# TECHNIQUES FOR EFFICIENT HIGH PERFORMANCE COMPUTING IN THE CLOUD

BY

ABHISHEK GUPTA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor Maria Garzaran
Professor Indranil Gupta
Doctor Dejan Milojicic, Hewlett Packard (HP) Laboratory

# Abstract

The advantages of pay-as-you-go model, elasticity, and the flexibility and customization offered by virtualization make cloud computing an attractive option for meeting the needs of some High Performance Computing (HPC) users, especially those with emerging or sporadic demands. "Computing or Infrastructure as a service" model in cloud has tremendous potential of spreading the outreach of HPC to wider scientific and industrial community. However, this potential has largely remained unrealized. This thesis makes an attempt to study the reasons for the lack of adoption of cloud computing by HPC community and alleviate them using software techniques. We hypothesize that current clouds are suitable for *some* HPC applications *not all* applications, and for those applications, clouds can be more cost-effective compared to typical dedicated HPC platforms using intelligent scheduling of applications to platforms in cloud. Through comprehensive performance evaluation and analysis, we find that there are gaps between the characteristic traits of many HPC applications and existing cloud environments. The poor interconnect and I/O performance in cloud, network virtualization overhead, HPC-agnostic cloud schedulers, and the inherent heterogeneity and multi-tenancy in cloud are some bottlenecks for efficient HPC in cloud. Our philosophy for bridging the divide between HPC and clouds is to a) use a complementary approach of making clouds HPC-aware and HPC cloud-aware, b) consider also the unique opportunities offered by cloud for HPC, such as virtual machine (VM) consolidation and elasticity, besides addressing the challenges posed by clouds, and c) consider views of both, HPC users and cloud providers, who sometimes have conflicting objectives: users must see tangible benefits (in cost or performance) while cloud providers must be able to run a profitable business. With this philosophy, the techniques presented in this thesis, viz. HPC-aware cloud scheduling and VM placement, cloud-aware load balancing for HPC applications, and parallel runtime for enabling dynamically shrinking or expanding parallel jobs, significantly improve HPC performance and cloud resource utilization for HPC in cloud. We

believe that our research will help users gain confidence in the capabilities of cloud for HPC, and enable cloud providers to run a more profitable business.

*To my loving parents, beautiful wife, and dearest son.*

# Acknowledgments

First and foremost, I want to thank my wife – *Kritika* and my son – *Atharva* for their love, and for the sacrifices they made when my time and energy were devoted to finishing this thesis. My wife Kritika's incessant support, care, and love throughout this work has been invaluable. Also, my son has brought smile on my face in times of distress and trouble. Paper rejections meant no despair when I could just forget about it all and become a kid with Atharva and Kritika.

I also thank my parents and in-laws for their unconditional love and understanding even though I am living so far from them. The PhD was the most challenging of all the academic degrees I have worked toward. My parents and in-laws were essential in keeping me on the right track all the time.

I would like to thank my advisor, Professor Kalé, for his invaluable guidance throughout this research. He has taught me how to be a good researcher. He generously supported my thesis research even though initially it was tangential to the research group's overall direction. Its heartening to see how he always supported me in my research endeavors.

Dejan Milojicic played a key role in shaping me as a sound and persistent researcher with a "go-get-it" attitude. During my internships at HP Labs, I learned the importance of targeting practical and important research issues. I am always amazed how quickly Dejan responds to my questions even though he is involved in so many projects. I thank him for always being available for me. I also thank Sudarsun Kannan, who was an intern colleague at HP Labs for two summers. I thank him for the discussions we had regarding our research, and for his friendship which made the internships extremely enjoyable experience.

Next, I would like to thank my labmates. Filippo Giaochin, Gengbin Zheng, and Eric Bohm, my colleagues at the Parallel Programming Laboratory, gave me initial insights into the design choices and implementation details in my early research projects. Further, I thank them for answering even the most naive questions I had regarding parallel computing in

general and CHARM++ runtime in particular. Also, Phil Miller helped me a lot during the process of getting my feet into the Linux environment. I realized the importance of team work and group study while preparing for my PhD qualifying exam. Thanks Anshu Arya, Yanhua Sun, and Jonathan Lifflander for teaching me that lesson. While I was struggling to focus on a research direction, Esteban Meneses, Osman Sarood, and Ramprasad Venkatraman were always available to listen to, discuss, and provide feedback on my projects. I also want to thank Lukasz Wesolowski, Xiang Ni, Nikhil Jain, Harshitha Menon, Akhil Langer, Eric Mikida, Ronak Buch, Ehsan Totoni, and Michael Robson – with whom I have worked on various research projects during my graduate studies. I learned lot of technical and interpersonal skills as a result of interaction with them. Bilge Acun helped me during the later part of my PhD. I would like to thank these and all members of the Parallel Programming Laboratory not just for their help, but above all for their friendship.

My friends in town – Manish, Vinod, Parikshit, Pranav, Kapil, and Rajesh were always there in difficult times or when I needed advice. They never made me miss home even when I initially arrived to the US. I would also like to thank my friends from my undergraduate college – IIT Roorkee. These smart but down-to-earth individuals remain to-go person in the face of problems even now.

Last but not least, I would like to thank all the people I ran into during my stay in Urbana-Champaign. I may not remember their names, but they made a difference in one of the most exciting adventures of my life.

# Grants

This work was partially supported by the following sources:

- **Coupled Models of Diffusion and Individual Behavior Over Extremely Large Social Networks**. This project was funded by the National Science Foundation (NSF) under grant SBC VA Tech 478206-19318 ARRA (NSF-0904844).

- **XSEDE Allocation**. The Parallel Programming Laboratory at the University of Illinois was granted an allocation on XSEDE through award ASC050039N. The Principal Investigator of this project is Celso Mendes.

- **CSE at UIUC**. The authors gratefully acknowledge the use of the parallel computing resource provided by the Computational Science and Engineering Program at the University of Illinois. The CSE computing resource, provided as part of the Taub cluster, is devoted to high performance computing in engineering and science.

- **HP Labs IRP**. This work was partially funded by HP Lab's Innovation Research Proposal (IRP) Awards. The Principal Investigator of this project is Laxmikant V. Kale.

- **Grid'5000**. Some of the experiments presented in this work were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several universities as well as other organizations (see https://www.grid5000.fr).

# Table of Contents

# List of Figures

# List of Tables

# List of Algorithms

CHAPTER 1

# Introduction

High Performance Computing (HPC) refers to the practice of aggregating computing power to deliver much higher performance than typical desktop/laptop computers with the goals of solving computationally intensive applications in science, engineering, or business. HPC has traditionally been restricted to a niche segment comprising scientists who use supercomputers at national laboratories to enhance human knowledge by understanding how the universe works. They use the tremendous computational power to understand how scientific phenomena operate and evolve in scientific fields such as cosmology, molecular dynamics, quantum chemistry, climate, and human genetics. More recently, HPC is also being used in industry for business and analytics. According to November 2013 top500 list, 56.4% of top 500 supercomputing systems are used by industry [1].

HPC applications have tremendous potential even outside of large national laboratories, big research groups, and large companies. Small and medium scale organizations can enhance their industrial processes by applying HPC to business and analytics. Similarly, academic researchers at small and medium universities can speed up the scientific discovery by experimenting with HPC as a solution for some of their problems. Some emerging use cases where HPC can fuel scientific research or industrial growth are computer animations [2], health care [3, 4], next-generation manufacturing and computer-aided engineering (CAE) [5], artificial intelligence [6], and large scale graph processing [4, 7].

## 1.1 Justification: Cloud as a Solution for HPC Users' Needs

Despite the significance of HPC towards scientific discovery and industrial growth, access to supercomputers is highly restricted and can only be obtained by securing an allocation

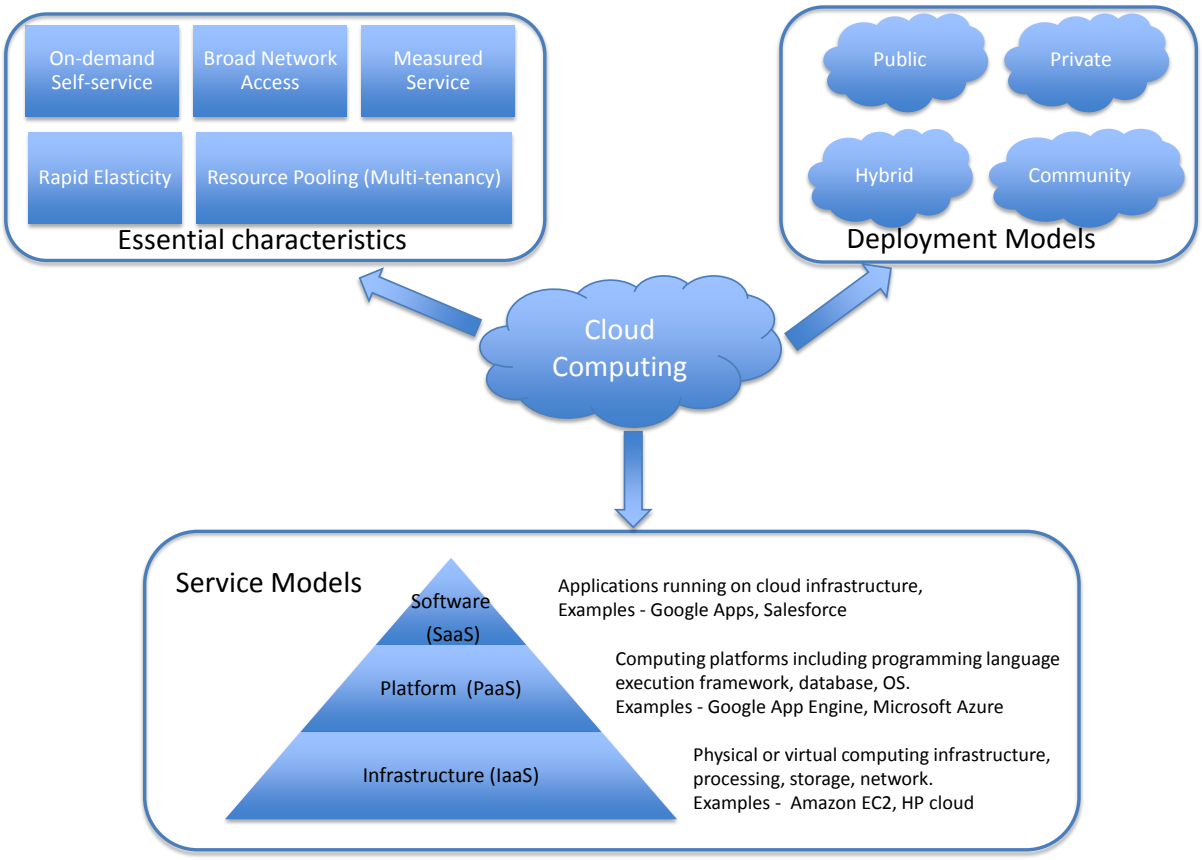



**Figure 1.1: Visualization of NIST definition of cloud computing**

through highly competitive research grant proposals. For small or medium-scale users with emerging HPC demands but limited or no allocation on supercomputers, the conventional option has been to consider the possibility of setting up their own in-house HPC cluster.

Setting up a dedicated infrastructure for HPC is a complex endeavor that requires a long lead time, high capital expenditure, and large operational costs. These barrier to entry have restricted HPC to a small number of significant users. A cheap, fast, and effective alternative can tremendously spread the outreach of HPC to such users.

Recently, cloud computing has successfully addressed similar challenges faced by startups looking to overcome business entry barriers in non-HPC domains and established organizations trying to reduce costs also in non-HPC domains. Cloud computing is a disruptive technology which is defined by the U.S. National Institute of Standards and Technology (NIST) as "a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or

cloud provider interaction" [8]. NIST further describes cloud by listing its five essential characteristics, three service models, and four deployment models. These are visualized in Figure 1.1.

Naturally, public clouds, such as Amazon EC2 [9], which provide Infrastructure-as-a-Service (IaaS) have emerged as a promising alternate to supercomputers, especially for users who cannot afford their own supercomputer due to limitations mentioned earlier. Cloud offers the following advantages to HPC users: (1) Cloud computing is a cost effective alternative with the potential of reducing some of these heavy upfront financial commitments, while yielding to faster turnaround times. (2) By *renting* rather than owning a cluster, additional advantages come in form of *elasticity*, which reduces the risks caused by under-provisioning, and reduces the underutilization of resources caused by over-provisioning. (3) Clouds can provide HPC users infrastructure at cheaper price compared to dedicated cluster since they benefit from economy of scale and also multiple users sharing resources resulting in improved utilization . (4) The built-in virtualization support in the cloud offers an alternative way to support flexibility, customization, security, migration and resource control to the HPC community.

Hence, "Computing as a service" model in cloud can enable High Performance Computing to reach out to wider scientific and industrial community. Whether and how this potential can be realized in practice is a research question that we aim to answer in this thesis.

## 1.2 Research Challenges

Despite the advantages of cloud for HPC, it still remains unclear whether, and when clouds can become a feasible substitute or complement to supercomputers[1]. There is a mismatch between the requirements and goals of HPC and the characteristics and goals of current cloud environments [11–14]. HPC is performance-oriented, whereas clouds are cost and resource-utilization oriented. With these goals, HPC wants dedicated execution to avoid any interference and maximize performance. On the contrary, *resource heterogeneity* and *multi-tenancy* are fundamental artifacts of running in cloud. Clouds evolve over time, leading to heterogeneous configurations in processors, memory, and network. Similarly, multi-tenancy is also intrinsic of cloud, enhancing the business value of providing a cloud. Multi-tenancy leads to multiple sources of interference due to sharing of CPU, cache, memory access, and interconnect. For tightly-coupled HPC applications, heterogeneity and multi-tenancy can result in severe performance degradation and unpredictable performance, since one slow

[1]Some portions reprinted with permission from [10], ©2013 IEEE

**Table 1.1: HPC-cloud divide: There is a mismatch between HPC requirements and cloud characteristics**

| Feature | High Performance Computing (HPC) | Cloud Computing |
|---|---|---|
| Focus/Goal | Application performance | Service, cost, resource utilization |
| Sharing | Dedicated execution | Multi-tenancy |
| Hardware | Conventionally Homogeneous | Heterogeneous |
| Network | HPC-optimized interconnects | Commodity network, virtualization overhead |

processor slows down the entire application. As an example, on 100 processors, if one processor is 30% slower compared to the rest, application will slowdown by 30% even though the system has 99.7% raw CPU power compared to the case when all processors are fast.

Furthermore, traditionally, clouds have been designed for running business and web applications, whose resource requirements are different from HPC applications. Unlike web applications, HPC applications typically require low latency and high bandwidth inter-processor communication to achieve best performance. In case of cloud, presence of commodity interconnect and effect of virtualization result in interconnect becoming a bottleneck for HPC applications.

In addition, *today's HPC is not cloud-aware*, and *today's clouds are not HPC-aware*. Current cloud management systems, such as OpenStack, are HPC-agnostic and current HPC runtimes, such as MPI, are cloud-agnostic.

Table 1.1 summarizes the HPC-cloud divide. While these challenges paint a rather pessimistic view of HPC clouds, recently there have been efforts towards HPC-optimized clouds (such as Amazon Cluster Compute [15] and DoE Magellan project [12, 13, 16]), HPC-aware cloud schedulers [17,18], and topology-aware mapping of application virtual machines (VMs) to physical topology [19]. These efforts point to a promising direction to overcome some of the fundamental inhibitors. However, much work remains to be done, and recent studies have shown that today only embarrassingly parallel or small scale HPC applications can be efficiently run in cloud [11–14].

Unlike previous works [12–14, 20–24] on benchmarking clouds for science, we take a more holistic and practical viewpoint. Rather than limiting ourselves to the problem – *what* is the performance achieved on cloud vs. supercomputer, we address the bigger and more important question – *why* and *who* should choose (or not choose) cloud for HPC, for *what* applications, and *how* should cloud be used for HPC? To this end, we address the following research questions.

- What are the performance-cost tradeoffs on cloud vs. supercomputer? How to decide

what applications are good candidates for running in cloud?

- Will it be beneficial to run some applications on supercomputer and some on cloud rather than running all on a single platform? How to effectively schedule and run HPC jobs in such multi-platform environments?

- What application characteristics are crucial to the determination of the suitable platform for an application under a combined supercomputer-cloud approach? e.g., Latency vs. Bandwidth bound application, point to point vs. collective communication, tightly-coupled vs loosely-coupled?

- What are the challenges and alternatives in VM scheduling for HPC? 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?

- How does a parallel adaptive runtime perform in a virtualized and dynamic cloud environment? How can we adapt a parallel runtime to improve application performance on cloud? Is an supercomputer optimized application good for cloud or is it possible to make applications cloud-friendly e.g. via granularity control?

- What are the unique opportunities offered by cloud for HPC? Can we leverage cloud capabilities such as elasticity and consolidation for HPC e.g. dynamically shrinking/-expanding parallel jobs?

- What are the possible models of delivering HPC in cloud and what are the challenges, practical issues, and benefits associated with them? How well do the current cost models function?

## 1.3 Thesis Overview and Organization

Our primary thesis is that cloud is suitable for *some* HPC applications *not all* applications, and for those applications, cloud can be more cost-effective compared to typical dedicated HPC platforms using intelligent application-to-platform *mapping*. Further, HPC-aware cloud *schedulers*, and cloud-aware HPC *execution* and parallel runtime system enable us to reduce the current divide between HPC and clouds.

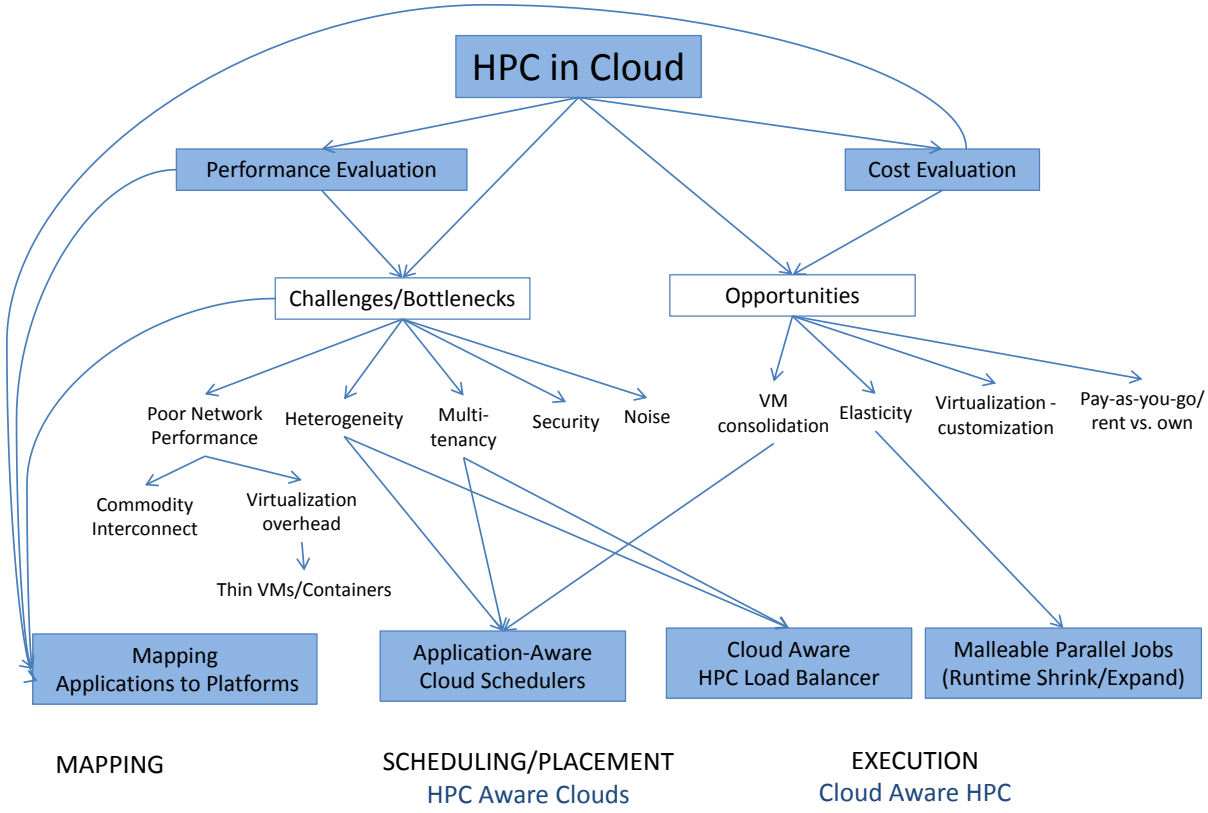



**Figure 1.2: Thesis overview: shaded boxes represent thesis contributions**

Figure 1.2 gives an overview of the contributions and organization of this thesis. First, Chapter 2 presents a comprehensive evaluation and analysis of the performance and cost of running a set of HPC benchmarks and application on a range of platforms varying from supercomputer to cloud. Chapter 2 also presents an economic analysis of HPC in cloud. Furthermore, this chapter identifies the performance bottlenecks of running HPC applications in cloud, also shown in Figure 1.2. These include the following: the absence of low-latency and high-bandwidth interconnect in clouds, network and I/O virtualization overhead, hardware heterogeneity, cross-application interference arising from multi-tenancy, and the HPC-agnostic cloud schedulers.

Next, we present techniques for increasing the efficiency and effectiveness of running HPC applications in cloud. We take a more holistic approach unlike past research: First, besides addressing the challenges of running HPC applications in cloud, we also explore the opportunities offered by cloud for HPC. Secondly, our research is aimed at improving HPC performance, resource utilization, and cost when running in cloud and hence it is beneficial to both – users and cloud providers. Finally, with the objective of providing a set of techniques to bridge the gap between HPC and clouds, we adopt a threefold complementary

approach:

- *Mapping* applications to multiple platforms in cloud intelligently: Chapter 3 co-relates the application characteristics with performance achieved on various platforms, and identifies what application and platform characteristics are most crucial for the selection of a platform for a particular application. Instead of considering cloud as a substitute of supercomputer, we investigate the co-existence of multiple platforms – supercomputer, cluster, and cloud. With this philosophy, first we consider a cloud provider perspective who owns multiple platforms and is looking to optimize the resource utilization while providing good application performance. Our contributions are novel heuristics for application-aware mapping of jobs in this multi-platform scenario. Next, we consider the HPC user's perspective and show how she can use the application knowledge to use a hybrid supercomputer-cloud approach that can be more cost-effective compared to running all applications on a dedicated supercomputer or all in cloud.

- Making cloud *schedulers* and VM placement HPC-aware: Chapter 4 presents and demonstrates techniques for application-aware consolidation and placement of VMs on physical machines. Through topology-awareness, heterogeneity-awareness, cross-VM interference accounting, and careful co-location of application VMs of complementary execution profiles, we achieve significant improvement in performance and resource utilization. Chapter 4 also discusses the challenges and alternatives of using consolidation for HPC applications. We present both – experimental results using OpenStack [25] and simulation results using CloudSim [26].

- Making HPC *execution* and runtime cloud-aware: Chapter 5 addresses the challenges of heterogeneity and multi-tenancy in cloud through dynamic load-balancing of parallel tasks (Charm++ [27, 28] objects or AMPI [28] threads) to VMs. Next, Chapter 6 presents a novel technique for providing parallel runtime support for malleable HPC jobs, that is jobs which can dynamically expand/shrink to benefit from the inherent *elasticity* in cloud. Chapter 6 demonstrates that malleable HPC runtime can benefit both cloud providers and users. First, cloud providers can increase the cluster utilization using such jobs. They can pass some benefits to users by reduced pricing as an incentive for making their jobs malleable. Second, there are emerging use cases, such as Amazon spot markets, which can be exploited by malleable runtimes in future.

Next, Chapter 7 summarizes the conclusions and contributions of this thesis. Chapter 8 provides direction of future research stemming from the ideas presented in this thesis.

CHAPTER 2

# Performance and Cost Analysis of HPC in Cloud

Increasingly, some academic and commercial HPC users are looking at clouds as a cost effective alternative to dedicated HPC clusters [13, 14, 21, 23]. *Renting* rather than owning a cluster avoids the up-front and operating expenses associated with a dedicated infrastructure. Clouds offer additional advantages of a) *elasticity* – on-demand provisioning, and b) virtualization-enabled flexibility, customization, and resource control[1].

Despite these advantages, it still remains unclear whether, and when, clouds can become a feasible substitute or complement to supercomputers. HPC is performance-oriented, whereas clouds are cost and resource-utilization oriented. Furthermore, clouds have traditionally been designed to run business and web applications. Previous studies have shown that commodity interconnects and the overhead of virtualization on network and storage performance are major performance barriers to the adoption of cloud for HPC [13, 14, 20, 21, 23, 29]. While the outcome of these studies paints a rather pessimistic view of HPC clouds, recent efforts towards HPC-optimized clouds, such as Magellan [13] and Amazon's EC2 Cluster Compute [15], point to a promising direction to overcome some of the fundamental inhibitors.

Unlike previous works [13, 14, 20–24] on benchmarking clouds for science, we take a more holistic and practical viewpoint. Rather than limiting ourselves to the problem – *what* is the performance achieved on cloud vs. supercomputer, we address the bigger and more important question in this chapter – *why* and *who* should choose (or not choose) cloud for HPC, for *what* applications, and *how* should cloud be used for HPC? While addressing this research problem, we make the following contributions in this chapter.

- We evaluate the performance of HPC applications on a range of platforms varying from

[1]Some portions reprinted with permission from [29], ©2013 IEEE and [14], ©2011 IEEE

supercomputer to cloud. Also, we analyze bottlenecks and the correlation between application characteristics and observed performance, identifying *what* applications are suitable for cloud. (§2.2, §2.3)

- We also evaluate the performance when running the same benchmarks on exactly same hardware, without and with different virtualization technologies, thus providing a detailed analysis of the isolated impact of virtualization on HPC applications (§2.5).

- To bridge the divide between HPC and clouds, we present the complementary approach of (1) making HPC applications cloud-aware by optimizing an application's computational granularity and problem size for cloud and (2) making clouds HPC-aware using thin hypervisors, OS-level containers, and hypervisor- and application-level CPU affinity, addressing – *how* to use cloud for HPC. (§2.4, §2.5)

- We investigate the economic aspects of running in cloud and discuss *why* it is challenging or rewarding for cloud providers to operate business for HPC compared to traditional cloud applications. We also show that small/medium-scale users are the likely candidates *who* can benefit from an HPC-cloud. (§2.6)

We believe that it is important to consider views of both, HPC users and cloud providers, who sometimes have conflicting objectives: users must see tangible benefits (in cost or performance) while cloud providers must be able to run a profitable business. The insights from comparing HPC applications execution on different platforms is useful for both. HPC users can better quantify the benefits of moving to a cloud and identify which applications are better candidates for the transition from in-house to cloud. Cloud providers can optimize the allocation of applications to their infrastructure to maximize utilization, while offering best-in-class cost and quality of service.

## 2.1 Evaluation Methodology

In this section, we describe the platforms which we compared and the applications which we chose for this study.

### 2.1.1 Experimental Testbed

We selected platforms with different interconnects, operating systems, and virtualization support to cover the dominant classes of infrastructures available today to an HPC user.

**Table 2.1: Testbed**

| Resource | Platform | | | | | |
|---|---|---|---|---|---|---|
| | Ranger | Taub | Open Cirrus | Private Cloud | Public Cloud | EC2-CC Cloud |
| Processors in a Node | 16×AMD Opteron QC @2.3 GHz | 12×Intel Xeon X5650 @2.67 GHz | 4×Intel Xeon E5450 @3.00 GHz | 2×QEMU Virtual CPU @2.67 GHz | 4×QEMU Virtual CPU @2.67 GHz | 16×Xen HVM VCPU @2.6 GHz |
| Memory | 32 GB | 48 GB | 48 GB | 6 GB | 16 GB | 60 GB |
| Network | Infiniband (1 GB/s) | QDR Infiniband | 10GigE internal, 1GigE x-rack | Emulated 1GigE | Emulated 1GigE | Emulated 10GigE |
| OS | Linux | Sci. Linux | Ubuntu 10.04 | Ubuntu 10.04 | Ubuntu 10.10 | Ubuntu 12.04 |



**Table 2.2: Virtualization Testbed**

| Resource | Virtualization | | |
|---|---|---|---|
| | Phy., Container | Thin VM | Plain VM |
| Processors in a Node/VM | 12×Intel Xeon X5650 @2.67 GHz | 12×QEMU Virtual CPU @2.67 GHz | 12×QEMU Virtual CPU @2.67 GHz |
| Memory | 120 GB | 100 GB | 100 GB |
| Network | 1GigE | 1GigE | Emulated 1GigE |
| OS | Ubuntu 11.04 | Ubuntu 11.04 | Ubuntu 11.04 |

Table 2.1 shows the details of each platform. In case of cloud a *node* refers to a virtual machine and a *core* refers to a virtual core. For example, "2 × QEMU Virtual CPU @2.67GHz" means each VM has 2 virtual cores. Ranger [30] at TACC was a supercomputer with a theoretical peak performance of 579 Tera FLOPS[2], and Taub at UIUC is an HPC-optimized cluster. Both use Infiniband as interconnect. Moreover, Taub uses scientific Linux as OS and has QDR Infiniband with bandwidth of 40 Gbps. We used physical nodes with commodity interconnect at Open Cirrus testbed at HP Labs site [31]. The next two platforms are clouds – a private cloud setup using Eucalyptus [32], and a public cloud. We use KVM [33] for virtualization since it has been shown to be a good candidate for HPC virtualization [34]. Finally, we also used an HPC-optimized cloud – Amazon EC2 Cluster Compute Cloud [15] of US West (Oregon) zone, *cc2.8xlarge* instances with Xen HVM virtualization launched in same *placement group* for best networking performance [15].

In case of cloud, most common deployment of multi-tenancy is not sharing individual physical cores, but rather done at the node, or even coarser level. This is even more true with increasing number of cores per server. Hence, our private and public cloud experiments involve physical nodes (not cores) which were shared by VMs from external users, hence providing a multi-tenant environment.

Another dedicated physical cluster at HP Labs Singapore (HPLS) is used for controlled tests of the effects of virtualization (see Table 2.2). This cluster is connected with a Gigabit Ethernet network on a single switch. Every server has two CPU sockets, each populated with a six-core CPU, resulting in 12 physical cores per node. The experiment on the HPLS cluster involved benchmarking on four configuration: physical machines (bare), LXC containers [35], VMs configured with the default emulated network (plain VM), and VMs with pass-through networking (thin VM). Both the plain VM and thin VM run on top of the KVM hypervisor. In the thin VM setup, we enable Input/Output Memory Management Unit (IOMMU) on the Linux hosts to allow VMs to directly access the Ethernet hardware, thus improving the network I/O performance [36]. This virtualization testbed is designed to test the isolated impact of virtualization, impossible to execute on public clouds, due to the lack of direct access to public cloud's hardware.

### 2.1.2 Benchmarks and Applications

To gain insights into the performance of selected platform over a range of applications, we chose benchmarks and applications from different scientific domains and those which

[2]Ranger was decommissioned in Feb 2013, concurrent with the deployment of a new supercomputer – Stampede at TACC. Ranger was ranked 50 in the November 2012 top500 supercomputer list.

differ in the nature, amount, and pattern of inter-processor communication. Moreover, we selected benchmarks written in two different parallel programming environments – MPI [37] and CHARM++ [38]. Similarly to previous work [14, 20, 21, 23], we used NAS Parallel Benchmarks (NPB) class B [39] (the MPI version, NPB3.3-MPI), which exhibit a good variety of computation and communication requirements.

Moreover, we chose additional benchmarks and real world applications:

- Jacobi2D – A 5-point stencil kernel to average values in a 2-D grid. Such stencil kernels are common in scientific simulations, numerical linear algebra, numerical solutions of Partial Differential Equations (PDEs), and image processing.
- NAMD [40] – A highly scalable molecular dynamics application and representative of a complex real world application used ubiquitously on supercomputers. We used the ApoA1 input (92k atoms) for our experiments.
- ChaNGa [41] (Charm N-body GrAvity solver) – A cosmological simulation application which performs collisionless N-body interactions using Barnes-Hut tree for calculating forces. We used a 300,000 particle system.
- Sweep3D [42] – A particle transport code widely used for evaluating HPC architectures. Sweep3D is the heart of a real Accelerated Strategic Computing Initiative (ASCI) application and exploits parallelism via a wavefront process. We ran the MPI-Fortran77 code in weak scaling mode maintaining $5 \times 5 \times 400$ cells per processor with 10 k-planes/3 angles per block.
- NQueens – A backtracking state space search problem where the goal is to determine a placement of $N$ queens on an $N \times N$ chessboard (18-queens in our runs) so that no two queens can attack each other. This is implemented as a tree structured computation, and communication happens only for load-balancing purposes.

On Ranger and Taub, we used available MVAPICH2 [43] for MPI and CHARM++ ibverbs layer. On rest of the platforms we installed Open MPI [44] and used net layer of CHARM++.

## 2.2 Benchmarking HPC Performance

Figure 2.1 shows the scaling behavior of our testbeds for the selected applications. These results are averaged across multiple runs (5 executions) performed at different times. We show strong scaling results for all applications except Sweep3D, where we chose to perform weak scaling runs. For NPB, we present results for only Embarrassingly parallel (EP), LU solver (LU), and Integer sort (IS) benchmarks due to space constraints. The first obser-

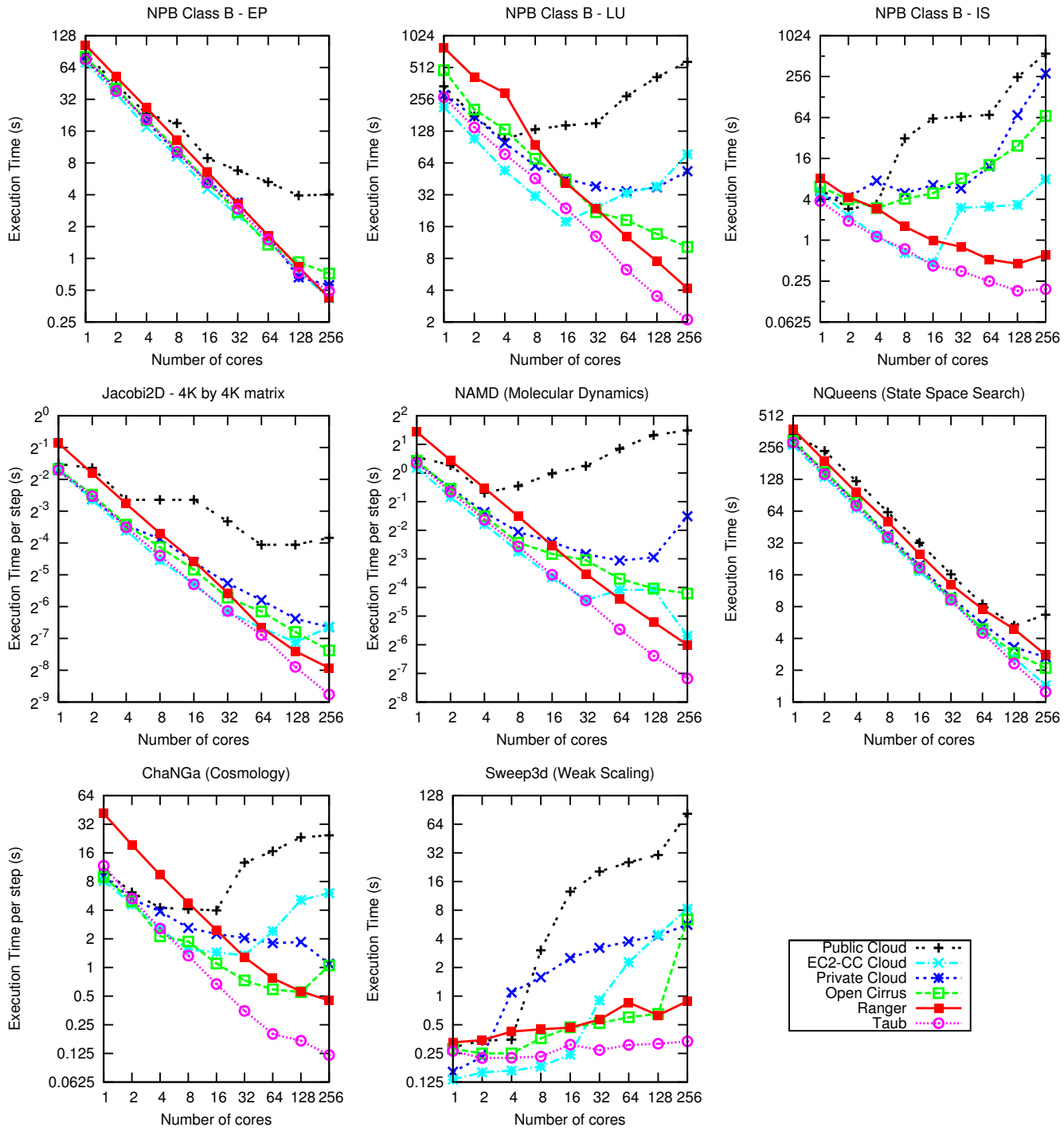
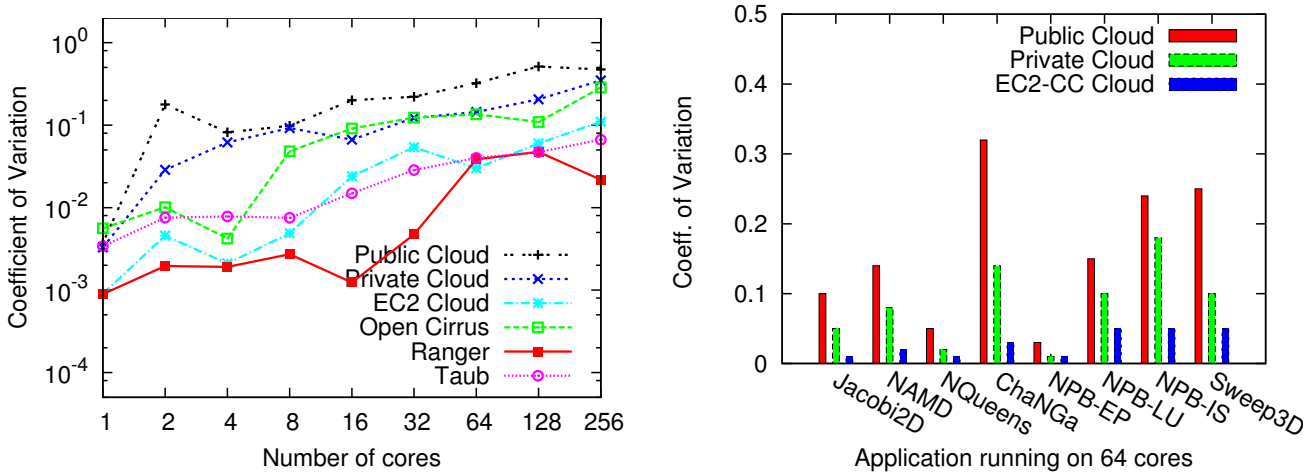



**Figure 2.1: Time in seconds (y-axis) vs. core count (x-axis) for different applications (strong scaling except Sweep3D). All applications scale well on supercomputers and most scale moderately well on Open Cirrus. On clouds, some applications scale well (e.g., EP), some scale till a point (e.g., ChaNGa) whereas some do not scale (e.g., IS).**




(a) ChaNGa: varying core counts

(b) Different Applications

**Figure 2.2: Performance variation**

vation is the difference in sequential performance: Ranger takes almost twice as long as the other platforms, primarily because of the older and slower processors. The slope of the curve shows how the applications scale on different platforms. Despite the poor sequential speed, Ranger's performance crosses Open Cirrus, private cloud and public cloud for some applications at around 32 cores, yielding a much more linearly scalable parallel performance. We investigated the reasons for better scalability of these applications on Ranger using application profiling, performance tools, and microbenchmarking and found that network performance is a dominant factor (Section 2.3).

We observed three different patterns for applications on these platforms. First, some applications such as EP, Jacobi2D, and NQueens scale well on all the platforms up to 128–256 cores. The second pattern is that some applications such as LU, NAMD, and ChaNGa scale on private cloud till 32 cores and stop scaling afterwards. These do well on other platforms including Open Cirrus. The likely reason for this trend is the impact of virtualization on network performance (which we confirm below). On public cloud, we used VM instances with 4 virtual cores, hence inter-VM communication starts after 4 cores, resulting in sudden performance penalty above 4 cores. Similar performance dip can be observed for EC2-CC cloud at 16 cores where each VM had 16 cores. However, in contrast to private and public cloud, EC2-CC cloud provides good scalability to NAMD. Finally, some applications, especially the IS benchmark, perform very poorly on the clouds and Open Cirrus. Sweep3D also exhibits poor weak scaling after 4 – 8 cores on cloud.

In case of cloud, we observed variability in the execution time across runs, which we quantified by calculating the coefficient of variation (standard deviation/mean) for runtime across 5 executions. Figure 2.2a shows that there is significant performance variability on

cloud compared to supercomputer and that the variability increases as we scale up, partially due to decrease in computational granularity. At 256 cores on public cloud, standard deviation is equal to half the mean, implying that on average, values are spread out in the range $[0.5 \times mean, 1.5 \times mean]$ resulting in low run to run predictability. In contrast, private cloud shows less variability. In contrast, EC2-CC cloud shows less variability. Also, performance variability is different for different applications (See Figure 2.2b). Co-relating Figures 2.1 and 2.2b, we can observe that the applications which scale poorly, e.g. ChaNGa and LU, are the ones which exhibit more performance variability.

One potential reason for the significant performance variation is the use of shared resources. We deliberately chose shared systems, shared at node level, not at the core level, for cloud. Using isolated system would be misleading and likely result in far better performance than what one can get from current cloud offerings. Noise induced by multi-tenancy is an intrinsic component of the cloud, inherent in the fundamental business model of the cloud providers.

Further analysis is required to determine what application and platform characteristics are affecting achieved performance and variability. We present our findings in the next section.

## 2.3 Performance Bottlenecks in Cloud

To investigate the reasons for the different performance trends for different applications, we obtained the applications' communication characteristics. We used tracing and visualization – Projections [45] tool for Charm++ applications and MPE and Jumpshot [46] for MPI applications. Table 2.3 shows the results obtained by running on 64 cores of Taub. These numbers are cumulative across all processes. For MPI applications, we have listed the data for point-to-point and collective operations (such as `MPI_Barrier` and `MPI_AlltoAll`) separately. The numbers in parentheses correspond to collectives. It is clear from Table 2.3 that Jacobi2D, NQueens, and EP perform relatively small amount of communication. Moreover, we can categorize applications as latency-sensitive, i.e. large message counts with relatively small message volume, e.g. ChaNGa, or bandwidth-sensitive, i.e. large message volume with relatively small message count, e.g. NAMD, or both, e.g. Sweep3D. The point-to-point communication in IS is negligible. However, it is the only application in the set which performs heavy communication using collectives. This was validated by using Jumpshot visualization tool for MPE logs [46]. Figure 2.3 shows the timeline of execution of IS during this benchmarking, with red (dark) color representing `MPI_Alltoallv` collective communication with contributions of 2MB by each of the 64 processors. It is evident that

**Table 2.3: Communication characteristics of different application, numbers in parentheses correspond to MPI collective calls, bold values may be bottlenecks.**

| Application | Total Message Count (K/s) | Total Message Volume (MB/s) |
|---|---|---|
| Jacobi2D | 220.9 | 309.62 |
| NAMD | 293.3 | **1705.63** |
| NQueens | 37.4 | 3.99 |
| ChaNGa | **2075.8** | 308.53 |
| NPB-EP | 0 (0.2) | 0 (0.01) |
| NPB-LU | **907.3**(0.2) | **415.31** (0.01) |
| NPB-IS | 0.3 (9.0) | 0.00 **(5632)** |
| Sweep3D | **2312.2** (7.2) | **2646.17** (0.02) |

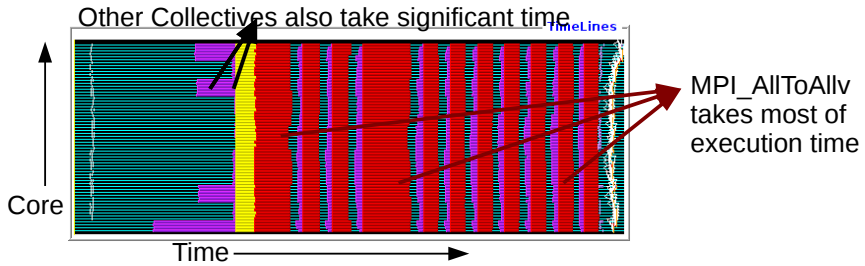



**Figure 2.3: Timeline of execution of IS on 64 cores on Taub. x-axis: time, y-axis: process number. Red (dark) color (MPI_AlltoAllv) is the dominating factor.**

this operation is the dominant component of execution time for this benchmark.

Juxtaposing Table 2.3 and Figure 2.1, we can observe the correlation between the applications' communication characteristics and the performance attained, especially on cloud. Figure 2.4 validates it further. In Figure 2.4, we plotted the slowdown caused by moving an application from supercomputer (Ranger) to private cloud vs. the parallel efficiency achieved on supercomputer. This was done for multiple applications run across different number of processors. Parallel efficiency ($E$) is defined as: $E = S/P$ where $P$ is the number of processors, and Speedup ($S$) is defined as: $S = T_s/T_p$ where $T_s$ is the sequential execution time and $T_p$ is the parallel execution time. From Figure 2.4, we see the trend that applications which achieve high parallel efficiency on supercomputer suffer less slowdown when moved to cloud compared with applications with low parallel efficiency. Hence, performance degradation on cloud is related to an application's communication to computation ratio, since that is the dominant factor that affects parallel efficiency.

To further validate that communication performance is the primary bottleneck in cloud,

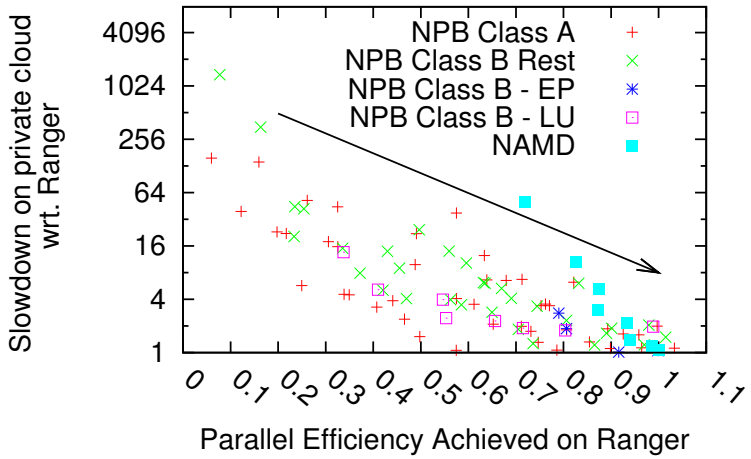
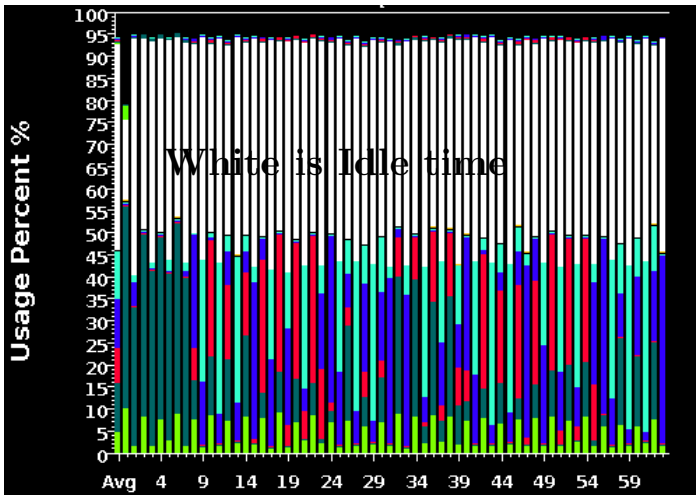



**Figure 2.4: Applications with high parallel efficiency are good candidates for cloud.**

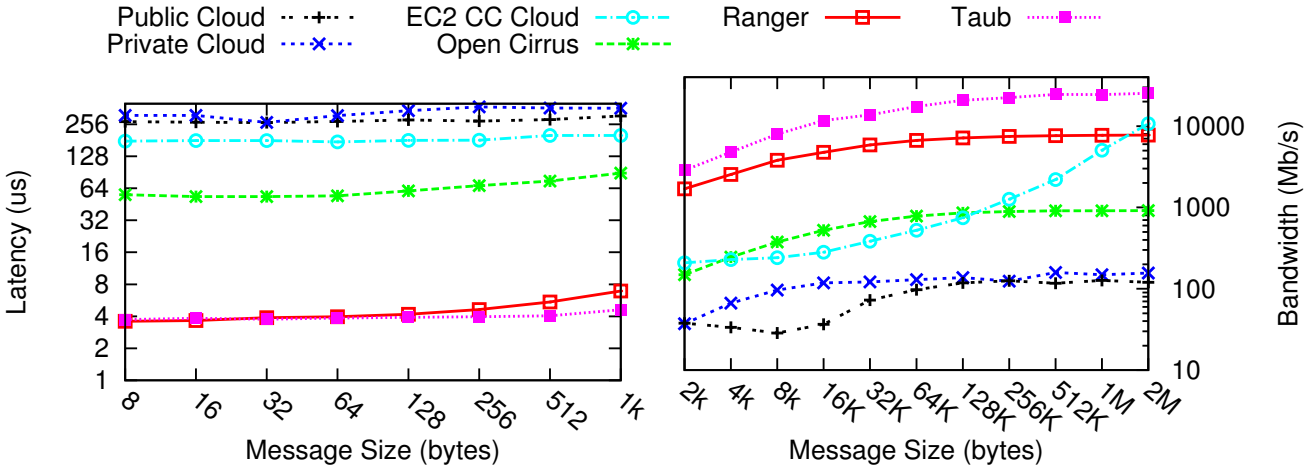
we used Projections performance analysis tool. Figure 2.5a shows the CPU utilization for a 64-core Jacobi2D experiment on private cloud, x-axis being the (virtual) core number. It is clear that CPU is under-utilized for almost half the time, as shown by the idle time (white portion) in the figure. A detailed timeline view revealed that this time was spent waiting to receive data from other processes. Similarly, for other applications which performed poorly on cloud, communication time was a considerable portion of the parallel execution time in cloud.

Since many HPC applications are highly sensitive to communication, we focused on network performance. Figure 2.5b shows the results of a simple ping-pong benchmark written in Converse, the underlying substrate of CHARM++. Unsurprisingly, we found that the latencies and bandwidth on cloud are a couple of orders of magnitude worse compared to Ranger and Taub, making it challenging for communication-intensive applications, such as IS, LU, and NAMD, to scale. EC2-CC cloud provides high bandwidth, enabling bandwidth-sensitive applications, such as NAMD, to scale. However, large latency results in poor performance of latency-sensitive applications, such as ChaNGa.

While the inferior network performance explains the large idle time in Figure 2.5a, the surprising observation is the notable difference in idle time for alternating cores (0 and 1) of each VM. We traced this effect to network virtualization. The light (green) colored portion at the very bottom in Figure 2.5a represents the application function which initiates inter-processor communication through socket operations, and interacts with the virtual network. The application process on core 0 of the VM shares the CPU with the network emulator.




(a) CPU% for Jacobi2D




(b) Latency and Bandwidth

**Figure 2.5: (a) CPU utilization for Jacobi2D on 32 2-core VMs of private cloud. White portion: idle time, colored portions: application functions. (b) Network performance on private and public clouds is off by almost two orders of magnitude compared to supercomputers. EC2-CC provides high bandwidth but poor latency.**

This interference increases as the application communicates more data. Hence, virtualized network degrades HPC performance in multiple ways: increases network latency, reduces bandwidth, and interferes with application process.

We also observed that, even when we used only core 0 of each VM, for iterative applications containing a barrier after each iteration, there was significant idle time on some processes at random times. Communication time could not explain such random idle times. Hence, we used the Netgauge [47] tool for measuring OS noise. We ran a benchmark that performs a

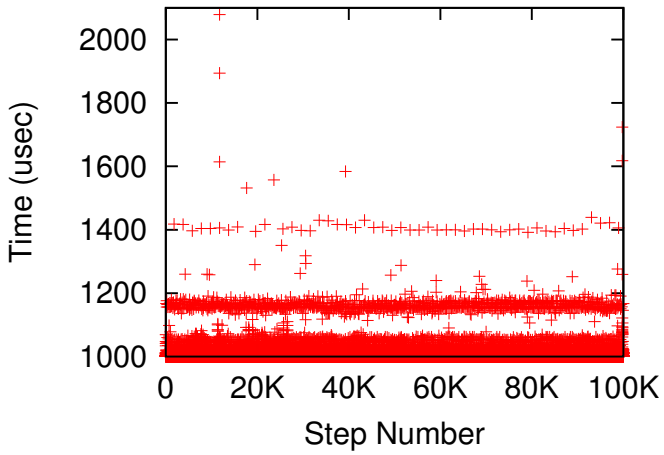



**Figure 2.6: Fixed Work Quantum Benchmark on a VM for measuring OS noise; in a noise-less system every step should take 1000 $\mu$s.**

fixed amount of work multiple times and records the time it takes for each run (Figure 2.6). Each benchmark step is designed to take 1000 microseconds in the absence of noise, but as evident from Figure 2.6, a large fraction of steps takes significantly longer time – from 20% up to 200% longer.

In general, system noise has detrimental impact on performance, especially for bulk-synchronous HPC applications since the slowest thread dictates the speed [48]. Unlike supercomputers, where OS is specifically tuned to minimize noise, e.g., Scientific Linux on Taub, cloud deployments typically run non-tuned operating systems. Clouds have a further intrinsic disadvantage due to the presence of the hypervisor.

## 2.4 Optimizing HPC for Cloud

In Section 2.3, we found that the poor cloud network performance is a major bottleneck for HPC. Hence, to achieve good performance in cloud, it is imperative to either adapt HPC runtime and applications to slow cloud networks (cloud-aware HPC), or improve networking performance in cloud (HPC-aware clouds). Next, we explore the former approach, that is making HPC cloud-aware. The latter approach is discussed in Section 2.5.

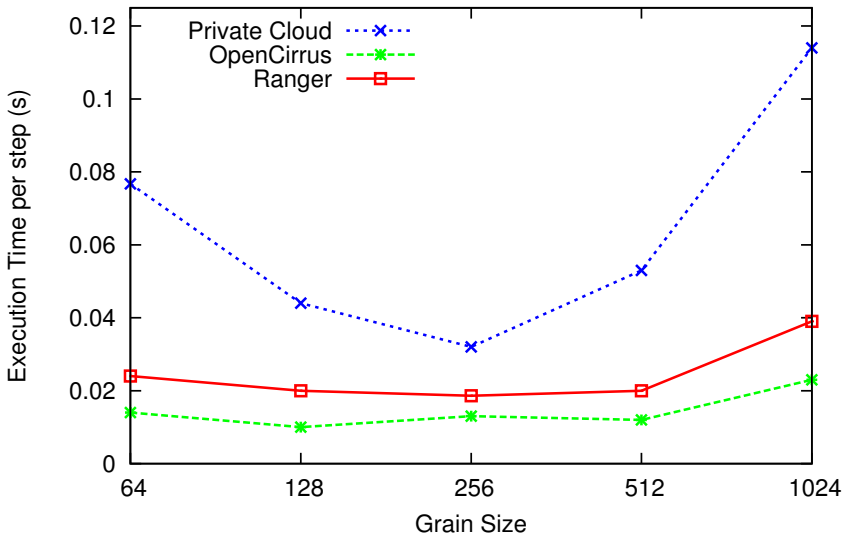



**Figure 2.7: Jacobi2d (4k×4k grid): Proper grain size tuning is critical for improving performance in cloud.**

### 2.4.1 Computational Granularity/Grain size

One way to minimize the sensitivity to network performance is to overlap computation and communication to the maximum possible extent to hide network latencies. A promising direction to achieve such overlap is to use asynchronous object/thread-centric message driven execution rather than MPI-style processor-centric approach.

When there are multiple medium-grained work/data units (objects/tasks) per processors (referred to as *overdecomposition*), and an object needs to wait for a message, control can be asynchronously transferred to another object which has a message to process, thus keeping the processor utilized. Using message-driven execution and an intelligent scheduler, overlap between computation and communication is automatically achieved, without programmer effort. Our hypothesis is that with increase in network latencies, advantages of using message driven execution over traditional MPI-style SPMD model will become more prominent.

To validate our hypothesis, we analyze the effect of the CHARM++ object grain size (or decomposition block size) on execution time of Jacobi2D on 32 cores of different platforms (Figure 2.7). Figure 2.7 shows that the variation in execution time with grain size is significantly more for private cloud as compared to other platforms. As we decrease the grain size, hence increasing number of objects per processor, execution time decreases due to increased overlap of communication and computation. However, after a threshold execution time increases. This trend results from the tradeoff between the speedup due to the overlap

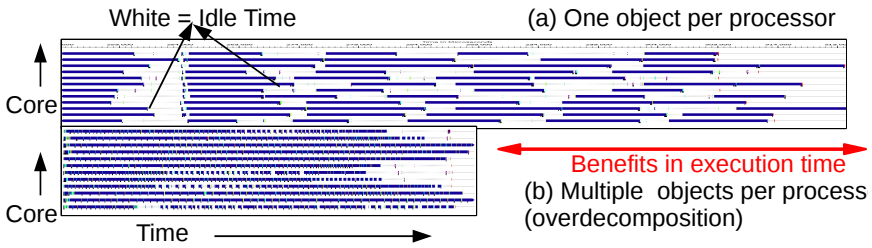



**Figure 2.8: Timelines: Benefits of overdecomposition**

and the slowdown due to parallel runtime's overhead of managing large number of objects.

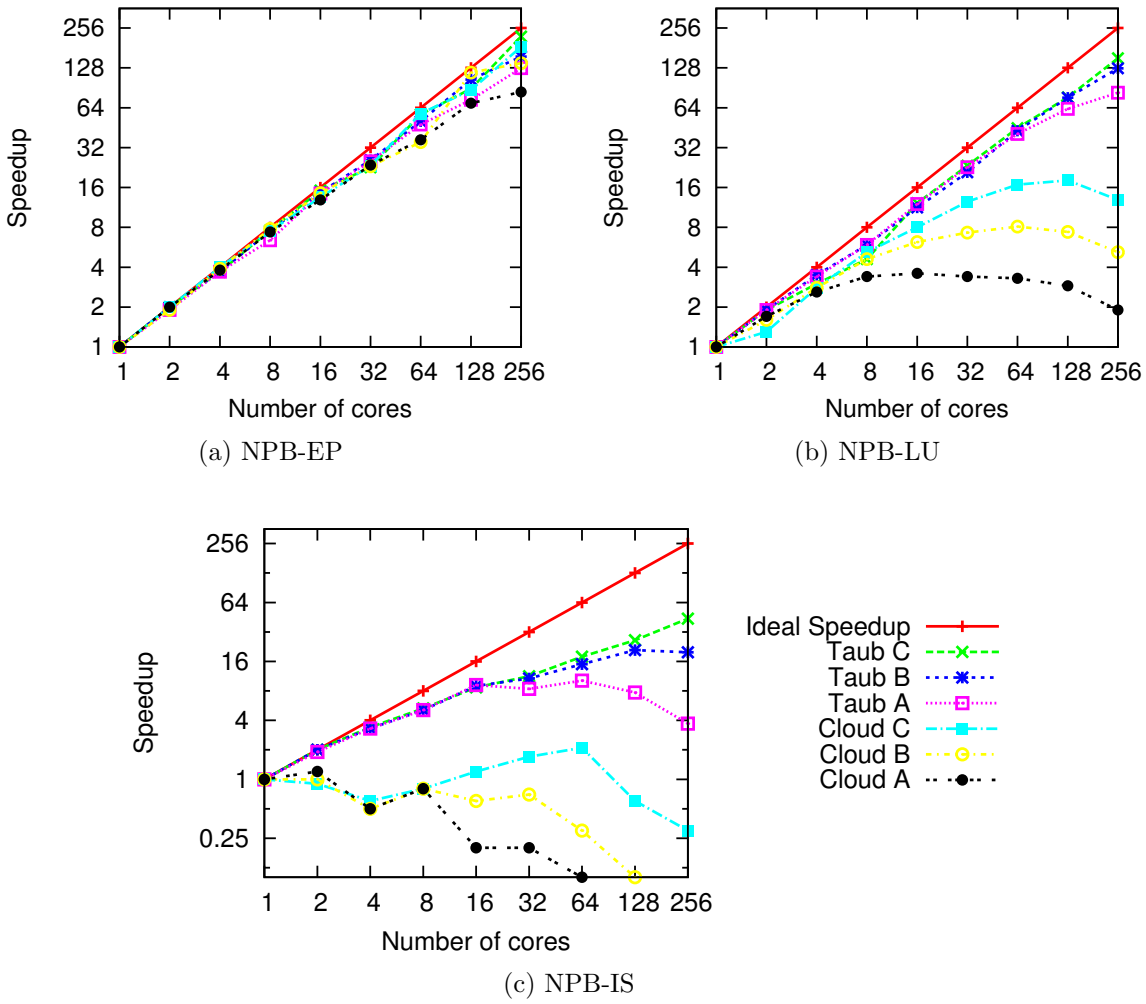
We used Projections tool to visualize the achieved benefit. Figure 2.8 shows the timelines of 12 processes of Jacobi2D execution with and without overdecomposition. Blue represent application functions whereas white represents idle time. There is lot less idling in Figure 2.8b resulting in reduced overall execution time.

### 2.4.2 Problem Sizes

Figure 2.9 shows the effect of problem size on performance (speedup) of different applications on private cloud and supercomputer (Taub). With increasing problem sizes (A→B→C), applications scale better, and the gap between cloud and supercomputer reduces. Figure 2.10 reaffirms the positive impact of problem size. For Jacobi, we denote class A as 1k × 1k, class B as 4k × 4k, and class C as 16k × 16k grid size. As problem size increases (say by a factor of $X$) with *fixed* number of processors, for most scalable HPC applications, the increase in communication (e.g. $\theta(X)$ for Jacobi2D) is less than the increase in computation ($\theta(X^2)$ for Jacobi2D). Hence, the communication to computation ratio decreases with increase in problem size, which results in reduced performance penalty of execution on a platform with poor interconnect. Thus, adequately large problem sizes such that the communication to computation ratio is adequately small can be run more effectively in cloud. Furthermore, applying our cost analysis methodology (Section 2.6), Figure 2.10 can be used to estimate the upper bound of the problem size where it would be cost-effective to run on supercomputer and after that it might be better to run on cloud.

### 2.4.3 Runtime Retuning

While performing experiments, one of the lessons learned was that the parallel programming systems' runtimes have been fine tuned to exploit the latency and bandwidth provided by the




(a) NPB-EP

(b) NPB-LU

(c) NPB-IS

**Figure 2.9: Effect of Problem size class on attained speedup on supercomputer (Taub) vs. private cloud**

HPC systems. To achieve best performance on cloud, some of the network parameters need to be retuned considering the performance of cloud network. E.g., in case of Charm++ , increasing the maximum datagram size from 1400 to 9000, reducing the windows size from 32 to 8, and increasing the acknowledgement delay from 5ms to 18ms resulted in 10–50% performance improvements for our applications.

## 2.5 Optimizing Cloud for HPC

Cloud-aware HPC execution reduces the penalty caused by the underlying slow physical network in clouds, but it does not address the CPU overhead of network virtualization.

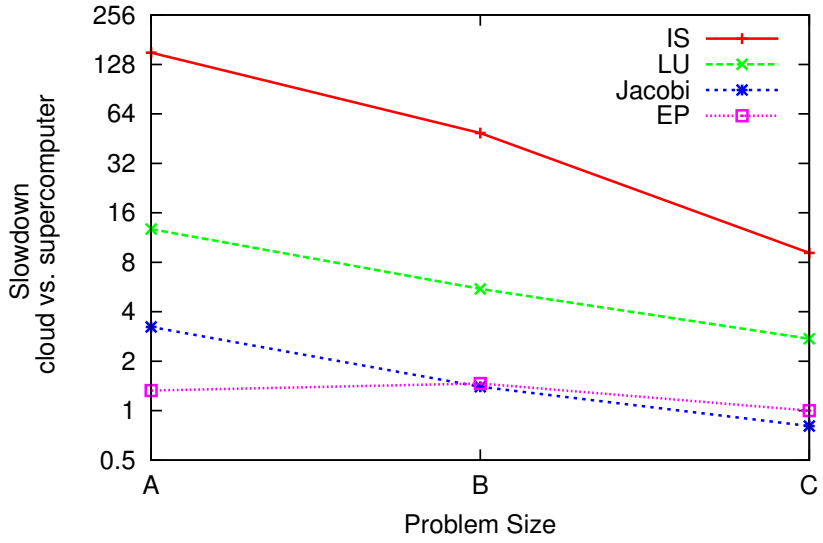



**Figure 2.10: Slowdown on cloud vs. supercomputer reduces with increasing problem size.**

To mitigate the virtualization overhead, we explore two optimizations which make clouds HPC-aware: lightweight virtualization and CPU affinity.

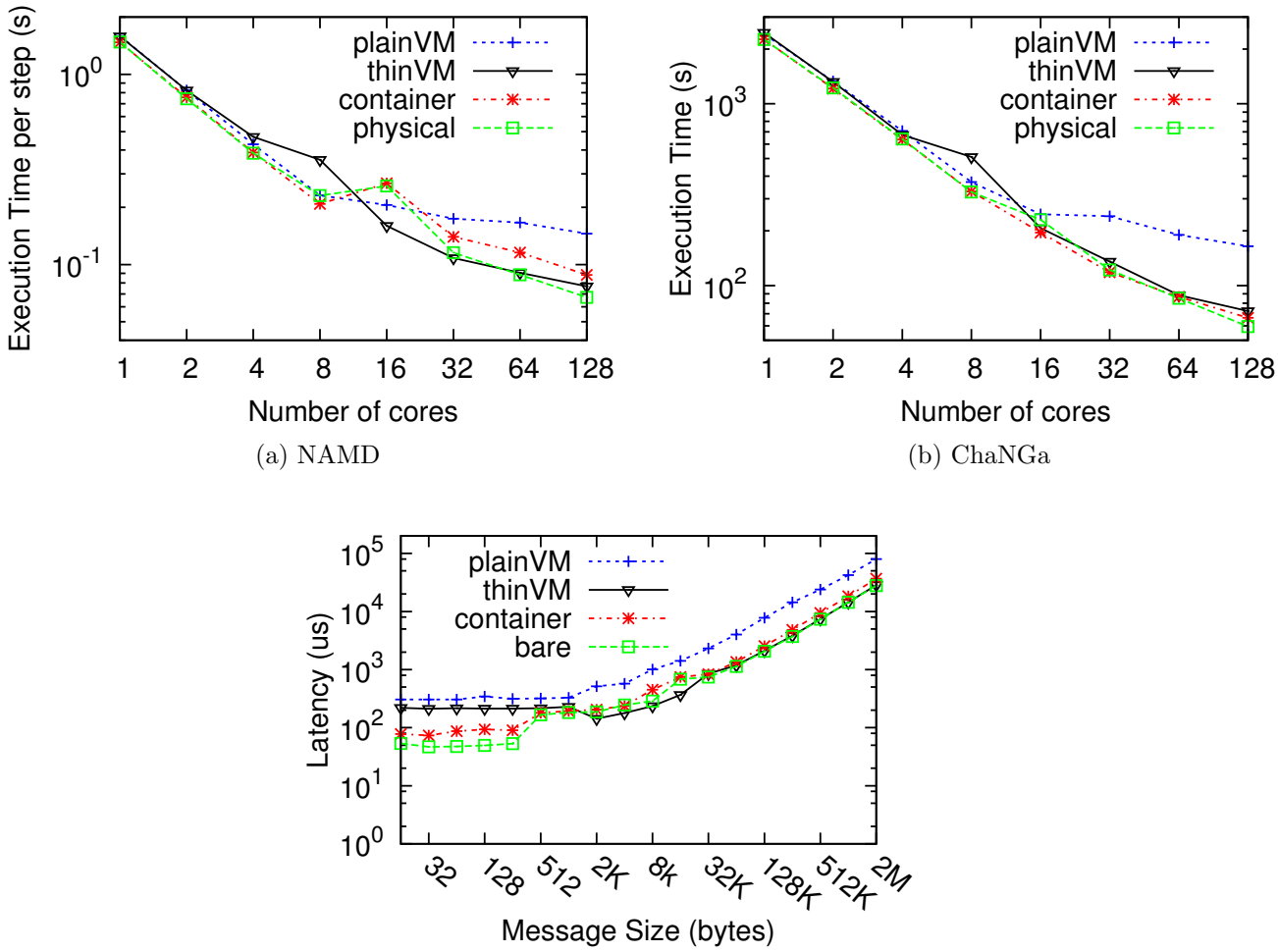
### 2.5.1 Lightweight Virtualization

We consider two lightweight virtualization techniques, *thin VMs* configured with PCI pass-through for I/O, and *containers*, that is OS-level virtualization. Lightweight virtualization reduces the latency overhead of network virtualization by granting VMs native accesses to physical network interfaces. In the thin VM configuration with IOMMU, a physical network interface is allocated exclusively to a VM, preventing the interface to be shared by the sibling VMs and the hypervisor. This may lead to under utilization when the thin VM generates insufficient network load. Containers such as LXC [35] share the physical network interface with its sibling containers and its host. However, containers must run the same operating system as their underlying host. Thus, there is a trade-off between resource multiplexing and flexibility offered by VM.

Table 2.4 first five columns (also visualized as Figure 2.11) validate that network virtualization is the primary bottleneck of cloud. These experiments were conducted on the virtualization testbed described earlier (Table 2.2). On plain VM, the scalability of NAMD and ChaNGa (Figure 2.11a–2.11b) is similar to that of private cloud (Figure 2.1). However, on thin VM, NAMD execution times closely track that of the physical machine even as

**Table 2.4: Impact of virtualization and CPU affinity settings on NAMD's performance**

| Cores | Execution Time per step (s) of NAMD for specific virtualization and affinity setting | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | bare | container | plain-VM | thin-VM | bare-appAFF | container-appAFF | plainVM-hyperAFF | plainVM-dualAFF | plainVM-appAFF | thinVM-hyperAFF | thinVM-dualAFF | thinVM-appAFF |
| 1 | 1.479 | 1.473 | 1.590 | 1.586 | 1.460 | 1.477 | 1.584 | 1.486 | 1.500 | 1.630 | 1.490 | 1.586 |
| 2 | 0.744 | 0.756 | 0.823 | 0.823 | 0.755 | 0.752 | 0.823 | 0.785 | 0.789 | 0.859 | 0.854 | 0.823 |
| 4 | 0.385 | 0.388 | 0.428 | 0.469 | 0.388 | 0.385 | 0.469 | 0.422 | 0.429 | 0.450 | 0.449 | 0.469 |
| 8 | 0.230 | 0.208 | 0.231 | 0.355 | 0.202 | 0.203 | 0.355 | 0.226 | 0.228 | 0.354 | 0.244 | 0.355 |
| 16 | 0.259 | 0.267 | 0.206 | 0.160 | 0.168 | 0.197 | 0.227 | 0.166 | 0.189 | 0.186 | 0.122 | 0.160 |
| 32 | 0.115 | 0.140 | 0.174 | 0.108 | 0.079 | 0.082 | 0.164 | 0.141 | 0.154 | 0.106 | 0.079 | 0.108 |
| 64 | 0.088 | 0.116 | 0.166 | 0.090 | 0.079 | 0.071 | 0.150 | 0.184 | 0.195 | 0.089 | 0.066 | 0.090 |
| 128 | 0.067 | 0.088 | 0.145 | 0.077 | 0.062 | 0.056 | 0.128 | 0.154 | 0.166 | 0.074 | 0.051 | 0.077 |




(a) NAMD

(b) ChaNGa

(c) Ping-Pong Test

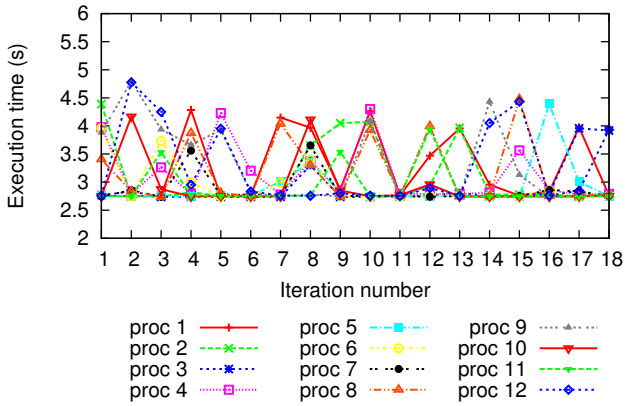
**Figure 2.11: Impact of Virtualization on Application Performance**

multiple nodes are used (i.e., 16 cores onwards). The performance trend of containers also resembles the one of the physical machine. This demonstrates that thin VM and containers impose a significantly lower communication overhead. This low overhead is further validated by the ping-pong test (Figure 2.11c).

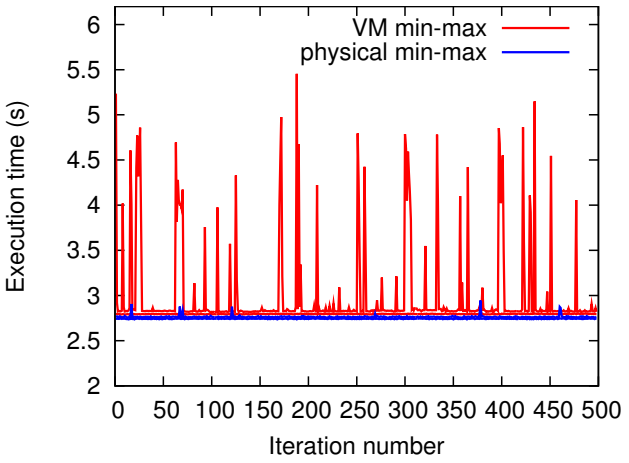
We note that there are other HPC-optimized hypervisors [49, 50]. However, an exhaustive comparison of hypervisors is not our intention. Our goal is to focus on the current state of device virtualization and provide valuable insights to cloud operators.

### 2.5.2 Impact of CPU Affinity

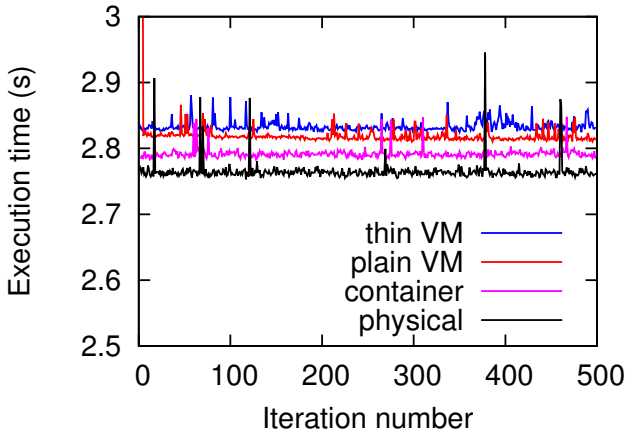
CPU affinity instructs the operating system to bind a process (or thread) to a specific CPU core. This prevents the operating systems to inadvertently migrate a process. If all important processes have non-overlapping affinity, it practically prevents multiple processes or threads to share a core. In addition, cache locality can be improved by processes or threads remaining on the same core throughout their execution. However, in the cloud, CPU affinity can be enforced at the *application level*, which refers to binding processes to the virtual CPUs of a VM, and at the *hypervisor level*, which refers to binding virtual CPUs to physical CPUs.




(a) Physical Machine (No CPU Affinity)




(b) Thin VM (Application-level Affinity only) and Physical Machine (CPU Affinity)




(c) All Configurations (CPU Affinity)

**Figure 2.12: Impact of CPU Affinity on CPU Performance**

In this experiment, we executed 12 processes on a single 12-core virtual or physical

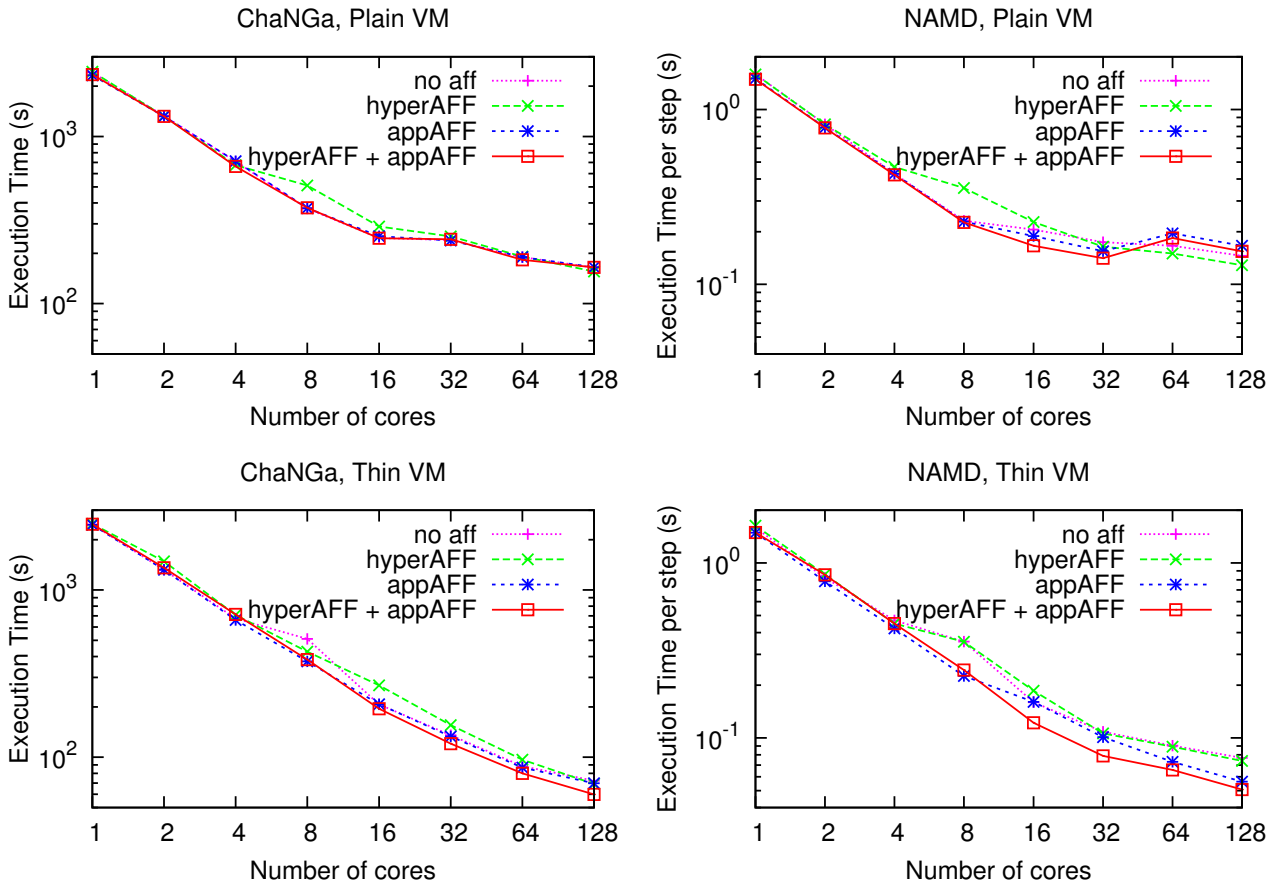



**Figure 2.13: Application performance with various CPU Affinity Settings, thin VM and plain VM; legend is at the bottom**

machine. Each process runs 500 iterations, where each iteration executes 200 millions of $y = y + rand()/c$ operations. Without CPU affinity (Figure 2.12a), we observe wide fluctuation on the process execution times, up to over twice of the minimum execution time (i.e., 2.7s). This clearly demonstrates that frequently two or more of our benchmark processes are scheduled to the same core. The impact of CPU affinity is even more profound on virtual machines: Figure 2.12b shows the minimum and maximum execution times of the 12 processes with CPU affinity enabled on the physical machine, while only application-level affinity is enabled on the thin VM. We observe that the gap between minimum and maximum execution times is narrowed, implying that load balance takes effect. However, on the thin VM, we still notice the frequent spikes, which is attributed to the absence of hypervisor-level affinity. Hence, even though each process is pinned to a specific virtual CPU core, multiple virtual cores may still be mapped onto the same physical core. When hypervisor-level affinity is enabled, execution times across virtual cores stabilizes close to

those of the physical machine (Figure 2.12c).

In conducting these experiments, we have learned several lessons. Firstly, virtualization introduces a small amount of computation overhead, where the execution times on containers, thin VM, and plain VM are higher by 1–5% (Figure 2.12c). We also note that it is crucial to minimize I/O operations unrelated to applications to attain the maximum application performance. Even on the physical machine, the maximum execution time is increased by 3–5% due to disk I/O generated by the launcher shell script and its `stdout/stderr` redirection. The spikes on the physical machine in Figure 2.12c are caused by short `ssh` sessions which simulate the scenarios where users log in to check the job progress. Thus, minimizing the unrelated I/O is another important issue for HPC cloud providers to offer maximum performance to their users.

Figure 2.13 and Table 2.4 show the positive impact of CPU affinity on thin VM and plain VM. HyperAFF denotes the execution where hypervisor-level affinity is enabled. Similarly appAFF means application-level affinity is enabled. Significant benefits are obtained for thin-VM, when using both application-level and hypervisor-level affinity compared to the case with no affinity. However, the impact of CPU affinity on NAMD on plain VMs is not clear, which indicates that optimizing cloud for HPC is non-trivial. Then the question is why and when should one move to cloud?

### 2.5.3 Network Link Aggregation

Even though network virtualization cannot improve network performance, an approach to reduce network latency using commodity Ethernet hardware is to implement link aggregation and a better network topology. Experiments from [51] show that using 4-6 aggregated Ethernet links in a torus topology can provide up to 650% improvement in overall HPC performance. This would allow cloud infrastructure using commodity hardware to improve raw network performance. Software Defined Networking (SDN) based on open standards such as Openflow, or similar concepts embedded in the cloud software stack, can be used to orchestrate the link aggregation and Vlan isolation necessary to achieve such complex network topologies on an on-demand basis. The use of SDN for controlling link aggregation is applicable to both bare-metal and virtualized compute instances. However, in a virtualized environment, SDN can be integrated into network virtualization to provide link aggregation to VM transparently.

## 2.6 HPC Economics in the Cloud

There are several reasons why many commercial and web applications are migrating to public clouds from fully owned resources or private clouds: variable usage in time resulting in lower utilization, trading CAPEX (capital expenditure) for OPEX (operating expenditure), and the shift towards a delivery model of Software as a Service. These arguments apply both to cloud providers and cloud users. Cloud users benefit from running in the cloud when their applications fit the profile we described e.g., variable utilization. Cloud providers benefit if the aggregated resource utilization of all their tenants can sustain a profitable pricing model when compared to the substantial upfront investments required to offer computing and storage resources through a cloud interface.

### 2.6.1 Why *not* cloud for HPC:

HPC is however quite different from the typical web and service-based applications. (1) Utilization of the computing resources is typically quite high on HPC systems. This conflicts with the desirable property of low average utilization that makes the cloud business model viable. (2) Clouds achieve improved utilization through consolidation enabled by virtualization – a foundational technology for the cloud. However, as evident from our analysis, the overhead and noise caused by virtualization and multi-tenancy can significantly affect HPC applications' performance and scalability. For a cloud provider that means that the multi-tenancy opportunities are limited and the pricing has to be increased to be able to profitably rent a dedicated computing resource to a single tenant. (3) Many HPC applications rely on optimized interconnect hardware to attain best performance, as shown by our experimental evaluation. This is in contrast with the commodity Ethernet network (1Gbps today moving to 10Gbps) typically deployed in most cloud infrastructures to keep costs small. When networking performance is important, we quickly reach diminishing returns of scaling-out a cloud deployment to meet a certain performance target. If too many VMs are required to meet performance, the cloud deployment quickly becomes uneconomical. (4) The CAPEX/OPEX argument is less clear for HPC users. Publicly funded supercomputing centers typically have CAPEX in the form of grants, and OPEX budgets may actually be tighter and almost fully consumed by the support and administration of the supercomputer with little headroom for cloud bursting. (5) Software-as-a-service offering are also rare in HPC to date, although that might change in the future.

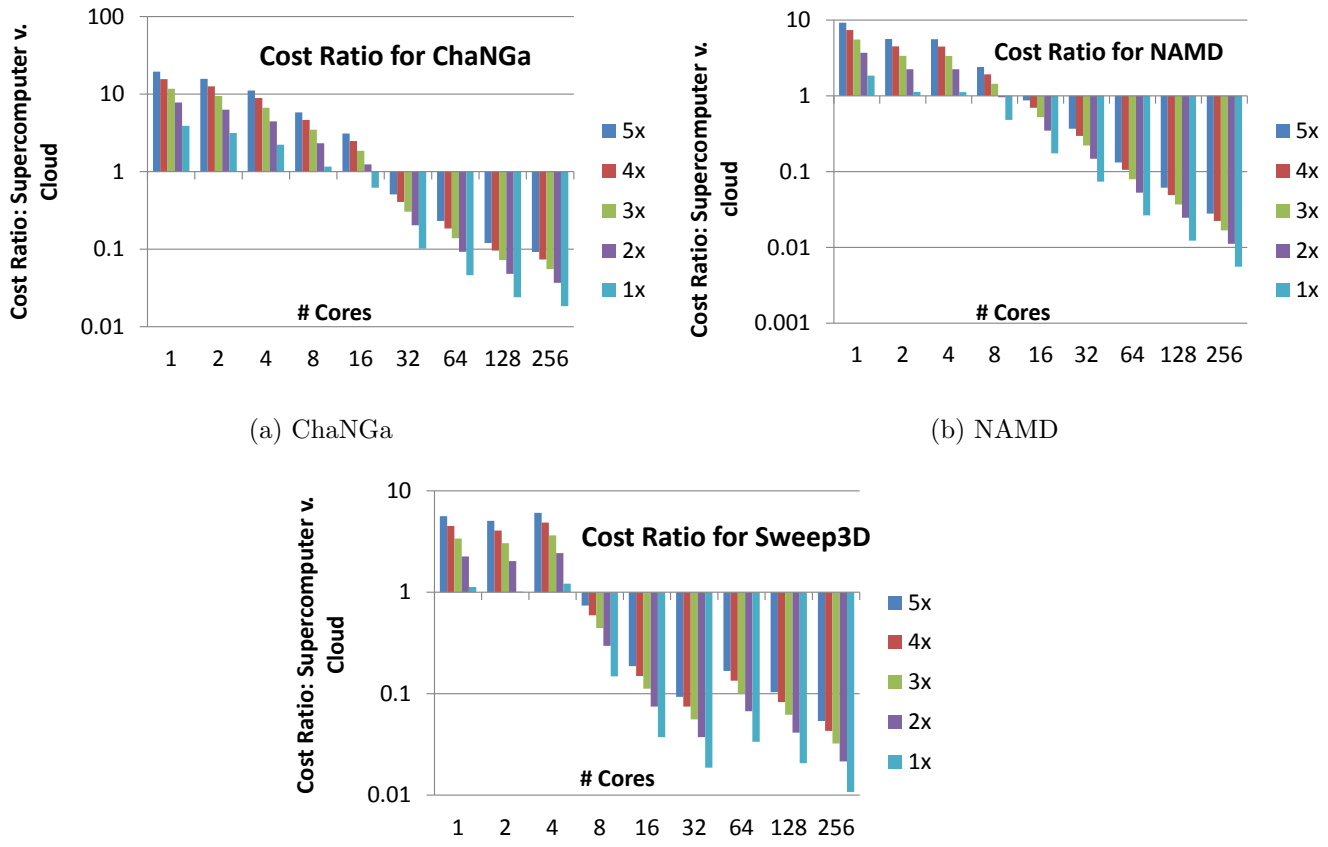
### 2.6.2 Why cloud for HPC:

So, what are the conditions that can make HPC in the cloud a viable business model for both, HPC users and cloud providers? Unlike large supercomputing centers, HPC users in small-medium enterprises are much more sensitive to the CAPEX/OPEX argument. These include startups with nascent HPC requirements (e.g., simulation or modeling) and small-medium enterprises with growing business and an existing HPC infrastructure. Both of them may prefer the pay-as-you-go approach in clouds vs establishing/growing on-premise resources in volatile markets. Moreover, the ability to leverage a large variety of heterogeneous architectures in clouds can result in better utilization at global scale, compared to the limited choices available in any individual organization. Running applications on the most economical architecture while meeting the performance needs can result in savings for consumers.

### 2.6.3 Quantifiable Analysis

To illustrate a few possible HPC-in-the-cloud scenarios, we collected and compared cost and price data of supercomputer installations and typical cloud offerings. Based on our survey of cloud prices, known financial situations of cloud operators, published supercomputing costs, and a variety of internal and external data sources [52], we estimate that a cost ratio between 2x and 3x is a reasonable approximate range capturing the differences between a cloud deployment and on-premise supercomputing resources today. In our terminology, 2x indicates the case where 1 supercomputer core-hour is twice as expensive as 1 cloud core-hour. Since these values can fluctuate, we expand the range to [1x–5x] to capture different future, possibly unforeseen scenarios.

Using the performance evaluations for different applications (Figure 2.1), we calculated the cost differences of running the application in the public cloud vs. running it in a dedicated supercomputer (Ranger), assuming different per core-hour cost ratios from 1x to 5x. Figure 2.14 shows the cost differences for three applications, where values>1 indicate savings of running in the cloud and values<1 an advantage of running it on a dedicated supercomputer. We can see that for each application there is a scale in terms of the number of cores up to which it is more cost-effective to execute in the cloud vs. on a supercomputer. For example, for Sweep3D, NAMD, and ChaNGa, this scale is higher than 4, 8, and 16 cores respectively. This break-even point is a function of the application scalability and the cost ratio. However our observation is that there is little sensitivity to the cost ratio and it is relatively straightforward to determine the break-even point. This is true even for the cost




(a) ChaNGa

(b) NAMD

(c) Sweep3D

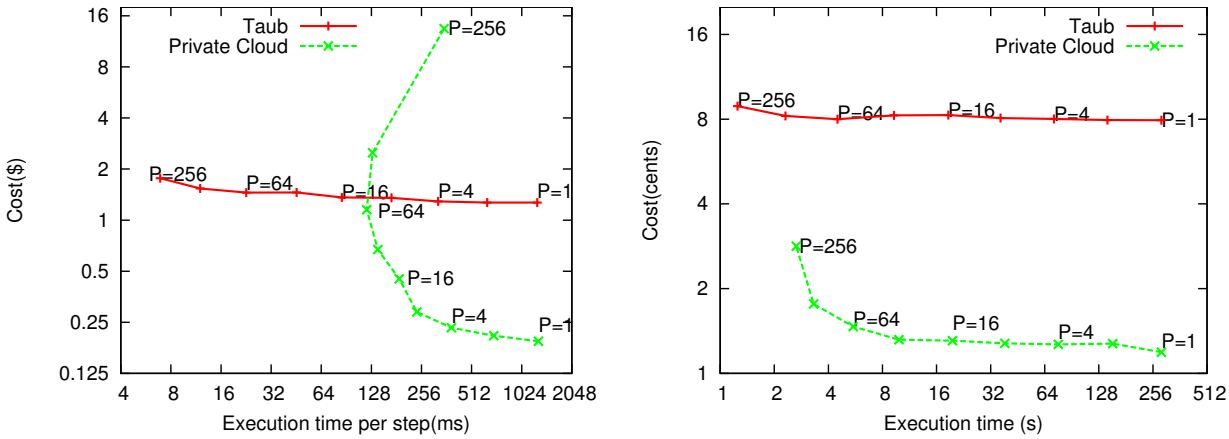
**Figure 2.14: Cost ratio of running in cloud and a dedicated supercomputer for different scale (cores) and cost ratios (1x–5x). Ratio>1 imply savings of running in the cloud, <1 favor supercomputer execution.**

ratio of 1. This might be the artifact of slower processors for the Ranger vs. newer and faster processors in the cloud.

### Cost as function of Performance

A direct comparison of the cost for running an application on Supercomputer vs cloud for identical number of processors can be sometimes unclear since they can achieve different sequential performance and different speedups for same value of P. A fairer comparison would be to study cost as a function of execution time.

In this section, we evaluate the cost-performance tradeoff of running an HPC application on private cloud vs HPC-optimized platform, in our case, Taub. We use a simple charging based cost model to evaluate the cost of running an HPC application. For cloud, we use a




(a) NAMD (b) NQueens

**Figure 2.15: Cost vs. Execution Time for two applications on Taub and Private Cloud. We see two different patterns here - for NAMD, it is better to run on Taub (except for small scale) whereas for NQueens, Cloud is the optimal platform**

charging rate of \$0.15 per core hour (Amazon EC2 pricing model for similar hardware). For Taub, we make a reasonable and conservative assumption for a charging rate of \$1.00 per core hour [53, 54]. Note that our primary interest is to observe the shape of cost-curve and not the actual values. With these charging rates, the cost of executing an application on P processors becomes

$$\$0.15 \times P \times T_{\text{Private cloud}}$$

for Private cloud and

$$\$1.00 \times P \times T_{Taub}$$

for Taub. Figure 2.15 shows this tradeoff for NAMD and NQueens. Each point on the curves represents a measurement from execution on the labeled number of processors. For all cases, cost increases as execution time decreases because of non-linear speedup. An ideal speedup would result in a flat cost curve. We note that for NAMD, its better to run on Taub (except for larger execution time) whereas for NQueens, Eucalyptus Cloud is the better platform. This difference can be attributed to poor scaling of NAMD on Cloud. Hence, we note that depending upon application characteristics (such as communication sensitivity) and user's preferences (cost, performance) or constraints (e.g limited budget or upper bound on execution time) it might be better to run on one platform in some scenarios and on the other in some different scenarios. Moreover, for the same application, the optimal platform can vary depending upon the desired performance.

### 2.6.4 Qualitative Discussion

Next, we address few economic topics on HPC in cloud.

We have primarily compared supercomputers vs. HPC in clouds. However, there are other alternatives that we discuss in Section 2.7, such as bursting out to cloud. Yet another alternative is outsourcing supercomputer. In many ways, we consider the latter similar to HPC in the cloud, with the exception of the way of use. Supercomputers are batch oriented while clouds offer dedicated use, at least at the virtualization level. There is also no reason why someone would not put a whole supercomputer behind cloud interfaces, and make it available on demand. Hence, these are variations of the key cases discussed in the paper. At the same time, current supercomputers are almost fully utilized, so there is little incentive to benefit from on-demand use of supercomputers, as compared to departmental level of servers, e.g. in Computer Aided Design (CAD) or Computer Aided Engineering (CAE) which can largely benefit from improved utilization.

In addition, cloud providers can offer the most recent equipment. Because they will share it among many customers they can amortize the high cost more easily than any single customer. This equipment can be used for exploration or in a production manner for early adopters. Movie rendering is a classic case of a cloud HPC (compute-intensive) application. Most recent case is post-production of the film "Life of Pi". Movie companies can always use the most recent equipment in the cloud and eventually acquire those that benefits them.

In this paper, we have not discussed accelerators, such as GPUs, which are becoming important for the HPC and compute-intensive applications. Because we have not conducted any experiments with GPUs, we cannot elaborate with any substance on the implications of the use of GPUs in the cloud. However, there is no reason not to treat them the same way as the regular compute instances. For example, Amazon prices them in the similar dedicated instances class with the price of $0.715 for GPU (g2.2xlarge) instance vs similarly sized (c3.2xlarge) compute instance for $0.462 per hour. The price difference is attributed to the hardware cost, the number of instances offered (many more compute than GPUs), and the power consumed.

One additional complication arising from the use of accelerators is that they do not virtualize well. While there is ongoing work making good progress in that direction, like NVidia GRID [55], it is still a young area with several unresolved issues. For example, the current sharing model of virtual GPUs is appropriate for concurrent execution of multiple jobs in a dedicated supercomputer, but does not provide the encapsulation, protection, and security support that would make it appropriate in the cloud. Any resource that does not virtualize at fine granularity poses a serious challenge to the cloud adoption model because it forces

the cloud provider to adopt a very rigid pricing scheme if the resource cannot be sliced for multiple concurrent users. We believe this is an interesting area for future research. Finally, we would like to conclude that it is was non-trivial to do a fair comparison of HPC in the cloud and supercomputers. For clouds, the prices are well documented and they are public information, however the costs are undocumented and they are proprietary and really a differentiator for each cloud provider. On contrary, the costs for supercomputers are well documented by owners while the prices are typically hidden and not publicized due to subsidies and price reduction. That was one of the primary reasons why we used range of cost ratios in Section 7.3 (Figure 2.14). Hence, the economic comparison needs to be taken conservatively.

## 2.7 Discussion: Cloud Bursting and Benefits

In the previous sections, we studied the performance-cost tradeoffs of running HPC applications on different platforms. These tradeoffs can guide us to optimize the mapping between applications and platforms. In this section, we discuss the case when the dedicated infrastructure cannot meet peak demands and the user is considering "cloud bursting" as a way to offload the peaks to the cloud. In this case, the knowledge of application and platform characteristics and their impact on performance can help answer (1) which applications from a set to burst to cloud, and (2) which cloud to burst to.

Consider (1), a simple allocation scheme may not even find a feasible solution, regardless of the cost. For example, first-come-first-served may exhaust the dedicated resources on cloud-friendly applications, and attempt bursting to the cloud, applications that do not scale and have no chance of meeting the performance target.

Knowledge of application characteristics can also help to answer (2), that is which cloud to select from the several commercially available options, each having different characteristics and pricing rates. For example, for some applications demonstrating good scalability within a given range, it would be cost effective to run on a low-cost ($ per core-hour) cloud. For other communication-intensive applications a higher-cost HPC-optimized cloud would be more effective.

Hence, we propose the co-existence of supercomputer and cloud with a two step methodology – 1) characterize applications using theoretical models, instrumentation, or simulation and 2) intelligently match applications to platforms based on user preferences. This approach is further explored in the next chapter.

## 2.8 Related Work

In this section, we summarize the related research on HPC in cloud, including performance evaluation studies.

### 2.8.1 Performance and Cost Studies of HPC on Cloud

Walker [20], followed by several others [13, 14, 21–24, 29, 56, 57], conducted the study on HPC in cloud using benchmarks such as NPB and real applications. Their conclusions can be summarized as:

- Primary challenges for HPC in cloud are insufficient network and I/O performance in cloud, resource heterogeneity, and unpredictable interference arising from other VMs [13, 14, 21, 23].
- Considering cost into the equation results in interesting trade-offs; execution on clouds may be more economical for some HPC applications, compared to supercomputers [14, 57–59].
- For large-scale HPC or for centers with large user base, cloud cannot compete with supercomputers based on the metric $/GFLOPS [13, 24].

In this chapter, we explored some of the similar questions from the perspective of smaller scale HPC users, such as small companies and research groups who have limited access to supercomputer resources and varying demand over time. We also considered the perspective of cloud providers who want to expand their offerings to cover the aggregate of these smaller scale HPC users.

Furthermore, our work explored additional dimensions: (1) With a holistic viewpoint, we considered all the different aspects of running in cloud – performance, cost, and business models, and (2) we explored techniques for bridging the gap between HPC and clouds. We improved HPC performance in cloud by (a) improving execution time of HPC in cloud and (b) by improving the turnaround time with intelligent scheduling in cloud.

### 2.8.2 Bridging the Gap between HPC and Cloud

The approaches taken to reduce the gap between traditional cloud offerings and HPC demands can be classified into two broad categories – (1) those which aim to bring clouds closer to HPC, and (2) those which want to bring HPC closer to clouds.

**Table 2.5: Findings and our approach to address the research questions on HPC in cloud**

| Question | Answers |
|---|---|
| Who | (1) Small and medium scale organizations, startups or growing businesses, which can benefit from pay-as-you-go model.<br>(2) Users with applications which result in best performance/cost ratio in cloud vs. other platforms. |
| What | (1) Applications with less-intensive communication patterns and less sensitivity to interference.<br>(2) Applications with performance needs that can be met at small to medium scale execution (in terms of number of cores). |
| Why | (1) Small-medium enterprises benefit from pay-as-you-go model since they are highly sensitive to CAPEX/OPEX argument.<br>(2) Clouds enables multiple organizations to access a large variety of shared architectures, leading to improved utilization. |
| How | (1) Technical approaches: (a) *Making HPC cloud-aware* e.g. tuning computational granularity and problem sizes, and (b) *making clouds HPC-aware* e.g. providing lightweight virtualization and enabling CPU affinity.<br>(2) Business models: Hybrid supercomputer–cloud approach with application-aware scheduling and cloud bursting. |

In this chapter, we presented techniques for both and showed that these two approaches can complement each other. For (1), We explored techniques in low-overhead virtualization, and quantified how close we can get to physical machine's performance for HPC workloads. There are other recent efforts on HPC-optimized hypervisors [49, 50]. Other examples of (1) include HPC-optimized clouds such as Amazon Cluster Compute [15] and DoE's Magellan [13] and hardware- and HPC-aware cloud schedulers (VM placement algorithms) [18,60].

The latter approach (2) has been relatively less explored, but has shown tremendous promise. Cloud-aware load balancers for HPC applications [61] and topology aware deployment of scientific applications in cloud [19] have shown encouraging results. In this chapter, we demonstrated how we can tune the HPC runtime and applications to clouds to achieve improved performance.

## 2.9 Lessons and Conclusions

Through a performance and economic analysis of HPC applications and a comparison on a range of platforms, we have shown that different applications exhibit different characteristics that make them more or less suitable to run in a cloud environment. Table 2.5 presents our

conclusions.

Although some of the findings are similar to the behavior of early Beowulf clusters, those clusters are quite different from today's clouds: processor, memory, and networking technologies have tremendously progressed. The appearance of virtualization introduces multi-tenancy, resource sharing and several other new effects. We believe our research will help better understand which applications are cloud candidates, and where we should focus our efforts to improve the performance. Next, we summarize the lessons learned from this research and the emerging future research directions.

*Clouds can potentially complement supercomputers, but using clouds to substitute supercomputers is infeasible.* By using an underutilized resource which is "good enough" to get the job done sooner and more cheaply, it is possible to get better performance for the same cost on one platform for some applications, and on another platform for another application. More work is needed to better quantify the "good enough" dimension, as well as the deep ramification of cloud business models on HPC.

*For efficient HPC in cloud, HPC need to be cloud-aware and clouds needs to be HPC-aware.* HPC applications and runtimes must adapt to minimize the impact of slow network, heterogeneity, and multi-tenancy in clouds. Simultaneously, clouds should minimize overheads for HPC using techniques such as lightweight virtualization and link aggregation with HPC-optimized network topologies. With low-overhead virtualization, web-oriented cloud infrastructure can be reused for HPC. We envisage hybrid clouds that support both HPC and commercial workloads through tuning or VM re-provisioning.

*Application characterization for analysis of the performance-cost tradeoffs for complex HPC applications is a non-trivial task, but the economic benefits are substantial.* More research is necessary to quickly identify important traits for complex applications with dynamic and irregular communication patterns. A future direction is to evaluate and characterize applications with irregular parallelism [62] and dynamic datasets. For example, challenging data sets arise from 4D CT imaging, 3D moving meshes, and Computational Fluid Dynamics (CFD). The dynamic and irregular nature of such applications makes their characterization even more challenging compared to the regular iterative scientific applications considered in this paper. However, their asynchronous nature, i.e. lack of fine-grained barrier synchronizations, makes them promising candidates for heterogeneous and multi-tenant clouds.

CHAPTER 3

# Mapping HPC Applications to Platforms in Cloud

HPC clouds rapidly expand the application user base and the available platform choices to run HPC workloads: from in-house dedicated supercomputers, to commodity clusters with and without HPC-optimized interconnects and operating systems, to resources with different degrees of virtualization (full, CPU-only, none), to hybrid configurations that offload part of the work to the cloud. HPC users and cloud providers are faced with the challenge of choosing the optimal platform based upon a limited knowledge of application characteristics, platform capabilities, and the target metrics such as cost[1]. .

This trend results in a potential mismatch between the required and selected resources for HPC application. One possible undesirable scenario can result in part of the infrastructure being overloaded, and another being idle, which in turn yields large wait times and reduced overall throughput. Existing HPC scheduling systems are not designed to deal with these issues. Hence, novel scheduling algorithm and heuristics need to be explored to perform well in such scenarios.

In the previous chapter, we provided empirical evidence that applications behave quite differently on different platforms. This observation opens up several opportunities to optimize the mapping and scheduling of HPC jobs to platforms and pass the benefits to both cloud providers and end users. In our terminology, *mapping* refers to selecting a platform for a job, and *scheduling* includes mapping and deciding when to execute the job on the chosen platform. Here, we first research techniques to perform intelligent scheduling from cloud provider's perspective and then consider the problem from a user's perspective. In both cases, we evaluate the benefits of proposed approaches using simulation.

[1]Some portions reprinted with permission from [29], ©2013 IEEE and [57], ©2012 ACM

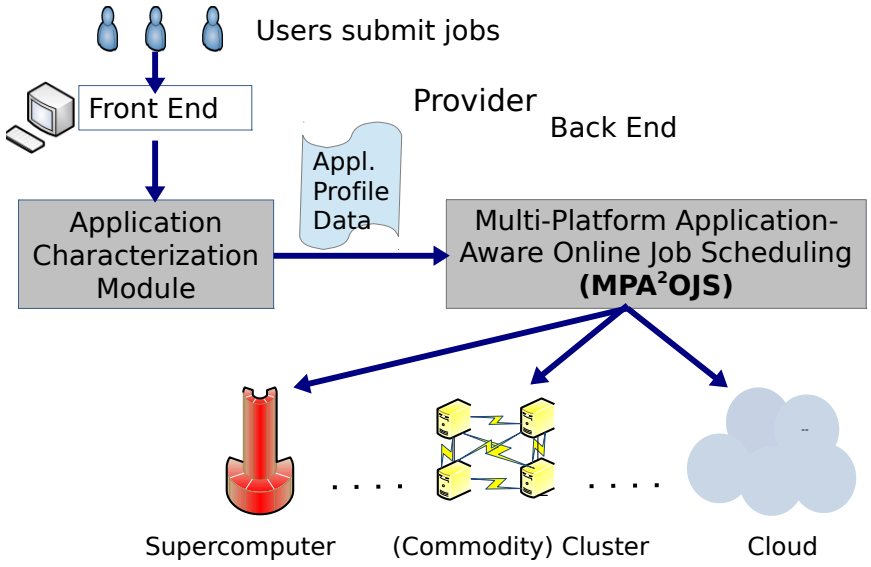



**Figure 3.1: Application-Aware Online Job Scheduling in presence of multiple platforms with different processor types, interconnection networks, and virtualization**

## 3.1 Cloud Provider Perspective: Problem Definition

Consider the case when the dedicated HPC infrastructure cannot meet peak demands and the provider is considering offloading jobs to additional available cluster or cloud. The problem can be defined as follows. Given a set of owned platforms with resources having different processor configurations, interconnection networks, and degrees of virtualization, how can we effectively schedule an incoming stream of HPC jobs to these platforms based on intrinsic application characteristics, job requirements, platform capabilities, and dynamic demand and load fluctuations to achieve the goals of improved job completion time, makespan, and hence throughput.

## 3.2 Scheduling Methodology and Heuristics

To address this problem, we adopt a two-step methodology, shown in Figure 3.1: 1) perform a one-time offline benchmarking or analytical modeling of applications-to-platforms performance sensitivity, and 2) use heuristics to schedule the application stream to available platforms based on the output of step 1 and current resource availabilities. In this thesis, our focus is on step 2, which translates to an online job scheduling problem with the additional complexity of having to decide *which platform* a job should be run on besides the decision

regarding *which job* to execute next. The problem is even more challenging since different application react differently to different platforms. We will refer to this as *Multi-Platform Application-Aware Online Job Scheduling ($MPA^2OJS$)*.

For step 1, we rely on one-time benchmarking of application performance on different platforms to obtain the performance data which can drive the scheduling decisions. In our earlier work, we have shown that in the absence of the benchmark data, it is possible to perform application characterization followed by relative performance prediction when considering multiple platforms [57]. Also, other known techniques for performance prediction can be used. These include analytical modeling, simulations, application profiling through sample execution (e.g. the first few iterations) on actual platform, and interpolation. It is not our intention in this thesis to research accurate techniques for parallel performance prediction of complex applications. Our goal is to quantify the benefits of $MPA^2OJS$ to develop an understanding and foundation for HPC in cloud, which can promote further research towards additional characterization and scheduling techniques.

Traditional HPC job scheduling algorithms do not consider the presence of multiple platforms. Hence, they are agnostic of the application to platform performance sensitivity. In $MPA^2OJS$, the mapping decision could be *static* or *dynamic*. Static decisions are independent of current platform load and made a-priori to job scheduling, whereas dynamic decisions are aware of the current resource availability and load and they are made when job is scheduled. With dynamic mapping, the same job can be scheduled to run on different platforms across its multiple executions depending upon the state of the system when it was scheduled.

Hence, $MPA^2OJS$ algorithms can be classified as static vs. dynamic, or job-characteristics aware vs. unaware. Next, we present some heuristics for $MPA^2OJS$.

### 3.2.1 Static Mapping Heuristics

The analysis in Chapter 2 (Section 2.3) showed that the slowdown in cloud vs. supercomputer depends on the application under consideration. Also, for the same application, the sensitivity to a platform varies with scale, i.e, with core counts. To visualize the behavior of HPC jobs along these two dimensions, i.e, application type and scale, we used our performance data to generate the map of a job's slowdown when running on the commodity cluster, i.e. Open Cirrus (Figure 3.2a) and private cloud (Figure 3.2b) with respect to its execution on supercomputer (Ranger).

In Figure 3.2, each grid cell represents the execution of a particular application (x-axis) at

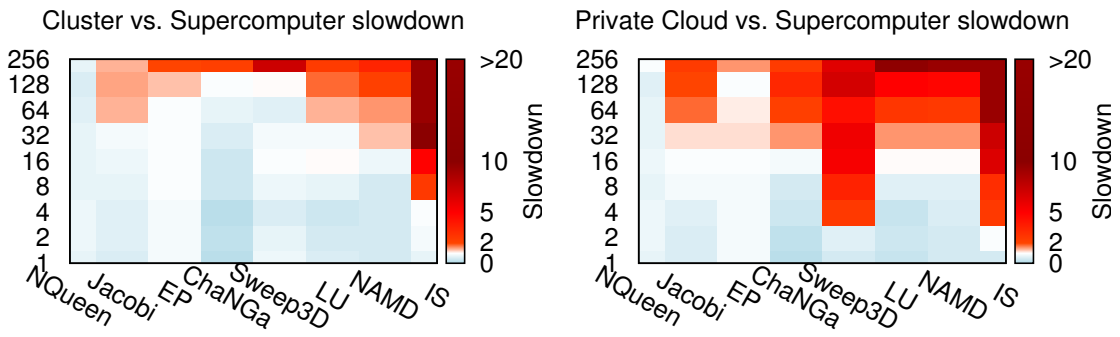



**Figure 3.2: Slowdown map: Effect of application and scale on slowdown. Light (Blue, white): no slowdown, Darker colors (orange, red) represent more slowdown.**

a particular scale (number of cores on y-axis). Here, light colors (blue and white) represent that job suffered no slowdown. In some cases, job attained speedup due to worse sequential performance on Ranger compared to the other platforms. Dark (reddish) shades represent slowdown. Based on Figure 3.2, two possible heuristics for static mapping are:

- *ScalePartition*: Assign large scale jobs (say 64–256 cores) to supercomputer, medium scale (16–32 cores) to cluster, and small scale (1–8 cores) to cloud by partitioning the slowdown map along y-dimension.
- *ApplicationPartition*: Assign specific applications to specific platforms by partitioning the map along x-dimension. E.g. IS, NAMD, and LU to supercomputer, ChaNGa and Sweep3D to cluster, and EP, Jacobi, and NQueens to cloud. A variation of *ApplicationPartition* can be to use finer application characteristics such as message count and volume for partitioning (Chapter 2, Table 2.3 in Section 2.3).

Other examples of static policies include scheduling all jobs to a supercomputer *(SCOnly)*, to a cluster *(ClusterOnly)*, or to a cloud *(CloudOnly)*.

### 3.2.2 Dynamic Mapping Heuristics

The motivation for dynamic selection of a platform for a job is to perform resource availability driven scheduling. Some such heuristics that we explored are:

- *MostFreeFirst*: Assign the current job to least loaded platform

**Table 3.1: Classification of heuristics for $MPA^2OJS$**

| Heuristic | Dynamic | Application-aware |
|---|---|---|
| SCOnly | | |
| ClusterOnly | | |
| CloudOnly | | |
| ScalePartition | | Yes |
| ApplicationPartition | | Yes |
| RoundRobin | Yes | |
| BestFirst | Yes | |
| MostFreeFirst | Yes | |
| Adaptive | Yes | Yes |

- *RoundRobin*: Assign jobs to platforms in round-robin fashion
- *BestFirst*: Assign the current job to platform with best available resources. E.g. in the order supercomputer, cluster, and cloud.
- *Adaptive*: Assign the job to the platform with largest *Effective Job-specific Supply (EJS)*.

*EJS* is defined to capture both, current resource availability and a job's suitability to a particular platform. We define a platform's *EJS* for a particular job as the product of free cores on that platform and the job's normalized performance obtained on that platform. The intuition is that the core-hours taken for a job to complete on a platform are directly proportional to the slowdown it suffers on that platform compared to the supercomputer. Hence, the Adaptive heuristic optimizes along two dimensions: it balances load across multiple platforms and it matches application characteristics to platforms. In contrast, the first three dynamic heuristics are application-agnostic.

Table 3.1 classifies our heuristics into static vs. dynamic, and application-aware vs. application-agnostic.

## 3.3 Implementation and Evaluation using CloudSim

We implemented the $MPA^2OJS$ heuristics in CloudSim [26], which is a widely used tool for simulation of scheduling algorithms in a data center or a cloud. We modified CloudSim to enable simulation of HPC job scheduling across multiple platforms.

In CloudSim, a fixed number of VMs are created at the start of simulation, and jobs (cloudlets in CloudSim terminology) can be submitted to these VMs. For our simulation purpose, a one-to-one mapping of cloudlets to VMs is sufficient but we needed to provide dynamic VM creation and termination. Also, we extended existing VM allocation policy in CloudSim to enable first come first serve (FCFS) scheduling of HPC jobs to resources.

The scheduling of cloudlets is performed by the datacenter broker. Hence, we created a new datacenter broker to perform $MPA^2OJS$. We introduced a periodic event in CloudSim, which checks for new job arrivals. The scheduler is triggered when a new job arrives or when a running job completes. Based on the current state of available datacenters and the scheduling heuristic, new jobs are assigned to a specific datacenter queue. Internally within a datacenter, FCFS policy is honored.

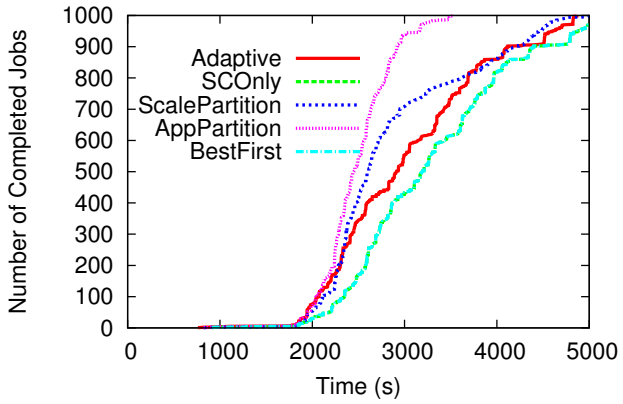
### 3.3.1 Simulation Approach

For our simulation, we created three datacenters – supercomputer, cluster, and cloud (256 cores each). These correspond to Ranger, Open Cirrus (typical commodity cluster), and private cloud (typical hypervisor-based resources) respectively. We simulated the execution of first 1000 jobs corresponding to the METACENTRUM-02.swf job logs of parallel workload archive [63]. Each job record contains the job's arrival time, requested number of cores (P), and its runtime. However, our goal is to simulate multiple platforms, where the runtime will vary from one platform to the other. Hence, we used a uniform distribution random number generator to map each job to one of the applications from the set evaluated in Chapter 2. We modified the job records to contain the application name (AppName) and the normalized performance for various platforms corresponding to (AppName,P). We used the same seed for random number generator while comparing different heuristics.

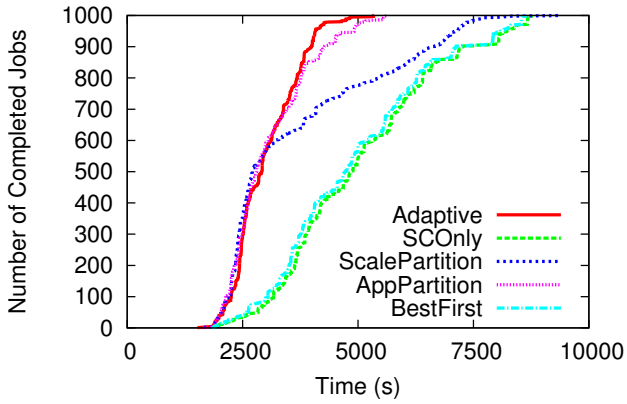
Furthermore, to evaluate how our heuristics perform under varying system load, we modified the runtimes of jobs in the log file. *Medium* load represents the original runtimes, *low* load represent runtimes scaled down by 2X, and *high* load represent runtimes scaled up by 2X.

### 3.3.2 Results: Makespan and Throughput

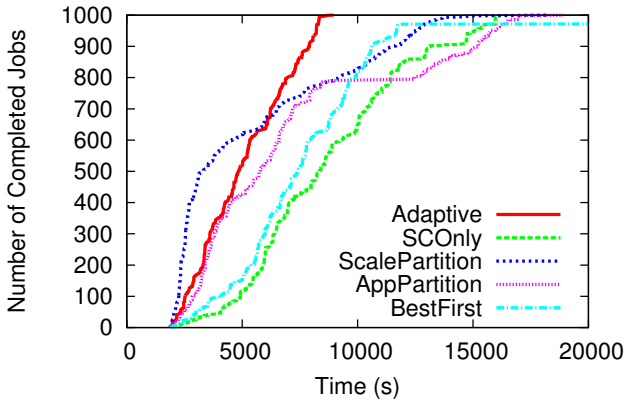
Using simulation we found that most application-agnostic strategies of Table 3.1, specifically ClusterOnly, CloudOnly, RoundRobin, and MostFreeFirst, performed very poorly compared to other heuristics. This is attributed to the tremendous slowdown that some application




(a) Low Load




(b) Medium Load




(c) High Load

**Figure 3.3: Adaptive heuristic significantly improves makespan and throughput when system is reasonably loaded**

suffer when running on cloud vs. their execution on supercomputer (e.g. up to 400X for IS). Hence, we present and analyze results of the remaining five heuristics, which yield more reasonable solutions.

Figure 3.3 compares the makespan (total completion time) and throughput for different heuristics under varying system load. It is evident from Figure 3.3 that Adaptive heuristic outperforms the rest when the system is reasonably loaded (medium and high load). Moreover, the benefits increase as the system load increases. Adaptive heuristic results in around 1.05X, 1.6X, and 1.8X improvement in makespan at low, medium, and high load respectively compared to SCOnly, that is running all applications on supercomputer. Similarly, there is significant improvement in throughput (number of completed jobs per second). For

instance, after 1 hour of execution (3600s), Adaptive strategy attains 1.25X, 4X, and 6X better throughput compared to SCOnly under different system loads. The benefits are even higher compared to other application-agnostic strategies, such as RoundRobin, as mentioned earlier. AppPartition performs well under low and medium load but yields poor results under high load. For better understanding of the reasons for the benefits and sensitivity to load, we measured various other metrics.

### 3.3.3 Breakdown Analysis of Benefits

Figure 3.4a shows that a potential cause of the benefits is the improvement in average response time (job's start time – arrival time). Adaptive, ScalePartition, and AppPartition achieve the most benefits in terms of response time. In some cases, AppPartition (at low and medium load) or ScalePartition (at low and high load) achieve even better response time compared to Adaptive. However, from Figure 3.3, we saw that overall Adaptive performed significantly better at medium and high load. This is because Adaptive performs the best in terms of average runtime in all three cases (loads) since ScalePartition and AppPartition are static mapping schemes (Figure 3.4b).

Static heuristics can not dynamically change the mapping of a job even if a better platform is available. Hence, high-end resources may be left unused waiting for a matching application to arrive. Also, on further investigation, we learned that 1-D characterization may not be sufficient since that can still result in some suboptimal mappings, e.g. ScalePartition maps IS at 32 cores to cluster even if supercomputer is free. For benefitting from multiple platforms, we need to a) consider both, the application characteristics and the scale at which it will be run, and b) dynamically adapt to the platform loads. Adaptive heuristic meets these two goals. We believe that the better performance of AppPartition compared to Adaptive heuristic at low load is an anomalous case.

Average turnaround times (Figure 3.4c), which are the sum of response times and runtimes, are consistent with the results of Figure 3.3. To ensure that our heuristics do not starve any job, we also calculated the maximum response time for any job. Figure 3.5 shows that our heuristics also improve the maximum response time.

### 3.3.4 How many cluster and cloud nodes to add?

Another research challenge is to determine how much additional cluster or cloud capacity a provider should add to an existing supercomputing facility to meet the demands under high

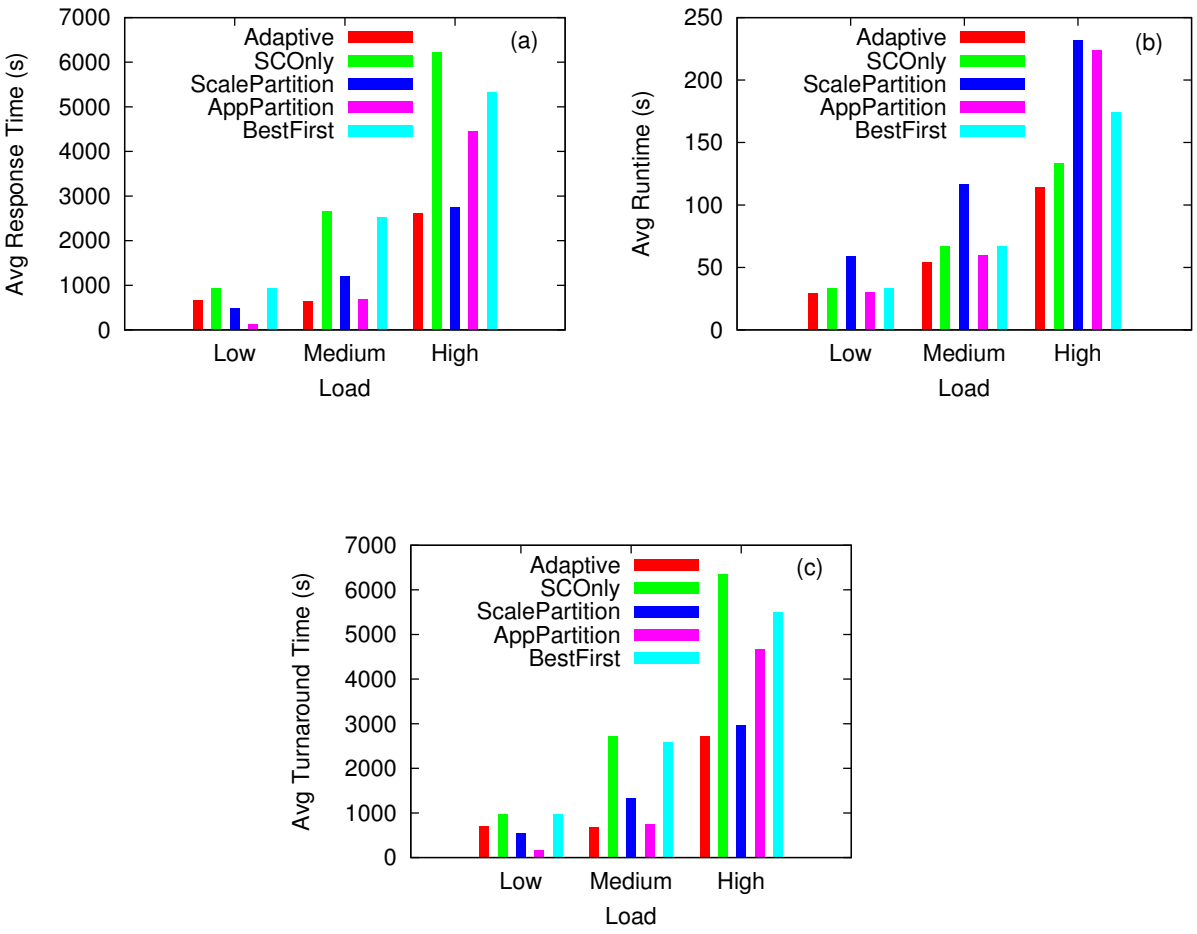



**Figure 3.4: Comparison and breakdown of benefits of scheduling heuristics. Adaptive performs the best.**

load. To this end, we simulated the execution of jobs under high load scenario keeping the number of supercomputer nodes constant (32), and varied the number of extra cluster and cloud nodes (each). Figure 3.6 shows that some of the benefits achieved by addition of more nodes tend to diminish beyond a point, e.g. makespan and maximum response time does not decrease much from 32 to 64 case. Using our simulation methodology, a provider can optimize the number of extra nodes to add to improve system metrics of interest.

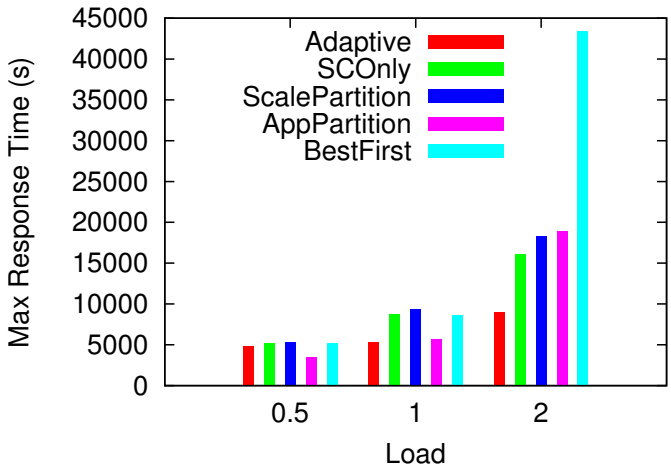



**Figure 3.5: Our heuristics ensures that jobs are not starved**

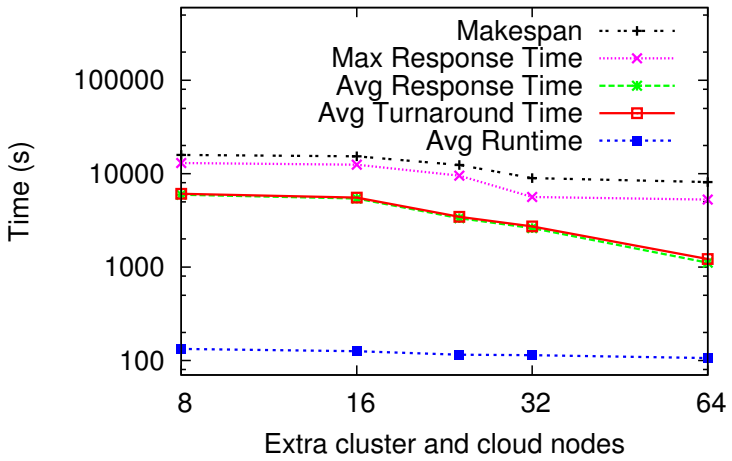



**Figure 3.6: Variation in benefits with incremental addition of cluster and cloud nodes to supercomputer**

## 3.4 Cloud User Perspective: Characterization and Mapping

Considering a user perspective, we developed a set of techniques (implemented in a tool) for mapping an incoming stream of applications to available platforms, and evaluated their benefits by comparing the target cost/performance metric of a naive mapping vs. a more intelligent mapping done with our proposed approach.

The conceptual architecture of our approach is presented in Figure 3.7. We start from an HPC application, and through characterization extract a *signature* capturing the most

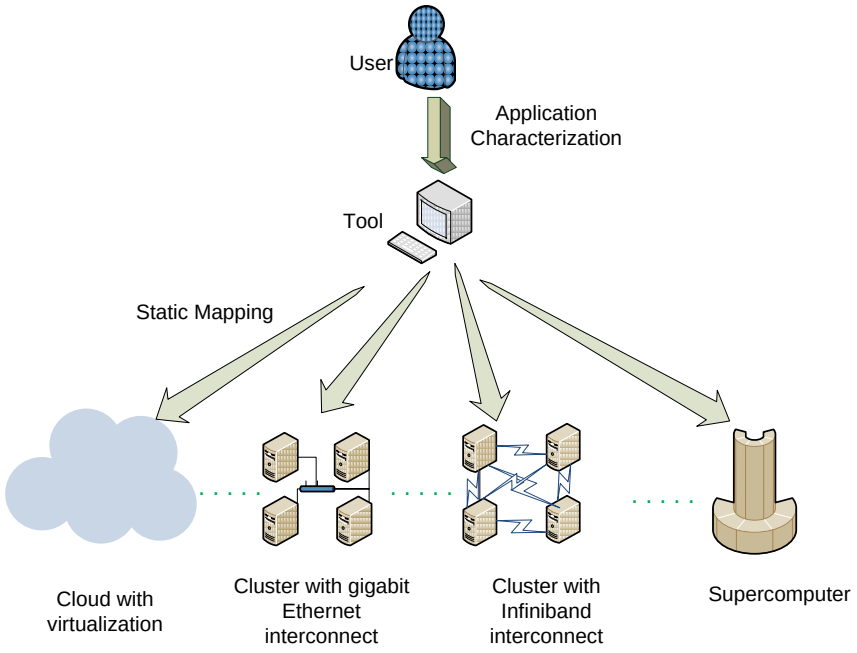



**Figure 3.7: Mapping of an application to a platform. We consider platforms with varying resources such as servers with different processor type and speed, different interconnection network and servers with and without virtualization.**

important dimensions. Subsequently, given a set of applications to execute and a set of target platforms, we define a set of heuristics to map the applications to the platforms that optimize *parallel efficiency* (static mapping in Figure 3.7). Parallel efficiency ($E$) is a crucial metric and a concept central to our approach towards application characterization. We had studied its impact in Chapter 2 and it was clear from Figure 2.4 that applications which achieve high parallel efficiency on supercomputer suffer less slowdown when moved to cloud compared with applications with low parallel efficiency. For mapping purposes, we focus on those application characteristics which can contribute to difference in application's expected parallel efficiency across different platforms. From the performance analysis, it is clear that communication time (and more precisely the communication-to-computation ratio) is a major contributor.

The two major components of our proposed solution are:

- *Application Characterization*: we extract a "signature" using application communication profiles, grain size, and problem size.

- *Static Mapping*: we apply a set of heuristics to maximize parallel efficiency assuming that the target platforms that we consider can vary in processor configuration, interconnection network, and virtualization environment.

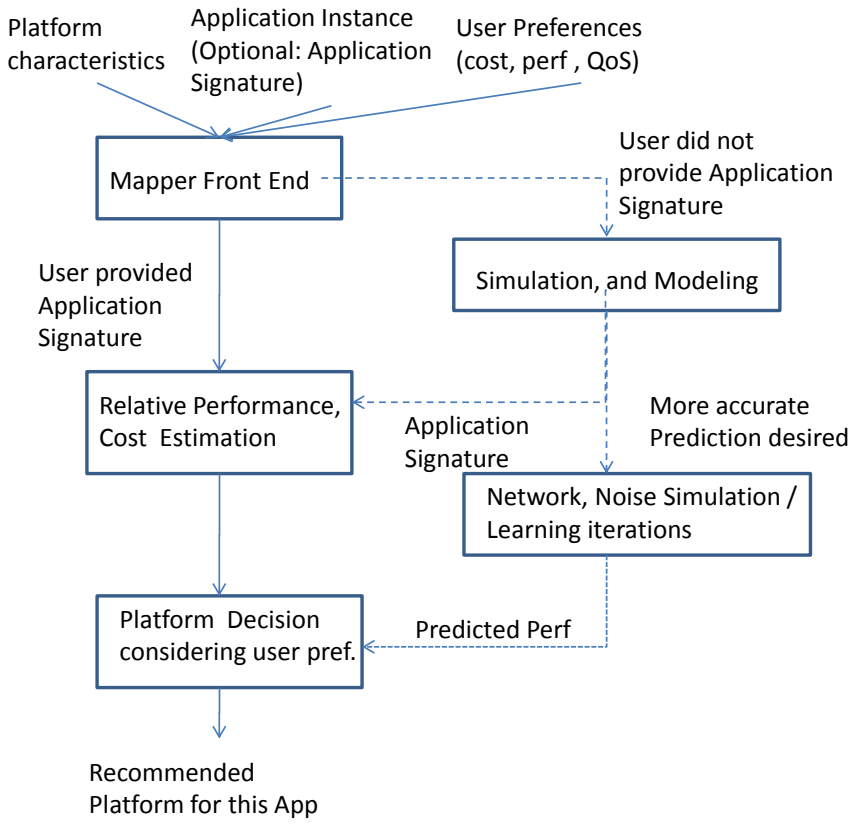



**Figure 3.8: Obtaining application characteristics and predicting performance for recommendations**

### 3.4.1 Designing the Mapper

Figure 3.8 shows a block diagram of our approach to the design and implementation of the *Mapper* tool. The inputs to the tool consist of:

**1. Platform Description**: basic server and interconnect parameters, such as CPU frequency, cores per node, memory per node, interconnect latency and bandwidth. These can be obtained by a one-time platform benchmarking pass, or through specification documents. The platform description also includes cost and power data, such as charging model (SU-based or money-based) and rate ($ per processor hour for money-based model), and typical power dissipation per processor at full speed and when idle.

**2. Application Instance Description**: application signature and problem size. Additionally, the number of physical processors (P) on non-virtualized machines, and virtual cores on virtualized machines, can be specified. Currently we do not consider the case where

```
AppSign
{
flops(N,P) [ ]  // Computation Time (FLOPS): Grain size
numMsg(N,P)[ ]  // Number of messages (array)
sizeMsg(N,P)[ ]  // Size of each message in bytes (array)
overlapFraction(N,P) [ ] // fraction of flops
                  //overlappable with communication
hasBarrier     // Is there a barrier after an iteration
needPeriodicLDB // Is periodic load balancing required
}
```

**Figure 3.9: Application signature. Contains the parameters crucial for determining mapping to a platform. N is representative of problem size and P denotes number of processors. flops, numMsg, sizeMsg represent array of functions of N and P.**

physical cores are shared between virtual cores, which is not a good match for HPC applications. Figure 3.9 shows the components of application signature. We chose these parameters as components of application signature since they are crucial to parallel application performance. Application signature is used to determine parallel efficiency with N and P as input, where N is representative of problem size. Expected parallel efficiency and sequential performance on different platforms can then be used for selecting the appropriate platform.

For simple applications and applications with regular communication structure, users can provide the application signature through an a priori understanding of the algorithm and its implementation. For complex application, we propose the use of simulation and modeling to estimate application signature for specific problem instances. Application profiling through sample execution (e.g. the first few iterations) on actual platform can also be used. Having obtained application profiles on different processor counts, we can use interpolation and curve-fitting techniques to estimate the application signature for complex applications. In this thesis, we consider proof-of-concepts applications where it is relatively easy to extract the application signature, and we ignore memory-bound and I/O-bound applications. To cover the full complexity of real world HPC scenarios, with the understanding and foundation that we present in this thesis, additional techniques for more accurate results and complex applications, such as those involving collective communication and periodic load balancing, can be developed in future.

**3. User Preferences** a vector of user weights to specify the relative importance of the target metrics, such as cost, service-level agreements, and quality of service (for example, expressed as average and standard deviation of execution time). Additional constraints, such

as limited budget, fixed SUs, or minimum performance requirements, can also be specified.

Using the application signature and platform characteristics, we estimate relative application performance, execution cost and other relevant metrics (such as $CO_2$ emissions for sustainability assessments) of the available platforms. The pseudo-code shown in Algorithm 1 estimates the normalized performance and cost for an application instance on a set of platforms, and recommends the best platform for a given application instance based on user preferences. For each platform, the algorithm estimates the communication to computation ratio (*cncpr*), calculates parallel efficiency (*pe*), scales it by sequential performance, and normalizes it. Subsequently, it calculates normalized cost and recommends the best platform based on user preferences. For estimating communication time ($T_{comm}$) of an $N$-byte message, we use the following (well-established) model:

$$T_{comm} = \alpha + N \times \beta$$

where $\alpha$ is the per-message cost (startup cost) and $\beta$ is the per-byte cost (inversely proportional of bandwidth).

The tool can also provide recommendations for mapping under various scenarios, and we implemented a set of additional algorithms to maximize performance under budget constraints, to minimize cost under performance guarantees, or to consider an application set as a whole instead of individual application instances. For example, with the group mapping algorithm, we can minimize cost of a collection of applications under a hard SU limitation while providing performance guarantees. Finally, the Mapper tool can be used to provide recommendations for scaling ranges, for applications (or groups of applications) that exhibit strong scaling (constant problem size, increasing processor count) and weak scaling (increasing problem sizes, increasing processor counts).

## 3.5 Benefits of Smart Mapping

To demonstrate the potential impact of such a tool and provide a proof-of-concept, we evaluate the results obtained by a simple mapper based on the characterization mentioned above. It is not our intention in this thesis to research accurate techniques for parallel performance prediction of complex applications. Our goal is to quantify the benefits of smart mapping to develop an understanding and foundation for HPC in cloud, which can promote further research towards additional techniques for more accurate results and complex applications. With this goal, we consider an application set with well-understood computation and communication patterns and available combination of two platforms: supercomputer

**Algorithm 1** Mapper for single application instance

1: Read Application Instance: $App(SIGN, N)$.
2: Read platforms (array IS). $numIS$ = number of platforms.
3: Read sequential runtimes on platforms (array $Seqperf$).
4: Read available processors. Call this array $Proc$.
5: Read User preferences. Call this $Pref$.
6: Evaluate sequential flops:
$SeqApp = App.EvaluateAppInstance(N, 1)$
7: $maxsuit = 0; maxindex = 1;$
8: **for** $i = 1$ to $i < numIS$ **do**
9: $curplat = IS[i]$
10: $App.evaluateAppInstance(N, Proc[i])$
11: $T_{comm} = 0, T_{comp} = 0$ // Comm, comp times
12: **for** $k = 1$ to $k = app.sizeMsg.size()$ **do**
13: $T_{comm} = T_{comm} + app.numMsg[k] * (curplat.a + app.sizeMsg[k]/curplat.b)$
14: $r_{comp} = r_{comp} + (app.flops[k]/seqapp.flops[k])$
15: **end for**
16: $T_{comp} = r_{comp} * Seqperf[i]$
17: $cncpr = T_{comm}/T_{comp}$
18: $pe = 1/(1 + cncpr)$
19: $perf = (pe * Proc[i])/Seqperf[k]$
20: **if** $k = 0$ **then**
21: $baseperf = perf$
22: **end if**
23: $normtime[i] = baseperf/perf$
24: $cost = normtime[i] * curplat.r * Procs[i]$
25: **if** $k = 0$ **then**
26: $basecost = cost$
27: **end if**
28: $normcost[i] = cost/basecost$
29: Calculate overall suitability as weighted average $suit[i] = 1/((normtime[i] * pref.wperf + normcost[i] * pref.wcost)/(pref.wcost + pref.wperf))$
30: **if** $suit[i] > maxsuit$ **then**
31: $maxsuit = suit[i]$
32: $maxindex = i$
33: **end if**
34: **end for**
35: Recommend platform is $IS[maxindex]$

(Ranger) and Eucalyptus cloud. From now on, we use "cloud" for Eucalyptus cloud (typical hypervisor-based resources), and "supercomputer" for Ranger. We consider the set of application instances shown in Table 3.2.

The IS benchmark contains all-to-all collective communication as its dominant communication pattern. Using a theoretical estimation for this collective primitive is non-trivial since

different MPI implementations use different broadcast algorithms which may even change with message sizes. Hence, we benchmarked the `MPI_Alltoall` for relevant message sizes for different processor counts on our platforms, and used these values in the mapper to predict parallel efficiency. Scientific applications are typically long-running, and are executed many times with different inputs, but same problem size. Such one-time benchmarking can be useful in those cases.

We will first consider mapping of individual applications to platforms and then mapping of a group of applications. In both cases, we execute the applications on the platform recommended by the mapper and analyze the results obtained. For the purpose of this study, we assume a charging rate of $1 per core-hour for Ranger and $0.15 per core-hour for cloud. However, we also study the effect of relative pricing on potential benefits of our approach to validate the usability of our methodology under different pricing ratios.

### 3.5.1 Accounting for User Preferences

The simplest scenario that we consider is when a user has access to a number P1 and P2 of processors on two platforms, and wants to run applications on the optimal platform in terms of cost/performance tradeoffs and preferences. For some users, cost may be the dominating factor, whereas for others performance may be more important. To study such variations, we evaluated the mapping for different performance/cost weight scenarios denoted by *Wt(performance weight, cost weight)*. For simplicity, let P1 = P2 = P. Table 3.2 shows the mapping suggested by our mapper. We can see that the recommended mapping varies with application, problem size, number of processors (scale), and user preference. Figure 3.10 shows the achieved performance and cost for different cases, normalized with respect to that obtained when all applications are executed on supercomputer. These values were obtained by performing normalization for each application, and then taking average across all applications. As the user weights shift towards cost, more applications are moved to cloud, normalized time increases, and normalized cost decreases. It is clear that there is a match between expected and achieved performance-cost tradeoffs which is not possible with random mapping. Even at the Wt(1,0) point, which represents a user preference inclined towards performance (performance weight = 1, cost weight = 0), we get better performance and cost using our mapping. This is due to the worse sequential performance on Ranger, and will not occur if there is a better processor on supercomputer. Instead, the important result is that there is only around 10-15% increase in execution time from Wt(1,0) to Wt(0,1) while the cost reduces by up to 60% of the original.

**Table 3.2: Mapping for different application instances under different user specified preferences (blue for cloud and red for supercomputer). The application suffix is the number of processors; for Jacobi, we consider multiple problem sizes, that is input matrix dimensions (e.g. size 1k × 1k).**

| Application | User Preference | | | | | | |
|---|---|---|---|---|---|---|---|
| | Wt (1,0) | Wt (0.75, 0.25) | Wt (0.5, 0.5) | Wt (0.25, 0.75) | Wt (0,1) | Fixed Perf | Fixed Cost |
| IS_classB_4 | | | | | | | |
| IS_classB_16 | | | | | | | |
| IS_classB_64 | | | | | | | |
| Jacobi1k_4 | | | | | | | |
| Jacobi1k_16 | | | | | | | |
| Jacobi1k_64 | | | | | | | |
| Jacobi2k_4 | | | | | | | |
| Jacobi2k_16 | | | | | | | |
| Jacobi2k_64 | | | | | | | |
| Jacobi4k_4 | | | | | | | |
| Jacobi4k_16 | | | | | | | |
| Jacobi4k_64 | | | | | | | |
| EP_classB_4 | | | | | | | |
| EP_classB_16 | | | | | | | |
| EP_classB_64 | | | | | | | |

### 3.5.2 Cost with Performance Guarantees

The research question that we explore here is whether we can reduce the cost of an application execution while maintaining the desired level of application performance. We use the same set of applications, with the target performance obtained by running the application on supercomputer. The Mapper estimates the number of processors required to achieve the same performance on cloud (typically larger than the number needed on supercomputer), and recommends the cost-optimal platform. Figure 3.11 shows the results for various supercomputer/cloud pricing ratios. We normalized cost with respect to execution on supercomputer, so that a normalized cost of 1 means that it is optimal to run the application on supercomputer for that pricing ratio (for example, that happens to all IS instances, regardless of the pricing ratio). The pricing ratio where the normalized cost $\neq 1$ is when the mapper recommends cloud as the optimal platform. Thus, we can deduce the pricing ratio at which the optimal platform shifts from supercomputer to cloud for each application. We see that this cross-over point occurs at pricing ratio of 1, 2, and 4, for different applications. Another important observation is that savings vary for different applications; for example, maximum savings were obtained for Jacobi4k_4 running on cloud. This observation is pivotal to intelligently map a group of applications under some constraints (e.g., a fixed SU budget) as discussed in Section 3.5.4.

Figure 3.12 shows the average performance gains for each pricing point (the labels show the crossover points). It is evident that we were able to get significant cost savings while meeting performance guarantees (normalized time under 1 in the figure, also indicative of accuracy of our mapper). Furthermore, we can deduce the maximum amount of savings of the intelligent mapping of an application set, irrespectively of the pricing ratio. The normalized cost curve flattens out at around 40% because some applications need to be executed on supercomputer regardless of the pricing ratio because of their inability to scale. It should also be noted that a combination of cloud and supercomputer results in better utilization of resources (see Figure 3.13).

### 3.5.3 Performance with Constrained Budget

Another common situation is an HPC user with hard budget constraints wanting to find the execution platform that maximizes performance. We can use the Mapper by constraining the budget of each application, for example with the cost of its execution on supercomputer, and using the fixed-cost mapping algorithm to estimate the number of processors that fit the budget, while taking into account the parallel efficiency drop as we scale. The Mapper

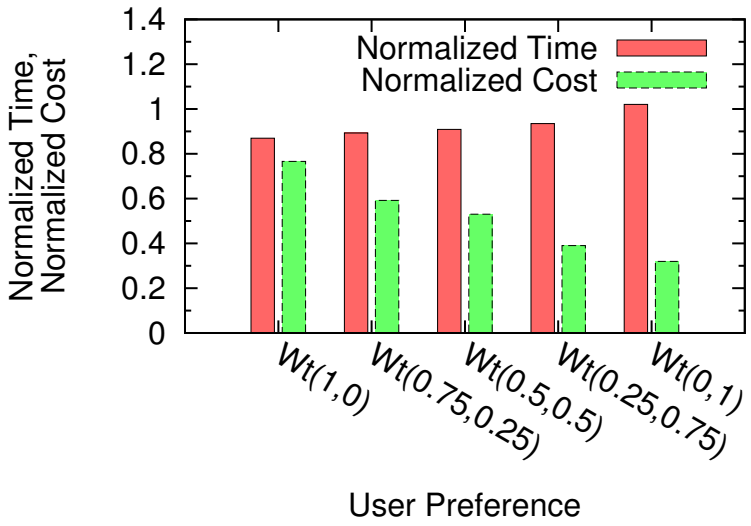



**Figure 3.10: Normalized Performance and Cost vs. execution on supercomputer for different user preferences. Figure shows there is a match between expected and achieved performance and cost.**

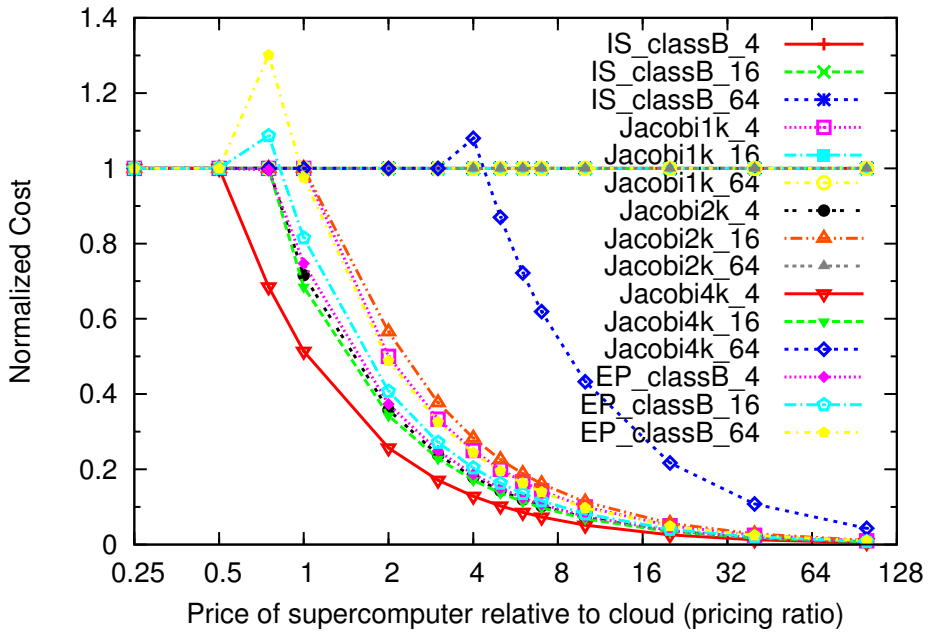



**Figure 3.11: Normalized Cost vs. execution on supercomputer vs. pricing ratio for FixedPerfMapper**

recommends the platform which would provide best performance, as we show in Figure 3.14

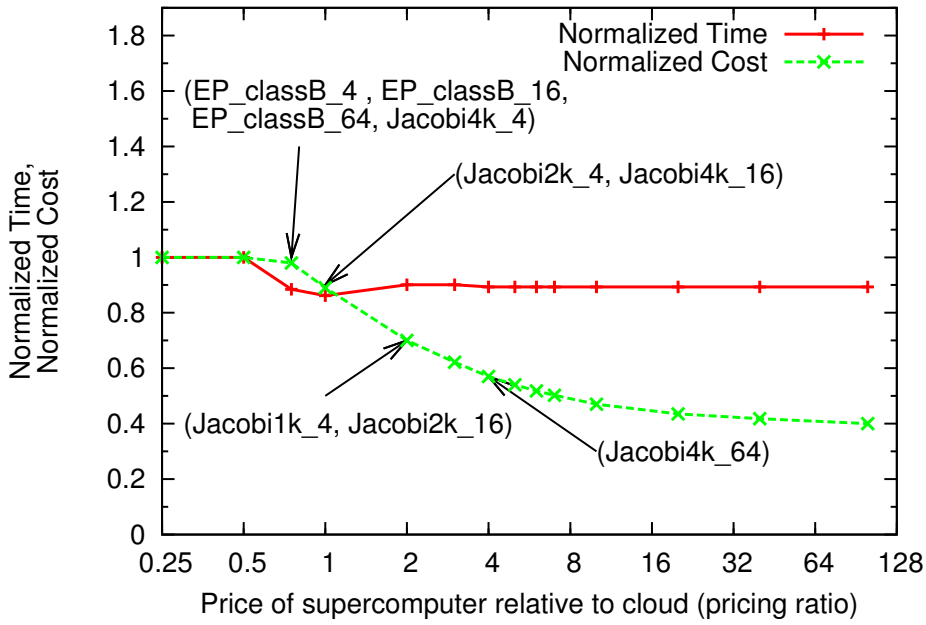



**Figure 3.12: Average Normalized Performance and Cost vs. execution on supercomputer vs. pricing ratio for FixedPerfMapper**

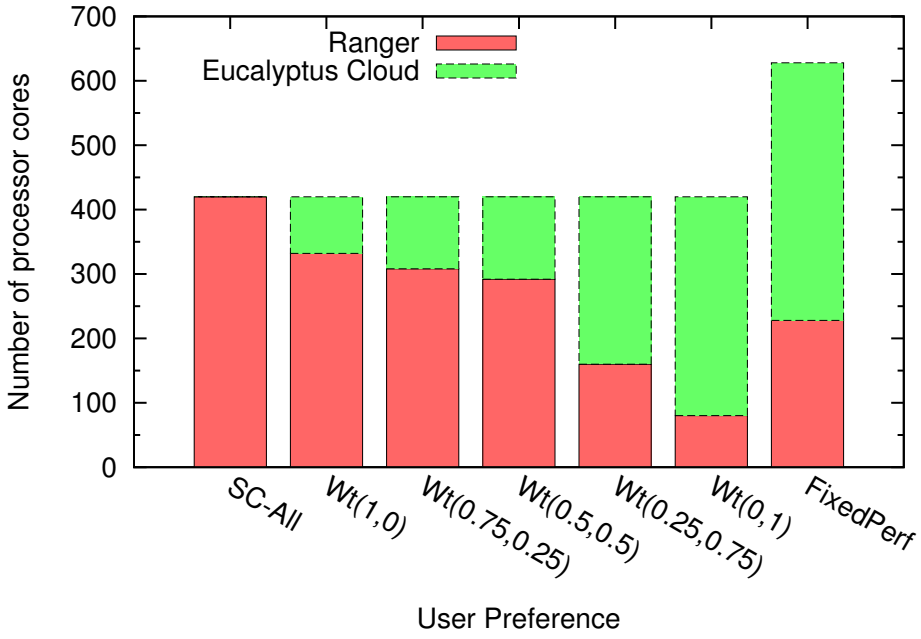



**Figure 3.13: All the platforms are utilized now**

where we summarize the achieved performance benefits under budget constraints. We did not consider EP (Embarrassingly Parallel) here since the cost curve of EP is close to horizontal (similar to NQueens in Figure 2.14 of Chapter 2) and hence it was not possible to run it on the large processor counts suggested by the mapper.

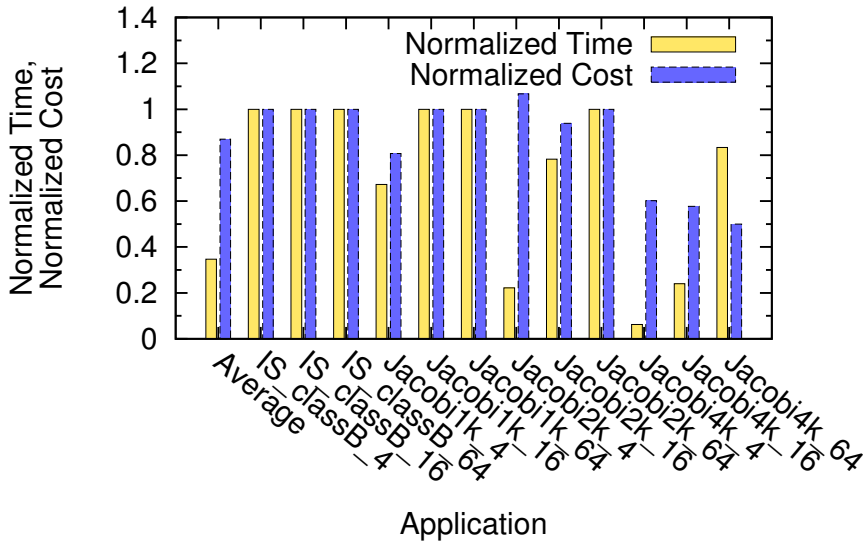
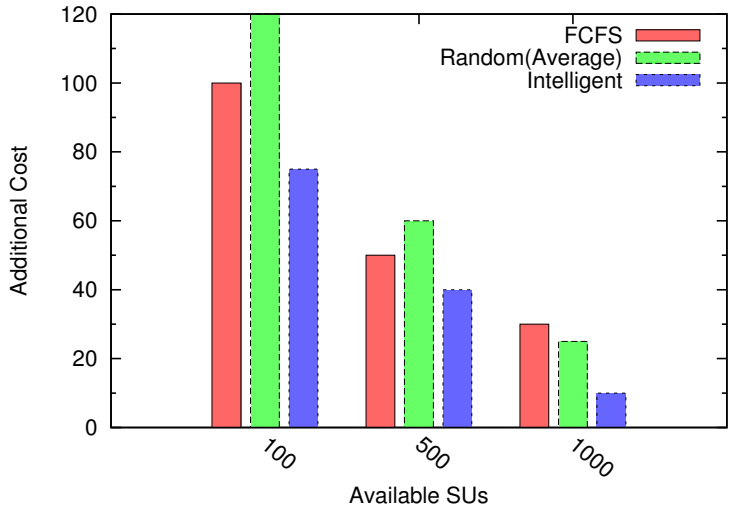


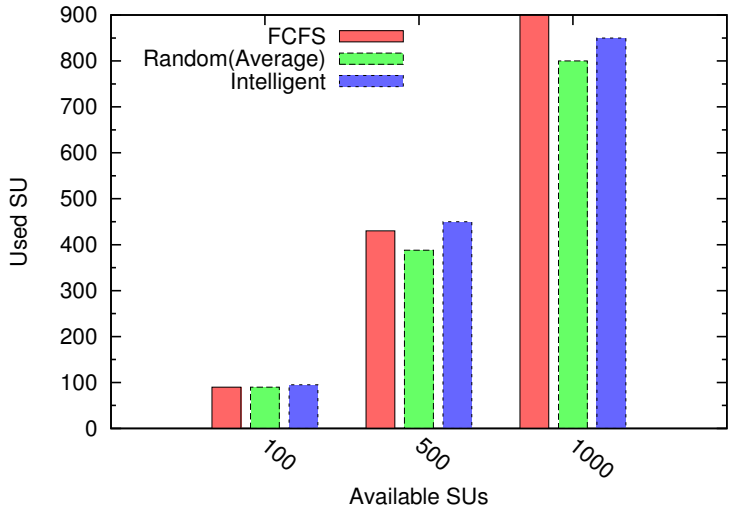

**Figure 3.14: Normalized Performance and Cost vs. execution on supercomputer for FixedCostMapper**

### 3.5.4 Mapping Application Sets

So far we have considered mapping of individual application instances to platforms, while in reality HPC users have often to deal with groups of applications related to a certain project. In this section, we consider a set of applications, each requiring a performance guarantee. The question that we explore here is the following: under a given performance constraint, how do we minimize the overall execution cost of a subset of applications on cloud? The problem can also be formulated as finding which applications should be given the highest execution priority. To solve this problem, we use a greedy iterative algorithm: first find the application that makes best use of supercomputer and map it. In other words, find the application with highest ratio of *(cost on cloud)/(SUs on supercomputer)* when achieving the same performance (same execution time). For example, consider six instances of Jacobi2D - $Jacobi[1k, 2k, 4k]\_[16, 64]$ with 1M iterations of Jacobi1k and 100K iterations of Jacobi2k and Jacobi4k, and a limit of 50 SUs. For this case, intelligent ordering results in




(a) Cost




(b) SUs

**Figure 3.15: Additional Cost and SUs for a group of applications under fixed performance requirements and SU limits**

approximately $18 cost, whereas a simpler allocation scheme may not even find a workable solution, regardless of the cost. For example, a first-come-first-served allocation may exhaust the user SU allocation on supercomputer by cloud-friendly applications, and leave for the cloud applications that do not scale and can never meet the performance target, regardless of the cost. Our group Mapper algorithm takes into account the imperative that applications that poorly scale on cloud are executed on supercomputer first to get maximum relative

benefits from constrained SUs.

Figure 3.15 compares three different strategies for scheduling in such scenarios. FCFS denote first come first serve, where we continue executing application on supercomputer till the SUs are exhausted and thereafter use cloud. Intelligent scheduling denotes the mapping suggested by our mapper. Average denote the average of all possible mapping. It is clear that intelligent scheduling works best.

Such scenarios can be very common since research groups are allocated fixed amount of SUs and hence cloud can be used as addition to supercomputer. Application execution in times of limited platform availability, e.g. in case of deadlines, can be handled similarly.

## 3.6 Related Work

Parallel application characterization and application signature for performance prediction have been extensively researched; with focus on run-time instrumentation, event-tracing and curve-fitting based performance-modeling approaches [64–68]. However our objective is to characterize applications in those dimensions which are the most important for the purpose of mapping of applications to platforms.

There have been are several efforts on job scheduling for HPC applications. Platform Computing offers LSF (Load Sharing Facility) - a commercial job scheduler for HPC applications which allows load sharing using distribution of jobs to available CPUs in heterogeneous network. Adaptive Computing MOAB, IBM Tivoli Workload Scheduler, SLURM, Globus Resource Allocation Manager (or GRAM), Oracle Grid Engine, Application Level Placement Scheduler (ALPS), PBS Professional and TORQUE are other similar systems. However, these efforts do not consider intrinsic HPC application characteristics to allow more intelligent decision making. Moreover, they lack policies which consider resources with different interconnection network and which could decide between virtualized and non-virtualized resources. They only focus on user preferences and do not take into account a combination of application signature and platform characteristics. This raises a possibility of overloading one platform while others remain under-loaded.

Kim et al. [69] discuss autonomic management of scientific workflow applications on hybrid infrastructures such as those comprising HPC grids and on-demand pay per use clouds. They present three usage models for hybrid HPC grid and cloud computing - acceleration, conservation and resilience. However, they use cloud for sequential simulation and do not consider execution of parallel applications on cloud. Hybrid cloud deployments have also been explored for enterprise applications [70]. GrADS [71] project addressed the problem

of scheduling, monitoring and adapting applications to heterogeneous and dynamic grid environment using an application signature based approach similar to ours. Another similar project in the context of grid is the Faucets [72] project where market-driven strategies were used. Our focus is on clouds which are widely adopted by industry and academia and hence we will address additional challenges such as virtualization, cost and pricing models for HPC in cloud.

## 3.7 Conclusions and Future Work

We have shown that by performing multi-platform dynamic application-aware scheduling, a hybrid cloud-supercomputer platform environment can actually outperform its individual constituents. By using an underutilized resource which is "good enough" to get the job done sooner, it is possible to get better turnaround time for job (user perspective) and improved throughput (provider perspective). Another potential model for HPC in cloud is to use cloud only when there is high demand (cloud *burst*). Our evaluation showed that application-agnostic cloud bursting (e.g. BestFirst heuristic) is unrewarding, but application-aware bursting is a promising research direction. More work is needed to consider other factors in multi-platform scheduling: job Quality of Service (QoS) contracts, deadlines, priorities, and security. Also, future research is required in cloud pricing in multi-platform environments. Market mechanisms and equilibrium factors in game theory can help automate such decisions.

We also considered cloud user perspective and showed that there is significant cost-saving potential in using hybrid platform environments and intelligent mapping of applications to available platforms. Finally, we described how we could automate the mapping using a combination of application characteristics, platform parameters, and user preferences. In the future, we plan to enhance the techniques to generate better "application signatures" and smarter mapping.

CHAPTER 4

# HPC-Aware Cloud Scheduler and VM Consolidation

The performance evaluation in Chapter 2 showed that only small scale HPC applications or applications with less intensive communication patterns 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 [13, 14][1].

Past research [13, 14, 20, 57] on HPC in cloud has primarily focused on evaluation of scientific parallel applications (such as those using MPI [37]) 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 [17–19].

In this chapter, 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

[1]Some portions reprinted with permission, from [60], ©2013 IEEE and [73], ©2011 ACM

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 chapter 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 chapter 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. (§4.1, §4.2)

- 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. (§4.2, §6.5)

- 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%. (§6.2.3, §6.5)

- We modify CloudSim [26] to make it suitable for simulation of HPC in cloud. To

**Table 4.1: Amazon EC2 instance types and pricing**

| Resource | Instance type | | |
|---|---|---|---|
| | High-Memory Extra Large | Double | 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 |

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. (§4.6)

## 4.1 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 [9]. Amazon EC2 offers a dedicated pool of resources for HPC applications known as *Cluster Compute*. We consider two instance types shown in Table 4.1 and, as a concrete example, Table 4.2 shows the distribution of actually executed jobs calculated from METACENTRUM-02.swf logs obtained from the Parallel Workload Archive [63]. 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 [74]. 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 from Table 4.1 pricing, for every 2.4$, one can get additional 0.9$. However, the

**Table 4.2: 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%) |

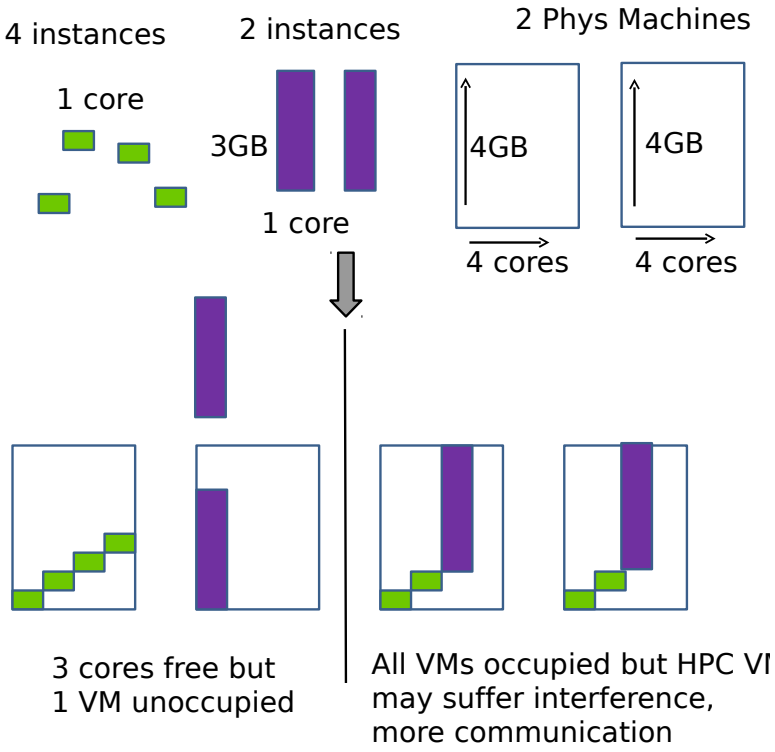



**Figure 4.1: Tradeoff: Resource packing vs. HPC-awareness**

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 $\frac{[2.4-2.4/(88/13)+0.9]-2.4}{2.4} = 23\%$.

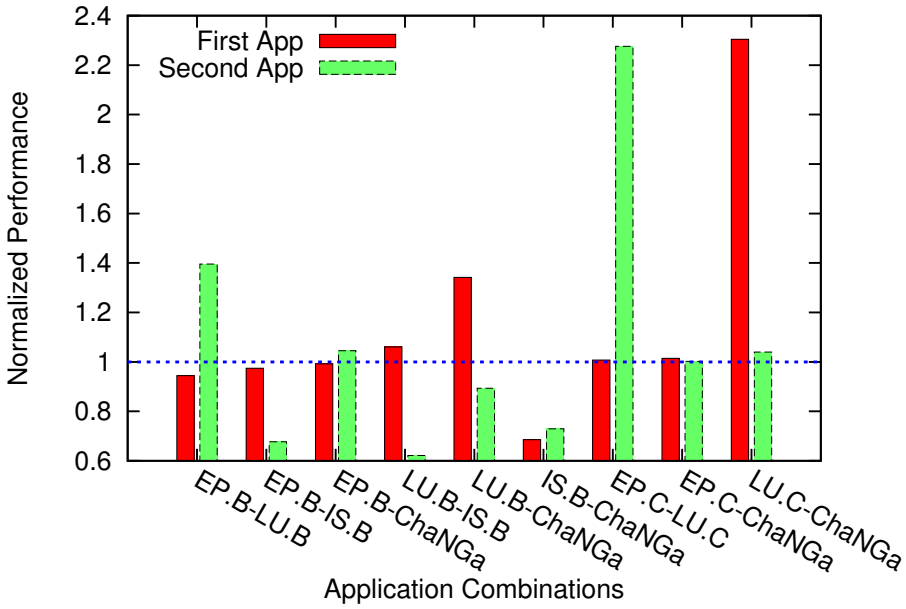
However, traditionally, HPC applications are executed on dedicated nodes to prevent any interference arising from co-located applications. This is because the performance of many HPC applications strongly depends on the slowest node (for example, when they synchronize through MPI barriers). Figure 4.1 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, 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.

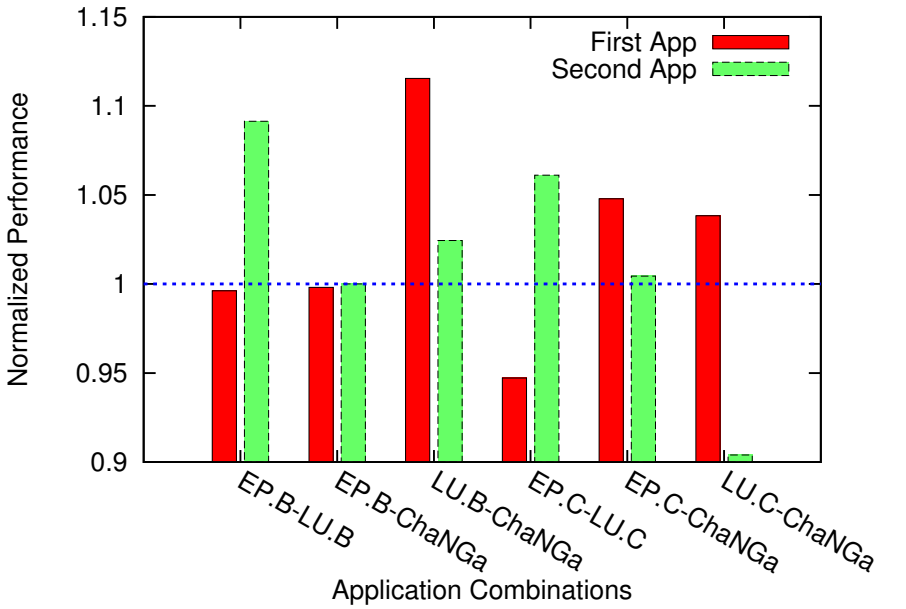
### 4.1.1 Cross-Application Interference

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 4.2 demonstrates the effect of running two different applications while sharing a multi-core node (4-core, 8GB, 3 GHz Open Cirrus [31] node). Each VM needs 1-vcpu, 2GB memory, and uses KVM-hypervisor and CPU-pinned configuration. Applications used here are NPB [39] (EP = Embarrasingly Parallel, LU = LU factorization, IS = Integer Sort) problem size class B and C and ChaNGa [41] = Cosmology. More details of testbed and applications are discussed in Section 4.4.

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 4.2a (4 VMs each application) and 4.2b (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

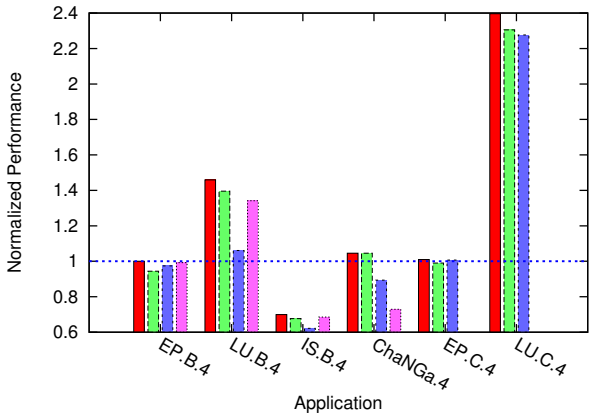



(a) Total cores (and VMs) per application = 4

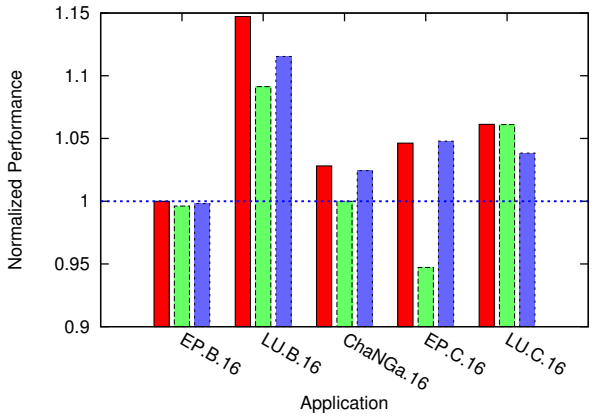



(b) Total cores (and VMs) per application = 16

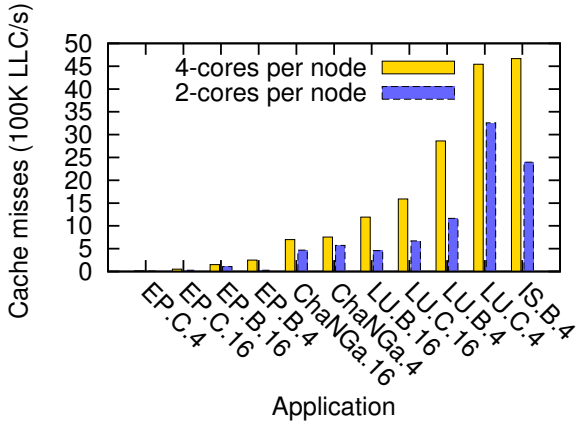
**Figure 4.2: Application Performance in shared node execution (2 cores for each application on a node) normalized wrt to dedicated execution (using all 4 cores of a node for same application). Total cores (and VMs) per application = 16, physical cores per node = 4**




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




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




(c) Last Level Cache Misses

**Figure 4.3: (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 4.2). (c) Average per core last level cache misses: Using only 2 cores vs. using all 4 cores of a node**

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 4.2).

### 4.1.2 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 $10\text{Gbps}/32 = 0.312$ 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 4.1.

### 4.1.3 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 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.

## 4.2 Methodology

Having identified the opportunities for HPC-aware VM consolidation in cloud, we discuss our methodology for addressing the challenges discussed in Section 4.1. 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.

### 4.2.1 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 4.2, 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 4.3a and 4.3b 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 4.3c 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 4.3a and 4.3c, 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 4.2.

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.

### 4.2.2 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 aware-

ness), Multi-dimensional Online Bin packing, and Interference minimization through cache-sensitivity awareness. We discuss the details of this scheduler in the next section.

## 4.3 An HPC-Aware Scheduler

Next, we discuss the design and implementation of the proposed techniques on top of OpenStack Nova scheduler [25].

### 4.3.1 Background: OpenStack Nova Scheduler

OpenStack [25] 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 nodes or *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).

- 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 [17, 18].

### 4.3.2 Design and Implementation

Algorithm 2 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 4.1. 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. [75]. 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 2 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.

Figure 4.4 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`.

## 4.4 Evaluation Methodology

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

**Algorithm 2** 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: $hostCapacity, rackCapacity, filteredHostList \leftarrow$ $CalculateHostAndRackCapacity(request\_spec, capabilities)$
7: **if** $(request\_spec.class == ExtremeHPC)\ ||\ (request\_spec.class == SyncHPC)$ **then**
8: $sortedHostList \leftarrow$ sort $filteredHostList$ by decreasing order of $hostCapacity[j]$ where $j \in filteredHostList$.
9: $PrelimBuildPlan \leftarrow$ stable Sort $sortedHostList$ by decreasing order of $rackCapacity[capability[j].rackid]$ where $j \in filteredHostList$.
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: $buildPlan \leftarrow MDOBP(request\_spec, Prebuildplan, capabilities)$
22: **end if**
23: return $buildPlan$

---

24: **procedure MDOBP**(**request_spec**, **Prebuildplan**, **capabil- ities**)

---

25: $buildPlan = new\ vector[int]$
26: **for** $i = 1$ to $i < request\_spec.numInstances$ **do**
27: **repeat**
28: $node \leftarrow chooseBestFitHost(request\_spec, Prebuildplan, capabilities)$ // 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: $\alpha$ = Total cache threshold for any application
36: $\beta$ = Total cache threshold for SyncHPC, in general $\alpha >> \beta$
37: $totalCacheScore = \sum_i i.cacheScore \forall i$ such that i is an instance currently running on node
38: **if** $(totalCacheScore + request\_spec.cacheScore) > \alpha$ **then**
39: return false
40: **end if**
41: **if** $i.class = SyncHPC$ for any i – an instance currently running on node **then**
42: **if** $(totalCacheScore + request\_spec.cacheScore) > \beta$ **then**
43: return false
44: **end if**
45: **end if**
46: return true

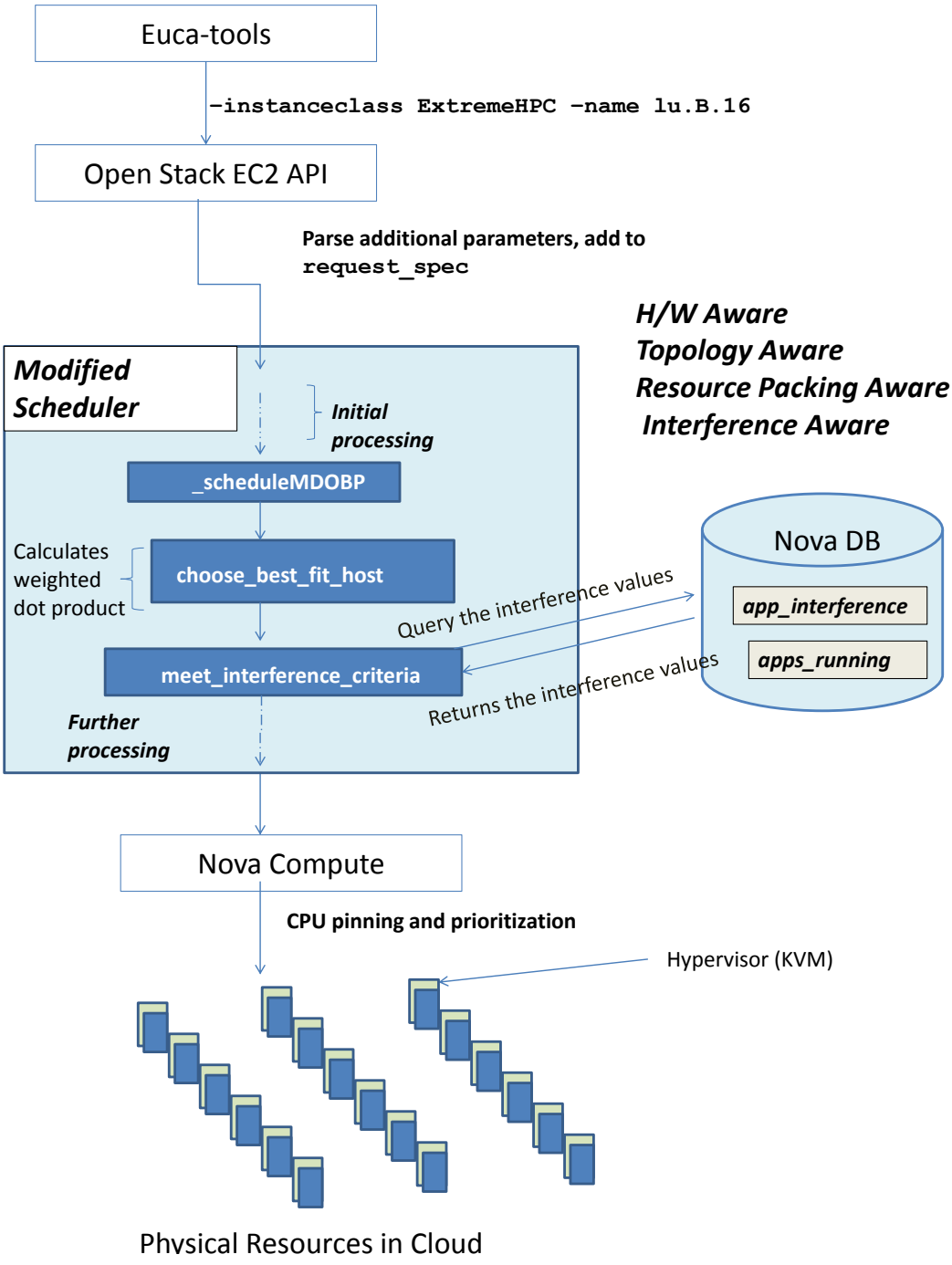



**Figure 4.4: Implementation details and control flow of a provisioning request**

### 4.4.1 Experimental Testbed

We evaluated our techniques on a cloud setup using OpenStack on Open Cirrus testbed at HP Labs site [31]. 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 4.1.2.

For virtualization, we chose KVM [33], since prior research has indicated that KVM is a good choice for virtualization for HPC clouds [34]. 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 [73]). 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.

### 4.4.2 Benchmarks and Applications

We used the NAS Parallel Benchmarks (NPB) [39] 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 [40] – A highly scalable molecular dynamics application used ubiquitously on supercomputers. We used the ApoA1 input (92k atoms) for our experiments.
- ChaNGa [41] – 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++ [76] which is an object-oriented parallel programming language. We used the `net-linux-x86-64` machine layer of Charm++ with –O3 optimization level.

## 4.5 Experimental Results

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

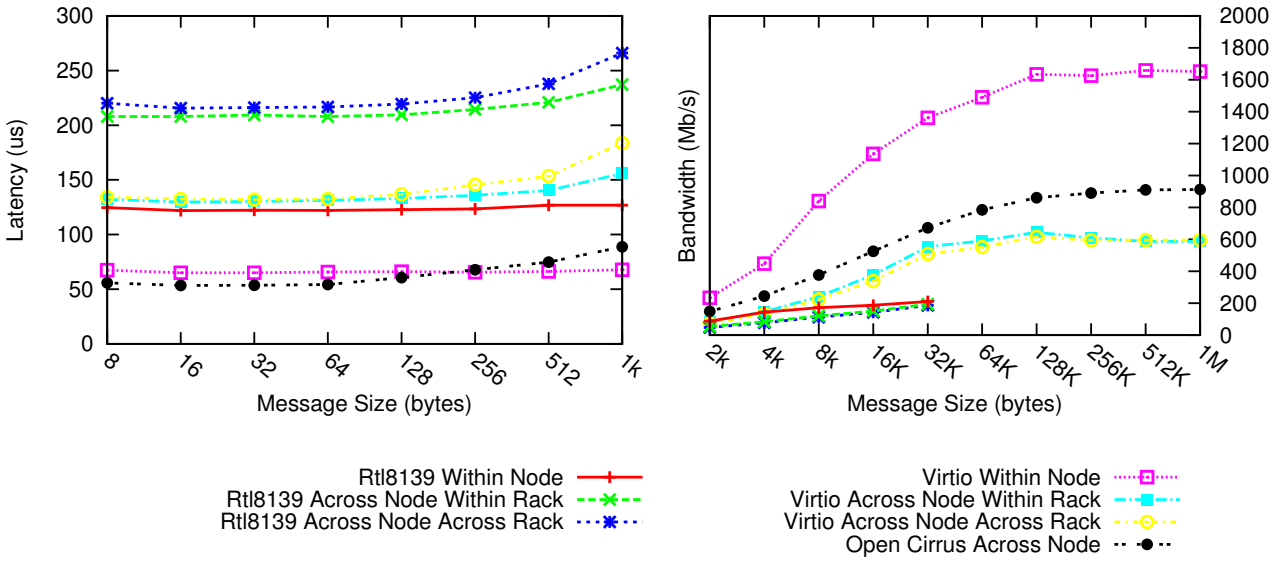



**Figure 4.5: Latency and Bandwidth vs. Message Size for different VM placement.**

### 4.5.1 HPC-Aware Placement

We first evaluate the effect of topology-aware scheduling. Figure 4.5 shows the results of a ping-pong benchmark. We used a Converse [77] (underlying substrate of Charm++) ping-pong benchmark to compare latencies and bandwidth for various VM placement configurations. Figure 4.5 presents several insights. First, we see that *virtio* outperforms *rtl8139* network driver both for intra-node and inter-node VM communication, making it a natural choice for remainder of the experiments. Second, there is significant virtualization overhead. Even for communication between VMs on same node, there is a 64 usec latency using virtio. Similarly, for inter-node communication, VM latencies are around twice compared to communication between physical nodes in Open Cirrus and there is also substantial reduction in achieved bandwidth, although the degradation in bandwidth (33% reduction) is less compared to the degradation in latencies (100% increase). Third, there is very little difference for latencies and bandwidth when comparing communication between VMs on different nodes but same rack and between VMs on different nodes on different racks.

This can be attributed to the use of wormhole routing in modern network which means that the extra hops cause very little performance overhead. As we discussed in section 4.1.2, effects of intra-rack and cross-rack communication become more prominent as we scale up or the application performs significant collective communication such as all-to-all data movement.

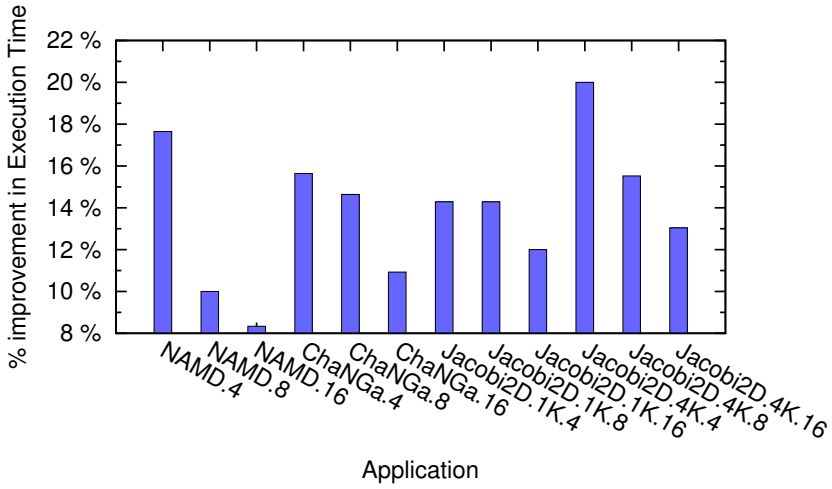
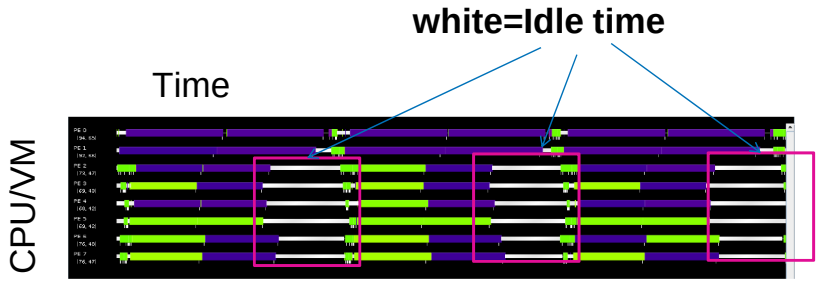
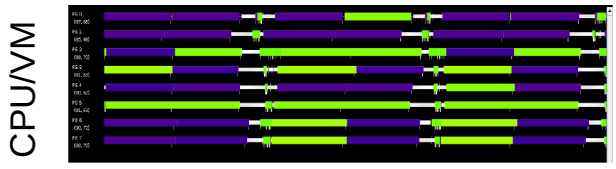



**Figure 4.6: % improvement achieved using HPC awareness (homogeneity) compared to the case where 2 VMs were on slower processors and rest on faster processors**



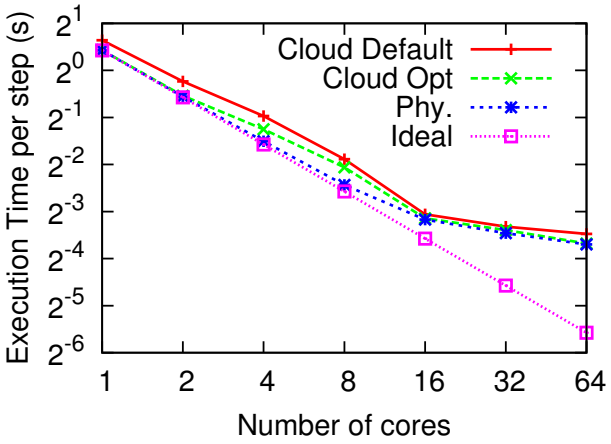

(a) Heterogeneous: First two VMs on slower processors



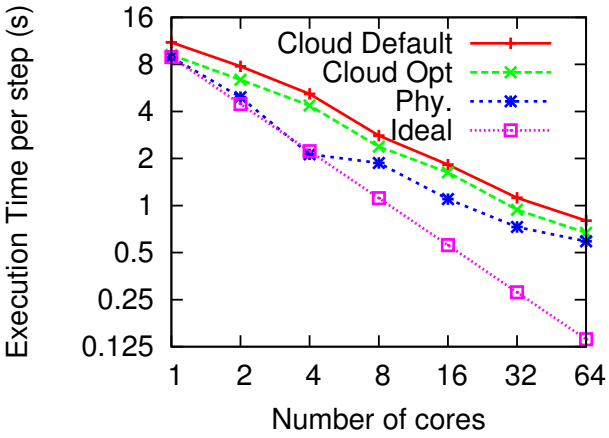

(b) Homogeneous: All 8 VMs on same type of processors

**Figure 4.7: CPU Timelines of 8 VMs running Jacobi2D**

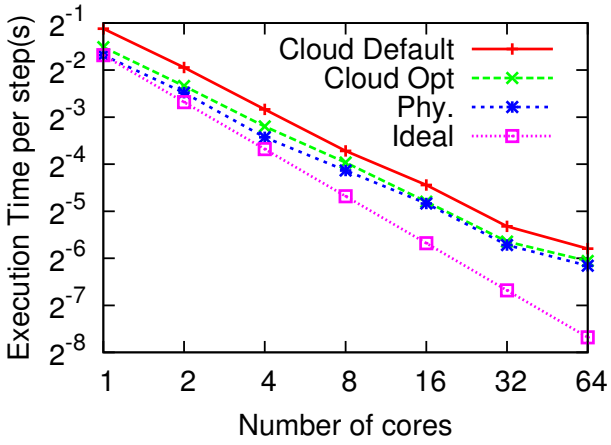
To demonstrate the impact of topology awareness and homogeneity, we compared the performance obtained by HPC-aware scheduler with random VM placement. In these experiments, we did not perform VM consolidation. Figure 4.6 shows the performance obtained by our VM placement (Homo) compared to the case when 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




(a) NAMD (Molecular Dynamics)




(b) ChaNGa (Cosmology)




(c) Jacobi2D – 4K by 4K matrix

**Figure 4.8: Runtime Results: Execution Time vs. Number of cores / VMs for different applications.**

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 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 [45] tool. Figure 4.7 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 4.7a, there is a lot more idle time on VMs 3-7 compared to first

| 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 |

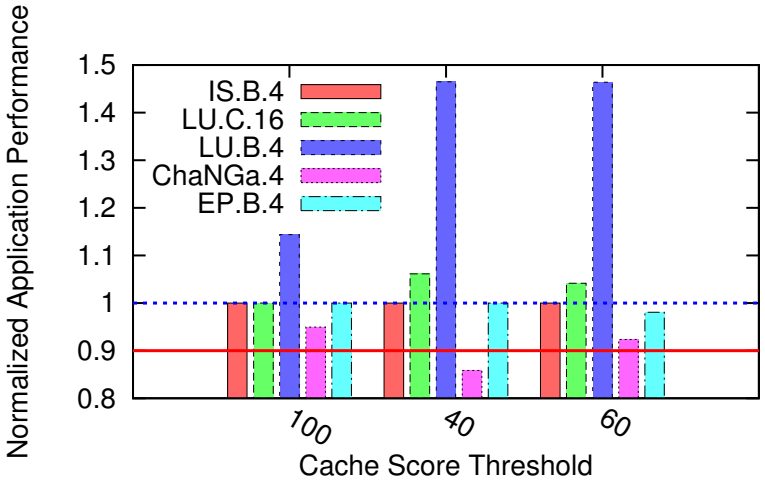



**Figure 4.9: Table of applications, and figure showing percentage improvement achieved using application-aware scheduling compared to the case when applications were run in dedicated manner**

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 4.7b 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 4.8). 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 4.8 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, 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 [73].

### 4.5.2 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 4.9 using the application-aware scheduler. The application suffix is the number of requested VMs. Figure 4.9 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 4.9, 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.

## 4.6 Simulation

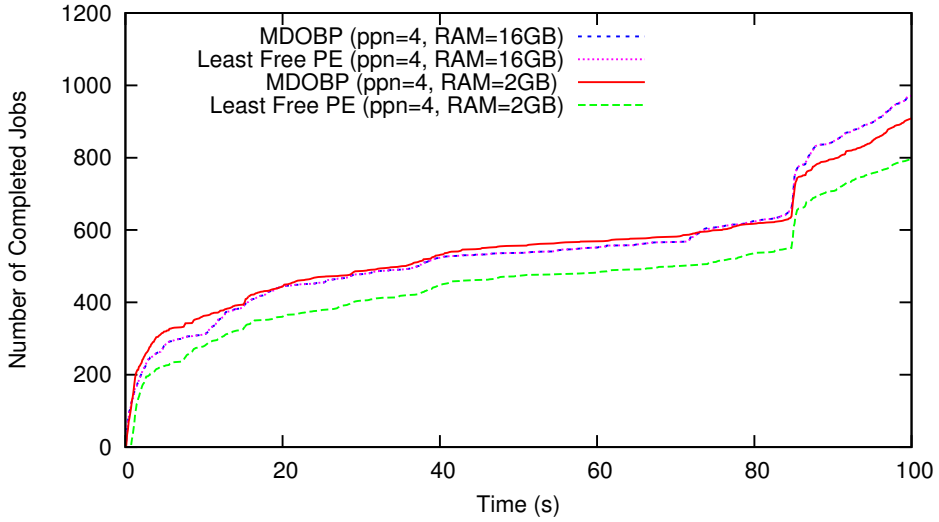
CloudSim is a simulation tool modeling a cloud computing environment in a datacenter, and is widely used for evaluation of resource provisioning algorithms [26]. In this work, we modified it to enable the simulation of High Performance Computing jobs in cloud. HPC machines have massive number of processors, 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 2).

At the start of simulation, a fixed number of VMs (of different specified types) are created,

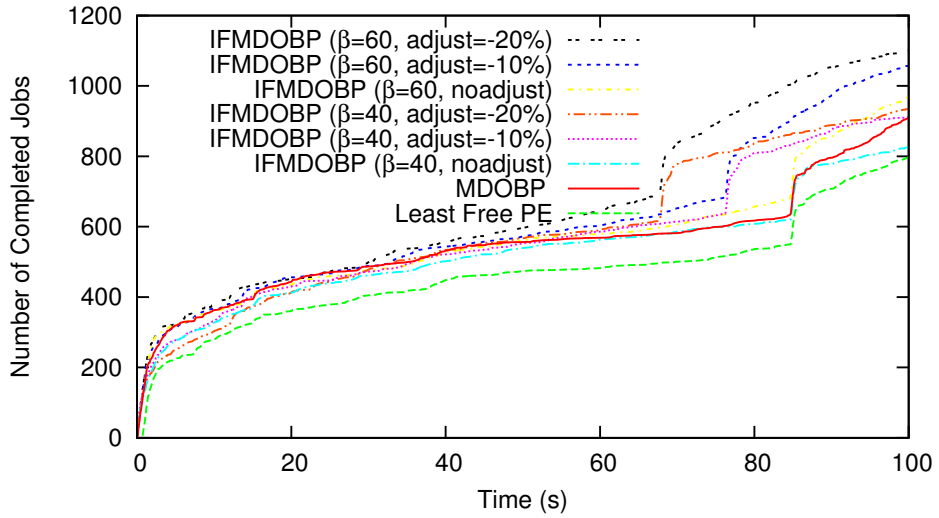
and jobs (cloudlets) are submitted to the data center broker which maps a job to a VM. When there are no pending jobs, all the VMs are terminated and simulation completes. Since our focus was on mapping of VMs to physical hosts, we created a one-to-one mapping between cloudlets and VMs. Moreover, we 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 [63]. We used the METACENTRUM-02.swf logs since these logs contain information 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 4.10a 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 4.10b 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 4.5. 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 4.5.2. 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.




(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)

**Figure 4.10: Simulation Results: Number of completed jobs vs time for different scheduling techniques, 1024 cores**

## 4.7 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 [13, 14, 20, 57]. However,

clouds can be cost-effective for some applications, specifically those with less communication and at low scale [14, 57]. 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 re mapping 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 [78, 79], 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 [25], Eucalyptus [32], and OpenNebula [80] 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 [81]. A detailed description and validation of VM consolidation heuristics is provided in [75]. 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 [19]. Recently, OpenStack community has been working on making the scheduler architecture-aware and suitable for HPC [17, 18]. 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 [15]. It is not known how strictly those guarantees are met and what techniques are used to meet them.

In this work, we extend previous research on cloud schedulers for HPC 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 [78, 82, 83].

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.

## 4.8 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%.

A future direction is 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 future direction is to research how to schedule a mix of HPC and non-HPC applications in an intelligent fashion to increase resource utilization.

CHAPTER 5

# Cloud-Aware HPC Load Balancer

In this chapter, we take the viewpoint of an HPC user and explore how she can optimize the execution of her application when given a resource allocation in cloud, with no control over VM placement. The insufficient network performance is a major bottleneck for HPC in cloud, and has been widely explored [11, 13, 14, 20]. Two less explored challenges are *resource heterogeneity* and *multi-tenancy* – which are fundamental artifacts of running in cloud. Clouds evolve over time, leading to heterogeneous configurations in processors, memory, and network. Similarly, multi-tenancy is also intrinsic of cloud, enhancing the business value of providing a cloud. Multi-tenancy leads to multiple sources of interference due to sharing of CPU, cache, memory access, and interconnect. For tightly-coupled HPC applications, heterogeneity and multi-tenancy can result in severe performance degradation and unpredictable performance, since one slow processor slows down the entire application. As an example, on 100 processors, if one processor is 30% slower compared to the rest, application will slowdown by 30% even though the system has 99.7% raw CPU power compared to the case when all processors are fast[1].

One approach to address the above problem is *making clouds HPC-aware*. This approach was discussed in Chapter 4. Examples are HPC-optimized clouds (such as Amazon Cluster Compute [15] and DoE Magellan project [13]) and HPC-aware cloud schedulers [17, 60]. In this chapter, we explore the other approach – *making HPC cloud-aware*, which is relatively less explored [19, 84].

Our primary hypothesis is that the challenges of heterogeneity and noise arising from multi-tenancy can be handled by an adaptive parallel runtime system. To validate our hypothesis, we explore the adaptation of Charm++ [27, 28] runtime system to virtualized environment. We present techniques for virtualization-aware load balancing to help appli-

[1]Some portions reprinted with permission from [61], ©2013 IEEE

cation users gain confidence in the capabilities of cloud for HPC. MPI [37] applications can also benefits from our approach using Adaptive MPI (AMPI) [28]. Also, our fundamental approach is applicable to other programming models which support migratable work/data units.

Efficient load balancing in a cloud is challenging since running in VMs makes it difficult to determine if (and how much of) the load imbalance is application-intrinsic or caused by extraneous factors. Extraneous factors include heterogeneous resources, other users' VMs competing for shared resources, and interference by virtualization emulator process (§ 5.1).

The primary contributions of this work are the following:

- We propose dynamic load balancing for efficient execution of tightly-coupled iterative HPC applications in heterogeneous and dynamic cloud environment. The main idea is periodic refinement of task distribution using measured CPU loads, task loads, and idle times (§ 5.3).
- We implement these techniques in Charm++ and evaluate their performance and scalability on a real cloud setup on Open Cirrus testbed [31]. We achieve 45% reduction in execution time compared to no load balancing (§ 5.5).
- We analyze the impact of load balancing frequency, grain size, and problem size on achieved performance (§ 5.5).

## 5.1 Need for Load Balancer for HPC in Cloud

In the context of cloud, the execution environment depends on VM to physical machine mapping, which makes it (a) dynamic and (b) inconsistent across multiple runs. Hence, a static allocation of compute tasks to parallel processes would be inefficient. Most existing dynamic load balancing techniques operate based exclusively on the imbalance internal to the application, whereas in cloud, the imbalance might be due to the effect of extraneous factors. These factor originate from two characteristics, which are intrinsic to cloud:

1. *Heterogeneity*: Cloud economics is based on the creation of a cluster from existing pool of resources and incremental addition of new resources. While doing this, homogeneity is lost.
2. *Multi-tenancy*: Cloud providers run a profitable business by improving utilization of underutilized resources. This is achieved at cluster-level by serving large number of

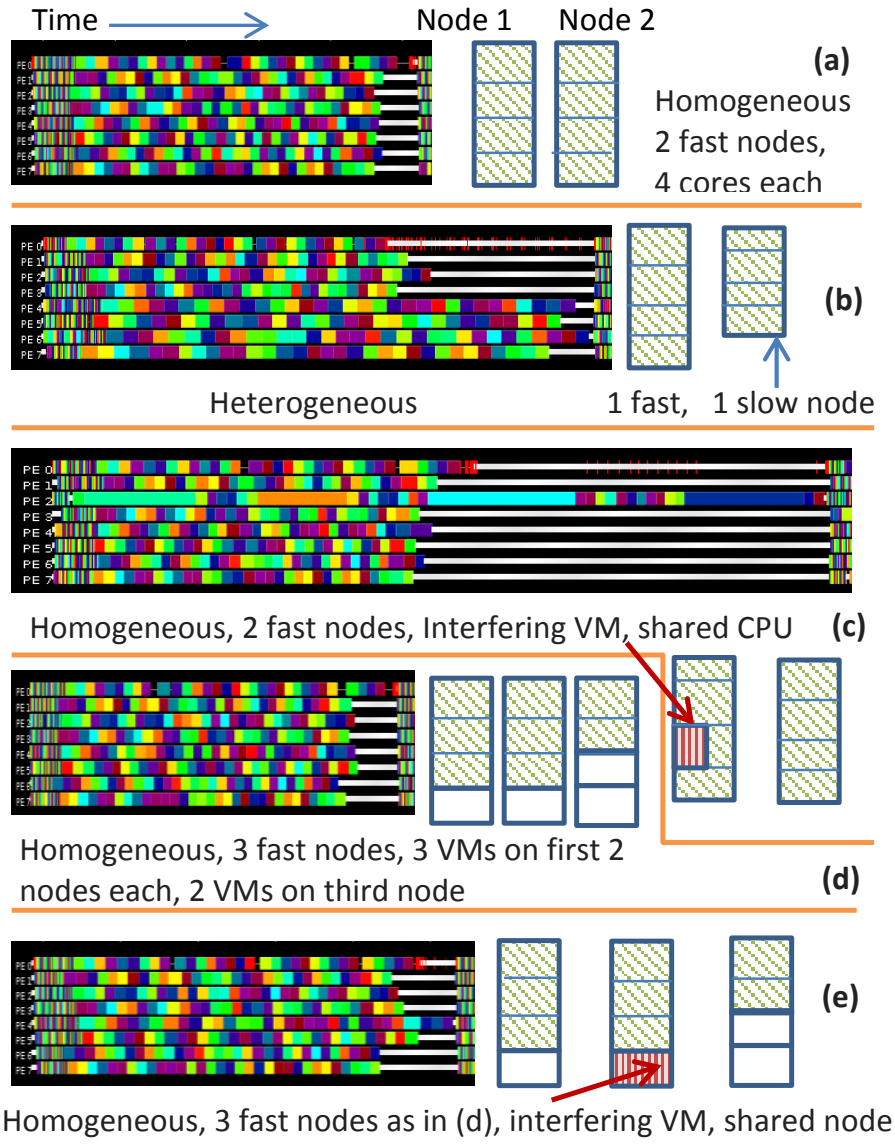



**Figure 5.1: Experimental setup (on right) and timeline of 8 VMs showing one iteration of Stencil2D: white portion = idle time, colored portions = application functions.**

users, and at server-level by consolidating VMs of complementary nature (such as memory- and compute-intensive) on same server. Hence, multi-tenancy can be at resource-level (memory, CPU), node-level, rack-level, zone-level, or data center level.

In such environment, application performance can severely degrade, especially for tightly-coupled applications where the application progress is governed by the slowest processor. To demonstrate its severity, we conducted a simple experiment where we ran a tightly-coupled 5-point stencil benchmark, referred to as *Stencil2D* (2K $\times$ 2K matrix), on 8-VMs, each with

a single virtual core (VCPU), pinned to a different physical core of 4-core nodes. More details of our experimental setup and benchmarks are given in Section 5.4. To study the impact of heterogeneity, we ran this benchmark in 2 cases - first, we used two physical nodes of same fast (3.00 GHz) processor type (Figure 5.1a) and second, we used two physical nodes with different processor types – fast (3.00 GHz) and slow (2.13 GHz) (Figure 5.1b). We used Projections [45] performance analysis tool. Figure 5.1 shows one iteration of these runs. Each horizontal line represents the timeline for a VM (or VCPU). Different colors (shades) represent time spent executing application tasks whereas white represents idle time. The length of timelines represent the iteration execution time, after which the next iteration can begin. In Figure 5.1a, the small idle time on VM#1-7 is present because the first process performs a small co-ordination work. In Figure 5.1b, there is a lot more idle time (hence wastage of CPU cycles) on first four VMs compared to the next four, since VMs#4-7 running on slower processors take longer to finish same amount of work.

Similar effect is also observed when there is an interfering VM. Figure 5.1c shows the case when we ran all VMs on fast processors but there is an interfering VM which shares the physical core with one of the VMs from our parallel job (VM#3). The interfering VM runs sequential NPB-FT (NAS Parallel Benchmark – Fourier Transform) Class A [39]. In this case, the Projections timelines tool includes the time spent executing the interfering task in the time spent for executing tasks of the parallel job on that processor because it can not identify when the operating system switches context. This gets reflected in the fact that some of the tasks hosted on VM#3 take significantly longer time to execute than others (longer bars in Figure 5.1c). Due to this CPU sharing, it takes longer for the parallel job to finish the same tasks. Moreover, the tightly-coupled nature of the application means that no other process can start the next iteration unless all processes have finished the current iteration (idle times on rest of the VMs).

If the VMs do not share physical core but share the multi-core physical node, the contention for limited shared cache capacity and memory controller subsystem can manifest itself as another source of interference (Figure 5.1e). Here. we ran the 8 VMs on 3 fast nodes, with first three VMs on one node, next three VMs on second node, and last two VMs on third node. On second node, we placed another VM mapped to the unused core and ran NPB-LU Class B benchmark on it. The unused cores on first and third nodes are left idle. Figure 5.1e shows that VM#5 is taking longer time than the rest compared to the case with exactly same configuration but no interfering VM (Figure 5.1d). It can also be noted that the time in Figure 5.1d is slightly better than Figure 5.1a. This can be attributed to the fact that the shared resources in the 4-core node are shared between 4 processes in Figure 5.1a, but by only 3 in Figure 5.1d.

The distribution of such interference is fairly random and unpredictable in a cloud. Hence, we need a mechanism to adapt to the dynamic variation in the execution environment.

## 5.2 Background: Charm++ and Load Balancing

Charm++ [27, 28] is a message-driven object-oriented parallel programming system, which is used by large-scale scientific applications such as NAMD [40]. In Charm++, the programmer needs to decompose (or over-decompose) the application into large number of medium grained pieces, referred to as Charm++ objects or *chares*. By over-decomposition, we mean that the number of objects or work/data units is greater than the number of processors. Each object consists of a state and a set of functions, including local and *entry methods*, which execute only when invoked from a local or remote processor through messages. The runtime system maps these objects onto available processors and they can be migrated across processors during execution. This message-driven execution and over-decomposition results in automatic overlap of computation and communication, and helps in hiding the network latency.

MPI [37] applications can leverage the capabilities of Charm++ runtime using the adaptive implementation of MPI (AMPI [28]), where MPI processes are implemented as user-level threads by the runtime.

The over-decomposition of application into migratable objects (or threads) facilitates dynamic load balancing, a concept central to our work. The runtime system instruments the application execution, and measures various statistics, such as computation time spent in each object, process time, and idle time. Using this measured data, the load balancer periodically re-maps objects to processors using a load balancing strategy. There is an inherent assumption that future loads will be almost same as the measured loads (*principle of persistence*) – which is true for most iterative applications.

## 5.3 Cloud-Aware Load Balancer for HPC

In a cloud, the application user has access only to virtualized environment which hides the underlying platform heterogeneity. Hence, for heterogeneity-awareness, we estimate the CPU capabilities for each VCPU, and use those estimates to drive the load balancing. An accurate performance prediction will depend on the application characteristics, such as FLOPS, number of memory accesses, and I/O demands. In this thesis, we demonstrate

```
for(i=0; i < iter_block; i++) {
    double b=0.1 + 0.1 * *result;
    *result=(int)(sqrt(1+cos(b * 1.57)));
  }
```

**Figure 5.2: Computation loop: Estimating relative CPU speeds**

the merits of heterogeneity-awareness using a simple estimation strategy, which works well in conjunction with periodic refinement of load distribution. We use a simple compute-intensive loop (Figure 5.2) to measure relative CPU frequencies, which are then used by the load balancing framework. Also, we assume that VMs do not migrate during runtime, and VCPUs are pinned to physical CPUs. We believe that these assumptions are valid for HPC in cloud since live migration leads to further noise and migration costs, and pinning VCPUs to physical CPUs results in better performance.

Other then the static heterogeneity, we need to address interfering tasks of other VMs, which can start and finish randomly. Hence, we propose a dynamic load balancing scheme which continuously monitors the loads for each VCPU and reacts to any imbalance. Our scheme uses task migration which enables the runtime to keep equal loads on all VCPUs. It is based on instrumenting the time spent on each task, and predicts future load based on the execution time of recently completed iterations. However, to incorporate the impact of interference, we need to instrument the load external to the application under consideration, referred to as the *background load*. For maintaining balanced loads, we need to ensure that all VCPUs have load close to the average load ($Tk_{avg}$) defined as:

$$Tk_{avg} = \frac{\sum_{p=1}^{P}((\sum_{i=1}^{N_p} t_i + O_p) * f_p)}{P} \tag{5.1}$$

where $P$ is the total number of VCPUs, $N_p$ is the number of tasks assigned to VCPU $p$, $t_i$ is the CPU time consumed by task $i$ running on VCPU $p$, $f_p$ is the frequency for VCPU $p$ estimated using the loop in Figure 5.2, and $O_p$ is the total background load for VCPU $p$. Notice that in Equation 5.1, we normalize the execution times to number of *ticks* by multiplying the execution times for each task and the overhead to the estimated VCPU frequency. The conversion from CPU time to ticks is performed to get a processor-independent measure of task loads, because the same task can take different time when executing on a different CPU. The task CPU time ($t_i$) are measured using CPU timers from

inside the VCPU, and recorded in Charm++ load balancing database. $O_p$ is given by:

$$O_p = T_{lb} - \sum_{i=1}^{N_p} t_i - t^p_{idle} \tag{5.2}$$

where $T_{lb}$ is the wall clock time between two load balancing steps, $t_i$ is the CPU time consumed by task $i$ on VCPU $p$ and $t^p_{idle}$ is the idle time for VCPU $p$ since the previous load balancing step. We extract $t^p_{idle}$ from the VM's `/proc/stat` file. Our objective is to keep the load for each VCPU close to the average load while considering the background load ($O_p$) and heterogeneity. Hence, we formulate the problem as:

$$\forall p \in P, \sum_{i=1}^{N_p} (t_i * f_{m_i^{k-1}}) + O_p * f_p - Tk_{avg} < \epsilon \tag{5.3}$$

where $P$ is the set of all VCPUs, $t_i$ is the CPU time consumed by task $i$, $O_p$ is the total background time for VCPU $p$, $f_p$ is the estimated frequency of VCPU $p$, $f_{m_i^{k-1}}$ is the frequency of VCPU where task $i$ ran in previous, that is $(k-1)^{th}$ iteration, and $\epsilon$ is the permissible deviation from the average load.

Algorithm 3 summarizes our approach with the definition of each variable given in Table 5.1. The main idea is to do periodic checks on the state of load balance and migrate objects from overloaded VCPUs to under-loaded VCPUs such that Equation 5.3 is satisfied (see Figure 5.3). Our approach starts with categorizing each VCPU as overloaded/underloaded (lines 2-7). To categorize a VCPU, our load balancer compares the sum of *ticks* assigned to a VCPU (including the background load) to the average number of *ticks* for the entire application i.e. $Tk_{avg}$ (lines 16-26). If current VCPU load is greater than $Tk_{avg}$ by a value greater than $\epsilon$, we mark that VCPU as overloaded and add it to the $overHeap$ (line 4). Similarly, if the VCPU load (assigned *ticks*) is less than $Tk_{avg}$ by a value greater than $\epsilon$, we categorize it as underloaded and add it to the $underSet$ (line 6). Due to space constraints, we omit the algorithm for method *isLight*. It is the same as *isHeavy* other then the change in condition at line 21 mentioned earlier. Once we have built the underloaded set and overloaded heap of VCPUs, we have to transfer tasks from the overloaded VCPUs i.e. $overHeap$, to the underloaded VCPUs i.e. $underSet$, such that there are no VCPUs left in the $overHeap$ (lines 10-15).

To decide the new task mapping for balanced load, our scheme removes the most overloaded VCPU from $overHeap$ i.e. *donor* (line 11), and the procedure $getBestCoreAndTask$ selects the $bestTask$, which is the largest task currently placed on *donor* such that it can be transferred to a core from $underSet$ without overloading it (line 12). $getBestCoreAndTask$

**Table 5.1: Description for variables used in Algorithm 3**

| Variable | Description |
|---|---|
| $\mathbf{P}$ | number of VCPUs |
| $\mathbf{Tk_{avg}}$ | average *ticks* per VCPUs |
| $\mathbf{t_i}$ | CPU time of task $i$ |
| $\mathbf{m_i^k}$ | VCPU number to which task $i$ is assigned during step $k$ |
| **overHeap** | heap of overloaded VCPUs |
| $\mathbf{O_p}$ | background load for VCPUs $p$ |
| $\mathbf{f_p}$ | estimated frequency of VCPU $p$ |
| **underSet** | set of underloaded VCPUs |

also selects the *bestCore*, which is a VCPU from *underSet*, which will remain underloaded after being assigned the *bestTask*. After the *bestTask* and *bestCore* are determined, we update the mapping of the task (line 13), the loads of both the *donor* and *bestCore*, and the *overHeap* and *underSet* with these new load values (line 14). This process is repeated till the *overHeap* gets empty i.e. no overloaded VCPUs are left.

We note that different VM technologies can expose different time semantics to the guest – virtual vs. real. Hence, the CPU times of tasks ($t_i$) (and hence $O_p$) can be inaccurate on the VCPUs which incur interference because they may include the time spent in background tasks. In this work, we used KVM hypervisor where CPU time ($t_i$) measurements include the time stolen by the interfering VM. Still, periodic migration of tasks from overloaded to underloaded VCPUs ensures that good load balance is achieved after a few steps, illustrating the wide applicability of our approach (Section 5.5).

## 5.4 Evaluation Methodology

We setup a cloud using OpenStack [25] on Open Cirrus testbed at HP Labs site [31]. We created our own cloud to have control over the VM placement strategy, which enabled us to get specific configurations to test the correctness and performance of our techniques. This testbed has inherent heterogeneity since it consists of 3 types of physical servers:

- 4 × Intel Xeon E5450 (12M Cache, 3.00 GHz) – *Fast*
- 4 × Intel Xeon X3370 (12M Cache, 3.00 GHz) – *Fast*
- 4 × Intel Xeon X3210 (8M Cache, 2.13 GHz) – *Slow*

**Algorithm 3** Refinement Load Balancing for Cloud

```
1: On Master VCPU on each load balance step
2: for p ∈ [1, P] do
3:    if isHeavy(p) then
4:       overHeap.add(p)
5:    else if isLight(p) then
6:       underSet.add(p)
7:    end if
8: end for
9: createOverHeapAndUnderSet()
10: while overHeap NOT NULL do
11:    donor = deleteMaxHeap(overHeap)
12:    (bestTask, bestCore) = getBestCoreAndTask(donor, underSet)
13:    m^k_bestTask = bestCore
14:    updateHeapAndSet()
15: end while

16: procedure isHeavy(p) {isLight(p) is same except that the condition at line 21 is
    replaced by by Tk_avg − totalTicks > ϵ}
17:       for i ∈ [1, N_p]
18:             totalTicks+ = t_i * f_p
19:       end for
20:       totalTicks+ = O_p * f_p
21:       if totalTicks − Tk_avg > ϵ
22:             return true
23:       else
24:             return false
25:       end if
26: end procedure
```

We will refer to the first two processors types as *Fast* and the third one as *Slow*. These nodes are connected using commodity Ethernet – 1Gbps internal to rack and 10Gbps cross-rack.

Similar to the setup in Chapter 2, we used KVM [33] for virtualization, since past research has suggested that KVM is a good choice as a hypervisor for HPC clouds [34]. We experimented with different network virtualization drivers – *rtl8139, eth1000*, and *virtio-net*, and chose *virtio-net* since it resulted in best network performance [85]. We present results with VMs of type *m1.small* (1 core, 2 GB memory, 20 GB disk), up to 64 VMs. It should be noted that these choices do not affect the generality of our results. Since there is one VCPU per VM, we use VCPU and VM interchangeably in Section 5.5. To get best performance, we pin the virtual cores to physical cores using `vcpupin` command.

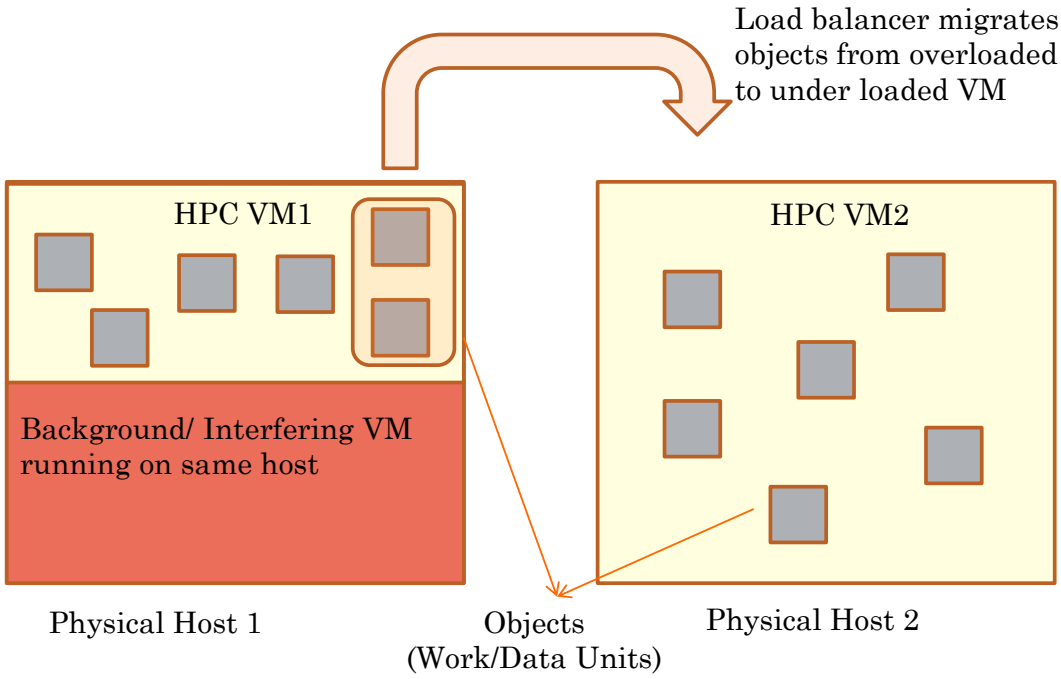



**Figure 5.3: Load balancing approach for HPC in cloud**

To evaluate the load balancer in presence of interfering VMs, we run the sequential NPB-FT (NAS Parallel Benchmark - Fourier Transform) Class A [39] in a loop to generate load on the interfering VM. The interfering VM is pinned to one of the cores that the VMs of our parallel runs use. The choice of NPB-FT is random and does not affect the generality of results. For experiments involving heterogeneity, we use one *Slow* node and rest *Fast* nodes.

The HPC benchmarks and application used are:

- Stencil2D – A computation kernel which iteratively averages values in a 2-D grid using 5-point stencil. It is widely used in scientific simulations and numerical algebra.
- Wave2D – A tightly coupled benchmark which uses finite differencing to calculate pressure information over a discretized 2D grid, for simulation of a wave motion.
- Mol3D – A 3-D molecular dynamics simulation application. We used the *Apoa1* dataset (92K atoms).

We used the `net-linux-x86-64` machine layer of Charm++ with –O3 optimization level. For Stencil2D, we used problem size $8K \times 8K$. For Wave2D, we used problem size $12K \times 12K$. Each object size is kept $256 \times 256$, unless otherwise specified. These parameters were determined through experimental analysis as discussed later.

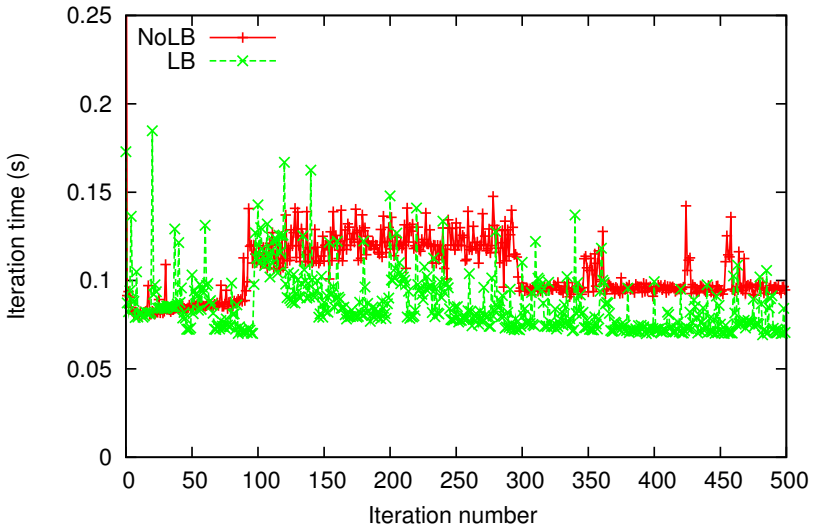
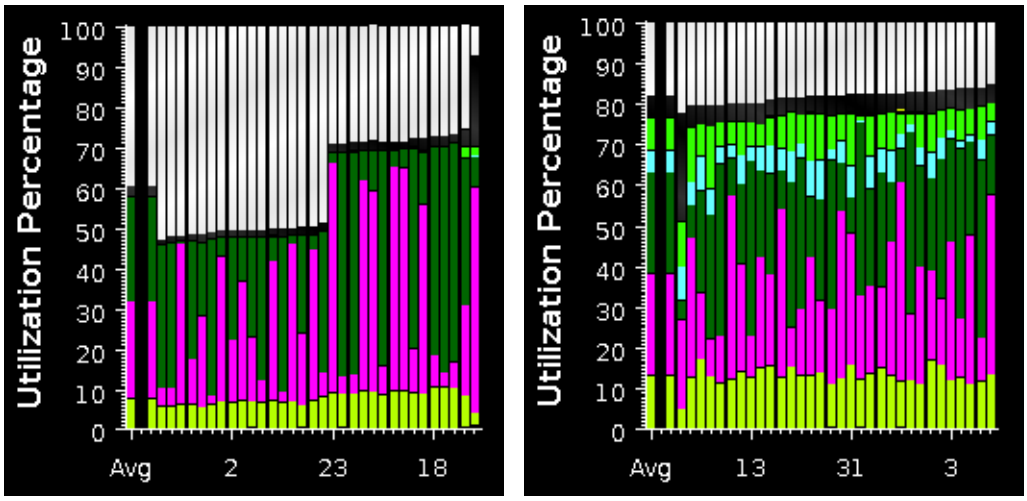



**Figure 5.4: LB vs. NoLB: 32 VMs Stencil2D on heterogeneous hardware, interfering VM from $100^{th}$ to $300^{th}$ iteration**

## 5.5 Experimental Results

To understand the effect of our load balancer under heterogeneity and interference, we ran 500 iterations of Stencil2D on 32 VMs (8 physical nodes – one *Slow*, rest *Fast*), and plotted the iteration time (execution time of an iteration) vs. iteration number in Figure 5.4. In this experiment, we started the job in interfering VM after 100 iterations of parallel job had completed. Load balancing was performed after every 20 steps, which manifests itself as spikes in the iteration time every $20^{th}$ step. We report execution times averaged across three runs and use wall clock time, which includes the time taken for object migration. The LB curve starts to show benefits as we reach 100 iterations, due to heterogeneity-aware load balancing. After 100 iterations, interfering job starts. When the load balancer kicks in, it restores load balance by redistributing tasks among VMs according to Algorithm 3. Now, there is a large gap between the two curves demonstrating the benefits of our refinement based approach, which takes around 3 load balancing steps to gradually reduce the iteration time after interference appears. The interfering job finishes at iteration 300 and hence the NoLB curve comes down. There is little reduction in LB curve, since the previous load balancing steps had reduced the impact of interference to a very small amount. The difference between the two curves after that is due to heterogeneity-awareness in load distribution.

To confirm that the achieved benefit is due to better load balance, we used Projections [45] tool for performance analysis. Figure 5.5 shows the improvement in CPU (VCPU) utilization with load balancing for Stencil2D on 32 VMs, all running on *Fast* nodes in this case, with one interfering VM. In this figure, y-axis represents CPU utilization, x-axis is (virtual) CPU




(a) NoLB (b) LB

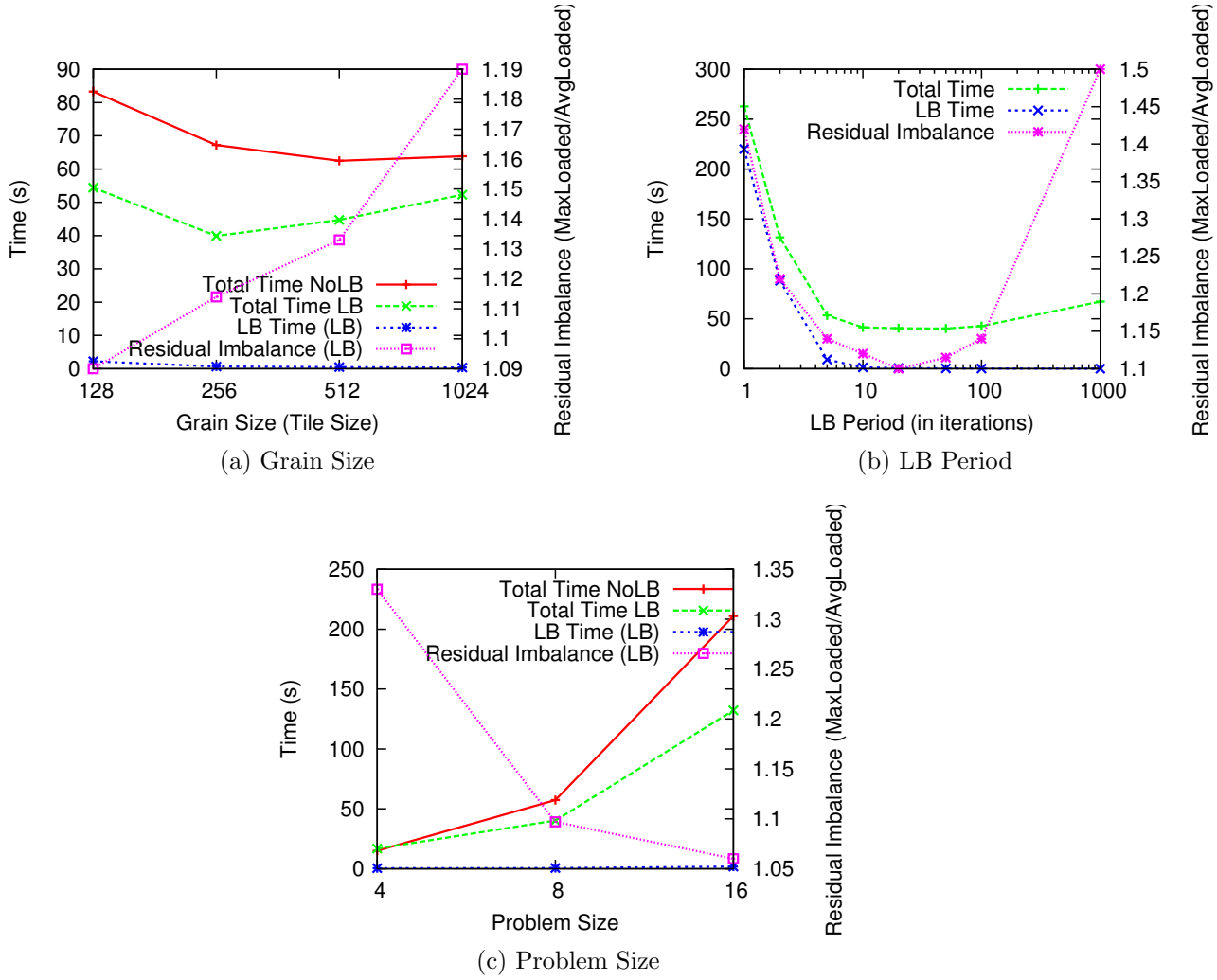
**Figure 5.5: CPU utilization of Stencil2D on 32 VMs: white = idle time, black = overhead (including background load), colored portions = application functions, x-axis = VCPU.**

number, with first bar as average. White portion (at top) represents idle time and colored portions (shaded) depict application functions. Black portion (below white) is overhead (see last processor in Figure 5.5a). The CPUs (x-axis) are sorted by the idle time using extrema analysis techniques. Observing Figure 5.5a, it is clear that there are 3 different clusters – 50%, 70%, and 90% utilization level. The difference between first two is due to the use of 2 types of processors. Though they have same processor frequency, the actual performance achieved is significantly different. There is very little load imbalance among processors in the same cluster, indicating that the distribution of work to processors is equal. The third cluster belongs to the VM which incurs interference. Figure 5.5b balances load. Here the idle time is due to the communication time, and is uniform across all CPUs. Overall, the average utilization increased from 60% to 82% using load balancing.

Next, we analyze the effect of few parameters on load balancer performance followed by results for various applications.

### 5.5.1 Analysis using Stencil2D

To achieve better understanding, we experimented with Stencil2d on 32 VMs (*Fast* processors, one interfering VM), and ran 500 iterations. We varied grain size, load balancing frequency and problem size. Here, we present our findings.




(a) Grain Size

(b) LB Period

(c) Problem Size

**Figure 5.6: Impact of various parameters on load balancing and overall performance of Stencil2D, 500 iterations**

### Effect of Grain Size and Overdecomposition

Grain size refers to the amount of work performed by a single task (object in Charm++ terminology). For Stencil2D, it can be represented by the matrix size of an object. Figure 5.6a shows the variation in execution time with object sizes. First, consider the NoLB case. As we decrease the grain size, hence increasing number of objects per processor, execution time decreases due to the benefits of over-decomposition: (*a*) better cache reuse by exploiting temporal locality and (*b*) hiding network latency through automatic overlap of computation and communication. After a threshold, time starts increasing, due to the overhead incurred by the scheduler and runtime for managing large number of objects. Hence, as we make objects very fine grained, performance degrades. Here, best performance is obtained with

grain size = $512 \times 512$ elements.

The LB case (load balancing every 20 steps) introduces additional factors – ($a$) time spent in load balancing and ($b$) load balancing quality, which affect the overall performance. From Figure 5.6a, we see that the total load balancing time (LB time), which includes time to make migration decisions and time spent in migrating objects, is negligible compared to the total time. However, LB time will be important as the computation time decreases (e.g. strong scaling), and the ratio of LB time to compute time increases. For ($b$), we calculate average $Residual\,Imbalance = \frac{Max\,Compute\,(+Background)\,Load}{Avg\,Compute\,(+Background)\,Load}$ over all iterations and plot it on the same figure, with the right y-axis being the legend. We see that we get better load balance as we decrease object size, achieving residual imbalance of 1.09 for $128 \times 128$. However, the best execution time is achieved for object size $256 \times 256$ due to the impact of additional factors, such as overhead. In general, we observed good load balance with degree of decomposition (ratio of objects to processors) $> 20$.

### Effect of Load Balancing Frequency

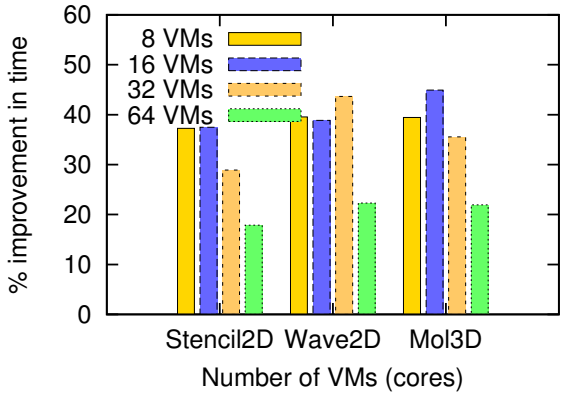
Next, we vary the load balancing frequency in the same setup, with fixed grain size of $256 \times 256$ based on the results above. Figure 5.6b shows that the there is an optimal load balancing period. Very frequent load balancing (small LB period) results in most of the time being spent in making task migration decisions and migrating the data associated with objects, which degrades application performance. Moreover, it also results in lack of enough instrumented data for the load balancer to make intelligent decisions, leading to large residual imbalance. With very infrequent load balancing, such as LB period = 100 iterations, load balancer will be slow to adapt to the dynamic variations, leading to large residual imbalance. Hence, we settle for LB period = 20 iterations (which equals approximately 2 seconds here) except for runs on 64 VMs, where we used a period = 50 iterations. The optimal LB period depends on the iteration time and should be larger for small iteration time so that the gains achieved by improved load balance are not offset by time spent in balancing load.

### Effect of Problem Size

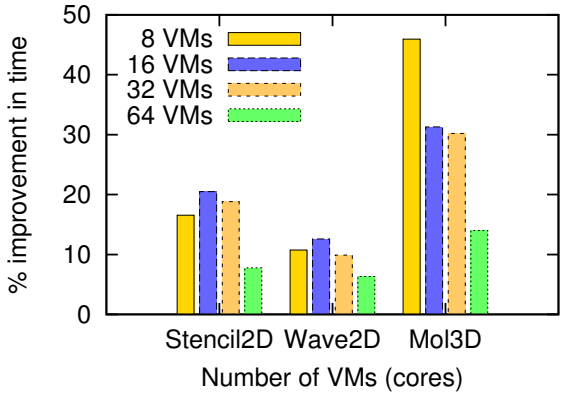
Finally, we analyze the performance benefits for different problem sizes of same application – Stencil2D in the same setup, with fixed grain size of $256 \times 256$ and LB period = 20. Figure 5.6c shows that the benefits increase with increasing problem size. With larger problem size, the computation granularity increases, reducing communication-to-computation ratio. Hence, any improvement in computation time through load balancing will result in higher

impact on the total execution time. Also, we get better load balance quality as we increase problem size – for $16K \times 16K$ matrix, we get residual imbalance of 1.05.

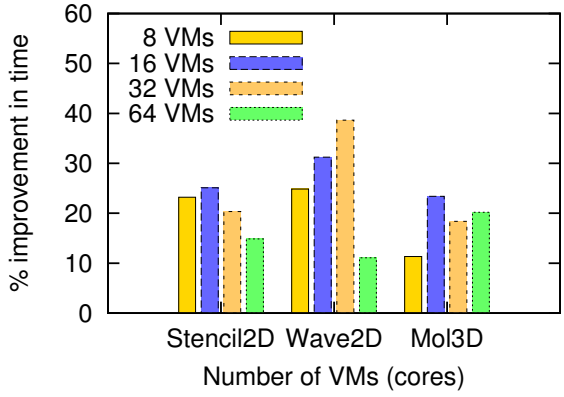
Application which are less communication-intensive are the most cloud-friendly one and are expected to be run in the cloud most because of better scalability. Also, the model expected to work better in cloud is weak scaling (same problem size per core with increasing cores) rather than strong scaling (same total problem size with increasing cores) [86]. In that context, our approach will be extremely useful for HPC in cloud.




(a) Interference




(b) Heterogeneity




(c) Heterogeneity, Interference

**Figure 5.7: % benefits obtained by load balancing in presence of interference and/or heterogeneity for three different applications and different number of VMs, strong scaling**

### 5.5.2 Performance and Scalability of Three Applications

To evaluate the robustness and efficacy of our load balancer in actual cloud scenario, we study three different cases (shown in Figure 5.7) – (a) **Interference** - one interfering VM, all *Fast* nodes, (b) **Heterogeneity** – one *Slow* node, hence four *Slow* VMs, rest *Fast* and (c) **Heterogeneity and Interference** – one *Slow* node, hence four *Slow* VMs, rest *Fast*, one interfering VM (on a *Fast* core) which starts at iteration 50. We ran 500 iterations for Stencil2D and Wave2D and 200 iterations for Mol3D, with load balancing every $20^{th}$ step. We kept same problem size while increasing number of VMs (strong scaling).

Figure 5.7 shows that we achieve significant % reduction ($= \frac{T_{NoLB} - T_{LB}}{T_{NoLB}} \times 100$) in execution time using load balancing compared to the NoLB case, for different applications and different number of VMs under all three configurations. With an interfering VM, we achieve up to 45% improvement in execution time (Figure 5.7a). The amount of benefits is different for different applications, especially in Figure 5.7b. We studied this behavior using Projections tool and found that our load balancer is distributing tasks well for all applications, but the difference in achieved benefits is due to the different sensitivity of these applications to CPU type and frequency. It can be inferred from Figure 5.7b that Mol3D is the most sensitive to CPU frequency, having most scope for improvement. Next, Figure 5.7c shows that our techniques are also effective in the presence of both the effects – the inherent hardware heterogeneity and the heterogeneity introduced by multi-tenancy.

Another observation from Figure 5.7 is the variation in the achieved benefits with increasing number of VMs. This is attributed to the tradeoff between two factors: (1) Since there is only one VM sharing physical CPU with interfering VM, running on larger number of VMs implies distributing the work of the overloaded VM to an increasing number of underutilized VMs, which results in larger reduction in execution time (e.g. Figure 5.7a Mol3D, 8 VM vs. 16 VM). (2) As we scale, the average compute load per VM decreases. Hence, other factors, such as communication time, dominate the total execution time. This implies that even with better load balance and higher % reduction in compute time, overall benefit is small, since communication time is unaffected. Hence, as we scale further, benefits decrease, but they still remain substantial.

Moreover, as we scale, benefits can actually be more important because we save across larger number of processors. As an example, from Figure 5.7b Stencil2D, we save 18.8% over 32 VMs, with absolute savings $= 32 \times 0.188 \times 48.09 = 289.28$ CPU-seconds, where application took 48.09 seconds on 32 VMs. Juxtaposing this with 20.5% reduction in time over 16 VMs which results in absolute savings $= 16 \times 0.205 \times 86.77 = 284.32$ CPU-seconds, where application took 86.77 seconds on 16 VMs, we see that the overall savings are more

for 32 VM case compared to 16 VMs. Also, since we achieve higher % benefits with larger problem sizes (Section 5.5.1, Figure 5.6c), the benefits will be even higher for weak scaling compared to strong scaling as the impact of factor 2 above is minimized.

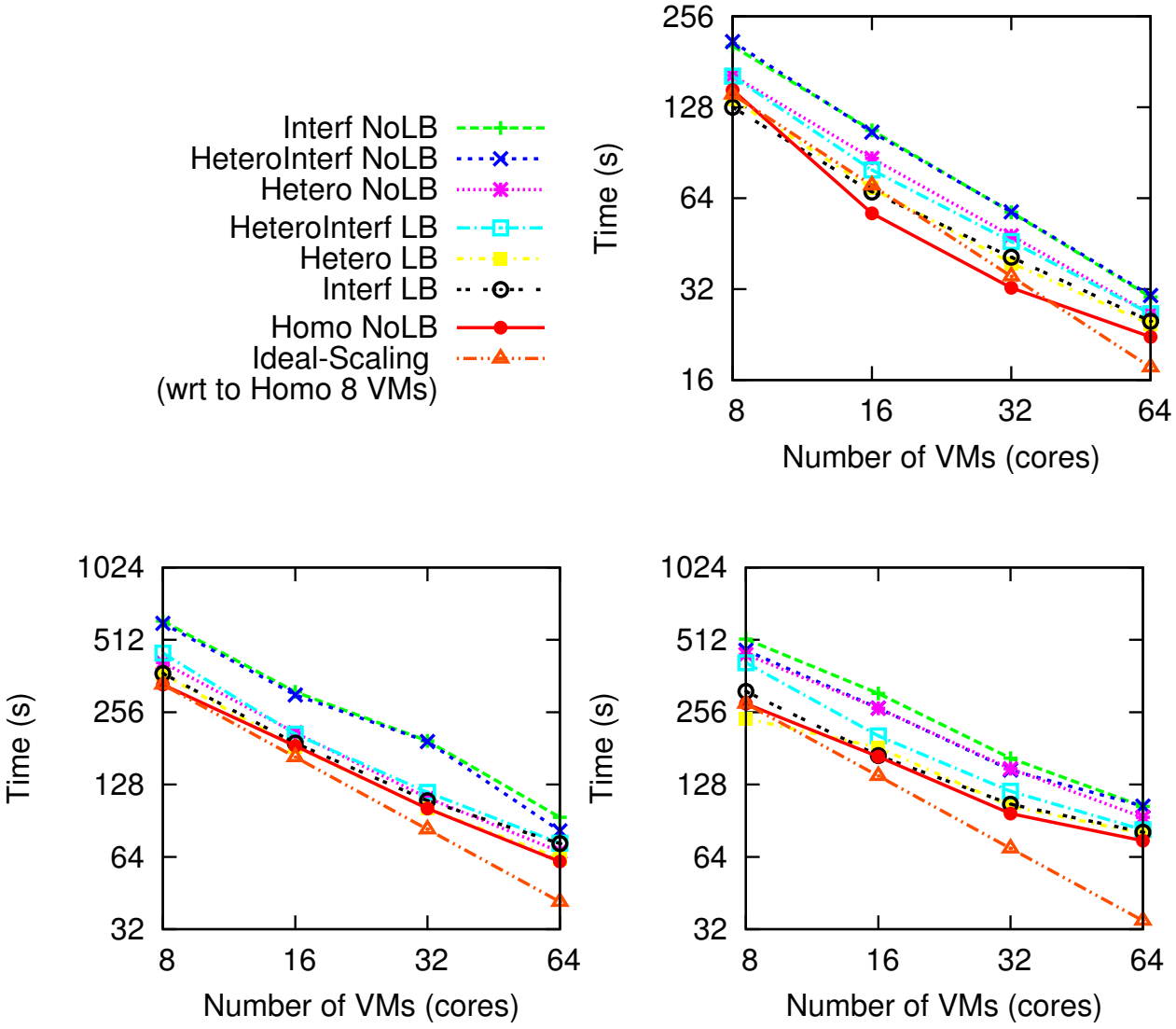



**Figure 5.8: Scaling curves with and without load balancing in presence of interference and/or heterogeneity for three different applications, strong scaling**

The primary objective of running in parallel is to get reduced execution time with increasing compute power. However, it is not clear from Figure 5.7 whether we achieve that goal. Hence, we plot execution time vs. number of VMs for same experiments, as shown in Figure 5.8. For the purpose of comparison, we also include the runs without any interfering VMs, with all VMs mapped to *Fast* nodes (*Homo* curve in Figure 5.8). Figure 5.8 shows that

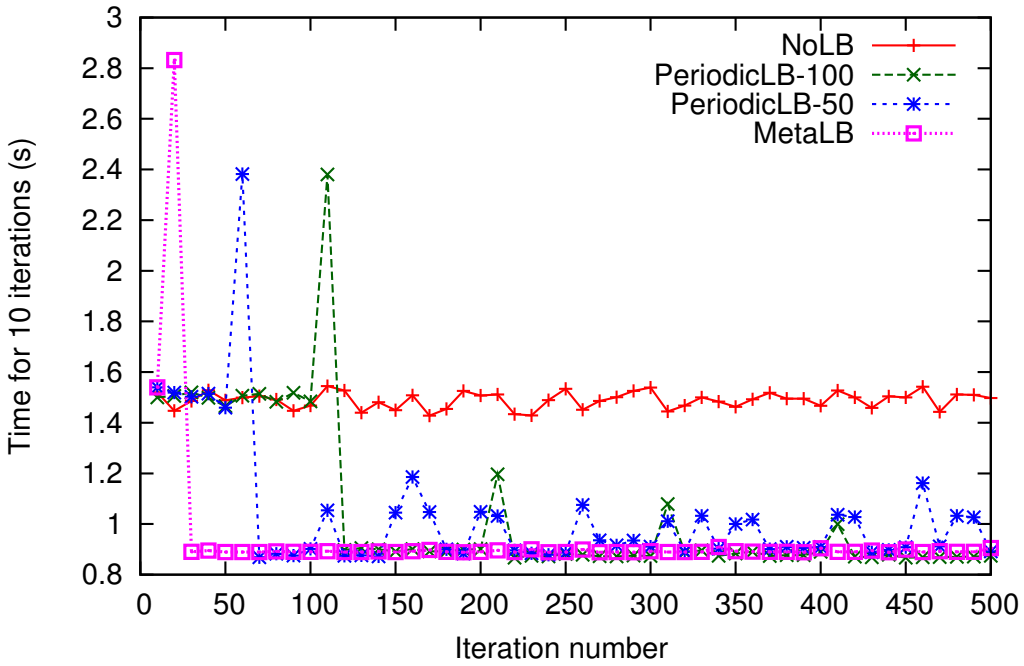



**Figure 5.9: Effect of Load balancing period on iteration time. Figure shows Wave2D on heterogeneous hardware on 64 cores (Grid'5000-Distem setup)**

our load balancer brings the NoLB curves down and close to the Homo curve. At some data points, the LB curves are below the homo curve, since even the Homo curves can benefit from load balancing (shown earlier in Figure 5.5 that application performance on two types of *Fast* processors is somewhat different). The super-linear speedup achieved in some cases can be attributed to better cache performance.

The scale of our experiments was limited by the availability of nodes in this cloud setup since we needed administrative privileges from provider perspective. For evaluation on lager scale, we use an alternative testbed as shown in the next section.

### 5.5.3 Larger-scale Evaluation on Grid'5000 using Distem

To evaluate our approach on a larger scale, we used a cloud setup on Grid'5000 [87] testbed – Graphene cluster at Nancy site. Linux containers [35] are used for virtualization and Distem [88] is used for creation of artificial heterogeneity in homogeneous clusters.

In Section 5.5.1, we saw that when performing periodic load balancing, there is a trade-off between the overhead of load balancing and the expected benefits. We further investigate the effect of load balancing period on 64 cores of this testbed (See Figure 5.9). Figure 5.9 shows

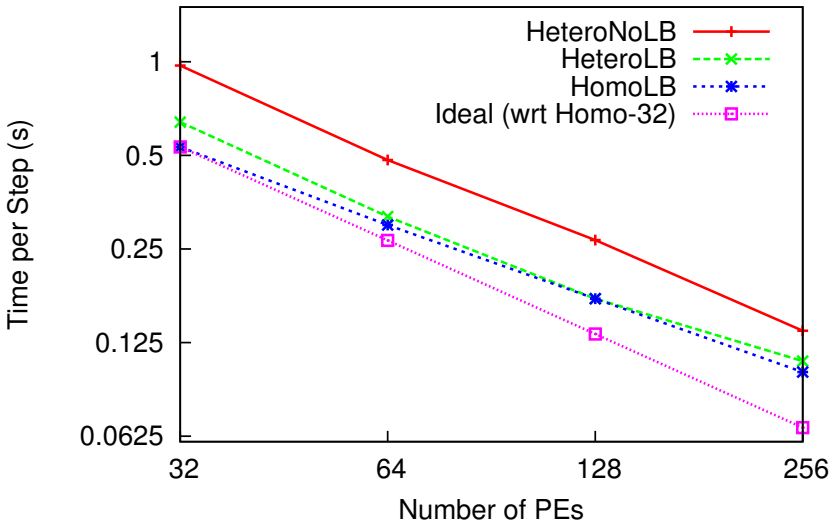



**Figure 5.10: Performance and scalability of LeanMD in heterogeneous cloud environments (Grid'5000-Distem setup)**

that when load balancing is performed very frequently (e.g 100 iterations – *PeriodicLB-100* curve), there is large overhead incurred due to load balancing, which is seen as frequent spikes in the iteration time. In contrast, when load balancing interval is too high (e.g after every 50 iterations – *PeriodicLB-50* curve), the overhead is small but it takes longer to adapt to the environmental variations.

The optimal load balancing period depends on the iteration time, which in turn depends on the application, data-set (problem size), and the scale of experiments. As we scale to larger core counts, iteration time decreases but the overhead of load balancing increases. Moreover, as we scale up, the interference becomes even more random, unpredictable, and varies significantly from run to run. It is impossible to a-priori determine optimal load balancing period, for instance, load balancing may not be required at all in a run where no interference occurs. Hence, instead of application-triggered periodic load balancing, we build upon a runtime-triggered approach [89]. In this approach, the Charm++ runtime system monitors the application continuously and invokes the load balancer when an imbalance is detected and if the cost of load balancing is not more than the benefit. We modified the approach in [89] to consider the interference when deciding about the load balancing period. Using this approach, load balancing is invoked only when necessary and when the cost of performing load balancing is predicted to be small compared to the expected benefits. Figure 5.9 *MetaLB* curve shows this runtime-triggered approach.

Figure 5.10 shows the performance benefits and scalability of our cloud-aware load bal-

ancing approach for LeanMD. We introduced heterogeneity in this testbed by making one node's effective CPU frequency as 0.7X of the rest. Figure 5.10 shows that with our approach, the performance attained is very close to that attained for homogeneous case. We believe that our load balancing techniques will be equally applicable to even larger scales since the effectiveness of similar load balancing strategies have been demonstrated on large-scale supercomputer applications [90].

## 5.6 Related Work

The approaches taken to reduce the gap between traditional cloud offerings and HPC demands can be classified into two broad categories – (1) those which aim to bring clouds closer to HPC and (2) those which want to bring HPC closer to clouds. Examples of (1) include HPC-optimized clouds such as Amazon Cluster Compute [15] and DoE's Magellan [13]. Another area of research in (1) is making cloud schedulers (VM placement algorithms) aware of the underlying hardware and the nature of HPC applications. Examples include our own work presented in Chapter 4 [60] and OpenStack community towards making OpenStack scheduler architecture-aware [17]. Related work on modifying the scheduling in virtualization layer include VM prioritization and co-scheduling of co-related VCPUs [91]. The latter approach (2) has been relatively less explored. Fan et al. proposed topology aware deployment of scientific applications in cloud, and mapped the communication topology of an HPC application to the VM physical topology [19].

In this work, we took the latter approach and explored whether we can make HPC applications more cloud friendly using a customized parallel runtime system. We built upon the earlier work on load balancing for addressing background loads [84]. This work extends previous work in multiple ways – First, in [84], we assumed that we can get accurate compute time for tasks, that is independent of the background load. However, in cloud, it may not be possible to separate the time taken by background load from application load since virtualization hides the presence of time-sharing of physical nodes among VMs. Secondly, we make the load balancer aware of both – hardware heterogeneity and multi-tenancy in cloud. Thirdly, we evaluate our techniques on an actual cloud with VMs whereas in the earlier work, we did not consider effects of virtualization. Brunner et al. [92] proposed a load balancing scheme similar to ours but in the context of workstations. Our work differs from theirs in the same ways as above. Moreover, our scheme uses a refined load balancing algorithm that reduces number of task migrations.

## 5.7 Lessons, Conclusions, and Future Work

In this chapter, we presented a load balancing technique which accounts for heterogeneity and interfering VMs in cloud and uses object migration to restore load balance. Experimental results on actual cloud showed that we were able to reduce execution time by up to 45% compared to no load balancing. The lessons learned and insights gained are summarized as:

- Heterogeneity-awareness can lead to significant performance improvement for HPC in cloud. Adaptive parallel runtime system are extremely useful in that context.
- Besides the static heterogeneity, multi-tenancy in cloud introduces dynamic heterogeneity, which is random and unpredictable. The overall effect being poor performance of tightly-coupled iterative HPC applications.
- Even without the accurate information of the nature and amount of heterogeneity (static and dynamic but hidden from user as an artifact of virtualization), the approach of periodically measuring idle time and migrating load away from time-shared VMs works well in practice.
- Tuning the parallel application for efficient execution in cloud is non-trivial. Choice of load balancing period and computational granularity can have significant impact on performance but the optimal values depend on application characteristics, size, and scale. Runtime systems which can automate the selection and dynamic adjustment of such decisions will be increasingly useful in future.

We believe that some of our approaches and analysis can also be leveraged for future exascale applications and runtimes. According to an exascale report, one of the Priority Research Direction (PRD) for exascale is to "develop tools and runtime systems for dynamic resource management" [93].

Future directions include exploration of the use of VM steal cycles, where supported. Also, we have demonstrated that our techniques work well with iterative applications, and when the external noise is quite regular. It needs to be explored how they work or how they should be adapted when the interference is irregular, such as fluctuating loads of web applications.

CHAPTER 6

# A Parallel Runtime for Malleable Jobs

On-demand provisioning and multi-tenancy are essential characteristics of cloud computing. Clouds serve diverse workloads with dynamic demands and jobs with different levels of priority. In cloud environment, adaptivity is crucial to achieve better utilization of system components. One direction to achieve such adaptivity is to enable malleable jobs. Adaptive or Malleable jobs are those which can change the number of processors on which they are executing at runtime in response to an external command. Such jobs can expand when the cluster has low demand, and shrink when there is high demand. Figure 6.1 illustrates this with an example. Each box represent the current utilization of the compute capacity (say 100 nodes) of a cluster by jobs A, B, and C. In this case, a long running job A can be made to shrink or expand to adapt to current demands. In the absence of malleability, job A which is using 60 nodes, can make job C which needs at least 50 nodes wait till its completion, resulting in wastage of 40 nodes. This wastage can be avoided if there are smaller jobs ready for execution, but this may not be always be the case. Malleable jobs are an excellent alternative solution to this problem, and it has been shown that they can potentially improve system utilization by up to 25%, and also reduce mean job response time [94–96].

Malleable jobs have wider applicability for HPC than just for clouds. As we move towards exascale era in High Performance Computing, supercomputers will need to operate under power constraints and failing components [93]. In such environment, malleable jobs can be extremely useful in increasing the level of adaptivity.

To enable malleable jobs, three components are critical (Figure 6.2) – (1) *a smart adaptive job scheduler*, which can make decisions on when and which jobs to expand or shrink, based on the job queue, current cluster state, and a job scheduling policy, (2) *an adaptive resource manager*, which allocates nodes to jobs (node scheduler) and executes the scheduling decisions by coordinating between the job scheduler and the cluster and running jobs

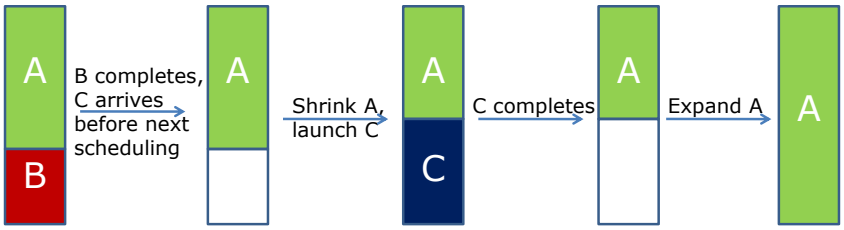



**Figure 6.1: Example Use Case**

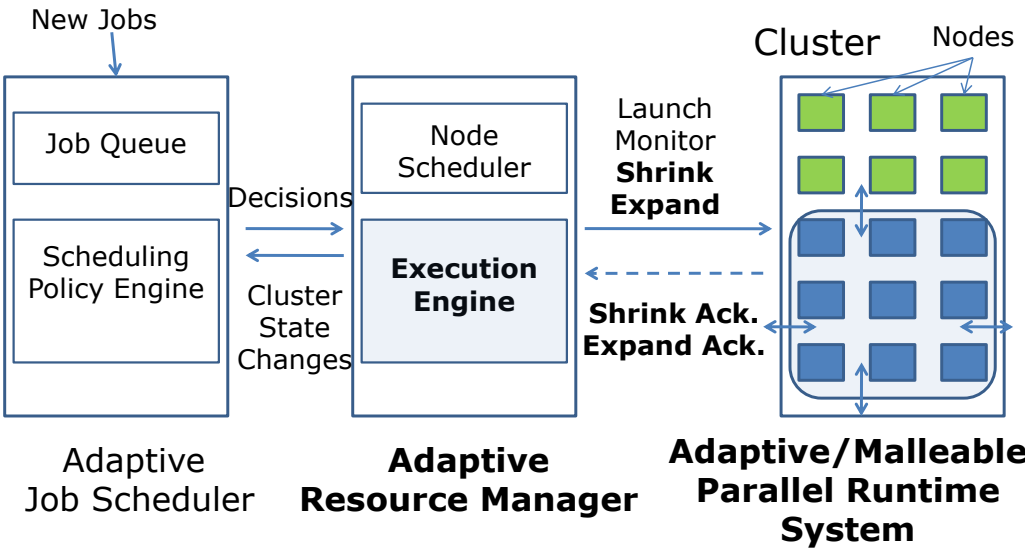



**Figure 6.2: System Overview**

(execution engine), and (3) *an adaptive parallel runtime system* which provides the dynamic shrink-expand capability. Although the job scheduling strategies for malleable jobs have been exhaustively researched [94, 95, 97–100], there are very few runtime systems which can actually perform shrink or expand on general purpose parallel programs. Existing techniques either perform pseudo shrink-expand by leaving residual process on nodes which are vacated as a result of shrink [94, 96] or require too much application-specific programmer effort for data re-decomposition after resize [97]. Further, the integration of these three components has been little researched. The scheduler needs to communicate to the application its shrink/expand decisions, and the application needs to acknowledge when it is done.

Figure 6.2 highlights our focus in this chapter. Our goal is to address the research challenges involved in designing an end-to-end system which can provide and exploit job shrink/expand capability. In addition, we aim to explore non-traditional applications of this capability. Towards realizing these objectives, we make the following contributions:

- A novel technique for providing a fast, efficient, and scalable shrink/expand capability to a parallel runtime system. Salient features of our scheme are task migration,

checkpoint-restart, load balancing, and use of Linux shared memory (§6.2).

- A technique for enabling split-phase execution of malleable job scheduling decisions in a shared cluster, incorporating scheduler-runtime communication (§6.3).
- Implementation atop Charm++ and demonstration of malleability using a benchmark and three mini-applications up to 2048 cores on Stampede supercomputer (§ 6.2.3, §6.5).
- Demonstration of the benefits of shrink/expand capability with traditional as well as novel use cases – proactive fault tolerance and HPC in cloud (§6.5.5, §6.6)

## 6.1 Related Work

Feitelson and Rudolph [101] classified parallel jobs into four categories based on – who decides the number of processors a job will be run on, and when it is decided (Table 6.1). In both moldable and malleable jobs, users specify a range of processors a job can be run on, based on factors such as its strong scaling performance and memory limitations. In this thesis, our focus is on malleable jobs where the scheduler can dynamically change the resources allocated to a job.

### 6.1.1 Runtimes with Shrink/Expand Capability

Kale et al. [96] demonstrated Charm++ jobs with the ability to shrink their node footprint by migrating objects away from one or more nodes used by the job or expand their node footprint by migrating objects back to nodes that were previously evacuated [96]. After the runtime system carries out object migrations to move load away from some processors, individual processes are still left on processors that are removed from the available processor pool. These processes contain low-level agents that carry out processor-based tasks, such as forwarding messages for migrated objects to their new homes, spanning tree-based reductions, and implementing communication optimizations.

This approach is not feasible in the context of real HPC systems since the residual processes can act as a source of interferences to other applications. The interference could be both – performance (noise) and security. Moreover, with multiple malleable jobs in the system, there would be multiple such residual processes on each node. The interference would quickly reach a level which would make it intolerable for large-scale HPC applications. Moreover, in this approach, for expand, the job size is limited to the number of nodes where it was initially launched. Hence, for efficient and true resize, one needs to eliminate the presence

**Table 6.1: Job Type Taxonomy**

| Who decides | When it is decided | |
|---|---|---|
| | At submission | During execution |
| User | Rigid | Evolving |
| System/scheduler | Moldable | Malleable |

of residual agents. However, simply migrating them to new locations will not serve the purpose, because their operation is highly dependent on the available number of processors as well as the topology of the system partition they run on. For example, if these agents implement spanning-tree based optimized reductions, simply migrating them to the new processors will introduce inefficiency. Novel techniques need to be designed to kill/launch a process dynamically, and then redistribute the load.

Cera et al. [94] demonstrated two techniques to provide malleable MPI applications – dynamic CPUSETs mapping and dynamic MPI, using OAR resource manager. Dynamic CPUSETs technique is specific to multi-core machines. It enables dynamic alteration in the number of cores per node allocated to an application. Their second technique is more general and allows shrinking or growing using MPI process spawning primitives (such as `MPI_Comm_spwan`). However, they do not vacate residual processes in case of shrinking. Moreover, significant application programmer effort is necessary to perform data re-decomposition after resize.

Perhaps the work most similar to ours is the research on dynamic malleability of iterative MPI applications using PCM (Process Checkpoint and Migration) library [97]. That work conceives malleability as split and merge operations supported using PCM calls added to application code. However, since MPI applications are processor-centric, their scheme needs significant application code modification for performing data re-decomposition after resize. We address this problem by leveraging the over-decomposition of data into migratable objects or medium-grained tasks. Our approach requires minimal application-level code changes to support malleability.

### 6.1.2 Adaptive Schedulers and Resource Managers

Several studies have demonstrated the benefits of scheduling algorithms which consider malleable jobs, using theoretical analysis [95, 98] and/or simulation of scheduling algorithms on supercomputer job traces [95, 99]. For instance, Hungershofer demonstrated that simple strategies such as *equipartitioning* can result in significant improvement in response time and

utilization using malleable jobs [99]. Utrera et al. [100] demonstrated benefits of a malleable processor allocation technique based on a combination of moldability and folding techniques – Folding by JobType (FJT). Scheduling policies for malleability have also been studied in context of grids using KOALA multi-cluster scheduler and DYNACO framework [102].

In this thesis, we do not intend to research scheduling algorithms for malleable jobs. Instead, we study the challenges in actual execution of malleable jobs by a resource manager in an HPC system. To this end, we present a mechanism for interaction between the scheduler and running parallel application, and address the issues in enforcing scheduling decisions in presence of malleable jobs.

## 6.2 Shrink/Expand in Parallel Runtime System

To enable malleable jobs, the foremost requirement is a parallel runtime system which can render applications malleable, preferably without much programmer effort. In this section, we discuss our approach towards malleability in a runtime system. When we say runtime, we mean a parallel runtime system. For enhanced understanding, we first define *shrink* and *expand* operations and present the challenges that we considered while designing such runtime system.

### 6.2.1 Definitions and Design Goals

***Shrink*:** A parallel application running on nodes of set A is resized to run on nodes of set B where $B \subset A$

***Expand*:** A parallel application running on nodes of set $A$ is resized to run on nodes of set $B$, where $B \supset A$

***Rescale:*** *Shrink* or *expand*

An alternative definition is possible, where the subset and super-set relationships for *shrink* and *expand* respectively are not necessary. For example, on *shrink*, a job may be allocated a new set of nodes to replace a subset of old nodes. One of the motivations for such re-allocations is to provide contiguous allocation on resize. In our definitions, such cases can be handled by performing *expand* followed by *shrink*.

While exploring mechanisms to provide *rescale* capability in a runtime system, we focused on certain design challenges. Our approach towards a malleable runtime should be:

- *Efficient:* It should ensure that achieved performance after *rescale* is proportional to the compute power.

- *Fast:* The *rescale* time ($T_{rescale}$) should be sufficiently small to satisfy the needs of its usage scenarios. We expect the granularity of *rescale* events to be few minutes or even more, so $T_{rescale}$ around 1 minute should be permissible.
- *Scalable:* The approach should scale well with increasing number of nodes and with increasing problem sizes.
- *Generic:* the approach should be generic from runtime perspective, any parallel runtime system which provide the features necessary for our approach should benefit from the findings of our research.
- *Practical:* It should be applicable to most supercomputers, commodity clusters, and possibly even clouds.
- *Low-effort:* The runtime should ensure that there is little or no application-specific programmer effort required to render a parallel application malleable.

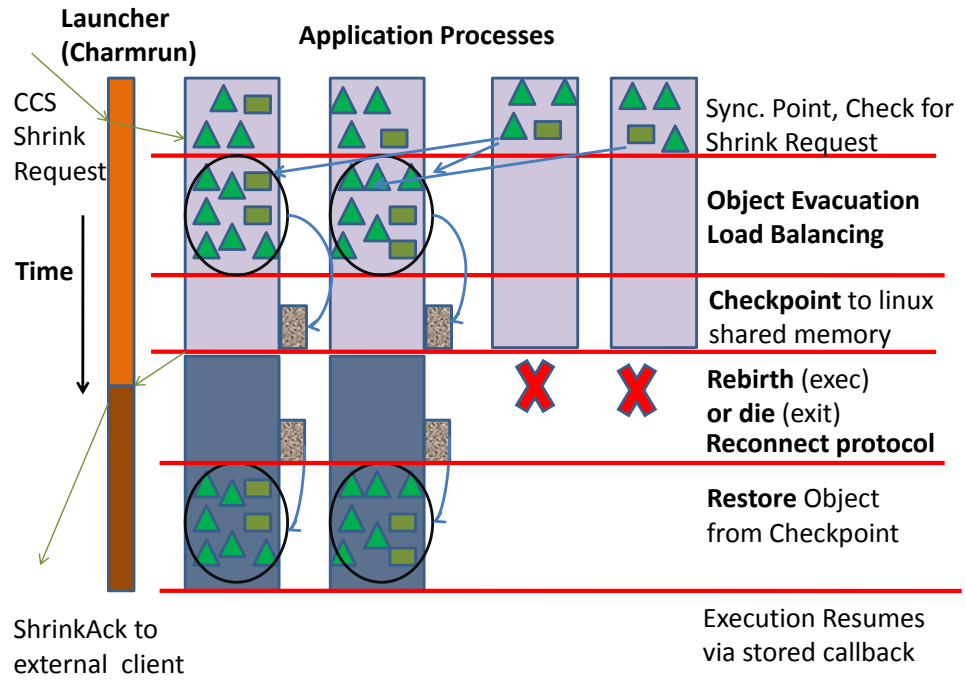
### 6.2.2 Approach

The primary enabling capabilities that our approach needs from a parallel runtime are over-decomposition and task (or object) migration. These features are present in many established [27, 103] and emerging [104] parallel programming environments, and are becoming more relevant as we approach exascale [93]. The main idea behind over-decomposition is that the application should be decomposed into medium-grained work/data units typically larger in number than the number of processors, which can be mapped (and adaptively re-mapped) by the runtime to processors.

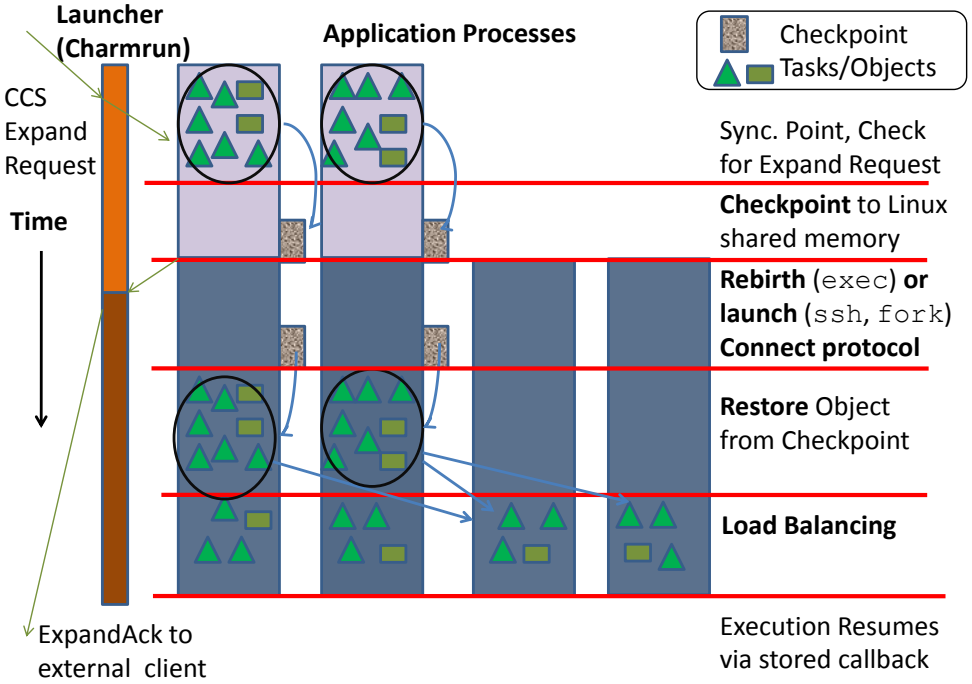
Figure 6.3 describes our technique. The parallel application acts as a server which can listen to incoming *rescale* requests from external sources, such as a job scheduler. These requests can be received at any time and are recorded in the system. However, they are handled at the next application-specified synchronization point, such as iteration boundary in iterative HPC applications. We first discuss our approach in the context of *shrink* (Figure 6.3a)

***Shrink:*** If the request is of type *shrink*, the request needs to specify which processes out of the original set, does the application need to relinquish. One mechanism is to specify a bit-vector $V_s$ of size $P_{old}$, where $P_{old}$ is the number of processes before *shrink*. If the $i^{th}$ bit in $V_s$ is zero, that means that the processor on which $i^{th}$ process resides will soon be unavailable.

At the synchronization point, a special load balancer or a task evacuation module is invoked, which is cognizant of this bit-vector information and migrates the tasks away from




(a) Shrink




(b) Expand

**Figure 6.3: Shrink Expand Runtime System Design**

the processors marked as unavailable. While evacuating the tasks from the unavailable processors, the load balancer must ensure that the load is well balanced in the remaining set

of processors.

After, the tasks/objects are evacuated, the next step is to eliminate the residual processes while ensuring that the application continues to seamlessly run after *rescale* is complete. This step is non-trivial since the application processes are closely tied. To get rid of residual processes and ensure correctness, performance, and reliability after *rescale*, all the runtime system data structures which depend on $P_{old}$ need to be modified. Examples of these include spanning trees, task location managers such as hash-tables, and any processor-level instrumentation and monitoring modules.

To avoid the need of such complex modification of runtime structures, we follow a three-step process: 1) checkpoint application state before *rescale*, 2) enable application rebirth, and 3) restore application state from checkpoint after rebirth. The naive approach would be to use disk-based checkpoint-restart, where one would checkpoint the application state to disk, terminate application, re-launch application on new set of processors ($P_{new}$), and restore the data from checkpoint. However, interaction with disk can severely degrade *rescale* performance and prevent us from meeting the second design goal (Section 6.2.1). Ideally, for fast *rescale*, in-memory checkpoint could be used. However, since we terminate and re-launch processes to achieve clean restart of application on the continuing set of processors, any state stored in process memory will be lost.

Our novel solution to this challenge of performing fast, stateful, and scalable process rebirth is to use *Linux shared memory* – which has the advantage of being persistent across process restart, and also being fast. The idea is to 1) checkpoint the state before *rescale* to Linux shared memory, 2) perform application process *rebirth* or *death* depending on whether the processor is marked available or not, 3) execute a reconnect protocol – a modified version of the application start-up protocol, and 4) restore the application state from the checkpoints. The essence of step 2 is to replace the current process image by a a clean application image using Linux `exec` system call for the available processors whereas *death* for the unavailable processors using Linux `exit` system call. A reconnect protocol, which is a modified version of application launch protocol, is necessary to establish appropriate communication channels between the reborn application processes, which would allow them to communicate after *rescale* is complete.

Hence, the overall solution is to use the combination of task migration, load balancing, Linux shared-memory, and checkpoint-restart to achieve dynamic *shrink*. The task evacuation phase prior to checkpoint-restart ensures that the dying processes are stateless since all application state is already migrated to the processors which will continue to exist. Checkpoint-restart enables a clean state after *rescale* event, whereas the use of Linux shared memory allows the approach to be fast.

After the state restoration from checkpoints, runtime system has completed *shrink* and an acknowledgment can be sent to the external source from which the request originated. Also, control needs to be transferred from the runtime system to the application at a pre-registered application resumption point. Applications can perform such registration at initialization.

***Expand*:** For the case of *expand*, the basic idea is similar to the handling of *shrink* request. However, there are some important differences: 1) The *expand* request needs to specify the list of newly available processors. The bit-vector of existing processors required for *shrink* is not needed. 2) There is no need of object evacuation before checkpoint, instead load balancing is required after restoring from checkpoint. This post-restore load balancing distributes tasks to newly available processors to effectively utilize the total compute power. 3) No processes need to be killed, instead new processes need to be launched. These and the reborn processes then participate in the reconnect protocol to enable inter-process communication after restart completes.

We discussed our approach to *rescale* in the context of malleable jobs, where the decision to *rescale* is external e.g. job scheduler driven. However, our techniques will be equally useful in other contexts, such as *dynamic* or *evolving* jobs where the decision to resize is application-intrinsic, and other non-traditional use cases (Section 2.7). Also, our approach needs only task migratabiliy feature from the parallel runtime, and shared memory from the operating system. According to the November 2013 top500 list, 96.4% of top 500 supercomputers use operating systems of Linux family [1]. Hence, we believe our approach meets two of our design goals – it is *generic* and *practical*.

### 6.2.3 Implementation atop CHARM++

We implemented our techniques on the top of Charm++ [27] runtime system. In Charm++ the objects*(chares)* form the basic unit of computation and can be moved from one processor to another by the sophisticated load balancing framework provided by the runtime system [90]. These capabilities fulfill the needs of our approach. AMPI (Adaptive Message Passing Interface) is a framework written on top of Charm++ which provides dynamic load balancing capabilities to MPI applications using migratable user-level threads [103]. Using AMPI, MPI applications can benefit from the approach and implementation presented in this thesis.

In Charm++ programs, the application developer can specify synchronization points using `AtSync()` calls. which act as hints to the runtime to perform adaptive control such as dynamic load balancing. In our implementation for *rescale*, we service *rescale* requests

by invoking a custom load balancer, which is aware of the bit-vector information about unavailable processors. We perform intelligent load redistribution rather than random object evacuation to ensure that load is well balanced in the new set of processors after *rescale*. We developed our load balancer on top of CHARM++ load balancing framework, which instruments the objects execution times and process wall clock time from previous `AtSync()` point. These instrumented times of previous iterations are used as estimates of loads of future iterations, which is a proven estimation technique for iterative scientific applications [90]. We incorporated two existing load balancing strategies – *RefineLB* and *GreedyLB* in our implementation. RefineLB performs periodic load refinement by moving objects from overloaded to under-loaded processors, whereas GreedyLB uses a greedy strategy which assigns heaviest compute object to most under-loaded processor.

For checkpoint-restart, we checkpoint the current state of chares, collection of chares (chare arrays), and CHARM++ groups. In CHARM++, groups are processor-level agents which can be used to perform system tasks such as load balancing. Our implementation also handles stateless application-level groups but stateful application groups are not currently handled. Our checkpoint-restart implementation uses CHARM++'s pack-unpack (*pup*) serialization mechanism and Linux shared memory (`shm`) calls.

We perform the reconnect protocol using `Charmrun`, which acts as the start-up manager for CHARM++ applications. Our launch and reconnect protocol is a slightly modified version of the node-aware start-up discussed by Gupta et al. [105]. The primary modification is that on *rescale* event, `Charmrun` perform `ssh` to launch the executable only on the newly added nodes rather than all the nodes. The processes on rest of the nodes use `exec` system call. When the processes start, they connect back to `Charmrun`. `Charmrun` facilitates the exchange of communication information, such as data-port for Ethernet or queue-pairs information for Infiniband, necessary for enabling inter-processes communication after restart is complete.

After *rescale* is complete, control is transferred to the application using CHARM++'s callback mechanism.

## 6.3 Adaptivity in Resource Manager

In the previous section, we provided a novel approach to enable a runtime system to *shrink* or *expand* parallel programs. However, to realize the benefits of malleable parallel jobs in a shared cluster environment, the job schedulers and resource managers also need to be made adaptive (see Figure 6.2). Researchers have proposed several adaptive job schedul-

ing algorithms which have been demonstrated to provide significant benefits with malleable jobs [95, 98, 99]. However, the research challenges which arise in the 'management' of malleable jobs and 'execution' of job scheduling decisions in presence of malleable jobs have mostly remained open. Prior solutions include stalling while the job reconfigures, or executing *shrink*, *expand*, and launch simultaneously, in the presence of residual processes. The primary research issues that we address here are *how* and *when* to (a) communicate the scheduling decisions to running application and (b) detect the success or failure of those actions. In next subsections, we present a general framework and protocol for resource management to address these questions.

While performing the integration of the job scheduler, resource manager, and malleable parallel runtime system, we made some important design decisions: (1) the mechanism used by resource manager for executing *rescale* decisions should be orthogonal to job scheduling algorithm and (2) the interaction between resource manager, application, and the scheduling algorithm should be orthogonal to parallel runtime's *rescale* mechanism. There is one exception, where information communication among the three components of our system can help scheduler make better decisions. This information is the expected time taken by an application to perform *rescale* ($T_{rescale}$). The scheduling algorithm can then decide the gap between any two *rescale* events for same job ($T_{gap\_rescale}$), such that $T_{gap\_rescale} >> T_{rescale}$.

### 6.3.1 Resource Manager – Parallel Runtime Communication channel

To answer the *how* question of communicating between running application and the resource manager, we establish a control and feedback channel between those two components. To establish this channel, we leverage the Converse Client-Server interface (CCS), provided by Charm++ runtime system. CCS is built around a client-server model in which a CCS client connects to a CCS server via a TCP/IP socket and asks the server to execute pre-registered handler functions to fulfill requests. Upon receiving a request, the CCS server runtime invokes the appropriate handler, and thus injects messages into a running parallel computation.

Creating this channel between the resource manager and parallel runtime systems entails modifications on both ends. Since Charm++ is already designed to operate as a CCS server, the main effort necessary on the runtime side was to implement handler functions to service incoming requests for the desired functionality – *shrink* or *expand*.

To demonstrate a working system, we implemented a simple job scheduler and resource manager in Python. To inform an application of a *rescale* decision, the resource manager

starts a CCS client, connects to the CCS server using the hostname and server port corresponding to that job, and send a message through that connection. Next, it listens for a response back from the application. The communication from application to resource manager happens through the same channel. To acknowledge that it has resized itself in response to the notification, the application sends back a completion notification after performing *rescale*.

We used CCS since it is inbuilt in CHARM++, and shown to be a scalable method for interacting with the parallel application [106]. The communication mechanism in our resource manager is a pluggable module and it is easy to use another protocol, such as RPC, for this communication.

### 6.3.2 Split-phase Execution of Scheduling Decisions

For traditional rigid jobs, the only scheduling decision that needs to be implemented by the resource manager is to start a new job. At a scheduling event, the job scheduler may decide that $k$ jobs need to be launched (Algorithm 4, line 2). After the node scheduler allocates them the corresponding number of nodes and updates its database (line 3–4), the resource manager can launch those $k$ jobs simultaneously (`ExecuteDecisions` line 5).

In contrast, a resource manager for malleable jobs needs to handle three actions - launch, *shrink*, and *expand*. Having developed a mechanism for communicating between the running application and the resource manager, the next challenge is to decide *when* to execute the scheduling decisions. The challenge is that the $k$ decisions provided by the job scheduler may have inter-dependencies. For example, considering the example of Figure 6.1, when job $B$ completes, the scheduler decides to *shrink* job $A$ from 60 to 50 nodes and launch job $C$ on remaining 50 nodes of say 100 node cluster. These two decisions cannot be executed simultaneously since the nodes of C include those which are currently used by A and will be available only when A has finished shrinking. Similarly *expand* decisions on one job may also depend on *shrink* decisions of another. To tackle this problem, we perform split-phase execution of the scheduling decisions. A naive solution would be to first issue all *shrink* requests, wait till completion acknowledgements arrive from all of them, and then perform launch and *expand* actions. However, this results in unnecessary blocking and prevents other jobs from getting launched or scheduled. Hence, we optimize our split-phase approach to asynchronously send all the *shrink* requests (line 14–15 in procedure `ExecuteDecisions`). Next, we launch or *expand* jobs with no dependencies (line 16–17), and record jobs for which launch or *expand* needs to be delayed (line 20). Later, while the process is wait-

**Algorithm 4** *Shrink-Expand* Split-phase Execution

```
1: while true do
2:    jobDecisions = ScheduleJobs(jobQueue, clusterFreeNodes, runningJobs, T_rescale, optionalArgs)
3:    nodeDecisions = ScheduleNodes(jobDecisions,
      clusterNodeState, optionalArgs)
4:    UpdateSchedNodeMap(nodeDecisions)
5:    postponedActions = ExecuteDecisions(jobDecisions)
6:    repeat
7:       ProcessBufferedShrinkAcks()
8:       ExecutePostponedDecisions(postponedActions)
9:    until (jobQueue != empty or a job finished)
10: end while
```

```
11: procedure UpdateSchedNodeMap(decision)
    Update scheduler's view of node to job mapping
```

```
12: procedure ExecuteDecisions(jobDecision)
13: for decision in jobDecisions do
14:    if decision.type == shrink then
15:       NotifyJobToShrink(decision)
16:    else if AreAllNodesFree(decision.jobid) then
17:       LaunchExpandJob(decision)
18:       UpdateActualNodeMap(decision)
19:    else
20:       postPonedActions.Add(decision)
21:    end if
22: end for
23: return postPonedActions
```

```
24: procedure AreAllNodesFree(jobid) Check if all the nodes of a job are marked free in actual node
    to job map
```

```
25: procedure UpdateActualNodeMap(decision)
    Update actual node to job mapping
```

```
26: procedure ProcessBufferedShrinkAck()
    Update actual node to job mapping on shrink completion
```

```
27: procedure LaunchExpandJob(decision)
28: if decision.type == launch then
29:    LaunchJob(decision)
30: else if decision.type == expand then
31:    NotifyJobToExpand(decision)
32: end if
```

```
33: procedure ExecutePostponedDecisions (postponedActions)
34: for decision in postponedActions do
35:    if AreAllNodesFree(decision.jobid) then
36:       LaunchExpandJob(decision)
37:       UpdateActualNodeMap(decision)
38:       postponedActions.Remove(decision)
39:    end if
40: end for
```

ing for the next scheduling trigger, such as a new job arrival or completion of a running job, it periodically checks and processes any buffered *shrink* completion acknowledgements and updates appropriate data structure to reflect the actual node to job mapping (line 7 `ProcessBufferedShrinkAck`). After `ProcessBufferedShrinkAck` the decisions

for which execution was postponed due to dependencies are checked to see if they can be executed now based on the current state (line 8, procedure `ExecutePostponedDecisions`).

In our implementation, to track the dependencies between jobs, we kept two data structures – *SchedNodeToApp* which reflect the scheduler's view of nodes to jobs mapping, and *ActualNodeToApp*, which tracks the actual state. The scheduler's view gets updated at scheduling event (line 4) or when a job completes (line 9) whereas the actual state is updated on *shrink* completion acknowledgement(line 7, 26), new job launch (line 18, 37), and issue of *expand* request(line 18, 37).

## 6.4 Evaluation Methodology

To analyze the performance and scalability of our approach to malleability, and to evaluate it against the design goals, we used following benchmarks and applications.

- **Stencil2D** is a 5-point stencil kernel which iteratively averages values in a 2-D grid using Jacobi relaxation. This benchmark represents a widely used kernel in HPC applications.
- **Wave2D** is a 2-D mesh based mini-application for simulating wave propagation. It is computation intensive and uses discretized finite differencing method.
- **LeanMD** is a molecular dynamics mini-application which performs simplified version of the force calculations of NAMD [40], a widely used molecular dynamics code. LeanMD uses two Charm++ object arrays – *cells*, which are collection of atoms in 3-D space, and *computes*, which perform force calculation on atoms.
- **Lulesh** is the Charm++ implementation of LULESH hydrodynamics mini-application [107]. It simulates explicit shock hydrodynamics in 3-D space using Lagrangian formulation with leap frog time integration.

We conducted experiments on Stampede supercomputer. Stampede has Dell PowerEdge server nodes, which have 2 Intel Xeon E5 processors each accounting for 16 cores per node. Each node has 32GB memory, and allows up to 16GB memory to be used for Linux shared memory. We used interactive job allocation on Stampede, which allowed us to send CCS requests to running applications from external client and demonstrate our malleable runtime system integrated with our own adaptive resource manager.

For most experiment, we used Charm++ `net-linux-x86_64-ibverbs` build that uses low-level Infiniband Verbs communication library. Charm++ also provides an SMP build for multi-core machines, which combines multiple worker threads and one communica-

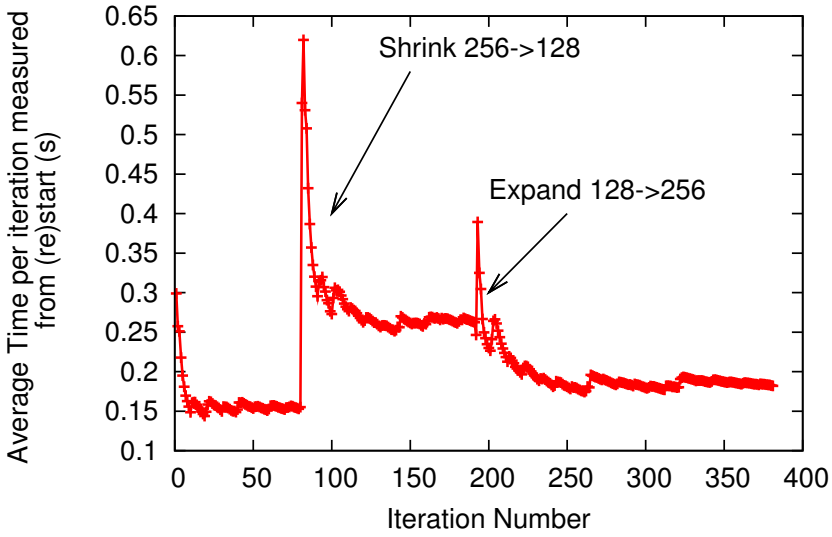



**Figure 6.4: Adapting load distribution on rescale (LeanMD)**

tion thread into a single CHARM++ process. However, interactive allocation on Stampede has a limit of 16 nodes (256 cores). Using SMP version would limit our experiments to even lower scale since we will have very few total processes. Hence, we use the non-SMP version to evaluate our approach till 256 processes. We used an optimization level $-O3$, and show results with *RefineLB* load balancer unless specified otherwise.

## 6.5 Results

We modified the applications presented in Section 6.4 to make them re-sizeable. Next, we analyze the effectiveness, performance, overhead, and benefits of our system.

### 6.5.1 Adapting Load Distribution on Rescale

We use an external CCS client to send the *rescale* request to the application. Figure 6.4 illustrates LeanMD's response to *rescale* requests and corresponding change in achieved performance on 256 processors on Stampede. Figure 6.4 shows the variation in average time per iteration, measured from a *rescale* event completion, with respect to the iteration number. When the *shrink* request is handled at iteration 80, the number of processors are reduced to half, that is 128. The average iteration time doubles since the average load on each processors doubles. The peaks in the graph reflect the time taken by load balancing,

which is performed every 20 iterations. On *expand* (iteration 200), the number of processors doubles. For *expand*, the load redistribution happens at the next load balancing step after restart. Hence, the average iteration time drastically reduces at step 220. After a few load balancing steps, we achieve good convergence in iteration time. Hence, our system is effective in adapting to the change in compute power caused by *rescale* events, meeting the first design goal (Section 6.2.1).

### 6.5.2 Shrink Expand Overhead

To quantify the overhead of application reconfiguration on a *rescale* event, we measured the breakup of time spent in different phases. We shrank various applications from 256 to 128 processors and expanded back to 256 processors. For these experiments we used the following configurations: Stencil2D with $12k \times 12k$ grid with block (or object) size of $2k \times 2k$, LeanMD on a $4 \times 4 \times 4$ cell with 2432 computes, Lulesh with a $512 \times 256 \times 320$ grid, and Wave2D with $64k \times 48k$ data on a $32 \times 24$ object grid.

***Shrink* vs. *Expand*:** Table 6.2 shows the the time taken in different stages of our scheme. The total time required is 2.6 - 4s for *shrink* and 7.1 - 8.7s for *expand* for different applications (except last row). The reason for the difference in the time between *shrink* and *expand* is evident from the breakup, which shows that reconnect time is the dominating factor. Reconnect phase includes a) the time taken by the launcher to ssh and launch new processes, which is done only in case of *expand*, and b) the time taken by the connection establishment phase. For *expand*, launcher needs to start 128 new processes. Also, the connection establishment happens for 256 processes compared to 128 after *shrink*. Hence, the reconnect time is more for *expand* compared to *shrink*.

The overhead breakup actually enabled us to optimize the *rescale* time from around $9s$ to $2.5s$ for *shrink* and from $19s$ to $7s$ for *expand*. Knowing that reconnect phase is the bottleneck, we optimized it by using a combination of startup techniques such as batching and node-awareness [105]. It is possible to further improve this by using variations of advance startup mechanism such as multi-level startup [105]. Rest of the phases – load balancing (LB), checkpoint, and restore are very fast, for both *shrink* and *expand*. Considering that *rescale* events are expected to be infrequent – every few tens or more minutes, our approach has very low overhead. Hence it meets our second design goal.

**Effect of network:** In addition, we evaluate the performance using Ethernet as the network option on Stampede. The motivation is to evaluate the applicability of our scheme on commodity cluster and clouds since Ethernet is the network commonly available there.

**Table 6.2: *Shrink Expand* time breakup (in seconds) for different applications**

| Application | *Shrink*: 256 → 128 | | | | | *Expand*: 128 → 256 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | LB | Checkpoint | Reconnect | Restore | Total | LB (Post) | Checkpoint | Reconnect | Restore | Total |
| LeanMD | 0.515 | 0.039 | 2.102 | 0.003 | 2.658 | 0.056 | 0.016 | 7.079 | 0.003 | 7.154 |
| Lulesh | 0.560 | 0.531 | 2.533 | 0.432 | 4.056 | 0.458 | 0.520 | 7.083 | 0.436 | 8.496 |
| Wave2D | 1.219 | 0.243 | 2.542 | 0.336 | 4.340 | 1.046 | 0.244 | 7.067 | 0.337 | 8.695 |
| Stencil2D | 0.299 | 0.050 | 2.501 | 0.054 | 2.904 | 0.133 | 0.036 | 7.076 | 0.038 | 7.283 |
| Stencil2D_Net | 5.86 | 0.057 | 2.584 | 0.056 | 8.556 | 4.096 | 0.042 | 9.495 | 0.044 | 13.678 |

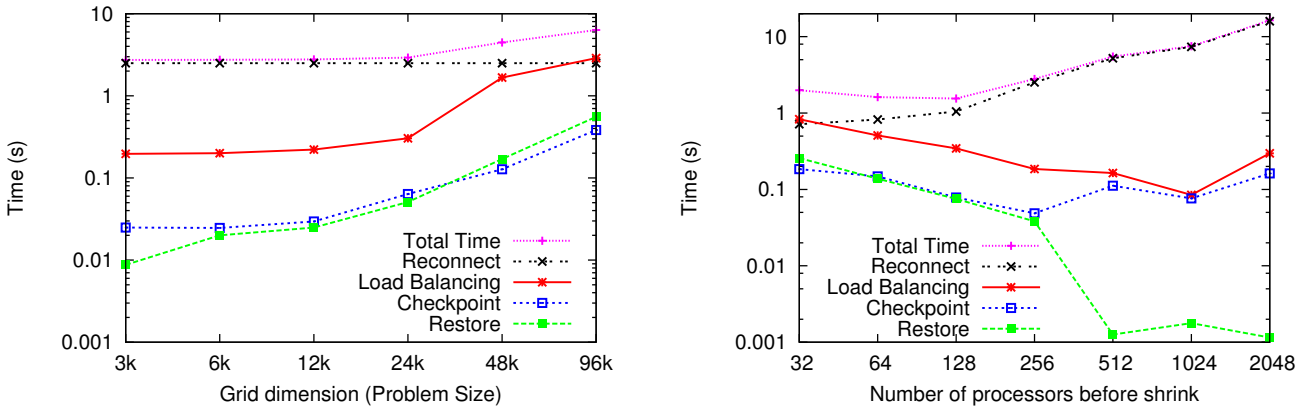



**Figure 6.5: Analysis of *rescale* performance using Stencil2D. (a) Effect of problem size, *shrink* from 256 to 128 cores and (b) Effect of strong scaling with $24k \times 24k$ problem size, shrinking to half**

The last row in Table 6.2 shows that even with Ethernet, our scheme performs *rescale* in a reasonable time. Comparing the results of Infiniband and Ethernet (last two rows in Table 6.2), it is clear that primarily load balancing (which involves object migrations), and reconnect phases are the ones which suffer because of a worse network. The only communication needed for checkpointing and restore phases is for some synchronizations, hence their performance is not much affected.

### 6.5.3 Scalability Analysis using Stencil2D

Having shown that our approach is effective and fast, we next analyze the scalability of our technique with respect to problem size and increasing node counts.

#### Effect of Problem Size

Figure 6.5a shows the effect of changing the problem size of Stencil2D, shrinking from 256 to 128 processors. As the problem size grows, load balancing, checkpoint, and restore times increase. This can be attributed to increased data per process. The checkpoint size is around 10MB per process for 12k size and 640 MB per process for 96k grid dimension. As the grid dimension doubles, checkpoint size increases by 4X, resulting in slowdown of checkpoint and restore phases. It is evident from Figure 6.5a that our shared memory based approach

works well since for even 640 MB checkpoints size, only 0.5s is needed. The load balancing time also increases with problem size since more data needs to be migrated with increasing problem size. The time of reconnect phase remains constant since it is independent of the problem size.

**Effect of Strong Scaling**

Next, we analyze the scalability of our approach with respect to increasing number of processors, but constant problem size (24k for Stencil2D). Since the allocation in interactive mode is limited to 256 processors on Stampede, for larger scale experiments we modified the application to initiate a *rescale* request itself (akin to evolving jobs).

Figure 6.5b shows that with increasing scale, the time for reconnect phase slowly increases and dominates the total *rescale* time at large scale. The checkpoint, restore, and load balancing phase scale very well till a particular point – 256 for checkpoint, 1k for load balancing. With increasing processor count, the per-process memory footprint proportionally decreases. This results in reduced checkpoint size and less communication per processor. However, as we scale further, the barrier synchronizations present in these phases become notable, resulting in increase in time. Overall though, the time is still small (16s for $2k \rightarrow 1k$ shrink). The corresponding time for expand, i.e, $1k \rightarrow 2k$ is 40s. Figures 6.5 showed that our approach performs reasonably well with both – increasing memory per process and increasing core counts, hence meeting our third design goal.

### 6.5.4 Programmer Effort

To quantify the programmer effort needed to make applications malleable using our runtime, we measured the original and the modified source lines of code (SLOC) for our benchmarks and applications, using `sloccount`. Table 6.3 shows that we needed to modify very few SLOC to make these codes malleable, meeting our last design goal (Section 6.2.1). For Lulesh, which is the largest of the mini-applications, we needed to modify only 15 SLOC, which was very little effort (<0.4% of original SLOC). The primary modifications required were to register the resume callback with the runtime and make the `mainChare` (main or entry point object in Charm++) as a migratable entity by providing its migration constructor and pack unpack routine.

**Table 6.3: Application-specific Development Effort**

| Application | Original SLOC | Modified SLOC |
|---|---|---|
| Stencil2D | 207 | 31 |
| LeanMD | 703 | 37 |
| LuLesh | 4066 | 15 |
| Wave2D | 363 | 37 |

### 6.5.5 Case Study with Adaptive Scheduler

To demonstrate that our approach towards integrating the resource manager and parallel runtime works in practice, we conducted a case study with an adaptive job scheduling algorithm. We implemented a variant of dynamic equipartitioning strategy which has previously been shown to have significant performance gains [96,108]. The strategy first assign each job its minimum required nodes on a first come first serve basis. After that, if each job received its minimum needs and more nodes remain available, they are equally distributed to the scheduled jobs. We modified this policy to consider $T_{gap\_rescale}$, that is the gap between two scheduling actions (launch, *shrink*, or *expand*) on same job. Only those jobs are considered for scheduling for which at least $T_{gap\_rescale}$ seconds have elapsed since they were launched or the since they were shrunk or expanded.

We used this job scheduler in conjunction with our resource manager and the malleable runtime. For the purpose of this case study, we consider the interactive node allocation on Stampede as our small cluster. Five jobs were submitted to the scheduler with arrival times of 0, 1, 3, 7, and 7 minutes from the start time respectively. For simplicity all the five jobs run same application (Stencil2D with 10000 iterations each). The range for all the applications to *shrink* and *expand* is from 4 to 16 nodes, with 16 cores per node. $T_{gap\_rescale}$ was set to 40s for this experiment. Figure 6.6-a shows the nodes to job mapping over time, hence depicting overall cluster utilization and how these jobs are *rescaled*. For example, when job 2 arrives, job 1 is shrunk from 16 to 8 nodes to give the rest to job 2. This is reflected by the change in the nodes to job mapping at time=100s. Similarly, towards the end, when job 4 finished, job 5 is expanded from 4 to 16 nodes at time=1700s.

Figure 6.6-b shows how the nodes are allocated when the same jobs are run but they are rigid. Here, we used a FCFS (First Come First Serve) scheduling policy. Also, we assigned number generated in the range [4-16] using a random number generator as the required number of nodes for a job. Figure 6.6 clearly illustrates the improvement in system utilization and total completion time using malleable jobs, compared to rigid jobs.

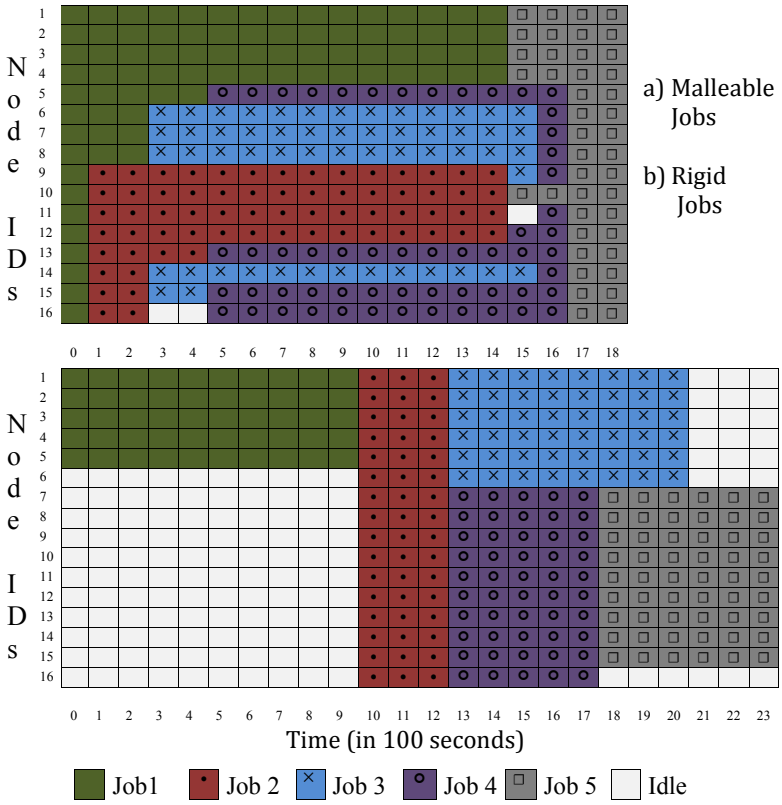



**Figure 6.6: Nodes allocated over time**

Table 6.4 compares the achieved performance with different job types and different values (40s, 100s, and 500s) of $T_{gap\_rescale}$ for malleable jobs. To emulate moldable jobs, we set a very large value of $T_{gap\_rescale}$, which eliminates any *rescale* events. The results for rigid case are the average of three runs with different random assignments for number of nodes to jobs.

The data of Table 6.4 is normalized and visualized using a spider chart (Figure 6.7). Here, the four dimensions represents our comparison metrics, with smaller being better. For utilization, we plotted the inverted values to get a consistent visualization. In Figure 6.7, the solid green quadrilateral which corresponds to rigid jobs is the worst since it perform poorly on all the dimensions. Figure 6.7 also shows that for this case study, most benefits of malleability are obtained in terms of mean response time, followed by utilization, total completion time, and mean completion time. Moreover, $T_{gap\_rescale}$ can have significant impact on achieved benefits. If $T_{gap\_rescale}$ is very small, there can be very frequent *rescale* events, leading to high performance overhead, which can increase mean completion time. If $T_{gap\_rescale}$ is very high, the system will not benefit much as there will be very few *rescale* events. For our case study, all the three values $T_{gap\_rescale}$ yielded benefits but the optimal value depends on the metric of interest. This can also be seen in Figure 6.7 by observing

**Table 6.4: Comparison of different scheduling policies**

| Scheduling Type | Total Time (s) | Mean Response | Mean Completion | Util--zation |
|---|---|---|---|---|
| Malleable40 | 2751 | 201 | 1767 | 97% |
| Malleable100 | 2672 | 142 | 1699 | 98% |
| Malleable500 | 2844 | 287 | 1454 | 97% |
| Moldable | 3792 | 289 | 1685 | 62% |
| Rigid | 3816 | 928 | 1817 | 70% |

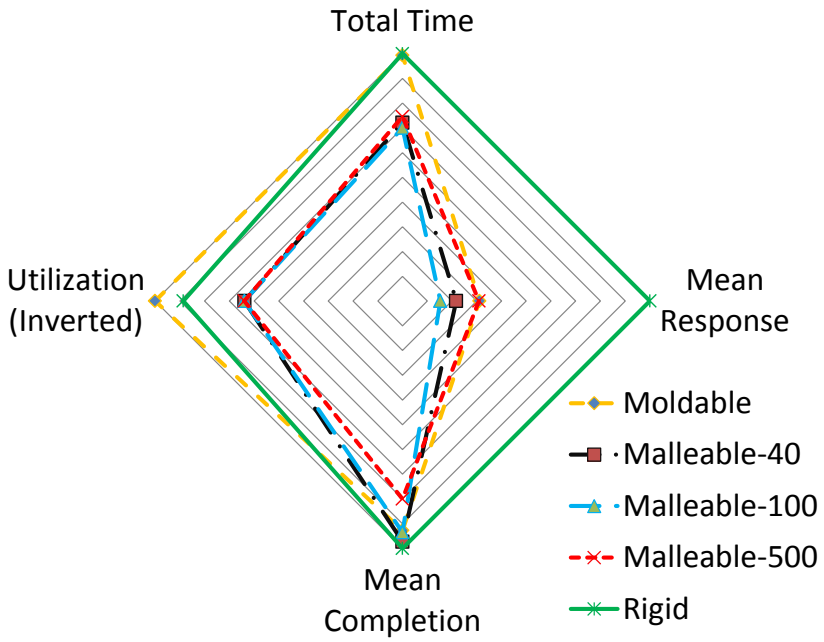



**Figure 6.7: Scheduling comparison along different metrics**

the malleable-40 and malleable-100 shapes along mean response and mean completion time dimensions.

In practice, we expect a good value of $T_{gap\_rescale}$ to be few tens of minutes. In this work, we ran short-duration jobs since our primary intention is to demonstrate the working system on a real supercomputer rather than quantifying the exact benefits of malleable job schedulers. Moreover, we were constrained by the the time limit on an interactive mode allocation on Stampede. The exact benefits of malleability depend on various factors, such as job arrival rate, job run times, range within which a job is malleable, and the scheduling algorithm. Thorough analysis and impact of these factors can be found in related work [94–97, 100].

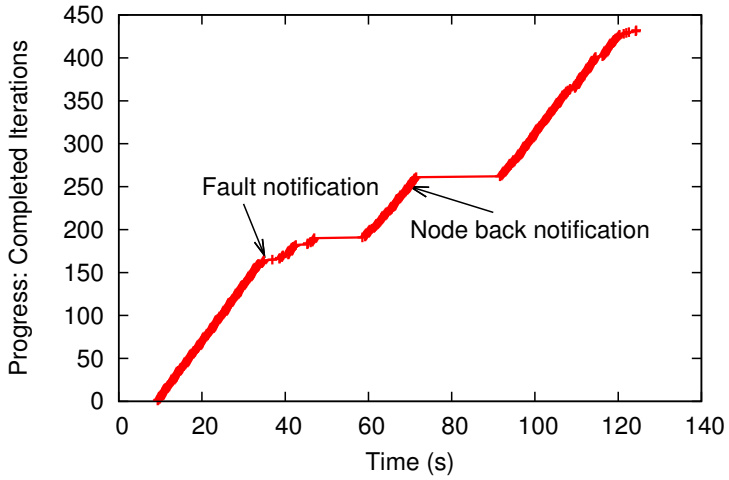



**Figure 6.8: Proactive fault tolerance using malleable runtime system and resource manager to application communication channel**

## 6.6 Non-traditional Use cases

In previous sections, we designed a malleable parallel runtime and demonstrated its integration and utility in conjunction with an adaptive resource manager and a job scheduler. The ability of a parallel runtime to *rescale* can be applied to other contexts. Here, we show two such emerging use cases.

### 6.6.1 Reliability: Proactive Fault Tolerance

High performance parallel systems with millions of cores are currently being used and even bigger systems are being planned as we move towards the exascale era. One of the biggest challenges for operating under such massive scale is to achieve fault-tolerant application execution since failures become more and more frequent as the number of system components increase [93]. The traditional solution to deal with failures is to *react* to failures by using mechanism, such as checkpoint-restart. A less explored approach is to predict failures, and *proactively* vacate the processor where fault is imminent [109]. Failure prediction can be done using hardware devices supporting early indication of failures, sensors, monitoring core temperatures, and other techniques. Inspired by the work of Chakravorty et al. [109], we demonstrate how we can leverage the ability of our runtime system to *shrink* and *expand*, and the bi-directional communication channel between the resource manager and the parallel runtime to enable proactive fault-tolerance.

Once the cluster resource manager predicts that a failure is imminent on a node, it can inform the application by sending a CCS request containing information regarding the failing node. Figure 6.8 demonstrates how an application (here LeanMD), initially running on 16 nodes (256 processors) on Stampede, reacts to the information communicated by the resource manager. In Figure 6.8, first few seconds are taken by the job to start-up, hence no application progress is made during that time. At the next synchronization point after receiving the fault notification, the application re-configures itself using *shrink* and continues running on remaining 15 nodes. Once, the node is up again, the application can be informed. On this notification, the parallel runtime expands the job to use 16 nodes again.

Our *rescale* mechanism provides us with rich proactive fault tolerance capability. We can tolerate failures at the level of a node, which could translate into $k$ application processes (e.g. $k = 16$ on Stampede) rather than a single application process. Most current fault-tolerance mechanism tolerate a single process failure. Furthermore, most current fault-tolerance mechanisms make an inherent assumption that either the failing node will be available instantaneously after failure or not at all, or a spare node will be available to replace the failing node. However, they do not allow the possibility of reusing a node which comes back into operation sometime after it failed, e.g. it may just need a restart. Our scheme allows to reuse that node at a later time using *expand*. Finally, proactive fault-tolerance gives an additional advantage by eliminating any execution rollback, such as restart from previous checkpoint in traditional fault tolerance schemes, when the failure actually occurs.

### 6.6.2 Cloud User Perspective: Elasticity

Amazon EC2 [9] has emerged as the leader in providing such infrastructure-as-a-service (IaaS). For HPC, Amazon offers the cluster compute instance (CC) [15]. There are three kinds of CC instances currently offered by Amazon EC2 – reserved, on-demand, and spot. Reserved instances require long reservations at a lower price and are only suitable when a user has long-term static demands. On-demand instances allow the user more flexibility but at a higher price. Spot instances offer the unused cloud capacity at a dynamically varying price, typically very small compared to on-demand instance price. Spot instances work on a bidding based model. A users places her bid for compute power. If and when her bid exceeds the current spot price, instances are allocated to the user. When the spot price exceeds the bid, the instances are terminated without notice. The user is charged with the spot price at the start of each instance-hour. The spot price changes periodically based on supply and demand. Figure 6.9 shows the pricing variation within a day (Jan 7, 2013) of the spot price

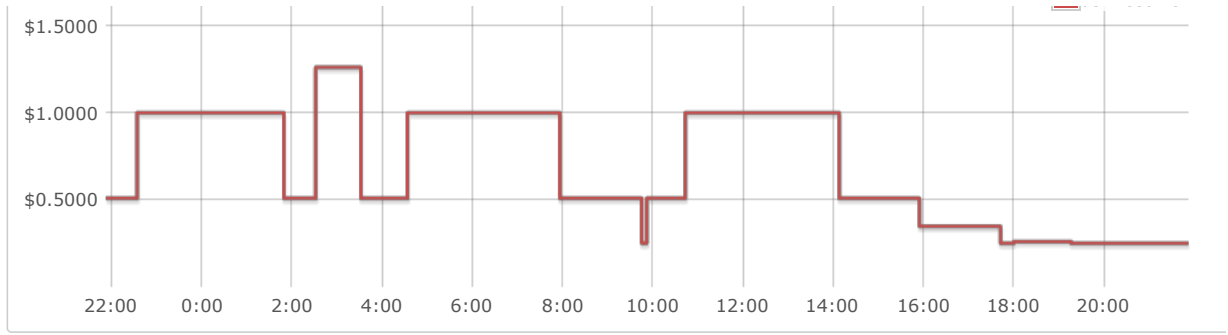



**Figure 6.9: Amazon EC2 spot price variation for cc2.8xlarge instance (zone us-west2c) on Jan 7, 2014**

of Amazon cc2.8.xlarge instance (zone us-west-2c). This data was obtained from the pricing history available from Amazon EC2 management console.

The malleability support in an HPC runtime can be used to exploit this dynamic spot pricing to achieve cost benefits. Our main idea is to 1) keep a certain minimum number of instances needed for running a job in the on-demand instance pool (static set) and 2) perform *price-sensitive rescale* over the spot instance pool to add more compute power (dynamic set). By price-sensitive *rescale* we mean performing *expand* when the spot price falls below a threshold, and performing *shrink* when it exceeds the threshold. By specifying the same availability zone and placement group when requesting the static on-demand and the dynamic spot instances, it can be possible to get them in the same physical cluster [15]. If that does not happen, one can construct the static set also from the spot instance by placing a high bid for that set.

Running without *rescale* capability necessitates setting the bid at a very high price and paying whatever the spot price at the start of the instance-hour is. Using the data of Figure 6.9, that would entail setting the bid price greater than $1.25, which results in a cost of $16.65 over a period of 24 hours for a spot-instance. This results in an effective price of $0.69 per instance-hour. In contrast, the combination of malleable parallel runtime and price-sensitive rescaling enables a user to choose the pricing point below which she wants to operate. As an example, setting a price threshold of $0.5 results in operating costs of $4.9 for 12 compute hours (since the instances will be used only for the hours where the price at the start of the hour is less than $0.5), resulting in average price of $0.41 per instance-hour. Overall, that results in around 40% better effective price for the same instances compared to the default usage. However, there is a trade-off between effective price attained using *rescale* and usable hours as reflected in Figure 6.10, which shows that one can achieve a lower effective price using *rescale* but the usable compute hours will be reduced. In most

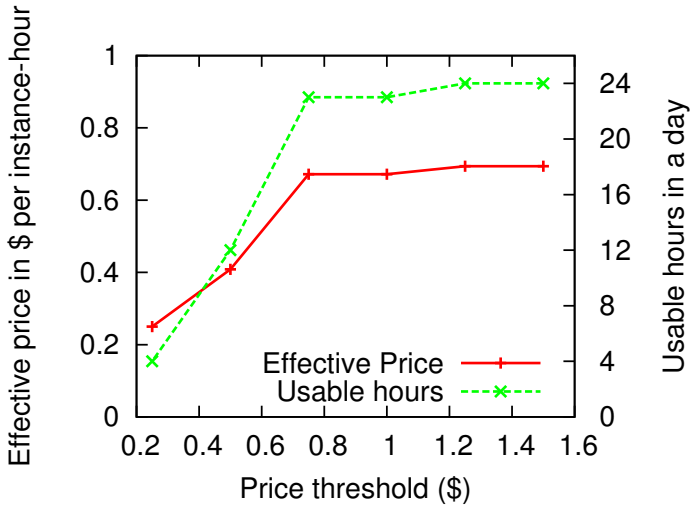



**Figure 6.10: Potential benefits of price-sensitive rescaling of spot instances on Amazon EC2, and the trade-off between effective price achieved and usable hours**

cases, this problem can be circumvented by either running more instances at this lower price or operating for longer periods.

## 6.7 Conclusions

We presented a novel technique to enable malleability in a parallel runtime system using task migration, load-balancing, checkpoint-restart, and Linux shared memory. We implemented this approach using Charm++ runtime and performed resize on one benchmark and three mini-applications. Through experimental evaluation and analysis on Stampede up to 2048 cores, we demonstrated that our approach is fast, scalable, and effective. In addition, we integrated our malleable runtime system with a resource manager and demonstrated split-phase execution of job scheduling decisions through a bi-directional communication channel between application and the resource manager. Although our focus in this work was on scheduler-triggered *shrink* or *expand*, the techniques developed here are also useful for evolving jobs and other emerging use cases such as proactive fault tolerance and HPC in cloud.

CHAPTER 7

# Concluding Remarks

This thesis makes one of the first software attempts at realizing the true potential of cloud computing towards spreading the outreach of HPC. Cloud computing with Infrastructure-as-a-Service offerings has recently emerged as a promising addition/alternative to traditional supercomputers. Clouds are especially attractive to small and medium scale organizations, especially those with emerging or sporadic HPC demands, since they can benefit from the pay-as-you-go model (renting vs. buying) and elasticity – on-demand provisioning in clouds. However, despite this potential, there is a mismatch between current cloud environments and typical HPC requirements as shown by the comprehensive performance and economic analysis performed in Chapter 2. This thesis presents techniques to bridge that gap by providing techniques for effective and efficient HPC in cloud.

Our philosophy for bridging the divide between HPC and clouds is to use a complementary approach of making clouds HPC-aware and HPC cloud-aware. In addition, instead of solely focusing addressing the challenges of running HPC applications in cloud, the thesis also explores how the unique opportunities offered by cloud can be exploited by HPC. Finally, we believe that it is important to consider views of both, HPC users and cloud providers, who sometimes have conflicting objectives: users must see tangible benefits (in cost or performance) while cloud providers must be able to run a profitable business. With this philosophy, our techniques attempt at improving HPC performance, resource utilization, and cost when running in cloud.

We summarize the insights, lessons and conclusions of this thesis in Section 7.1. Next, we present the contributions of this thesis in Section 7.2. Finally, we end this chapter with closing remarks.

## 7.1 Conclusions

The conclusions of this thesis are:

**Performance-cost evaluation and models for HPC in cloud (Chapter 2 and Chapter 3)**

- **Performance trends:** Different applications exhibit different characteristics that make them more or less suitable to run in a cloud environment. Applications with non-intensive communication patterns are good candidates for cloud deployments. For communication-intensive applications, supercomputers remain the optimal platform, largely due to the overhead of network virtualization in the cloud. The poor network performance in clouds is attributed to the absence of an HPC-optimized interconnect and the virtualization overhead.

- **Performance bottlenecks:** Besides the poor network performance; heterogeneity, multi-tenancy, performance variability, and noise or *jitter* are dominant challenges for HPC in cloud. To address these challenges, HPC need to be cloud-aware and clouds needs to be HPC-aware. For instance, lightweight virtualization is necessary to remove overheads for HPC in cloud.

- **Cost:** Even though the performance in cloud is suboptimal, clouds can be cost-effective compared to supercomputers for some HPC applications. There are interesting performance-cost tradeoffs when running in cloud vs. a supercomputer.

- **Cloud as substitute or complement to supercomputers:** Clouds can successfully complement supercomputers, but using clouds to substitute supercomputers for all applications and all scale is infeasible. Bursting to cloud is also promising. By performing multi-platform dynamic application-aware scheduling, a hybrid cloud-supercomputer platform environment can actually outperform its individual constituents.

- **Application characterization:** Application characterization and performance prediction for complex HPC applications across multiple platforms is a non-trivial task, but the economic benefits are substantial.

**HPC-aware cloud scheduler (Chapter 4)**

- **VM consolidation:** Although it may be counterintuitive, HPC can benefit greatly by consolidating VMs using smart co-locations. There exists a trade-off between an HPC application's performance and cloud's resource utilization. Surprisingly, it is possible

to achieve a win-win solution. 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.

- **HPC-awareness:** 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. Experimental evaluation shows that introducing HPC-awareness in cloud scheduler results in significant benefits.

**Cloud-aware HPC runtime (Chapter 5 and Chapter 6)**

- **Heterogeneity and multi-tenancy:** Besides the static heterogeneity, multi-tenancy in cloud introduces dynamic heterogeneity, which is random and unpredictable. The overall effect being poor performance of tightly-coupled iterative HPC applications.

- **Adaptive HPC runtime:** Dynamic environments in cloud necessitate the need of adaptivity in the HPC runtime system for achieving good performance. Even without the accurate information of the nature and amount of heterogeneity (static and dynamic but hidden from user as an artifact of virtualization), the approach of periodically measuring idle time and migrating load away from time-shared VMs works well in practice.

- **Elasticity:** It is possible to adapt HPC runtime to exploit the unique opportunity of elasticity offered by cloud. We presented a technique to perform fast, scalable, and effective shrinking and expanding of parallel jobs.

## 7.2 Contributions

The main contributions of this thesis are summarized in Figure 7.1 and follow below:

**Performance-cost evaluation and models for HPC in cloud (Chapter 2 and Chapter 3)**

- **HPC in Cloud Performance and Cost Analysis:** In Chapter 2, we evaluate the performance of HPC applications on a range of platforms varying from supercomputer to cloud. Also, we analyze bottlenecks and the correlation between application characteristics and observed performance, identifying what applications are suitable for cloud. We also investigate the economic aspects of running in cloud and discuss why it

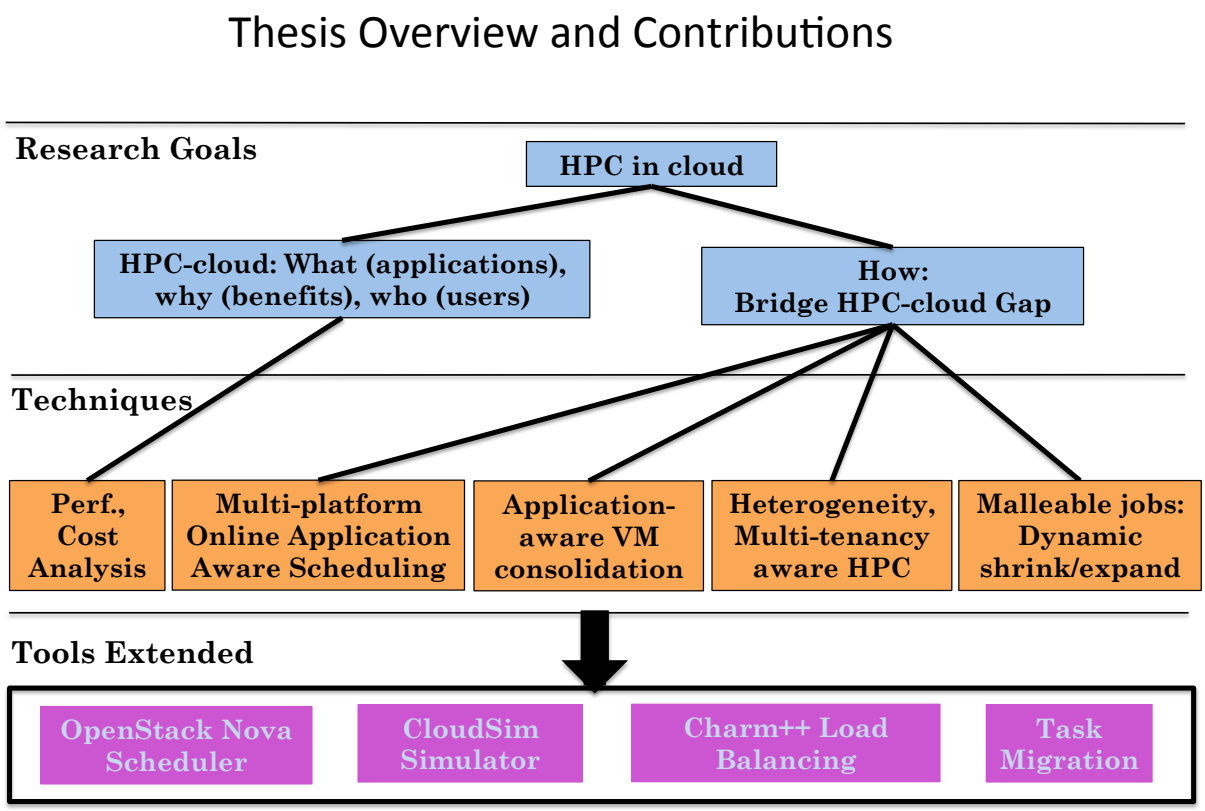



**Figure 7.1: Thesis overview: research goals, contributions (techniques and tools extended)**

is challenging or rewarding for cloud providers to operate business for HPC compared to traditional cloud applications. We also show that small/medium-scale users are the likely candidates who can benefit from an HPC-cloud.

- **HPC-aware Cloud virtualization and Cloud-Aware HPC Tuning:** To address the challenge of slow network performance in clouds, Chapter 2 presents the complementary approach of (1) making HPC applications cloud-aware by optimizing an application's computational granularity and problem size for cloud and (2) making clouds HPC-aware using thin hypervisors, OS-level containers, and hypervisor- and application-level CPU affinity.

- **Multi-platform Online Application-Aware Job Mapping:** Instead of considering cloud as a substitute of supercomputer, we investigate the co-existence of multiple platforms – supercomputer, cluster, and cloud. In Chapter 3 we research, analyze,

and evaluate novel heuristics for application-aware mapping of jobs in this multi-platform scenario significantly improving average job turnaround time (up to $2X$) and job throughput (up to $6X$), compared to running all jobs on supercomputer.

**HPC-aware cloud scheduler (Chapter 4)**

- **Intelligent HPC-aware VM placement:** 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.

- **Application-Aware VM Consolidation and Scheduling:** 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. We present the techniques, implementation, and evaluation of the proposed application-aware VM scheduling algorithm in OpenStack Nova scheduler. We also modify CloudSim to make it suitable for simulation of HPC in cloud and present simulation results.

**Cloud-aware HPC runtime (Chapter 5 and Chapter 6)**

- **Heterogeneity and Multi-tenancy-aware HPC runtime and Dynamic Load Balancing:** We present dynamic load balancing techniques for efficient execution of tightly-coupled iterative HPC applications in heterogeneous, multi-tenancy, and dynamic cloud environment. The main idea is periodic refinement of task distribution using measured CPU loads, task loads, and idle times. Chapter 5 presents the techniques, implementation in Charm++, and evaluation of performance and scalability on a real cloud setup on Open Cirrus testbed.

- **Malleable Parallel Runtime:** In Chapter 6, we present a novel technique for providing a fast, efficient, and scalable shrink/expand capability to a parallel runtime system. Salient features of our scheme are task migration, checkpoint-restart, load balancing, and use of Linux shared memory. We also developed a technique for enabling split-phase execution of malleable job scheduling decisions in a shared cluster, incorporating scheduler-runtime communication.

## 7.3 Closing Statement

In this thesis, we have taken the position that HPC in current clouds is suitable for emphsome applications *not all*. We have also stated that the gap between HPC and current clouds can be bridged by intelligent applications to platforms *mapping*, HPC-aware cloud *scheduling*, and cloud-aware HPC *runtimes*. We believe that our research will be extremely helpful in realizing the true potential of clouds for HPC.

CHAPTER 8

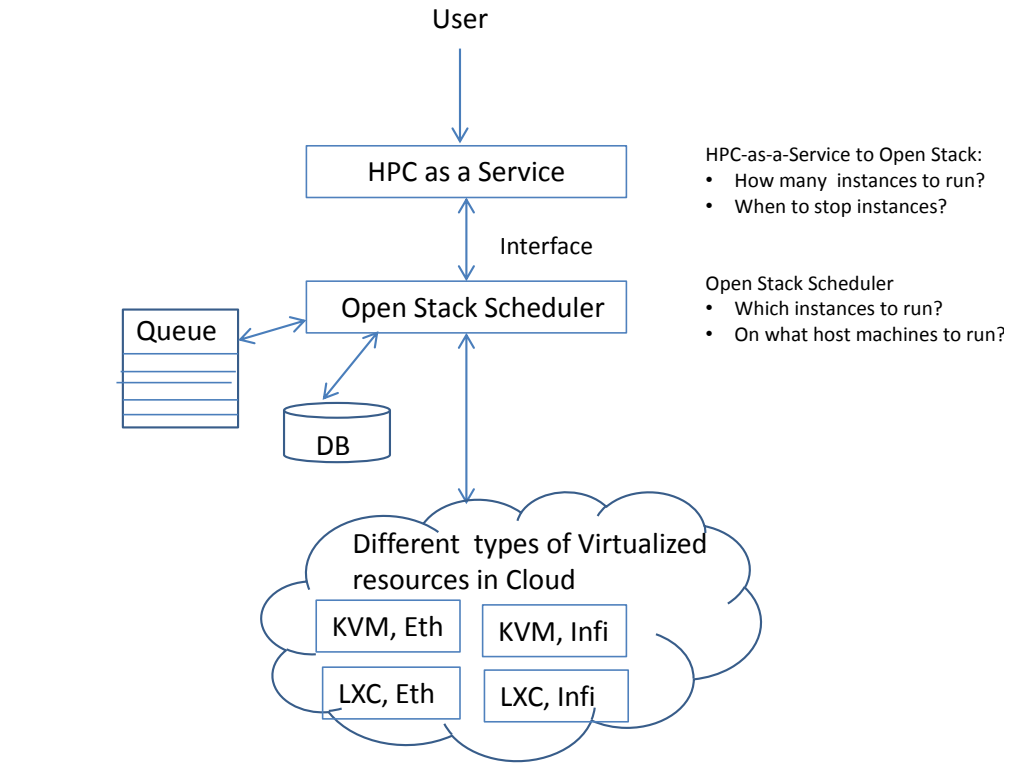
# Future Work

In this chapter, we present some promising future directions derived from the contributions in this thesis.

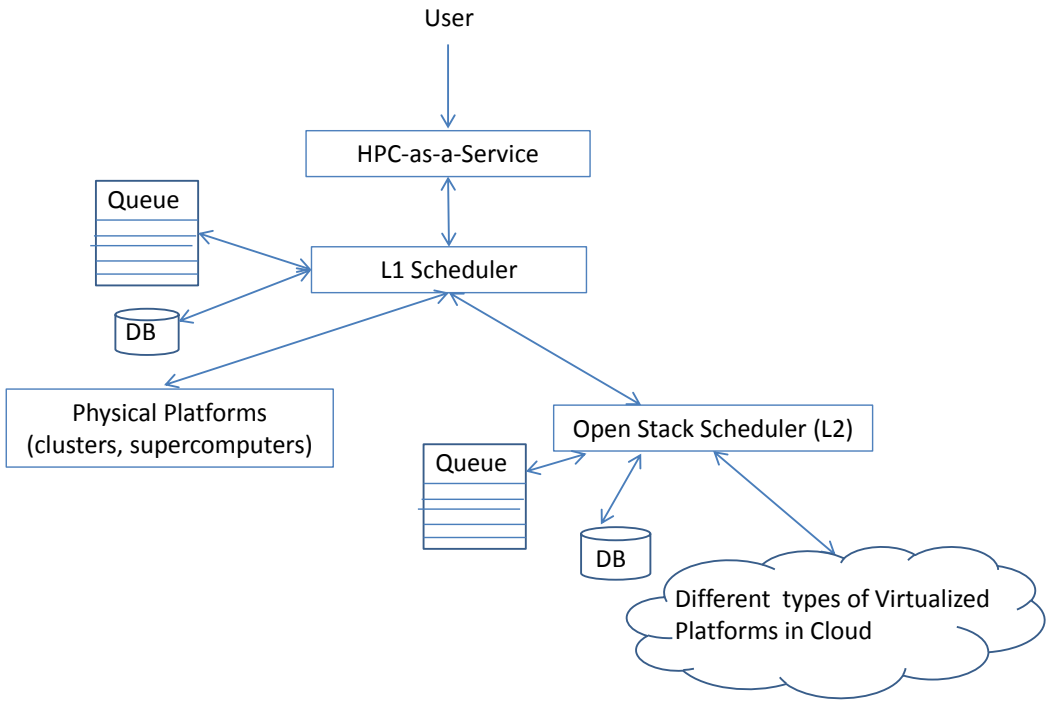
## 8.1 Application Characterization and Models for HPC in Cloud

**Application Characterization:** In Chapter 2, we showed the benefits of selecting a suitable platform from a set for a given application. We considered proof-of-concept applications where it is relatively easy to extract the application signature. We did not consider memory-bound and I/O-bound applications. To cover the full complexity of real world HPC scenarios additional techniques for more accurate results and complex applications need to be explored. This would require further research on identifying characteristics crucial for the purpose of application to platform mapping for applications with complex and irregular communication patterns, especially those with collective communication, involving overlap of computation and communication, and dynamic load balancing.

**HPC-as-a-Service and Multi-level Cloud Scheduling System:** In the IaaS model, an HPC user typically runs all her applications on the same platform. In Chapter 3, we presented intelligent algorithmic heuristics for scheduling of applications to platforms in cloud, and showed the significant benefits using simulation. With this observation, a future direction is to design a system providing HPC-as-a-Service atop infrastructure. In this model, user does not have to worry about the selection of a suitable platform for her application set. Different applications can be mapped by the middle-ware to different platforms based on platform and application characteristics. However, cur-




(a) Multi-platform scheduling system




(b) 2-level scheduling system

**Figure 8.1: HPC-as-a-Service and multi-level cloud scheduling system:**

rent cloud management systems such as Open Stack [25] lack an intelligent scheduling system for HPC applications.

Figure 8.1a shows one proposed design. A cloud can internally have different types of resources as shown in the figure. The system should leverage the knowledge of application characteristics to allocate VM instances in a more intelligent fashion. E.g. it would be beneficial to place VMs which will be used for running communication intensive applications on resources with good interconnect and low-overhead virtualization (LXC,Infi in our example) whereas embarrassingly parallel applications can perform almost equally well on all the resources shown in the figure.

**Multi-level Scheduling:** Open stack (and similar softwares such as Eucalyptus) require the presence of a hypervisor on the host machines and hence cannot be used to schedule applications on a mix of physical and virtualized resources. For such case, we need to perform an additional scheduling task at a level above the open stack scheduler (See Figure 8.1b)

**Management Portal:** We also envisage a portal which allows users to access HPC-as-a-service rather than Infrastructure-as-a-service. One possibility is to user science-as-a-service platforms such as n3phele [110], which is a cloud-based workbench, to allow users to submit applications through a web-interface and a set of commands rather than worrying about the complex management of scientific software stack. This model would be similar to the supercomputer job submission model which most HPC users are familiar with. The complexity and challenges of this approach need to be explored.

**Dynamic Re-mapping of Applications to Platforms:** In Chapter 3, we demonstrated the benefits of selecting a suitable platform for an application based on the knowledge of application characteristics. We assumed that the application can be profiled offline. However, that may not be always possible or accurate. A promising research direction is to perform dynamic adjustment of the static mapping through run-time monitoring. For this we would have to actively manage the application mapping by monitoring application execution, performing online profiling and analysis to determine if re-mapping is required, and perform the re-mapping. Re-mapping can be accomplished by using a runtime agent on each platform's master node, and by leveraging existing migration mechanisms such as checkpoint-restart or hypervisor-supported migration.

For optimizing the job during the same execution, we also need accurate online application characterization. This strategy rests on the observation that many scientific codes are iterative in nature, and their characteristic behavior changes very slowly

over time. This observation, sometimes known as the principle of persistence, can be used to generate a form of signature for the job during its execution. The runtime system can derive such a signature, e.g. after a few iterations in the run, by collecting statistics such as communication patterns or computation to communication ratios. The runtime system can then pass the captured signature to the cloud management system, which, in turn, could use this information to refine the VM allocation of that particular job.

**Colocating HPC and non-HPC Application:** In Chapter 4, we demonstrated the benefits of co-locating HPC applications with complementary profiles on same physical node. A promising related research direction is to explore techniques to schedule a mix of HPC and non-HPC applications in an intelligent fashion to increase resource utilization. E.g., co-locate VMs which are network bandwidth intensive with VMs which are compute intensive to increase resource utilization. Following two cases can be considered: a) not sharing physical cores between virtual cores and b) sharing physical core – which can benefit applications which are insensitive to noise. It needs to be explored whether real applications can benefit from this approach and to what extent. The major challenges include cross-application interference, security concerns, and guaranteeing SLAs in different metrics, e.g. response time (web application) vs. execution time (HPC application).

## 8.2 HPC-aware Clouds

**Interaction between Application and Resource Manager:** The bidirectional *control and feedback channel* developed in Chapter 6 can be used to communicate between the scheduler or the resource manager and the parallel runtime system or the application. One promising research direction is to leverage this rich channel of information transfer between them to remedy the current separation between applications and cloud job schedulers. Major questions to be addressed in this direction include: could a scheduler make better decisions if it knew more about the types of applications waiting in the queue? Could the applications run more efficiently if they cooperated with the scheduler via a dialogue? Could the overall use of a resource produce more useful results if the system could accommodate specific aspects and needs of individual jobs? Closing the semantic gap between the applications and the cloud management system can enable more efficient and robust use of existing and future clouds. By coupling the

scheduler with an adaptive runtime system, existing applications will be able to utilize these advantages without modification. We demonstrated the benefits of such channel in the context of malleable jobs and proactive fault tolerance (Chapter 6). Some other potential usage scenarios where the application can adapt to variations in extraneous factors if informed by the resource manager are: a) variations in expected effective compute and network performance of a job's allocation in a multi-tenant scenario, and b) dynamic control of system components, including CPU frequency, caches, and network switches by the resource manager.

**Job Interference Categorization:** With more aggressive interaction with the underlying infrastructure, i.e. the scheduling environment, one could be able to annotate performance data with insights about interference. With the ability to document the lightning strikes of interference, we can direct analytical tools to focus their attention on the regions which have interference, or don't have interference, depending on what problem they are trying to address. Existing performance analysis tools only tell about how the application performed. They do not give any information about the other jobs that might have consumed the shared bandwidth. The goals of future research will be to record known environmental overheads, collect static information from the cloud scheduler, acquire dynamic information from monitoring and interaction with cloud scheduler, and annotate performance data with "cloud weather report" information. We expect that through the capabilities of current cloud monitoring solutions, such as Amazon CloudWatch, one should be able to distinguish between "normal overheads" and the actions of the cloud infrastructure or other applications sharing resources.

**Security for HPC in Cloud:** In this thesis, our focus was on bridging the HPC-cloud divide by improving performance, cost, and resource utilization. Another major challenge which discourages HPC users to move to cloud are the security concerns. Some of the cloud security challenges are: lack of trust on cloud provider, loss of control, and multi-tenancy. Future research is needed to address these issues to enable secure HPC in cloud.

## 8.3 Cloud-aware HPC

**Adaptive Communication-aware Load Balancing:** A possible extension of our work on load balancing for cloud environment (Chapter 5) is to develop more intelligent and complex dynamic load balancing techniques using measured statistics based on

performance counters, and VM steal times for more accurate load instrumentation. In this thesis, our focus was on compute heterogeneity. Another kind of dynamic heterogeneity in clouds is communication heterogeneity, also arising from multi-tenancy. E.g., a communication-heavy VM from another user running on same physical nodes where one VM of a large HPC job is also running, can severely affect the performance of an HPC application. Dynamic load redistribution to address both compute and communication heterogeneity in clouds would be a challenging and promising research direction.

**Communication Optimizations in HPC Runtime:** In Chapter 2, we showed that poor network performance in clouds is a dominant challenge for HPC. We also showed that both commodity networks and network virtualization overhead contribute to the slowness of network. Chapter 2 also demonstrated that over-decomposed message-driven runtimes can alleviate the effect of slow network on HPC applications' performance. We believe that further exploration of the utility of communication optimization techniques in HPC runtimes is a fruitful research direction. Concrete directions include – 1) Use of adaptive message compression scheme to address slow commodity network, in particular bandwidth, 2) Use of topological routing and message aggregation to minimize virtualization overhead, and 3) Task/Computation duplication to address variable network performance in clouds, e.g. instead of waiting for a message stuck in network, we can re-calculate the result on replicated data/task.

**Evolving Jobs:** In Chapter 6, we presented a runtime which enables applications to expand/shrink during execution. We demonstrated the merits of such runtime in context of malleable jobs, that is where the decision to shrink/expand is external to the application, e.g., it may be triggered by job scheduler. The decision do not consider application characteristics or phases. A promising research direction is to take the decision based on application needs (evolving jobs) rather than an application-agnostic scheduling algorithm. Each application's footprint can be dynamically adjusted based on the sweet spot in the strong scaling performance curve to maximize the economic efficiency of cloud use. Moreover, application can be dynamically expanded for refinement, expanded for multiple instances, e.g., distinct interventions in an agent-based contagion simulation [4]. Similarly, a job can be contracted for coarsening, reaping of stochastic walkers, or unscalable phases of significant duration, such as those occurring in multiscale/multiphysics ensembles when integrating information across domains.

# REFERENCES


[1] "Top500 supercomputing sites," http://top500.org.

[2] "Lighting Up DreamWorks with High Performance Computing." Compete. Council on Competitiveness, Tech. Rep., November 2009, http://www.compete.org/publications/detail/1276/lighting-up-dreamworks-with-high-performance-computing.

[3] "HPC Case Study: Bringing the Power of HPC to Drug Discovery and the Delivery of Smarter Health Care." Compete. Council on Competitiveness, Tech. Rep., June 2011, http://www.compete.org/publications/detail/1742/hpc-case-study-bringing-the-power-of-hpc-to-drug-discovery-and-the-delivery-/of-smarter-health-care.

[4] J.-S. Yeom, A. Bhatele, K. Bisset, E. Bohm, A. Gupta, L. Kale, M. Marathe, D. S. Nikolopoulos, M. Schulz, and L. Wesolowski, "Overcoming the Scalability Challenges of Epidemic Simulations on Blue Waters," in *28th International Parallel and Distributed Processing Symposium(IPDPS) '14*, 2014.

[5] V. K. S. V. N. Uchida, V. H. Kuraishi, and V. J. Wagner, "Hpc solutions for the manufacturing industry," *FUJITSU Sci. Tech. J*, vol. 44, no. 4, pp. 458–466, 2008.

[6] M. Richards, A. Gupta, O. Sarood, and L. V. Kale, "Parallelizing Information Set Generation for Game Tree Search Applications," in *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, ser. SBAC-PAD '12. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: http://dx.doi.org/10.1109/SBAC-PAD.2012.42 pp. 116–123.

[7] "Graph500," http://graph500.org.

[8] P. Mell and T. Grance, "The nist definition of cloud computing," *NIST special publication*, vol. 800, no. 145, p. 7, 2011.

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

[10] A. Gupta and L. Kale, "Towards efficient mapping, scheduling, and execution of hpc applications on platforms in cloud," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2013.125 pp. 2294–2297.

[11] A. Iosup et al., "Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 6, pp. 931 –945, june 2011.

[12] 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.

[13] "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.

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

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

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

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

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

[19] 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.

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

[21] C. Evangelinos and C. N. Hill, "Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2." Cloud Computing and Its Applications, Oct. 2008.

[22] 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.

[23] P. Mehrotra, J. Djomehri, S. Heistand, R. Hood, H. Jin, A. Lazanoff, S. Saini, and R. Biswas, "Performance Evaluation of Amazon EC2 for NASA HPC applications," in *Proceedings of the 3rd workshop on Scientific Cloud Computing*, ser. ScienceCloud '12. New York, NY, USA: ACM, 2012, pp. 41–50.

[24] J. Napper and P. Bientinesi, "Can cloud computing reach the top500?" in *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, ser. UCHPC-MAW '09. ACM, 2009.

[25] "Open Stack Open Source Cloud Computing Software," http://www.openstack.org/.

[26] 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.

[27] L. Kale and S. Krishnan, "Charm++: A Portable Concurrent Object Oriented System Based on C++," in *OOPSLA*, September 1993.

[28] 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.

[29] A. Gupta, L. V. Kalé, D. S. Milojicic, P. Faraboschi, R. Kaufmann, V. March, F. Gioachin, C. H. Suen, and B.-S. Lee, "The Who, What, Why, and How of HPC Applications in the Cloud." in *5th IEEE Intl. Conf. on Cloud Comp. Techno. and Sc. (CloudCom) '13, Best Paper*.

[30] "Ranger User Guide," http://services.tacc.utexas.edu/index.php/ranger-user-guide.

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

[32] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus Open-source Cloud-computing System," in *Proceedings of Cloud Computing and Its Applications*, Oct. 2008.

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

[34] A. J. Younge et al., "Analysis of Virtualization Technologies for High Performance Computing Environments," *Cloud Computing, IEEE Intl. Conf. on*, vol. 0, pp. 9–16, 2011.

[35] D. Schauer et al., "Linux containers version 0.7.0," June 2010, http://lxc.sourceforge.net/.

[36] "Intel(r) Virtualization Technology for Directed I/O," Intel Corporation, Tech. Rep., Feb 2011, http://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf.

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

[38] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, 1993, pp. 91–108.

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

[40] 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.

[41] 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.

[42] "The ASCII Sweep3D code," http://wwwc3.lanl.gov/pal/software/sweep3d.

[43] M. Koop, T. Jones, and D. Panda, "MVAPICH-Aptus: Scalable high-performance multi-transport MPI over InfiniBand," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008, pp. 1 –12.

[44] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proc. of 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, 2004.

[45] L. Kalé and A. Sinha, "Projections : A Scalable Performance Tool," in *Parallel Systems Fair, Intl. Parallel Processing Sympos ium*, Apr. 1993.

[46] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *The International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, Fall 1999. [Online]. Available: citeseer.ist.psu.edu/zaki99toward.html

[47] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm, "Netgauge: A Network Performance Measurement Framework," in *Proceedings of High Performance Computing and Communications, HPCC'07*, vol. 4782. Springer, Sep. 2007, pp. 659–671.

[48] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation," in *Supercomputing 10*, Nov. 2010.

[49] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell, "Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing," ser. IPDPS '10, pp. 1–12.

[50] B. Kocoloski, J. Ouyang, and J. Lange, "A case for dual stack virtualization: consolidating HPC and commodity applications in the cloud," ser. SoCC '12, New York, NY, USA, 2012, pp. 23:1–23:7.

[51] T. Watanabe, M. Nakao, T. Hiroyasu, T. Otsuka, and M. Koibuchi, "Impact of topology and link aggregation on a PC cluster with ethernet," in *CLUSTER*. IEEE, Sep.-Oct. 2008, pp. 280–285.

[52] C. Bischof, D. anMey, and C. Iwainsky, "Brainware for Green HPC," *Computer Science - Research and Development*, pp. 1–7, 2011. [Online]. Available: http://dx.doi.org/10.1007/s00450-011-0198-5

[53] "ISC launches hosted HPC service," http://insidehpc.com/2008/04/01/isc-launches-hosted-hpc-service.

[54] "SGI Announces Cyclone Cloud Computing for Technical Applications," http://www.sgi.com/company_info/newsroom/press_releases/2010/february/cyclone.html.

[55] "NVIDIA GRID," http://www.nvidia.com/object/virtual-gpus.html.

[56] J. Ekanayake, X. Qiu, T. Gunarathne, S. Beason, and G. C. Fox, *High Performance Parallel Computing with Clouds and Cloud Technologies*, 07/2010 2010.

[57] 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.

[58] E. Roloff, M. Diener, A. Carissimi, and P. Navaux, "High Performance Computing in the Cloud: Deployment, Performance and Cost Efficiency," in *CloudCom 2012*, 2012, pp. 371–378.

[59] A. Marathe, R. Harris, D. K. Lowenthal, B. R. de Supinski, B. Rountree, M. Schulz, and X. Yuan, "A Comparative Study of High-performance Computing on the Cloud," ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 239–250.

[60] A. Gupta, L. Kale, D. Milojicic, P. Faraboschi, and S. Balle, "HPC-Aware VM Placement in Infrastructure Clouds ," in *IEEE Intl. Conf. on Cloud Engineering IC2E '13*, March. 2013.

[61] A. Gupta, O. Sarood, L. Kale, and D. Milojicic, "Improving HPC Application Performance in Cloud through Dynamic Load Balancing," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pp. 402–409.

[62] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, "How much parallelism is there in irregular applications?" *SIGPLAN Not.*, vol. 44, no. 4, pp. 3–14, Feb. 2009.

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

[64] C. da Lu and D. Reed, "Compact Application Signatures for Parallel and Distributed Scientific Codes," in *Supercomputing, ACM/IEEE 2002 Conference*.

[65] A. Wong, D. Rexachs, and E. Luque, "Parallel Application Signature," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 31 2009-sept. 4 2009, pp. 1 –4.

[66] J. S. Vetter, N. Bhatia, E. M. Grobelny, P. C. Roth, and G. R. Joubert, "Capturing Petascale Application Characteristics with the Sequoia Toolkit," in *In Proceedings of Parallel Computing 2005. Malaga*, 2005.

[67] D. H. Bailey and A. Snavely, "Performance Modeling: Understanding the Past and Predicting the Future," in *in Euro-Par 2005*, p. 185.

[68] A. Snavely, N. Wolter, and L. Carrington, "Modeling Application Performance by Convolving Machine Signatures with Application Profiles," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 149–156.

[69] H. Kim, Y. el Khamra, I. Rodero, S. Jha, and M. Parashar, "Autonomic Management of Application Workflows on Hybrid Computing Infrastructure," *Scientific Programming*, vol. 19, no. 2, pp. 75–89, Jan. 2011.

[70] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani, "Cloudward bound: planning for beneficial migration of enterprise applications to the cloud," in *Proceedings of the ACM SIGCOMM 2010 conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851212 pp. 243–254.

[71] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-crummey, D. Reed, L. Torczon, and R. Wolski, "The GrADS Project: Software Support for High-Level Grid Application Development," *International Journal of High Performance Computing Applications*, vol. 15, pp. 327–344, 2001.

[72] L. V. Kalé, S. Kumar, J. DeSouza, M. Potnuru, and S. Bandhakavi, "Faucets: Efficient resource allocation on the computational grid," Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep. 03-01, Mar 2003.

[73] A. Gupta, D. Milojicic, and L. V. Kalé, "Optimizing VM placement for HPC in the cloud," in *Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit*, ser. FederatedClouds '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2378975.2378977 pp. 1–6.

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

[75] 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.

[76] 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.

[77] *The CONVERSE programming language manual*, Department of Computer Science,University of Illinois at Urbana-Champaign, 2006.

[78] 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.

[79] 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.

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

[81] 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.

[82] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "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.

[83] 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.

[84] O. Sarood, A. Gupta, and L. V. Kale, "Cloud Friendly Load Balancing for HPC Applications: Preliminary Work," in *Parallel Processing Workshops (ICPPW), 2012 41st Intl. Conf. on*, sept. 2012, pp. 200 –205.

[85] 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.

[86] T. Sterling and D. Stark, "A High-Performance Computing Forecast: Partly Cloudy," *Computing in Sci. and Engg.*, pp. 42–49, July 2009.

[87] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard, "Grid'5000: A large scale and highly reconfigurable grid experimental testbed," in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, ser. GRID '05. Washington, DC, USA: IEEE Computer Society, 2005. [Online]. Available: http://dx.doi.org/10.1109/GRID.2005.1542730 pp. 99–106.

[88] L. Sarzyniec, T. Buchert, E. Jeanvoine, and L. Nussbaum, "Design and evaluation of a virtual experimental environment for distributed systems," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, Feb 2013, pp. 172–179.

[89] H. Menon, N. Jain, G. Zheng, and L. V. Kalé, "Automated load balancing invocation based on application characteristics," in *IEEE Cluster 12*, Beijing, China, September 2012.

[90] G. Zheng, "Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing," Ph.D. dissertation, Dept. of CS, Univ. of Illinois at Urbana-Champaign, 2005.

[91] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for smp vms?" ser. EuroSys '11. NY, USA: ACM, 2011, pp. 257–272.

[92] R. K. Brunner and L. V. Kalé, "Adapting to Load on Workstation Clusters," in *7th Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, Feb. 1999, pp. 106–112.

[93] D. Brown et al., "Scientific Grand Challenges: Crosscutting Technologies for Computing at the Exascale." U.S. DOE PNNL 20168, Report from Workshop on Feb. 2-4, 2010, Washington, DC, Tech. Rep., 2011.

[94] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. O. A. Navaux, "Supporting Malleability in Parallel Architectures with Dynamic CPUSETs Mapping and Dynamic MPI," in *11th international conference on Distributed computing and networking*, ser. ICDCN'10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2018057.2018090

[95] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "Theory and Practice in Parallel Job Scheduling," in *Job Scheduling Strategies for Parallel Processing*, ser. IPPS '97, London, UK, 1997. [Online]. Available: http://dl.acm.org/citation.cfm?id=646378.689517 pp. 1–34.

[96] L. V. Kalé, S. Kumar, and J. DeSouza, "A Malleable-Job System for Timeshared Parallel Machines," in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, May 2002.

[97] K. El Maghraoui, T. Desell, B. Szymanski, and C. Varela, "Dynamic Malleability in Iterative MPI Applications," in *IEEE CCGrid 2007*.

[98] R. A. Dutton and W. Mao, "Online scheduling of malleable parallel jobs," in *19th IASTED Intl. Conference on Parallel and Distributed Computing and Systems*, ser. PDCS '07. [Online]. Available: http://dl.acm.org/citation.cfm?id=1647539.1647566

[99] J. Hungershofer, "On the Combined Scheduling of Malleable and Rigid Jobs," in *SBAC-PAD 2004*. IEEE.

[100] G. Utrera, J. Corbalan, and J. Labarta, "Implementing Malleability on MPI Jobs," in *13th IEEE Intl Conf. on Parallel Arch. and Compilation Techniques (PACT'04)*.

[101] D. G. Feitelson and L. Rudolph, "Toward Convergence in Job Schedulers for Parallel Supercomputers," in *Job Scheduling Strategies for Parallel Processing*, 1996.

[102] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema, "Scheduling Malleable Applications in Multicluster Systems," in *IEEE Cluster, 2007*.

[103] M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoeflinger, "Object-Based Adaptive Load Balancing for MPI Programs," in *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, May 2001, pp. 108–117.

[104] G. R. Gao, T. L. Sterling, R. Stevens, M. Hereld, and W. Zhu, "Parallex: A study of a new parallel computation model," in *IPDPS*, 2007, pp. 1–6.

[105] A. Gupta, G. Zheng, and L. V. Kale, "A Multi-level Scalable Startup for Parallel Applications," in *Proceedings of International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '11, Tucson, AZ, USA, 5 2011.

[106] F. Gioachin, C. W. Lee, and L. V. Kalé, "Scalable Interaction with Parallel Applications," in *Proceedings of TeraGrid'09*, Arlington, VA, USA, June 2009.

[107] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[108] S.-H. Chiang and M. K. Vernon, "Dynamic vs. Static Quantum-based Parallel Processor Allocation," in *Job Scheduling Strategies for Parallel Processing*, 1996.

[109] S. Chakravorty, C. L. Mendes, and L. V. Kalé, "Proactive fault tolerance in mpi applications via task migration." in *HiPC*, ser. Lecture Notes in Computer Science, vol. 4297. Springer, 2006, pp. 485–496.

[110] "Bioinformatics in the cloud," http://n3phele.com.