Cloud Friendly Load Balancing for HPC Applications: Preliminary Work

Osman Sarood, Abhishek Gupta and Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
sarood1, gupta59@illinois.edu

***Abstract*—Cloud computing is increasingly being explored as a cost effective alternative and addition to supercomputers for some High Performance Computing (HPC) applications. However, dynamic environment and interference by other virtual machines are some of the factors which prevent efficient execution of HPC applications in cloud.**

Through this research, we leverage a message driven adaptive runtime system to develop techniques that reduce the gap between application performance on cloud and supercomputers. Our scheme uses object migration to achieve load balance for tightly coupled parallel applications executing in virtualized environments that suffer from interfering jobs. While restoring load balance, it not only reduces the timing penalty caused by interfering jobs, but also reduces energy consumption significantly. With experimental evaluation using benchmarks and a real HPC application, we demonstrate that our scheme reduces the timing penalty and energy overhead associated with interfering jobs by at least 50%.

I. Introduction

Cloud computing has emerged as a promising solution to the needs of High Performance Computing (HPC) application users. Cloud benefits from economy of scale and hence has the potential for being a cost-effective and timely solution to academic and commercial HPC users. Further, cloud offers the advantages of elasticity, eliminates risks caused by under-provisioning as well as avoids under utilization of resources resulting from over- provisioning. Also, virtualization support in cloud can provide better flexibility, customization, security, isolation, migration and resource control to HPC community.

However, clouds have not been adopted by HPC community to a large extent. A primary reason for this is that clouds have been traditionally designed to be used in a multi-tenant environment, where multiple virtual machines share physical cores, while maintaining job isolation. However, virtualization is shown to adversely impact HPC applications by degrading network performance and acting as a source of interference to tightly coupled parallel processes [1], [2].

Past research [1]–[3] on HPC in cloud has primarily focused on evaluation of MPI applications and have been mostly pessimistic. To the best of our knowledge, there have been no efforts to improve HPC application performance using a parallel runtime adapted to cloud. With this as motivation, the primary question that we address through this research is the following: Can we customize a parallel adaptive runtime system to improve application performance in a virtualized environment?

Our hypothesis is that techniques leveraging an adaptive parallel runtime system can help in mitigating some problems caused by virtualized environment in cloud. To validate this, we explore the adaptation of Charm++ [4], [5] parallel runtime system to virtualized environment and show the extent to which the performance gap induced by virtualization is reduced. Load balancing schemes that use CHARM++ runtime system have proved to be beneficial in reducing execution time and energy consumption [6]. In this paper, we present a cloud friendly load balancing scheme for HPC applications to improve application performance. Existing MPI applications can leverage the benefits of our approach using Adaptive MPI (AMPI) [5]. Our work not only improves the execution time for virtualized environments running interfering jobs, but also results in considerable reduction in energy consumption. Given the expansion of modern day data centers, reducing energy consumption has become one of the most coveted goals for data centers. Our results look promising as we were able to reduce the execution time penalty and energy overhead by at least 50% for all the applications.

The remainder of this paper is organized as follows: Related work is discussed in section II. Section III provides background on Charm++ and the concept of periodic load balancing. Section IV discusses our approach. We present the results in Section V and give concluding remarks along with an outline of future work in Section VI.

II. Related Work

The first study on benchmarking the performance of HPC applications in cloud was conducted by Walker [1] who used NPB benchmarks on Amazon EC2 [7]. Many others [8] performed similar evaluation of Amazon EC2 for HPC applications using benchmarks. He et al. [9] and Ekanayake et al. [10] used real applications in addition to running classical benchmarks and compared the results with that from dedicated HPC systems.

A comprehensive evaluation was performed under the US Department of Energy's (DoE) Magellan project [2], [11], where the researchers compared conventional HPC platforms

to Amazon EC2 using real applications representative of the workload at a typical DoE supercomputing center. One of the findings of that study was that HPC applications achieve poor performance and suffer from significant variability when run on commercial clouds. Brunner et al. [12] proposed a load balancing scheme to deal with interfering jobs in the context of workstations. Our scheme differs from them as it uses a refined load balancing algorithm that achieves load balance while minimizing task migrations. Moreover, we present detail analysis of three different applications along with experimental results to show how our scheme reduces energy consumption.

III. Charm++ and Load Balancing

Charm++ [4], [5] is an object oriented parallel programming system, which is used by many scientific and engineering applications such as NAMD [13]. Here, the programmer decomposes the application into large number of medium grained pieces, which we call charm++ objects or *chares*. Each object has a state and a set of functions for dealing with incoming messages. Typically the number of objects needs to be more than the number of available processors for efficient execution. These objects are mapped by the runtime system onto available processors. Objects have methods associated with them, called entry methods, which execute only when invoked from a local or remote processors through messages. This message driven execution and overdecomposition results in automatic overlap of computation and communication and helps in hiding the network latency.

MPI programs can leverage the capabilities of Charm++ runtime system using the adaptive implementation of MPI (AMPI [5]) where user specifies large number of MPI processes implemented as user-level threads by the runtime.

The object (or thread) based overdecompositon facilitates dynamic load balancing, a concept central to our work. The load balancing system automatically instruments the runtime system and monitors the computation time spent in each function (task). Using this data, it then periodically remaps objects to processors as the application execution progresses using an existing load balancing strategy while assuming that future loads will be almost the same as measured loads (*principle of persistence*). Programmers can add their own application or platform specific strategy to the load balancing framework. These capabilities of the load balancing system offered by Charm++ facilitate the implementation of our techniques presented in the next section.

IV. Approach

In the context of cloud, where a HPC application is executed in a virtualized environment and the execution environment varies from run to run based on extraneous factors such as VM to physical machine mapping and interference by co-located VMs, a static load allocation to parallel processes cannot achieve good performance. Moreover, existing load balancing strategies consider only the load imbalance internal to application and not the imbalance introduced by extraneous factors, which can occur even if all the processes are allocated equal work as if they were executing in a homogeneous and isolated environment. We explore the problem of restoring load balance taking into account external factors such as other applications that interfere because they are running on co-located VMs i.e. VMs on the same CPU.

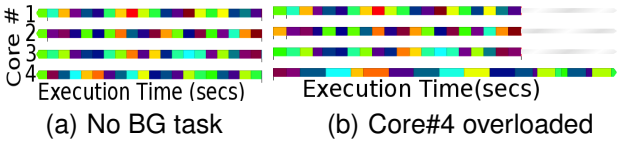

Figure 1: BG task on Core#4 disturbing load balance

In such cases, application performance can severely degrade, especially for tightly coupled iterative applications where a single overloaded processor can delay the entire application. To demonstrate it, we conducted a simple experiment where we ran Wave2D, a tightly coupled 5-point stencil application, on 4-cores of a single node machine. To study the impact of interference, we ran a single-core job of the same application on Core#4 after the 4-core run has executed for a few iterations. Figure 1 shows two iterations of that run with each horizontal line representing the timeline for a core. Different colors represent time spent while executing tasks of the 4-core run whereas the grayish-white parts (towards the right side of Figure 1(b)) represent idle time. The first iteration is considerably faster (smaller) than the other iteration as there is no interference from the single core job. After the first iteration, the single core run starts and results in Core#4 dividing its time between the 1-core interfering task and the tasks of the 4-core run. The Projections timelines [14] tool we used includes the time spent executing the 1-core run in the time spent for executing tasks of 4-core run because it cannot identify when the operating system switches context. This gets reflected in the fact that some of the tasks hosted on Core#4 take significantly longer time to execute than others (longer bars in Figure 1(b)). Due to this CPU sharing, it takes longer for Core#4 to now finish the same amount of work for the 4-core run that was equally distributed amongst all 4 cores at the start. The tightly coupled nature of the application means that no other core can start the next iteration unless all cores are done executing the current iteration which results in cores 1-3 waiting for Core#4 to complete its iteration. The length of timelines for both the iterations in Figure 1 represent the execution time for each iteration which is much larger for the later iteration which has interference. Since the distribution of such interference is fairly random on a shared-

resource based platform, we need a dynamic load balancing scheme which continuously monitors the loads for each core and reacts to any imbalance.

Keeping in view these requirements, we now propose a load balancing scheme with the objective of reducing the execution time for parallel iterative applications that encounter interfering tasks in a cloud setting. Our scheme achieves its objective by using task migration which enables the runtime to balance loads on all the cores. It is based on measuring the time spent on each task and predicting future load for a task based on the time it took in the recently completed iteration. However, to incorporate the interference from different VMs on the same node, we need to instrument the load external to the application under consideration. We will refer to that load as the *background load* for rest of the paper. To keep the application in load balanced stage, we need to ensure that all the cores have load close to the average load (T_{avg}) defined as:

$$T_{avg} = \sum_{p=1}^{P}(\sum_{i=1}^{N_p} t_i^p + O_p)/P \quad (1)$$

where P is the total number of cores the application is using, N_p is the total number of tasks assigned to core p, t_i^p is the CPU time consumed by task i hosted on core p, and O_p is the total background load for core p. The task CPU time are fetched from CHARM++ load balancing database whereas the O_p is calculated as:

$$O_p = T_{lb} - \sum_{i=1}^{N_p} t_i^p - t_{idle}^p \quad (2)$$

where T_{lb} is the wall clock time between two load balancing steps, t_i^p is the CPU time consumed by task i hosted on core p and t_{idle}^p is the idle time spent by core p since the last load balancing step. We are currently extracting t_{idle}^p from the `/proc/stat` file. The objective of our scheme is to keep the load for each core close to the average load after taking into account the background load. Hence, we can formulate the problem as:

$$\forall p \in P, (\sum_{i=1}^{N_p} t_i^p + O_p) - T_{avg} < \epsilon \quad (3)$$

where P is the set of all cores, t_i^p is the CPU time consumed by task i hosted on core p, O_p is the total background time for core p, and ϵ is the deviation from the average load that the cloud operator is willing to allow.

In order to achieve the above mentioned objective, we use Charm++ dynamic load balancing framework which in turn uses object migration. The broad level idea is to do periodic checks on the state of load balance and migrate objects from overloaded cores to underloaded cores such that Equation 3 is satisfied. Algorithm 1 summarizes our approach with the definition for each variable given in Table I. Our approach starts with categorizing each core as overloaded/underloaded (lines 2-7). In order to categorize a core it compares the sum of CPU time of tasks assigned to it and the background load to the average load for the entire application i.e. T_{avg} (lines 17-27). If current core load is greater than T_{avg} by a value greater than ϵ, we mark that core as an overloaded core and add it to the $overHeap$ (line 4). Similarly, if the core load is less than T_{avg} by a value greater than ϵ (line 34), we categorize it as an underloaded core and add it to the $underSet$ (line 6). Once we have built the underloaded set and overloaded heap of cores, we have to transfer tasks from cores in the overloaded heap i.e. $overHeap$, to the underloaded cores i.e. $underSet$, such that there are no cores left in the $overHeap$ (lines 10-15).

Table I: Description for variables used in Algorithm 1

Variable	Description
p	number of cores
T_{avg}	average execution time per core
t_i^p	CPU time of task i assigned to core p
m_i^k	core number to which task i is assigned during step k
$overHeap$	heap of overloaded cores
O_p	background load for core p
$underSet$	set of underloaded cores

In order to decide the new task mapping to achieve load balance, our scheme starts by popping the most overloaded core from $overHeap$ i.e. $donor$ (line 11). It then tries to find an underloaded core from $underSet$ such that we can transfer the biggest task from $donor$ (line 11) to that underloaded core. While doing so our approach ensures that we only pick an underloaded core that does not get overloaded after the task transfer has taken place (lines 12). After the $bestTask$ and $bestCore$ are determined, we update the mapping of the task (line 13). Once it is done we update the loads of both the $donor$ and $bestCore$ and update the $overHeap$ and $underSet$ with these new load values (line 14). This process is repeated until the $overHeap$ is empty i.e. no overloaded cores are left.

V. PERFORMANCE RESULTS

We expect our load balancing scheme to help in narrowing the performance gap between a supercomputer and cloud for iterative applications. To evaluate the performance of our scheme, we use 8 nodes (32 cores) of a testbed located at Department of Computer Science at the University of Illinois Urbana Champaign. Each node is a single socket machine equipped with a quad core Intel Xeon X3430 processor. This testbed can report power consumption readings for each node on a per second basis. We investigate the effectiveness of our scheme, using three different CHARM++ applications. The first one is Wave2D described earlier in Section IV. The second one is Jacobi2D, which is a canonical benchmark that iteratively applies a 5-point stencil over a 2D grid of points.

Algorithm 1 Refinement Load Balancing for VM Interference

On Master core on each load balance step
for $p \in [1, P]$ **do**
  if $isHeavy(p)$ **then**
    $overHeap.add(p)$
  else if $isLight(p)$ **then**
    $underSet.add(p)$
  end if
end for
createOverHeapAndUnderSet()
while $overHeap$ NOT NULL **do**
  $donor$ = deleteMaxHeap($overHeap$)
  $(bestTask, bestCore)$ = getBestCoreAndTask($donor, underSet$)
  $m^k_{bestTask} = bestCore$
  updateHeapAndSet()
end while

procedure isHeavy(p)
    for $i \in [1, N_p]$
      $totalTime+ = t^p_i$
    end for
    $totalTime+ = O_p$
    if $totalTime - T_{avg} > \epsilon$
      return **true**
    else
      return **false**
    end if
end procedure

procedure isLight(p)
    for $i \in [1, N_p]$
      $totalTime+ = t^p_i$
    end for
    $totalTime+ = O_p$
    if $T_{avg} - totalTime > \epsilon$
      return **true**
    else
      return **false**
    end if
end procedure

The third application is a classical molecular dynamics code called Mol3D.

We now compare application performance with (referred as *LB*) and without (referred as *noLB*) load balancing on the basis of execution time and energy consumption. In our analysis we also consider the degradation (increase in execution time) in background load. In order to create interference with our parallel runs we run a 2-core job of Wave2D as the background load on two of the cores allocated to application under test. In our analysis, we mainly want to optimize larger parallel run with secondary importance to the 2-core background job. For each experiment, we start the parallel run along with the background load. In order to make it a fair comparison, we kept the background load exactly the same for all our experiments i.e. running wave2D application on 2 cores. We report execution time measurements using the wall clock times which includes the time taken for object migration. We also collect total power consumption for the run using actual power meters. All the results shown are averages over three similar runs and represent actual measurements without any simulation or estimation. In the remaining part of this section, we look at the effects of our load balancing on execution time and energy consumption.

A. Impact on execution time

Figure 2 shows the reduction in execution time achieved by our load balancing scheme. It shows the timing penalty for both the parallel job and the background load (referred as *BG* in figures) for each application after running the applications on different number of cores. Timing penalty for the parallel job expresses the additional time it takes to run the parallel job with interference from the 2-core job as a percentage of time taken by the same run without any interference. Similarly, in case of the background load, timing penalty refers to the additional time it takes to run the same background load when there is nothing else interfering with it i.e. no other job running on those cores. It is evident from Figure 2 that our load balancing scheme reduces the timing penalty significantly as compared to equivalent run without load balancing for all applications.

The tight coupling in all three applications results in very high timing penalties for the cases where we do not use load balancing (*noLB* bars in Figure 2). Although CPU was almost equally shared for most cases, we saw a significant preference to the background load in the case of Mol3D. Hence, the timing penalty for Mol3D for the *noLB* case was very high (up to 400%). But our load balancing scheme reduces the timing penalty significantly (Figure 2(c)) to less than 25% for any number of cores.

Our results show that our load balancing scheme helps reducing the timing penalty as we increase the number of cores for all applications. It is due to the fact that an increase in the number of cores results in more cores to which the work of the overloaded core can be distributed. Since our background load is only running on two cores, running on larger number of cores implies distributing the work of the 2 overloaded cores to an increasing number of under utilized cores.

Other than showing significant reduction in the timing penalty for the three CHARM++ applications, Figure 2 also shows its effectiveness in reducing the timing penalty for the interfering 2-core job. Our scheme significantly reduces the timing penalty for the background load (referred as *BG*

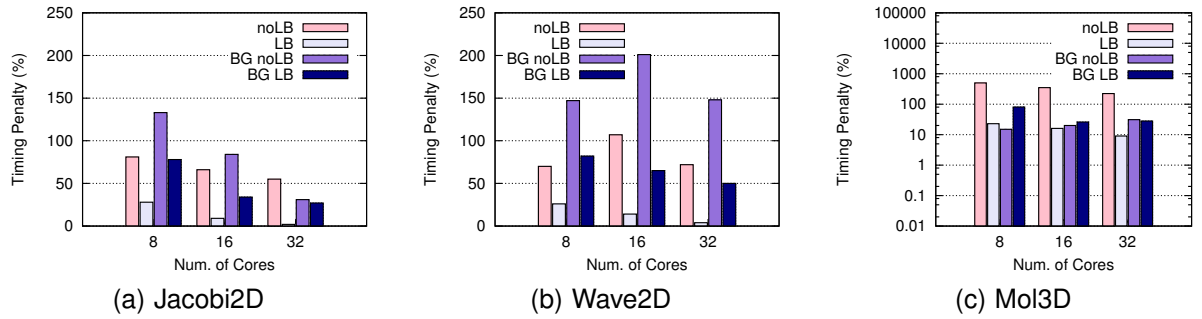

Figure 2: Effect of load balancing in execution time

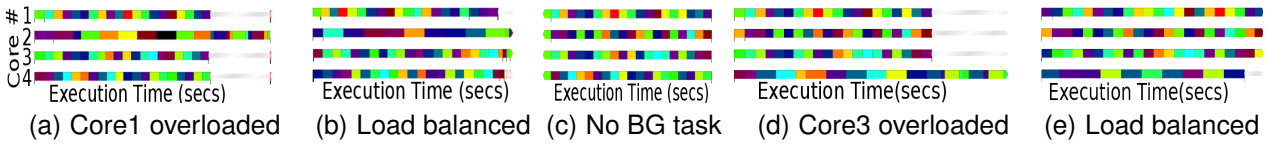

Figure 3: Cores timeline showing effect of load balancing for a 4 core run with background load on Core1 and then Core3

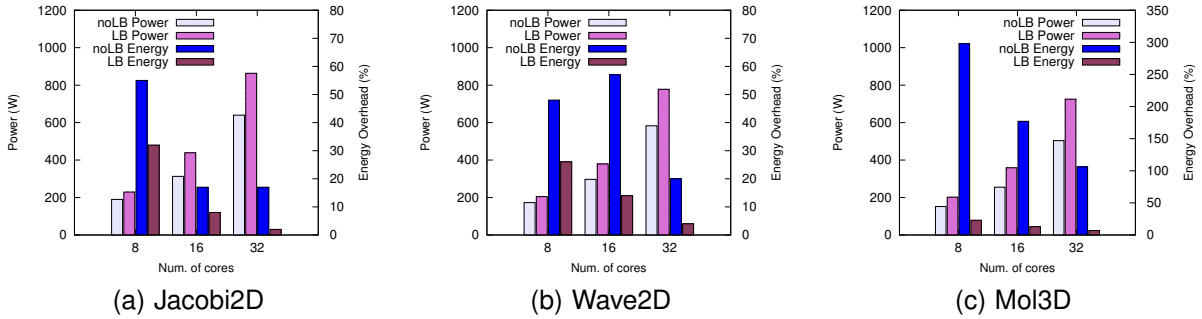

Figure 4: Effect of load balancing in energy and power consumption

LB) in case of Jacobi2D and Wave2D. However, in case of Mol3D, the runs without load balancing end up having a lower timing penalty for background load. This is due to the large proportion of CPU time that the operating system is allocating to the background load, resulting in small timing penalties for the background load. However, the same preference to the background load results in very high timing penalty for the application under consideration.

Considering the nature of cloud data centers, a successful load balancing mechanism should be robust to dynamic changes in interfering tasks as they might come and go randomly. Figure 3 demonstrates core timelines for a 4-core run of Wave2D where we experimented by putting background load on two different cores at different times. To start with, Core#1 had an interfering job running on it (Figure 3(a)) which caused load imbalance and hence a greater time per iteration. Once the application hits the load balancing mark, it redistributes the loads according to Algorithm 1 and restores load balance (smaller length for timeline in Figure 3(b)) by migrating tasks from Core1 to other cores. The iteration shown in Figure 3(c) represents a period where the interfering task ended and our load balancer migrated some tasks back to Core1 as it now can handle load equal to other cores. But after that another interfering task started at Core3 which caused load imbalance again (Figure 3(d)). Once again, when the application comes to the load balancing step, our load balancer detects the load imbalance and migrates the tasks away from Core3 to the other 3 cores which restores load balance (Figure 3(e)).

B. Impact on power and energy consumption

After establishing the benefits of our load balancing scheme to save execution time, we now analyze its impact on power and energy consumptions. Figure 4 shows the average power consumption for both cases i.e. with and without load balancer. For all three cases, load balanced runs end up consuming more power as compared to the

runs where we do not use load balancing. It is because load balancing removes the idle time for most cores and increases their CPU utilization. Since dynamic power consumption of a processor forms a significant part of machine power consumption, and the fact that it is directly related to the amount of computation the processor is doing, we see much higher power consumption for the load balanced run. But does this mean we end up increasing energy consumption as well?

Energy consumption is the product of execution time and power draw. Although power increases for load balanced runs as compared to the same runs without load balancing, the decrease in execution time resulting from load balancing is far greater than it. Moreover, even when the cores are sitting idle waiting for the overloaded core to finish their tasks in case of no load balancing, their power consumption is not zero. Although the processor cores are consuming very little power when idle, the rest of the machine does consume some power usually referred to as *base* power. *Base* power can be a significant proportion of the total machine power consumption. For our testbed, the base power is 40W per node which is significant considering the maximum power consumption of 107W per node when running a highly computation intensive workload. The high base power coupled with significant reduction in execution time, enables our scheme to reduce energy consumption (in comparison to noLB runs) as shown by Figure 4 which plots the normalized energy consumption for each application. The normalization was done with respect to a base run where the application ran without any interference from the background load. This interference results in additional energy consumption i.e. energy overhead. However, as Figure 4 shows, our scheme successfully minimizes this overhead.

VI. Conclusions and future work

In this paper, we presented a load balancing technique which accounts for background loads and uses object migration to restore load balance. Experimental results showed that we were able to reduce timing penalty and energy consumption by more than 50% compared to the case where there is no load balancing. In future, we plan to evaluate our technique on a public cloud where multiple VMs share CPU resources. Due to the inferior performance of network, we also plan to explore a strategy where load balancing decisions are performed every time a load balancer is invoked, however, data migration is performed only if we expect gains that can offset the cost of migration.

References

[1] E. Walker, "Benchmarking Amazon EC2 for High-Performance Scientific Computing," *LOGIN*, pp. 18–23, 2008.

[2] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud," in *CloudCom'10*, 2010.

[3] A. Gupta and D. Milojicic, "Evaluation of HPC Applications on Cloud," in *Best Student Paper, Open Cirrus Summit*, 2011.

[4] L. Kale and S. Krishnan, "Charm++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[5] L. V. Kale and G. Zheng, "Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects," in *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.

[6] O. Sarood and L. V. Kale, "A 'cool' load balancer for parallel applications," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 21:1–21:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063412

[7] "Amazon Elastic Compute Cloud (Amazon EC2)," http://aws.amazon.com/ec2.

[8] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, pp. 931–945, June 2011.

[9] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn, "Case Study for Running HPC Applications in Public Clouds," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 395–401.

[10] J. Ekanayake, X. Qiu, T. Gunarathne, S. Beason, and G. C. Fox, *High Performance Parallel Computing with Clouds and Cloud Technologies*. CRC Press (Taylor and Francis), 07/2010 2010.

[11] "Magellan - Argonne's DoE Cloud Computing," http://magellan.alcf.anl.gov.

[12] R. K. Brunner and L. V. Kalé, "Adapting to load on workstation clusters," in *The Seventh Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, February 1999, pp. 106–112.

[13] J. A. Board, L. V. Kalé, K. Schulten, R. Skeel, and T. Schlick, "Modeling biomolecules: Larger scales, longer durations," *IEEE Computational Science and Engineering*, vol. 1, no. 4, 1994.

[14] L. Kalé and A. Sinha, "Projections: A Scalable Performance Tool," in *Parallel Systems Fair, International Parallel Processing Symposium*, April 1993.