior work [11, 18] explores the performance implications of running MPI and MapReduce at the infrastructure level. By doing so at the infrastructure level, this requires the user to bundle the software themselves and configure it for optimal use, which is a task that requires much more knowledge of the underlying system than the average user possesses. By doing so at the platform level, we eliminate these requirements - users simply specify the MPI or MapReduce code to run and the platform sets up the environment and optimizes it for them.

Other work, e.g. [7, 15] explore the use of specialized software packages in cloud systems. However, these works only support a single software package, while Neptune supports a variety of software packages and is extensible to support the addition of arbitrary software packages as administrators require. Neptune supports both software that is distributed by nature as well as those that are not–although scalability is limited to that intrinsically supported by the given software.

6. CONCLUSIONS

We contribute Neptune, a Domain Specific Language (DSL) that abstracts away the complexities of deploying and using high performance computing services within cloud platforms. We integrate support for Neptune into AppScale, an open-source cloud platform and add cloud software support for five disparate HPC software packages: MPI, X10, MapReduce, and the SSA packages DFSP and dwSSA. Neptune allows users to deploy supported software packages over varying numbers of nodes with minimal effort, simply, uniformly, and scalably.

We also contribute techniques for placement support of critical components within cloud platforms in a way that ensure that running cloud software does not negatively impact existing services. This also entails hybrid cloud placement techniques, facilitating deployment of applications by developers across cloud infrastructures without application modification. We implement these techniques within AppScale and provide sharing support that allows users to share the results of Neptune jobs between one another and to publish data to the scientific community via a data tagging system. The system is also flexible enough to allow users to reuse Neptune job outputs as inputs to other Neptune jobs. Neptune is open-source and can be downloaded from `http://neptune-lang.org`. Users with Ruby installed can also install Neptune directly via Ruby's integrated software repository by running `gem install neptune`.

7. REFERENCES

[1] Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3/`.

[2] Boto. `http://code.google.com/p/boto/`.

[3] C. Bunch, N. Chohan, C. Krintz, J. Chohan, J. Kupferman, P. Lakhina, Y. Li, and Y. Nomura. An Evaluation of Distributed Datastores Using the AppScale Cloud Platform. In *IEEE International Conference on Cloud Computing*, Jul. 2010.

[4] Cassandra Operations. `http://wiki.apache.org/cassandra/Operations`.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[6] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *ICST International Conference on Cloud Computing*, Oct. 2009.

[7] V. D. Cunsolo, S. Distefano, and A. Puliafito. Cloud@Home on top of RESERVOIR. In *ICST International Conference on Cloud Computing*, 2009.

[8] B. J. Daigle, M. K. Roh, D. T. Gillespie, and L. R. Petzold. Automated estimation of rare event probabilities in biochemical systems. *J. Phys. Chem.*, 2011.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of 6th Symposium on Operating System Design and Implementation(OSDI)*, pages 137–150, 2004.

[10] B. Drawert, M. J. Lawson, L. Petzold, and M. Khammash. The diffusive finite state projection algorithm for effficient simulation of the stochastic reaction-diffusion master equation. *J. Phys. Chem.*, 132(7), 2010.

[11] J. Ekanayake and G. Fox. High Performance Parallel Computing with Clouds and Cloud Technologies. In *ICST International Conference on Cloud Computing*, 2009.

[12] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.

[13] Hadoop Distributed File System. `http://hadoop.apache.org`.

[14] Engaging the Missing Middle. `http://www.hpcinthecloud.com/features/Engaging-the-Missing-Middle-in-HPC-95750644.html`.

[15] G. Koslovski, T. T. Huu, J. Montagnat, and P. Vicat-Blanc. Executing distributed applications on virtualized infrastructures specified with the VXDL language and managed by the HIPerNET framework. In *ICST International Conference on Cloud Computing*, 2009.

[16] Lustre. `http://www.lustre.org/`.

[17] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-source Cloud-computing System. In *IEEE International Symposium on Cluster Computing and the Grid*, 2009. `http://open.eucalyptus.com/documents/ccgrid2009.pdf`.

[18] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. In *ICST International Conference on Cloud Computing*, 2009.

[19] Pbspro home page – `http://www.altair.com/software/pbspro.htm`.

[20] RightScale. RightScale Gems. `http://rightaws.rubyforge.org/`.

[21] T. J. Rolfe. A Specimen MPI Application: N-Queens in Parallel. *inroads (bulletin of the ACM SIG on Computer Science Education)*, 40(4), 2008.

[22] Ruby on Rails. `http://www.rubyonrails.org`.

[23] W. E. Nagel and A. Arnold and M. Weber and H.-Ch. Hoppe and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, 1996.