# HPC-Aware VM Placement in Infrastructure Clouds

Abhishek Gupta,
Laxmikant V. Kalé
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
(gupta59, kale)@illinois.edu

Dejan Milojicic,
Paolo Faraboschi
HP Labs
Palo Alto, CA, USA
firstname.lastname@hp.com

Susanne M. Balle
HP Cloud Services
Hudson, NH, USA
susanne.balle@hp.com

***Abstract*—Cloud offerings are increasingly serving workloads with a large variability in terms of compute, storage and networking resources. Computing requirements (all the way to High Performance Computing or HPC), criticality, communication intensity, memory requirements, and scale can vary widely. Virtual Machine (VM) placement and consolidation for effective utilization of a common pool of resources for efficient execution of such diverse class of applications in the cloud is challenging, resulting in higher cost and missed Service Level Agreements (SLAs). For HPC, current cloud providers either offer dedicated cloud with dedicated nodes, losing out on consolidation benefits of virtualization, or use HPC-agnostic cloud scheduling resulting in poor HPC performance.**

**In this work, we address application-aware allocation of $n$ VM instances (comprising a single job request) to physical hosts from a single pool. We design and implement an HPC-aware scheduler on top of OpenStack Compute (Nova) and also incorporate it in a simulator (CloudSim). Through various optimizations, specifically topology- and hardware-awareness, cross-VM interference accounting and application-aware consolidation, we demonstrate enhanced VM placements which achieve up to 45% improvement in HPC performance and/or 32% increase in job throughput while limiting the effect of jitter (or noise) to 8%.**

***Keywords*-Cloud; High Performance Computing; Scheduling; Placement;**

## I. Introduction

Cloud computing is increasingly being explored as a cost effective alternative (and addition) to supercomputers for some HPC applications [1]–[4]. Cloud provides the benefits of economy of scale, elasticity, flexibility, and customization (through virtualization) to the HPC community. It is attracting several users who cannot afford to deploy their own dedicated HPC infrastructure due to up-front investment or sporadic demands.

Despite these benefits, *today's HPC is not cloud-aware*, and *today's clouds are not HPC-aware*. As a consequence, only embarrassingly parallel or small scale HPC applications are good candidates today to run in cloud. The cloud commodity interconnects (or better, the absence of low-latency interconnects), the performance overhead introduced by virtualization, and the application-agnostic cloud schedulers are the biggest obstacles for efficient execution of HPC applications in cloud [2], [3].

Past research [1]–[4] on HPC in cloud has primarily focused on evaluation of scientific parallel applications (such as those using MPI [5]) and has reached pessimistic conclusions. HPC applications are usually composed of tightly coupled processes performing frequent inter-process communication and synchronizations, and pose significant challenges to cloud schedulers. There have been few efforts on researching VM scheduling algorithms which take into account the nature of HPC applications and have shown promising results [6]–[9].

In this paper, we postulate that the placement of VMs to physical machines can have significant impact on performance. With this as motivation, the primary questions that we address are the following: Can we improve HPC application performance in cloud through VM placement strategies tailored to application characteristics? Is there a cost-saving potential through increased resource utilization achieved by application-aware consolidation? What are the performance-cost tradeoffs in using VM consolidation for HPC?

We address the problem of how to effectively utilize a common pool of resources for efficient execution of very diverse classes of applications in the cloud. For that purpose, we solve the problem of simultaneously allocating multiple VM instances comprising a single job request to physical hosts taken from a pool. We do this while meeting Service Level Agreement (SLA) requirements (expressed in terms of compute, memory, homogeneity, and topology), and while attempting to improve the utilization of hardware resources. Existing VM scheduling mechanisms rely on user inputs and static partitioning of clusters into availability zones for different application types (such as HPC and non-HPC).

The problem is particularly challenging because, in general, a large-scale HPC application would ideally require a dedicated allocation of cloud resources (compute and network), since its performance is quite sensitive to variability (caused by noise, or jitter). The per-hour charge that a cloud provider would have to establish for dedicating the resources would quickly make the proposition uneconomical for customers. To overcome this problem, our technique identifies suitable application combinations whose execution profiles well complement each other and that can be consolidated on the same hardware resources without compromising the overall HPC performance. This enables us to better utilize the hardware, and lower the cost for HPC applications while maintaining performance and profitability, hence greatly enhancing the business value of the solution.

The methodology used in this paper consists of a two step process – 1) Characterizing applications based on their use of shared resources in a multi-core node (with focus on

shared cache) and their tightly coupledness, and 2) using an application-aware scheduler to identify groups of applications that have complementary profiles.

The key contributions of this paper are:

- We identify the opportunities and challenges of VM consolidation for HPC in cloud. In addition, we develop scheduling algorithms which optimize resource allocation while being HPC-aware. We achieve this by applying Multi-dimensional Online Bin Packing (MDOBP) heuristics while ensuring that cross-application interference is kept within bounds. (§II, §III)
- We optimize the performance for HPC in cloud through intelligent HPC-aware VM placement – specifically topology awareness and homogeneity, showing performance gains up to 25% compared to HPC-agnostic scheduling. (§III, §VI)
- We implement the proposed algorithm in OpenStack Nova scheduler to enable intelligent application-aware VM scheduling. Through experimental measurements, we show that compared to dedicated execution, our techniques can result in up to 45% better performance while limiting jitter to 8%. (§IV, §VI)
- We modify CloudSim [10] to make it suitable for simulation of HPC in cloud. To our knowledge, our work is the first effort towards simulation of HPC job scheduling algorithms in cloud. Simulation results show that our techniques can result in up to 32% increased throughput compared to default scheduling algorithms. (§VII)

## II. VM Consolidation for HPC in Cloud: Scope and Challenges

There are two advantages associated with the ability to mix HPC and other applications on a common platform. First, better system utilization since the machines can be used for running non-HPC applications when there is low incoming flux of HPC applications. Secondly, placing different types of VM instances on the same physical node can result in advantages arising from resource packing.

To quantify the potential cost savings that can be achieved through consolidation, we performed an approximate calculation using pricing of Amazon EC2 instances [11]. Amazon EC2 offers a dedicated pool of resources for HPC applications known as *Cluster Compute*. We consider two instance types shown in Table I and, as a concrete example, Table II shows the distribution of actually executed jobs calculated from METACENTRUM-02.swf logs obtained from the Parallel Workload Archive [12]. It is clear that there is a wide distribution and some HPC applications have small memory footprint while some need large memory. Also, according to the US DoE, there is a technology trend towards decreasing memory per core for exascale supercomputers, indicating that memory will be even more crucial resource in future [13]. If the memory left unused by some applications in the *Cluster Compute* instance can be used by placing a *High Memory* instance on the same node by trading 13 EC2 Compute Units and 34.2 GB memory (still leaving 60.2 - 34.2 = 26 GB), then

**TABLE I: Amazon EC2 instance types and pricing**

| Resource | Instance type | |
|---|---|---|
| | High-Memory Double Extra Large | Cluster Compute Eight Extra Large |
| API name | m2.2xlarge | cc2.8xlarge |
| EC2 Comp. Units | 13 | 88 |
| Memory | 34.2 GB | 60.5 GB |
| Storage | 850 GB | 3370 GB |
| I/O Perf. | High | Very High |
| Price ($/hour) | 0.9 | 2.4 |

**TABLE II: Distribution of job's memory requirement**

| Memory per core | Number of Jobs |
|---|---|
| <512MB | 87075 (84.00%) |
| 512MB-1GB | 10062 (9.71%) |
| 1GB-2GB | 5946 (5.74%) |
| 2GB-4GB | 379 (0.37%) |
| 4GB-8GB | 161 (0.16%) |
| >8GB | 33 (0.03%) |
| Total | 103656 (100%) |

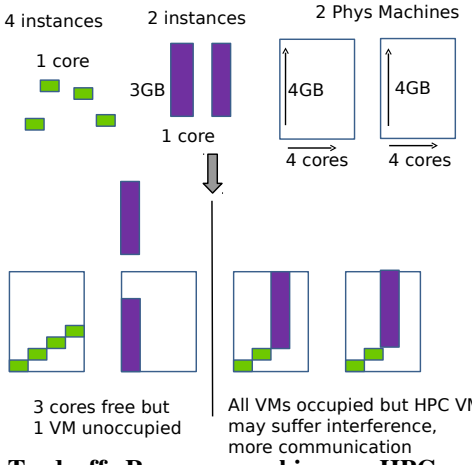
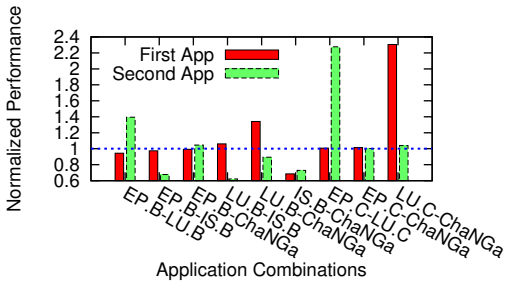


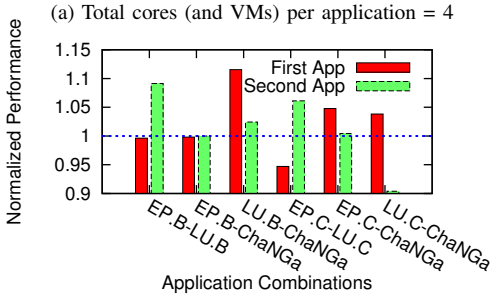

**Fig. 1: Tradeoff: Resource packing vs. HPC-awareness**

from Table I pricing, for every 2.4$, one can get additional 0.9$. However, the price of cluster compute instance needs to be reduced by a factor of $2.4/(88/13)$, since that instance will have 13 EC2 units less. Hence, through better resource packing, we can get % benefits of $\frac{[2.4-2.4/(88/13)+0.9]-2.4}{2.4} = 23\%$.

However, traditionally, HPC applications are executed on dedicated nodes to prevent any interference arising from co-located applications. This is because the performance of many HPC applications strongly depends on the slowest node (for example, when they synchronize through MPI barriers). Figure 1 illustrates this tradeoff between resource packing and optimized HPC performance with an example. Here we have two incoming VM provisioning requests, first – for 4 instances each with 1 core, 512 MB memory and second – for 2 instances each with 1 core, 3 GB memory. There are two available physical servers each with 4 cores and 4 GB memory. Figure shows two ways of placing these VMs on physical servers. The boxes represent the 2 dimensions – x dimension being cores, y dimension being memory. Both requests are satisfied in the right figure, but not in left figure, since there is not enough memory on an individual server to meet the 3 GB requirement although there is enough memory in the system as a whole. Hence, the right figure is a better strategy since it is performing 2-dimensional bin packing. Now consider that the 1 core 512 MB VMs (green) are meant for HPC. In that case,




(a) Total cores (and VMs) per application = 4




(b) Total cores (and VMs) per application = 16

**Fig. 2: Application Performance in shared node execution (2 cores for each application on a node) normalized wrt to dedicated execution (using all 4 cores of a node for same application). Total cores (and VMs) per application = 16, physical cores per node = 4**

the left figure can result in better HPC performance compared to the right one because of two reasons – a) No interference from applications of other users running on same server, and b) all inter-process communications is within node. This tradeoff between better HPC performance vs. better resource utilization makes VM scheduling for HPC a challenging problem.

### A. Cross-Application Interference

Even though there are potential benefits of using consolidation of VMs for HPC, it is still unclear whether (and to what extent) we can achieve increased resource utilization at an acceptable performance penalty. For HPC applications, the degradation due to interference accumulates because of the synchronous and tightly coupled nature of many HPC applications. We compared the performance of a set of HPC applications in a co-located vs. dedicated execution. Figure 2 demonstrates the effect of running two different applications while sharing a multi-core node (4-core, 8GB, 3 GHz Open Cirrus [14] node). Each VM needs 1-vcpu, 2GB memory, and uses KVM-hypervisor and CPU-pinned configuration. Applications used here are NPB [15] (EP = Embarrisingly Parallel, LU = LU factorization, IS = Integer Sort) problem size class B and C and ChaNGa [16] = Cosmology. More details of testbed and applications are discussed in Section V.

In this experiment, we first ran each application using all 4 cores of a node. We then ran VMs from 2 different applications on each node (2 VMs of each application on a node). Next, we normalized the performance for both applications in second case (shared node) with respect to the first case (dedicated node), and plotted them as shown in Figures 2a (4 VMs each application) and 2b (16 VMs each applications) for different application combinations. In the figures, the x-label shows the application combination, and the first bar shows normalized performance for the first application in x-label. Similarly the second bar shows that of second application in x-label. We can observe that some application combinations have normalized performance close to one for both applications e.g. EP.B-ChaNGa. For some applications, co-location has a significant detrimental impact on performance on at least one application e.g. combinations involving IS.B.

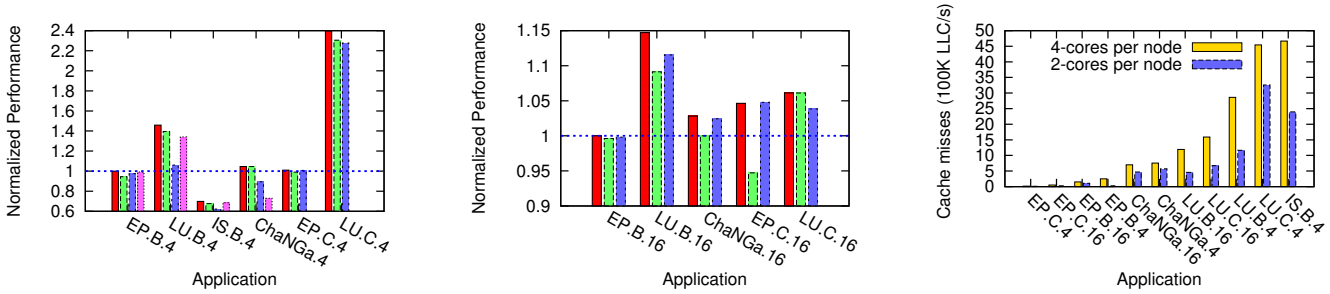
The other facet of the interference problem is positive interference. Through experimental data, we notice that we can achieve significant performance improvement for some application combinations e.g. LU.C-ChaNGa for 4 cores case shows almost 120% better performance for LU.C with ChaNGa's normalized performance close to 1. The positive impact on performance when co-locating different HPC applications presents to us another opportunity for optimizing VM placement. What needs to be explored is why some co-locations perform well while others do not (Section III).

### B. Topology Awareness

The second challenge to VM consolidation for HPC in cloud is the applications' sensitivity to network topology. Since many parallel processes constituting an HPC application communicate frequently, time spent in communication forms a significant fraction of total execution time. The impact of cluster topology has been widely researched by HPC researchers, but in the context of cloud, it is up to the cloud provider to use VM placement algorithms which map the multiple VMs of an HPC application in a topology-aware manner to minimize inter-VM communication overhead. The importance of topology awareness can be understood by a practical example – Open Cirrus HP Labs cluster has 32 nodes (4-cores each) in a rack, and all nodes in a rack are connected by a 1Gbps link to a switch. The racks are connected using a 10Gbps link to a top-level switch. Hence, the 10Gbps link is shared by 32 nodes with an effective bandwidth of 10Gbps/32 = 0.312 Gbps between two nodes in different racks for all-to-all communication. However, the point-to-point bandwidth between two nodes in the same rack is 1 Gbps. Thus, packing VMs to nodes in the same rack will be beneficial compared to a random placement policy, which can potentially distribute them all over the cluster. However, topology-aware placement can conflict with the goals of achieving better resource utilization as demonstrated by Figure 1.

### C. Hardware Awareness

Another characteristic of HPC applications is that they are generally iterative and bulk synchronous, with computation phase followed by barrier synchronization phase. Since all processes must finish the previous iteration before next iteration can be started, a single slow process can slow down the entire application. Since clouds evolve over time and demand, they consist of heterogeneous servers. Furthermore, the underlying hardware is not visible to the user who expects all VMs to achieve identical performance. The commonly used approach




(a) Total cores (and VMs) per application = 4 (b) Total cores (and VMs) per application = 16 (c) Last Level Cache Misses

**Fig. 3: (a,b) Application Performance using 2 cores per node normalized wrt to dedicated execution using all 4 cores of a 4-core node for same application. First bar for each application shows the case when leaving 2 cores idle (no co-located applications). Rest bars for each application show the case when co-located with other applications (combinations of Figure 1). (c) Average per core last level cache misses: Using only 2 cores vs. using all 4 cores of a node**

to address heterogeneity in cloud is to create a new compute unit (e.g. Amazon EC2 Compute unit) and allocate hardware based on this unit. This allows allocation of a CPU core to multiple VMs using shares (e.g. 80-20 CPU share). However, it is impractical for HPC applications since the VMs comprising a single application will quickly get out of sync when sharing CPU with other VMs, resulting in much worse performance. To overcome these problems, Amazon EC2 uses a dedicated cluster for HPC. However, the disadvantage is lower utilization which results in higher price. Hence, the third challenge for VM placement for HPC is to ensure homogeneity. VM placement needs to be hardware-aware to ensure that all $k$ VMs of a user request are allocated same type of processors.

## III. Methodology

Having identified the opportunities for HPC-aware VM consolidation in cloud, we discuss our methodology for addressing the challenges discussed in Section II. We formulate the problem as an initial VM placement problem: Map $k$ VMs $(v_1, v_2, .., v_k)$ each with same, fixed resource requirements (CPU, memory, disk etc.) to $n$ physical servers $P_1, P_2, ..., P_n$, which are unoccupied or partially occupied, while meeting resource demands. Moreover, we focus on providing the user an HPC-optimized VM placement. Our solution consists of a) One-time application characterization, and b) application-aware scheduling. Next, we discuss these two components.

### A. Application Characterization

Our goal is to identify what characteristics of applications affect their performance when they are co-located with other applications on a node. To get more insights into the performance observed in Figure 2, we plot the performance of each application obtained when running alone (but using 2 VMs on a 4 core node, leaving 2 cores idle) normalized with respect to the performance obtained when using all 4 cores for same application (See Figures 3a and 3b first bar for each application). We can see that LU benefits most when run in 2 core per node case, EP and ChaNGa achieve almost same performance, and IS suffers. This indicates that the contention of shared resources in multi-core nodes is a critical factor for these applications. To confirm our hypothesis, we measured the number of last level cache (LLC) misses per sec for each application using hardware performance counters and Linux tool `oprofile`. Figure 3c shows LLC misses/sec for our application set, and demonstrates that LU suffers a huge number of misses, indicative of larger working set size (or cache-intensiveness). In our terminology, cache-intensive refers to larger working set. Co-relating Figures 3a and 3c, we see that applications which are more cache-intensive (that is suffer more LLC misses per sec) are the ones that benefit most in 2-core per node case, whereas applications which are low to moderate cache-intensive (e.g. EP and ChaNGa) are mostly not affected by the use of 2 or 4-cores per node. One exception to this is IS.B.4, because this application is highly communication-intensive and hence suffers because of the inter-node communication happening in 2-core per node case. Barring this exception, one fairly intuitive conclusion that can be drawn from this experiment is that it is indeed beneficial to co-locate cache-intensive applications (such as LU) and application with less cache usage (such as EP) on same node. This is confirmed by more closely examining Figure 2.

HPC applications introduce another dimension to the problem of accounting cross-application interference. In general, the effect of noise/interference gets amplified in applications which are bulk synchronous. For synchronous HPC applications, even if only one VM suffers a performance penalty, all the remaining VMs would have to wait for it to reach the synchronization point. Even though the interference suffered by individual processes may be less over a period of time, the overall effect on application performance can be significant due to the accumulation of noise over all processes. Hence, we characterize applications along two dimensions:

1) Cache-intensiveness – We assign each application a cache score (= 100K LLC misses/sec), representative of the pressure it puts on the shared cache and memory controller subsystem. We acknowledge that one can use working set size as a metric, but we chose LLC misses/sec since it can be experimentally measured using hardware performance counters.

2) Parallel Synchronization and Network Sensitivity – We map applications to four different application classes, which can be specified by a user when requesting VMs:

- ExtremeHPC: Extremely tightly coupled or topology-sensitive applications for which the best will be to provide dedicated nodes, example – IS.
- SyncHPC: Sensitive to interference, but less compared to ExtremeHPC and can sustain small degree of interference to get consolidation benefits, examples – LU, ChaNGa.
- AsyncHPC: Asynchronous (and less communication sensitive) and can sustain more interference than SyncHPC, examples – EP, MapReduce applications.
- NonHPC: Do not perform any communication, can sustain more interference, and can be placed on heterogeneous hardware, example – Web applications.

### B. Application-aware Scheduling

With this characterization, we devise an application-characteristics aware VM placement algorithm which is a combination of HPC-awareness (topology and homogeneity awareness), Multi-dimensional Online Bin packing, and Interference minimization through cache-sensitivity awareness. We discuss the details of this scheduler in the next section.

## IV. An HPC-Aware Scheduler

Next, we discuss the design and implementation of the proposed techniques on top of OpenStack Nova scheduler [17].

### A. Background: OpenStack Nova Scheduler

OpenStack [17] is an open source software, being developed by collaboration of multiple inter-related projects, for large-scale deployment and management of private and public clouds from large pools of infrastructure resources (compute, storage, and networking). In this work, we focus on the compute component of OpenStack, known as Nova. Nova scheduler performs the task of selecting physical nodes where a VM will be provisioned. Since OpenStack is a popular cloud management system, we implemented our scheduling techniques on top of existing Nova scheduler (Diablo 2011.3).

The default scheduler makes VM placement based on the VM provisioning request (`request_spec`), and the existing state and occupancy of physical hosts (capability data). `request_spec` specifies the number and type of requested instances (VMs), instance type maps to resource requirements such as number of virtual cores, amount of memory, amount of disk space. Host capability data contains the current capabilities (such as free CPUs, free memory) of physical servers (hosts) in the cloud. Using `request_spec` and capabilities data, the scheduler performs a 2-step algorithm:

1) Filtering – excludes hosts incapable of fulfilling the request (e.g free cores $<$ requested virtual cores).
2) Weighing – computes the relative fitness of filtered list of hosts to fulfill the request using cost functions such as least free host. Multiple cost functions can be used.

Next, the list of hosts is sorted by the weighted score, and VMs are provisioned on hosts using this sorted list.

However Nova Scheduler is HPC-agnostic since:

- Existing filtering and weighing strategies do not consider the nature of application (e.g. HPC vs. non-HPC).

**Algorithm 1** Pseudo code for Scheduler Algorithm

```
1: capability = list of capabilities of unique hosts
2: request_spec = request specification
3: numHosts = capability.length()
4: filteredHostList = new vector < int >
5: rackList = new set < int >
6: hostCapacity, rackCapacity, filteredHostList ← CalculateHostAndRackCapacity(request_spec, capabilities)
7: if (request_spec.class == ExtremeHPC) || (request_spec.class == SyncHPC) then
8:     sortedHostList ← sort filteredHostList by decreasing order of hostCapacity[j] where j ∈ filteredHostList.
9:     PrelimBuildPlan ← stable Sort sortedHostList by decreasing order of rackCapacity[capability[j].rackid] where j ∈ filteredHostList.
10: else
11:    PreBuildPlan = filteredHostList
12: end if
13: if request_spec.class == ExtremeHPC then
14:    buildPlan = new vector[int]
15:    for i = 1 to i <= numFilteredHosts do
16:        for j = 1 to j <= hostCapacity[PreBuildPlan[i]] do
17:            buildPlan.push(PreBuildPlan[i])
18:        end for
19:    end for
20: else
21:    buildPlan ← MDOBP(request_spec, Prebuildplan, capabilities)
22: end if
23: return buildPlan

24: procedure MDOBP(request_spec, Prebuildplan, capabilities)

25: buildPlan = new vector[int]
26: for i = 1 to i < request_spec.numInstances do
27:    repeat
28:        node ← chooseBestFitHost(request_spec, Prebuildplan, capabilities) // use a multi-dimensional heuristic
29:    until meetsInterferenceCriteria(node, request_spec, capabilities)
30:    buildPlan.insert(node);
31:    update temporary capability database
32: end for
33: return buildPlan

34: procedure meetsInterferenceCriteria(node, request_spec, capabilities)

35: α = Total cache threshold for any application
36: β = Total cache threshold for SyncHPC, in general α >> β
37: totalCacheScore = Σ_i i.cacheScore ∀i such that i is an instance currently running on node
38: if (totalCacheScore + request_spec.cacheScore) > α then
39:    return false
40: end if
41: if i.class = SyncHPC for any i – an instance currently running on node then
42:    if (totalCacheScore + request_spec.cacheScore) > β then
43:        return false
44:    end if
45: end if
46: return true
```

- Scheduler ignores processor heterogeneity and network topology.
- Scheduler considers the $k$ VMs requested by an HPC user as $k$ separate placement problems, there is no co-relation between the placement of VMs of a single request.

There has been recent and ongoing work on adapting Nova scheduler to make it architecture- and HPC-aware [8], [9].

### B. Design and Implementation

Algorithm 1 describes our scheduling algorithm using OpenStack terminology. The VM provisioning request (`request_spec`) now contains application class and name in addition to existing parameters. The algorithm proceeds by calculating the current host and rack free capacity, that is

number of additional VMs of requested specification that can be placed at a particular host and rack (line 6). While doing so, it sets the capacity of all the hosts which have a running VM as zero if the requested VM type is ExtremeHPC to ensure that only dedicated nodes are used for ExtremeHPC. Next, if the class of requested VM is ExtremeHPC or SyncHPC, the scheduler creates a preliminary build plan which is a list of hosts ordered by `rackCapacity` of the rack to which a host belongs and `hostCapacity` for hosts of same rack. The goal is to allocate VMs to same host and same rack to the extent possible to minimize inter-VM communication overhead for these application classes. For ExtremeHPC, this `PreBuildPlan` is used for provisioning VMs, whereas for the rest classes, the algorithm performs multi-dimensional online bin packing to fit VMs of different characteristics together on same host (line 21). Procedure `MDOBP` uses a bin packing heuristic for selecting a host from available choices (line 28). We use a dimension-aware heuristic – select the host for which the vector of requested resources aligns the most with the vector of remaining capacities. The key intuition can be understood by revisiting the example of 2-dimensional bin-packing in Figure 1. For best utilization of the capacity in both dimensions, it is desirable that the final sum of all the VM vectors on a host is close to the top right corner of the host rectangle. Hence, we select the host such that placing the requested VM on it would move the vector representing its occupied resources towards the top right corner. Our heuristic is similar to those studied by Lee et al. [18]. Formally, consider remaining or residual capacities ($CPURes$, $MemRes$) of a host, i.e. subtract from the capacity (total CPU, total memory) the total demand of all the items (VM cores, VM memory) currently assigned to it. Also consider requested VM: ($CPUReq(=1)$, $MemReq$). This heuristic selects the host with the minimum $\theta$ where cos($\theta$) is calculated using dot product of the two vectors, and is given by: $\frac{(CPUReq*CPURes)+(MemReq*MemRes)}{\sqrt{CPURes^2+MemRes^2}\sqrt{CPUReq^2+MemReq^2}}$, with $CPURes >= CPUReq, MemRes >= MemReq$.

Next, the selected host is checked to ensure that placing the requested VM on it does not violate the interference criteria (line 29). We use the following criteria – the sum of cache scores of the requested VM and all the VMs running on a host should not exceed a threshold, which needs to be determined through experimental analysis. This threshold is different if the requested VM or one or more VMs running on that host is of class SyncHPC since applications of this class can tolerate lesser interference (line 44). In addition, we maintain a database of interference indices to record interference between those applications which suffer large performance penalty when sharing hosts. This information is used to avoid co-locations which are definitely not beneficial. The output of Algorithm 1 is `buildPlan` which is the list of hosts where the VMs should be provisioned.

To ensure homogeneity, hosts are grouped into different lists based on their processor type, and the algorithm operates on these groups. Currently, we use CPU frequency as the distinction criteria between processor types. For more accurate distinction, additional factors such as MIPS can be considered.

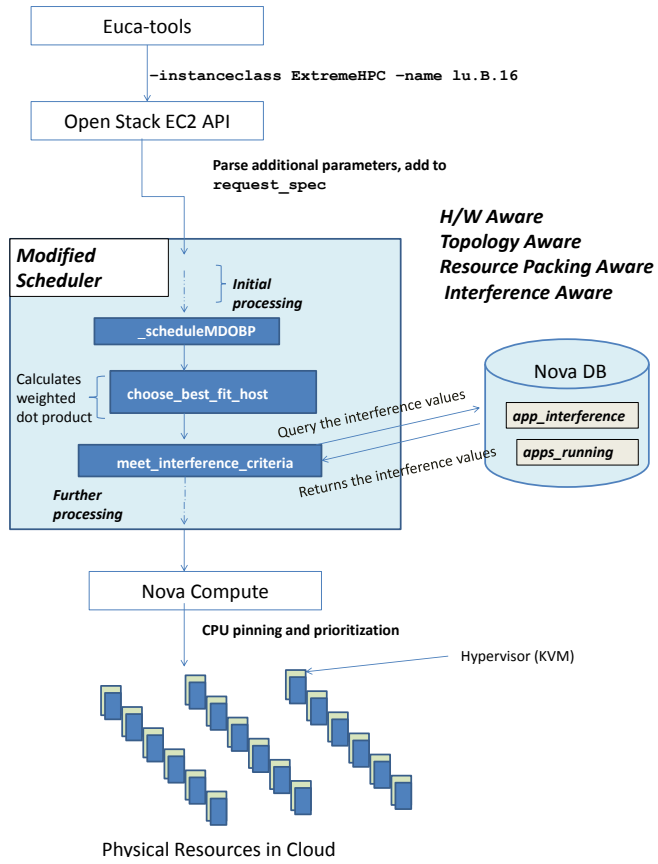



**Fig. 4: Implementation details and control flow of a provisioning request**

Figure 4 shows the overall control flow for a VM provisioning request, highlighting the additional features and changes that we introduced while implementing the HPC-aware scheduling algorithm in OpenStack Nova. We modified both euca-tools and OpenStack EC2 API to allow additional parameters to be passed along with a VM provisioning request. Further, we store these additional properties (such as application class and cache score) of running VMs in the Nova database. Also, we create and maintain additional information – interference indices, which is a record of interference suffered by each application with other applications during a characterization run. New tables `app_running` with columns that store the host name and applications running on it, and `app_interferences` that stores the interference between any of them were added to Nova DB. Nova DB API was modified to include a function to read this database.

We extended the existing `abstract_scheduler.py` in Nova to create `HPCinCloud_scheduler.py` which contains the additional functions `_scheduleMDOBP`, `choose_best_fit_host`, and `meet_interference_criteria`.

## V. Evaluation Methodology

In this section, we describe our cloud setup and the applications which we used.

### A. Experimental Testbed

We evaluated our techniques on a cloud setup using OpenStack on Open Cirrus testbed at HP Labs site [14]. This cloud has 3 types of servers:

- Intel Xeon E5450 (12M Cache, 3.00 GHz)
- Intel Xeon X3370 (12M Cache, 3.00 GHz)
- Intel Xeon X3210 (8M Cache, 2.13 GHz)

The cluster topology is as described in Section II-B.

For virtualization, we chose KVM [19], since prior research has indicated that KVM is a good choice for virtualization for HPC clouds [20]. For network virtualization, we experimented with different network drivers such as *rtl8139, eth1000, virtio-net*, and settled on *virtio-net* because of better network performance (also shown in [6]). We used VMs of type *m1.small* (1 core, 2 GB memory, 20 GB disk). However, these choices do not influence the generality of our conclusions.

### B. Benchmarks and Applications

We used the NAS Parallel Benchmarks (NPB) [15] problem size class B and C (the MPI version, NPB3.3-MPI), which are widely used by HPC community for performance benchmarking, and provide good coverage of computation, communication, and memory characteristics. We also used three larger HPC applications:

- NAMD [21] – A highly scalable molecular dynamics application used ubiquitously on supercomputers. We used the ApoA1 input (92k atoms) for our experiments.
- ChaNGa [16] – A cosmology application which perform collisionless N-body simulation using Barnes-Hut tree for force calculation. We used a 300,000 particle system.
- Jacobi2D – A 5-point stencil computation kernel which averages values in a 2-D grid, and is used in scientific simulations, numerical algebra, and image processing.

These applications are written in Charm++ [22] which is an object-oriented parallel programming language. We used the `net-linux-x86-64` machine layer of Charm++ with –O3 optimization level.

## VI. Experimental Results

Next, we evaluate the benefits of HPC-aware VM placement and the effect of jitter arising from VM consolidation.

### A. HPC-Aware Placement

To demonstrate the impact of topology awareness and homogeneity, we compared the performance obtained by HPC-aware scheduler with random VM placement. In these experiments, we did not perform VM consolidation. Figure 5 shows the performance obtained by our VM placement (Homo) compared to the case when two VMs are mapped to a slower processors, rest to the faster processor (Hetero). We calculated % improvement = $(T_{Hetero} - T_{Homo})/T_{Hetero}$. We can see that the improvement achieved depends on the nature of application and the scale at which it is run. Also, the improvement is not equal to the ratio of sequential execution time on slower processor to that on faster processor. This can

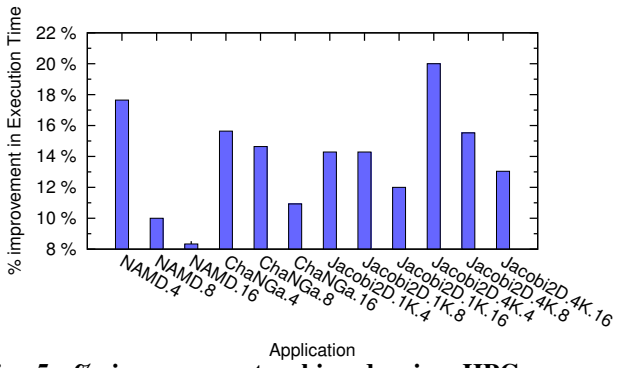
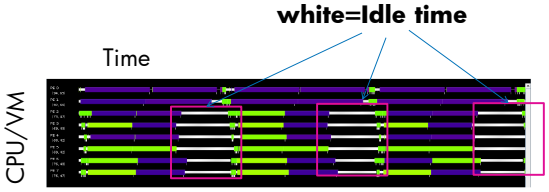
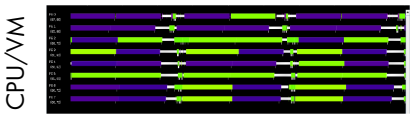



**Fig. 5: % improvement achieved using HPC awareness (homogeneity) compared to the case where 2 VMs were on slower processors and rest on faster processors**




(a) Heterogeneous: First two VMs on slower processors




(b) Homogeneous: All 8 VMs on same type of processors

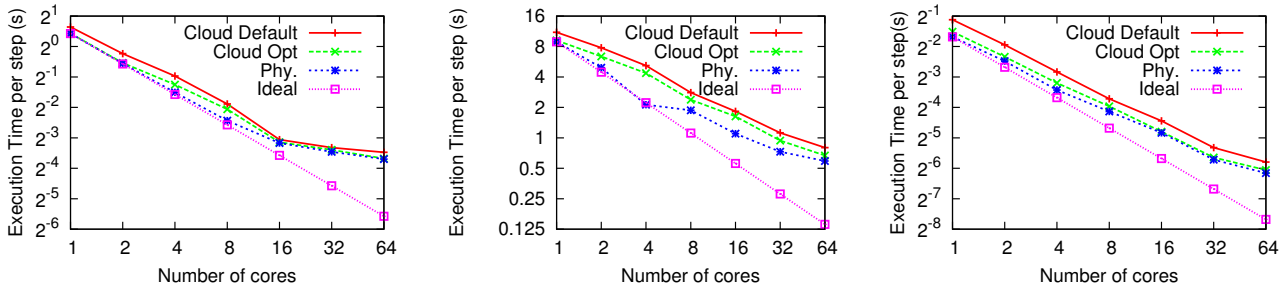
**Fig. 6: CPU Timelines of 8 VMs running Jacobi2D**

be attributed to the communication time and parallel overhead, which is not necessarily dependent on the processor speeds. For these applications, we achieved up to 20% improvement in parallel execution time, which means we save *20% of time * N CPU-hours*, where N is the number of processors used.

We analyzed the performance bottleneck using the Projections [23] tool. Figure 6 shows the CPU (VM) timelines for an 8-core Jacobi2D experiment, x-axis is time, y-axis is the (virtual) core number, white portion shows idle time, and colored portions represent application functions. In Figure 6a, there is lot more idle time on VMs 3-7 compared to first 2 VMs (running on slower processors) since VMs 3-7 have to wait for VMs 0-1 to reach the synchronization point. The small idle time in Figure 6b due to the communication time.

Next, we compared the performance obtained when using the VM placement provided by HPC-optimized algorithm vs. the default VM placement vs. without virtualization on the same testbed (see Figure 7). The default placement selects the host with least free CPU cores (or PEs) agnostic of its topology and hardware. In this experiment, the first host in the cloud had slower processor type. Figure 7 shows that even communication-intensive applications such as NAMD and ChaNGa scale well for Cloud-opt case, and achieve performance close to that obtained on physical platform. Benefits up to 25% are achieved compared to the default scheduler.

However, performance achieved on the physical platform itself is up to 4X worse compared to ideal scaling at 64 cores,




(a) NAMD (Molecular Dynamics) (b) ChaNGa (Cosmology) (c) Jacobi2D – 4K by 4K matrix

**Fig. 7: Runtime Results: Execution Time vs. Number of cores / VMs for different applications.**

| Application | Class | Cache Score | Time ded. run | Placement $\beta$=100 | $\beta$=40 | $\beta$=60 |
|---|---|---|---|---|---|---|
| IS.B.4 | ExtremeHPC | 47 | 89.5 | 4×N1 | 4×N1 | 4×N1 |
| LU.C.16 | SyncHPC | 16 | 180.18 | 4×(N2-N5) | 2×(N1-N8) after App1,App3-App5 | 3×(N2-N6) +1×N7 |
| LU.B.4 | SyncHPC | 29 | 147 | 3×N6+1×N7 | 1×(N2-N5) | 2×N1+2×N8 after App1 |
| ChaNGa.4 | SyncHPC | 7.5 | 100.42 | 1×N6+3×N7 | 1×(N2-N5) | 1×(N2-N5) |
| EP.B.4 | AsyncHPC | 2.5 | 101.5 | 4×N8 | 1×(N2-N5) | 1×N6+3×N7 |

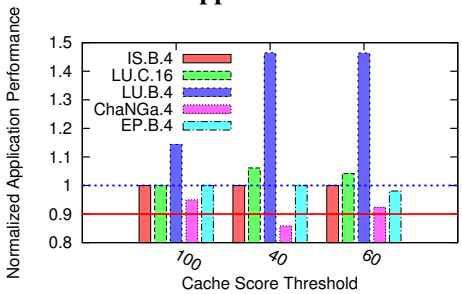



**Fig. 8: Table of applications, and figure showing percentage improvement achieved using application-aware scheduling compared to the case when applications were run in dedicated manner**

likely due to the absence of an HPC-optimized network. A detailed analysis of the communication performance of this cloud (with different virtualization drivers) was done in [6].

### *B. Case Study of Application-Aware Scheduling*

Here, we consider 8 nodes (32 cores) of our experimental testbed, and perform VM placement for the application stream shown in Figure 8 using the application-aware scheduler. The application suffix is the number of requested VMs. Figure 8 shows the characteristics of these applications and the output of scheduler with three different cache thresholds ($\beta$). The output (Placement) is presented in the form of the nodes (and cores per node) to which the scheduler mapped the application. This figure also shows the achieved performance for these cases compared to the dedicated execution using all 4 cores per node. When the cache threshold is too large, there is less performance improvement due to aggressive packing of cache-intensive applications on the same node. On the contrary, a very small threshold results in unnecessary wastage of some CPU cores if there are few applications with very small cache scores. This is illustrated by the placement shown in Figure 8, where the execution of some applications was deferred because the interference criteria were not satisfied due to small cache threshold. Moreover, there is additional penalty (communication overhead) associated when not using all cores of a node for running an HPC application. Hence, the cache threshold needs to be chosen carefully through extensive experimentation. In this case, we see that the threshold of 60 works the best. For this threshold and our application set, we achieve performance gains up to 45% for a single application while limiting negative impact of interference to 8%.

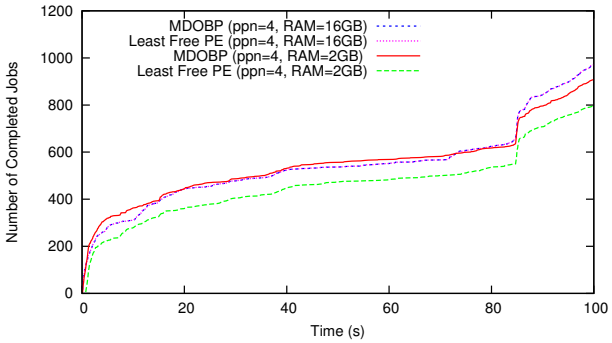
We also measured the overhead of our scheduling algorithm by measuring the execution time. The average time to handle a request for 1, 16 instances was 1.58s, 1.80s respectively by our scheduler compared to 1.54s, 1.67s for default scheduler.

## VII. Simulation

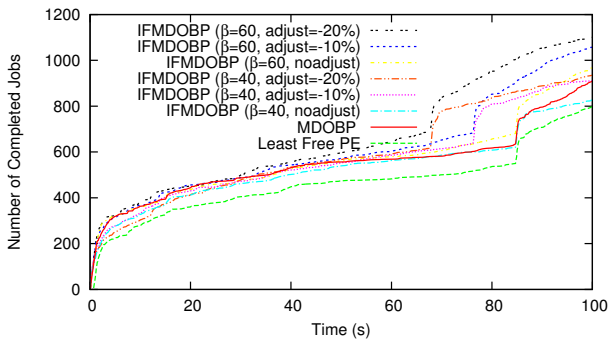
CloudSim is a simulation tool modeling a cloud computing environment in a datacenter, and is widely used for evaluation of resource provisioning algorithms [10]. In this work, we modified it to enable the simulation of High Performance Computing jobs in cloud. HPC machines have massive number of processors, whereas CloudSim is designed and implemented for cloud computing environment, and works mainly with jobs which needs single processor. Hence, for simulating HPC in cloud, the primary modification we performed was to improve the handling of multi-core jobs. We extended the existing `vmAllocationPolicySimple` class to create a `vmAllocationPolicyHPC` which can handle a user request comprising multiple VM instances and performs application-aware scheduling (discussed in Algorithm 1).

At the start of simulation, a fixed number of VMs (of different specified types) are created, and jobs (cloudlets) are submitted to the data center broker which maps a job to a VM. When there are no pending jobs, all the VMs are terminated and simulation completes. Since our focus was on mapping of VMs to physical hosts, we created a one-to-one mapping between cloudlets and VMs. Moreover, we implemented VM termination during simulation to ensure complete simulation. Without dynamic VM creation and termination, the initial set of VMs run till the end of simulation, leading to indefinitely blocked jobs in the system since the VMs where they can run never get scheduled because of the limited datacenter capacity.

We simulated the execution of jobs from the logs obtained from parallel workload archive [12]. We used the METACENTRUM-02.swf logs since these logs contain in-




(a) MDOBP vs. default "Least Free PE" heuristic for different amount of RAM per node. ppn = processors (cores) per node




(b) Interference-aware MDOBP for different cache thresholds ($\beta$), adjusted execution times (accounting for better performance with cache awareness)

**Fig. 9: Simulation Results: Number of completed jobs vs time for different scheduling techniques, 1024 cores**

formation about a job's memory consumption. For each job record ($n$ cores, $m$ MB memory, execution time) in the log file, we create $n$ VMs each with 1-core and $m/n$ MB memory. We simulated the execution of first 1500 jobs from the log file on 1024 cores, and measured the number of completed jobs after 100 seconds. Figure 9a shows that the number of completed jobs after 100 seconds increased by around $109/801 = 13.6\%$ when using MDOBP instead of the default heuristic (selecting a node with least free PEs) for the constrained memory case (2GB per node), whereas there was no improvement for this job set when the nodes had large memory per core. This is attributed to the fact that this job set has very few applications with large memory requirement. However, with the trend of the big memory applications, also true for the next generation exascale applications, we expect to see significant gains for architectures with large memory per node as well.

We also simulated our HPC-aware scheduler (including cache-awareness) by assigning each job a cache score from (0-30) using a uniform distribution random number generator. We used two different values of the cache threshold (See Figure 9b IFMDOBP). We simulated the jobs with modified execution times of all jobs by -10% and -20% to account for the improvement in performance resulting from cache-awareness as seen from results in Section VI. The number of completed jobs after 100 seconds further increased to 1060 for the cache threshold of 60 and adjustment of -10%, which is a reasonable choice based on the results obtained in Section VI-B. Hence, overall we get improvement in throughput by $259/801 = 32.3\%$ compared to default scheduler. Also, we can see that a small cache threshold ($\beta$=40) can actually degrade overall throughput because some cores will be left unused to ensure that the interference requirements are obeyed.

## VIII. Related Work

Previous studies on HPC applications in cloud have concluded that cloud cannot compete with supercomputers based on the metric $/GFLOPS for large scale HPC applications because of bottlenecks such as interconnect and I/O performance [1]–[4]. However, clouds can be cost-effective for some applications, specifically those with less communication and at low scale [3], [4]. In this work, we explore VM placement techniques to make HPC in cloud more economical through improved performance and resource utilization.

Work on scheduling in cloud can be classified into three areas: 1) Initial VM Placement – where the problem is to map a (set of) VM(s) of a single user request to available pool of resources. 2) Offline VM Consolidation – where the problem is to map VM(s) from different user requests, hence with different resource requirements to physical resources to minimize the number of active servers to save energy. 3) Live Migration – where remapping decisions are made for live VMs. Our focus is on the first problem, since our research is towards infrastructure clouds (IaaS) such as Amazon EC2, where VM allocation and mapping happen as and when VM requests arrive. Offline VM consolidation has been extensively researched [24], [25], but is not applicable to IaaS. Also, live migration has associated costs, and introduces further noise.

For initial VM placement, existing cloud management systems such as OpenStack [17], Eucalyptus [26], and OpenNebula [27] use Round Robin (next available server), First Fit (first available server), or Greedy Ranking based (best fit according to certain criteria e.g. least free RAM) strategies, which operate in one-dimension (CPU or memory). Other researchers have proposed genetic algorithms [28]. A detailed description and validation of VM consolidation heuristics is provided in [18]. However, these techniques ignore the intrinsic nature of HPC VMs – tightly coupledness.

Fan et al. discuss topology-aware deployment for scientific applications in cloud, and map the communication topology of a parallel application to the VM physical topology [7]. Recently, OpenStack community has been working on making the scheduler architecture-aware and suitable for HPC [8], [9]. Amazon EC2 has a *Cluster Compute* instance which allows *placement groups* such that all instances within a placement group are expected to get low latency and full bisection 10 Gbps bandwidth [29]. It is not known how strictly those guarantees are met and what techniques are used to meet them.

In this work, we extend our previous work on HPC-aware scheduler [6] in multiple ways - First, we use multi-dimensional online bin packing (MDOBP) for considering resources along all dimensions (such as CPU and memory).

MDOBP algorithms have been explored in offline VM consolidation research, but we apply these to initial VM placement problem and in the context of HPC in cloud. Second, we leverage the additional knowledge about the application characteristics, such as HPC or non-HPC, synchronization, communication, and cache characteristics to limit cross-application interference. We got insights from studies which have explored the effects of shared multi-core node on cross-VM interference, both in HPC and non-HPC domain [24], [30], [31].

There are many tools for scheduling HPC jobs on clusters, such as Oracle Grid Engine, ALPS, OpenPBS, SLURM, TORQUE, and Condor. They are all job schedulers or resource management systems for cluster or grid environment, and aim to utilize system resources in an efficient manner. They differ from scheduling on cloud since they work with physical *not* virtual machines, and hence cannot benefit from the traits of virtualization such as consolidation. Nodes are typically allotted to a single user, and not shared with other users.

## IX. Lessons, Conclusions and Future Work

We summarize the lessons learned through this research:

- Although it may be counterintuitive, HPC can benefit greatly by consolidating VMs using smart co-locations.
- A cloud management system such as OpenStack would greatly benefit from a scheduler which is aware of the application characteristics such as cache, synchronization and communication behavior, and HPC vs non-HPC.
- Careful VM placement and execution of HPC and other workloads can result in better resource utilization, cost reduction, and hence broader acceptance of HPC clouds.

Through experimental research, we explored the opportunities and challenges of VM consolidation for HPC in cloud. We designed and implemented an HPC-aware scheduling algorithm for VM placement which achieves better resource utilization and limits cross-application interference through careful co-location. Through experimental and simulation results, we demonstrated benefits of up to 32% increase in job throughput and performance improvement up to 45% while limiting the effect of jitter to 8%.

In future, we plan to consider other factors which can affect performance of a VM in a shared multi-core node such as I/O (network and disk). Another direction of research is to address other challenges for adoption of cloud by HPC community.

## Acknowledgments

We thank Arpita Kundu, Ivan Nithin, Chaitra Padmanabhan, Jiban J. Sarma and R Suryaprakash for setting up the cloud environment and helping with the scheduler implementation. We thank Alex Zhang for the discussions on server consolidation. First author was supported by HP Lab's 2012 IRP award.

## References

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

[2] "Magellan Final Report," U.S. Department of Energy (DOE), Tech. Rep., 2011, http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Magellan_Final_Report.pdf.

[3] A. Gupta and D. Milojicic, "Evaluation of HPC Applications on Cloud," in *Open Cirrus Summit (Best Student Paper)*, Atlanta, GA, Oct. 2011, pp. 22 –26. [Online]. Available: http://dx.doi.org/10.1109/OCS.2011.10

[4] A. Gupta et al., "Exploring the Performance and Mapping of HPC Applications to Platforms in the cloud," in *HPDC '12*. New York, NY, USA: ACM, 2012, pp. 121–122.

[5] "MPI: A Message Passing Interface Standard," in *M. P. I. Forum*, 1994.

[6] A. Gupta, D. Milojicic, and L. Kale, "Optimizing VM Placement for HPC in Cloud," in *Workshop on Cloud Services, Federation and the 8th Open Cirrus Summit*, San Jose, CA, 2012.

[7] P. Fan, Z. Chen, J. Wang, Z. Zheng, and M. R. Lyu, "Topology-Aware Deployment of Scientific Applications in Cloud Computing," *Cloud Computing, IEEE International Conference on*, vol. 0, 2012.

[8] "Nova Scheduling Adaptations," http://xlcloud.org/bin/download/Download/Presentations/Workshop_26072012_Scheduler.pdf.

[9] "HeterogeneousArchitectureScheduler," http://wiki.openstack.org/HeterogeneousArchitectureScheduler.

[10] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning algorithms," *Softw. Pract. Exper.*, vol. 41, no. 1, pp. 23–50, Jan. 2011.

[11] "Amazon Elastic Compute Cloud," http://aws.amazon.com/ec2.

[12] "Parallel Workloads Archive." [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/

[13] "Exascale Challenges," http://science.energy.gov/ascr/research/scidac/exascale-challenges.

[14] A. Avetisyan et al., "Open Cirrus: A Global Cloud Computing Testbed," *IEEE Computer*, vol. 43, pp. 35–43, April 2010.

[15] "NAS Parallel Benchmarks," http://www.nas.nasa.gov/Resources/Software/npb.html.

[16] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively Parallel Cosmological Simulations with ChaNGa," in *IDPPS*, 2008, pp. 1–12.

[17] "OpenStack Cloud Computing Software," http://openstack.org.

[18] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder., "Validating Heuristics for Virtual Machines Consolidation," Microsoft Research, Tech. Rep., 2011.

[19] "KVM – Kernel-based Virtual Machine," Redhat, Inc., Tech. Rep., 2009.

[20] A. J. Younge, R. Henschel, J. T. Brown, G. von Laszewski, J. Qiu, and G. C. Fox, "Analysis of Virtualization Technologies for High Performance Computing Environments," *Cloud Computing, IEEE Intl. Conf. on*, vol. 0, pp. 9–16, 2011.

[21] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale, "Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms," in *IPDPS 2008*, April 2008, pp. 1–12.

[22] L. Kalé, "The Chare Kernel parallel programming language and system," in *Proceedings of the International Conference on Parallel Processing*, vol. II, Aug. 1990, pp. 17–25.

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

[24] A. Verma, P. Ahuja, and A. Neogi, "Power-aware Dynamic Placement of HPC Applications," ser. ICS '08. New York, NY, USA: ACM, 2008, pp. 175–184.

[25] S. K. Garg, C. S. Yeo, A. Anandasivam, and R. Buyya, "Energy-Efficient Scheduling of HPC Applications in Cloud Computing Environments," *CoRR*, vol. abs/0909.1146, 2009.

[26] D. Nurmi et al., "The Eucalyptus Open-source Cloud-computing System," in *Proceedings of Cloud Computing and Its Applications*, 2008.

[27] "The Cloud Data Center Management Solution ," http://opennebula.org.

[28] J. Xu and J. A. B. Fortes, "Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments," ser. GREENCOM-CPSCOM '10. Washington, DC, USA: IEEE Computer Society, pp. 179–188.

[29] "High Performance Computing (HPC) on AWS," http://aws.amazon.com/hpc-applications.

[30] J. Mars et al., "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations," ser. MICRO-44 '11. New York, NY, USA: ACM, 2011, pp. 248–259.

[31] J. Han, J. Ahn, C. Kim, Y. Kwon, Y.-R. Choi, and J. Huh, "The Effect of Multi-core on HPC Applications in Virtualized Systems," ser. Euro-Par 2010. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 615–623.