J Grid Computing
DOI 10.1007/s10723-012-9213-8

Language and Runtime Support for Automatic Configuration and Deployment of Scientific Computing Software over Cloud Fabrics

Chris Bunch · Brian Drawert · Navraj Chohan · Chandra Krintz · Linda Petzold · Khawaja Shams

Received: 16 August 2011 / Accepted: 8 March 2012

Abstract In this paper, we present the design and implementation of Neptune, a simple, domain-specific language based on the Ruby programming language. Neptune automates the configuration and deployment of scientific software frameworks over disparate cloud computing systems. Neptune integrates support for MPI, MapReduce, UPC, X10, StochKit, and others. We implement Neptune as a software overlay for the AppScale cloud platform and extend AppScale with support for elasticity and hybrid execution for scientific computing applications. Neptune imposes no overhead on application execution, yet significantly simplifies the application deployment process, enables portability across cloud systems, and promotes lock-in avoidance by specific cloud vendors.

Keywords Cloud platform · Service placement · Domain specific language

C. Bunch (✉) · B. Drawert · N. Chohan · C. Krintz · L. Petzold
Computer Science Department,
University of California,
Santa Barbara, CA, USA
e-mail: cgb@cs.ucsb.edu

K. Shams
Jet Propulsion Laboratory,
California Institute of Technology,
Pasadena, CA, USA

Mathematics Subject Classifications (2010)
D.3.2 · C.2.4

1 Introduction

Cloud computing is a service-oriented methodology that simplifies distributed computing through dynamic resource (compute, storage, database, software) acquisition and management. Cloud computing differs from Grid computing in that resources are both shared and opaque. Specifically, users do not know about or control the geographic location, low-level organization, capability, sharing, or configuration of the physical resources they use. Moreover, these resources can grow and shrink dynamically according to service-level agreements, application behavior, and resource availability. Despite making vast resources procurable at very low cost and providing a scalable, effective execution model for a wide range of application domains, cloud computing has yet to achieve widespread use for scientific computing.

Beyond the differences between clouds and Grids, there are three barriers to the adoption of cloud computing for the execution of distributed, cluster-based, scientific applications. First, cloud systems currently in use have been designed for the execution of applications from the web services domain. As a result, developers must implement additional services and frameworks to

Published online: 30 March 2012

support applications from other domains. Such infrastructure (tools, services, packages, libraries) presents challenges to efficient reuse, and requires non-trivial installation, configuration, and deployment efforts to be repeatable. Second, cloud systems today are vastly diverse between one another, and code written for one system is not easily portable to other cloud systems, despite using common services and APIs provided by the cloud system. Differing interfaces can impose large learning curves and lead to lock-in – the inability to easily move from one cloud system to another. Third, the self-service nature of cloud infrastructures require significant user expertise to manipulate, control, and customize virtual machines (the execution unit of cloud infrastructures), making them inaccessible to all but expert users [15].

The goal of our work is to reduce the real-world impact of these barriers-to-entry and to facilitate greater use of cloud fabrics by the scientific computing community. This is also part of an effort to enable a cost-effective computation alternative to that of the cluster that is still viable for large scale scientific problems. Toward this end, we present and evaluate Neptune, a domain-specific language for automatically configuring and deploying disparate cloud-based services and applications. Neptune is a high-level language that is a superset of Ruby [31], a dynamic, open source programming language that is easy to learn and facilitates programmer productivity. Neptune adds to Ruby a series of keywords and constructs that developers use to describe a computational job at a very high level. Neptune executes any Ruby code using the Ruby interpreter and uses this job description along with a set of API calls to build, configure, and deploy the services, libraries, and virtual machines necessary for the distributed execution of complex scientific applications and systems over cloud platforms. Neptune abstracts away all of the low level details of the underlying cloud platforms (and by extension, cloud infrastructures) and provides a single, simple interface with which developers can deploy their applications. Neptune thus enables application portability across clouds and precludes lock-in to any single cloud vendor. Moreover, developers can use Neptune to employ multiple clouds concurrently (hybrid cloud computing), without application modification.

To enable this, Neptune interfaces to the AppScale [8, 9, 24] cloud platform. AppScale is a distributed, scalable software system that exposes a set of popular cloud service APIs (based on those of Google App Engine), and executes over the Amazon Web Services and Eucalyptus [27] clouds.

In this paper we present the design and implementation of Neptune, as well as a set of AppScale extensions that enable automatic configuration and deployment of scientific applications. These extensions include dynamic instantiation of virtual machines, placement of application and cloud service components within virtual machines for elasticity, and hybrid cloud computing. We extend AppScale with a set of popular software systems that are employed by a wide range of scientific application domains, such as MPI, UPC, and MapReduce for general-purpose HPC, as well as more science-specific toolkits such as StochKit [33] for stochastic biochemical simulation, DFSP [12] for spatial stochastic biochemical simulation, and dwSSA [10] for the estimation of rare event probabilities. Moreover, Neptune's design makes it straightforward for users to add additional frameworks, libraries, and toolkits.

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 Neptune using distributed HPC frameworks, stochastic simulation applications, and different placement strategies. We then present related work and conclude.

2 Neptune

Neptune is a domain-specific language that gives cloud application developers the ability to easily configure and deploy computational science software over cloud systems. Configuration refers to writing the configuration files that HPC software requires to execute in a distributed fashion, while deployment refers to starting HPC services in the correct order, to enable user code to be executed. 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 can leverage a large set of Ruby libraries to 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 to users as a library that can be utilized in the same fashion as other Ruby libraries.

Neptune uses a reserved keyword (denoted throughout this work via the `neptune` keyword) to identify services within a cloud platform. Legal Neptune code follows the syntax:

```
neptune :type => :service_name,
 :option1 => 'setting1',
 :option2 => 'setting2'
```

The semantics of the Neptune language are as follows: each valid Neptune program consists of one or more `neptune` invocations, each of which indicate a job to run in a cloud. The `service-name` marker indicates the name of the job (e.g., MPI, X10), and is associated with a set of parameters that are necessary for the given invocation. This design choice is intentional: not all jobs are created equal, and while some jobs require little information be passed, other job types can benefit greatly from increased information. As a further step, we leverage Ruby's dynamic typing to enable the types of parameters to be constrained by the developer. If the user specifies a Neptune job but fails to provide the necessary parameters, Neptune informs them which parameters are required and aborts execution.

The value that the invocation returns is also extensible, but by default, a Ruby hash is returned, whose items are job specific. In most cases, this hash contains a key named `:success` whose Boolean value corresponds to whether or not the request succeeded. Other scenarios allow for additional parameters to be included. For example, in the scenario where the invocation asks for the access policy for a particular piece of data stored in the underlying cloud platform, there is an additional key named `:acl` whose value is the current data access policy.

Finally, when the user wishes to retrieve data via a Neptune job, the invocation returns the location on the user's filesystem where the output can be found. Work is in progress to expand the number of failure messages 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 robust, and to adequately deal with failures at the cloud level. The typical format of a user's Neptune code is thus of the following form:

```
result = neptune :type => :mpi,
 :code => '/code/powermethod',
 :nodes_to_use => 4
if result[: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 [23]. These languages work well for configuration but, since they are not Turing-complete 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 [32], and Neptune's interoperability enables Rails users to easily spawn instances of scientific software without explicit knowledge of

how Neptune or the scientific computing software operates.

Markup and workflow languages are powerful in the types of computation that they enable. Similarly, Neptune 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. As Neptune does not automatically resolve data dependencies between jobs, we manually delay the execution of the X10 job until after the MapReduce job has completed and generated its output.

```
neptune :type => :mapreduce,
 :input  => '/rawdata/webdata',
 :output => '/output/mrgraph',

 :mapreducejar => '/code/graph.jar',
 :main => 'main',

 :nodes_to_use => 64

# wait for the mapreduce job to
finish loop {
 result = neptune
  :type => :get-output,
  :output => '/output/mrgraph'
 if result[:success]
   break
 end
 sleep(60)
}

neptune :type => :mpi,
 :input => '/output/mrgraph',
 :output => '/output/shortestpath',

 :code => '/code/ShortestPath',
 :nodes_to_use => 64
```

To enable code reuse, we allow operations to be reused across multiple Neptune job types. For example, retrieving data and setting ACLs on data are two operations that occur throughout all the job types that 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, and the ability to leverage existing support as needed.

3 Implementation

To enable the 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 [28]. The cloud platform must be able to receive Neptune jobs, acquire computational resources to execute jobs on, run these jobs asynchronously, and place the output of these jobs in a way that enables users to retrieve them later or share them with other users. 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 a web interface. An AppScale cloud consists of one or more distributed database components, web servers, and a monitoring daemon (the AppController) that coordinates services across nodes in the AppScale cloud. AppScale implements a wide range of datastores for its database interface via popular open source technologies. As of its most recent release (AppScale 1.5), it includes support for HBase, Hypertable, MySQL Cluster, Cassandra, Voldemort, MongoDB, MemcacheDB, Scalaris, and Amazon SimpleDB. AppScale runs over virtualized and un-virtualized cluster resources as well as over the Amazon EC2 and Eucalyptus cloud infrastructures. The full details of AppScale are described in [4, 9].

The execution of a Neptune job follows the pattern shown in Fig. 1. The user invokes the `neptune` executable on a Neptune script they have written, which results in a SOAP message being sent to the Neptune runtime (a separate thread in AppScale's AppController service). In the case of a compute job, the Neptune runtime

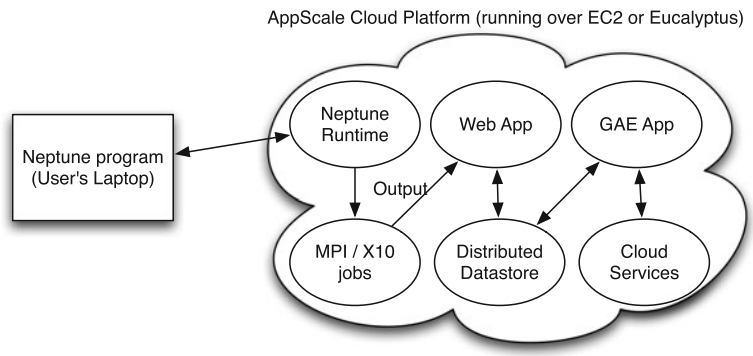


Fig. 1 AppScale cloud platform with Neptune configuration and deployment support

acquires nodes to run the code over, configures them for use, and executes the code, storing the output for later retrieval. In the case of a data input or output job, the Neptune runtime stores or retrieves the data via the datastore.

In this section, we overview the AppScale components that are impacted by our extensions enabling customized placement, automatic scaling, and Neptune support, the AppScale command-line tools and the AppController

3.1 Cloud Support

Our extensions to AppScale facilitate interoperation with Neptune. In particular, we modify AppScale to acquire and release machines used for computation, and to enable static and dynamic service placement. 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 AppServers hosting the user's application.

We extend this configuration model to enable users to provide a configuration file that identifies which nodes in the system should run each service (e.g., Database Master, Database Slave, AppLoadBalancer, AppServer). For example, users can specify that they want to run each service on a dedicated machine by itself. 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. We also allow users to designate certain nodes in the system as "open", which tells the AppController that this node is free to use for Neptune jobs (a hot spare).

We extend this support to enable hybrid cloud deployment of AppScale, in which nodes are not limited to a single cloud infrastructure. Here, users specify which nodes belong to each cloud infrastructure, and then export environment variables that correspond to the credentials needed for each cloud. This is done 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 use their Eucalyptus deployment to test and optimize their code, and deploy to Amazon EC2 when more nodes are needed. Similarly, Neptune users can use hybrid cloud support to run jobs in multiple availability zones simultaneously, providing them with the ability to run computation as close as possible to their data. For scenarios where the application to be deployed is not a compute-intensive application (e.g., 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 users always have access to a nearby instance. This deployment strategy gives users some degree of fault-tolerance, in the rare cases when an entire availability zone is down or temporarily inaccessible [18].

3.1.2 AppController

The AppController is a monitoring service that runs on every node in an AppScale deployment. It configures and instantiates all necessary services, which typically involves starting databases and running Google App Engine applications. AppControllers also monitor the status of each service it runs, and periodically send heartbeat messages to other AppControllers to aggregate this information. It currently queries each node to learn its CPU, memory, and hard drive usage, although it is extensible to collecting other metrics.

Our extensions enable the AppController to receive and understand RPC (via SOAP) messages from Neptune and to coordinate Neptune activities across other nodes in an AppScale deployment. Computational jobs and requests for output data run asynchronously within AppScale, and do not block the user's Neptune code. All Neptune requests are authenticated with a secret established when starting AppScale, and are performed over SSL to prevent request sniffing.

If running in hybrid cloud deployments, AppScale spawns machines for each cloud in which the user has requested machines, with the credentials that the user has provided. Any hot spares (machines indicated as "open") are acquired before new nodes are spawned. The AppController records which cloud each machine runs in, so that Neptune jobs can ask for nodes within specific cloud or more than one cloud. 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. Within AppScale, any virtual machine that is not running a Neptune job at the 55 min mark is terminated; all other machines are renewed for another hour.

Administrators query AppScale via either the AppScale Tools or the web interface provided by the AppLoadBalancer. These interfaces inform administrators about the jobs in progress and, in hybrid cloud deployments, which clouds are running which jobs. These interfaces do not actually run Neptune jobs or interact with them, but simply describe their status as reported to them by the AppController.

A perk of offering this service at the cloud platform layer is that the platform can profile the usage patterns of the underlying system and act accordingly (since a well-specified set of APIs are offered to users). We provide customizable scheduling mechanisms for scenarios when the user is unsure how many nodes are required to achieve optimal performance. This use case is unlikely to occur for highly tuned codes, but more likely to occur within HTC and MTC applications, where the code may not be as well tuned for high performance. Users only need specify how many nodes the application can run over, a required parameter because Neptune does not perform static analysis of the user's code, and oftentimes specific numbers of nodes are required (e.g., powers of two). Neptune then employs a hill-climbing algorithm to determine how many machines to acquire: given an initial guess, Neptune acquires that many machines and runs the user's job, recording the total execution time for later use. On subsequent job requests, Neptune tries the next highest number of nodes, and follows this strategy until the execution time fails to improve. Our initial release of Neptune provides scheduling based on total execution time, total cost incurred (e.g., acquire more nodes only if it costs less to do so), or a weighted average of the two. This behavior is customizable, and is open to experimentation via alternative schedulers.

More appropriate to scientists using cloud technologies is the ability to automatically choose the type of instance acquired for computation. Cloud infrastructure providers offer a wide variety of machines, referred to as "instance types", that differ in terms of cost and performance. Inexpensive instance types offer less compute power and memory, while more expensive instance types offer more compute power and memory. If the user does not specify an instance type to use, Neptune will automatically acquire a compute-intensive instance. A benefit of this strategy is that since these machines are among the more expensive machines available, the virtual machine

reuse techniques we employ amortize their cost between multiple users for jobs that do not run in 60 min increments (the billing quantum used in Amazon EC2).

3.1.3 AppServer

The AppServer is a modified version of the Google App Engine SDK that runs a user's App Engine application. Applications can be written in Python, Java, or Go, and can utilize APIs that provide a variety of features, including storage capabilities (via the Datastore and Blobstore) and communication capabilities (via Mail and XMPP).

For this work, we modify the AppServer to add an additional API: the Neptune API. This API allows users to initiate Neptune jobs from within App Engine applications hosted on AppScale, and thus provides a mechanism by which web applications can execute high performance computation. This also opens up HPC to greater audiences of users, including those who want to run their codes from different types of platforms (e.g., via their smartphone or tablet computer).

3.2 Job Data

Clouds that run Neptune jobs must allow for data stored remotely to be imported and used as job inputs. Jobs can consume zero or more files as inputs, but always produce exactly one piece of output, a string containing the standard out generated by the executed code. Neptune refers to data as three-tuple: a string containing the job's identification number, a string containing the output of the job, and a composite type indicating its access policy. The access policy used within 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, via 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. The data itself is accessed via a Google App Engine application that is automatically started when AppScale starts, and can be stored internally via AppScale or externally via Amazon S3. This allows jobs to automatically save their outputs in any datastore that AppScale supports, or any service that is API-compatible with Amazon S3 (e.g., Google Storage, Eucalyptus Walrus). The Neptune program to set the ACL of a particular piece of data to be public is:

```
neptune :type => 'set-acl',
 :output =>'/mydata/nqueens-output',
 :acl => 'public'
```

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

```
acl_data = neptune
 :type => 'get-acl',
 :output => '/mydata/nqueens-output'
puts 'The current ACL is:'
   + acl_data[:acl]
```

Retrieving the output of a given job is also done via a Neptune job. By default, it returns a string containing the results of the job. 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 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 = neptune
   :type => 'get-output',
 :output => '/mydata/boo'
puts 'Output is: ' + result[:output]
# for a job with much larger output
result = neptune
   :type => 'get-output',
 :output => '/mydata/boo-large',
 :save_to_local =>
   '/shared/boo-large.txt'
if result[:success]
 puts 'Output copied successfully.'
end
```

3.3 Employing Neptune for HPC Frameworks

To support HPC applications within cloud platforms, we *service-ize* them for use via Neptune.

Specifically, Neptune provides support for MPI, X10, MapReduce, UPC, and Erlang, to enable users to run arbitrary codes for different computational models. While these general purpose languages and frameworks are useful for the scientific community as a whole, Neptune also seeks to engender support from the biochemical simulation community. These groups of HPC perform simulations via kinetic Monte Carlo methods (specifically, the Stochastic Simulation Algorithm), and often need to run a large number of these simulations (on a minimum order of 10^5) to gain statistical accuracy. Neptune supports use of StochKit, a general purpose SSA implementation, as well as DFSP and dwSSA, two specialized SSA implementations.

As users may not have these libraries and runtimes installed locally, Neptune also provides the ability to remotely compile their code (required for the non-SSA computational models), and is extensible to support non-compute intensive application domains, such as web services.

3.3.1 MPI

The Message Passing Interface (MPI) [16] 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. Implementations exist for many other programming languages, such as Fortran, Java, and Python. AppScale employs the C/C++ version, enabling developers to write code in either of these languages to access MPI bindings within AppScale. The developer uses Neptune to specify the location of the compiled application binary and output data, and this information is sent from Neptune to the AppController.

Following the MPI execution model, one compute node is designated as a master node, and all other nodes are referred to as slave nodes. The master node starts up NFS on all nodes, mounts a shared filesystem on all slave nodes, runs `mpdboot` on its own node, and executes the user's code on its node via `mpiexec`, piping the output of the job to a file on its local filesystem. Once it has completed, the master node runs `mpdallexit` and stores the standard output and standard error of the job (the results) in the database that the user has requested, for later retrieval. An example of how a user would run an MPI job is as follows:

```
neptune :type => :mpi,
 :code => '/code/powermethod',
 :nodes_to_use => 4,
 :output => '/output/powermethod.txt'
```

In this example, we specify the location where the compiled code to execute is located (stored via a previous Neptune job). The user also indicates how many machines are required to run their MPI code and where the output of the job should be placed. Note that this program does not use any inputs, nor need to write to any files on disk as part of its output. Neptune can be extended 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 [25].

We also note that many HPC applications require a high performance, low latency interconnect. If running over Amazon EC2, users can acquire this via the Cluster Compute Instances they provided, and in Eucalyptus, a cloud can be physically constructed with the required network hardware. If the user does not have access to this type of hardware, and their program requires it, their program may suffer from degraded performance, or may not run at all.

3.3.2 X10

While MPI is suitable for many types of application domains, one 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 is also desirable for a new technology to be able to leverage this work. In this spirit, IBM introduced the X10 programming language [7], which uses a Java-like syntax, and can execute transparently over either a non-distributed Java backend or a distributed MPI backend. The Java backend enables developers to develop and test their code quickly, and utilize Java libraries, while the MPI

backend allows the code to be run over as many machines as the user can acquire.

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:

```
neptune :type => :mpi,
 :code => '/code/NQueensDist',
 :nodes_to_use => 2,
 :output => '/output/nqueensx10.txt'
```

With the combination of MPI and X10 within Neptune, users can trivially write algorithms in 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 [30], an algorithm that, given an chess board of size $n \times n$, determines how many ways n queens can be placed on the board without threatening one another. The following Neptune code illustrates how to verify the results produced by an MPI implementation against that of an X10 implementation (assuming both codes are already stored remotely):

```
# run mpi version
neptune :type => :mpi,
 :code => '/code/MpiNQueens',
 :nodes_to_use => 4,
 :output => '/mpi/nqueens'

# run x10 version
neptune :type => :mpi,
 :code => '/code/X10NQueens',
 :nodes_to_use => 4,
 :output => '/x10/nqueens'

# wait for mpi version to finish
loop {
 mpi_data = neptune
    :type => 'output',
  :output => '/mpi/nqueens'
 if mpi_data[:success]
  break
 end
 sleep(60)
}

# wait for x10 version to finish
loop {
 x10_data = neptune
     :type => 'output',
  :output => '/x10/nqueens'
 if x10_data[:success]
  break
 end
 sleep(60)
}

if mpi_data[:output]
   == x10_data[:output]
 puts 'Output matched!'
else
  puts 'Output did not match.'
end
```

Output jobs return a hash containing a `:success` parameter, indicating whether or not the output exists. We leverage this to determine when the compute job that generates this output has finished. The `:output` parameter in an `output` job contains a string corresponding to the standard out of the job itself, and we use Ruby's string comparison operator (==) to compare the outputs for equality.

3.3.3 MapReduce

Popularized by Google in 2004 for its internal data processing [11], the map-reduce programming paradigm (MapReduce) has experienced a resurgence and renewed interest. In contrast to the general-purpose message passing paradigm embodied in MPI, MapReduce targets embarrassingly parallel problems. Users provide input, which is split across multiple instances of a user-defined Map function. The output of this function is then sorted based on a key provided by the Map function, and all 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 of MapReduce that runs over the Hadoop Distributed File System (HDFS) [17]. 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 provide a Java archive file (JAR) for the standard implementation, or Map and Reduce applications for the Streaming implementation.

AppScale retrieves the user's files from the desired data storage location, and runs the job on the Neptune-specified nodes in the system. In particular, the AppController contacts the Hadoop JobTracker node with this information, and polls Hadoop until the job completes (indicated by the output location having data written to it). When this occurs, Neptune copies the data back to a user-specified location. From the user's perspective, the necessary Neptune code to run code written with the standard MapReduce implementation is:

```
neptune :type => :mapreduce,
 :input  => '/input/input-text.txt',
 :output => '/output/mr-output.txt',

 :mapreducejar =>
    '/code/example.jar',
 :main => 'wordcount',

 :nodes_to_use => 4
```

As was the case with MPI jobs, the user specifies where the input to the MapReduce job is located, where to write the output to, and where the code to execute is located. Users also specify how many nodes they want to run their code over. AppScale normally stores inputs and outputs in a datastore it supports or Amazon S3, but for MapReduce jobs, it also supports the Hadoop Distributed File System (HDFS). This can result in Neptune copying data to HDFS from S3 (and vice-versa), but an extra parameter can be used to indicate that the input already exists in HDFS, to skip this extra copy operation.

3.3.4 Unified Parallel C

Unified Parallel C [13] is a superset of the C programming language that aims to simplify HPC applications via the Partitioned Global Address Space (PGAS) programming model. UPC allows developers to write applications that use shared memory in lieu of the message passing model that other programming languages offer (e.g., MPI). UPC also can be deployed over a number of runtimes; some of these backends include specialized support for shared memory machines as well as optimized performance when specialized networking equipment is available. UPC programs deployed via Neptune can use any backend supported by the underlying cloud platform, and as we use AppScale in this work, three backends are available: the SMP backend, optimized for single node deployments, the UDP backend, for distributed deployments, and the MPI backend, which leverages the mature MPI runtime.

UPC code can be deployed in Neptune in a manner analogous to that of other programming languages. If a UPC backend is not specified in a `Makefile` with the user's code, the MPI backend is automatically selected. As we have compiled our code with the MPI backend, the Neptune code needed is identical to that used in MPI deployments:

```
result = neptune :type => :mpi,
 :code => '~/ring-compiled/Ring',
 :nodes_to_use => 4,
 :procs_to_use => 4,
 :output => '/upc/ring-output'

# inspect is Ruby's method to print
        a hash
puts result.inspect
```

As shown here, users need only specify the location of the executable, how many nodes to use, and where the output should be placed. We extend the MPI support that Neptune offers to enable users to specify how many processes should be spawned. This allows for deployments where the number of processes is greater than that of the number of available nodes (and are thus over-provisioned), and can take advantage of scenarios where the instance types requested have more than a single core present.

3.3.5 Erlang

Erlang [2] is a concurrent programming language developed by Ericsson that uses a message passing interface similar to that of MPI. While other HPC offerings try to engender a larger user community by basing their language's syntax, semantics, or runtime on that of C or Java (e.g., MPI, UPC, and X10), Erlang does not. The stated reason for this is that Erlang seeks to optimize the user's code via the single assignment model, which enables a higher degree of compile-time optimization than the model used by C-style languages.

While Erlang's concurrent programming constructs extend to distributed computing, Erlang does not provide a parallel job launcher analogous to those provided by MPI (via `mpiexec`), Hadoop MapReduce, X10, and UPC. These job launchers do not require the user to hardcode IP addresses in their code, as is required by Erlang programs.

Due to this limitation, we support only the concurrent programming model that Erlang offers. We are currently investigating ways to automate the process for the distributed version. Users write Erlang code with a `main` method, as is standard Erlang programming practice, and this method is then invoked by the AppScale cloud platform on a machine allocated for use with Erlang.

The Neptune code needed to deploy a piece of Erlang code is similar to that of the other supported languages:

```
neptune :type => :erlang,
 :code => '~/ring-compiled/
           ring.beam',
 :output => '/erlang-output.txt',
 :nodes_to_use => 1
```

In this example, we specify that we wish to use a single node, the path on the local filesystem where the compiled code can be found, and where the output of the execution should be placed.

3.3.6 Compilation Support

Before MPI, X10, MapReduce, UPC, or Erlang jobs can be run, they require the user's code to be compiled. Although the target architecture (the machines that AppScale runs over) may be the same as the architecture that the scientist has compiled their code on, it is not guaranteed to be so. It is therefore necessary to offer remote compilation support, so that no matter what platform the user runs, whether it be a 32-bit laptop, a 64-bit server, or even a tablet computer that has a text editor and internet connection, code can be compiled and run. The Neptune code required to compile a given piece of source code is:

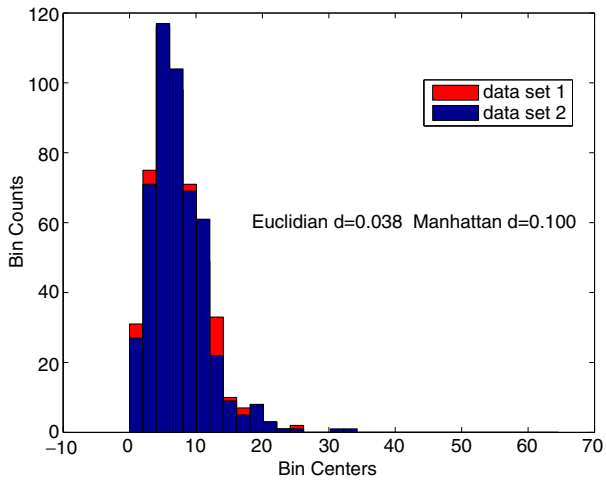
```
result = neptune :type => :compile,
 :code => '~/ring',
 :main => 'Ring.x10',
 :output => '/output/ring',
 :copyto => '~/ring-compiled'

puts result.inspect
```

This Neptune code requires the user to indicate only where their code is located and which code is the main executable (as opposed to being a library or other ancillary code). Scientists may provide `makefile`s if they like. If they do not, Neptune attempts to generate one for them based on the file's extension or its contents. Neptune cannot generate `makefile`s for all scenarios, but can do so for many scenarios where the user may not be comfortable with writing a `makefile`.

3.3.7 StochKit

To enable general purpose SSA programming support for scientists, Neptune provides support for StochKit, an open source biochemical simulation software package. StochKit provides stochastic solvers for several variants of the Stochastic Simulation Algorithm (SSA), and provides the mechanisms for the stochastic simulation of arbitrary models. Scientists describe their models by specifying them in the Systems Biology Markup Language (SBML) [20]. In this work, we simulate a model included with StochKit, known as `heat-shock-10x`. This model is a ten-fold expansion of the system that models the heat shock response in *Escherichia coli* [14]. Figure 2 shows results from a statistical analysis on an ensemble of simulated trajectories from this model.

Typically scientists utilizing the SSA run a large number of simulations to ensure enough statistical accuracy in their results. As the number of simula-



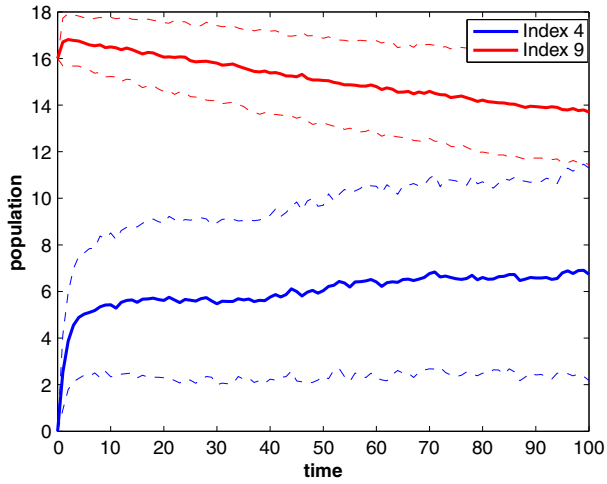


Fig. 2 Plots showing statistical results from StochKit stochastic simulations of the heat shock model. (*Left*) Comparison of probability density histograms from two independent ensembles of trajectories, as well as the histogram distance between them. The histogram self-distance, a measure of the difference between independent ensembles from the same model, is used to determine the confidence for a given ensemble size. (*Right*) Time-series plots of the mean (*solid lines*) and standard-deviation bounds (*dashed lines*) for two biochemical species

tions to run may not be known a priori, scientists often have to run a number of simulations, see if the requested confidence level has been achieved, and if this has not occurred, the process repeats. The Neptune code required to do this is trivial:

```
confidence_needed = 0.95
i = 0
loop {
 neptune :type => :ssa,
  :nodes_to_use = 4,
  :tar => '/code/ssa.tar.gz'
  :simulations = 100_000,
  :output = '/mydata/run-#{i}'

 # wait for ssa job to finish
 loop {
  ssa_data = neptune
       :type => 'get-output',
   :output => '/mydata/run-#{i}'
  if ssa_data[:success]
   break
  end
  sleep(60)
 }

 confidence_achieved
       = ssa_data[:output]
 if confidence_achieved
       >confidence_needed
  break
 else
  puts 'Sufficient confidence not
        reached.'
 end

 i += 1
}
```

To enable StochKit support within Neptune, we automatically install StochKit within newly created AppScale images by fetching it from a local repository. It is placed in a predetermined location on the image and made available to user-specified scripts via its standard executables. It is possible to require users to run a Neptune `:compile` job that would install StochKit in an on-demand fashion, but we elect to preinstall it, to reduce the number of steps required to run a StochKit job. Additionally, while forcing a compilation step is possible, the user's StochKit code often consists of biochemical models and a `bash` script, which do not need to be compiled to execute and thus do not fall under the domain of a `:compile` job.

As StochKit does not run in a distributed fashion, the AppController coordinates the machines that the user requests to run their SSA computa-

tion. For the example above, in which four nodes are to be used to run 100,000 simulations, Neptune instructs each node to run 25,000 simulations.

3.3.8 DFSP

One specialized SSA implementation supported by Neptune is the Diffusive Finite State Projection algorithm (DFSP) [12], a high-performance method for simulating spatially inhomogenous stochastic biochemical systems, such as those found inside living cells. The example system that we examine here is a biological model of yeast polarization, known as the G-protein cycle example, shown in [12]. Yeast cells break their spatial symmetry and polarize in response to an extra-cellular gradient of mating pheromones. The dynamics of the system are modeled using the stochastic reaction-diffusion master equation. Figure 3 shows visualizations from stochastic simulations of this model.

The code for the DFSP implementation is a tarball containing C language source and an accompanying `makefile`. The executable produces a single trajectory for each instance that is run. As this simulation is a stochastic system, an ensemble of independent trajectories are required for statistical analysis; 10,000 trajectories are needed to minimize error to acceptable levels. The Neptune code needed to run this is:

```
status = neptune :type => :ssa,
 :nodes_to_use => 64,

 :tar => '/code/dfsp.tar.gz',
 :output => '/outputs/ssa-output',

 :storage => 's3',
 :EC2_ACCESS_KEY =>
    ENV['S3_ACCESS_KEY'],
 :EC2_SECRET_KEY =>
    ENV['S3_SECRET_KEY'],
 :S3_URL => ENV['S3_URL'],

 :trajectories => 10_000,

puts status.inspect
```

In this example, the scientist has indicated that they wish to run their DFSP code, stored remotely at `/code/dfsp.tar.gz`, over 64 machines. The scientist here has also specified that their code should be retrieved from Amazon S3 with the provided credentials, and that the output should be saved back to Amazon S3. Finally, the scientist has indicated that 10,000 simulations should be run. The storage-specific parameters used here

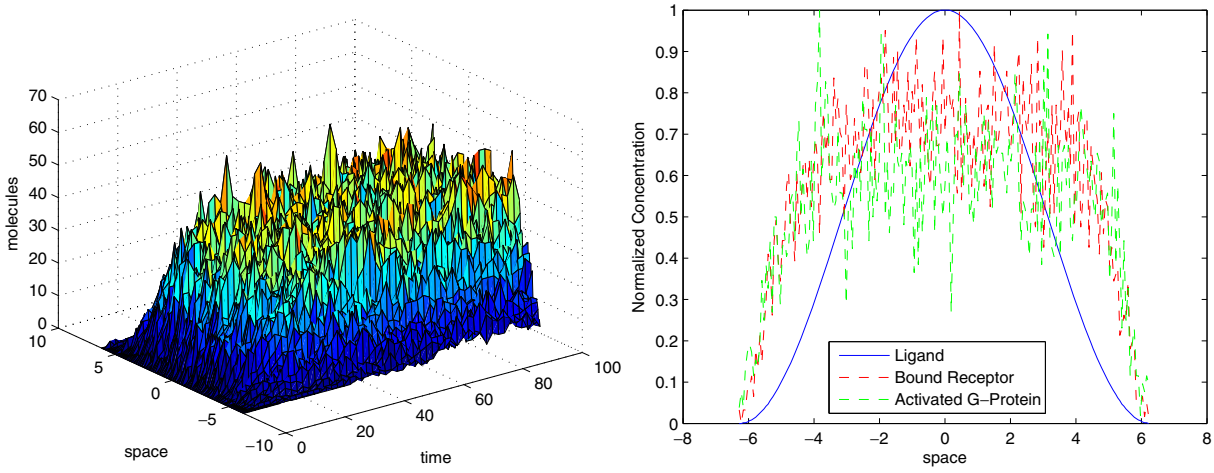


Fig. 3 Two plots of the DFSP example model of yeast polarization. (*Left*) Temporal-Spatial profile of activated G-protein. Stochastic simulation reproduces the noise in the protein population that is inherent to this system. (*Right*) Overlay of three biochemical species populations across the yeast cell membrane: the extra-cellular pheromone ligand, the ligand bound with membrane receptor, and the G-protein activated by a bound receptor

are not specific to DFSP or SSA jobs, and can be used with any type of computation.

To enable DFSP support within Neptune, we automatically install support for the GNU Scientific Library (GSL) when we generate a new AppScale image. The user's DFSP code can then utilize it within its computations in the same fashion as if it were installed on their local computer. Neptune does not currently provide a general model for determining library dependencies, as versioning of libraries can make this problem difficult to handle in an automated fashion. However, Neptune does allow an expert user to manually install the required libraries a single time and enable the community at large to benefit.

3.3.9 dwSSA

Another specialized SSA implementation we support within Neptune is the dwSSA, the doubly weighed SSA coupled with the cross-entropy method. The dwSSA is a method for accurate estimation of rare event probabilities in stochastic biochemical systems. Rare events, events with probabilities no larger than 10^{-9}, often have significant consequences to biological systems, yet estimating them tends to be computationally infeasible. The dwSSA accelerates the estimation of these rare events by significantly reducing the number of trajectories required. This is accomplished using importance sampling, which effectively biases the system toward the desired rare event, and reduces the number of trajectories simulated by several orders of magnitude.

The system we examine in this work is the birth-death process shown in [10]. The rare event that this model attempts to determine is the probability that the stochastic fluctuations of this system will double the population of the chemical species. The model requires the simulation of 1,000,000 trajectories to accurately characterize the rare event probability. The code for this example is a coupled set of source files written in R (for model definition and rare event calculations) and C (for efficient generation of stochastic trajectories). The Neptune code needed to run the dwSSA implementation is identical to that of DFSP and StochKit: users simply supply their own tarball with the dwSSA code in place of a different SSA implementation. As each dwSSA simulation takes a trivial amount of time to run, we customize it to take, as an input, the number of simulations to run. This minimizes the amount of time wasted setting up and tearing down the R environment.

To enable dwSSA support within Neptune, we automatically install support for the R programming language when we generate a new AppScale image. We also place the R executables in a predetermined location for use by AppScale and Neptune and use R's batch facilities to instruct R to never save the user's workspace (environment) between R executions, as is the default behavior.

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 control how services are placed. This allows Neptune to be used for both high throughput computing (HTC) and many task computing (MTC). In the former case, resources can be claimed from multiple cloud infrastructures to serve user jobs. In the latter case, Neptune can be used to serve both compute-intensive jobs as well as web service programs.

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 require. This reduces the typically difficult problem of scaling up a cloud to the following Neptune code:

```
neptune :type => :appscale,
 :nodes_to_use => {:cloud1 => 3,
                   :cloud2 => 6},
 :add_component => 'appengine',
 :time_needed_for => 3600
```

In this example, the user has specified that they wish to add nine 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. Defining which cloud is the "first cloud" and which cloud is the "second cloud" is done by the cloud administrator, via the AppScale Tools (see Section 4.1.1). This type of scaling is useful when 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.

Scaling and automation are only amenable to the same degree as the underlying services allow for. For example, while the Cassandra database allows 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) [6]. 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.

3.5 Limitations

Neptune enables automatic configuration and deployment of software by a 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:

- Codes 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., multi-node Erlang computations). This is distinct from the case where the framework requires IP addresses to be hardcoded, as these frameworks (like MPI) do not require the end-user's code to be modified in any way or be aware of a node's IP address.
- 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.
- Algorithms that require a high-speed interconnect that run in a cloud infrastructure that does not offer one. These algorithms may suffer from degraded performance or may not work correctly at all. The impact of this can be mitigated by choosing a cloud infrastructure that does provide such an offering (e.g., Cluster Compute Instances for Amazon EC2, or a Eucalyptus cloud with similar network hardware).

We are investigating how to mitigate these limitations as part of our future work. 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 a per-use basis. AppScale can then guide developers to clouds that have the appropriate licenses for their application.

3.6 Extensibility

Neptune is designed to be extensible, both in the types of job supported and the infrastructures that it can harness. Developers who wish to add support for a given software framework within Neptune need to modify the Neptune language component as well as the Neptune runtime within the cloud platform that receives Neptune job requests. In the Neptune language component, the developer needs to indicate which parameters users need to specify in their Neptune code (e.g., how input and output should be handled), and if any framework-specific parameters should be exposed to the user. At the cloud platform layer, the developer needs to add functionality that can understand the particulars of their Neptune job. This often translates into performing special requests based on the parameters present (or absent) in a Neptune job request. For example, MapReduce users can specify that the input be copied from the local file system to the Hadoop Distributed

File System. Our implementation within AppScale skips this step if the user indicates that the input is already present within HDFS. Once a single, expert developer has added support for a job type within Neptune and AppScale, it can then be automatically configured and deployed by the community at large, without requiring them to become an expert user.

4 Evaluation

We next use Neptune to empirically evaluate how effectively the supported services execute within AppScale. 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 benchmarks and sample applications provided by each. We also measure the cost of running Neptune jobs with and without VM reuse.

To evaluate our support for 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 because it 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, our evaluation uses an NQueens implementation publicly available from the X10 team that is optimized for multiple machines. 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 [30]. 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 our support for MapReduce, we use the publicly available Java WordCount benchmark, which takes an input data set and finds the number of occurrences of each word in that set. Each Map task is assigned a series of lines from 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.

We evaluate UPC and Erlang by means of a Thread Ring benchmark, and compare them to reference implementations in MPI and X10. Each code implements the same functionality: a fixed number of processes are spawned over a given number of nodes, and each thread is assigned a unique identifier. The first thread passes a message to the next thread, who then continues doing so until the last thread receives the message. The final thread sends the message to the first thread, connecting the threads in a ring-like fashion. This is repeated a given number of times to complete the program's execution.

In our first Thread Ring experiment, we fix the number of messages to be sent to 100 and fix the number of threads to spawn to 64. We vary the number of nodes to use between 1, 2, 4, 8, 16, 32, and 64 nodes, to determine the performance improvement that can be achieved by increasing the amount of available computation power.

In our second Thread Ring experiment, we fix the number of messages to be sent to 100, and fix the number of nodes to use at 8 nodes. We then vary the number of threads to spawn between 2, 4, 8, 16, 32, and 64 threads, to determine the impact of increasing the number of threads that must be scheduled on a fixed number of machines.

Our third Thread Ring experiment fixes the number of nodes to use at 8 nodes, and fixes the

number of threads to use at 64 threads. We then vary the number of messages to send between 1, 10, 100, 1000, and 10000, to determine the performance costs of increasing the number of messages that must be sent around the distributed thread ring in each implementation.

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 that scientists typically must run to achieve a reasonable accuracy.

We execute the non-Thread Ring experiments 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. The physical machines that we deploy VMs to execute with 8 processors, 1TB of disk, and 16GB of memory. We employ a placement strategy provided by AppScale where one node deploys an AppLoadBalancer (ALB) and Database Peer (DBP), and 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, MPICH2 1.2.1p1, X10 2.1.0, Hadoop MapReduce 0.20.0, UPC 2.12.1, Erlang R13B01, the DFSP implementation graciously made available by the authors of the DFSP paper [12], the dwSSA implementation graciously made available by the authors of the dwSSA paper [10], and the StochKit implementation publicly made available on the project's web site [35].

4.2 Experimental Results

We begin by discussing the performance of the MPI and X10 Power Method codes within Neptune. We time only the computation and any necessary communication required for the computation; thus, we exclude the time to start NFS, to write MPI configuration files, and to start pre-

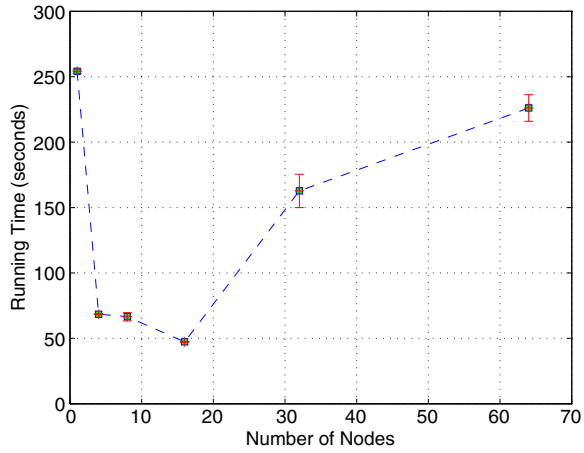


Fig. 4 Average running time for the Power Method code utilizing MPI over varying numbers of nodes. These timings include running time as reported by MPI_Wtime and do not include NFS and MPI startup and shutdown times

requisite MPI services. Figure 4 presents these results. Table 1 presents the parallel efficiency, given by the standard formula:

$$E = \frac{T_1}{pT_p} \tag{1}$$

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.

Both Fig. 4 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 using 16 nodes. Analysis using VAMPIR

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

[26], 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 on varying numbers of nodes, the original code's bottlenecks remain present and are not optimized away.

The MPI and X10 n-queens codes encounter a different type of scaling compared to our Power Method code. Figure 5 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 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 test. We omit parallel efficiencies for this code because the MPI code dedicates the first node to coordinate the computation, which precludes us from computing the time needed to run this code on a single node (a required value).

MapReduce WordCount experiences a superior scale-up compared to our MPI and X10 codes. This is largely because this MapReduce code is

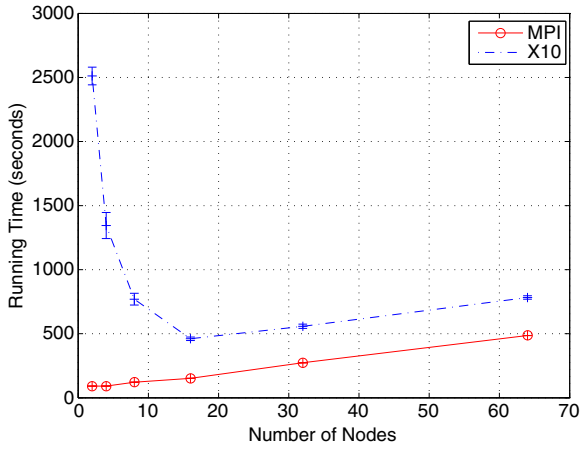


Fig. 5 Average running time for the n-queens code utilizing MPI and X10 over varying numbers of nodes. These timings include running time as reported by *MPI_Wtime* and *System.nanoTime*, respectively. These times do not include NFS and MPI startup and shutdown times

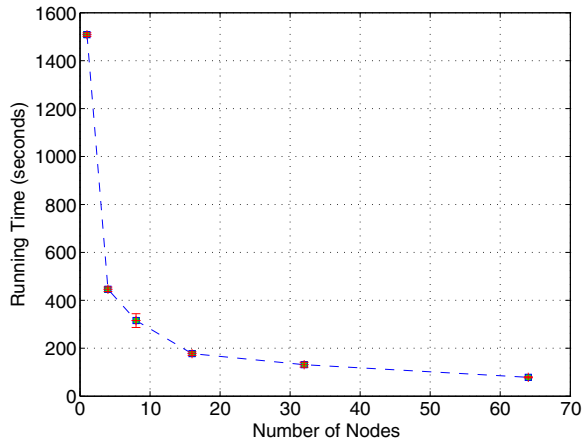


Fig. 6 Average running time for WordCount utilizing MapReduce over varying numbers of nodes. These timings include Hadoop MapReduce run times and do not include Hadoop startup or shutdown times

optimized by Hadoop and does not communicate between nodes, except between the Map and Reduce phases. Figure 6 and Table 2 show the running times of WordCount via Neptune. As with MPI, we measure computation time and not the time incurred starting and stopping Hadoop.

Figure 6 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 with a diminishing impact 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 smaller standard deviations when compared with the Power Method

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

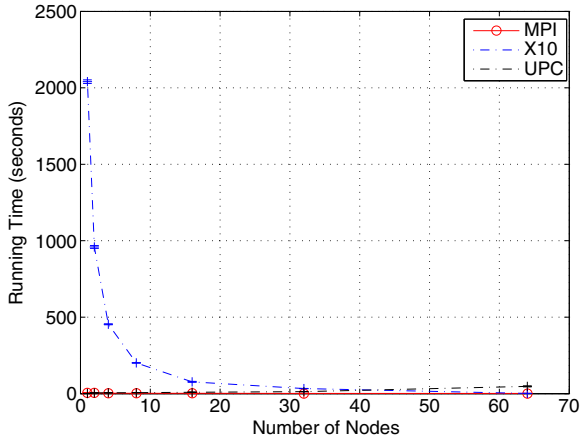


Fig. 7 Average running time for the Thread Ring code utilizing MPI, X10, and UPC over varying numbers of nodes. These timings only include execution times reported by each language's timing constructs

MPI code, as the communication is strictly dictated and optimized by the runtime itself.

In our first Thread Ring experiment, we measure time taken to send 100 messages through a ring of 64 threads. We vary the number of nodes used between 1, 2, 4, 8, 16, 32, and 64. Figure 7 shows the amount of time taken for implementations written in X10, MPI, and UPC, while Table 3 shows the parallel speedup achieved. Both the MPI and X10 codes improve in execution time as nodes are added. While the X10 code achieves a better parallel efficiency than the MPI code, it is on average one to three orders of magnitude slower. The reason behind this has been explained by the X10 team: the X10 runtime currently is not optimized to handle scenarios where the system is overprovisioned (e.g., when the number of processes exceeds the number of nodes). This is confirmed by the scenario in which 64 nodes are used: here, the system is not overprovisioned and runs in an equivalent amount of time as the MPI code. The UPC code exhibits a very different scaling pattern compared to the MPI and X10 codes: as it relies on synchronization via barrier statements, it runs quickly when the number of nodes is small, and becomes slower as the number of nodes increases.

Table 3 Parallel efficiency for the Thread Ring code utilizing MPI, X10, and UPC over varying numbers of nodes

# of nodes	MPI	X10	UPC
1	1.0000	1.0000	1.0000
2	0.5000	0.5000	0.5000
4	0.4518	1.1251	0.0014
8	0.3068	1.2663	5.6904e-04
16	0.1955	1.6518	2.0528e-04
32	0.1189	1.9190	7.0025e-05
64	0.0642	25.1618	9.8221e-06

Our second Thread Ring experiment fixes the number of nodes at the median value (8 nodes), and measures the amount of time needed to send 100 messages through thread rings of varying sizes. Here, we vary the sizes between 8, 16, 32, 64, and 128 threads. The results of this experiment for the X10, MPI, and UPC codes are shown in Fig. 8. As expected, all codes become slower as the size of the thread ring grows. The overall execution time is fastest for the MPI code, followed by that of the UPC and X10 codes. The reason for these differences is identical to that given previously: the UPC code relies on barriers. As the number of threads increases, it becomes more expensive to perform these barrier operations. The X10 code is also overprovisioned in most cases, so it slows down in these scenarios as well. In the scenario when it is not overprovisioned (e.g., when there are 8 threads and 8 nodes), the X10 code performs on par with the MPI code.

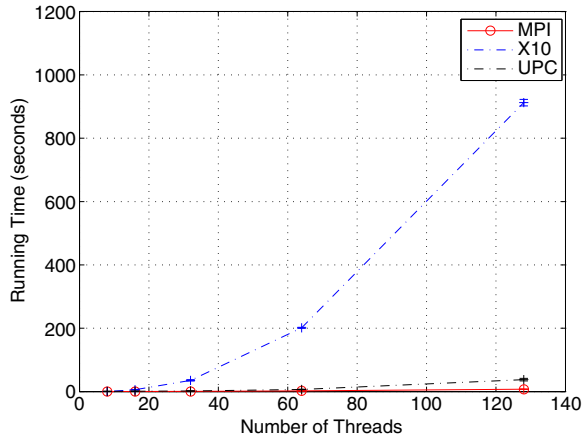


Fig. 8 Average running time for the Thread Ring code utilizing MPI, X10, and UPC over varying numbers of threads. These timings only include execution times as reported by each language's timing constructs

Our third Thread Ring experiment fixes the number of nodes at the median value (8 nodes) once again and measures the amount of time needed to send a varying number of messages through the thread ring. Specifically, we vary the number of messages to be sent between 1, 10, 100, 1000, and 10000 messages for the X10, MPI, and UPC codes. Figure 9 shows the results of this experiment: for all codes, excluding the single message scenario, the time to send additional messages increases linearly. Unlike the other benchmarks, the X10 and UPC codes perform within an order of magnitude of the MPI code. For the X10 code, this is because all machines are well-provisioned (specifically because we run 8 threads over 8 nodes), avoiding the performance degradation that the other experiments revealed. The UPC code also maintains relatively close performance to the MPI and X10 codes due to the low number of nodes: the barriers, which are the bottleneck of the UPC code, are inexpensive when a relatively small number of nodes are used.

To evaluate the performance of our Erlang code, we compare our Erlang Thread Ring implementation with that of our MPI code deployed over a single node. We fix the number of messages to send at 1000 and vary the number of threads that make up the ring between 4, 8, 16, 32, and 64. The results of this experiment are shown in Fig. 10. The Erlang code scales linearly, and performs two to three orders of magnitude faster than the MPI code for all numbers of threads tested. This is likely due to Erlang's long history as a concurrent language and lightweight threading model, which makes it highly amenable to this experiment. Similarly, MPI is designed to be a distributed programming language, and as we have seen in the other experiments, suffers performance degradations when overprovisioned. We are looking into support for Threaded MPI (TMPI) [34], which provides optimizations for the concurrent use case shown here.

We next analyze the performance of the specialized SSA packages supported by Neptune. This includes the specialized implementations present in DFSP and dwSSA, which here focus on the yeast polarization and birth-death models discussed previously.

Like the MapReduce code analyzed earlier, DFSP also benefits from parallelization and support via Neptune. This is because the DFSP implementation used has no internode communication during its computation, and is embarrassingly parallel. In the DFSP code, once each node knows how many simulations to run, they work with no communication from other nodes. Figure 11 and Table 4 show the running times for 10,000 simulations via Neptune. Unlike MapReduce and MPI,

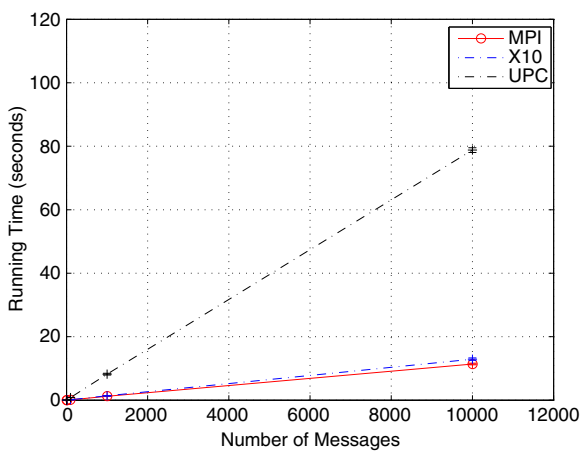


Fig. 9 Average running time for the Thread Ring code utilizing MPI, X10, and UPC over varying numbers of messages. These timings only include execution times as reported by each language's timing constructs

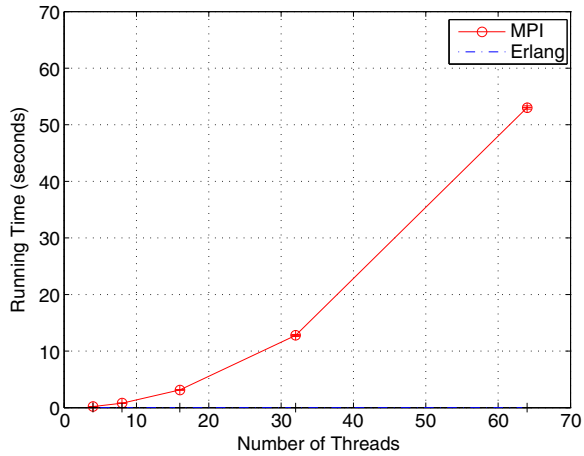


Fig. 10 Average running time for the single node Thread Ring code utilizing MPI and Erlang over varying numbers of threads. These timings only include execution times as reported by each language's timing constructs

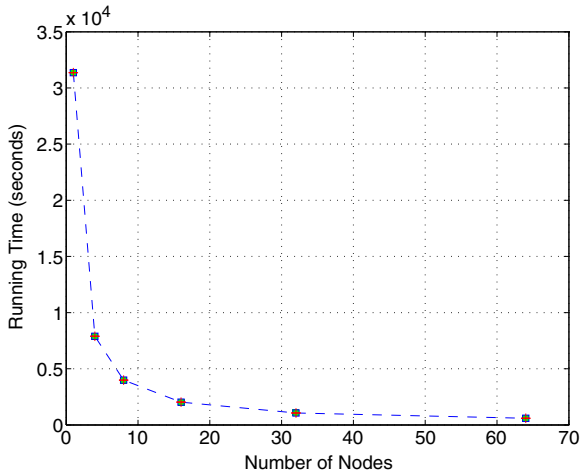


Fig. 11 Average running time for the DFSP code over varying numbers of nodes. As the code used here does not have a distributed runtime, timings here include the time that AppScale takes to distribute work to each node and merge the individual results

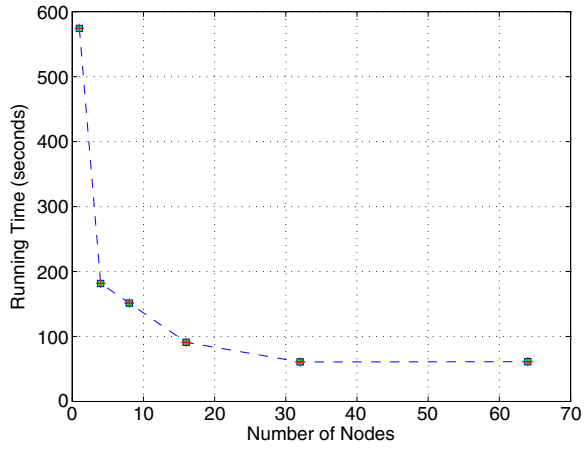


Fig. 12 Average running time for the dwSSA code over varying numbers of nodes. As the code used here does not have a distributed runtime, timings here include the time that AppScale takes to distribute work to each node and merge the individual results

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 11 and Table 4 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 communication within computation, as 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 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.

Another example that follows similar trends to the DFSP code is the other Stochastic State Algorithm, dwSSA, shown in Fig. 12 and Table 5. 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. This is because the execution time for a single dwSSA trajectory is much smaller than a single DFSP trajectory, which results in wasted time setting up and tearing down the R environment.

4.3 VM Reuse Analysis

Next, we perform a brief examination of the costs of the experiments in the previous section if run

Table 4 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 5 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

over Amazon EC2, with and without the VM reuse techniques described previously. 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 (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|
  neptune :type => :x10,
   :code => '/code/NQueensDist',
   :nodes_to_use => i,
   :output =>'/nqueensx10/#{i}/#{j}'
 }
}
```

Table 6 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, as many scenarios employ large numbers of nodes to run experiments that only run for a fraction of an hour (VMs are charged for by AWS by the hour). All of the experiments except for DFSP also cost roughly the same because they use similar numbers of CPU-hours of computation within AWS and thus are similarly priced. We see much greater variation in time and cost on a per-minute or per-second pricing model instead of a per-hour pricing model.

Table 6 Cost to run 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

5 Related Work

An early version of this work was presented at the Workshop on Scientific Cloud Computing (ScienceCloud) and was entitled "Neptune: A Domain Specific Language for Deploying HPC Software on Cloud Platforms" [5]. The work developed by others that is most similar to Neptune is `cloudinit.d` from Nimbus [22]. `cloudinit.d` provides an API that users employ to automatically launch, configure, and deploy nodes in a cloud infrastructure. In contrast to Neptune, `cloudinit.d`'s programming model places the onus of configuration and deployment on the user who writes `cloudinit.d` scripts. Neptune takes an alternate approach, hiding the complexity behind correct configuration and deployment.

Other works exist that provide either language support for cloud infrastructures or automated configuration or deployment, but not both. In the former category exist projects like SAGA [21], the RightScale Gems [29] and `boto` [3]. SAGA enables users to write programs in C++, Python, or Java that interact with Grid resources, with the recent addition of support for cloud infrastructure interaction. A key difference between SAGA and Neptune is that SAGA is conceptually designed to work with Grid resources, and thus the locus of control remains with the user. The programming paradigm embodied here serves use cases that favor a static number of nodes and an unchanging environment. Conversely, Neptune is designed to work over cloud resources, and can elastically add or remove resources based on the environment. The RightScale Gems and `boto` are similar to SAGA but only provide interaction with cloud infrastructures (e.g., Amazon EC2 and Eucalyptus).

In the latter category exist projects such as the Nimbus Context Broker [22] and Mesos [19]. The Nimbus Context Broker automates configuration and deployment of otherwise complex software packages in a matter similar to that of Neptune. It acquires a set of virtual machines from a supported cloud infrastructure and runs a given series of commands to unify them as the user's software requires. Conceptually, this is similar to what Neptune offers. However, it does not offer a language by which it can be operated, like Neptune and SAGA. Furthermore, the Nimbus Cloud Broker, like SAGA, does not make decisions dynamically based on the underlying environment. A set of machines could not be acquired, tested to ensure a low latency exists, and released within a script running on Nimbus Cloud Broker. Furthermore, it does not employ virtual machine reuse techniques such as those seen within Neptune. This would require a closer coupling with supported cloud infrastructures or the use of a middleware layer to coordinate VM scheduling, which would effectively present a cloud platform.

Like the Nimbus Context Broker, Mesos also automates configuration and deployment of complex software packages, but aims to do so for only a very specific set of packages (MapReduce, MPI, Torque, and Spark). It requires supported packages to be modified, and once they are "Mesos-aware", they can be utilized towards goals of better resource utilization for the cluster as a whole and better performance for individual jobs. Mesos also positions itself in the cluster computing space, in which jobs can dynamically scale up and down in the number of nodes that they use, but where the cluster as a whole must be statically partitioned. Cluster administrators can manually add or remove nodes, but the size of the cluster as a whole tends to remain static. This is in contrast to the cloud model employed by Neptune, where the number of nodes is in flux and is controllable by Neptune itself.

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 MPI, X10, MapReduce, UPC, Erlang, and the SSA packages StochKit, 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 components within cloud platforms, while ensuring that running cloud software does not negatively impact other services. This entails hybrid cloud placement techniques, facilitating application deployment across cloud infrastructures without modification. We implement these techniques within AppScale and provide sharing support that allows users to share the results of Neptune jobs, and to publish data to the scientific community. The system is 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`. Our modifications to AppScale have been committed back to the AppScale project and can be found at http://appscale.cs.ucsb.edu.

Acknowledgements We thank the anonymous reviewers for their insightful comments. This work was funded in part by Google, IBM, and NSF grants CNS-CAREER-0546737 and CNS-0905237.

References

1. Amazon Simple Storage Service (Amazon S3): http://aws.amazon.com/s3/. Last accessed 31 December 2011
2. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in ERLANG (1993)
3. Boto: http://code.google.com/p/boto/. Last accessed 31 December 2011
4. Bunch, C., Chohan, N., Krintz, C., Chohan, J., Kupferman, J., Lakhina, P., Li, Y., Nomura, Y.: An evaluation of distributed datastores using the AppScale Cloud Platform. In: IEEE International Conference on Cloud Computing (2010)
5. Bunch, C., Chohan, N., Krintz, C., Shams, K.: Neptune: a domain specific language for deploying HPC software

on cloud platforms. In: ACM Workshop on Scientific Cloud Computing (2011)
6. Cassandra Operations: http://wiki.apache.org/cassandra/Operations
7. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. **40**, 519–538 (2005)
8. Chohan, N., Bunch, C., Krintz, C., Nomura, Y.: Database-agnostic transaction support for cloud infrastructures. In: IEEE International Conference on Cloud Computing (2011)
9. Chohan, N., Bunch, C., Pang, S., Krintz, C., Mostafa, N., Soman, S., Wolski, R.: AppScale: scalable and open AppEngine application development and deployment. In: ICST International Conference on Cloud Computing (2009)
10. Daigle, B.J., Roh, M.K., Gillespie, D.T., Petzold, L.R.: Automated estimation of rare event probabilities in biochemical systems. J. Phys. Chem. **134**, 044110 (2011)
11. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of 6th Symposium on Operating System Design and Implementation (OSDI), pp. 137–150 (2004)
12. Drawert, B., Lawson, M.J., Petzold, L., Khammash, M.: The diffusive finite state projection algorithm for effficient simulation of the stochastic reaction-diffusion master equation. J. Phys. Chem. **132**(7), 074101-1 (2010)
13. El-Ghazawi, T., Cantonnet, F.: UPC performance and potential: a NPB experimental study. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing. Supercomputing '02, pp. 1–26. IEEE Computer Society Press, Los Alamitos, CA, USA (2002)
14. El-Samad, H., Kurata, H., Doyle, J.C., Gross, C.A., Khammash, M.: Surviving heat shock: control strategies for robustness and performance. Proc. Natl. Acad. Sci. USA **102**(8), 2736–2741 (2005)
15. Engaging the Missing Middle: http://www.hpcinthecloud.com/features/Engaging-the-Missing-Middle-in-HPC-95750644.html. Last accessed 31 December 2011
16. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Comput. **22**(6), 789–828 (1996)
17. Hadoop Distributed File System: http://hadoop.apache.org. Last accessed 31 December 2011
18. Heroku Learns from Amazon EC2 Outage: http://searchcloudcomputing.techtarget.com/news/1378426/Heroku-learns-from-Amazon-EC2-outage. Last accessed 31 December 2011
19. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A., Katz, R., Shenker, S., Stoica, I.: Mesos: a platform for fine-grained resource sharing in the data center. In: Networked Systems Design and Implementation (2011)
20. Hucka, M., Finney, A., Sauro, H.M., Bolouri, H., Doyle, J.C., Kitano, H., the rest of the SBML Forum, Arkin, A.P., Bornstein, B.J., Bray, D., Cornish-Bowden, A., Cuellar, A.A., Dronov, S., Gilles, E.D., Ginkel, M., Gor, V., Goryanin, I.I., Hedley, W.J., Hodgman, T.C., Hofmeyr, J.-H., Hunter, P.J., Juty, N.S., Kasberger, J.L., Kremling, A., Kummer, U., Le, N., NovÃĺre, Loew, L.M., Lucio, D., Mendes, P., Minch, E., Mjolsness, E.D., Nakayama, Y., Nelson, M.R., Nielsen, P.F., Sakurada, T., Schaff, J.C., Shapiro, B.E., Shimizu, T.S., Spence, H.D., Stelling, J., Takahashi, K., Tomita, M., Wagner, J., Wang, J.: The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. Bioinformatics **19**(4), 524–531 (2003)
21. Kaiser, H., Merzky, A., Hirmer, S., Allen, G., Seidel, E.: The SAGA C++ reference implementation: a milestone toward new high-level Grid applications. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06. ACM, New York, NY, USA (2006)
22. Keahey, K., Freeman, T.: Nimbus or an open source cloud platform or the best open source EC2 no money can buy. In: Supercomputing (2008)
23. Koslovski, G., Huu, T.T., Montagnat, J., Vicat-Blanc, P.: 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)
24. Krintz, C., Bunch, C., Chohan, N.: AppScale: Open-Source Platform-A s-A-Service. Technical Report 2011-01, University of California, Santa Barbara (2011)
25. Lustre: http://www.lustre.org/. Last accessed 31 December 2011
26. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.-Ch., Solchenbach, K.: VAMPIR: visualization and analysis of MPI resources. Supercomputer **12**, 69–80 (1996)
27. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The eucalyptus open-source cloud-computing system. In: IEEE International Symposium on Cluster Computing and the Grid. http://open.eucalyptus.com/documents/ccgrid2009.pdf (2009). Last accessed 31 December 2011
28. Pbspro home page: http://www.altair.com/software/pbspro.htm. Last accessed 31 December 2011
29. RightScale: RightScale Gems http://rightaws.rubyforge.org/. Last accessed 31 December 2011
30. Rolfe, T.J.: A specimen MPI application: N-Queens in parallel. Inroads (bulletin of the ACM SIG on Computer Science Education), vol. 40(4) (2008)
31. Ruby language: http://www.ruby-lang.org. Last accessed 31 December 2011
32. Ruby on Rails: http://www.rubyonrails.org. Last accessed 31 December 2011
33. Sanft, K.R., Wu, S., Roh, M., Fu, J., Lim, R.K., Petzold, L.R.: StochKit2: software for discrete stochastic simulation of biochemical systems with events. Bioinformatics (2011)
34. Shen, K., Tang, H., Yang, T.: Adaptive two-level thread Management for fast MPI execution on shared memory machines. In: Proceedings of ACM/IEEE SuperComputing '99 (1999)
35. StochKit: http://www.cs.ucsb.edu/cse/StochKit/. Last accessed 31 December 2011