A Framework for Elastic Execution of Existing MPI Programs

Aarthi Raveendran Tekin Bicer Gagan Agrawal
Department of Computer Science and Engineering, Ohio State University
{*raveendr,bicer,agrawal*}*@cse.ohio-state.edu*

***Abstract*—There is a clear trend towards using cloud resources in the scientific or the HPC community, with a key attraction of cloud being the *elasticity* it offers. In executing HPC applications on a cloud environment, it will clearly be desirable to exploit elasticity of cloud environments, and increase or decrease the number of instances an application is executed on during the execution of the application, to meet time and/or cost constraints. Unfortunately, HPC applications have almost always been designed to use a fixed number of resources.**

This paper describes our initial work towards the goal of making existing MPI applications elastic for a cloud framework. Considering the limitations of the MPI implementations currently available, we support adaptation by terminating one execution and restarting a new program on a different number of instances. The components of our envisioned system include a decision layer which considers time and cost constraints, a framework for modifying MPI programs, and a cloud-based runtime support that can enable redistributing of saved data, and support automated resource allocation and application restart on a different number of nodes.

Using two MPI applications, we demonstrate the feasibility of our approach, and show that outputting, redistributing, and reading back data can be a reasonable approach for making existing MPI applications elastic.

I. Introduction

Scientific computing has traditionally been performed using resources on supercomputing centers and/or various local clusters maintained by organizations. However, in the last 1-2 years, cloud or utility model of computation has been gaining momentum rapidly. Besides its appeal in the commercial sector, there is a clear trend towards using cloud resources in the scientific or the HPC community.

The notion of on-demand resources supported by cloud computing has already prompted many users to begin adopting the Cloud for large-scale projects, including medical imaging [1], astronomy [2], BOINC applications [3], and remote sensing [4], among many others. Many efforts have conducted cost and performance studies of using Cloud environments for scientific or data-intensive applications. For instance, Deelman *et al.* reported the cost of utilizing cloud resources to support a representative workflow application, Montage [2]. Palankar *et al.* conducted an in-depth study on using S3 for supporting large-scale computing[5]. In another work, Kondo *et al.* compared cost-effectiveness of AWS against volunteer grids [3]. Weissman and Ramakrishnan discussed deploying Cloud proxies [6] for accelerating web services.

Multiple cloud providers are now specifically targeting HPC users and applications. Though initial configurations offered by the cloud providers were not very suited for traditional *tightly-coupled* HPC applications (typically because they did not use high performance interconnects), this has been changing recently. In November 2010, Mellanox and Beijing Computing Center have announced a public cloud which will be based on 40 Gb/s Infiniband. Amazon, probably the single largest cloud service provider today, announced *Cluster Compute Instances* for HPC in July 2010. This allows up to a factor of 10 better network performance as compared to a regular collection of EC2 instances, and an overall application speedup of 8.5 on a "comprehensive benchmark suite"[1].

The key attractions of cloud include the *pay-as-you-go* model and *elasticity*. Thus, clouds allow the users to instantly scale their resource consumption up or down according to the demand or the desired response time. Particularly, the ability to increase the resource consumption comes without the cost of *over-provisioning*, i.e., having to purchase and maintain a larger set of resources than those needed most of the time, which is the case for the traditional *in-house* resources. While the elasticity offered by the clouds can be beneficial for many applications and use-scenarios, it also imposes significant challenges in the development of applications or services. Some recent efforts have specifically focused on exploiting the elasticity of Clouds for different services, including a transactional data store [7], data-intensive web services [8], and a cache that accelerates data-intensive applications [9], and for execution of a bag of tasks [10].

While executing HPC applications on a cloud environment, it will clearly be desirable to exploit elasticity of

[1]Please see www.hpcinthecloud.com/offthewire/Amazon-Introduces-Cluster-Compute-Instances-for-HPC-on-EC2-98321019.html

cloud environments, and increase or decrease the number of instances an application is executed on during the execution of the application. For a very long running application, a user may want to increase the number of instances to try and reduce the completion time of the application. Another factor could be the resource cost. If an application is not scaling in a linear or close to linear fashion, and if the user is flexible with respect to the completion time, the number of instances can be reduced, resulting in lower $nodes \times hours$, and thus a lower cost.

Unfortunately, HPC applications have almost always been designed to use a fixed number of resources, and cannot exploit elasticity. Most parallel applications today have been developed using the Message Passing Interface (MPI). MPI versions 1.x did not have any support for changing the number of processes during the execution. While this changed with MPI version 2.0, this feature is not yet supported by many of the available MPI implementations. Moreover, significant effort is needed to manually change the process group, and redistribute the data to effectively use a different number of processes. Thus, existing MPI program are not designed to vary the number of processes. Adaptive MPI [11] can allow flexible load balancing across different number of nodes, but requires modification to the original MPI programs, and can incur substantial overheads when load balancing is not needed. Other existing mechanisms for making data parallel programs adaptive also do not apply to existing MPI programs [12], [13]. Similarly, the existing cloud resource provisioning frameworks cannot help in elastic execution of MPI programs [14], [15], [16].

This paper describes our initial work towards the goal of making existing MPI applications elastic for a cloud framework. Considering the limitations of the MPI implementations currently available, we support adaptation by terminating one execution and restarting a new program on a different number of instances. To enable this, we create a modified version of the original program. This version of the code allows monitoring of the progress and communication overheads, and can terminate at certain points (typically, at the end of an iteration of the outer time-step loop) while outputting the live variables at that point. Moreover, it is capable of restarting the computation, by reading the live variables, and knowing the iteration to restart with.

The components of our envisioned system include:

- An automated framework for deciding when the number of instances for execution should be scaled up or down, based on high-level considerations like a user-desired completion time or budget, monitoring of the progress of the application and communication overheads.
- A framework for modifying MPI programs to be elastic and adaptable. This is done by finding certain points in the program where the monitoring code can be added, and where variables identified to be live can be output to allow a restart from a different number of nodes. Our long-term goal is to develop a simple source-to-source transformation program to generate the two versions with these capabilities.
- A cloud-based runtime support that can enable redistributing of saved data, and support for automated resource allocation and application restart on a different number of nodes.

This paper describes the runtime framework developed for the Amazon EC2 environment to enable a proof-of-concept validation of our overall idea. Using two MPI applications, we demonstrate the feasibility of our approach, and show that our monitoring framework has a negligible overhead, and outputting, redistributing, and reading back data for adapting application can be a reasonable approach for making existing MPI applications elastic.

II. Background: Amazon Cloud

We now give some background on the Amazon Web Services (AWS), on which our work has been performed.

AWS offers many options for on-demand computing as a part of their Elastic Compute Cloud (EC2) service. EC2 nodes (*instances*) are virtual machines that can launch snapshots of systems, i.e., *images*. These images can be deployed onto various *instance types* (the underlying virtualized architecture) with varying costs depending on the instance type's capabilities.

For example, a Small EC2 Instance (`m1.small`), according to AWS[2] at the time of writing, contains 1.7 GB memory, 1 virtual core (equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor), and 160 GB disk storage. AWS also states that the Small Instance has *moderate* network I/O. In stark contrast, an Extra Large EC2 instance (`m1.xlarge`) contains 15 GB memory, 4 virtual cores with 2 EC2 Compute Units each, 1.69 TB disk storage with *high* I/O. Many other such instance types exist, also with varying costs and capabilities.

Amazon's persistent storage framework, Simple Storage Service (S3), provides a key-value store with simple ftp-

[2]AWS Instance Types, http://aws.amazon.com/ec2/instance-types

style API: `put`, `get`, `del`, etc. Typically, the unique keys are represented by a filename, and the values are themselves the data objects, i.e., files. While the objects themselves are limited to 5 GB, the number of objects that can be stored in S3 is unrestricted. Aside from the simple API, the S3 architecture has been designed to be highly reliable and available. It is furthermore very inexpensive to store data on S3.

Another feature of AWS is the availability of compute instances with three options: *on-demand instances*, *reserved instances*, and *spot instances*. An *on-demand* instance can be acquired and paid for hourly. Organizations can also make a one-time payment for *reserved instances*, and then receive a discount on the hourly use of that instance. With *spot instances*, Amazon makes its unused capacity, at any time, available through a lower (but variable) price.

III. Framework Design

This section describes the design of our dynamic resource allocation framework. We will first describe the overall goals of our framework. Then, we will explain the necessary modifications on the source code for allowing elastic execution and then explain the functionality of the various modules of the framework in detail.

A. Overall Goals

As we stated earlier, we are driven by the observation that parallel applications are most often implemented using MPI and designed to use a fixed number of processes during the execution. This is a crucial problem considering the driving features of cloud services, i.e., elasticity and the pay-as-you-go model.

The problems we address with our framework can be categorized as *dynamic resource (de)allocation* and *data distribution* among the reallocated resources during the execution. We considered two constraints that can be specified by the user. The user defined constraints are either based on a specific *time frame* within which the user would want the application to complete, or based on a *threshold value of the cost* that they are willing to spend. Clearly, it is possible that the execution cannot be finished within the specified time or the cost. Thus, these constraints are supposed to be *soft* and not *hard*, i.e, the system makes the best effort to meet the constraints. The system is also being designed to be capable of providing feedback to the user on its ability to meet these constraints.

These goals are accomplished with several runtime support modules and making modifications to the application source code. In the long-term, we expect these source code modifications to be automated through a simple source-to-source transformation tool.

B. Execution Model and Modifications to the Source Code

Input: $monitor\_iter$, $iter\_time\_req$, $curr\_iter$, $range$
Output: $true$ if solution converges, $false$ otherwise

```
{* Initiate MPI *}
data := read_data();
t0 := current_time();
While curr_iter<MAX_ITER Do
  {* Call update module *}
  rhonew := resid(data);
  If rhonew<EPS Then
      return true;
  Endif
  If (curr_iter%monitor_iter) = 0 and curr_iter ≠ 0 Then
      t1 := current_time();
      avg_time := (t1-t0)/curr_iter;
      If avg_time>(iter_time_req + range) Then
          {* Store data to a file *}
          {* Inform decision layer and expand resources *}
      Else If avg_time<(iter_time_req − range) Then
          {* Store data to a file *}
          {* Inform decision layer and shrink resources *}
      Endif
  Endif
  curr_iter := curr_iter+1;
Endwhile
return false;
```

Figure 1. *Execution Structure* Dynamic Resource Allocation for MPI Programs

Our framework specifically assumes that the target HPC application is iterative in nature, i.e., it has a *time-step loop* and the amount of work done in each iteration is approximately the same. This is a reasonable assumption for most MPI programs, so this does not limit the applicability of our framework. This assumption, however, has two important consequences. First, the start of each (or every few) iteration(s) of the time-step loop becomes a convenient point for monitoring of the progress of the application. Second, because we only consider redistribution in between iterations of the time-step loop, we can significantly decrease the *check-pointing* and *redistribution* overhead. Particularly, a general check-pointing scheme will not only be very expensive, it also does not allow redistribution of the data to restart with a different number of nodes.

In Figure 1, we show how we modified the source code of an iterative application and implemented the decision logic.

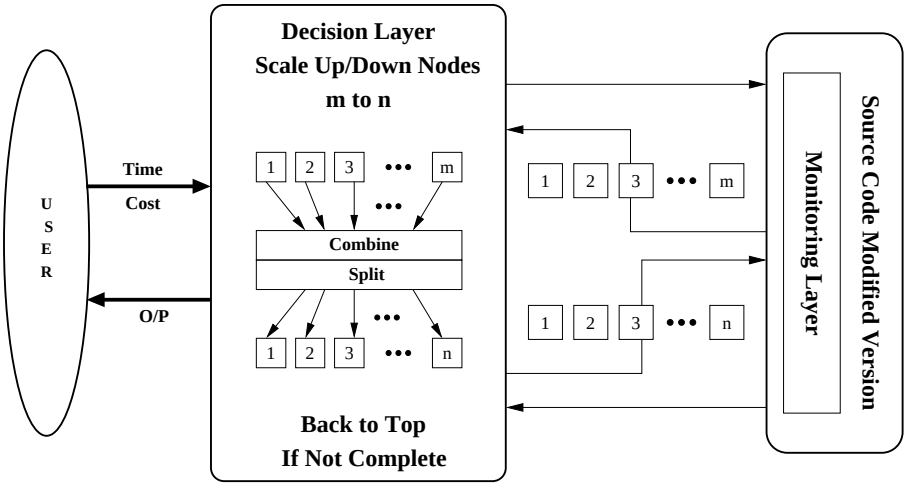

Figure 2. Execution Flow

The *monitoring interval*, the *required iteration time*, *current iteration*, and the *range* are taken as runtime parameters. The monitoring interval determines the number of iterations after which the monitoring has to be done.

Our approach assumes that the user has a fixed execution time in mind, and the system can allocate more nodes to finish the execution if needed. In practice, our framework can consider other constraints, such as the need to minimize the execution cost while staying within a time-limit. In such a case, our framework can reduce the number of nodes, provided the predicted execution time will stay within the limit.

In Fig. 1, the required iteration time is calculated by the decision layer based on the user's input. This value is used for checking the progress of the program based on the average iteration time. At any point, if there is a necessity to stop and restart the program on a new number of nodes, it is important to know how many iterations have already been completed and from which point the new set of nodes have to continue. This is given as one of the inputs, *curr_iter*, to the program. The main iteration is thus started from curr_iter. The value of this variable in the first run is zero. It is also important to make sure that reallocation of the processing nodes is not done so frequently, otherwise the overhead of restart of the nodes and redistribution of the data might not be tolerable. A control parameter called *range* in this code, is given as another input to the program. Hence, every time the average iteration time is compared with the required iteration time, it is checked if the former falls within a range around the latter. Based on the deviation from the range, the decision to change the number of nodes is made. For each iteration, after processing the matrix, the application checks the computation time if the monitoring interval has been completed. If the average iteration time is above the required range, it means that the progress is not good enough and hence scaling up of the number of nodes will be necessary. On the contrary, if it is below the range, the application can afford to run slower and instance cost can be cut down by deallocating some of the nodes as the progress is much better than expected.

In the case where scaling has to be done, a decision is made and so the data needs to be redistributed among a new number of processes. The data which is distributed among the current set of processes needs to be collected at the master node and redistributed to the new set of nodes. It is important to note here that not all data involved in the program needs to be carried over to the subsequent iterations. Only the *live* variables (and arrays) at the start of a new iteration need to be stored and redistributed. Furthermore, if the array is read-only, i.e. it has not been modified since the start of the execution, it does not need to be stored back before terminating one execution. Instead, the original dataset can be redistributed and loaded while restarting with a different set of nodes.

In our framework, each processes stores a portion of the array that needs to be collected and redistributed to a file in the local directory. Other components of our framework are informed of the decision of the monitoring layer to expand or shrink the resources. The application returns *true* if the solution is converged, so that the decision layer does not

restart it again. In case, the solution is not converged, *false* is returned which indicates that restarting and redistribution are necessary. The application is terminated and the master node collects the data files from the worker nodes and combines them . After launching the new nodes or deallocating the extra nodes based on the decision made by the monitoring layer, the decision layer in the master node splits the data and redistributes it to the new set of nodes.

The application is started again and all the nodes read the local data portions of the live arrays that were redistributed by the decision layer. The main loop is continued from this point and the monitoring layer again measures the average iteration time and makes a decision during the monitoring interval. If the need of restarting does not arise and the desired iteration time is reached, then the application continues running . Otherwise, the same procedure of writing the live data to local machines, copying them to master node and restarting the processes are repeated. Figure 2 depicts the execution flow of the system.

As we stated above, our goal is to develop a simple source-to-source transformation tool which can automatically modify the MPI source code. The main steps in the transformation will be: 1) identifying the time-step loop, which is typically the outer-most loop over the most compute-intensive components of the program, 2) finding live variables or arrays at the start of each iteration of the time-step loop, and finding the read-only variables, and 3) finding the distribution of the data used in the program.

C. Runtime Support Modules

We now describe the various components of our framework. The interaction between these components is shown in Figure 3. Also, note that the role of the monitoring layer has already been explained above, so we do not elaborate it any further. The monitoring layer keeps interacting with the decision layer, which initiates a checkpointing, to be followed by resource allocation, and then redistribution and restart.

Decision Layer: The decision layer interacts with the user to get the inputs and preferences of time and cost. It also interacts with the application program on monitoring the progress and deciding if a redistribution is needed. Most of the underlying logic has been explained in the previous subsection. In near future, we will expand the decision layer to monitor the communication overheads incurred by the application. This will allow a better prediction of the execution time of the application while using fewer or more nodes.

Resource Allocator: One of the ways our framework enables elastic execution is by transparently allocating (or deallocating) resources in the AWS environment, and configuring them for the execution of the MPI program. New instances are requested and monitored by resource allocator until they are ready. For execution of the program, an MPI cluster needs to be set-up. The MPI implementation used is *MPICH2* and the process manager employed is *mpd* (multipurpose daemon) that provides both fast startup of parallel jobs and a flexible run-time environment that supports parallel libraries. The mpd daemon needs to be started so that it can spawn the mpich processes. The main advantage of mpd is that they can start a job on several nodes in less than a second. It also has fault tolerance - it can start jobs and deliver signals even in the face of node failure. For the mpd job launcher to work correctly, each of the nodes has to open connection with two of its nearest neighbors and hence all the nodes should form a *communication ring*. The order of this ring does not matter. A Python script is used for this MPI configuration process. It runs commands to get the current state of the instances, their host-names and the external domain names. A *hosts* file containing a list of host-names has to be present in every node, as this will help mpd to initiate the execution among the processes. The nodes are configured for a password-less login from the master node and the host file is copied over from the master node to the slave nodes. Then, mpd is booted on the master node and all the other nodes join the ring. Once this is completed, the MPI environment is set and ready to run parallel jobs on multiple instances. The binary code has to be present on all nodes before execution and is hence copied from the master node to the other nodes by the resource allocator layer. The application can now be launched by giving the required iteration time as a runtime parameter.

In case a reallocation or deallocation is needed, the data portions residing on the nodes need to be collected at the master node, combined together, and then redivided into the new number of nodes. This is illustrated in Figure 2 where the number of nodes is being scaled from m to n. The redivided data is then transferred to the new group of instances, which read the data and continue working on them from the point where they were terminated. Thus the whole process is repeated until the program finishes its execution.

Check-pointing and Data Redistribution Module: Data collection and redistribution depend on the type of application and the type of data. Multiple design options were considered for this, in view of the support available on AWS.

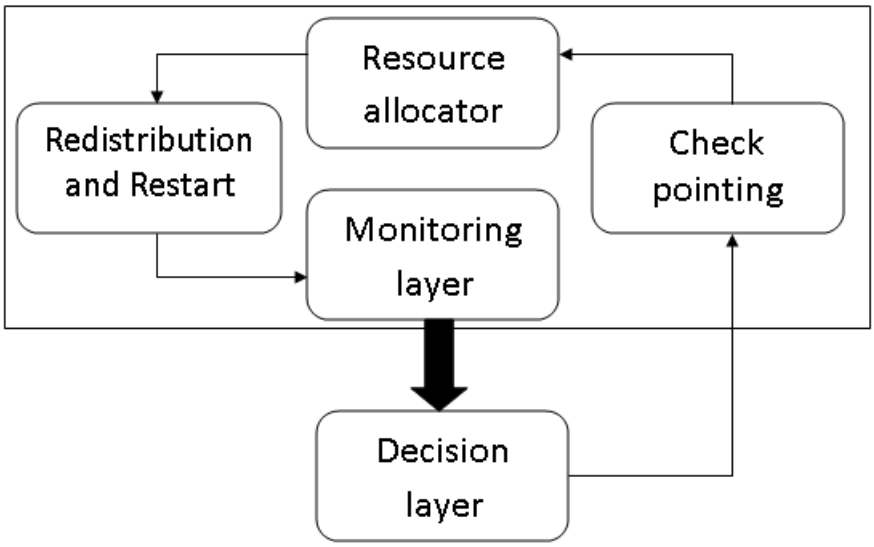

Figure 3. Components of Our Framework

Amazon S3 is a storage service provided by Amazon that can be accessed by the EC2 instances. For arrays that are not modified at each iteration can be stored in small sized chunks in S3. Later, during a node launch, each of the nodes can download the chunks of data required by them and continue with their computation. This design is very efficient for unaltered data as it saves the overhead of writing to and reading from a file. For variables that are modified in each iteration, file writes and reads are used to write and read the data. The remote file copy command, *scp*, is used to transfer files to the master node, and again for copying the new pieces of each node.

Combining and redistributing data require the knowledge of how the original dataset was divided among the processes (e.g.: row-wise, column-wise, two-dimensional etc.). Currently, this information is provided as an annotation to our framework. In the future, our source-to-source transformation tool can be used to automatically extract this information. Based on the data distribution information and the number of initial and final instances, our redistribution module can generate the portions of the dataset each node will need.

Note that our current design performs aggregation and redistribution of data centrally on a single node. This can easily be a bottleneck if the initial and/or the final number of instances is quite large. In the future, we will implement more decentralized array redistribution schemes [17].

IV. Applications and Experimental Results

In this section, we evaluate our approach and framework performance with 2 applications. We demonstrate the feasibility of our approach and show the performance of the runtime modules we have implemented.

Our experiments were conducted using 4, 8, and 16 Amazon EC2 small instances. The processing nodes communicate using MPICH2. We evaluated our framework with two MPI applications: *Jacobi* and *Conjugate Gradient* (CG). Jacobi is a widely used iterative method which calculates the eigenvectors and eigenvalues of a symmetric matrix. Since the Jacobi application processes and manipulates the matrix in each iteration, the updated matrix needs to be collected and redistributed among the compute nodes in the case of adaptation. The data redistribution is done using parallel data transfer calls, and the overhead of the data transmission time is significantly reduced. The NAS CG is a benchmark application that calculates the largest eigenvalue of a sparse, symmetric, positive definite matrix, using the inverse iteration method. A specific number of outer iterations is used for finding the eigenvalue estimates and the linear system is solved in every outer iteration. The dominant part of the CG benchmark is a matrix vector multiplication. The matrix is a sparse one and stored in compressed row format. This matrix is not manipulated during the program execution, thus it can be stored in a shared storage unit, i.e. the Amazon S3. The matrix is divided into chunks, and these chunks can be distributed, retrieved, and processed by the allocated EC2

instances.

The matrix processed for our Jacobi execution had 9K $\times$ 9K double values ($\sim$618 MB). This matrix needs to be collected and redistributed in the case of compute instance reallocation. For CG, the matrix has 150K $\times$ 150K double values. However, only the vector needs to be redistributed, and its size is 1.14 MB.

Table I
JACOBI APPLICATION WITHOUT SCALING THE RESOURCES

No. Nodes	W/O Redist. (sec)	W/ Redist. (sec)	MPI Config. (sec)	Data Movement (sec)	Overhead (%)
4	2810	2850	71	3	0.01
8	1649	1720	89	2.5	0.04
16	1001	1087	87	3.6	0.09

Our first experiment involved executing Jacobi for 1000 iterations, and redistributing once (after 500 iterations). To be able to evaluate the redistribution and restart overhead, we "redistributed" the execution with the same number of nodes. This version included overheads of MPI configuration, data redistribution, copying of the source files to all nodes, and the actual program restart. Table I presents the execution times and the major overheads associated with redistribution of data, particularly, the MPI configuration and data movement costs. The overheads of our system range from 0.01% to 0.1% and show small increments with increasing number of compute instances. The main reason for the overhead is due to the MPI configuration time. It consists of collecting the host information of the newly initiated computing instances and preparing the configuration file that lets MPI daemon set up the MPI groups. This process can introduce small overheads, however the larger computation times are expected to further dominate these times. The parallel data redistribution effectively transfers the updated data, and minimizes the data transmission time.

We can also see reasonable speedups with increasing number of instances. Since the communication tends to be relatively slow between Amazon EC2 instances, the speedups are not close to linear. As stated above, we are redistributing once between 1000 iterations. Our overheads will be relatively higher if we redistribute more often. But, even if the redistribution was done after every 50 iterations, based on the above experiments, the overheads will still be less than 2%.

In Table II, we show how our runtime modules perform when the system actually scales up and down. The *Overhead* column now shows the *estimated overheads*, considering a projected execution time derived from the related *W/O Redist.* columns of Table I. As we saw in the previous experiments, the overheads stay very low (0.02-0.05% overhead for redistributing once in 1000 iterations). The MPI configuration is the dominating factor for the overall overhead and it increases while the numbers of final nodes increase for the same starting nodes configuration.

Table II
JACOBI APPLICATION WITH SCALING THE RESOURCES

Starting Nodes	Final Nodes	MPI Config. (sec)	Data Movement (sec)	Total (sec)	Overhead (%)
4	8	81	3	2301	0.03
4	16	84	3	1998	0.05
8	4	80	3	2267	0.02
8	16	95	3.8	1386	0.04
16	4	99	3.5	2004	0.05
16	8	97	3	1390	0.05

Table III
CG APPLICATION WITHOUT SCALING THE RESOURCES

No. Nodes	W/O Redist. (sec)	W/ Redist. (sec)	MPI Config. (sec)	Data Movement (sec)	Overhead (%)
4	834	879	25	2.5	0.05
8	997	980	58	3	0
16	1030	1105	96	2.7	0.07

The same set of experiments were reported with CG and the results are presented in Tables III and IV. Redistribution was now performed once during 75 iterations of the application. The first observation from these results is that unlike Jacobi, the performance of CG does not improve with an increasing number of nodes. This is because CG is more communication-intensive. We expect that improvements in communication performance in clouds will help speedup an application like CG in the future (Amazon Cluster Compute Instance apparently has improved performance, though we have not yet experimented with it).

Table IV
CG APPLICATION WITH SCALING THE RESOURCES

Starting Nodes	Final Nodes	MPI Config. (sec)	Data Movement (sec)	Total (sec)	Overhead (%)
4	8	43	3	930	0.02
4	16	60	3	999	0.07
8	4	40	4	942	0.03
8	16	81	3	1060	0.05
16	4	58	3	1003	0.08
16	8	82	3	1080	0.07

Table IV shows the execution times when the system scales up and down. The overhead of the system increases with the increasing number of the compute nodes. The short computation time, the nodes that need to be configured and the data redistribution result in extra overhead with large number of compute instances. However, the overheads are still quite low.

V. Conclusions

This paper has described our initial work towards the goal of making existing MPI applications elastic for a cloud framework. We have proposed an overall approach and have developed several runtime modules. Our approach is based on terminating one execution and starting another with a different number of nodes. Our evaluation with 2 applications has shown that this has small overheads, and elastic execution of MPI programs in cloud environments is feasible.

References

[1] C. Vecchiola, S. Pandey, and R. Buyya, "High-performance cloud computing: A view of scientific applications," *Parallel Architectures, Algorithms, and Networks, International Symposium on*, vol. 0, pp. 4–16, 2009.

[2] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: the montage example," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[3] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D. P. Anderson, "Cost-benefit analysis of cloud computing versus desktop grids," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[4] J. Li*, et al*., "escience in the cloud: A modis satellite data reprojection and reduction pipeline in the windows azure platform," in *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2010.

[5] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon s3 for science grids: a viable solution?" in *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*. New York, NY, USA: ACM, 2008, pp. 55–64.

[6] J. Weissman and S. Ramakrishnan, "Using proxies to accelerate cloud applications," in *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[7] S. Das, D. Agrawal, and A. E. Abbadi, "ElasTraS: An Elastic Transactional Data Store in the Cloud," in *Proceedings of Workshop on Hot Topics in Cloud (HotCloud)*, 2009.

[8] H. Lim, S. Babu, and J. Chase, "Automated Control for Elastic Storage," in *Proceedings of International Conference on Autonomic Computing (ICAC)*, Jun. 2010.

[9] D. Chiu, A. Shetty, and G. Agrawal, "Elastic cloud caches for accelerating service-oriented computations," in *Proceedings of SC*, 2010.

[10] M. Mao, J. Li, and M. Humphrey, "Cloud Auto-Scaling with Deadline and Budget Constraints," in *Proceedings of GRID 2010*, Oct. 2010.

[11] C. Huang, G. Zheng, L. V. Kalé, and S. Kumar, "Performance evaluation of adaptive mpi," in *PPOPP*, J. Torrellas and S. Chatterjee, Eds. ACM, 2006, pp. 12–21.

[12] G. Edjlali, G. Agrawal, A. Sussman, and J. Saltz, "Data parallel programming in an adaptive environment," in *Proceedings of the Ninth International Parallel Processing Symposium*. IEEE Computer Society Press, Apr. 1995, pp. 827–832.

[13] J. B. Weissman, "Predicting the cost and benefit of adapting data parallel applications in clusters," *J. Parallel Distrib. Comput.*, vol. 62, no. 8, pp. 1248–1271, 2002.

[14] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling, "Scientific workflow applications on amazon ec2," *CoRR*, vol. abs/1005.2718, 2010.

[15] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *CCGRID*, F. Cappello, C.-L. Wang, and R. Buyya, Eds. IEEE Computer Society, 2009, pp. 124–131.

[16] L. Rodero-Merino, L. M. V. Gonzalez, V. Gil, F. Galán, J. Fontán, R. S. Montero, and I. M. Llorente, "From infrastructure delivery to service management in clouds," *Future Generation Comp. Syst.*, vol. 26, no. 8, pp. 1226–1240, 2010.

[17] R. Thakur, A. Choudhary, and J. Ramanujam, "Efficient algorithms for array redistribution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 6, pp. 587–594, Jun. 1996.