Technical Report

TR-2010-013

Unibus-managed Execution of Scientific Applications on Aggregated Clouds

by

Jaroslaw Slawinski, Magdalena Slawinska, Vaidy Sunderam

Mathematics and Computer Science

EMORY UNIVERSITY

Unibus-managed Execution of Scientific Applications on Aggregated Clouds[0]

Jaroslaw Slawinski, Magdalena Slawinska, Vaidy Sunderam
Math and Computer Science, Emory University
Atlanta, GA 30322, USA
{jaross,magg,vss}@mathcs.emory.edu

***Abstract*—In this paper we examine the feasibility of running message passing applications across clouds. Our on-going Unibus project aims to provide a flexible approach to *provisioning* and *aggregation* of multifaceted resources and facilitate resource usage for both resource providers and end-users. In order to achieve that the Unibus approach explores: (1) virtualization of access to diverse resources via proposing the Capability Model and developing specific resource drivers to concrete resources, and (2) *soft* and *successive* conditioning to enable automatic and transparent to the user's resource provisioning.**

Cloud computing opens exciting research opportunities for Unibus in terms of *aggregation* challenges. This particular work briefly describes main Unibus components and concepts. We also demonstrate how to employ Unibus to create an aggregated set of computing chunks provided by two commercial cloud providers and how Unibus virtualizes access to those resources. We also present how Unibus simplifies usage of the aggregated computing platform on the example of the execution of NAS Parallel Benchmarks (NPB). We show and discuss the performance results obtained from NPB and the mpptest benchmark.

I. Introduction

The vigorous and rapid growth of the cloud industry creates new computing opportunities that change the character of computing for business enterprises, academia and individuals. Aside from traditional computational centers, clusters, Grids, local desktops, etc, computational power can be delivered on-demand to everyone who can afford it. The increased availability and affordability of computing opportunities [2] encourage to explore the feasibility of aggregation of various computing resources available to end-users. Whereas executing applications on aggregated resources may not be always as efficient as intended due to, for instance, resource heterogeneity at different levels (hardware, software), network issues (bandwidth, throughput, latency, etc), required Quality of Service, there are many situations in which the capability of aggregating resources brings profits (e.g., Grids). This, however, becomes challenging taking into account resource heterogeneity, proprietary authentication/authorization policies, security and cross-resource network issues, resource provisioning (build, software deployment, etc), the configuration of execution environments, to name a few issues that need to be addressed by resource aggregation.

Our on-going research project, titled Unibus, aims to simplify running applications on heterogeneous resources. In particular, it focuses on exploring new methods for resource provisioning and aggregation in an automatic and transparent to end-users fashion. We approach that by *resource access virtualization* that provides necessary abstractions which allow for access and use of multifarious resources in a unified manner. Resource access virtualization is based on (1) the *Unibus Capability Model* that models resource capabilities and delivers respective resource abstractions and (2) *mediators* that act as resource drivers for abstractions defined in the Capability Model. Resource access virtualization allows to automate provisioning and resource aggregation, and allows to handle intricacies of resource provisioning and aggregation by Unibus.

In this work, we focus on two goals: (1) we practically examine the Unibus approach to aggregation of heterogeneous cloud resources and execution of scientific applications on initially unconditioned resources, and (2) we experimentally assess the viability of executing message passing applications across clouds. To achieve those goals we selected NAS Parallel Benchmarks (NPB) [3] as a representative set of scientific applications. Although this choice might be debatable (e.g., NPB were designed to evaluate homogeneous systems), yet (1) NPB allow to demonstrate how the Unibus automates various aspects of resource provisioning and aggregation as NPB require

[0]This is the extended version of the CCGRID'10 paper [1]. Research, publication, and presentation supported in part by DOE grant DE-FG02-06ER25729, and NSF grant CNS-0720761

the dependency-related build effort (Fortran and C compilers, MPI), and (2) obtained results can still provide a decent quantitative insight into performance of both *individual* clouds (which are homogeneous) as well as *aggregated* cloud resources (NPB implement different degrees of intensity of interprocessor communication).

II. Related Work

Enabling resources to execute software applications requires assuring appropriate execution environments on computing chunks by deploying relevant dependencies and performing necessary configuration. Several research projects (e.g., ReST [4], NetBuild [5], Modules [6], our companion project Harness Workbench Toolkit [7]) have attempted to address issues related to resource provisioning. There is a number of package management systems that support deployment of binary and source files (e.g., rpm [8], apt-build [9], apt-build [9]). At this stage of the project, Unibus does not introduce a new resource provisioning system but focuses on the orchestration of tools and toolkits (such as package managers, or proprietary application build/deployment systems) in order to provision resources and setup the execution environment for the software application. Provisioning is orchestrated through the *conditioning* process that changes the resource specialization level using typical deployment software.

Resource aggregation requires 'homogenization' at a certain level, e.g., (1) system level (hardware, software) by utilizing homogeneous systems, (2) standardization level by enforcing conformity to standards or specifications, (3) abstraction level by generalization, or (4) application level by using specific APIs or libraries.

The classical example of resource homogenization at the system level is hardware virtualization. Virtualization-based IaaS (Infrastructure as a Service) such as Amazon EC2 [10], Rackspace [11], GoGrid [12] virtualize hardware and enable users to setup a homogeneous execution environment by allowing them to run instantiable images in many copies. Still, however, aggregation of computing chunks delivered by different providers might be challenging as particular clouds may substantially differ with respect to exposure levels (hardware, OS kernel), access APIs, image formats (if any), security policies, etc.

The standardization approach, in turn, explicitly enforces unification at the level of the standardized entity. Indubitably, the standard-based approach greatly supports cross-resource aggregation (e.g., Globus [13], Grid Interoperation Now [14], InterGrid [15], P-GRADE Grid Portal [16], Lattice [17]). In fact, there are formal and informal standardization initiatives in the growing cloud community such as efforts to propose a standard API (e.g., Simple Cloud API [18], Open Cloud Manifesto [19]), or projects (e.g., EUCALYPTUS [20], Nimbus [21], AppScale [22]) that follow or refer to guidelines established by cloud leaders.

One of the important Unibus foundations is to grant resource providers freedom in exposing their resources to end-users (contrary, for instance, to Grids that restrict resource providers to Virtual Organizations). In this context, resource aggregation at the system or standardization level is inadequate. Therefore, Unibus proposes resource access unification at the abstraction level. Several projects attempt to enable resource aggregation also at that level, e.g., libcloud [23], VGrADS [24], SAGA [25]. The libcloud project aims to find a minimal common API and currently defines a few API unified operations, namely, *list*, *reboot*, *create*, *destroy*, *images*, *sizes*. The Unibus approach to abstracting operations is not as extreme as in the libcloud project and goes beyond one resource class. The Unibus Capability Model abstracts access to resources by specifying abstract *operations* logically grouped into *interfaces* to present resource's capabilities in a unified manner. The VGrADS project is similar to Unibus in that it virtualizes access to different classes of computational resources such as Grids, clouds, job schedulers. However, Unibus goes beyond *matching* application requirements and available resources as it is in the case of VGrADS. Unibus takes a step beyond and attempts to *adapt* resources to the required specialization level, if necessary.

The SAGA project bears a very close resemblance to our approach as it defines standard interfaces such as *job*, *file*, or *MapReduce* and adaptors to particular resources that implement those interfaces. Conceptually, our approach is reversed in regard to SAGA: Unibus mediators expose *all* resources' native operations and Unibus attempts to bind those native operations with abstract interfaces. Aside from that, there is also a technical difference between Unibus and SAGA as SAGA *requires* application modification. Unibus offers API, however, it allows for execution of unmodified applications.

The other example of enabling aggregation at the API level is the GridSolve [26] project. In order to obtain access to aggregated resources, applications need to be linked with the GridSolve client library. As mentioned earlier, Unibus does not require linking applications with any library to execute the application on aggregated resources. In typical situations, Unibus users will write metaapplications that will guide Unibus *what* resources and *how* to use them to execute applications.

In the context of cloud benchmarking, our work is complementary to [27] as they executed NPB-MPI (class B) on Amazon EC2 and on a local cluster; we execute NPB-MPI (class A) on *aggregated* cloud resources provided by two different providers.

III. Unibus Infrastructure Framework

The more detailed description of Unibus is provided in our previous works [28], [29]. In this section, for the completeness sake we briefly describe essential Unibus concepts and components on which resource access virtualization and provisioning are based, in particular: (1) the architectural overview, (2) the Capability Model that defines resource capabilities and mediators that act as resource drivers, (3) conditioning that enables resource provisioning.

A. Unibus Architecture

Unibus foundations draw inspiration from the traditional VO (Virtual Organization) resource sharing model. Unlike in Grids, however, where resource virtualization and aggregation take place at the resource provider's side and are performed by *resource providers*, the Unibus goal is to relieve them from that burden and shift it to *software* (Unibus) at the resource client's side. In this manner, Unibus benefits both resource providers and resource users, as the former expose their resources in an arbitrary way, and the latter can execute their applications on resources orchestrated by Unibus in accordance to their requirements.

The Unibus architecture is presented in Figure 1. In Unibus, resources are exposed by resource providers through *access points*, typically represented by access daemons (e.g., sshd, ftpd, etc). Resources are described semantically by their OWL-DL [30] *resource descriptors* that contain resource-specific data related to authentication methods, available access points, installed system software (OS, libraries, compilers, etc), environment variables, etc. The example resource descriptors are presented in Listing 2 and Listing 3, for the readability reason in the Turtle notation (Terse RDF Triple Language) [31]. Resource virtualization is carried out by the Unibus Capability Model that specifies useful abstractions in the form of abstract *operations* that are logically grouped into resource *interfaces*. The specifics of a particular resource is handled by a *mediator* that implements the access point protocol(s). The Unibus

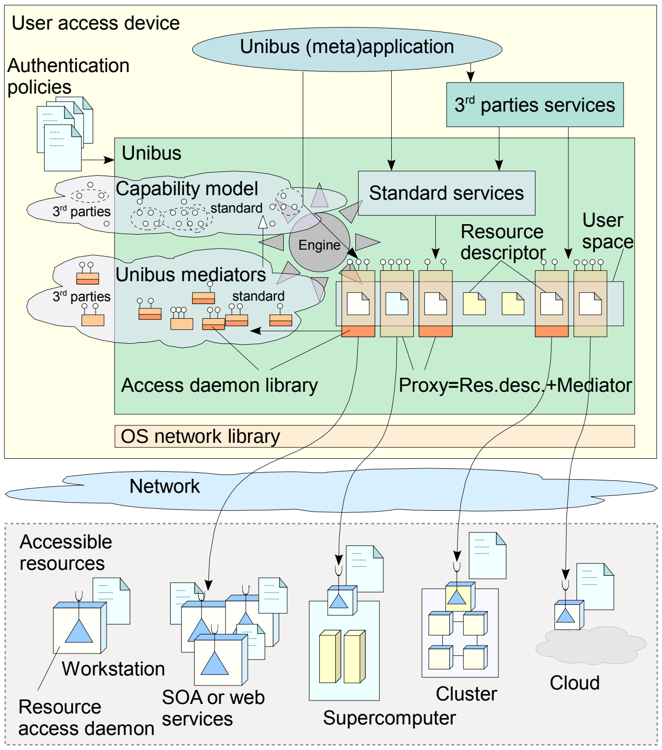


Fig. 1. The overview of Unibus

knowledge engine selects and loads at run-time mediators that are compatible with interfaces requested by the user, and then exposes the resource as a *resource proxy object*. Proxies represent 'connected' resources in Unibus. They implement abstract interfaces' operations from the Unibus Capability Model (via mediators compatible with requested interfaces).

Resource aggregation occurs at the proxy level and at the *synthetic resource* level. Synthetic resources are introduced to Unibus in order to provide a useful abstraction. This is convenient in certain situations, e.g., using one conceptual entity such as the ParallelSsh synthetic resource is more practical than exercising a set of individual ssh-enabled resources in order to execute actions on those resources in parallel.

In order to facilitate common tasks such as resource conditioning, monitoring, resource discovery, etc, Unibus offers its API and *Unibus services* (standard or provided by third parties).

B. Unibus Capability Model

The Unibus Capability Model, schematically presented in Figure 2, is at the heart of resource access virtualization as it specifies abstract *operations* logically grouped into *interfaces*. Both operations and interfaces are defined semantically in the model and uniquely identified by their URIs.

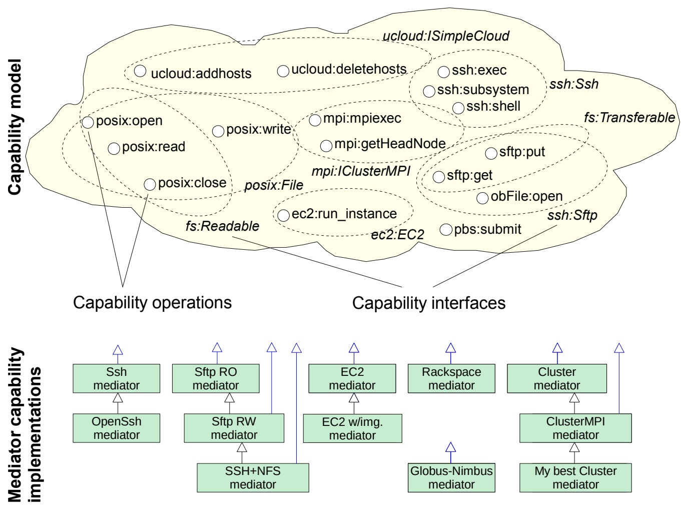


Fig. 2. The Unibus Capability Model

Operations correspond to actions that can be directly performed on resources (and consequently, in Unibus, on resource counterparts, i.e., proxies) such as *mpiexec*, *run_instance*, *get*, *put*. The interfaces combine operations that are logically connected, e.g., represent allowable operations on a given access point. For instance, the interface *IReadable* can combine operations such as *open*, *read*, and *close*; *ISsh – exec*, *put*, *get*; *ISimpleCloud – addhosts*, *deletehosts*.

Particular implementations of abstract operations (and, accordingly, interfaces) are provided via *mediators* that implement the specifics of resource access points. Unibus is capable of selecting the appropriate mediator for a requested interface since mediators' operations are mapped to relevant abstract operations in the Capability Model. Typically, mediators' developers will reuse available libraries to implement access point capabilities. For instance, currently some of supported standard Unibus mediators take advantage of third party libraries; the ssh and sftp mediator is based on the Paramiko SSH2 library [32], the EC2 mediator uses the Boto EC2 library [33]. The rest of mediators provided by Unibus is developed from scratch: the Rackspace mediator, the PBSQueue mediator and the ElasticWulf cluster [34] mediator.

C. Conditioning

Unibus provisions resources through the process known as *conditioning* that increases the resource specialization level. In particular, there are two classes of conditioning services: (1) *soft conditioning*, and (2) *successive conditioning*. Soft conditioning alters resource capabilities in terms of installed software, e.g., installing an MPI implementation on a resource or a gcc compiler allows to execute MPI applications or compile C programs, respectively. Successive conditioning results in enhancement of resource *access* capabilities, e.g., Globus Toolkit installation adds the Grid access point on the resource. Typically, successive conditioning will be supported by soft conditioning in order to enable new access points on a resource.

IV. Cloud Orchestration in Unibus

The aim of this work was to exercise the Unibus approach to automated provisioning and aggregation of cloud resources. For the reasons explained in Section I, we selected NAS Parallel Benchmarks (NPB) [3]. NPB cover a representative spectrum of classes of HPC applications and implement different degrees of intensity of interprocessor communication (no significant interprocessor communication, regular and irregular short and long distance communication, etc). NPB-MPI introduce a few classes with respect to a problem size. Since in our experiment the financial factor was involved, and cloud providers charge also for the in/out bandwidth, after preliminary tests in the class B, finally we decided to perform benchmarks in the smallest class A.

A. Experimental Testbed

The experiment setup to execute benchmarks is presented in Figure 3. Our experiment was hosted on two clouds provided by Amazon EC2 [10] and Rackspace [11]. In many aspects both cloud infrastructures are similar. For instance, hardware virtualization is Xen-based, computational chunks are exposed to end-users via ssh and, to perform typical cloud tasks (image selection, creating/terminating/rebooting/shutdown instances (Amazon EC2) or servers (Rackspace)) can be accessed and managed via API or interactively, via a web page. Differences are subtle, yet remarkable and concern, for instance, processors types and their speed; granularity of computational power available for the user (only 64-bit 4-core hosts with a choice of RAM/HDD sizes up to 16GB/620GB on Rackspace versus a few classes of resources offered by Amazon EC2 including 32-bit/64-bit architectures, 1-8 cores, up to 68GB RAM and 1.67TB HDD); operating systems, only a few Linux-based options on Rackspace versus a plethora of images including Windows-family on Amazon; access tools provided for managing a cloud resource, the lack of commandline

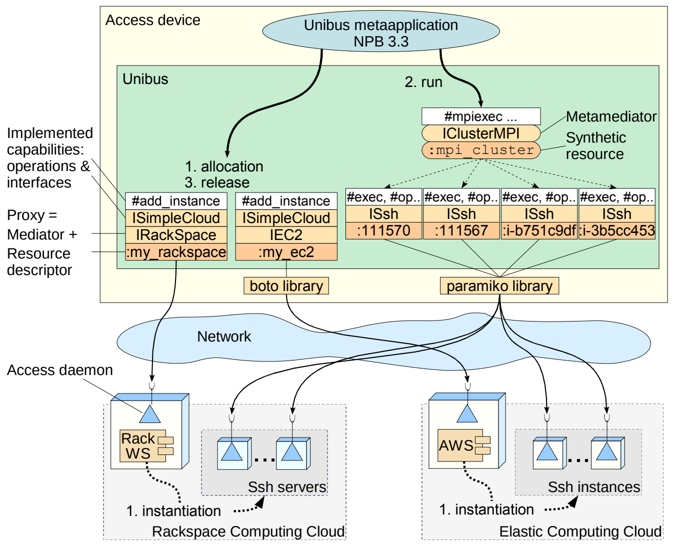


Fig. 3. The experiment setup for the execution of NPB on aggregated Rackspace and Amazon EC2 clouds. The performance degradation is explained in the caption of Figure 4.

tools in the case of Rackspace; expenses monitoring, full and detailed real-time expenses provided on Rackspace and 1-2 day delayed usage reports provided by Amazon EC2; security policies and authentication methods, a RSA key pair to obtain access to Amazon instances and a one-time generated root password to access Rackspace servers, etc.

B. *Virtualization and Aggregation in Practice*

In order to orchestrate cloud resources to execute benchmarks we employed Unibus. The typical Unibus usage scenario requires the creation of a metaapplication that is executed on the local user's machine. As the Unibus framework is implemented in Python, the most straightforward approach to implement the metaapplication is to write a Python script. Listing 1, presents a metaapplication script (inconsiderably edited in comparison to the original script for the readability reason) that we utilized for this experiment. The script performs a few tasks with regard to: (1) managing the clouds, (2) instantiation the individual computing hosts on-demand, (3) provisioning the MPI cluster, (4) launching the benchmarks, and (5) gathering the benchmarks' results.

In order to provide a unified access to alike yet different cloud resources, we introduced an abstract interface, named *ISimpleCloud* that contains two abstract operations: *addhosts* and *deletehosts*. We can take advantage of the *ISimpleCloud* interface after loading the code (the file *clouds.py*, line 7 in Listing 1) that implements the so-called *composite operations*. Composite operations are a convenient method for highlighting that they may be composed from the other, previously specified operations. For instance, the Rackspace implementation of *addhosts* uses *rs:create_server*, *rs:list_nodes*, whereas the Amazon EC2 implementation uses *ec2:instance_run*. Loading composite operations from the file *clouds.py* enables obtaining relevant proxies (which implement the interface *ISimpleCloud*) to cloud hosts. We also had to describe cloud resources in respective resource descriptors for Rackspace and Amazon EC2 resources, as presented in combined Listing 2, for the readability reason in the Turtle (Terse RDF Triple Language) [31] notation. Resource descriptors are dynamically loaded by Unibus (line 7 in Listing 1). The operation *addhosts* (line 23 in Listing 1) launches new hosts in the cloud and updates the Unibus *resource user space* with the information related to the created host (host type: Rackspace server, EC2 instance, Linux OS; hostname, user name, authentication methods (password or private key)). The operation may be provided with optional parameters with regard to the required host architecture, number of cores, RAM, and image identifier. The complementary operation *deletehosts* (line 25 in Listing 1) terminates unnecessary hosts and removes their definitions from the resource user space.

The metaapplication script presents usage of the Unibus *synthetic resources* to facilitate host aggregation. In particular, the script generates the *mpi_cluster* synthetic resource descriptor, presented in Listing 3, to provide a single abstraction for hosts orchestrated to act as an MPI cluster. Meanings of *ClusterMPI*, *headNode* or *node* (line 31, Listing 1) are defined in the Unibus core semantics. The ClusterMPI proxy performs tasks related to the MPICH2 configuration (creates necessary files such as *mpd.conf*, *mpd.hosts*, *machinefile*), benchmarks executions, and obtaining the results.

C. *Technical Experiment Setup*

Benchmarks (NPB-MPI version 3.3) were compiled with default parameters in the problem size class 'A' for 64 processes, and were always executed on 16 4-core computational hosts, i.e., total 64 cores, the 64-bit architecture. We used the MPICH2 [35] MPI implementation. The technical specification of the machines used in our experiment is summarized in Table I.

The script executed two series of all benchmarks. The MPI cluster head node was always allocated in the Amazon's cloud and shared the CPU with benchmark

Listing 1. The Unibus metaapplication that allows to run NAS Parallel Benchmarks on aggregated resources (edited for readability)

```
from unibus import *
#also other imports, helping functions and constants

#load my user space definitions (clouds)
import_resources("my_resources.owl")
#load composite ops addhosts, deletehosts (ISimpleCloud)
load_composite('components/operations/clouds.py')

my_clouds = UResource('my_rackspace'), UResource('my_ec2')
cloud_prx = [r.createProxy('ISimpleCloud') for r in my_clouds]

for needs, cpus in benchmark_node_number_generator(HOST_NO):
  #manage the ssh hosts for the current benchmark series
  hosts = list_Rack(), list_AEC2() #how many hosts currently?

  #need more rackspace/elastic hosts? - use prepared images
  for i in (RS_INDEX, AEC2_INDEX):
    diff = len(hosts[i]) - needs[i]
    #select prepared image
    img_id = (16400, 'ami-fc50b395')[i]
    #need less/more RS/EC2? - remove/add hosts
    if diff < 0:
      cloud_prx[i]['#addhosts'](-diff[i], {'IMG':img_id, 'CORE':4, 'ARCH':64, 'RAM':1024})
    else:
      cloud_prx[i]['#deletehosts'](hosts[i][:diff[i]])

  hosts = list_Rack(), list_AEC2() #how many hosts currently?
  #if 16-1 hosts the one from AEC2 is only for the head node
  all_hosts = (hosts[0] + hosts[1])[:HOST_NO]
  head_node = hosts[1][0] #always exists (at least one)
  add_resource(':mpi_cluster a cluster:ClusterMPI; \
      cluster:headNode %s; cluster:node %s.' %
      (head_node, ','.join(str(i) for i in all_hosts)))

  mpi_res = UResource('mpi_cluster')
  mpi = mpi_res.createProxy('IClusterMPI')

  for test in ('ep', 'cg', 'bt', 'ft', 'is', 'lu', 'mg', 'sp'):
    test_file = '%s.%s.%d' % (test, CLASS, cpus)
    #execute the benchmark
    mpi['#mpiexec'](NPB_PATH + test_file + ' > ' + LOG_PATH, cpus)
    #read the the log and write in the local file
    output_name = '_'.join((test_file, '%02d_%02d' % needs, time.strftime('%m%d%H%M%S')))
    with open(output, 'w') as f:
      head_node_sftp = mpi.get_head_node().createProxy('ISftp')
      f.write(head_node_sftp['#open'](LOG_PATH).read())

  #remove proxy and resource. They will be recreated in the next series
  mpi = None
  remove_resource(mpi_res)
```

Listing 2. Cloud resources' descriptors in the TTL (Turtle) notation (edited for readability)

```

@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix my_resources: <http://www.dcl.mathcs.emory.edu/hwb/ontologies/my_resources.owl#>.
@prefix uc: <http://www.dcl.mathcs.emory.edu/hwb/ontologies/unibus_core.owl#>.
@prefix rs: <http://www.dcl.mathcs.emory.edu/hwb/ontologies/unibus_rackspace.owl#>.
@prefix ec2: <http://www.dcl.mathcs.emory.edu/hwb/ontologies/unibus_ec2.owl#>.
@prefix : <http://www.dcl.mathcs.emory.edu/hwb/ontologies/my_resources.owl#>.
@base <http://www.dcl.mathcs.emory.edu/hwb/ontologies/my_resources.owl>.

<http://www.dcl.mathcs.emory.edu/hwb/ontologies/my_resources.owl> a owl:Ontology;
  owl:imports <http://www.dcl.mathcs.emory.edu/hwb/ontologies/unibus_ec2.owl>,
    <http://www.dcl.mathcs.emory.edu/hwb/ontologies/unibus_rackspace.owl>.

:my_ec2 a ec2:EC2; ec2:accessKeyPair :access_key_pair .

:access_key_pair a ec2:AWSKeyPair; ec2:accessKeyId :access_key_id;
  ec2:secretAccessKey :secret_access_key.

:access_key_id a ec2:AWSAccessKey; uc:filePath "../keys/ec2/ak.txt".

:secret_access_key a ec2:AWSSecretKey; uc:filePath "../keys/ec2/secret_ak.txt".

:my_rackspace a rs:Rackspace; rs:passwordFile "../keys/rackspace_password.txt";
  uc:userName "magg".
```

Listing 3. :mpi_cluster resource descriptor for the 14/2 case in the TTL (Turtle) notation (edited for readability)

```

@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix cluster: <http://www.dcl.mathcs.emory.edu/hwb/ontologies/unibus_cluster.owl#>.
@prefix ec2: <http://www.dcl.mathcs.emory.edu/hwb/ontologies/unibus_ec2.owl#>.
@prefix rs: <http://www.dcl.mathcs.emory.edu/hwb/ontologies/unibus_rackspace.owl#>.
@prefix : <http://unibus.owl#>.
@base <http://unibus.owl>.

<http://unibus.owl> a owl:Ontology;
  owl:imports <http://www.dcl.mathcs.emory.edu/hwb/ontologies/unibus_cluster.owl>.

:mpi_cluster a cluster:ClusterMPI;
  cluster:headNode ec2:i-b751c9df;
  cluster:node rs:111570, rs:111569, rs:111557, rs:111556, rs:111565,
    rs:111555, rs:111568, rs:111561, rs:111558, rs:111564, rs:111567,
    rs:111566, rs:111571, rs:111560,
    ec2:i-b751c9df, ec2:i-3b5cc453.
```

processes. It performed administrative tasks such as launching *mpdboot*; *mpiexec*: first Rackspace nodes are taken (four processes per host), then Amazon EC2 nodes (four processes per host); result staging. We enforced MPI to use exclusively public IP addresses for *all* setups (instead of the local IP addresses) on both clouds in order to enable inter- cloud communication.

We created two images with preinstalled necessary software such as MPICH2 and NPB for each cloud, and performed only the necessary configuration after running instances (setup of host names, i.e., adding *public_DNS* to */etc/hosts*, and executing *hostname public_DNS*).

D. Experimental Results

We performed two types of tests: (1) Unibus-managed execution of NPB (two series), showed in Figures 4, 6, 7; and (2) manually carried on the mpptest benchmark [36] to examine message passing performance in terms of latency and bandwidth in and between two clouds, presented in Figure 5.

1) NPB: The main experiment was performed in two series. The results, in the form of the arithmetic average

TABLE I

THE SPECIFICATION OF MACHINES USED FOR THE EXPERIMENT (SOME INFORMATION OBTAINED FROM THE CPUINFO PROGRAM). THE BANDWIDTH AND PRICES ARE AS REPORTED IN REPORT USAGES AND REGARD ONE FULL SERIES OF TESTS, I.E., ALL BENCHMARKS EXECUTED FOR EACH SETUP

	Rackspace server	EC2 instance
Compute Node	CPU: 64-bit 4-core AMD Opteron(tm) Processor 2350 HE, 2GHz, 512KB cache; RAM: 1GB; HDD: 40GB	CPU: 64-bit 4-core Intel(R) Xeon(R) E5430, 2.66GHz, 6MB cache; RAM: 15GB; HDD: 1.69TB
OS	Fedora Core 8	Ubuntu 9.04
Software	gcc/gfortran 4.3.3; Mpich2-1.2; NPB 3.3	gcc/gfortran 4.1.2; Mpich2-1.2; NPB 3.3
Bandwidth used	IN: 81.8GB; OUT: 78.8GB	Inside EC2: 155.6GB; IN: 16.5GB; OUT: 16.2GB
Price	73.75 [1]USD (reported)	56.97 USD (estimated)

of those two series, are presented in Figure 4. We also present the cumulative time of the average series in Figure 6, and measured cumulative bandwidth IN/OUT reported by Rackspace for the second series, as shown in Figure 7.

2) Mpptest: The results of the *point-to-point* test are presented in Figure 5. This benchmark follows the ping-pong pattern to measure the latency and bandwidth in the function of an MPI message size (0-1024 bytes). The setup for this experiment was as follows: we ran two servers R_1 and R_2 on Rackspace and two instances A_1 and A_2 on Amazon EC2, and we instructed mpptest (via relevant configuration files and input parameters) to run two processes allocated in the following settings: (1) $R_1 \leftrightarrow R_2$ (intra-Rackspace), (2) $A_1 \leftrightarrow A_2$ (intra-Amazon EC2), (3) $R_1 \leftarrow A_1$, (inter-cloud) (4) $R_1 \leftrightarrow R_1$ (intra- Rackspace cores), (5) $A_1 \leftrightarrow A_1$ (intra- Amazon cores). The latency and bandwidth for settings (1) - (3) are presented in Figure 5(a)(b), and Figure 5(c)(d) shows values for inter-cores communication in respective clouds.

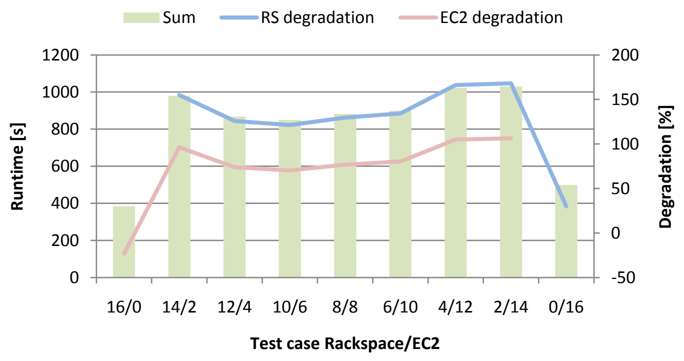


Fig. 6. Total (cumulative) runtime time of the average series of NPB benchmarks (eight benchmarks)

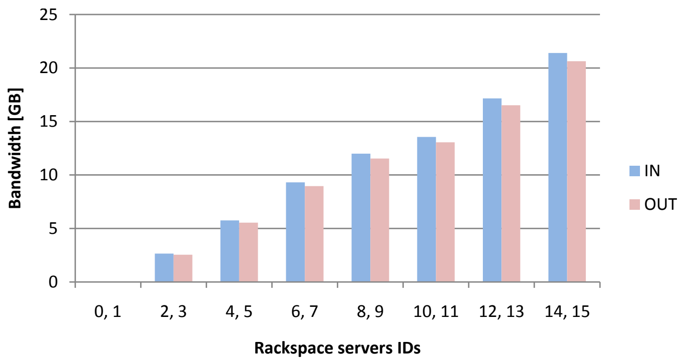


Fig. 7. Reported by Rackspace, the cumulative IN/OUT bandwidth to/from the Rackspace cloud for one series/ The number of running hosts changes as hosts are dynamically added and removed, to reduce the experiment expenses.

E. Discussion

In general, the overall performance of the 'aggregated' cloud is about 2-3 times inferior compared to performance of individual clouds.

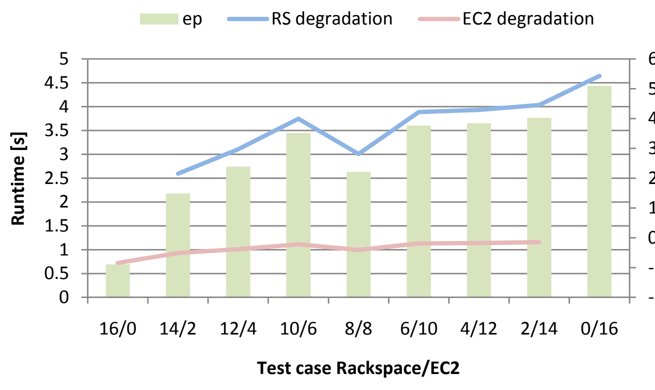
Despite slower Rackspace CPUs (Tab. I, IS in Fig. 4(e) that is computationally demanding), the Rackspace's network better copes with small MPI messages that results in outperforming EC2 (compare LU (Fig. 4(f)) where a huge number of very small (40 bytes) MPI messages is transmitted [37]).

The process allocation impacts benchmarks (multigrid partitioning tests MG, CG, BT, LU (Figs 4 (g)(b)(c)(f)); setups 0/16, 4/12, 8/8, 12/4, 16/0 outperform 2/14, 6/10, 10/6, 14/2 resulting in the comb-like characteristics; to get the reasonable performance processes across clouds should be created with step 2^2 instead of 2 [37]).

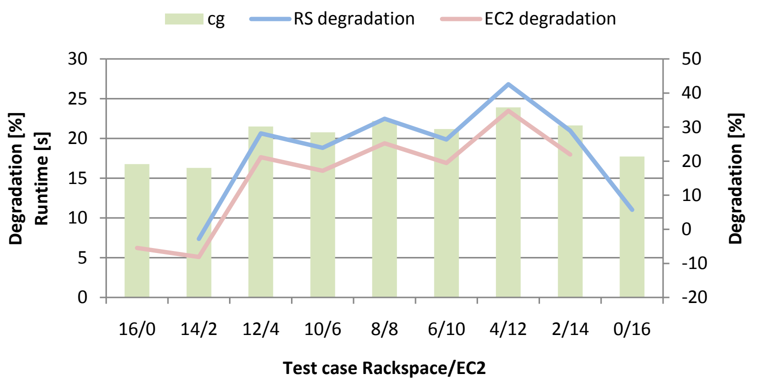
BT and SP (Fig. 4 (c)(h)) that involve coarse grain communication are very sensitive to the slow Internet connection or communication imbalance (e.g., SP: the less inter- communication between clouds and the better communication balance the better overall performance).

EP causes the most difficulties for the interpretation

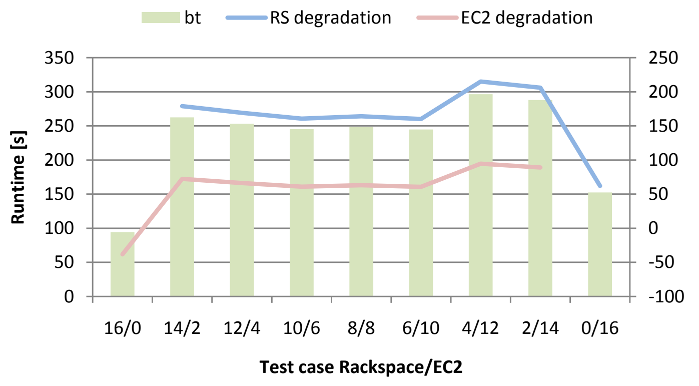
[1]This number does not correspond to the number reported in [1]; it is updated accordingly to a correction we received from the Rackspace, a few months after the experiment



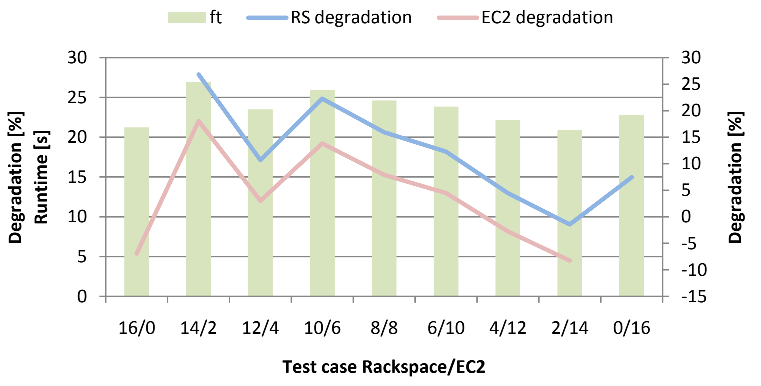
(a) EP



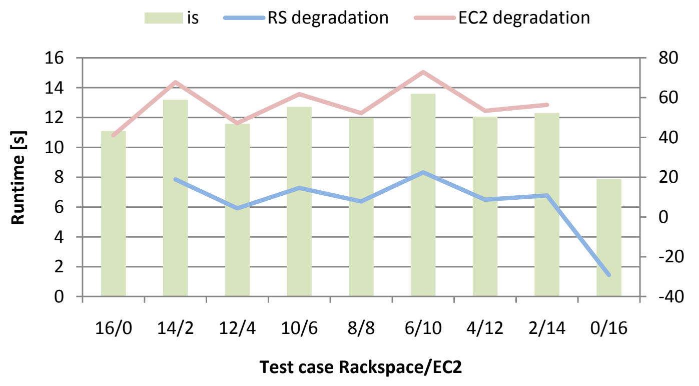
(b) CG



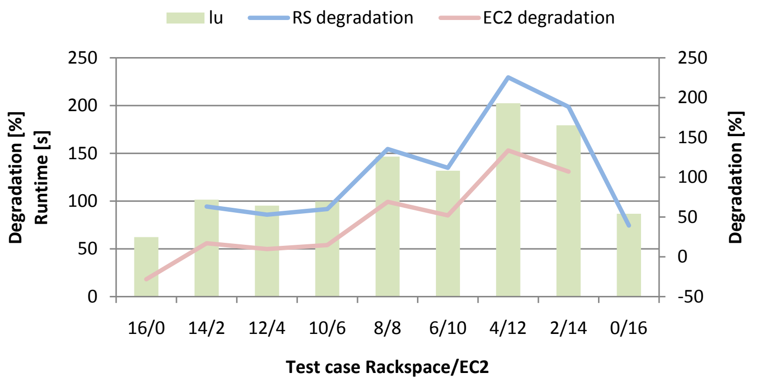
(c) BT



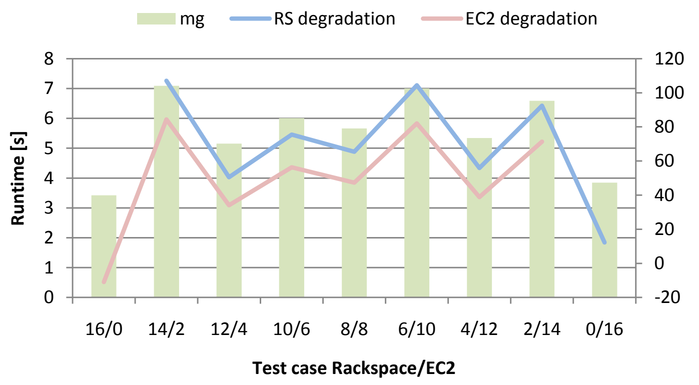
(d) FT



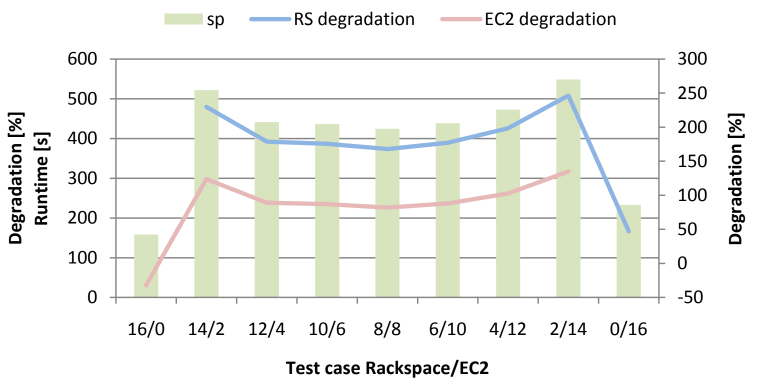
(e) IS



(f) LU



(g) MG



(h) SP

Fig. 4. NPB Benchmarks results. Times (as reported in benchmarks' logs) are the arithmetic average of two series. Blue and red lines show the benchmark performance degradation in the runs on the MPI testbed assembled across two clouds compared to runs executed on the MPI testbed assembled entirely either in Rackspace (16/0, the blue line) or in EC2 (0/16, the red line). Said differently, the performance degradation answers the question how much longer the benchmark runs on a heterogeneous testbed compared to a homogeneous testbed.

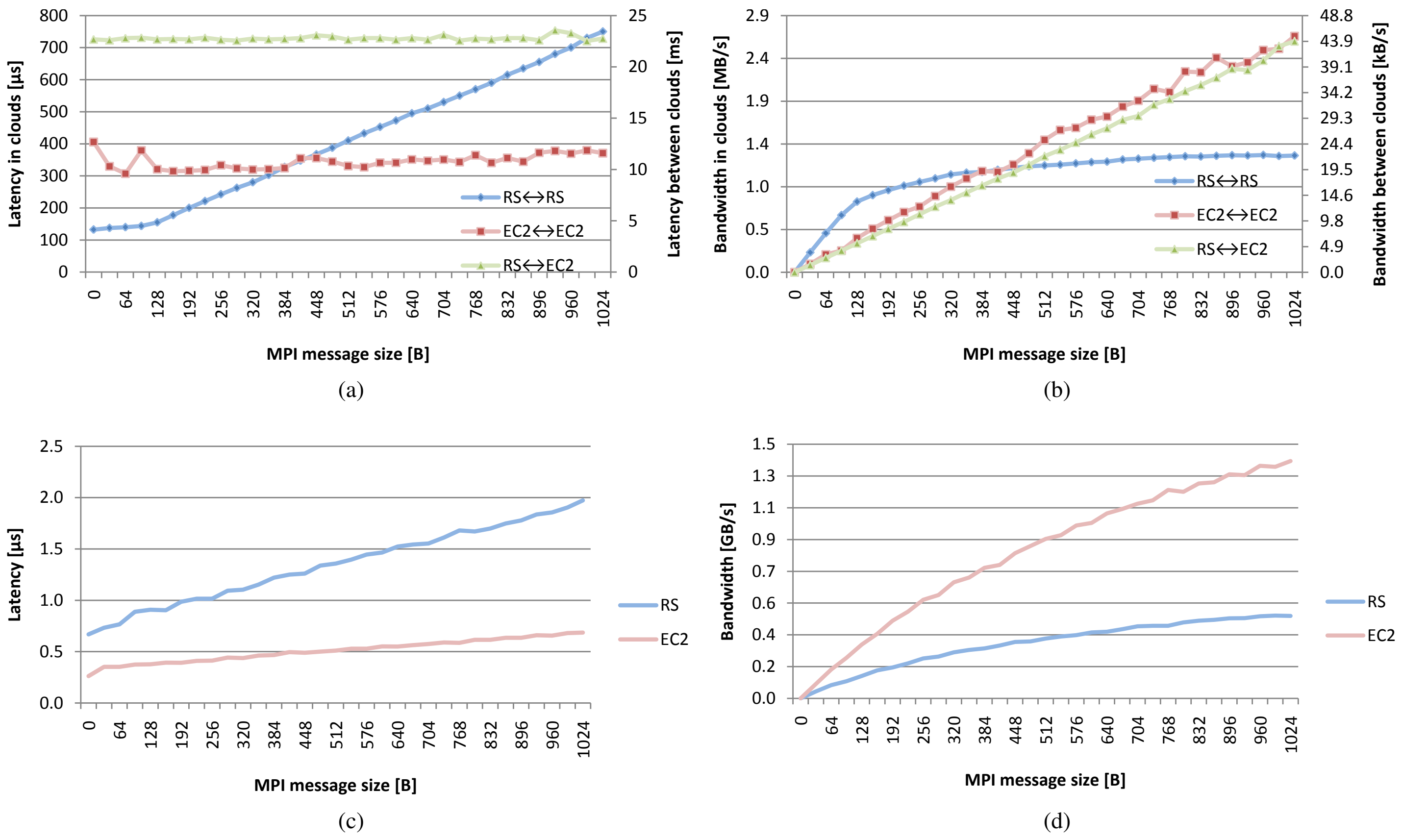


Fig. 5. Latency and bandwidth reported by mpptest for: (a)(b) inter- and intra- cloud communication, (c)(d) the same node inter- core communication

(Fig. 4 (a)) as it should better perform on EC2 than on Rackspace. The reason for that might be the short execution time of the test (an imbalanced ratio processing/communication may favor faster network operations on Rackspace). The problem size class B would increase the execution time of the EP test.

To conclude, we expected even worse performance of the 'aggregated' cloud than we observed (i.e., 2-3 inferior compared to individual clouds). However, relatively slow network connections inside clouds cause that the Internet connection between clouds does not make much difference in the overall benchmark performance. Considering the monetary factor, computing in a single cloud might be even a preferred option to maintaining local clusters (the total cost of a 100 Rackspace servers' cluster starts from $1.5 per hour). However, computing on *aggregated* clouds might be relatively expensive as costs for computing and network usage may significantly vary from one provider to another (e.g., in Rackspace 1 GB bandwidth costs 15 times more than 1 hour of a server; EC2 make those costs at invert).

We also observed the reduction of the effort related to provisioning tasks. In this experiment, Unibus reduces the resource provisioning time from approximately 2 man-hours to roughly 40 minutes (build, staging, image instantiation, etc).

V. Summary

This work aims to demonstrate how to use Unibus to automatically provision and aggregate cloud resources to execute message passing applications. This can be tedious, especially if aggregation of many hosts is considered (even in the single cloud). Our infrastructure framework, Unibus, simplifies and automates many provisioning and aggregation related tasks (deployment, configuration, authentication, etc). To achieve that, we use our Capability Model that allows to virtualize access to resources, mediators that implement the specifics of the resource access points, and conditioning that allows to increase the resource specialization level. The conducted experiment demonstrated that the performance of aggregated clouds is 2-3 times inferior in comparison to the performance of individual clouds.

REFERENCES

[1] J. Slawinski, M. Slawinska, and V. Sunderam, "Unibus-managed Execution of Scientific Applications on Aggregated Clouds," in *CCGRID'10. Melbourne, Australia*, May 2010.

[2] A. Conry-Murray, "From Amazon To IBM, What 12 Cloud Computing Vendors Deliver ," *InformationWeek*, September 2009.

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, *et al.*, "The NAS Parallel Benchmarks," *Intl. J. of HPC Apps*, vol. 5, no. 3, p. 63, 1991.

[4] E. Meek, J. Larkin, and J. Dongarra, "Remote Software Toolkit Installer," Tech. Rep. ICL-UT-05-04, ICL UT, June 2005.

[5] K. Moore and J. Dongarra, "NetBuild: Transparent Cross-Platform Access to Computational Software Libraries," *Concurrency and Computation: Practice and Experience, Special Issue: Grid Computing Environments*, vol. 14, pp. 1445–1456, Nov/Dec 2002.

[6] NERSC, "Modules Approach to Software Management." http://www.nersc.gov/nusers/resources/software/os/modules.php, 2008.

[7] M. Slawinska, J. Slawinski, and V. Sunderam, "Portable builds of HPC applications on diverse target platforms," in *IEEE Intl. Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009*, pp. 1–8, 2009.

[8] E. Bailey, "Maximum rpm. taking the red hat package manager to the limit.," *Online: http://www.rpm.org/max-rpm*, 1997.

[9] "Apt-build." http://freshmeat.net/projects/apt-build/, 2009.

[10] Amazon EC2. http://www.amazon.com/ec2/, 2010.

[11] "The Rackspace Cloud." http://www.rackspacecloud.com/, 2009.

[12] "GoGrid." http://www.gogrid.com/, 2009.

[13] I. T. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems.," *J. Comput. Sci. & Technol.*, vol. 21, pp. 513–520, July 2006.

[14] M. Marzolla, P. Andreetto, V. Venturi, A. Ferraro, S. Memon, S. Memon, B. Twedell, M. Riedel, D. Mallmann, A. Streit, S. van de Berghe, V. Li, D. Snelling, K. Stamou, Z. A. Shah, and F. Hedman, "Open standards-based interoperability of job submission and management interfaces across the grid middleware platforms glite and unicore," in *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, (Washington, DC, USA), pp. 592–601, IEEE Computer Society, 2007.

[15] M. D. D. Assuncao, R. Buyya, and S. Venugopal, "InterGrid: A case for internetworking islands of Grids," in *Concurrency and Computation: Practice and Experience (CPE), Online*, Wiley Press, 2007.

[16] P. Kacsuk, T. Kiss, and G. Sipos, "Solving the grid interoperability problem by P-GRADE portal at workflow level," *Future Gener. Comput. Syst.*, vol. 24, no. 7, pp. 744–751, 2008.

[17] A. L. Bazinet and M. P. Cummings, *Weber, M. H. W. (Ed.) Distributed & Grid Computing – Science Made Transparent for Everyone*, ch. The Lattice Project: a Grid research and production environment combining multiple Grid computing models. Principles, Applications and Supporting Communities. Tectum, (to appear). Rechenkraft.net, Marburg. In press.

[18] "Zend, The PHP Company. The Simple Cloud API." http://www.simplecloudapi.org/, 2009.

[19] "Open Cloud Manifesto." http://www.opencloudmanifesto.org/, 2009.

[20] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus Open-source Cloud-computing System," in *9th IEEE International Symposium on Cluster Computing and the Grid, Shanghai, China*, 2009.

[21] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, and M. Tsugawa, "Science Clouds: Early Experiences in Cloud Computing for Scientific Applications," in *Cloud Computing and Its Application (CCA-08)*, October 2008.

[22] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski, "AppScale Design and Implementation," tech. rep., UCSB Technical Report Number 2009, 2009.

[23] "libcloud – a unified interface to the cloud." http://libcloud.org/, 2009.

[24] Y. Kee and C. Kesselman, "Grid Resource Abstraction, Virtualization, and Provisioning for Time-targeted Applications," in *8th IEEE Intl. Symp. on CCGRID, 2008.*, pp. 324–331, 2008.

[25] A. Merzky, K. Stamou, and S. Jha, "Application Level Interoperability between Clouds and Grids," *Grid and Pervasive Computing Conference, Workshops at the*, pp. 143–150, 2009.

[26] A. YarKhan, J. Dongarra, and K. Seymour, "Gridsolve: The evolution of network enabled solver," in *Grid-Based Problem Solving Environments: IFIP TC2/WG 2.5 Working Conference on Grid-Based Problem Solving Environments*, (Prescott, AZ, USA), pp. 215–226, Innovative Computing Laboratory, University of Tennessee, 2006.

[27] E. Walker., "Benchmarking Amazon EC2," *;Login:*, vol. 33, pp. 18–23, Oct 2008.

[28] J. Slawinski, M. Slawinska, and V. Sunderam, "The Unibus Approach to Provisioning Software Applications on Diverse Computing Resources," in *International Conference On High Performance Computing, 3rd International Workshop on Service Oriented Computing*, Dec 2009.

[29] M. Slawinska, J. Slawinski, and V. Sunderam, "The Unibus Approach to Aggregation of Heterogeneous Computing Infrastructures," in *International Conference On HPC, WGUC*, Dec 2009.

[30] W3C, "OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax, W3C Working Draft," Dec 2008.

[31] W3C, "Turtle – Terse RDF Triple Language." http://www.w3.org/TeamSubmission/turtle/, Jan 2008.

[32] "Paramiko, SSH2 protocol for Python." http://www.lag.net/paramiko, Jan 2009.

[33] M. Garnaat, "Boto. Python interface to Amazon Web Services." http://code.google.com/p/boto/, 2009.

[34] P. Skomoroch, "ElasticWulf: Beowulf cluster run on Amazon EC2." http://code.google.com/p/elasticwulf/, 2009.

[35] W. Gropp, "MPICH2: A New Start for MPI Implementations.," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (D. Kranzlmueller, P. Kacsuk, J. Dongarra, and J. Volkert, eds.), no. LNCS2474 in Lecture Notes in Computer Science, Springer Verlag, 2002.

[36] W. Gropp and E. L. Lusk, "Reproducible measurements of mpi performance characteristics," in *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in PVM and MPI*, (London, UK), pp. 11–18, Springer-Verlag, 1999.

[37] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," tech. rep., 1995.