An Autonomic Framework for Time and Cost Driven Execution of MPI Programs on Cloud Environments

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

***Abstract*—This paper gives an overview of a framework for making existing MPI applications elastic, and executing them with user-specified time and cost constraints in 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 key component of our system is a decision layer. Based on the time and cost constraints, this layer decides whether to use fewer or a larger number of instances for the applications, and when appropriate, chooses to migrate the application to a different type of instance. Among other factors, the decision layer also models the redistribution costs.**

I. Introduction and Overview

A. Motivation

Multiple cloud providers are now specifically targeting HPC users and applications. 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. While 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. 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.

We are developing a framework for making existing MPI applications elastic, and executing them with user-specified time and cost constraints in 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.

B. Overview of the Functionality

We consider 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.

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 assumption 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.

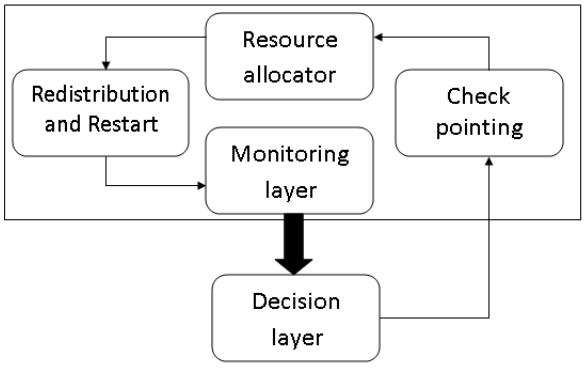

Fig. 1. Components of Our Framework

In our framework, each process 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.

II. Decision Layer

As stated in the previous section, we focus on iterative applications that do a constant amount of work at each iteration. The time taken per iteration, which is communicated by the application to the decision layer, is, therefore, used to determine the progress of the application. In addition to this, the communication time between nodes is also taken into consideration, as an increase in the communication time impacts scalability. For communication-intensive applications, an increase in the number of nodes will certainly not result in linear or close to linear scalability. If we need to meet a time constraint, it is desirable to switch to more powerful processing nodes (such as a large instances in EC2), rather than increasing the number of instances. On the other hand, if cost is a constraint, one can scale down to fewer nodes to improve cost-effectiveness. This is because using half as many nodes will likely not slow down the application by a factor of two, and therefore, the product of time and node count will be lower.

We consider two situations separately: 1) the user might have a fixed *time constraint*, and 2) the user might have a fixed *cost constraint*. We briefly discuss the main steps for first case here.

Suppose the application is executing on n_c nodes. The time elapsed is t_{tn}. Of the total of it_{tot} iterations over which the application has to be executed, it_{rem} are remaining, i.e., $it_{tot} - it_{rem}$ iterations have been executed. Therefore, the time spent per iteration with n_c nodes is $t_{pi} = \frac{t_{tn}}{it_{tot} - it_{rem}}$ The time taken for one iteration on one node (assuming perfect scalability) is given by $n_c \times t_{pi}$. Time taken for remaining iterations on the new number of nodes, n_p, is given by $\frac{n_c \times t_{pi} \times it_{rem}}{n_p}$. Since this value should ideally be $t_c - t_{tn}$, the number of nodes required can be calculated as: $n_p = \frac{n_c \times t_{pi} \times it_{rem}}{t_c - t_{tn}}$. Note, however, that the this analysis is based on the assumption of *perfect scalability*, which is rarely true for applications. To some extent, this can be overcome by the fact that the above analysis is applied repeatedly, and therefore, additional nodes can be added later to meet the constraint. But, we do not use the above model at all for *communication-intensive* applications. Instead, these applications are transferred to a cluster built of *large* instances in EC2 which are likely to allow speedups even for communication-intensive applications.

III. Experimental Results

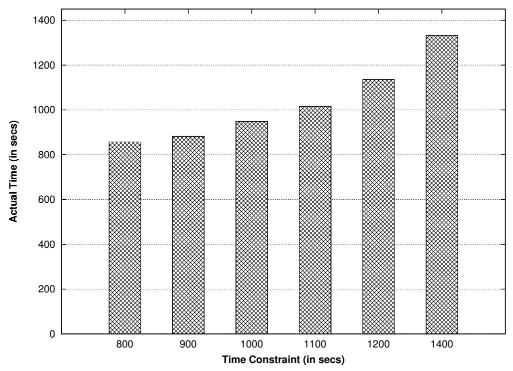

Fig. 2. Time Constraints vs Actual Time - Jacobi

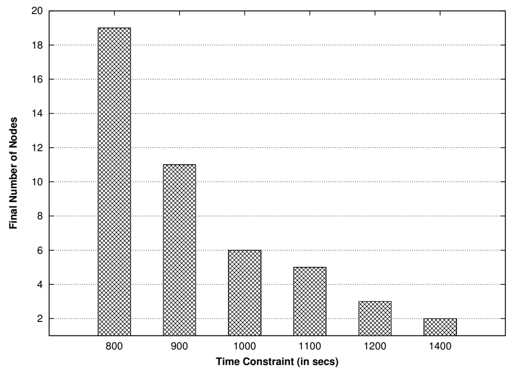

Fig. 3. Time Constraint vs Nodecount - Jacobi

We briefly describe the results obtained from Jacobi, executed with different time constraints. Since the amount of time spent on interprocess communication is relatively small, this application can be scaled by adding additional instances. Figure 2 shows the application execution time obtained by our framework in response to different time constraints. It can be observed that the actual time taken is less than the constraint specified, for time constraint values other than 800 seconds. In this case, the system is not able to meet the requirements even with maximum number of nodes it could allocate (which was 19 nodes in our experiments). Figure 3 shows the final node count values for different time constraints. We can see that the framework does not overallocate nodes, instead, it uses an appropriate number of instances to just meet the time constraint. A lower the time constraints, a higher number of instances are allocated. For the time constraint of 800 seconds, the system uses 19 nodes , which is the maximum possible in our experiments. For each higher time constraint, the decision layer allocates fewer nodes so that the cost is minimized as much as possible, at the same time keeping the total process time within the given limit.