vrije Universiteit amsterdam

MASTER PROJECT COMPUTER SCIENCE

ElViC: Malleable MPI with Virtual Machines

Supervisor:
Prof. Dr. Ir. Henri BAL

Author:
Plamen DIMITROV

Daily Supervisor:
Dr. Rutger HOFMAN

January 28, 2015

Contents

ElViC: Malleable MPI with Virtual machines

Plamen Dimitrov

January 28, 2015

Abstract

Traditional MPI parallel applications have the number of processes used in the computation hard-wired in the program model. This makes it impossible to adjust this number during execution - i.e. scaling to use more or less resources is impossible at the application level. Virtualization and live migration of VMs can be used to dynamically add and remove physical resources to a virtual cluster to implement malleability at the infrastructure level. In a compute cluster setting, this makes it possible to reserve and add idle compute nodes to an MPI application running on a virtual cluster, and release them if they are needed elsewhere or too many jobs are queued on the cluster. ElViC, a system that transparently manages an elastic virtual cluster was implemented and used to dynamically scale rigid MPI applications. A series of experiments revealed that embarrassingly parallel and coarse-grained parallel applications can benefit from this approach if they are scaled up to more physical resources. More communication-intensive applications were shown to be too badly affected by virtualization alone to benefit from this approach.

Acknowledgements

I would like to thank:

Rutger Hofman - for guiding me through the whole process, helping me with numerous technical issues and giving me useful feedback every week.

Kaveh Razavi - for sharing his expertise with me and finding the time to grade my thesis despite his busy schedule.

Kees Verstoep - for providing me with the infrastructure needed to perform my experiments, his immediate reaction in cases of technical problems and giving me the tools to solve some of these problems myself, whenever possible.

My family and friends - for all the encouragement and for constantly reminding me that I should graduate sooner, rather than later.

1 Introduction

Message passing interface (MPI) is widely used in high-performance parallel and distributed applications. Typically, in traditional MPI, the number of processes that are used in the computation is hard-wired in the program model and cannot be changed at runtime. This implies that the amount of physical resources allocated to an MPI parallel program remains fixed during its execution. While more recent versions of MPI such as MPI version 2.0 allow for changing the number of processes during the execution, most existing MPI applications are not designed to support this. Having the ability to change the number of computational entities in an application, also known as *malleability*, can be very useful in situations where one needs to scale the applications up and down following the availability of resources or to save energy. Techniques that make it possible to dynamically scale "rigid" MPI applications and make them "malleable" are, therefore, high in demand.

Modern virtualization technology creates the illusion of making OS instances and hardware available to multiple users on a single physical machine and enables deployments that previously required separate physical machines. Most recent hypervisors, such as KVM and Xen, allow the relocation of virtual machines between physical hosts via a feature called migration. Live migration refers to migration done in a seamless way - i.e. the perceived downtime during migration is so short that it makes the whole process transparent to the application layer. Normally, during live migration the applications being run inside the migrated VM will only see a negligible increase in latency. This feature can be used to effectively shift workloads by adjusting the number of physical machines used in computational tasks. For example, a parallel application can be started on $4N$ VMs, each running P processes, distributed on a set of N hosts. When more resources become available, the VMs can be migrated and spread out evenly across $2N$ or $4N$ hosts to achieve a speed-up by decreasing the load of each individual host. Since live migration is transparent and application-agnostic it is a possible solution for scaling rigid parallel (MPI) applications. While, in theory, this is an effective way to distribute workloads, virtualization is known to bring overhead, both in terms of computational efficiency and especially communication, that can negatively affect the performance of a parallel application being run across VMs.

The aim of this study is to design and implement a system that transparently scales rigid MPI applications via live migration of virtual machines and determine the feasibility and applicability of this approach. To this end, ElViC, an application built around OpenNebula and Sun Grid Engine that manages an elastic virtual cluster was implemented. It manages the allocation and the release of physical resources in a compute cluster setting and live-migrates the VMs on which an MPI application is run to dynamically grow and shrink a virtual cluster. It can adapt to the load on the underlying physical compute cluster and dynamically scale by adding new resources to the virtual cluster as they become available or releasing the reserved resources if they are needed for another task or by another user. A series of experiments with different parallel applications that vary in their granularity and communication patterns were performed with ElViC to identify the types of applications that can benefit the most from this approach, and those for which it brings no real benefits.

The findings from these experiments suggest that live migration performed to scale a running parallel MPI application to use more physical resources in a virtual cluster setting can improve performance, if done shortly after the application is started. This, however, is only true for CPU-intensive applications that do no communicate as much - embarrassingly parallel and coarse-grained parallel applications, such as matrix multiplication (PDGEMM, PSGEMM). A speed-up of up to 79% was achieved through live migration in embarrassingly parallel applications, performance on bare metal being the baseline. More communication-intensive applications, such as solving a three-dimensional discrete Poisson equation using the multigrid method (MG), were too

negatively affected by virtualization alone and thus could not benefit from this approach. These findings, together with the design and implementation of ElViC constitute the main contribution of this work.

The document is structured as follows: Section 2 briefly presents a selection of related work. Section 3 introduces the main concepts and tools that are used to solve the problem. Section 4 then describes the design and implementation specifics of ElViC in detail. In Section 5 the reasoning behind the experiments designed to test the feasibility of our approach is discussed. The results, accompanied by discussion, are presented in Section 6. Finally, after the concluding remarks in Section 7, possible future developments are briefly discussed in Section 8.

2 Related work

Many approaches to achieving malleability for MPI that do not involve virtualization are discussed in literature. A short overview of several such is presented below:

[Maghraoui and Desell, 2005] introduced a software framework that improves the performance of MPI applications through adaptive middleware for load balancing that includes process checkpointing and migration. The new approach showed major improvements in flexibility, scalability and performance.

[Van Nieuwpoort et al., 2010] proposed Satin - a high-level programming model for parallel grid applications with support for malleability, migration and fault tolerance. The model allows for adding and removing compute resources automatically to match the application's requirements and the utilization of resources on the grid.

[Cera and Georgiou, 2009] achieve malleability for MPI applications via the OAR resource manager. A module for the latter was developed to manage and identify the resources availability, sending such information to the MPI application. Experiments indicated an improvement of almost 35% in resource utilization.

[Desell et al., 2006] designed a modular decentralized middleware framework for dynamic application reconfiguration which allows HPC applications to scale up to arbitrary large numbers of processing nodes in a relatively transparent way. Experiments demonstrate that malleable components improve application performance and resource utilization with little added overhead.

[Iglesias, 2007] used virtual malleability to improve performance and reduce fragmentations in MPI jobs. In contrast with the approach taken in ElViC (live migration of VMs), "virtual malleability" is achieved by varying the multiprogramming level. "Self coscheduling", a mechanism proposed in their work, was used to make the partition of an MPI job modifiable at runtime.

The following articles deal with virtualization:

[Huang et al., 2007] presented "Nomad" - a design for migrating modern interconnects in cluster environments running VMs. A prototype based on the Xen VM monitor and InfiniBand was found effective in achieving migration of network resources.

[Ruivo et al., 2014] modified OpenNebula and the Linux's hypervisor and employed the Single Root I/O virtualization (SR-IOV) to minimize the virtualization overhead. Although the use or SR-IOV was impossible in ElViC, due to some limitations of the underlying compute cluster, the components of the system support it and can be adapted to use it, if enabled.

[Huang, 2008] addressed the problems of reducing the network I/O virtualization overhead, the cost of inter-VM communication on the same physical host, reducing the management overhead and improving transparency in VM-based environments using modern interconnects. As mentioned above, the use of high-speed modern interconnects was not possible in ElViC. Enabling

this feature can significantly improve performance.

[Knauth and Fetzer, 2011] used virtual machines with live migration to dynamically scale non-elastic applications. The perceived downtime in their analysis, however, tended to be very long - ranging between 20 and 50 seconds. They argue that in certain scenarios, especially when the page dirtying rate of a VM is high, live migration may not be possible at all. Having knowledge about the page dirtying rate of a VM can be useful in estimating the expected downtime during live migration.

[Kudryavtsev and Koshelev, 2012] looks into the prospects of using virtualization in a high performance computing setting. The authors try to provide an optimal configuration for the KVM and Palacios hypervisors and managed to significantly reduce the performance degradation in multiple scenarios. Additionally, techniques to configure the host OS for optimal performance were discussed. The results showed similar performance for KVM and Palacios, when properly tuned, with Palacios outperforming KVM at fine-grained tests and KVM providing more stable and predictable results. VM migration was not discussed. The experimental part of this thesis deals mostly with executing high performance applications in KVM-powered virtual clusters. The scenarios involving KVM described in the article, however, do not involve migration and can, therefore, only be useful to understand the effects of virtualization alone on performance.

[Wang and Varela, 2010] proposed a model that allows for malleability of virtual machines using a VM malleability middleware and the component migration features of the SALSA programming language. While in ElVIC the number of VMs stays constant during execution and malleability is achieved by live-migrating VMs, "virtual machine malleability" refers to splitting and merging the actual VMs to change the distribution of processes across physical resources.

[Ye et al., 2012] look into various migration strategies for virtual clusters. A framework "VC-Migration" that controls the migration of virtual clusters was discussed and a series of experiments were performed to assess the performance and overhead for each migration strategy. Based on that, some optimization techniques were proposed. The results suggest that selection of suitable concurrent migration granularity is crucial to optimizing performance - large concurrent granularity (migrating more VMs at the same time) can be detrimental.

[Zhao and Figueiredo, 2007] discussed the VM-based resource reservation problem, i.e. the reservation of physical (CPU, memory, network) resources for individual VM instances and VM clusters. A model was proposed that aims to characterize the VM migration process and make it feasible to predict the performance characteristics of a migrated virtual machine. Scenarios where multiple VMs are migrated sequentially or in parallel are studied. The findings suggest that migration strategies can significantly affect the applications being run in the migrated VMs - migrating VMs sequentially can reduce the performance overhead, whereas doing it in parallel can speed-up the migration process.

[Lagar-Cavilla, 2009] showed that virtualization can be employed to relocate tasks and dynamically scale the footprint of computationally-intensive tasks in a public cloud. Live migration was used to balance the load between the edges (user) and the core (distant servers) of cloud server farms - interactive phases of the application can be executed locally or closer to the user, while a "crunch phase" can be run on distant compute servers. While the approach used it similar to ours in that it uses live VM migration, it requires dynamic adaptation to the state of the application. In contrast, ElViC is completely agnostic to the state of the applications run on the virtual cluster.

[Garces et al., 2012] presented an approach for allowing opportunistic infrastructures to support MPI applications. They suggest that a component which automates the execution processes of MPI jobs should be used to achieve this. The latter must be responsible for deploying, monitoring and restarting MPI clusters and jobs when needed. The concept of a opportunistic cloud computing infrastructures is closely related to that of virtual clusters. However, their proposed approach uses Checkpoint/Restart recovery techniques to reliably run MPI applications in volatile virtual

clusters distributed across multiple (desktop) computers, whereas ElViC is implemented on top of a compute cluster and relies on VM live migration.

[Raveendran et al., 2011] propose an approach for making existing MPI applications elastic in a cloud setting. The proposed system includes an automated framework that decides when scaling happens depending on performance, cost or user-set criteria. This approach involves reconfiguring and restarting a running MPI application to continue running on a different number of nodes, instead of live VM migration.

3 Methods and tools

The tools and technologies used to implement the elastic virtual cluster and the monitor application that manages it are discussed below. The Distributed ASCI Supercomputer 4 (DAS-4)[1] provides the infrastructure to implement this system. It is a Linux cluster running Red Hat 4.4.6-4 and has Open MPI and Qemu/KVM pre-installed. An OpenNebula instance is also run and managed on DAS-4. The reservation of physical resources, virtualization, live migration and the associated systems used to manage these on DAS-4 are discussed below.

3.1 Open MPI

Open MPI is a modern and flexible MPI implementation. While some MPI implementations focus on different aspects of high-performance computing or are tailored to solve a particular research question, Open MPI's component architecture makes it a stable platform for third-party research while being extendible through independent software add-ons [Gabriel et al., 2004].

3.2 Sun Grid Engine

A reservation system is necessary to schedule and execute user jobs on a compute cluster. Sun Grid Engine (SGE) is a commercially supported open-source batch-queuing system for distributed resource management [Gentzsch, 2001]. It is the system being used on DAS-4 where the experiments were performed. A standard set of PBS commands (*qsub, qstat, qdel*) is used to respectively submit, monitor and delete jobs. Since in our system no jobs are directly executed on the compute nodes of the cluster, this set of commands is only used to manage the (de)allocation of physical resources to a virtual cluster and monitoring.

3.3 Virtualization

Instead of creating major portions of an operating system kernel themselves, as most other hypervisor do, KVM essentially uses the Linux kernel as a hypervisor. This is made possible by developing KVM as a kernel module - a minimally intrusive method. Following this approach, virtualization capabilities are added to a standard Linux kernel and each virtual machine can run as a standard Linux process, scheduled by the Linux scheduler. While, typically most Linux processes run in either kernel or user mode, KVM introduces a new "guest" execution mode, whereby guest mode processes are run from within the virtual machine [Kivity et al., 2007].

With multiple distinct virtualization systems available for the Linux Kernel: KVM, Xen, VMI, IBM's System p and z and others, each until recently having their own block, network, console and other drivers with different features and optimizations, the complexity of managing the drivers for each system grows constantly. Virtio is a series of efficient, well-maintained Linux drivers that can be adapted to use in various hypervisors using a shim layer, which transparently intercepts an API

[1] http://www.cs.vu.nl/das4/

and changes the passed parameters, redirects an operation or handles it itself. It aims to eliminate this complexity by providing a common ground for drivers to be used in virtualized systems. KVM uses a ring buffer transport implementation (vring) provided by virtio [Russell, 2008].

The Palacios hypervisor is a lightweight hypervisor designed specifically to be used for high performance computing. [Kudryavtsev et al., 2012] assessed the performance or KVM and Palacios in a high performance computing setting and concluded that with proper tuning the results are similar. KVM tends to provide more stable and predictable results, whereas Palacios was shown to perform much better on larger-scale fine-grained tests but had abnormal behavior during several tests.

3.4 Live Migration of Virtual Machines

Migrating virtual machines that contain an OS instance across physical hosts can be very useful in a cluster setting. The separation between hardware and software introduces a high level of flexibility and facilitates fault management, load balancing and system maintenance. In live migration, the majority of the migration is carried out as the guest OS continues to run. Modern techniques for live migration make it possible to greatly minimize the service downtimes. This enables the use of live migration for practical purposes.

There are two common techniques for migrating virtual machines - pre-copy and post-copy migration.

3.4.1 Pre-copy migration

In pre-copy migration, the copying of the VM to the recipient host starts while that VM is still running on the source host. Memory pages of the migrated VM are iteratively copied from the source to the destination host while the execution of the VM continues. The transfer of a consistent snapshot is ensured by using page-level protection hardware and a rate-adaptive algorithm controls the impact of migration traffic on running services. If a transfered page is dirtied it is resent to the destination at the next round. Once certain conditions regarding the page-fault rate are met, the VM is paused at the source host, the remaining pages are copied to the destination and the copy of the VM is started on the destination host. Such condition is usually the detection of a small writeable working set [Clark et al., 2005], [Agrawal and Pateriya, 2013].

Pre-copy migration proceeds in the following stages:

- **Push phase**
  The hypervisor first copies all memory pages to the destination host. It then starts iteratively copying the memory pages modified at the previous round - pages transfered at round n are those dirtied at round $n-1$. This process continues until the set of pages that are dirtied often (*writeable working set* or WWS) becomes small. This is done to help achieve the minimum possible downtime during the next stage.

- **Stop-and-copy phase**
  In pre-copy migration this is typically a very short phase, in which the running OS instance at the source VM is suspended and the network traffic is redirected to the destination VM. The CPU state and any remaining modified memory pages are copied to the destination - both the source and the destination hosts have a suspended consistent copy of the VM at this point. The copy at the source host can still be resumed in the case of failure.

- **Commitment and Activation phases**
  The VM copy at the destination host communicates to the source host that a consistent OS image has been received. Once an acknowledgement is sent the original VM can be

discarded and the destination host becomes the primary host for that VM. The migrated VM is then activated at the destination host during the activation phase, device drivers are reattached and the moved IP addresses are advertised.

3.4.2 Post-copy migration

Post-copy live migration defers the transfer of a VM's memory until after its processor memory has been successfully sent to the destination host. The goal of this approach is to further minimize the perceived downtime during live migration by minimizing the amount of VM state that is being transfered. In contrast with pre-copy migration, post-copy migration is a pull-dominated process, where the memory pages of the VM are actively pushed from the source to the destination after the processor state is transfered. Faulted memory pages at the destination copy that are not pushed yet are demand-pulled over the network from the source. This technique guarantees that each memory page is transmitted at most once and in that aspect improves on standard pre-copy migration, where there is some overhead due to duplicate transmissions [Hines et al., 2009].

The KVM/Qemu hypervisor with pre-copy migration was used in our design.

3.5 OpenNebula

OpenNebula is a popular open-source solution for the comprehensive management of virtualized data centers. OpenNebula makes it easy to manage the creation and deletion of virtual hosts and machines. It manages the entire life cycle of the virtual machines. It also controls VM migration and manages the associated virtual network to ensure the migrated VMs stay mapped to their respective IP addresses [Milojičić et al., 2011].

4 Monitor design and implementation

ElViC, a multithreaded monitor application that takes user input and communicates with Sun Grid Engine and OpenNebula to create and manage an elastic virtual cluster (on top of a physical one) was implemented in Python. It takes input from the command line or a dedicated UI, manages the allocation and release of physical resources via SGE, the lifecycle of VMs and VM migration via OpenNebula, and transparently runs an MPI application on the resulting elastic virtual cluster. A *createONEnode* routine combines the tasks associated with reserving a physical compute node and adding it as a VM host to OpenNebula. A *releaseHandler* routine manages the release of a compute node that hosts one or more VMs in the virtual cluster by reserving a new node and safely migrating the VMs away before releasing the old node, or redistributing VMs evenly across the available nodes if a new one cannot be reserved. By scheduling the run of the *releaseHandler* forward in time, shortly before the reservation for a node expires, the monitor implements a keep-alive behavior that keeps the MPI application alive for as long as there are physical resources to add to the virtual cluster. A dedicated *watchdog* thread manages the growing, i.e the addition of more physical resources to the cluster and the associated migration to spread out VMs evenly, if more resources become available or if the demands change as a response to a user-generated event. A diagram representing the high-level design of the system can be seen on Fig. 1. Multiple instances of ElViC can run in parallel, managing several virtual clusters, without interfering with each other.

The behavior of ElViC is mostly determined by the M and m parameters, where M refers to the total number of VMs in the virtual cluster which is fixed during the run and m refers to the preferred number of hosts (can be set by user or scheduled to change during the run). If the number of hosts currently in use in a virtual cluster is smaller than m, new hosts are added as soon as they

can be reserved and VMs will be migrated away and spread out evenly by the *watchdog* thread - i.e. the virtual cluster grows. Setting *m* to a higher value after a virtual cluster has been initialized will also cause it to grow and the VMs to be redistributed evenly. Setting a smaller *m* will only take effect (i.e. shrink the virtual cluster) after the reservations for the currently used hosts expire. A separate *shrink* routine additionally manages the release of these resources to immediately shrink the cluster if necessary. The operation of ElViC and the complete set of parameters and arguments are discussed in the sections below.

In the text below, until the end of the thesis, the terms *(physical/compute) node* and *(physical) host* will be used interchangeably. A *virtual host* will refer to the notion of a "virtual host" in OpenNebula - an abstraction on top of a physical host that designates it as a host on which VMs can be deployed via OpenNebula.

Since the monitor is the core component of ElViC, the terms "monitor" and "ElViC" may be used interchangeably in the text.

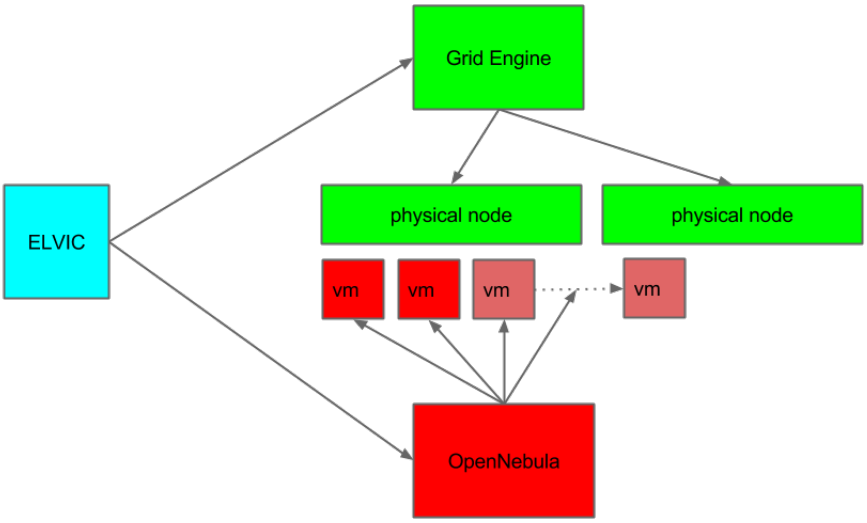

Figure 1: System design

The monitor takes the following arguments from the command line:
monitor.py **M FILE** [-m] [-g] [-p] [-mem] [-vcpu] [-mpirun] [-i] [-nompi] [-na] [-nodeploy]

M number of VMs
m number of hosts
g schedule growing to a specified number of hosts at a specified time
p number of processes per VM (slots in hostfile)
mem amount of RAM (in MB) per VM
vcpu number of virtual CPUs per VM
mpirun a list of arguments to be supplied to the MPI application
i initialize from the command line and do not start user interface

nompi do not start mpirun internally
nodeploy do not copy executable to VMs, assume image contains executable
na run in non-adaptive mode (*watchdog* and *release handler* are disabled)
FILE path to MPI executable to be run on the virtual cluster

4.1 User Interface

The monitor uses a simple *curses*-like user interface powered by *Urwid*. The user can input commands and see real-time information about the current reservations, virtual machines and virtual hosts.

The following commands are available:

init Initializes a virtual cluster. Starts the MPI application on the virtual cluster (unless the -nompi parameter is set on the command line).

m *int* Sets the number of hosts. If the hosts currently in use are fewer than this and more resources can be reserved, the monitor starts scaling up and migrating VMs away to new hosts immediately.

release *hostname* Deletes the specified host and the associated reservation. If more resources are available and can be reserved, migrates the VMs residing on the host being released to a newly reserved one. Otherwise, spreads the VMs evenly across the hosts currently in use.

newnode Creates a new host

deploy *filename* Copies the specified file to all VMs

shrink *int* Shrinks the virtual cluster to the specified number of hosts. Takes effect immediately. The hosts corresponding to the reservations with the longest remaining time will be selected and left running. The VMs are spread as evenly as possible between these hosts.

4.2 Initialization

When the *init* command is entered, the monitor runs the initialization routine. It first attempts to create as many virtual hosts as specified by the *m* parameter. If at any point during this stage the initialization routine timeouts while waiting on the creation of a host, the routine proceeds with the currently allocated hosts. If the monitor is run in non-adaptive mode (-na option from command line) it will exit if *m* hosts cannot be reserved. The monitor then starts *M* VMs, determines how many VMs on average should be on a host given the current number of available hosts and proceeds to distribute the VMs evenly across the hosts. Temporary VM templates for OpenNebula are generated and used to set the amount of memory and number of virtual CPUs per VM and deploy each VM to a specific host. During the initialization of the VMs the monitor waits until each VM gets an IP address on the virtual network and stores this in a globally accessible hash table referenced by the VM's unique ID. A *hostfile* containing the IP addresses for all VMs in the virtual cluster is then generated. The number of slots per VM in the hostfile is determined by the *p* argument which specifies the number of processes to be run per VM.

The location of each VM is stored in another hash table referenced by hostname. The monitor keeps and updates this state information internally to keep the frequency of polling the OpenNebula instance to a minimum.

Once the VMs are initialized and distributed accordingly on the hosts the executable is deployed on each VM. This is done with Fabric[2]. A *fabfile* containing a *deployFiles* routine sits in

[2]http://www.fabfile.org/

the project directory. Fabric uses this routine and the IP addresses of the VMs to streamline the deployment of the executable file (defined by the *filename* parameter or set via the *deploy* command in the UI) to all VMs via SSH. Fabric makes is easy to automate any administration task - additionally, libraries or arbitrary packages can be installed during the deployment. If the VM images already contain the executable and the necessary libraries, the deployment can be disabled via the "-nodeploy" command-line option.

Before the deployment is done the monitor waits until all VMs have booted up and started an ssh server. The ssh daemon has to be running in all VMs to start the MPI application. In order to detect when the deployment can be done or the MPI application can be run a port-scan is done every 10 seconds after the virtual cluster is initialized and the VMs are started. Port 22 is scanned on all VMs via nmap[3]and the resulting report is periodically checked until it shows the port is open on all of them.

After the executable is copied to all VMs and all necessary libraries and packages are installed, the *watchdog* is started and mpirun is called in separate threads. Alternatively, one can choose not to run mpirun from within the monitor and just use the generated *hostfile* to do it from the command line.

mpirun is started as:

```
mpirun -n <HOSTS> --hostfile hostfile <EXECUTABLE>
```

from the command line or using *subprocess* in Python.

4.3 Interaction with OpenNebula and SGE

The monitor application communicates to both OpenNebula and SGE in order to create and manage the virtual cluster.

The interaction with SGE is done by the *createONEnode* routine, the *release handler*, the termination routine and the UI and is limited to the *qstat*, *qdel* and *qsub* functions. During termination *qdel* is used to release the reservation for a host. The UI thread only uses

```
qstat -f |grep <USERNAME>
```

every second to display the list of current reservations to the user. During the creation of a virtual host, the *createONEnode* routine submits an "empty" batch job attempting to reserve a whole physical node:

```
#!/bin/bash
#$ -pe openmpi 8
#$ -l h_rt=0:15:00
#$ -N EVC
#$ -cwd
sleep 15m
```

(for a 15 minute maximum job duration). It does this via the *qsub* command.

In order to check if the reservation succeeded and find which one the reserved node is, the *qstat -f -xml* command is used to get the desired output in an XML format. Using Python's subprocess module a pipe can be established to the qstat process and *xpath* can be used to process the XML and obtain the hostname of the node associated with the reservation:

```
qstat = subprocess.Popen(['qstat', '-f', '-xml'], stdout=subprocess.PIPE)
```

[3]http://nmap.org/

```
qName = (etree.fromstring(qstat.communicate()[0])
    .xpath("/job_info/queue_info/Queue-List[job_list/JB_job_number/text()
    = '%s' ]/name/text()" % jobID))
```

The *qdel* command is used to release a reservation of a node, once the VMs it hosts are deleted or migrated away and the associated virtual host is deleted.

Similarly, the commands *onehost* and *onevm* used to create and manage virtual hosts and machines in OpenNebula can output formatted XML. This makes it possible to use pipes and *xpath* to communicate with OpenNebula in a similar fashion.

4.4 Resource Reservation (virtual host creation)

All tasks associated with creating a new virtual host are combined in a single routine - *createONEnode*. These tasks include reserving a node on the physical cluster, creating a virtual host in OpenNebula on top of that node and scheduling a *release handler* for the new host. This routine is used by the *watchdog*, the *release handler* and the UI when the creation of a new host is to be attempted.

After the reservation is submitted, the *createONEnode* routine waits for it to appear in the list of running jobs on the physical cluster. If the reservation is queued or if it is not run within a certain period of time, the routine timeouts and the caller is notified that no hosts are currently available. If the reservation is successful, the newly reserved physical node is identified and a virtual host is created in OpenNebula and added to our virtual cluster. The "deadline" to release this host is then calculated as $currentTime + jobTime$ and a *release handler* is scheduled to be run *releaseTime* seconds before the deadline. The routine then waits for the new virtual host to change its state from "INIT" to "RUNNING" and returns its name to the caller.

4.5 Resource Deallocation (virtual host deletion)

In order to make sure we don't exceed the duration of our reservations on the physical cluster, special care is taken to safely release resources while keeping our virtual cluster (and the applications running on it) alive. While the KVM processes (owned by the root user) may survive after a host node is released at the end of its reservation by SGE, the correctness of operation of ElViC can no longer be guaranteed. This is also violates the user policies on DAS-4 and may interfere with the work of other people. The virtual host for which the reservation on the physical cluster is expiring will be referred to as *expiring host*.

A *release handler* tries to migrate all VMs on the expiring host away and then deletes that host and the associated reservation. It handles the migration of each VM on that host individually. If the maximum allowed number of hosts is not currently exceeded (if the user has set a new, smaller m) the handler will attempt to create a virtual host and start migrating a VM to this host. If the user-set maximum number of hosts does not currently allow the handler to create more hosts or a reservation is denied during the creation of a new host, the VM is migrated to the host with the smallest number of VMs that has the most time remaining until its release. This causes the handler to spread out the VMs as evenly as possible and avoid migrating any VMs to a host that is about to be released soon. If a new host cannot be created and none of the currently available hosts have sufficient time left, a terminal condition is reached - all resources are deallocated and the applications proceeds to exit.

If all VMs on the expiring host are safely migrated away, it can be deleted together with its reservation. Since OpenNebula requires that no VMs can be present on a host that is to be deleted, the handler first waits for all migrating VMs to completely leave the host. This is done outside the critical section as it requires polling OpenNebula to check the current state until the condition that no VMs are left on the host is met.

The *release handler* is scheduled to release a host *releaseTime* seconds before its reservation expires. This is done during the creation of that host to ensure it is released on time. A *Timer* object in Python is used to schedule the handler.

It is important to note that, while in our implementation a VM is assumed to be able to safely migrate within some pre-defined period of time (*releaseTime*), efforts have been made ([Zhao and Figueiredo, 2007, Wu and Zhao, 2011, Akoush et al., 2010]) to develop models that predict the performance of live VM migration. For example, the *AVG* and *HIST* models described in [Akoush et al., 2010] can be used to accurately predict migration times. This can eliminate the need for hard-coding a *releaseTime* and make it possible to safely release the resources, while utilizing them better.

4.6 Termination

When the *exit* command is triggered, or the monitor application reaches a terminal condition (such as the inability to safely migrate a VM before the reservation for a host expires) all resources are released. All VMs are deleted, followed by the virtual hosts and the associated reservations are canceled. The application then proceeds to exit.

4.7 Adaptation to System Load

The number of physical nodes the monitor needs is defined by the *m* argument (and the corresponding command in the UI). During initialization and the release of virtual hosts, the monitor sometimes fails to obtain enough resources to create *m* hosts if the physical cluster is too busy. This causes the VMs to be evenly distributed between the available virtual hosts. A *watchdog* thread checks if the virtual hosts currently in use are fewer than *m* every second. If this is the case, it tries to create a new host and redistribute the VMs to be spread out as evenly as possible again. When a new host is created by the *watchdog*, the new *VMs-per-host* is calculated (as $\frac{|VMs|}{|hosts|}$) and the newly created host is populated with this number of VMs. It migrates VMs from the host having the most VMs until the condition is met.

The *release handler* always tries to create a new host if such is needed and resources become available during the VM migration. The adaptation the system load is essentially achieved by the interaction between the *watchdog* and the *release handler*.

4.8 Synchronization and Multithreading

A single lock is used to ensure only one party can change the state of the virtual cluster at a given time. The *watchdog*, the *release handler* and the UI thread all read and modify the state of the virtual cluster and therefore can only do that in their critical sections to avoid any race conditions.

The multithreaded design of the monitor allows for quick adaptation to the dynamically changing environment. For example, the commands of the user (e.g. shrinking or setting a new *m*) can take effect during (and not strictly after) the release of a large group of virtual hosts.

5 Experimental setup

In order to evaluate the impact of virtualization and VM migration on performance, a set of experiments is carefully designed and performed. The DAS-4 supercomputer that provides the infrastructure to run our system has a number of constraints and configuration specifics that need to be taken into account when designing these experiments. The physical nodes that host the VMs each have 24 GB of RAM and 8 physical CPU cores with hyperthreading enabled - 16 virtual CPU cores in total. While an Infiniband connection is available between the compute nodes on DAS-4,

the module that allows for its use on VM level is disabled for easier maintenance. This makes it necessary to use the slower 1Gbit Ethernet network both between VMs and also between physical hosts when performing experiments on bare metal for a more fair comparison.

The experiment design is discussed below, together with a more in-depth look into the various limitations and constraints in Section 5.4.

5.1 Latency and throughput measurements

Before any experiments with a real-life parallel application (or a benchmark that runs a large number of processes) are performed, the effects of virtualization and VM migration on round-trip latency and throughput need to be measured on a smaller scale. To this end, multiple latency and throughput tests were performed in the following setups:

Without virtualization (i.e. directly on compute nodes through SGE's reservation system):

- 2 processes on 1 cluster node
- 2 processes on 2 cluster nodes (one on each)

With virtualization:

- 2 processes (one per VM) on 1 virtual cluster node
- 2 processes (one per VM) on 2 virtual cluster nodes (one VM on each node)
- migrating VMs from 1 to 2 nodes and back while the measurements are being done

By comparing the results from the tests when no VM migration is done (on both the physical and virtual clusters) we aim to isolate and measure the effects of virtualization on performance. In the case when both processes are on the same physical node and no virtualization is used, the interprocess communication is done via shared memory. We expect this to have an impact on both latency and throughput, as in the virtualization scenario the communication is done over TCP (via virtual network interfaces) when the 2 machines reside on the same node. TCP is also used for transport in all other cases.

The following command is used in SGE jobfiles to force MPI to use TCP for transport:

```
$MPI_RUN --mca btl tcp,self,sm --mca btl_tcp_if_include br0
-np $totcores -hostfile $HOSTFILE $APP $ARGS
```

where "–mca btl tcp,self,sm –mca btl_tcp_if_include br0" is used to force the compute nodes to communicate with each other via TCP over the "br0" interface and use shared memory when interprocess communication is happening within the nodes. Unless shared memory is enabled, the loopback "lo" interface is used for interprocess communication on nodes - this is known to be very slow. The rest of the parameters are very commonly used when submitting jobs to SGE on DAS-4 and are, therefore, not discussed in detail.

A simple MPI latency benchmark was implemented in C and used to make the measurements. It measures round-trip latency by sending packets of size 1 byte as soon as possible, waiting for an acknowledgement and then recording the measured time difference and the associated timestamp.

A point-to-point bandwidth test from the OSU (Ohio State University) microbenchmark suite[4] was used to measure the effects of virtualization on bandwidth/throughput on bare metal (no virtualization) and in the virtual cluster.

[4] http://mvapich.cse.ohio-state.edu/benchmarks/

5.2 Parallel benchmarks

In order to estimate how a real-life application will be affected when run on our elastic virtual cluster, a set of benchmarks needs to be selected and carefully configured. The NPB (NAS Parallel Benchmarks) suite is a popular choice for testing the performance of parallel supercomputers. It contains several applications, written in multiple languages, that provide a wide choice of communication/memory access patterns. The user can compile any of the benchmarks to use a specific number of processes and input size. This allows for many degrees of freedom in the experiment design - using different classes of applications (e.g. ones that vary in their communication patterns), as well as different settings for each.

Running the benchmarks on a virtual cluster opens up a new dimension of parameters to experiment with. The ones of particular interest are the number of VMs per host, the number of MPI processes running on a VM, the amount of memory and the number of virtual CPU-s (*VCPU*-s) allocated to each VM.

On the highest level, there are parameters to the virtual cluster monitor that can be subject to experimentation, such as the time during execution at which a migration event (e.g. "growing" from N to $2N$ physical host nodes) is triggered.

Ultimately, the goal is to examine a set of benchmarks of different classes, find the parameters that provide the optimal performance for each, compare this with the results from running the benchmarks directly on the physical nodes and identify the type of applications most positively affected by VM migration.

EP (embarrassingly parallel), *CG* (conjugate gradient) and *MG* (multi-grid on a sequence of meshes) are good candidates to form this set - *EP*, as the name suggests, being not very communication-intensive and *CG* and *MG* being the opposite.

benchmark	**type**	**num.**	**volume**
EP	collective	5	12.5KB
	point-to-point	0	0
CG	collective	1	0.03B
	point-to-point	47104	2.24GB
MG	collective	100	126KB
	point-to-point	11024	384MB
IS	collective	33	348MB
	point-to-point	15	0.06KB

Table 1: Dynamic measurement for collective and point-to-point communications in EP, CG, MG and IS NPB benchmarks [Faraj and Yuan, 2002]

Table 5.2 shows the number of collective and point-to-point communications and their associated volume for the EP, CG, MG and IS NPB benchmarks, compiled for 16 processes and problem size A [Faraj and Yuan, 2002]. This can be used as a guideline to the amount of computation each benchmark is doing.

Table 2 shows a set of parameters used in our experiments. Note that the results for only a small subset of these (focusing mostly on EP) are presented in Section 6.

5.3 Coarse-grained matrix multiplication

In addition to the benchmarks from NPB, matrix multiplication using the PDGEMM and PSGEMM routines from ScaLAPACK/PBLAS ([Choi et al., 1996]) was performed to evaluate the effects of virtualization and live-migration of coarse-grained parallel applications.

parameters	values
benchmark	ep, cg, mg, is
problem size	B, C, D, E
processes	8. 16. 32. 64, 128, 256
physical hosts	1, 2, 4, 8, 16, 32, 64
VCPU	1, 2, 4, 8, 16
RAM per VM	300 ... 3000
processes per VM	1, 2, 4, 8, 16

Table 2: parameters for experiments with parallel benchmarks

PDGEMM is a level 3 matrix multiplication routine for block cyclic data distribution included in PBLAS, the parallel implementation of BLAS (Basic Linear Algebra Subprograms). It is based on the DIMMA ([Choi, 1997]) algorithm. The algorithm requires $O(N^3)$ flops and $O(N^2)$ communications - it is computationally intensive but also performs a fair amount of communication.

A PDGEMM example program in Fortran provided by ScaLAPACK was adapted to use as a benchmark in the experiments. It generates random matrices and allows the user to set the matrix size, block size and the dimensions of the process grid on a per-run basis. This allows for a lot more freedom than in the NPB benchmarks, where the problem sizes (A, B ... E) are pre-defined. The PSGEMM and PDGEMM benchmarks both perform the matrix multiplication as $C := \alpha \times op(A) \times op(B) + \beta \times C$ (where $op(A)$ is defined as A or its transpose) but differ in the datatype used - REAL being used in PSGEMM and DOUBLE PRECISION being used in PSGEMM. PSGEMM is, therefore, expected to use less memory and less bandwidth than PDGEMM. In order to make the application run for a longer time, the PSGEMM/PDGEMM routine was called multiple times in some of the experiments.

5.4 Limitations and constraints

Configuring the parameters of both the parallel application and the virtual cluster is not a trivial task. In case the amount of memory made available to a VM in the virtual cluster is exceeded by the cumulative amount of memory required by the application processes being run within it, performance is significantly degraded due to swapping on the VM level. In the OpenNebula setup used for the experiments, swapping within VMs results in a very high NFS activity on the head node of DAS-4, as the VM disk images are mounted over NFS. This not only results in a failed experiment, but also negatively affects the work of other DAS-4 users and should certainly be avoided. Additionally, if the cumulative amount of memory allocated to VMs on a host exceed the amount of physical memory available to that host, swapping on the host level results in degraded performance, as well. While the memory contention in the latter scenario can be helped by migrating VMs away to new hosts, the effects of swapping on the host level are so detrimental that it makes starting the computation on a set of memory-contended hosts pointless. Since swapping, both on the VM and host levels, is to be avoided, all experiments are planned to always satisfy the memory requirements of both VMs and hosts.

Since some performance penalty is expected during migration and this can possibly be compensated by the speedup achieved when the VMs are migrated to more hosts, the duration of the migration process (as in migrating a group of VMs to achieve a new distribution on hosts) should be short relative to the runtime of the parallel application. In other words, the longer the runtime and the shorter the time it takes to get from $N/2$ to N physical nodes, the less of an impact migration is expected to have. Assuming physical resources are available and can be reserved instantaneously, the time it takes to grow the virtual cluster and spread out VMs evenly is

mostly determined by multiple factors - the number of VMs and the amount of memory allocated to each VM. A third important factor that can greatly influence the migration time is the size of the writable working set - more memory-intensive applications can take much longer to reach the stop-and-copy phase in pre-copy migration.

Naturally, the fewer the VMs are, the fewer individual migration operations will take place. This hints towards running several processes per VM in order to limit the total number of VMs. However, having more processes per VM would (for most parallel applications) require more memory to be allocated per VM. The preliminary tests of the experimental setup reveal that migrating VMs too quickly (as in starting too many individual migrations at the same time) results in many failed migrations - a VM enters the *migr* state in OpenNebula and the copying to the destination host begins, but soon after migration fails and the VM continues to run on the source host. Whether this is OpenNebula or Qemu/KVM related should be further investigated in the future. In order to get around this problem, the application should wait for a fixed amount of time between individual VM migrations. Another trend revealed in the preliminary tests is that when more memory is allocated per VMs, the waiting time should be increased as well in order to prevent migrations from failing.

The experiments are designed to always use at least 100% of the available CPU resources on a physical host - typically going from 200% to 100% when migrating from $N/2$ to N hosts. Since the physical nodes on DAS-4 used in the experiment each have 8 cores (16 with hyperthreading), parallel applications running at least 16 processes are required to adequately exploit migration to decrease the CPU contention but yet use at least 100% in the new configuration. This translates to having no fewer than 8 processes run per host, regardless of the number of VMs per host - the number of virtual CPUs (VCPUs) per VM can be set accordingly. If hyperthreading is to be fully employed in the base (100%) scenario, at least 32 processes, no fewer than 16 per host are needed. As the processes in VMs are managed by the scheduler on the host, some overhead is expected due to CPU overcommitting if too many processes are being run per host [Huber et al., 2011, Ranadive et al., 2008].

The time at which migration occurs is expected to seriously affect performance in our system. While this can be dependent on the application, its computation and communication patterns, assuming the workload is evenly distributed, both temporally and spatially, the sooner VMs are migrated to more nodes - the sooner computing is expected to speed-up if the application is CPU-bounded.

In summary, applications that run for longer and are migrated sooner (and in a shorter time) are expected to show better results. This, however, applies mostly to embarrassingly parallel and coarse-grained parallel applications, for which migration done with the aim to lower the level of CPU contention is not likely to cause much communication overhead during and after migration. The applications run in our virtual cluster have to be flexible enough to easily adapt to be run on 16, 32, 64 and more processes, and use such amount of memory per process as to allow for a placement of processes per VM and VMs per host, whereby swapping is avoided both on hosts and in VMs. The total number of VMs and the number of processes per VM should be balanced to provide optimal performance benefits while keeping the duration/cost of migration low. Parameters should be such that at least 100% of the CPU resources are used but special care has to be taken not to cause overhead due to CPU overcommitting.

6 Results and Discussion

In this section the results from the experiments designed in Section 5 are shown and described. Some difficulties and anomalies encountered during these experements are also mentioned in the section below.

6.1 Anomalies and technical difficulties

Many technical problems were encountered during the experiments. Apart from the individual VM migrations failing that was discussed in Section 5.4, a variety of other problems related to OpenNebula and Qemu/KVM were observed.

Deleting VMs using the standard *onevm delete* command removes the records from the list of running VMs in OpenNebula but the associated KVM processes are sometimes left running. This is problematic because, since these processes are owned by the root user, they are kept alive after the host node is released. These processes can thus interfere with the work of other users - they might end up reserving a node that hosts several KVM processes, that they are unable to kill. A setuid wrapper that gives the monitor application administrative rights to kill the KVM processes during termination (Section 4.6) was used as a work-around.

The OpenNebula installation on DAS-4 was often becoming unresponsive - running the commands used to create/delete hosts and VMs and show their status resulted in timeouts. This made it necessary to often contact the administrator to restart the OpenNebula daemon.

Some unusual spikes in round-trip latency were observed during VM migration. They are outlined in Section 6.2.

6.2 Latency and throughput

	BM	**VC**
same host / separate VMs	1	68
same host / same VM	n/a	3
separate hosts / separate VMs	117	182

Table 3: Round-trip latency (in microseconds) between two processes - running on the same and two separate hosts on bare metal (BM) and in the virtual cluster (VC), and in the same and separate VMs on VC

Table 3 shows the results from the round-trip latency test run on bare metal and on a virtual cluster. Since the compute nodes used in the bare metal test were configured to use shared memory, the results from having both processes on the same node without virtualization are best - the latency is 1μs. Communication within the same VM seems to be almost as fast (3μs). Since TCP over Ethernet is used as the transport between compute nodes in the experiment, having the test run on two separate physical nodes shows a sudden increase in latency - from 1μs with both processes on the same host to 117μs. The results from running the latency test in VMs, except for the case when both processes are run in the same VM, reveal that even when no high-speed interconnects are used in the bare metal case, latency is negatively influenced by virtualization alone. Running the test on separate VMs, both on the same and separate hosts show a higher latency than bare metal - 68μs to 1μs and 182μs to 117μs.

	BM	**VC**
same host / separate VMs	4780.84	1211.86
same host / same VM		4391.02
separate hosts / separate VMs	117.64	117.62

Table 4: Bandwidth measured using osu microbenchmark in MB/s, bytes sent = 4194304 (4.1943 MB) on bare metal (BM) and in the virtual cluster (VC)

A similar trend can be seen in the latency/throughput measurements. Virtualization alone adds

a lot of communication overhead - the speed decreases from 4780.84 (bare metal) to 1211.86 (virtual cluster) MB/s when the processes are run on the same host in two separate VMs. Running the processes in the same VM on the same host, as was seen in the latency test, shows a great improvement over the separate VMs scenario. The resulting 4391.02 MB/s get very close to the 4780.84 MB/s on bare metal with shared memory. Interestingly, when running the experiment on separate hosts, virtualization does not seem to have a big effect - both results (from bare metal and virtual cluster) are close to 118 MB/s.

These findings suggest that when using the current setup, one should ideally group processes that communicate more often with each other together on the same VM. Parallel applications whereby intensive communication happens within such groups but not across groups can potentially benefit from such a placement.

Nahanni[5] is a mechanism for sharing host memory with the VMs running on a host. It enables the use of shared memory for communication between KVM VMs running on the same host and can potentially be used to speed up the communication between processes that reside on the same host but are isolated in separate VMs.

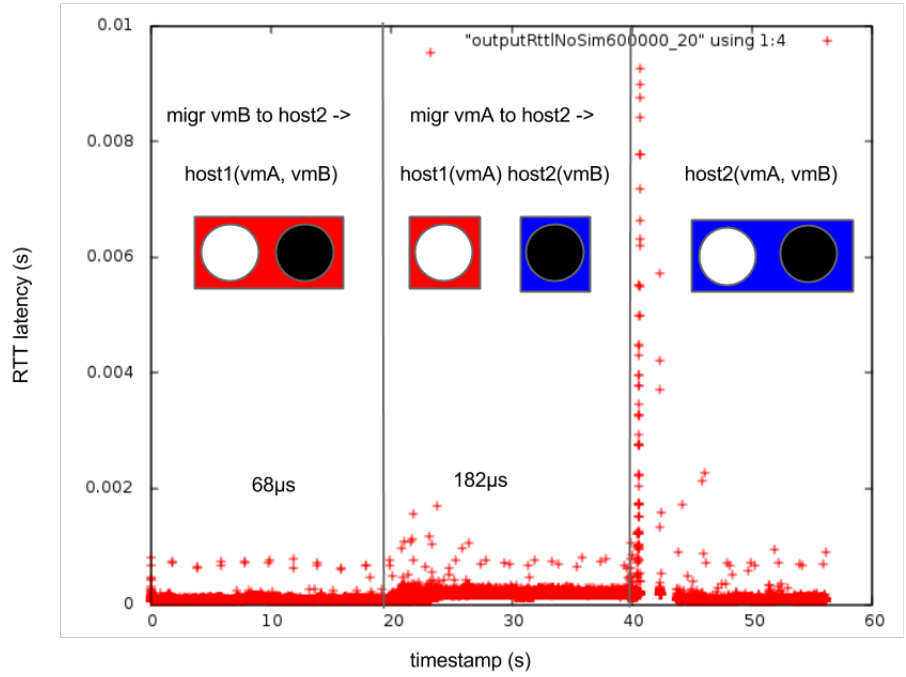

Figure 2: Round-trip latency during live migration of two VMs, zoomed in and with annotations. Vertical lines point to migration events, average latency is shown above data points.

Fig. 2 shows the effects of pre-copy live migration on latency in real time. On the left hand size, the latency (reported as often as possible) between two processes residing on the same physical host, in separate VMs is shown. The two rectangles (red and blue) shown above the data points represent the two hosts and the two circles (white and black) represent the two VMs. The vertical lines designate two migration events - the first one is the migration of one of the VMs to a new host after 20 seconds. Accordingly, the latency increases (from 68 to 182 μs) and stays higher throughout the central section, while the VMs are on two separate physical hosts. The other

[5]https://gitorious.org/nahanni

VM is then also migrated on the new host after another 20 seconds - the event designated by the second vertical line. The second migration causes a steep linear increase in the round-trip latency, followed by a spike (excluded in this figure). As the VMs are now again on the same host, the latency drops back to the range observed in the left-hand side of the figure. Since the timestamp of each event (message sent and acknowledgement received) is buffered as soon as it is measured and this data is only reported once at the end, after the measurements have taken place, reporting has a negligible effect on the experiment.

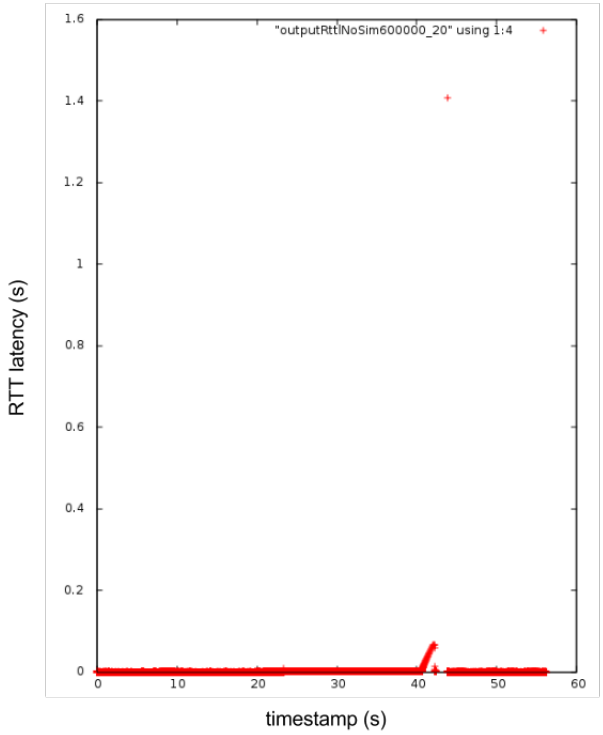

Figure 3: Round-trip latency during live migration of two VMs

Fig. 3 shows the results from the same experiment including the outlier value which is observed at the second migration event. Visual aids have been omitted. Such outliers, preceded by a linear increase in latency, were consistently observed at every second migration event throughout our experiments with OpenNebula and Qemu. This behavior was also observed when using ping. This shows that it is not specific to MPI or the latency-measuring application. The traffic between the hosts and VMs during these events was captured and analyzed using tcpdump and Wireshark. The analysis revealed the problem is caused by a timeout due to a delayed RARP response, used by Qemu to announce the new physical location of the VM.

An excerpt from a packet capture during one of these events is shown below.

80.55 migration starts, handshake between host nodes
80.76 2x ARP: who has IP of migrated VM?
81.66 last packet between VMs before RARP
81.76 ARP: who has IP of migrated VM?
traffic between VMs ceases, traffic only between hosts
82.91 RARP
82.91 FIN, ACK - migration ends, hosts close the connection

82.96 RARP
83.11 RARP
83.14 traffic between VMs resumes

This can be traced back to a known bug[6] in the version of Qemu used and can possibly be avoided by using a newer version. While this could have a serious effect on all experiments, the infrastructure (OpenNebula and Qemu setup) was provided and managed by the university and changes of that kind were considered out of the scope of this study.

Obviously, this can have a very negative impact on scenarios that involve the migration of many VMs.

6.3 Parallel benchmarks

The results from running parallel benchmarks from the NPB suite on our virtual cluster, with and without migration are presented below. They are compared to the results from running the same benchmarks directly through the reservation system on DAS-4, without virtualization - bare metal (BM). The compute nodes were configured to use TCP over Ethernet in these experiments.

The following color-coding was used for consistency and to make it easier to relate the numerical data in Table 5 to the histograms in Fig. 4, 5 and 6: The color Blue is associated with the results from running the benchmarks on the virtual cluster, lighter blue corresponding to computing on more physical hosts, whereas dark-blue corresponds to using fewer hosts. Similarly, bare-metal results are represented by the color Red, with lighter red indicating the use of more compute nodes and dark-red that of fewer. The green bars on the histograms and cells in tables correspond to the results obtained from growing the virtual cluster and migrating the VMs from fewer to more hosts during the computation. The number of hosts "from" and "to" which the VMs are migrated stays consistent throughout each experiment and uniquely identifies each experiment, together with the application, problem size, number of processes and the time of migration.

application	VMs	hosts	BM	VC	VC + migration
ep.E.32	32	4	3471	3552	3613
ep.E.32	32	2	4131	5221	
ep.E.256	64	16	529	598	752
ep.E.256	64	8	1119	1305	
ep.D.64	64	8	94	115	133
ep.D.64	64	4	141	164	211
ep.D.64	64	2	261	369	
ep.D.32	32	4	186	230	251
ep.D.32	32	2	262	329	

Table 5: Execution time (in seconds) of EP benchmark run on bare metal (BM), virtual cluster (VC) with and without migration.

Table 5 displays the results from running the NPB EP benchmark on bare metal and in a virtual cluster, with migration from N to $2N$ physical nodes shown in green in the rightmost column. The times at which migration was performed varies across experiments and so does the time waited between individual VM migrations. The first two rows of the table (ep.E.32) refer to migrating the EP benchmark compiled for 32 processes and problem size E from 2 to 4 physical hosts (going from 16 to 8 VMs per host) and also show the reference results from running the benchmark on 2

[6] https://lists.gnu.org/archive/html/qemu-devel/2009-10/msg01457.html

and 4 physical hosts on bare metal and in the virtual cluster, without virtualization. These results are also shown as a histogram on Fig. 5. Similarly, rows 3 and 4 and Fig. 6 refer to migrating EP compiled for 256 processes with problem size D from 8 to 16 hosts with 4 processes per VM (8 to 4 VMs, 32 to 16 processes per host).The next 3 experiments - ep.D.64 migrated from 2 to 4 and 4 to 8 hosts and ep.D.32 migrated from 2 to 4 hosts with 1 process per VM are shown at the bottom and the respective histograms are grouped together in Fig. 4. Migration in the experiments shown in Fig. 4 (ep.D.64 and ep.D.32) is done 1 minute after the benchmark is run and 1 VCPU is used per VM.

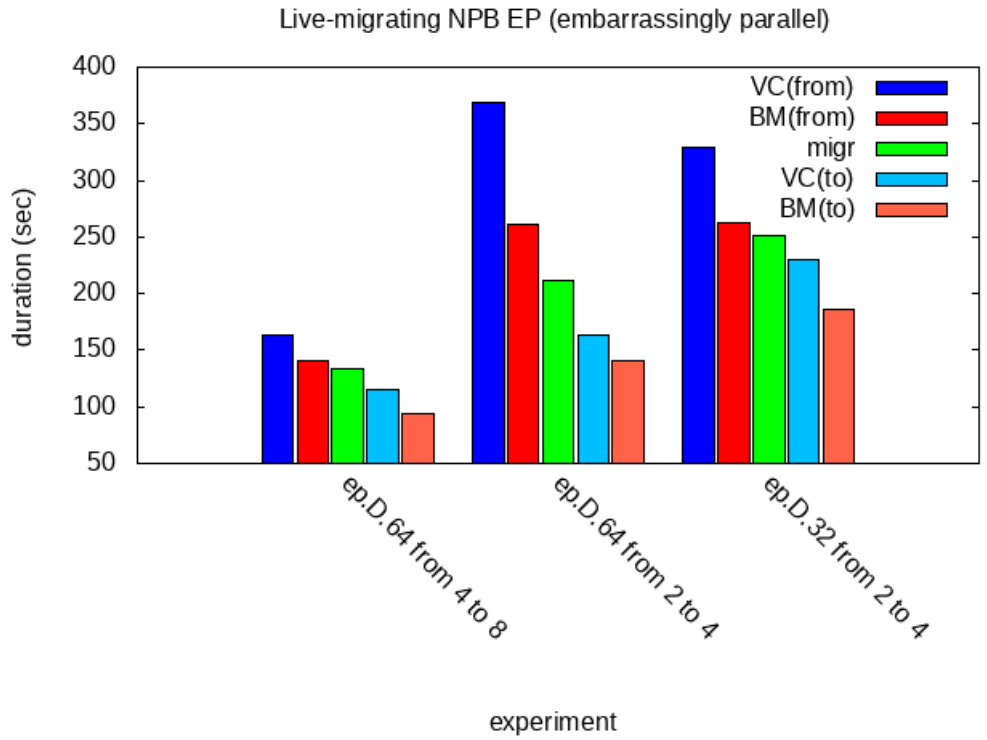

Figure 4: Execution time of EP benchmark on BM (red) and VC (blue), with (green) or without migration

From the reference values for bare metal (red) and the virtual cluster without migration (blue) on Fig. 4 it appears that computing on more hosts is always faster. The difference, both on bare metal and in the virtual cluster, seems most prominent in ep.D.64 when run on 2 and 4 hosts - the number of processes per host is reduced from 32 to 16. In the other two, the number of processes is reduced from 16 to 8 - essentially, hyperthreading is not used when the benchmark is run on more hosts. This was done to also measure if performance by migrating VMs away to use as many processes per host as there are physical cores, as opposed to 2 per physical core with hyperthreading.

From the adjacent blue and red bars (both dark and light), it is evident that there is always some loss of performance due to virtualization alone. The blue bar, corresponding to the benchmark being run in the virtual cluster is always higher than its adjacent red one, corresponding to the same benchmark with the same configuration, only run on bare metal. Migrating from N to $2N$ nodes (green bar and cell) always results in a shorter runtime than bare metal (light red) on N nodes and longer than both bare metal (dark red) and the virtual cluster (dark blue).

If the results on a less-contended set of hosts on bare metal are taken as a reference, a 42%

speed-up is achieved when ep.D.64 is migrated from 2 to 4 hosts after 1 minute. The other two experiments shown on that figure (ep.D.64 from 4 to 8 and ep.D.32 from 2 to 4) show a smaller improvement.

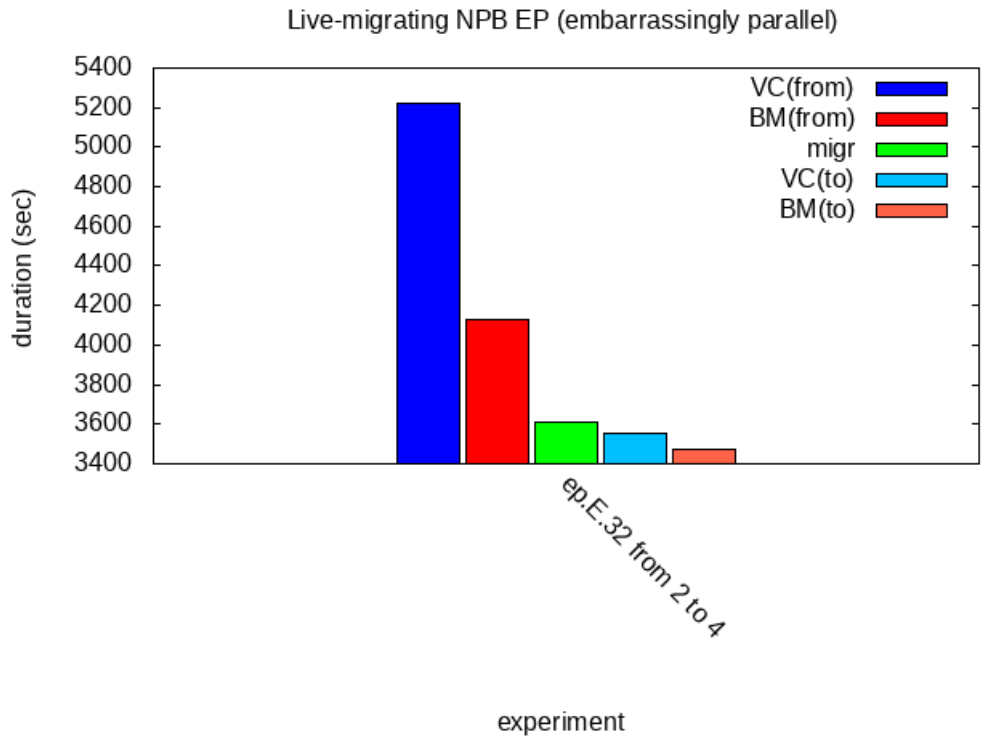

Figure 5: Execution time of EP (size E, 32 processes) benchmark on BM (red) and VC (blue), with (green) or without migration

Fig. 5 shows the results from migrating ep.E.32 from 2 to 4 hosts with 1 process per VM - going from 16 to 8 processes per host. The general observations from Fig. 4 about the overhead due to virtualization and the worse performance when computing on fewer nodes also hold here. Running the benchmark on the more saturated set of hosts (2) in the virtual cluster results takes 129% longer than if run on bare metal. This difference is 102% on the less saturated set of hosts (4). It takes the benchmark 147% longer to finish when run on fewer nodes in the virtual cluster alone and 119% when run on bare metal.

Since migration was done much earlier relative to the total runtime (after 100 seconds, with the benchmark running for a total of 5200 on VC and 4131 on BM), the result is very close to that of running the benchmark on the virtual cluster on 4 nodes - a speed-up of 79% is achieved taking the results from bare metal on 4 nodes as the baseline.

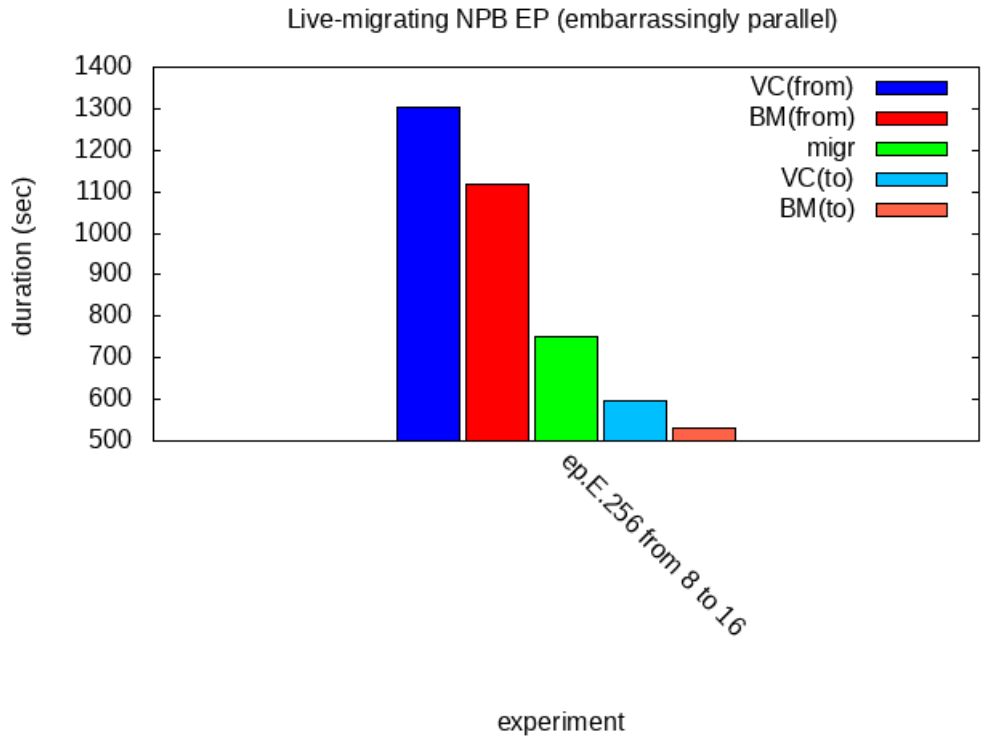

Figure 6: Execution time of EP (size E, 256 processes) benchmark on BM (red) and VC (blue), with (green) or without migration

Fig. 6 shows the results from migrating ep.E.256 from 8 to 16 hosts with 4 processes per VM - going from 64 to 32 processes per host. Migration is also done 100 seconds after the benchmark is run. Here the difference between the runtimes for BM and VC (dark blue and red) is not as pronounced as in Fig. 5. What is especially interesting is that the difference between the results on the more-contended set of hosts (dark blue and red) is relatively small, even though the duration of the experiment is long - running the benchmark in the virtual cluster on 8 nodes is 116% slower than on bare metal. The speed-up achieved by migration is 63%.

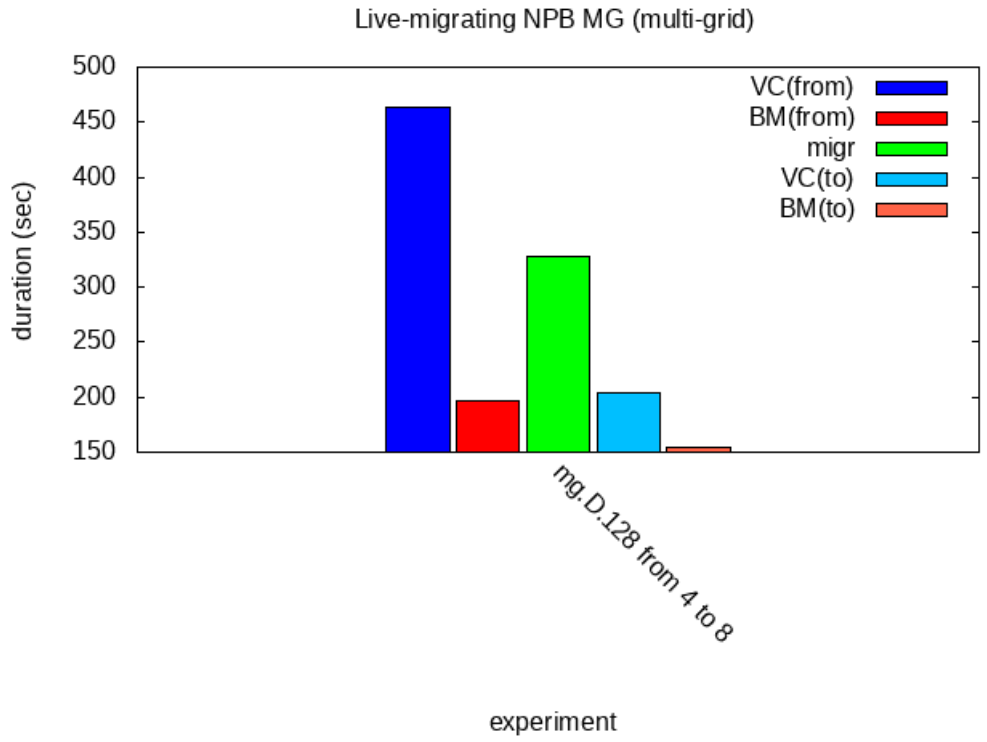

Figure 7: Execution time of MG (size D, 128 processes) benchmark on BM (red) and VC (blue), with (green) or without migration

Fig. 7 shows the results from migrating the NPB MG (multi-grid) benchmark, compiled for 128 processes and a problem size D, from 4 to 8 hosts - going from 32 to 16 processes per host. A total of 16 VMs were used with 8 processes per VM - there were 4 VMs per host on the more-saturated set of hosts (4) and 2 on the less-saturated (8). It appears that the performance on the virtual cluster is much worse than on bare metal - having the benchmark run until completion on 8 hosts in the virtual cluster already takes much longer than running it on 4 hosts on bare metal. Running the benchmark on 4 hosts in the virtual cluster takes twice as long as on bare metal. Even though migration was done at the very beginning, just 30 seconds after the benchmark was run, the speed-up provides no real benefit as it does not even between the reference values (dark and light red) for bare metal, on 4 and 8 hosts.

6.4 Matrix multiplication

The results from running the PDGEMM and PSGEMM-based matrix multiplication programs are presented below. The color coding in the tables and figures is consistent with the one described in Section 6.3.

6.4.1 PDGEMM

Table 6 and Fig. 8 show the results from running PDGEMM with two square matrices of size 28000^2 on a 64-process grid (8x8) and a block size of 3500. The block size (3500) was chosen because it allows the program to run for a long time (37 minutes on two hosts, 32 processes/VMs each on bare metal) and allows for a VM placement and memory allocation such as to avoid swapping both on the host and VM levels. PDGEMMx2 refers to a run of the program, whereby the PDGEMM routine is called twice, with everything else left intact - e.g. generating the matrices

is done only once. This was done to let the program compute for a longer time, while keeping its memory consumption the same. Migration was done 5 minutes after the run.

application	VMs	hosts	BM	VC	VC + migration
PDGEMM	64	4	16	20	29
PDGEMM	64	2	37	52	
PDGEMMx2	64	4	32	38	50
PDGEMMx2	64	2	74	114	

Table 6: Execution time (in minutes) of PDGEMM test run on bare metal (BM), virtual cluster (VC) with and without migration.

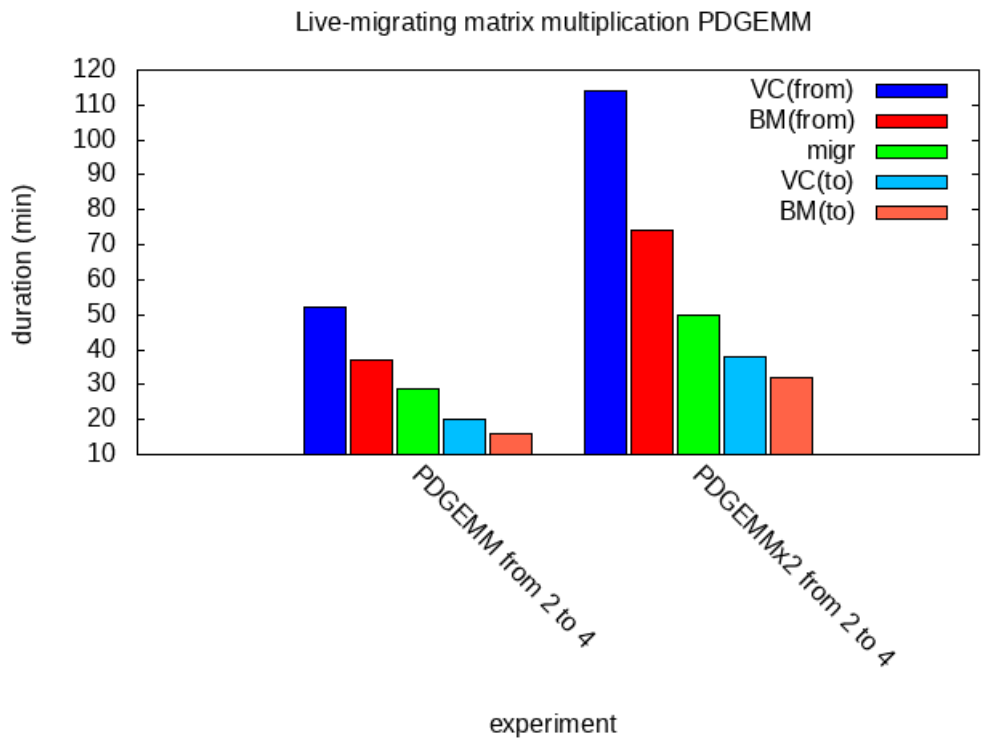

Figure 8: Execution time of PDGEMM test on BM (red) and VC (blue), with (green) or without migration

What appears from Fig. 8 is, again that virtualization slows down the application, with the gap between the performance on bare metal and in the virtual cluster growing bigger as the hosts get more saturated. In addition, the figure shows that running the PDGEMM routine twice causes the overall duration on the virtual cluster to grow even faster - the bare metal tests (on both 2 and 4 hosts) take 200% longer when PDGEMM is run twice, while on the virtual cluster we get a 190% increase in duration on 4 hosts and 219% on 2. When PDGEMM is run once and migrated from 2 to 4 hosts 5 minutes after it was run (left hand side) a speed-up of 39% is achieved. The speed-up when PDGEMM was run twice and migrated from 2 to 4 hosts after 5 minutes (right hand side) is 57%.

6.4.2 PSGEMM

Table 7 and Fig. 9 show the results from running PSGEMM in exactly the same setup as for PDGEMM in Section 6.4.1.

application	VMs	hosts	BM	VC	VC + migration
PSGEMM	64	4	15	18	24
PSGEMM	64	2	36	45	
PSGEMMx2	64	4	29	35	45
PSGEMMx2	64	2	59	90	

Table 7: Execution time (in minutes) of PSGEMM test run on bare metal (BM), virtual cluster (VC) with and without migration.

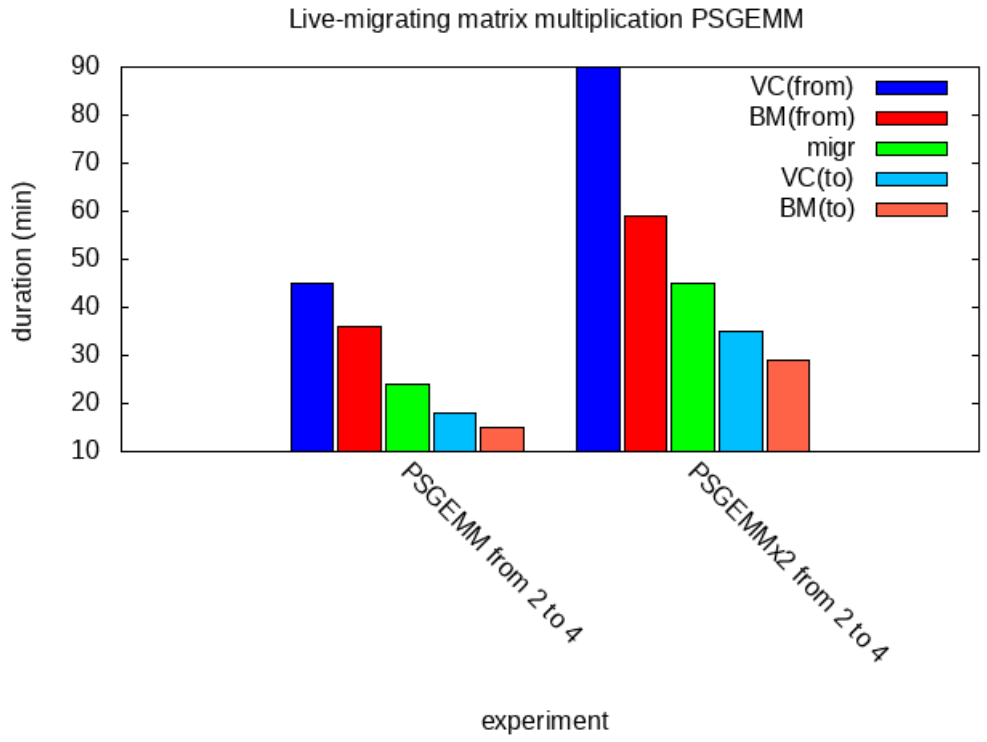

Figure 9: Execution time of PSGEMM test on BM (red) and VC (blue), with (green) or without migration

The results from PSGEMM in Fig. 9 follow a similar trend to these from PDGEMM in Fig. 8, only the scale being slightly smaller. This is due to PSGEMM using a datatype of smaller size and therefore using less memory, bandwidth and CPU than PDGEMM. The performance on the virtual cluster also degrades faster when the PSGEMM routine is run twice. The speedup achieved by migration after 5 minutes when PSGEMM is run once is 57% and if run twice - 47%.

Table 8 and Fig. 10 below show the results from running both PDGEMM and PSGEMM with two square matrices of size 56000^2 on a 256-process grid (16x16) and block size of 3500, on 16 and 8 hosts and 4 processes per VM on the virtual cluster. The block size was kept the same as in the PDGEMM and PSGEMM experiments above. The dimensions of the matrices were doubled to allow for having a 16x16 grid. This was done to keep the memory requirements and load per process the same, while increasing the number of processes and allowing for more options to group

the processes together in VMs. The granularity is also increased as and more communication is expected to take place as a result of this. The more-contended set of hosts (8) has 8 VMs per host (32 processes) and the less-contended (16) has 4 VMs per host (16 processes). Migration was done after 5 minutes in both scenarios.

application	VMs	hosts	BM	VC	VC + migration
PDGEMM	256	16	34	54	68
PDGEMM	256	8	77	126	
PSGEMM	256	16	31	47	57
PSGEMM	256	8	73	113	

Table 8: Execution time (in minutes) and comparison of PDGEMM and PSGEMM tests run on bare metal (BM), virtual cluster (VC) with and without migration.

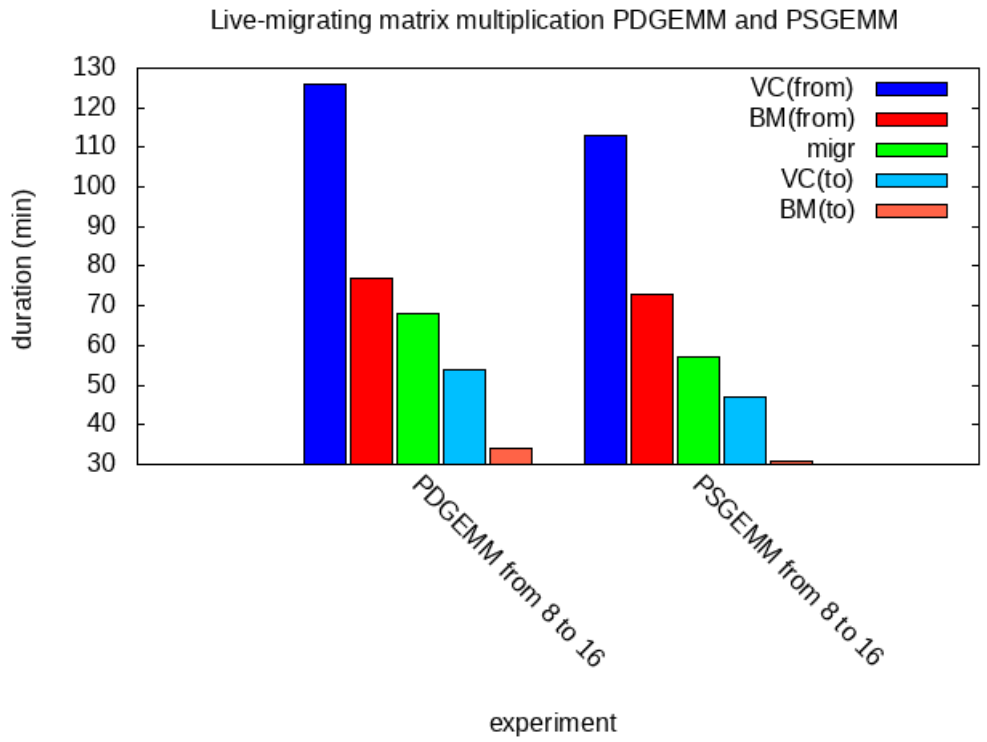

Figure 10: Execution time and comparison of PSGEMM and PDGEMM tests on BM (red) and VC (blue), with (green) or without migration

What can be noticed on Fig. 10 is the large difference in the runtime achieved on bare metal and in a virtual cluster when running both applications, especially on the set of less saturated hosts (16 hosts running 16 processes each). This can be explained by the use of shared memory on the compute nodes - since the application in that setting is more fine-grained than in the previous experiments with PDGEMM and PSGEMM (Fig. 8, 9) and is expected to communicate more, shared memory makes the communication between processes situated on the same host much faster. It takes around 150% (159% for PDGEMM and 152% for PSGEMM) longer to run the applications to completion in the virtual cluster in this case. The speed-up achieved by migration is 21% for PDGEMM and 30% for PSGEMM. Although the results from bare metal do not differ much between PDGEMM and PSGEMM here, the effects of running the application in a virtual

cluster are more negative in the PDGEMM case. In addition, migration shows a greater benefit when done for PSGEMM. This can be explained by the larger amount of bandwidth used by PDGEMM as explained in Section 5.3.

To summarize the findings from the analysis above, live migration of VMs can provide a reasonable speed-up (in several cases 57%, 63%, and 79% if performance on bare metal is used as a baseline) in both embarrassingly parallel and coarse-grained parallel GEMM-based matrix multiplication applications, if done shortly after the computation begins. While, in theory, one can achieve more of a speed-up if migrating even earlier, such a scenario would likely be unrealistic as the possibility/need to scale will probably not arise immediately after the program is started.

Virtualization was shown to always negatively affect performance when compared to bare metal, even when embarrassingly parallel application are run on the virtual cluster. This degradation seems to get worse with time as shown in the individual PDGEMM and PSGEMM experiments and in the case for EP seemed to be helped by grouping more processes into fewer VMs as on Fig. 6. More bandwidth-intensive applications seem to be slightly more negatively affected by virtualization and to benefit less from migration when compared to the results when a datatype of smaller size is used to reduce communication overhead - Fig. 10. This is consistent with the measurements done on latency and bandwidth in Section 6.2.

More communication-intensive applications such as MG were shown to be too badly affected by virtualization alone to benefit from being migrated to a less saturated set of hosts.

An improvement in performance through VM migration was observed both when 16 and 8 (no hyperthreading) processes were used on the less-saturated set of hosts.

7 Conclusions

ElViC, a multithreaded application that transparently manages physical and virtual resources to achieve malleability for MPI (and potentially other non-elastic parallel) applications was implemented in Python. It makes it possible for MPI applications to adapt to the load on a compute cluster by dynamically reserving and adding idle compute nodes as hosts in a virtual cluster as they become available, and releasing them if they are needed for another task or the job queue on the cluster is full. ElViC was used as a testbed for a series of experiments that measured both the effects on performance of virtualization alone, when compared to performance on bare metal, and that of live-migration in a virtual cluster compared to results from bare metal and virtual cluster without migration.

Scaling up computation-intensive only applications to more compute nodes via VM migration was shown to have a significant positive impact on performance if done early during the run. The effects of virtualization alone, especially on more saturated hosts bring much overhead, that is hard to be balanced out if migration is done later. The approach was not found suitable for more-communication intensive applications, such as NPB MG.

8 Future work

In our experiments TCP over Ethernet was used as the transport, both in the virtual cluster and on bare metal. Patches for KVM and OpenNebula exist that allow for using high-speed interconnects (e.g. Infiniband) in a virtual cluster, if the underlying physical infrastructure supports it. This is made possible by SR-IOV. While the hardware on DAS-4 supports SR-IOV, certain kernel settings on the compute nodes need to be changed in order to enable this feature. In theory, this should make it possible to repeat the same experiments using Infiniband for transport in the future.

Some of our experiments showed a degradation in performance when too many virtual CPUs were used in the virtual machines and the total number of virtual cores starts to exceed some

threshold value. While this is to be expected, it is a limitation in scenarios such as having 1 VM per host with 16 virtual CPUS (equal to physical cores with hyperthreading) on N hosts and then shrinking the virtual cluster to $N/4$ or $N/8$ physical hosts. [Shao et al., 2011] investigated the reasons behind the performance penalty of running MPI applications in overcommited virtualized systems and take multiple approaches to minimize that. They take the approach of exposing scheduling information of the guest OS to the hypervisor and wrote a patch for the XEN hypervisor and mpich2 that makes this possible.

In addition to this, there are patches for OpenNebula and KVM that allow for CPU ballooning - i.e. dynamically changing the number of virtual CPUs in a virtual machine at runtime. This, however, may pose serious limitations on the possibility to live-migrate virtual machines.

This study only focused on using KVM as a hypervisor in our virtual cluster. While other studies suggest that the difference in using different hypervisors may not be significant, they do not focus on migration, in particular. Using a hypervisor such as Palacios, which promises better performance in fine-grained parallel applications, might provide some interesting results in a migration context.

Post-copy migration was not explored in this study. *Yabusame* is a lightweight extension for Qemu/KVM that allows for post-copy migration. [Hirofuchi et al., 2010] first proposed the prototype for this and made the implementation publicly available on github. In [Hirofuchi et al., 2012] their approach for consolidating virtual machines with post-copy live migration was shown to greatly contribute to improving performance assurance of IaaS data centers.

Finally, models can be developed that predict the optimal time to migrate an application if a certain degree of speed-up is desired.

References

[Agrawal and Pateriya, 2013] Agrawal, N. and Pateriya, R. (2013). A Survey Of Pre-Copy Live Migration Techniques In Virtual Network Environment. *International Journal of Engineering Research and Technology*, 2(4):2202–2207.

[Akoush et al., 2010] Akoush, S., Sohan, R., and Rice, A. (2010). Predicting the performance of virtual machine migration. *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 37–46.

[Cera and Georgiou, 2009] Cera, M. C. and Georgiou, Y. (2009). Supporting MPI malleable applications upon the OAR resource manager. *Colibri: Colloque d'Informatique: Bresil/INRIA, Cooperations, Avancees et Defis*, pages 2399–2402.

[Chierici and Veraldi, 2010] Chierici, A. and Veraldi, R. (2010). A quantitative comparison between xen and kvm. *Journal of Physics: Conference Series*, 219(4):042005.

[Choi, 1997] Choi, J. (1997). A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. *High Performance Computing on the Information Superhighway, 1997. HPC Asia'97*.

[Choi et al., 1996] Choi, J., Demmel, J., and Dhillon, I. (1996). ScaLAPACK: A portable linear algebra library for distributed memory computersDesign issues and performance. *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*.

[Clark et al., 2005] Clark, C., Fraser, K., Hand, S., and Hansen, J. (2005). Live migration of virtual machines. *Proceedings of the 2nd conference on Symposium on Networked Systems Design \& Implementation-Volume 2*, pages 273–286.

[Desell et al., 2006] Desell, T., Maghraoui, K. E., and Varela, C. (2006). Malleable components for scalable high performance computing. *Proceedings of the HPDC15 Workshop on HPC Grid programming Environments and Components (HPCGECO/CompFrame)*, pages 37–44.

[Faraj and Yuan, 2002] Faraj, A. and Yuan, X. (2002). Communication characteristics in the NAS parallel benchmarks. *IASTED PDCS*, pages 724–729.

[Gabriel et al., 2004] Gabriel, E., Fagg, G., and Bosilca, G. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104.

[Garces et al., 2012] Garces, N., Sotelo, G. a., Villamizar, M., and Castro, H. (2012). Running MPI Applications over Opportunistic Cloud Infrastructures. *2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 309–314.

[Gentzsch, 2001] Gentzsch, W. (2001). Sun Grid Engine: towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid$*, pages 35–36. IEEE Comput. Soc.

[Habib, 2008] Habib, I. (2008). Virtualization with KVM. *Linux Journal*, 2008(166):8.

[Hines et al., 2009] Hines, M. R., Deshpande, U., and Gopalan, K. (2009). Post-copy live migration of virtual machines. *ACM SIGOPS Operating Systems Review*, 43(3):14.

[Hirofuchi et al., 2010] Hirofuchi, T., Nakada, H., Itoh, S., and Sekiguchi, S. (2010). Enabling Instantaneous Relocation of Virtual Machines with a Lightweight VMM Extension. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 73–83. IEEE.

[Hirofuchi et al., 2012] Hirofuchi, T., Nakada, H., Itoh, S., and Sekiguchi, S. (2012). Reactive cloud: Consolidating virtual machines with postcopy live migration. *Information and Media Technologies*, 7(2):614–626.

[Huang, 2008] Huang, W. (2008). High performance network i/o in virtual machines over modern interconnects.

[Huang et al., 2007] Huang, W., Liu, J., Koop, M., Abali, B., and Panda, D. (2007). Nomad: migrating OS-bypass networks in virtual machines. *Proceedings of the 3rd international conference on Virtual execution environments*, pages 158–168.

[Huber et al., 2011] Huber, N., Quast, M. V., Hauck, M., and Kounev, S. (2011). Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments. *CLOSER*.

[Iglesias, 2007] Iglesias, G. U. (2007). *Virtual malleability" applied to MPI jobs to improve their execution in a multiprogrammed environment"*.

[John and Hacker, 2012] John, M. and Hacker, T. (2012). AC 2012-3122: DEVELOPING VIRTUAL CLUSTERS FOR HIGH PERFORMANCE COMPUTING USING OPENNEBULA. *asee.org*, (December 2010).

[Kivity et al., 2007] Kivity, A., Kamay, Y., and Laor, D. (2007). kvm: the Linux virtual machine monitor. *Proceedings of the Linux Symposium*, 1:225–230.

[Knauth and Fetzer, 2011] Knauth, T. and Fetzer, C. (2011). Scaling Non-elastic Applications Using Virtual Machines. *2011 IEEE 4th International Conference on Cloud Computing*, pages 468–475.

[Kudryavtsev and Koshelev, 2012] Kudryavtsev, A. and Koshelev, V. (2012). Virtualizing HPC applications using modern hypervisors. *Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit*, pages 7–12.

[Kudryavtsev et al., 2012] Kudryavtsev, A., Koshelev, V., and Avetisyan, A. (2012). Modern HPC cluster virtualization using KVM and palacios. In *2012 19th International Conference on High Performance Computing*, pages 1–9. IEEE.

[Lagar-Cavilla, 2009] Lagar-Cavilla, H. (2009). Flexible Computing with Virtual Machines.

[Maghraoui and Desell, 2005] Maghraoui, K. and Desell, T. (2005). Towards a middleware framework for dynamically reconfigurable scientific computing. *Advances in Parallel Computing*, 14:275–301.

[Milojičić et al., 2011] Milojičić, D., Llorente, I. M., and Montero, R. S. (2011). OpenNebula: A Cloud Management Tool. *IEEE Internet Computing*, 15(2):11–14.

[Pan and Ren, 2006] Pan, Z. and Ren, X. (2006). Executing MPI programs on virtual machines in an internet sharing system. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*.

[Peng et al., 2009] Peng, J., Zhang, X., Lei, Z., Zhang, B., Zhang, W., and Li, Q. (2009). Comparison of Several Cloud Computing Platforms. *2009 Second International Symposium on Information Science and Engineering*, pages 23–27.

[Ranadive et al., 2008] Ranadive, A., Kesavan, M., Gavrilovska, A., and Schwan, K. (2008). Performance implications of virtualizing multicore cluster machines. *Proceedings of the 2nd workshop on System-level virtualization for high performance computing - HPCVirt '08*, (1):1–8.

[Raveendran et al., 2011] Raveendran, A., Bicer, T., and Agrawal, G. (2011). A Framework for Elastic Execution of Existing MPI Programs. *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 940—-947.

[Ruivo et al., 2014] Ruivo, T. P. P. D. L., Altayo, G. B., Garzoglio, G., Timm, S., Kim, H. W., Noh, S.-Y., and Raicu, I. (2014). Exploring Infiniband Hardware Virtualization in OpenNebula towards Efficient High-Performance Computing. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 943–948.

[Russell, 2008] Russell, R. (2008). virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*.

[Shao et al., 2011] Shao, Z., Wang, Q., Xie, X., Jin, H., and He, L. (2011). Analyzing and Improving MPI Communication Performance in Overcommitted Virtualized Systems. *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 381–389.

[Van Nieuwpoort et al., 2010] Van Nieuwpoort, R. V., Wrzesiska, G., Jacobs, C. J. H., and Bal, H. E. (2010). Satin: A High-Level and Efficient Grid Programming Model. *ACM Transactions on Programming Languages and Systems*, 32(3):1–39.

[Voorsluys et al., 2009] Voorsluys, W., Broberg, J., Venugopal, S., and Buyya, R. (2009). Cost of virtual machine live migration in clouds: A performance evaluation. *Cloud Computing*, pages 1–12.

[Wang and Varela, 2010] Wang, P. and Varela, C. (2010). Support for Virtual Machine Malleability in Cloud Computing Using Migratable Components. pages 1–7.

[Wu and Zhao, 2011] Wu, Y. and Zhao, M. (2011). Performance modeling of virtual machine live migration. *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 492–499.

[Ye et al., 2012] Ye, K., Jiang, X., Ma, R., and Yan, F. (2012). VC-Migration: Live Migration of Virtual Clusters in the Cloud. *2012 ACM/IEEE 13th International Conference on Grid Computing*, pages 209–218.

[Zhao and Figueiredo, 2007] Zhao, M. and Figueiredo, R. J. (2007). Experimental study of virtual machine migration in support of reservation of cluster resources. *Proceedings of the 3rd international workshop on Virtualization technology in distributed computing - VTDC '07*, pages 1–8.