Evaluating Cloud Computing Techniques for Smart Power Grid Design Using Parallel Scripting

Ketan Maheshwari*, Ken Birman*, Justin M. Wozniak‡, Devin Van Zandt†
*Department of Computer Science
Cornell University, Ithaca, NY 14853
†GE Energy Management, Schenectady, NY 12345
‡MCS Division, Argonne National Laboratory
Argonne, IL 60439

***Abstract*—Applications used to evaluate next-generation electrical power grids ("smart grids") are anticipated to be compute and data-intensive. In this work, we parallelize and improve performance of one such application which was run sequentially prior to the use of our cloud-based configuration. We examine multiple cloud computing offerings, both commercial and academic, to evaluate their potential for improving the turnaround time for application results. Since the target application does not fit well into existing computational paradigms for the cloud, we employ parallel scripting tool, as a first step toward a broader program of adapting portable, scalable computational tools for use as enablers of the future smart grids. We use multiple clouds as a way to reassure potential users that the risk of cloud-vendor lock-in can be managed. This paper discusses our methods and results. Our experience sheds light on some of the issues facing computational scientists and engineers tasked with adapting new paradigms and infrastructures for existing engineering design problems.**

***Keywords*-Parallel scripting, cloud computing, smart grid**

I. Introduction

With the advent of cloud computing, users from multiple application areas are becoming interested in leveraging inexpensive, "elastic" computational resources from external services. Engineers designing an autonomic electrical power grid ("smart grid") constitute one such user group. Stakeholders from both the production and consumption sides of the emerging smart grid are developing computation-intensive applications for multiple aspects of the problem. Some of these concepts will require major technology steps, such as the deployment of synchrophasor based monitoring technologies that could enable real-time grid-state estimation on the production and delivery side of the equation, and the widespread use of smart-meter based technologies to optimize behavior on the consumption side [1]–[3]. As the size of the systems modeled by the software and the number of sensitivities increase, the need to improve computation time for the analysis engines has become crucial.

Our overarching premise is that cloud computing may be best matched to the computation and data management needs of the smart grid, but also that a step-by-step process will be required to learn to carry out tasks familiar from other settings in a smart-grid environment and that, over time, a series of increasingly difficult problems will arise. In this paper we describe our experience in deploying one representative commercial smart grid application to the cloud, and leveraging resources from multiple cloud allocations seamlessly with the help of the Swift parallel scripting framework [4]. The application is used for planning and currently has a time horizon suited primarily to relatively long-term resource allocation questions. Our goal here is to show that cloud resources can be exploited to gain massive speedups without locking the solution to any specific cloud vendor. We present the following

1) A seamless approach for leveraging cloud resources from multiple vendors to perform smart grid applications;
2) A use case that involved parallelizing an existing smart grid application and deploying it on cloud resources;
3) An evaluation of the resulting paradigm for portability and usability to novel application areas.

Applications from many engineering and scientific fields show similar complexities in their characteristics and computational requirements. Thus, one such deployment brings the promise for more applications. The potential benefit is that once the applications are coded, the effort invested pays itself off over a long period by applying the same pattern to similar applications. At the same time, however, we attempt to reduce the complexity of application development by using parallel scripting.

Scripting has been a popular method of automation among computational users. Parallel scripting builds on the same familiar practice, with the advantage of providing parallelization suitably interfaced to the underlying computational infrastructures. Adapting applications to these models of computation and then deploying the overall solution, however, is still a challenge. Parallel scripting has reasonably addressed this challenge by playing a role in deployment of many solutions on HPC platforms [5]. A familiar C-like syntax and known semantics of parallel scripting make the

process usable and adaptable. Its flexibility and expressibility far exceed that of rigid frameworks such as MapReduce.

Traditionally, such applications have been run on established computing facilities. However, organizations have either halted acquisition of new clusters or downsized existing clusters because of their high maintenance costs. Clouds are different from organizational clusters: from a management point of view, the cloud resource provisioning model is accounted at fine granularity of resources; from a computational point of view, the work cycles are readily available with virtualized resources and absence of shared scheduling. In certain contexts, clouds present a model of computation infrastructure management where clusters might not be suitable [6].

In practice, cloud allocations are granted to groups in institutions and often a group ends up having its slice from multiple cloud allocations pies. Furthermore, one is limited by the allocation policies on how much of the resources one can obtain simultaneously from a single allocation. For instance, standard Amazon EC2 allocation allows only 20 cloud instances *per allocation* for a region at a time [7]. Even when cloud resources are virtualized, accessing resources across multiple clouds is not trivial and involves specialized setup, configuration, and administrative routines posing significant challenges.

In our implementation, we seamlessly and securely span application runs to multiple clouds. We use Amazon's EC2, Cornell's RedCloud (*www.cac.cornell.edu/redcloud*), and the NSF-funded FutureGrid (*portal.FutureGrid.org*) cloud in this study. Using Swift, we orchestrate the application tasks that they run on multiple clouds in parallel while preserving the application semantics.

II. Application Characterization

The GE Energy Management's Energy Consulting group has developed the *Concorda Software Suite*, which includes the Multi Area Production Simulation (*MAPS*) and the Multi Area Reliability Simulation (*MARS*). These products are internationally known and widely used [8] for planning and simulating smart power grids, assessing the economic performance of large electricity markets, and evaluating generation reliability.

The *MARS* modeling software enables the electric utility planner to quickly and accurately assess the ability of a power system, comprising a number of interconnected areas, to adequately satisfy the customer load requirements. Based on a full, sequential Monte Carlo simulation model [9], *MARS* performs a chronological hourly simulation of the system, comparing the hourly load demand in each area with the total available generation in the area, which has been adjusted to account for planned maintenance and randomly occurring forced outages. Areas with excess capacity will provide emergency assistance to those areas that are deficient, subject to the transfer limits between the areas.

MARS consists of two major modules: an input data processor and the Monte Carlo simulator. The input processor reads and checks the study data, and arranges it into a format that allows the Monte Carlo module to quickly and efficiently access the data as needed for the simulation. The Monte Carlo module reads the data from the input processor and performs the actual simulation, replicating the year until the stopping criterion is satisfied. The execution of *MARS* can be divided by executing each replication–*marsMain* separately and merging *marsOut*, the generated output for all replications at the end.

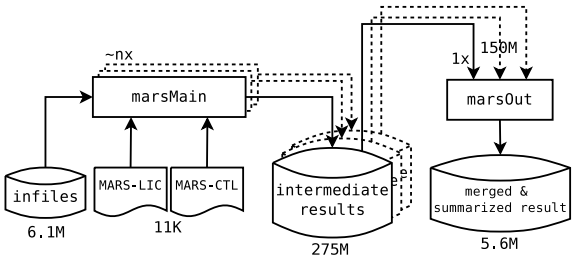

Figure 1: Characterization of the GE MARS application dataflow

A dataflow characterization diagram of *MARS* is shown in Figure 1. The application is a two-stage computational application involving data flow characteristics. The input to first stage consists of raw data, control files, and a license file. The input amounts to 6.1 MB in size. The output to this stage consists of the intermediate results of each replica. The size of outputs varies between 193 and 352 MB, amounting to 275 MB on average. For a medium-sized run, 100 such instances are executed followed by one merge task, totalling 101 jobs. This size could expand to between 1,000 and 10,000 runs in practice. The execution time of each *marsMain* job on a lightly (load average between 0.0 and 0.5) and heavily loaded (load average between 2.0 and 3.5) local host is 35.65 and 37.5 seconds, respectively. However, the time varies significantly depending on the processor load and available compute cycles on the target virtualized environment. See table I for execution time averaged over 100 runs on individual cloud instances for the three cloud infrastructure subjects of this experiment. The *marsOut* stage is highly optimized data merging stage, which takes between 5 and 15 seconds on the resources used for this study. One *marsMain* job submitted from a submit-host will involve the following steps: (1) stage in the 6.2 MB of input data from submit-host to cloud instance; (2) execute the *marsMain* job; and (3) stage out the 275 MB of intermediate results from cloud instance to the submit-host. These steps are performed each time the *marsMain* application is invoked (100 times in this study). The intermediate results are important for application and used for analysis and archival purposes. The *marsOut* stage requires a partial subset of intermediate

results, which amounts to 150 MB for a single run. The ultimate result of *marsOut* amounts to 5.6 MB. Consequently the *marsOut* application run involves the following steps: (1) stage in the 150 MB of input data from submit-host to cloud instance; (2) execute the *marsOut* job; and (3) stage out the 5.6 MB of results from cloud instance to the submit-host.

III. Cloud Infrastructures

In this section, we briefly describe the cloud infrastructures used in the current work and their key properties.

Amazon EC2: Amazon EC2 is a large-scale commercial cloud infrastructure (*aws.amazon.com/ec2/*). Amazon offers compute resources on demand from its virualized infrastructures spanning eight centers from worldwide geographical regions. Three of the centers are in the United States, two in Asia, and one each in the EU, South America, and Australia. An institutional allocation from Amazon will typically allow one to acquire 20 instances of any size per region. In addition, Amazon provides a mass storage device called S3, which can be configured to be mounted on instances as a local file-system. For the current work, we considered the US-based regions mainly for the proprietary and secondly for performance reasons. Consequently, we were limited to a maximum of 60 instances from the Amazon EC2 cloud. Amazon provides a web-based console and a native command-line implementation to create, configure, and destroy resources.

Cornell RedCloud: Cornell's Advanced Computing Center offers a small-scale cloud computing infrastructure through its RedCloud facility . One RedCloud allocation typically allows a maximum of 35 cloud instances drawn from a single 96-core physical HPC cluster on a multi-Gigabit network backbone. The resources are managed through a command-line implementation of the Eucalyptus [10] middleware tool.

NSF FutureGrid Cloud: The NSF-funded FutureGrid cloud is administered by Indiana University. It offers a variety of resources via a multitude of interfaces. Currently, it offers cloud resources via three different interfaces: Eucalyptus, Nimbus (*www.nimbusproject.org*), and OpenStack (*www.openstack.org*). The total number of resources at FutureGrid is close to 5000 CPU cores and 220 TB of storage from more than six physical clusters. We use the resources offered by one such cluster via the Nebula middleware.

Neither RedCloud nor FutureGrid offers a web-based interface to manage resources similar to the one offered by Amazon EC2.

IV. Parallel Scripting in Clouds

We parallelize our application using the parallel scripting paradigm for high performance computing. Swift has been traditionally used on clusters, supercomputers, and computational grids. Recently, it also has gained momentum on cloud environments. In the present work, we employ Swift to run our application on multiple clouds in a seamless fashion. We use Swift and related technologies to express, configure, and orchestrate the application tasks.

Swift script: Swift script provides an efficient and compact C-like syntax and advanced parallel semantics to express an application's tasks and dataflow. Parallel constructs such as `foreach` and future variables provide for implicit parallelism. Advanced mappers and `app` definitions easily map script variables to application data and executables, respectively.

Coasters: The Swift Coasters [11] framework provides a service-worker interfaced with Swift task dispatching framework on the inside and a variety of computing infrastructures from the outside. The execution provider schedules and coordinates application execution on target infrastructure. The data provider stages data. Coaster services connect to worker agents on remote nodes securely using ssh tunnels, thus providing crucial data communications security across clouds.

Collective Data Management: Collective data management (CDM) [5] techniques improve the data staging performance when data is available on shared filesystems. It creates symbolic links instead of actually moving data, thus saving on data staging time.

Karajan Execution Engine: The Karajan engine [12] orchestrates the tasks defined and ensures the right connections between the tasks dictated by dataflow semantics of application.

V. Experiments: Setup and Implementation

In this section, we describe the experiments conducted on cloud infrastructures via a parallel scripting implementation and execution of GE MARS application using the Swift framework.

Bringing up cloud instances: Suitable "machine-images" were prepared in advance for each of the cloud infrastructures. This is a one-time activity: the images can be stored in the cloud account and reused to create instances. The application binaries and supporting libraries were pre-installed to these images. No special software was required for Swift, since coaster workers run standard Perl, which is installed by default. Data is largely dynamic, so it is of little practical value to have data on the images. A separate Swift script was used to run in parallel to bring up the cloud instances on multiple clouds. In our case, parameterized commands to the cloud middleware run in parallel to bring up a desired combination of cloud instances.

Data movement: The data from the first stage of computation, *marsMain*, is required as input to the second stage, *marsOut*. Since there are 100 instances of results from first stage, they all would be required to stage at the location of the execution of second stage. In order to avoid this

localhost1	localhost2	Amazon EC2	Cornell RedCloud	FutureGrid
35.65 ±5.01	49.62 ±7.41	68.49 ±11.43	55.21 ±10.41	47.89 ±7.71

Table I: Average execution time in seconds of a single *marsMain* task with standard deviation on the three cloud instances

expensive staging, the lightweight *marsOut* was set up to run on the submit-host.

Security and firewalls: Each of the cloud environments we used has its own security policies; and in all cases the connection to outside world were closed, which required special configuration to open. However, the port 22 for secure ssh connections was open for all cases. We used the ssh port forwarding and tunneling strategy, which saved us the effort of configuring firewalls on each of the instances while providing a secure data channel.

Distributed file system: A parallelizing environment must run efficiently in both a shared and a distributed file system. In order to form run strategies that spans multiple infrastructures.

Network bias: In order to avoid a network affinity bias, the experiments were conducted from a remote machine outside the network domains of the target cloud infrastructures, especially the Cornell RedCloud.

```
type file;
app (file _maino,file _res[],file _binres) marsmain (...){
  mars @_mainctl stdout=@_maino stdin="/dev/null";
}
app (file _outo, file _outres[]) marsout (...){
  marsout @_outctl stdout=@_outo;
}
// list of control files
string ctlfilelist[] = readData ("ctlfilelist.txt");

//map the items in above list to actual files
file ctl[]<array_mapper; files=ctlfilelist>;
file inp[]<filesys_mapper; location="infiles/">;
file out[]<simple_mapper; location="outs">;

string binresfilelist[] = readData ("binresfilelist.txt");
file binres[]<array_mapper; files=binresfilelist>;
// Licence file
file licence<single_file_mapper; file="MARS-LIC">;
foreach ctlfile, i in ctl {
  file res[]<ext; exec="mapper.sh", arg=i>;
  (out[i],res,binres[i]) = marsmain (ctlfile,licence,inp);
}
file outo<"outo.txt">;
file outctl<"mars-out.ctl">;
file msgerr<"result0/mars.ot09">;
string outresfilelist[] = readData("outresfilelist.txt");
file outres[]<array_mapper; files=outresfilelist>;
(outo, outres)=marsout (outctl, binres, msgerr);
```

Listing 1: A Swift script specification of GE MARS application: lines 2-7 define app calls; lines 20-23 make parallel calls to *marsMain*.

In less than 30 lines of code, Swift can specify the application flow. Although the real work is done by the Swift framework and application code, the abstraction helps users rapidly express, parallelize, and productionalize applications.

VI. Results

In this section, we present the results we obtained by parallelizing the application and deploying it on multiple clouds: Amazon EC2, Cornell's RedCloud, and NSF-funded FutureGrid cloud.

We first present the cloud characterization results by measuring network and data-movement properties of clouds. We then perform our application execution on incrementally sophisticated scenarios: starting from a single localhost to single cloud in serial mode to multiple clouds in task-parallel mode. The application submission was done from a single remote submit-host. The application data resides on the submit-host, and the executables with supporting libraries were preinstalled on cloud images from which cloud instances were spawned.

Figure 2 shows an asymmetric bandwidth matrix between the cloud instances and the submit-host considered in this work. All measurements are obtained by using the Linux "iperf" network performance measurement utility. The rows are servers and the columns are clients. Separate measurements of 20 iperf sessions were recorded over 20 days. Mean and standard deviation of bandwidths were recorded. A spectrum of bandwidth values across the cloud instances and between the instances of the same cloud is seen. Some of the measurements that go beyond 1 Gbit gives an indication that those instances are probably sliced from a single high-speed cluster or even a single physical machine. The bandwidth between two regions of Amazon EC2 was observed to be significantly and unusually lower compared with that of other pairs.

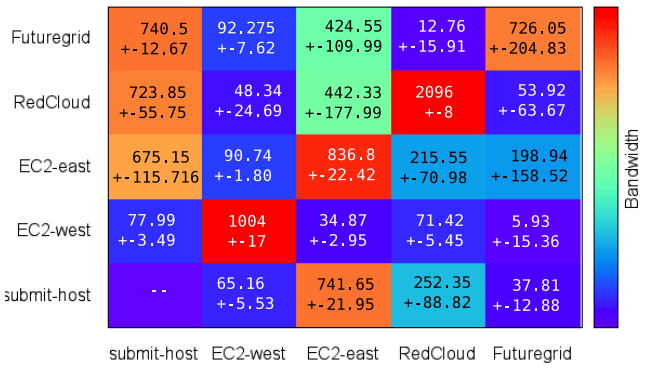

Figure 2: Heatmap for intercloud network performance matrix. The measurements are average bandwidths in Mbits/sec over 20 readings with standard deviation. Color blue indicates a low bandwidth, while red indicates a high bandwidth.

Shown in Figure 3 are the performance results of moving data in different sized files (1 M to 1000 M) to different cloud locations. The measurements were made for the Linux *scp* secure copy utility. In the special case of Amazon S3, the system was mounted on a running cloud instance using the "fuse" [13] software service. The data was written to the S3 mount point by using the Linux "dd" utility. We use the measurements over local file system as benchmarks and see that a locally mounted S3 drive performs worst for 10 M and only second from worst for the 1000 M case.

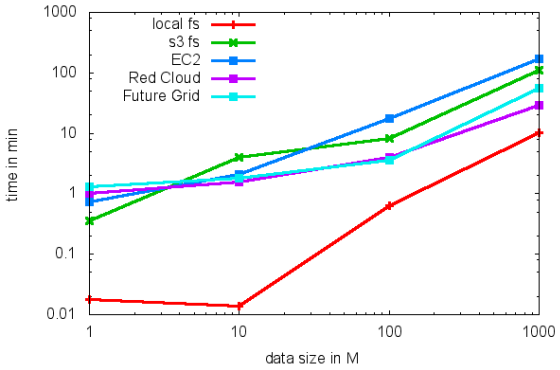

Figure 3: Data movement times across file systems.

The plot shown in Figure 4 is the application performance on a single host. The host has 32 CPU cores and was set up to run on successively higher degrees of parallelism utilizing from 1 to 32 cores. The application was run in two modes: a simple file-based data staging and under the CDM mode where in the input files were symbolically linked to the execution directory for each run (this saved 100 $\times$ 6.2 M of data movement for complete application run). We see significant improvement in performance for up to 8 cores however, no performance gain was achieved beyond this because of a high volume of disc I/O dominating the run.

The plots in Figure 5 show the time to bring up the cloud instances after the command was invoked, reflecting the elasticity of each cloud. We see a marked increase in time to an order of magnitude between those of Amazon EC2 compared with RedCloud and FutureGrid clouds. Note that by default, FutureGrid running the Nimbus interface does not have a means to submit multiple requests in parallel; therefore, a semi-parallel method had to be implemented, running requests in close succession to each other in order to avoid instance-ids to collide.

Figure 6 shows the application's performance on individual cloud resources using a single core in a sequential data staging and execution order versus a parallel execution and data staging on 10 cores. While we clearly get an advantage in speed for parallel execution, a significant performance variation is also seen in serial execution among the cloud infrastructures.

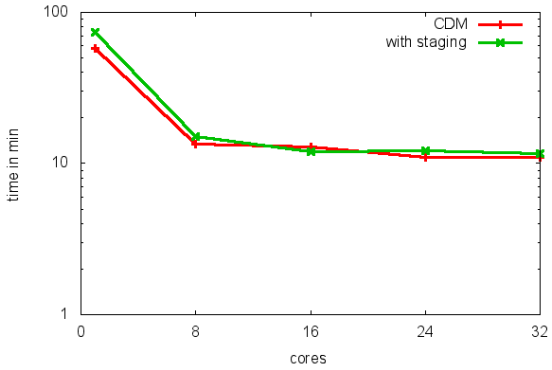

Figure 4: Performance on a single large machine (32 cores). Shown here is performance on an increasing number of cores in two modes of file movement: staging and Collective Data Management (CDM).

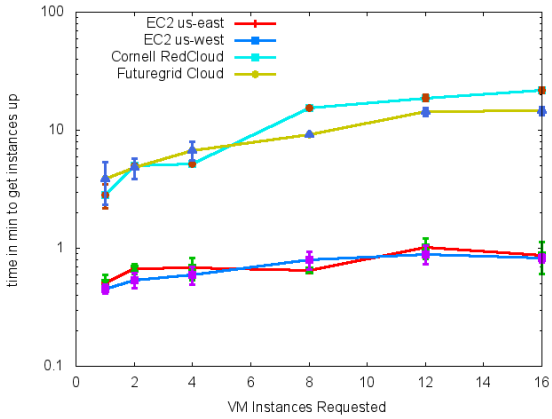

Figure 5: Elasticity measurement for clouds.

The plot in Figure 7 shows the timeline for the serial execution on cloud shown in Figure 6 (FutureGrid). The time line is plotted from an analysis of the Swift log for this run. In terms of percentage, stage-in activity is 1.08%, stage-out is 48.8%, and execution is 50.06% of the time. Note that the stage-in stage completes rapidly and does not get recorded for most instances. A zoomed-in version of a small interval shows the stage-in stage with respect to the adjacent running and stage-out stages. The time line shows potentials for parallelization not only in application execution but also data staging. The parallel version of application execution performs a configurable number of stagings and executions in parallel. This is especially beneficial in cases where staging time is almost equal to or is greater than the execution time.

Figure 8 shows application performance results on different combinations on instances on multiple clouds. We notice a significant performance improvement going from 10 to 20 instances. However, the performance improvements

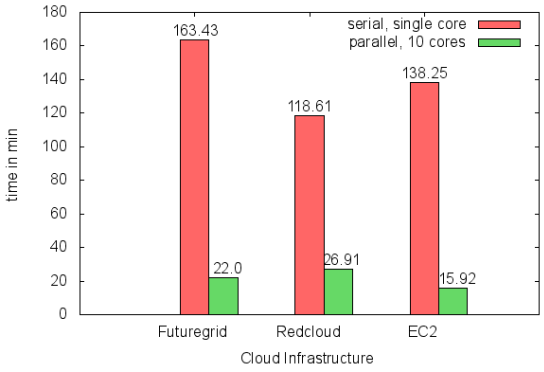

Figure 6: Performance on individual cloud infrastructures: serial on single instance versus parallel on ten instances.

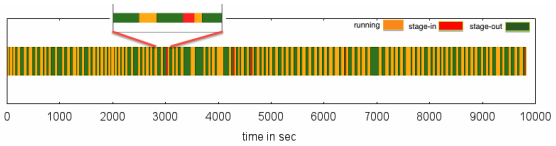

Figure 7: Serial execution timeline on FutureGrid showing intervals for application run, data stage-in and stage-out.

are not linearly proportional as we increase the number of cloud instances successively. This behavior is caused by a significant stage-out time in the run which is bound to a single input channel of fixed bandwidth coming into the submit-host.

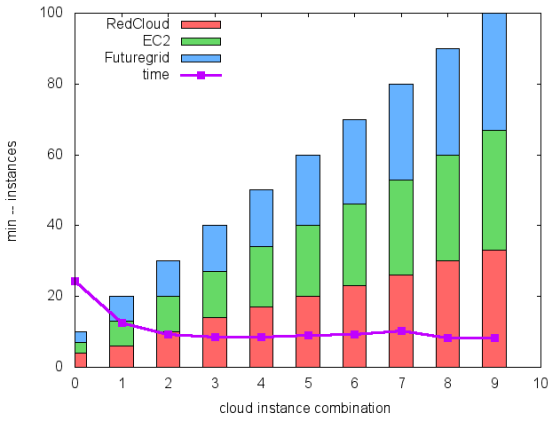

Figure 8: Performance on combinations of instances from multiple clouds.

VII. Related Work

Our work concerns the three broad research areas, cloud computing, parallel and distributed application orchestration, and smart power grid computations. In this section, we discuss related work from each area.

A. Cloud Computing

A large section of the community has a collective vision [14]–[17] for the near and long-term future of distributed and cloud computing comprising the following salient points:

1) A wide scale spread and adaptation of cloud models of computation across HPC and HTC infrastructures
2) Economical utilization of storage space and computational power by adapting more and more new application areas to run in clouds

Workflow-oriented applications have been reported to be specially suited to the cloud environment [18], [19]. Swift has been ported and interfaced to one cloud [20]. Ours is the first multi-cloud implementation.

Cloud performance issues have been studied in the past [17], [21]. Our work covers these areas, albeit with a finer view of evaluating cloud characteristics for a new application area. With this approach, we attempt to validate the community vision while at the same time solve a real-world problem.

B. Parallel and Distributed Application Orchestration

Interoperability among multiple distributed systems has been a hot topic in distributed computing community. The recent SHIWA [22] project addressed many of the challenges facing users seamlessly running precoded workflow applications on multiple distributed computing infrastructures. The dominant approach in SHIWA has been to wrap the workflow expression in order achieve interoperable workflows on top of already-running workflows ported to selected infrastructures. We believe that the scripting approach to workflow [23] and the coaster mechanism makes interoperability easier by providing a portable and compact representation of application ready to be interfaced to infrastructure without wrappers.

MapReduce [24] is a system designed to run two function combinators in a distributed environment. Modern MapReduce distributions such as Hadoop (*hadoop.apache.org*) come with many components that have their own adaptation curve involving learning, familiarizing, installation and setup. These steps often prove to be barriers to effective usage by scientific end-users.

Swift is a Turing-complete language that can run arbitrary compositions of applications on a distributed system, including MapReduce-like systems. In short, Swift can do MapReduce, but MapReduce cannot do Swift. Some attempts have been made to improve the applicability of MapReduce to scientific applications, such as the addition of features to support iteration [25]. We feel, however, that the conventional control constructs (`foreach` loops, `if` blocks) in Swift enable a more natural, expressive language for quickly constructing scientific workflow prototypes or adding to existing scripts.

C. Smart Power Grid Applications

The timely availability of processed data, supporting configuration, and application libraries is a key to performance computing for smart grid applications. Many smart grid applications are inherently distributed in nature because of a distributed deployment of devices and buses. The work described in [26] is the closest treatment of steering smart grid computations into the clouds. The work analyzes smart grid application use-cases and advocates a generic cloud-based model. In this regard, our work verifies the practical aspects of the model presented, by evaluating various aspects of clouds.

VIII. Evaluation

In this section we present an evaluation of the cloud infrastructure characteristics and parallel scripting paradigm in light of our experience deploying the GE-MARS application.

A. Usability

Clouds present a familiar usage model of traditional clusters with an advantage of direct, super-user, scheduler-less access to the virtualized resources. This gives the users much required control over the resources and simplifies the computing without jeopardizing the system security.

We do observe disparities between the commercial and academic clouds in terms of elasticity and performance. Network bandwidth plays a crucial role in improving application performance. Data movement in clouds is only as fast as the underlying network bandwidths. Bandwidth disparities in clouds and those between regions of a single cloud must be taken into account before designing an application distribution strategy. In a mixed model such as ours, prioritizing tasks could alleviate many of these disparities.

Swift is easy to set up. Installation is required only on the submit host. Coasters uses the native, local file system and dynamically installs worker agents to run on the target cloud instances. Swift is less invasive of applications compared with systems such as Hadoop which requires close integration with applications and a customized file system installation on resources. However, it is a relatively new paradigm of parallel computing. This arguably poses adaptability challenges for new applications. The concept of a highly expressive yet implicitly parallel programming language does impose a learning curve on the users used to traditional imperative scripting paradigms. Debugging in such scenarios is one of the biggest challenges for users. However, the returns on investment are expected to be positive as many applications in the smart grid domain exhibit similar patterns [27].

B. Economy

The economy of computation in presence of commercial-academic collaboration is especially notable. Thanks to a universal, pay-as-you-go model of computation, we are not dealing with cluster maintenance and cross-institutional access issues. With the ability to run the application on multiple clouds, we can move on to another cloud if need be and avoid vendor lock-in. A high-level policy and usage agreement allows the costs of cloud allocation to be shared among multiple parties having stakes in the same research.

IX. Conclusions and Ongoing Work

In this paper we discuss and evaluate the cloud side of a network-intensive problem characterized by wide-area data collection and processing. We use a representative parallel scripting paradigm. We analyze the properties of multiple cloud systems as applied to our problem space. One notable limitation of each of the environments is that they do not have efficient support for fault tolerance and seamless assurance of data availability in the event of failure.

Not only computational but performant bandwidth resources are needed in order to achieve desired application performance. Apart from a basic application execution, in a complex and networked environment, additional requirements are foreseen. These requirements include high assurance, dynamic configuration, fault tolerance, transparent connection migration, distributed data repository, and overall task coordination and orchestration of computation. Not all requirements are addressed in this work. However, the resource provision model of our implementation forms a strong basis to address these requirements.

The intercloud bandwidth analysis is useful for large scale task placement. Each independent instance of pipelines could be placed on nodes showing high affinity in terms of bandwidth. Additionally, future work is taking advantage of specialized multi-core platforms offered by cloud vendors, usage of efficient distributed caching technologies offered by tools such as `memcached`.

Acknowledgment

We thank the funding agency, ARPA-e, for the research grant. We thank the department colleagues Robbert van Renesse, David Bindel, and Colin Ponce for their valuable inputs in this work.

References

[1] A. Bose, "Smart transmission grid applications and their supporting infrastructure," *IEEE Transactions on Smart Grid*, vol. 1, no. 1, pp. 11–19, Jun. 2010.

[2] J. Hazra, K. Das, D. P. Seetharam, and A. Singhee, "Stream computing based synchrophasor application for power grids," in *Proceedings of the first international workshop on High Performance Computing, Networking and Analytics for the Power Grid*, ser. HiPCNA-PG '11. New York, NY, USA: ACM, 2011, pp. 43–50. [Online]. Available: http://doi.acm.org/10.1145/2096123.2096134

[3] E. Lightner and S. Widergren, "An orderly transition to a transformed electricity system," *Smart Grid, IEEE Transactions on*, vol. 1, no. 1, pp. 3–10, Jun. 2010.

[4] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 39, no. 9, pp. 633–652, September 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111000524

[5] J. M. Wozniak and M. Wilde, "Case studies in storage access by loosely coupled petascale applications," in *Proc. Petascale Data Storage Workshop at SC'09*, 2009.

[6] S. Jha, D. S. Katz, A. Luckow, A. Merzky, and K. Stamou, "Understanding scientific applications for cloud environments," in *Cloud Computing: Principles and Paradigms*, R. Buyya, J. Broberg, and A. M. Goscinski, Eds., March 2011, ch. 13, p. 664.

[7] "Amazon EC2 FAQ." [Online]. Available: http://aws.amazon.com/ec2/faqs/#How_many_instances_can_I_run_in_Amazon_EC2

[8] L. A. Freeman, D. T. Van Zandt, and L. J. Powell, "Using a probabilistic design process to maximize reliability and minimize cost in urban central business districts," in *18th International Conference and Exhibition on Electricity Distribution, 2005. CIRED 2005.*, june 2005, pp. 1–5.

[9] J. S. Liu and R. Chen, "Sequential monte carlo methods for dynamic systems," *Journal of the American Statistical Association*, vol. 93, pp. 1032–1044, 1998.

[10] 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/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09)*, May 2009, pp. 124–131.

[11] M. Hategan, J. Wozniak, and K. Maheshwari, "Coasters: uniform resource provisioning and access for scientific computing on clouds and grids," in *Proc. Utility and Cloud Computing*, 2011.

[12] G. von Laszewski, M. Hategan, and D. Kodeboyina, "Java CoG kit workflow," in *Workflows for e-Science*, I. Taylor, E. Deelman, D. Gannon, and M. Shields, Eds. Springer, 2007, ch. 21, pp. 341–356.

[13] "FUSE: Filesystem in Userspace." [Online]. Available: http://fuse.sourceforge.net/

[14] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X08001957

[15] K. Yelick, S. Coghlan, B. Draney, and R. S. Canon, "The Magellan Report on Cloud Computing for Science," US Department of Energy, Washington DC, USA, Tech. Rep., Dec. 2011. [Online]. Available: http://www.nersc.gov/assets/StaffPublications/2012/MagellanFinalReport.pdf

[16] D. S. Katz, S. Jha, M. Parashar, O. Rana, and J. B. Weissman, "Survey and analysis of production distributed computing infrastructures," *CoRR*, vol. abs/1208.2649, 2012.

[17] A. Iosup, S. Ostermann, M. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 931–945, june 2011.

[18] S. Crago, K. Dunn, P. Eads, L. Hochstein, D.-I. Kang, M. Kang, D. Modium, K. Singh, J. Suh, and J. Walters, "Heterogeneous cloud computing," in *IEEE International Conference on Cluster Computing (CLUSTER)*, sep 2011, pp. 378–385.

[19] Y. Zhao, X. Fei, I. Raicu, and S. Lu, "Opportunities and challenges in running scientific workflows on the cloud," in *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on*, oct. 2011, pp. 455 –462.

[20] K. Maheshwari, J. M. Wozniak, A. Espinosa, D. Katz, and M. Wilde, "Flexible cloud computing through Swift Coasters," in *Proc. Cloud Computing and its Applications*, 2011.

[21] A. Iosup, M. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *11th IEEE/ACM Int'l Symp. on Cluster, Cloud, and Grid Computing (CCGrid)*. IEEE, May 2011, pp. 104–113. [Online]. Available: http://dx.doi.org/10.1109/CCGrid.2011.22

[22] V. Korkhov, D. Krefting, J. Montagnat, T. Truong Huu, T. Kukla, G. Terstyanszky, D. Manset, M. Caan, and S. Olabarriaga, "SHIWA workflow interoperability solutions for neuroimaging data analysis," *Stud Health Technol Inform*, vol. 175, 2012.

[23] K. Maheshwari and J. Montagnat, "Scientific workflows development using both visual-programming and scripted representations," in *International Workshop on Scientific Workflows(SWF'10)*, ser. . Miami, FL.: IEEE, Jul. 2010. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00677817/PDF

[24] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[25] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative MapReduce," in *Proc. of 19th ACM Intl. Symp. on High Performance Distributed Computing*, ser. HPDC '10. New York: ACM, 2010, pp. 810–818. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851593

[26] S. Rusitschka, K. Eger, and C. Gerdes, "Smart grid data cloud: A model for utilizing cloud computing in the smart grid domain," in *2010 First IEEE International Conference on Smart Grid Communications (SmartGridComm)*, Oct. 2010, pp. 483–488.

[27] K. Maheshwari, M. Lim, L. Wang, K. Birman, and R. van Renesse, "Toward a reliable, secure and fault tolerant smart grid state estimation in the cloud," in *Innovative Smart Grid Technologies*. Washington DC, USA: IEEE-PES, Feb. 2013, *accepted*.