MIKELANGELO

D5.4

First report on the Integration of sKVM and OSv with HPC

Workpackage	5	Infrastructure Integration	
Author(s)	Uwe Schilling		HLRS
	Nico Struckmann		HLRS
	Gregor Bergnic		XLAB
	Shiqing Fan		HUAWEI
	John Kennedy		Intel
	Joel Nider		IBM
Reviewer	Holm Rauchfuss		HUAWEI
Reviewer	Philipp Wieder		GWDG
Dissemination Level	PU		

Date	Author	Comments	Version	Status
2015-11-16	Uwe Schilling	Initial draft, outline and first sections	V0.0	Draft
2015-12-01	Nico Struckmann	Additions to the draft, and some new sections; content added, too	V0.1	Draft
2015-12-02	Gregor Berginc	MPI and OSv sections	V0.2	Draft
2015-12-03	Shiqing Fan	RDMA section	V0.3	Draft
2015-12-09	Joel Nider	sKVM section	V0.4	Draft

2015-12-14	John Kennedy	Monitoring content added	V0.5	Draft
2015-12-18	Nico Struckmann	Document ready for review	V0.6	Review
2012-12-28	Uwe Schilling, Nico Struckmann, Gregor Berginc	Addressed the comments from the internal review	V1.0	Final

Executive Summary

This is the first of three deliverables regarding the integration of new technologies, namely sKVM and OSv, into state of the art HPC-environments and workloads. The document describes the integration status of the software stack developed by the MIKELANGELO consortium, points out required modifications to the batch system Torque, and further reports on the progress of the overall integration.

The final objectives of the HPC integration is the execution of HPC applications in a virtualized compute node environment with the least possible overhead spent on the virtualization.

This document presents at first the concept for the integration of virtualized workloads into the batch system Torque, which is used in many HPC environments. It presents the mandatory modifications required in Torque and some extensions of its work-flow enabling the execution of batch jobs within virtual machines that is referred to as virtualized workloads. The initial implementation of the modified work-flow is presented next detailing the first proof-of-concept supporting the execution of these workloads. The virtualized compute node environment is another significant aspect covered in this document. It is outlined how technical approaches from Clouds can be adopted for the integration of virtualized workloads into HPC.

This document further describes the current progress of the HPC integration work that took place within WP5 comprising all the components and software packages involved. It describes the progress achieved for each component in detail and points out challenges and issues that arose during development that need to be addressed in succeeding development stages.

At the end of the document some conclusions and an outlook for the next development phase can be found. Initial measurements are provided in the Appendix A that compare the runtime of the Cancellous Bones application (HPC Use Case) executed on bare metal with the execution inside a virtual guest. Appendix B contains the templates used for the extension of Torque's workflow in order to run batch jobs inside virtual machines.

Acknowledgement

The work described in this document has been conducted within the Research & Innovation action MIKELANGELO (project no. 645402), started in January 2015, and co-funded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme (H2020-ICT-07-2014: Advanced Cloud Infrastructures and Services)

Table of contents

Table of Figures

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
DB	Database
ECC	Error Correction Code
HPC	High-Performance-Computing
I/O	Input / Output
IMG	Image (for VMs)
IPoIB	IP based connection over Infiniband protocol
KVM	Kernel-based Virtual Machine
MPI	Message Passing Interface
MPM	MIKELANGELO Package Manager
NFS	Network File System
Open MPI	Message Passing Interface library
PBS	Portable Batch System
POSIX	Portable Operating System Interface
RAM	Random Access Memory
RAM-disk	In Memory filesystem
RDMA	Remote Direct Memory Access
RESTful	Representational State Transfer
RM	Resource Manager
RoCE	RDMA over Converged Ethernet
SQL	Structured Query Language
VM	Virtual Machine

QEMU	Quick Emulator
sKVM	super KVM
SSH	secure shell
vRDMA	virtual RDMA

Glossary

contextualization	Configuration of a virtual guest on startup. Comprises for example the user's account creation, additional sw packages , configuration of services and applications, mount points, etc
cloud-init	Linux software package that is used for the contextualization in Cloud environments during a VM instance's boot sequence. It can utilize many different metadata sources[20], like Amazon EC2, OpenStack or OpenNebula's metadata servers.
metadata	Metadata provides all information for a virtual guest's contextualization
metadata disk	A disk image as metadata source for cloud-init's NoCloud data source
Resource Manager	Software that manages beside the job queue(s) all resources in batch system environment, like compute nodes, sw licenses and other limited resources. Further, it deploys batch jobs on requested resources. It works together with an scheduler, that plans the allocation of resources and instructs the RM when and where to deploy a job.
Scheduler	Component that schedules jobs residing in a queue, meaning the planning the allocation of resources at a certain duration and point in time under consideration of already scheduled jobs. Works in cooperation with a resource manager that reports available resources and their current status/availability.
Torque	Torque is an open source Resource Manager developed by Adaptive Computing. It comprises client tools to submit and monitor jobs, a server component, compute node daemon and an optional, simple scheduler. It supports external schedulers like Maui or Moab.
pbs_server	Torque's server component. Runs on the cluster's head node, beside pbs_sched that is responsible for the actual scheduling.
pbs_mom	Torque's compute node daemon. It is responsible for job for the actual job execution on the resources allocated by the scheduler
qsub	Torque's client tool for the job submission.

1 Introduction

This document reports on the progress of the HPC integration within the first 12 months of the MIKELANGELO project[1]. In this section, an overview about the document's structure and content is provided.

Section 2 starts with an overview that compares Cloud and HPC environment characteristics. It further points out which aspects of the Cloud concept are beneficial to HPC environments.

In section 3, the actual HPC integration work is in the focus. At first, the standard work-flow of resource manager Torque is introduced. The extended work-flow is then described in-depth, followed by the basic requirements that need to be covered for virtual job execution in HPC environments.

In the fourth section, the integration plan is outlined. It comprises all developments essential to the HPC integration and describes their advantages over currently existing technologies: Hypervisor sKVM, Guest OS OSv, Application Management, RDMA integration into the Hypervisor and Guest OS, MPI modifications required for OSv, and the monitoring & instrumentation component. All these components are implemented within the MIKELANGELO project.

The section 5 then provides an overview about the HPC test system setup at HLRS that is used for the actual integration of the several components and the validation of the concept.

In the subsequent section 6, the next steps planned for the HPC integration are described. Followed by section 7 that summarizes the document's key takeaways. Lastly, concluding remarks are provided in section 8.

At the end of the document, references and applicable documents are listed, preceded by an appendix that compares measurements taken on both, bare metal systems and virtual guest systems.

2 HPC Environments and Clouds

In HPC environments there are resource managers responsible for the management of the available resources like compute nodes and software licenses that are shared amongst all users. These resource managers also take care of the batch job deployment and compute node monitoring. In addition to the resource managers there are schedulers responsible for scheduling queued batch jobs that have been submitted by users, according to policies and priorities.

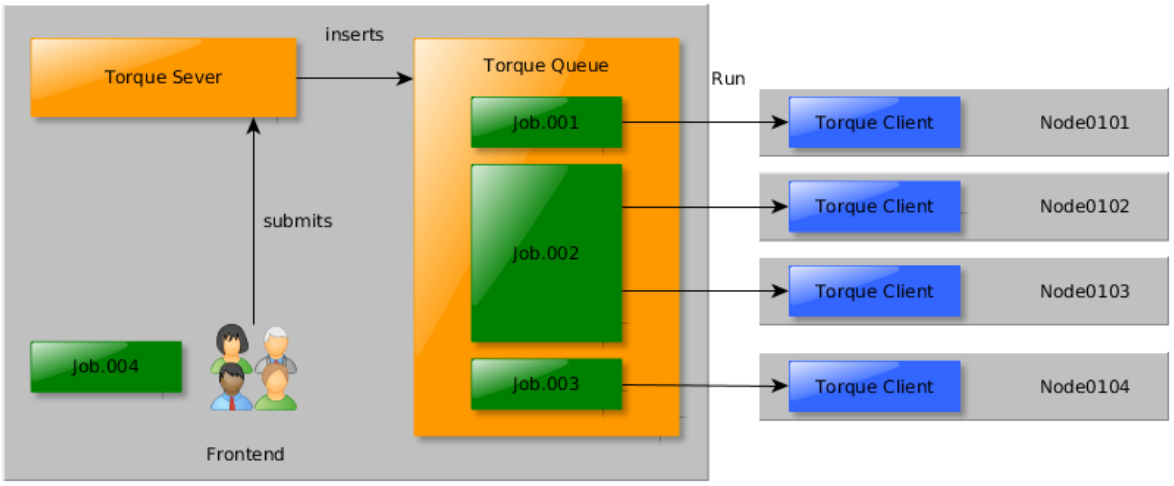

Figure 1: Typical HPC environment.

In traditional HPC environments physical nodes are usually reserved exclusively for a single user, since HPC applications are either CPU, memory, or I/O bound and sharing a node would result in sharing a bottleneck.

Traditional Clouds however follow the opposite approach, they put as much workload on a single node as possible to utilize the physical hardware the best possible way. Workloads deployed in Cloud are typically less CPU, memory and I/O bound. Thus, physical nodes are usually not allocated exclusively, but are shared amongst virtual instances owned by several users.

In both environments, no direct access to physical nodes is provided. There are schedulers taking care of resource allocation and the actual deployment, in order to enable users to best utilize the resources.

While Clouds allow their administrators to do maintenance of the physical nodes anytime with the help of live migration of the virtualized guests, the HPC administrators are required to schedule a downtime and wait for all user jobs to finish or cancel them.

Another advantage of Clouds is the fact that the operating system installed on the physical nodes is not of any relevance to the end-user. He is free to demand the operating system, kernel version and libraries matching his needs best, while HPC end-users are forced to utilize the provided compute environment as it is.

The goal of the MIKELANGELO project is to complement traditional HPC environments with certain Cloud technologies combining the advantages of both worlds. In the past, the overhead of the virtualization was too high and consequently prevented a wider adoption of these Cloud advantages in HPC environments. sKVM and OSv are addressing exactly the aforementioned overhead of virtualized environments. Being an integral part of the MIKELANGELO technology stack, they are going to support significantly improved performance over the traditional virtualization, in particular with respect to the I/O. With these emerging technologies, limitations and disadvantages associated with today's Cloud technology that makes it undesirable for HPC, disappear.

The advantage for HPC users would be a virtual, user defined software environment that serves the user with exactly the operating system, library versions available and software installed he needs.

Live migration technology or the suspend-and-resume functionality from Clouds can provide advantages to HPC environments as well and are worth to be investigated further. The live migration would be a very beneficial feature when it comes to exascale computing that suffers from the mean-time-between-failures (MTBF) of the physical hardware. It would allow to move the compute instance during the job's runtime to another physical (spare) resource. This is very useful when a physical node's health degrades, i.e. CPU temperature too high, a fan is broken, some RAM is defect, S.M.A.R.T values of HDD are bad, etc.

For HPC admins, the possibility of doing maintenance of physical nodes at any time is very desirable and could be achieved with the help of live migration or suspend-and-resume. Nowadays they are forced to schedule a downtime of the physical hardware and wait for running jobs to finish. Or in worst case they are forced to cancel them with the disadvantage of wasted computation time (jobs may run for several days). For example: a node degrades, a kernel security patch is required to be applied or new drivers/firmware needs to be installed.

3 HPC Integration with Torque

Torque[3] is one of the most widespread resource management software frameworks for HPC environments, offering basic scheduling functionality as well. It is used in many HPC sites, including the production environments at HLRS.

It is an open source product and licensed under the TORQUE v2.5+ Software License v1.1[5] that is based on the OpenPBS license v2.3[6]. Adaptive Computing is actively developing the code base in cooperation with the Torque community, thus making it a good choice for the integration of virtualized workloads.

The compute nodes are utilized with the help of batch job scripts that contain all steps required for the automatic execution of an HPC application, including preparation and tear down steps like copying data from a slow long term storage to a fast short term working space. Such a job script is submitted by an user to Torque's server that queues it. A separate scheduling component (not included in the figures Figure 2 and Figure 3) is then responsible for the resource allocation. The job is usually executed as soon as there are enough resources available, except there are additional policies in place preventing it - like for example total walltime per day or any fair-share limitations.

3.1 Standard Work-flow

Torque's workflow for batch jobs comprises several steps that are described in the figure below. From the batch job's submission over its preparation and execution to its tear down. The workflow for interactive jobs, that Torque is also capable of, differs in one major aspect only: the user is provided with shell access on the allocated compute node(s) in order to work interactively.

Figure 2 below shows the components of Torque that are relevant to the extension of its work flow for virtualized execution. That is the `qsub` client for job script submission, the server component `pbs_server` that takes delivery of job submissions and the `pbs_mom` daemon that runs on the compute nodes. This daemon actually starts the submitted job script as well as triggering the prologue and epilogue scripts that run before and after a job. The scheduler component that can be `pbs_sched` or an external, sophisticated scheduler is not of interest for the HPC integration of sKVM and OSv, thus it is not considered in any way.

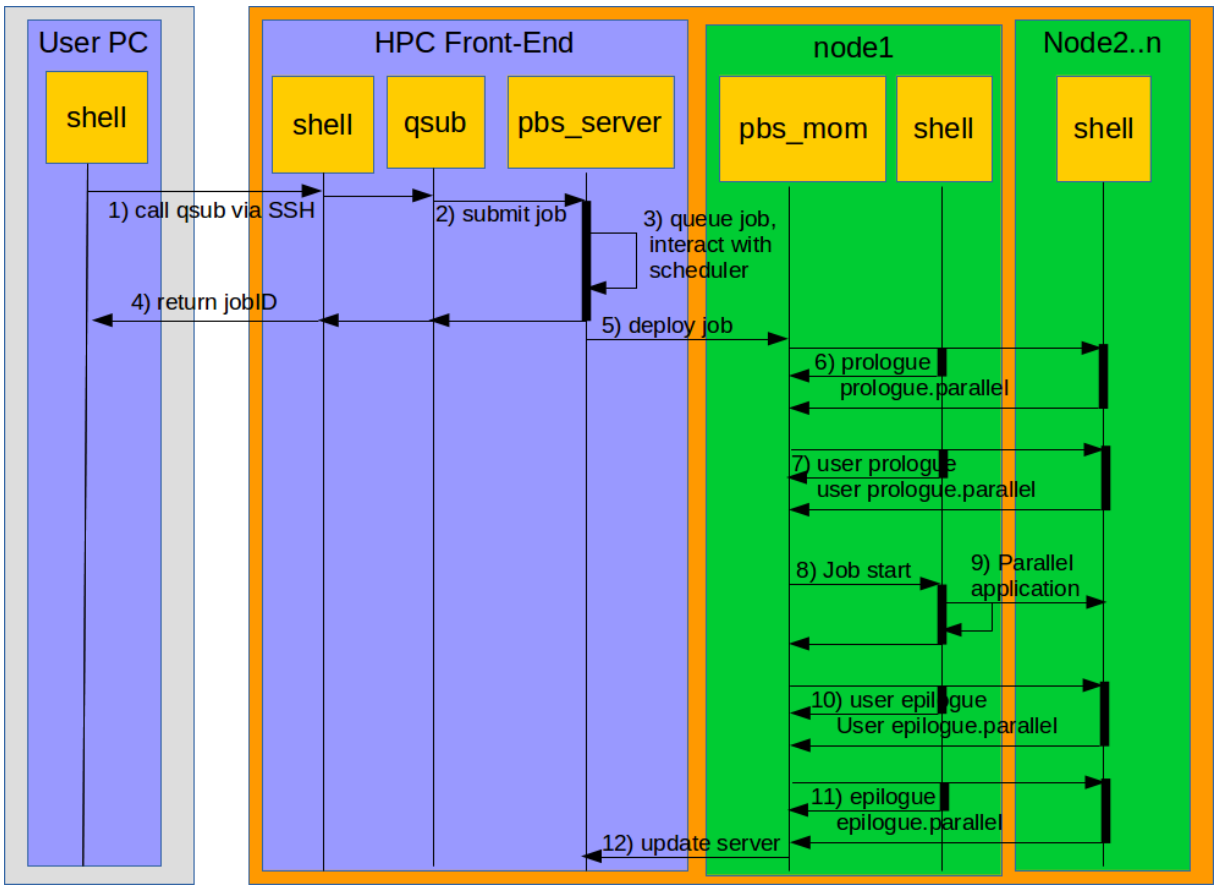

Figure 2: Torque's Workflow.

1) The user logs into the HPC front-end via SSH where he then submits the batch job script with the `qsub` client.
2) The server component `pbs_server` takes delivery of the job script submission
3) Scheduler is invoked, since there's a new job to schedule
4) Return the job id to the user

Meanwhile the scheduler figures out where and when the job can run

5) The server then deploys the job on the allocated and exclusively reserved compute node(s).
6) Before the actual job starts, the optional but statically configured prologue and prologue.parallel scripts are executed (root level)
7) And the also optional user's prologue and prologue.parallel scripts are executed (user level)
8) The job script is started on the first allocated compute node
9) The job is spawn over all nodes (parallel application)
10) After the actual job finished, the optional epilogue and epilogue.parallel scripts are executed (user level)

11) And the also optional but statically configured root prologue and prologue.parallel scripts are executed (root level)
12) `pbs_server` is updated by `pbs_mom` that the job has finished and the node is available again.

3.2 Extended Work-flow

The main idea is to swap the physical compute nodes with virtual nodes on a one-to-one basis, i.e. there is only one virtual guest per node at a time. The compute node host is now merely the minimum runtime to bring up the virtual guest. The work-flow of Torque needs therefore to be extended for the management of virtual guests. These modifications that are mandatory for the execution of batch jobs within virtual guests, are explained in the figure and corresponding text below.

All involved components of Torque, that are required to be modified in order to enable virtualized job execution, are described in the succeeding subsections. Together with an elaboration of the extensions implemented for the initial proof-of-concept prototype.

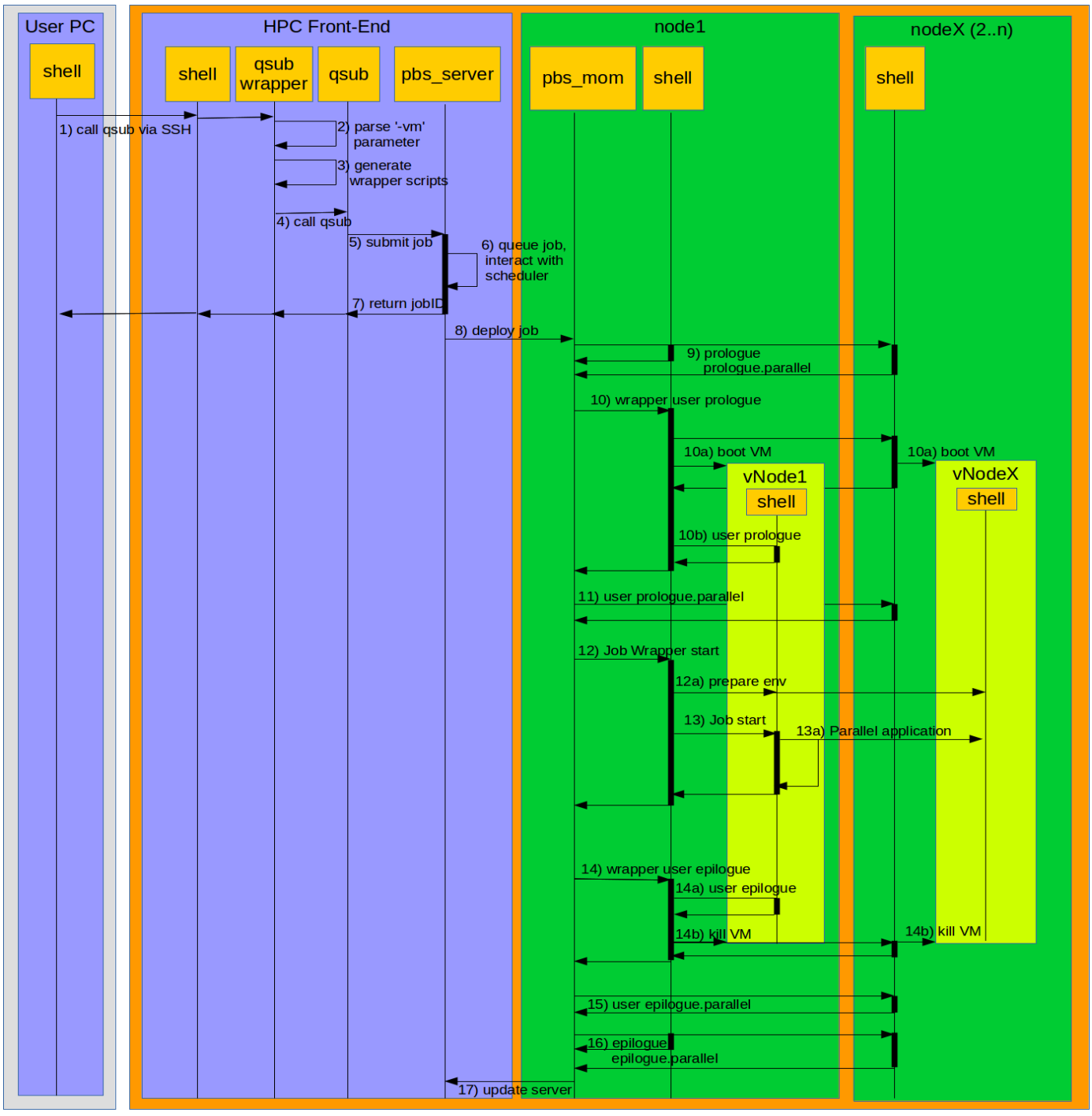

Figure 3: Torque's extended Workflow.

1) The user logs into the HPC front-end via SSH where he then submits the batch job script with the qsub client. The qsub client is wrapped by a script which is achieved by prepending the PATH environment variable with the scripts location
2) The wrapper script accepts `-vm` parameters (extended resource requests for virtual guest image, etc)
3) Needed scripts are generated

a) Prologue script that will boot the virtual guest, wrapping any optional user prologue given via qsub *-l prologue=<script>* parameter
b) Job wrapper script that is responsible for the preparation of the virtual nodes environment that can only be done during the job's runtime when it's available
c) Epilogue script that destroys the guests and cleans up the nodes afterwards, wrapping any optional user epilogue given via qsub *-l epilogue=<script>* parameter

4) The `qsub` binary will be called by the qsub wrapper.
5) This submits the job to the `pbs_server`. The (unmodified) server component `pbs_server` takes delivery of the job script submission that itself is modified in following aspects
   a) optional user prologue and epilogue scripts are wrapped by the generated ones
   b) the submitted batch script is wrapped by the job wrapper script that prepares the virtual node(s)
6) The job will get an ID and is appended to the queue
7) The jobID will be return to the user
8) The job gets deployed by the scheduler to the physical node (as soon as the job can run)
9) On this node prolog.parallel runs and the normal prolog on the first node. They could be used, i.e. to mount a RAM-disk or do similar things at system level.
10) The wrapper for the user prologue, that is responsible for booting a virtual guest on all allocated nodes, is executed (user level).
    a) Boots the virtual guests. It generate the required metadata, based on a template, that is used for the virtual guest's contextualization.
    b) Wrapped optional user prologue is executed within the first virtual guest.
11) Optional but statically configured user prologue.parallel scripts are executed on all bare metal nodes by Torque.
12) The job wrapper script starts
    a) Preparation of the environment on each VM-node by the job wrapper, with the help of a shared filesystem for user's HOME and the workspace (additional data staging between virtual and physical host can be circumvented by that).
13) Job Wrapper starts the user's Job script on the first VM-node
    a) Parallel application is executed and spawns over all nodes.
14) The wrapper for the user epilogue is executed (user level).
    a) Wrapped optional user epilogue is executed within the first virtual guest.
    b) Virtual guests on all allocated nodes are destroyed by the epilogue wrapper.

15) Optional but statically configured user epilogue.parallel scripts are executed on the bare metal nodes by Torque
16) Also optional but statically configured root prologue and prologue.parallel scripts are executed (root level) before the job completely finishes.
17) `pbs_server` is updated by `pbs_mom` that the job has finished and the node is available again.

3.2.1 Job Submission Extension (qsub wrapper script)

The first component that requires a modification is the job submission command `qsub`. This command is used to submit batch job scripts to the queue. We have extended this command with additional `-vm` parameters allowing one to define the properties of the virtual guest operating system for the batch job execution. This is achieved currently by a wrapper script that is parsing these parameters and taking care of additional steps required, like the basic metadata generation (based on templates) that is used for the virtual guest's contextualization. Please note that this wrapper is completely different from the Torque job submission filter[4] that is referred to as qsub-wrapper sometimes as well.

These additional VM parameters are accepted as a string of key-value pairs prepended with `-vm` and separated by commas, similar as the requests for resources that are issued with the help of `-l` succeeded by a comma separated list of key-value pairs.

```
qsub -l nodes=<nodeCount>[:ppn=<cont_of_cpus>,walltime=...] \
     -vm img=<imageFile.qcow2.img>, \
          distro=<debian|redhat|osv>, \
          vcpus=<count>, \
          metadata=<file.yaml>, \
          hypervisor=<kvm|skvm> \
     jobScript.sh
```

Mandatory VM parameters

img:	The virtual guest's operating system image.
distro: templates	Linux distribution, required for the contextualization since the for Redhat and Debian derivatives differ. For example due to differently named sw-packages. And further, OSv needs a different mechanism for the contextualization as software needs to be included in its image and cannot be installed during boot.

Optional VM parameters

vcpus:	Virtual CPUs for the guest, default is '*allCores - 1*'. The remaining core is dedicated to I/O.

metadata: YAML file containing metadata, like for ex. packages to be installed. Not available, yet, still under development.

hypervisor: The hypervisor used for the virtualization. Currently KVM is the default, but with availability of sKVM the default will become sKVM.

In addition to the parameter extension, the qsub wrapper script handles the generation of metadata based on predefined templates. This metadata is then used for the contextualization of a virtual guest. Besides that it further generates the job wrapper and user prologue/epilogue scripts based on predefined templates, too.

The proof-of-concept implementation currently wrapps the qsub binary. The final target is a source code patch for upstream Torque project. That requires at least, the `pbs_server` and `qsub` to be extended in order to make Torque work with the additional `-vm` parameters, and maybe the `pbs_mom` compute node daemon as well.

3.2.2 Virtual Node SetUp and TearDown (Prologue and Epilogue)

Torque allows executing scripts before and after a batch job runs. There is a prologue script that runs before the job starts, and an epilogue script that runs as soon as the job has finished. These scripts can be utilized for node preparation and clean up. The system prologue, prologue.parallel and epilogue, epilogue.parallel scripts are executed as root, while the user prologue and prologue.parallel scripts are executed with the user's privileges [2].

These scripts unfortunately lack a possibility to pass on any parameters. And the predefined, not extendable parameters provided by Torque during the execution of these scripts are limited to a very basic set[21]. To address this limitation we make use of templates that are generated on the fly with all required parameters inside the script we want to pass on.

3.2.2.1System Level Prologue and Epilogue

The system prologue could be used to prepare the physical host, e.g. creating a RAM-disk for the guest image that allows faster I/O to the guest's local file-system and the epilogue script can be used to clean things up and set them back to their default state.

The system level scripts are executed with root user privileges and have, in comparison to the user level scripts, fewer parameters. Since they are required to be owned by the root user, they cannot be generated on the fly during job submission that happens with default user rights.

3.2.2.2User Level Prologue and Epilogue

Torque's user prologue and epilogue scripts are generated on the fly and are utilized for the instantiation of the virtual guest and its shut down.

The user prologue and epilogue scripts are generated by qsub based on templates that can be found in the Appendix B. These generated scripts wrap user defined prologue and epilogue scripts if provided by the user.

3.2.3 Job Execution

After the prologue scripts have prepared the virtual environment, the job execution takes place. This is achieved by a wrapper script that prepares the instantiated virtual guest. It prepares the job environment by setting the required PBS environment variables and by creating a virtual nodes file. Then, the wrapper starts the actual job-script inside the VM.

The following steps take place during the job start:

At first the IP address of all guests running on the allocated nodes are fetched. For the collection of these IPs the wrapper script will connect to each node via ssh and run the command `virsh list`, to get the names of the VMs and their associated mac address. In a next step it makes use of the command `arp -an` to determine the VM IPs.

Then the script sets the job specific environment variables a batch jobs expects to be there. One of these environment variables is for example the `PBS_NODEFILE` pointing to a flat file that contains a list of all allocated nodes for the job. Other variables required to be set are for example the job ID or the job's name. For a complete list of relevant environment variables refer to section 3.2.2.

Since the user's `$HOME` is mounted inside the virtual guest, no data staging is required. Input files can be accessed transparently via the guest's local file system. Output files and job results are written back to this shared file-system as well, making it unnecessary to transfer it to somewhere.

The last step is then the actual user job script start on the first node of the allocated ones. It is to mention that there are three kinds of batch jobs that differ in some aspects. Torque allows the executing of:

- script files
- piped commands (i.e. `echo "mpirun -np3 .." | qsub -l ..`)
- interactive jobs

The wrapper allows all three kind of jobs, however it requires a patch for the interactive jobs since we need to execute the wrapper first, before providing the user with a shell inside the guest.

3.3 Requirements for Virtual Job Execution

The aspects that need to be considered for virtualized batch job execution are described in detail in this section.

3.3.1 Hypervisor on Physical Nodes

To execute virtual jobs, the physical compute nodes must provide a hypervisor, namely QEMU[7] in combination with the respective kernel module for KVM or its improved version sKVM. The libvirt[8] tools are used for the management of the virtual guests on top of the physical compute nodes.

3.3.2 Virtualized Compute Node Environment

All environment variables required for the execution of a batch job, namely the PBS_xxx variables[22], have to be defined inside the virtual node without the demand for any user interaction. They must match values passed to the physical host before the job gets executed.

The following PBS environment variables exist and need to be set in the virtual node environment:

- `PBS_NODEFILE` *(the file's content needs to be replaced by the VM hostnames)*
- `PBS_O_HOST`
- `PBS_O_QUEUE`
- `PBS_QUEUE`
- `PBS_O_WORKDIR`
- `PBS_ENVIRONMENT`
- `PBS_JOBID`
- `PBS_JOBNAME`
- `PBS_O_HOME`
- `PBS_O_PATH`
- `PBS_VERSION`
- `PBS_TASKNUM`
- `PBS_WALLTIME`
- `PBS_GPUFILE`
- `PBS_MOMPORT`
- `PBS_O_LOGNAME`
- `PBS_O_LANG`
- `PBS_JOBCOOKIE`

- PBS_NODENUM
- PBS_NUM_NODES
- PBS_O_SHELL
- PBS_VNODENUM
- PBS_MICFILE
- PBS_O_MAIL
- PBS_NP
- PBS_NUM_PPN
- PBS_O_SERVER

The software environment, like kernel version, library version, or installed packages are either provided by the image or defined via metadata that is used for the contextualization of the virtual node. Users can, for example, request with the help of the metadata additional software packages that are installed during startup and are available when the job runs (refer to the next section for more details).

3.3.3 Guest VM Image

The requirements for VM images are very basic, but differ for OSv and standard Linux guests.

Standard Linux guests just need to have the cloud-init package installed and configured to make use of a metadata disk that provides the data required for the customization of a generic image for a certain user. All other software packages are optional to be pre-installed inside the image. With the help of cloud-init and metadata, users can define additional software they need for their HPC application on top of a generic standard Linux image.

The requirements for an OSv image differ slightly, due to the fact that applications are required to be packed with the image and cannot be installed during instantiation. Therefore, OSv guests need to be self-contained due to the limited support for contextualization via metadata, for example it is not possible to create user accounts or install software packages from a repository during boot. The MIKELANGELO Package Manager is going to address this limitation and allow for dynamic composition of virtual machine images based on users' specifications.

3.3.3.1NFS client

One basic software package that should be integrated into the image to reduce the startup time is an NFS client. This is crucial for the virtualized execution since it is mandatory that the user's home is shared with the guests to circumvent data-staging issues. Furthermore, many HPC applications demand a shared file system across all compute nodes.

3.3.3.2Open MPI support

Another basic software package that needs to be in place is Open MPI. This is a crucial requirement for many HPC applications. For standard Linux guests it is recommended, while for OSv it is mandatory to pre-install it in the image due to the limitations mentioned above.

3.3.4 Torque Patch

Torque does not allow submitting a job script in combination with the parameter `-I` to request an interactive job, which provides the user with a shell on the first allocated node. With such a shell the user can work interactively in an HPC environment, useful for complex visualizations that require HPC resources as well as for development and debugging purposes.

This restriction can be removed by patching Torque's source code.

It is necessary to run the job script wrapper, preparing the environment for the job-execution, before the actual user job starts (see section 3.2.3). This is valid for interactive jobs as well, but cannot be achieved without getting rid of the aforementioned limitation by patching the source code.

4 Integration of MIKELANGELO Components into HPC

This section describes one the one hand the integration of the various components that are implemented within the MIKELANGELO project and on the other hand their advantages over currently existing technologies.

Some of these components are still under development while others are in the process of being integrated. In the following subsections, their availability, planned or already executed steps for the integration and further, achieved improvements are outlined.

4.1 Hypervisor sKVM

sKVM[9] has been designed to be completely backwards-compatible with the existing KVM interfaces and infrastructure. This is important from the standpoint of interoperability with guest operating systems, as well as adoption by the wider community. The changes in sKVM are centered around improving I/O performance in virtual devices. This is achieved by streamlining the design of the virtual I/O backend code.

Virtual I/O devices in KVM have two parts: the front-end, which resides in the guest, and the back-end, which resides in the hypervisor. The two parts communicate over a protocol called virtio[23]. sKVM does not modify the protocol, nor the front-end component, thus ensuring compatibility with any guest operating system that currently works with KVM. As such, any guest image that has been designed to run on top of KVM will work without modification on sKVM as well. sKVM only modifies the back-end component of the virtual I/O devices, an approach which is completely transparent to the guest.

sKVM does add some additional hypervisor configuration options to allow tuning the virtual I/O performance. This tuning is intended to be done “live” as the guests are running, since the computational loads and I/O loads are not constant throughout the life of the guest. To gain maximum performance, a process runs in the background to monitor various aspects of the I/O and CPU performance. As conditions change, this process can modify the configuration accordingly. If at any time the process ceases to run, or is not functioning correctly, the system will suffer a “soft” failure. All guests will continue to run as expected, but I/O and CPU performance may be affected.

4.2 Guest Operating System OSv

OSv [19] is an unikernel with extremely low footprint and very fast boot time. It is POSIX compliant and supports execution of unmodified binaries which makes it an ideal lightweight replacement for existing general purpose operating systems, such as Linux. Nevertheless, there are certain limitations that have to be considered when using OSv as a guest operating

system for running HPC applications. The following list presents some of the challenges MIKELANGELO project plans to address:

Single process. Because OSv is a unikernel, it only allows execution of a single process within one OSv instance. Multiple binaries can be launched in separate threads of the shared process and each binary can start as many additional threads as necessary.

HPC workloads typically involve the execution of multiple processes on a host machine through some sort of middleware such as Open MPI. Due to extensive use of global and environment variables in typical HPC applications, replacing processes with threads does not solve the main issue. Heavyweight processes work on separate environments while lightweight threads running in a single process all share the same environment.

To this end, we have analysed in full detail the Open MPI and the way it manages the execution of parallel processes. This lead to very specific requirements for the OSv which have already been partly implemented within WP4 (see report D4.4[24]). Consequently, this enabled us to implement a preliminary version of the patch for Open MPI integrating the support for replacing MPI processes with threads. The following section presents the results of this work in more details.

No support for scripting. As a consequence of the previous limitation, OSv also has no built-in support for scripting, such as BASH. On the other hand, OSv does expose a feature-rich RESTful API providing a set of endpoints to monitor and control the operating system and applications. This API can be used by the controller node to script the execution within OSv virtual machine.

Binary formats. OSv executes unmodified Linux executables. Currently it only supports dynamically-linked executables. Statically-linked and "ordinary" non-relocatable executables are not yet supported. Therefore, to run an executable on OSv it must be compiled as either a shared object or a position-independent executable (PIE). Typically, this only requires recompiling the source code with modified compiler and linker switches.

Application management. Because applications typically need at least a recompilation from source code, it is currently not possible to use any of the existing application management mechanisms used in Linux distributions (for example RPM or DEB). WP4 is thus also addressing this aspect working on an improved and much simplified way to build application packages that are to be used with OSv. This is

going to facilitate tight integration with HPC and Cloud infrastructures leading to a unified approach.

The primary focus of the integration of OSv with HPC infrastructure during the first phase of the project was the support for running parallel applications within OSv, i.e. addressing the first challenge mentioned above (single process). The following subsections presents the work done and some preliminary results.

4.2.1 Running Open MPI Applications on OSv

Open MPI is an open source implementation of the MPI standard [11] used by many HPC applications that run on supercomputers. MPI, which stands for Message Passing Interface, provides a standard specification for a library for a parallel execution model. It was first introduced in 1994 with version 1.0 and has been evolving through the last twenty years. As of 2015, the latest stable release of MPI specification is 3.1.

MPI works by spawning the requested number of processes on one or more compute nodes. A central process is responsible for proper initialisation of each process. Although many implementations of the MPI specification provide a common startup command (mpirun) for execution of MPI-enabled processes, the specification itself does not specify this as a mandatory requirement. Rather, it presents a possible interface towards the end users and proposes some optional command parameters.

Recent additions to the MPI specification have also addressed multi-threaded environments. Default initialisation of the underlying MPI infrastructure does not allow simultaneous access from multiple threads of the same process. With these additions it is possible to request different level of thread-safety in the MPI infrastructure allowing software developers to write code exploiting the MPI infrastructure in multiple threads. Open MPI has introduced support for this part of the MPI specification since version 1.2 (in 2010). However, many applications, such as OpenFOAM, still rely on process-based parallelism as it greatly simplifies the management of source code.

As discussed in the previous section, running multiple processes in a single OSv-based virtual machine is one of the major challenges MIKELANGELO as a whole has to focus on in order to allow execution of HPC workloads. When it comes to running MPI applications these limitations become blockers preventing any kind of parallel execution of applications. In order to address these issues we have conducted an iterative approach that comprises the following phases:

1. Analyse parallel execution of MPI applications (section 4.2.2)
2. Run each MPI process in a separate OSv virtual machine (section 4.2.3)

3. Integrate an MPI daemon responsible for launching new MPI application instances into OSv VMs (section 4.2.4)

All these phases are briefly presented next.

4.2.2 Analysis of Parallel Execution

4.2.2.1Running MPI Program on a Local Node

In order to evaluate the entire lifecycle of an MPI-enabled application we have conducted a series of tests on the actual use case of the MIKELANGELO project, namely the Aerodynamics use case [15]. Two important findings were discovered during this analysis:

- multiple MPI processes do not share file descriptors of opened files and
- there are no signals called during the execution of the parallel simulation that could cause problems to the underlying guest OS (for example `fork`).

Both of these revealed that there are no additional technical limitations that could prevent the execution of the parallel MPI processes as threads within one OSv virtual machine. This further lead to the implementation of the aforementioned support in OSv for separation of environment of different threads.

4.2.2.2Running MPI Program on a Remote Node

When running MPI program on a cluster, one node (i.e. the head or main node) is usually used only for management tasks, while the remaining nodes are performing actual computational tasks. The simplest case of starting a single `mpi_program` instance is depicted below. The `mpirun` command connects to remote nodes via ssh. Instead of immediately starting `mpi_program`, a helper MPI daemon called `orted` is started. This daemon is then responsible for starting the actual `mpi_program`, for monitoring its execution and for forwarding possible failure message to the `mpirun`.

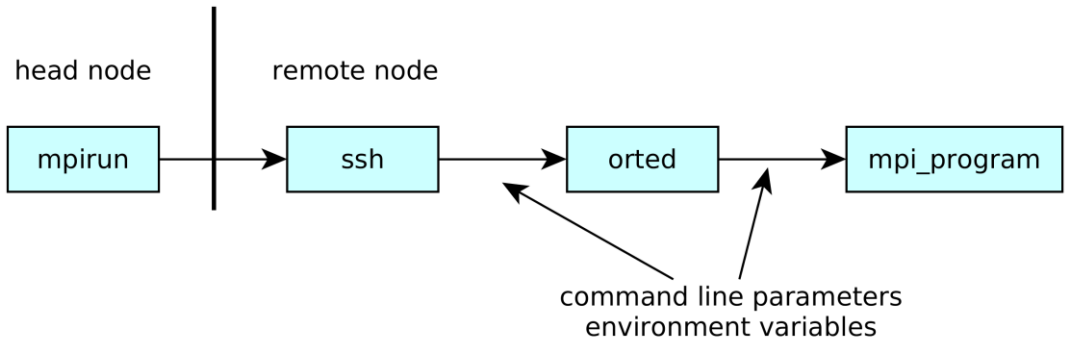

Figure 4: Components and the relationships between them participating in the instantiation of an MPI application process.

During execution, command line parameters are used for communication (in direction from "parent" to the "child" process). Both `ssh` and `orted` also setup environment variables, which are then used by its child process. The `orted` itself receives the command line parameter `–mca orte_hnp_uri`, which is the TCP socket, where `mpirun` is listening. The `orted` connects to that socket in order to establish bidirectional communication with `mpirun`. The `mpirun` then uses that communication channel to direct orted to start actual worker processes.

The environment variables set up by orted for each started worker process (by convenience all have a prefix `OMPI_`) are particularly important here. For example, they include the total count of all worker processes (`OMPI_COMM_WORLD_SIZE`), the rank of current worker process (`OMPI_COMM_WORLD_NODE_RANK`, `OMPI_COMM_WORLD_LOCAL_RANK`), the runtime configuration of various Open MPI modules (`OMPI_MCA_initial_wdir`), etc. Just as orted connects back to mpirun, also the worker process connects back to the socket opened by the orted - the exact IP and port are sent in `OMPI_MCA_orte_local_daemon_uri` environment variable. The orted also prepares a session directory for each worker process (`OMPI_FILE_LOCATION`).

In general, each worker process has and requires different environment variables than the rest of them. For example, each has a unique value of `OMPI_COMM_WORLD_NODE_RANK`. Thus, in order to run an Open MPI program inside OSv VM, a copy of environment variables needs to be provided by `orted` to the OSv VM before starting the `mpi_program`.

The `orted` starts the Open MPI program with the use of `fork` and `execve` system calls. Part of preparation involves setup of `stdin/stdout/stderr` file descriptors and a dedicated pipe to report failures from the child (`mpi_program`) back to the parent (`orted`). The child also has to close unneeded file descriptors. However, in OSv, the child process can be implemented only as a thread. Consequently, all file descriptors are visible to all threads and the child should not close them. Otherwise, this would prevent the parent from using them.

4.2.2.3Running Multiple MPI Program Instances on a Remote Node

When an additional worker process is added to the same node, the situation slightly changes, as shown in the picture below. Since orted knows that both workers reside on the same physical host, it prepares a suitably, optimized environment. In particular, shared memory communication mechanism is set up instead of the slower TCP/IP based communication. This mechanism uses dedicated files in the directory defined by the `OMPI_FILE_LOCATION` variable. The directory and these files are created by the `orted`, and `mpi_program` only uses them.

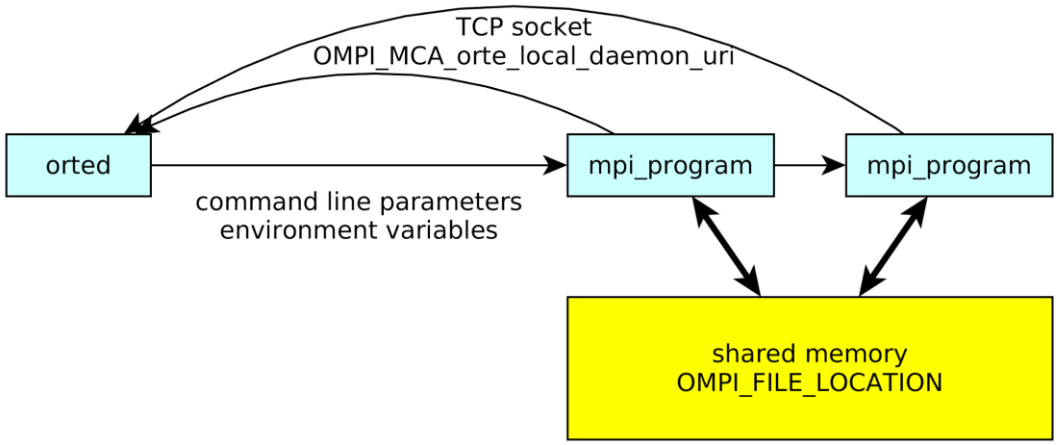

Figure 5: The process of configuring the shared memory in MPI application.

As `orted` has to prepare shared memory file, it has to access the same filesystem as the `mpi_program`(s). This means that `orted` has to run inside the OSv VM. In order to be certain that running parallel applications in OSv is feasible we have designed and integrated a preliminary solution using TCP-based communication between workers instead of aforementioned shared memory mechanism. The following section presents this approach.

4.2.3 Execution of a Single MPI Process on a VM

This is the simplest scenario of running parallel MPI processes in OSv. Since every process is running in a separate VM, there is no risk in running into issues related to sharing the environment (environment and global variables) by these processes.

We implemented such a solution using a utility proxy script. Usually, `orted` starts `mpi_program` directly. In this modified flow, however, orted starts a proxy script that

1. examines environment variables,
2. copies the existing VM base image (so that multiple OSv VMs have their own image each),
3. starts OSv,
4. initialises environment variables exploiting the OSv REST API, and finally
5. starts the Open MPI program.

By default, orted knows that it will start multiple Open MPI processes on the same host, and will try to use shared memory for communication between them. As each Open MPI process runs in a separate VM, we must prevent shared memory communication with the runtime configuration option `-mca btl self,tcp`. This forces the use of TCP/IP communication between worker processes running in different VMs (even when on the same host).

As a consequence, the overall performance of this approach is significantly slower than running MPI processes. An increase of performance is expected with the availability of the vRDMA component being developed as part of Work Package 3. This will exploit shared memory based communication between workers running in separate VMs on the same physical node.

Although this approach is the simplest solution with the least changes necessary to Open MPI, running each MPI worker in a separate virtual machine induces certain amount of overhead, which can be quite large on modern machines with large number of CPU cores. To this end we have extended our work to support running multiple threads within one OSv-based virtual machine. The next section details this approach.

4.2.4 Integration of MPI Daemon into OSv Virtual Machine

To achieve higher performance, efficient communication between worker OSv threads is mandatory. As we have mentioned before, for processes/threads running on the same host this means using shared memory for sharing information between workers.

When considering OSv in this kind of deployment, this requires:

1. multiple Open MPI processes (threads) to run inside one OSv VM,
2. `orted` is executed within the same OSv VM as the actual MPI threads,
3. each Open MPI program has its own environment,
4. each Open MPI program and `orted` have separate global variables (`BSS` and `DATA` memory sections should not be shared), and
5. code related to managing `stdin/stdout/stderr` file descriptors and error reporting pipe has to be modified.

The reasoning for 1, 2 and 3 was given in section 4.2.2 *Running MPI Program on Remote Node*. The problems with global variables were identified as soon as the `orted` was executed within OSv VM. It was able to run a trivial Linux (e.g. not Open MPI program), so it was basically functioning correct. But when we tried to run an Open MPI program, it did not work. Closer examination showed that communication component initialised by `orted` was marked as such. This lead to a problem that succeeding `mpi_program` instances found that the communication component is already set up and completely ignored the initialisation. This resulted in `mpi_program` representing itself as `orted` instead of a separate worker with a specific identifier, which caused the application to fail.

Having the ability to completely separate environment of OSv threads (separate global variables, ELF namespaces) allowed us to run a parallel simulation using Open MPI infrastructure. The shared memory communication mechanism can be used unmodified. Minor modifications of the `orted` daemon were required, dealing with child/parent file

descriptor preparation. The modified version detects if it is running as a thread on OSv and closes only a subset of file descriptors. It also replaces the `fork` and `execve` system calls with a creation of a new thread with a different environment as that of orted.

Finally, once child `mpi_program` starts as a new OSv thread, it should not blindly close all its file descriptors. It should only close file descriptors intended for its exclusive use (e.g. write end of the error reporting pipe). In similar sense also the parent orted is required to close only file descriptors not used by the child (say read end of the error reporting pipe).

Preliminary study based on the OpenFOAM use case (Figure 6) shows that the performance of these integrated patches running 2 or 4 threads on OSv outperforms 2 or 4 processes on a Linux-based guest. Even though OpenFOAM simulations are mostly CPU-bound, the performance of the host is still better than that of both virtualised cases.

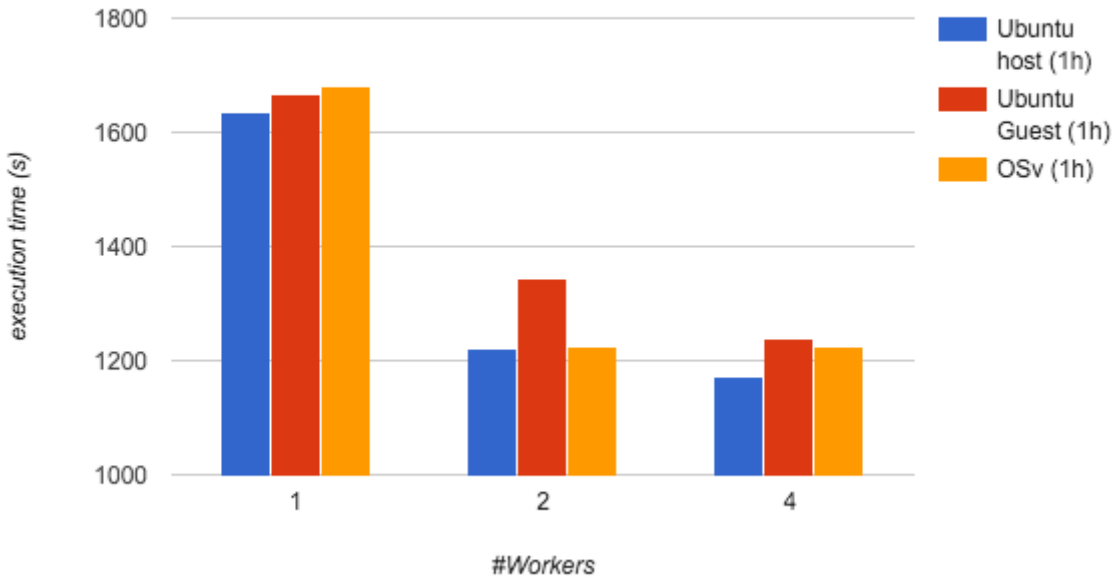

Figure 6: OpenFOAM benchmark comparing execution times of a single case with 1, 2 and 4 workers (lower is better). Both Ubuntu-based configurations use unmodified OpenFOAM and Open MPI, while the OSv used modified Open MPI launching threads instead of processes.

It is worth noting that this integration is still at its early stage. Our main plan is to extend the usability of these patches and make sure they are stable across a wide variety of MPI-based applications, starting with all MIKELANGELO use cases. We are also going to investigate the possibility of providing our changes to Open MPI upstream as a component. This would facilitate the use of unmodified Open MPI with just this component turned on responsible for transition from processes to threads.

4.3 Application Management

As we have already described in previous sections, Linux-based virtual machine images (VMI) are expected to either contain all the data, applications and libraries or a module installed in VMI facilitating installation of additional components.

On the other hand OSv-based images must be self-contained prior to being instantiated by the Torque system. Report D4.7 [25] has introduced the two existing mechanisms for building virtual images using build scripts that are part of the OSv source tree and Capstan. The report also introduced the extensions that MIKELANGELO has already provided and those that are planned for future releases. Although these extensions are currently being applied to Capstan, we refer to them as MIKELANGELO Package Manager (MPM) to separate the two before changes are available upstream.

Similarly, to the integration with the chosen Cloud management framework presented in D5.1 [26], integration with HPC management layer will be done using the upcoming RESTful API of the MIKELANGELO Package Manager (MPM). However, the integration of MPM into HPC layer will not be as tight as that of the Cloud due to the different workflows as well as nature of the applications.

MPM already supports application maintainers to package libraries, tools and applications, as well as compose them into executable OSv-based virtual machine images. These images may be used locally or stored in a centralised repository. Because of that, HPC integration will focus on using the images from the aforementioned repository when launching them. To prevent unauthorised access, user identification will be integrated into MPM. This should use existing user's credentials and not require any additional authentication making it transparent for HPC end users.

We will also be considering a more advanced integration that will compose virtual images on the fly based on existing packages. This approach is best explained using an example:

```
$ mpirun -np 8 simpleFoam -case /share/case -parallel
```

This example uses Open MPI (`mpirun`) to execute a parallel simulation using 8 (`-np 8`) OpenFOAM processes (`simpleFoam`). It also specifies the location where to find the input data (`-case /share/case`). The above command is never launched manually. It is passed to Torque scheduler that will invoke it when the resources for the job are reserved.

The above command consists of two separate applications (`mpirun` and `simpleFoam`), each optionally comprised of several other packages. For example, report D4.7 [25] presents the simpleFoam application package and its requirements

- OpenFOAM core libraries,

- OpenFOAM simpleApplication solver and
- HTTP server for debugging purposes

Given the command passed to Torque and the metadata of the packages in the central repository, it is possible to compose the target virtual image on the fly. Composition of the image can occur *just-in-time*, i.e. when the Torque launches the prologue script, or immediately after the job has been submitted. The former approach loses valuable time when the resources are already available, while the latter risks composing images for jobs that are later cancelled.

Although the above example used a specific case, it can be generalised through a powerful metadata that will be available in MPM. Open MPI is one of the most commonly used parallelisation middlewares, including both HPC use cases in MIKELANGELO, allowing us to specialise the integration to this case.

4.4 RDMA Integration

Remote Direct Memory Access (RDMA) is a technology that allows accessing the memory of another computer directly without involving the operation system of either systems. It provides high bandwidth and low latency communications over Infiniband/Ethernet networks.

In D2.13 [10], we have proposed three prototypes of RDMA virtualization (vRDMA) solutions. Design prototype I has been planed to be implemented for the first project year. The implementation details, basic integration and instruction manual has been presented in D4.1 [12]. In this section, we will discuss more specific requirements and issues for integrating the design prototype I in an HPC environment, as well as the instrumentation and monitoring integration for vRDMA on HPC platform.

Configurations for HPC

The integration manual described in sections 7.1 to 7.3 of D4.1 [12] targets on a general case of setting up the environment for vRDMA prototype I, where each workstation has to be explicitly and manually installed and configured. However, for a HPC environment, software packages are normally installed on a NFS to be used among all the compute nodes, i.e. install once and use everywhere. The use of NFS introduces additional complexities for integrating design prototype I into the system. For example, all the compute nodes may use the same software installation, which normally results access conflicts of the same context files or metadata that are stored in the installation directory with the same default file names. Therefore, the main challenge here is to distinguish the context and metadata for each compute node automatically during the initialization phase, and also to manipulate the correct configuration for VM startup.

In order to correctly configure all the run-time settings, all the necessary packages have to be correctly installed, including DPDK 2.1.0, Open vSwitch 2.4.0, libvirt 1.2.19, QEMU 2.4.0. The installation steps are the same as described in section 7.1 to 7.2 of D4.1. However, section 7.3 of D4.1 [12] does not apply for the HPC case, where the software installation paths is usually on a NFS. With the default settings of the installed software, different compute node may try to access the same metadata or context file, which will result run-time conflicts. In order to distinguish the context and metadata for each compute node to use the same software installation, special configurations have to be taken as described in section 7.4 of D4.1. For different compute node, when running the same command line, for example *virsh list*, additional parameter "-c" has to be provided for connecting to a user specified socket:

```
# virsh -c \
qemu+unix:///system?socket=/opt/libvirt/libvirt-1.2.19/var/run/libvirt-cn1/libvirt-sock \
list
```

Here, libvirt-sock is the connection socket between libvirt and qemu, and it has to be stored in a directory specifically for this node, i.e. in /opt/libvirt/libvirt-1.2.19/var/run/libvirt-cn1, where cn1 is the node name. We use the node name as the identity for each command from different node, and this can be configured automatically in the environment variables.

As a few of the commands related to Open vSwitch and libvirt have to follow this scenario, environment variables are used. For example, to run the equivalent command above, we are able to simply run:

```
# virsh -c $LIBVIRT_URI list
```

In order to further hide the complexities, command aliases have been applied by adding a prefix "mike-" to these commands. So the following command is identical to the previous one:

```
# mike-virsh list
```

This prefix rule has been applied to a few Open vSwitch and libvirt related commands, including *virsh*, *ovs-vsctl*, and *virt-manager*.

A detailed description on the above issues have been addressed in section 7.4 of D4.1, including an example script.

This will simply use the methodologies described in section 11 and section 12. For each VM, a unique IP address has to be assigned for vhost-user device.

4.5 Instrumentation & Monitoring

A complete instrumentation and monitoring stack is being created in Task 5.3 to meet the specific needs of the MIKELANGELO project. This work is creating a scalable, highly extensible, telemetry gathering and processing stack, the progress of which can be seen in Deliverable D5.7[13] of the complete MIKELANGELO software stack.

The design and construction of the HPC testbed has progressed in parallel and in close consultation with the instrumentation and monitoring efforts. Requirements were fed into the scoping of the work, and are being clarified, prioritised and managed via regular conference calls.

At the time of writing, the instrumentation and monitoring framework, built on the recently open-sourced snap telemetry framework[27], now includes the functionality to

- capture hundreds of metrics from the MIKELANGELO HPC hardware, host operating system, hypervisor (sKVM), and any hosted instances of OSv.
- calculate moving averages of data as required
- publish data to MySQL, PostgreSQL, InfluxDB, OpenTSDB and other back-ends
- expose data via arbitrary graphical user interfaces such as customisable Grafana dashboards

Additionally, snap supports

- arbitrary additional functionality via an extensive plugin-based architecture
- signing of plugins
- encryption of telemetry data transmission
- tribe-based distributed management
- remote, dynamic configuration through command line and RESTful interfaces

As the HPC testbed infrastructure construction and configuration completes, snap will be deployed on all hosts and plugins to capture as much data as is possible from the complete hardware and software stack. Appropriate metrics will be routed through the Moving Average plugin, and the Anomaly Detection plugin when it becomes available. This latter plugin will avoid unnecessary transmission of static telemetry data: high resolution data will only be transmitted when anomalies are detected.

The initial intent is for HPC testbed telemetry data to be fed into an InfluxDB database instance, from which Grafana based dashboards and other tools will be used to query, investigate and compare results of experimental runs. As monitoring needs are further

refined, inputs will be fed into the Instrumentation and Monitoring activities to help develop as useful a Performance Analysis platform as possible.

5 HPC Test Environment

The HLRS HPC test environment for MIKELANGELO is briefly described in this section. The software and hardware stack of the test system mirrors as close as possible a real, but scaled-down HPC production environment. In the last month this testbed has been configured and upgraded with more RAM to fulfill the needs of this project.

There is a dedicated front-end server that is accessible via the Internet. Further, there are 14 separate physical compute nodes. In addition to these nodes there are four more nodes needed. Two of them are equipped with Mellanox ConnectX-3 VPI cards[14] feasible for RDMA over Converged Ethernet (RoCE) tests, and two nodes are dedicated to kernel building and testing. These four separate nodes are not integrated into the batch-system.

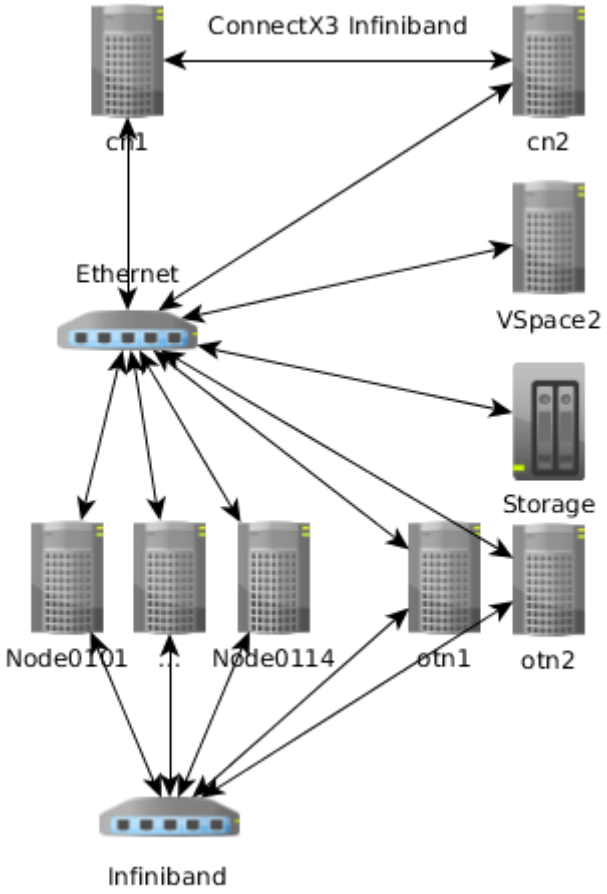

Figure 7: Testbed network topology.

Each compute node has two sockets, each with 8 Core Intel Xeon CPU X5560 (2.80GHz)[17] and 96GB of ECC DDR3 RAM. They have one 1 Gbit Ethernet network card and a 10 Gbit Infiniband network that supports RDMA.

A NFS server provides shared file-systems for:

- the user's `/home`,
- the global application installation dir `/opt`, and
- and a fast workspace mounted under `/scratch`.

6 Next Steps

The next steps regarding the HPC integration are summarized in a bullet point list below. .

- **Kernel patch installation**
  The first installation on the testbed was an Ubuntu 14.04.01 operating system shipped with kernel 3.13. For a full integration of MIKELANGELO, the kernel version 3.18 is mandatory, since it is compatible with the latest version of sKVM. The patch comprises backported patches from the v4.x kernel. It will be installed begin of Q1 2016 as soon as available.

- **Hypervisor sKVM**
  sKVM will be installed as soon as all kernel patches are available and successfully installed in HLRS' HPC test environment. The version installed on the testbed needs to be compatible with kernel v3.18.

- **Guest Operating System OSv**
  OSv is planned to be used with the modified Torque workflow to execute, at first, simple job scripts within OSv during Q1 2016. The NFS client support mandatory for the HPC integration is already available, but Open MPI still requires some additional changes for complete compatibility

- **Virtual interactive jobs**
  Interactive jobs running inside virtual machines require the preparation of the VM's environment the same way as script file based jobs. This means that interactive jobs need to be handled by the job wrapper as well, in order to be able to prepare the virtual node. However, Torque does not allow submitting job script files in combination with the qsub parameter `-I` that requests interactive jobs. Thus, we need to patch Torque's source code in order to enable this functionality.

  The indented point in time to start the patch development is end of Q1, begin of Q2 in 2016. This can take place in parallel to the other tasks, with a lower priority though as interactive jobs are rarely used in comparison to script based batch jobs.

- **Patching Torque's source code**
  To replace the proof-of-concept qsub wrapper and to allow the submission of interactive virtual jobs (with `-vm` parameters), patching Torque's source code is inevitable. The patch further allows pushing our changes upstream.

  Since the proof-of-concept prototype works and is feasible to proceed further with the other component's integration, this work is planned to be started end of Q2, begin of Q3 in 2016.

- **Integration of MIKELANGELO Package Manager**
  MIKELANGELO Package Manager is going to support dynamic, on-the-fly composition of virtual machine images, based on some form of metadata description. This is going to simulate the behaviour of Linux contextualization. This will be available for the first initial release of the MIKELANGELO stack in M18.

- **Execution of the Cancelous Bones use case within OSv**
  To validate MIKELANGELO's overall architecture, the execution of HLRS' use case Cancelous Bones[18] inside OSv is the final target. This task is dependent from the finalization of the Open MPI integration within OSv as it depends on Open MPI, besides a NFS client that is meanwhile in place.

7 Key Takeaways

This report presented the progress made during the first year of the MIKELANGELO project related to the HPC integration:

- Identification of high-level (business) requirements for the virtualized job execution. We have based these requirements on a thorough analysis of the benefits of the Cloud technology that are also interesting from the perspective of HPC, for example emergency maintenance or moving away from degrading nodes during the job runtime that is achieved by live migration of virtual machines or suspend and resume functionality.
- Definition of low-level, technical specifications required to achieve the business requirements
- Identification of relevant components of Torque and the workflow for the intended modification to enable Torque to manage VMs and run job scripts inside them
- Implementation of an initial proof-of-concept prototype already supporting the extended workflow for submitting batch jobs that run in virtualized Ubuntu guests
- Implementation of preliminary patches for Open MPI's `mpirun`'s start-up mechanism facilitating simplified execution of parallel applications in OSv

In the next phase of the project, towards the first release of the MIKELANGELO stack, we are planning to integrate the initial proof-of-concept concept for virtual job execution into Torque's source code. This is going to serve two purposes. First, it is going to provide transparent user experience for end users. Second, this will allow us to push the changes upstream and, consequently, promote the MIKELANGELO project through additional exploitation channel. We are also going to overcome Torque's limitation of not allowing script files to be submitted in combination with interactive jobs.

Majority of high performing applications uses the infrastructure and the framework offered by Open MPI, an open source implementation implementation of the Message Passing Interface. We have seen in this report that the initial integration is already in place in this early stage of the project. However, the way OSv applications are launched significantly different than those in Linux, thus requiring further integration to be fully compliant with Torque.

We furthermore envision the integration of sKVM, the super KVM, providing improved I/O management, virtualized RDMA and better security into the virtualized HPC. Because the two hypervisors are fully compliant, they are also completely interchangeable allowing dynamic selection of the one used for specific workloads. Lastly, the lightweight and scalable

monitoring will allow for fine-grained measurements of the overall performance clearly pointing out the overhead of the virtualization and achieved improvements.

8 Concluding Remarks

With the initial proof of concept implementation we are able to show that our approach is working in general. Standard Linux guests can be used to execute batch jobs with Torque inside virtual machines on top of KVM. Measurements of the Cancellous Bones application's total runtime, provided in the Appendix B, that compare the bare metal execution to the execution in a standard Ubuntu guest with KVM, show an overhead of approximately 10% for the virtualization.

However, for the objective to run the Cancellous Bones application in an HPC environment inside OSv, there is still some conceptional work to be done as OSv differs in some major aspects severely from standard Linux operating systems.

The most challenging aspect of the HPC integration regarding OSv is the support for Open MPI. As pointed out in section 4.2.1, MPI depends on several components that are used to run in separate processes. The presented design for the Open MPI support in OSv runs parts of the Open MPI on the physical nodes while other parts run inside the guest OS. This requires our proof-of-concept to be extended by a mechanism to satisfy a distributed design. We need to modify submitted job scripts on the fly, instead of running the whole user job script transparently in the guest. This requires the jobs to be analysed and modified accordingly to be able to run the MPI startup in a distributed way. How this can be solved for binary executables that make use of mpirun seems to become the major challenge.

9 References and Applicable Documents

[1] The MIKELANGELO project, http://www.mikelangelo-project.eu/

[2] Torque v6.0.0 online documentation, Prologue/Epilogue scripts, https://docs.adaptivecomputing.com/torque/6-0-0/help.htm#topics/torque/13-appendices/prologueAndEpliogueScripts.htm

[3] Torque Resource Manager, http://www.adaptivecomputing.com/products/open-source/torque/

[4] Job Submission Filter TORQUE, http://docs.adaptivecomputing.com/torque/6-0-0/help.htm#topics/torque/13-appendices/jobSubmissionFilter.htm

[5] Torque License v1.1, https://github.com/adaptivecomputing/torque/blob/master/LICENSE

[6] OpenPBS license v2.3, http://www.webcitation.org/60b6KmP55

[7] QEMU Open Source Project Page, http://wiki.qemu.org/Main_Page

[8] libvirt The virtualization API, http://libvirt.org/

[9] Introducing sKVM, https://www.mikelangelo-project.eu/2015/10/how-skvm-will-beat-the-io-performance-of-kvm/

[10] MIKELANGELO Report D2.13 The first sKVM hypervisor architecture, https://www.mikelangelo-project.eu/deliverables/deliverable-d2-13/

[11] MPI standard, http://www.mpi-forum.org/docs/

[12] MIKELANGELO Report D4.1 The First Report on I/O Aspects, https://www.mikelangelo-project.eu/deliverables/deliverable-d4-1

[13] MIKELANGELO Report D5.7 First Report on the Instrumentation and Monitoring, https://www.mikelangelo-project.eu/deliverables/deliverable-d5-7/

[14] Mellanox ConnectX-3 Adapter with VPI, http://www.mellanox.com/page/products_dyn?product_family=119&mtag=connectx_3_vpi

[15] MIKELANGELO Report D2.10 The First Aerodynamic Map Use Case Implementation Strategy, https://www.mikelangelo-project.eu/deliverables/deliverable-d2-10/

[16] Open FOAM - The Open Source CFD Toolbox, http://www.openfoam.com

[17] Intel Xeon Processor X5560 data sheet, http://ark.intel.com/products/37109/Intel-Xeon-Processor-X5560-8M-Cache-2_80-GHz-6_40-GTs-Intel-QPI

[18] MIKELANGELO Report D2.1 First Cancellous bone simulation Use Case Implementation strategy, https://www.mikelangelo-project.eu/deliverables/deliverable-d2-1/

[19] OSv operating system for the Cloud, http://osv.io

[20] Cloud-init datasources, http://cloudinit.readthedocs.org/en/latest/topics/datasources.html

[21] Torque Manual: Script Environment, http://docs.adaptivecomputing.com/torque/6-0-0/help.htm#topics/torque/13-appendices/scriptEnvironment.htm

[22] Torque Manual: Exported Batch Environment Variables, http://docs.adaptivecomputing.com/torque/6-0-0/help.htm#topics/torque/2-jobs/exportedBatchEnvVar.htm?Highlight=environment%20variables

[23] Virtual I/O Device (VIRTIO) Version 1.0, http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.pdf

[24] MIKELANGELO Report D4.4 OSv - Guest Operating System - First Version, https://www.mikelangelo-project.eu/deliverables/deliverable-d4-4/

[25] MIKELANGELO Report D4.7 First version of the application packages, https://www.mikelangelo-project.eu/deliverables/deliverable-d4-7/

[26] MIKELANGELO Report D5.1 First Report on the Integration of sKVM and OSv with Cloud Computing, https://www.mikelangelo-project.eu/deliverables/deliverable-d5-1/

[27] MIKELANGELO Report D5.7 First report on the Instrumentation and Monitoring of the complete MIKELANGELO software stack, https://www.mikelangelo-project.eu/deliverables/deliverable-d5-7/

[28] MIKELANGELO Report D7.17 The first update on the Data Management Plan, https://www.mikelangelo-project.eu/deliverables/deliverable-d7-17/

Appendix A - UC Cancellous Bones (Runtime Measurements)

Following table shows ten different individual measurement of the Cancellous Bones application (the HLRS Use Case) taken on bare metal hardware as well as taken on a virtualized Ubuntu guest system on top of KVM in HLRS' HPC Testbed.

Bare Metal		**Difference in %**	**Virtual Guest**	
Measurement	**Runtime in mm:ss.ms**		**Measurement**	**Runtime in mm:ss.ms**
Bare 1	25:10.825	**10,44 %**	VM 1	27:48.526
Bare 2	25:08.897	10,29 %	VM 2	27:44.218
Bare 3	25:11.058	10,27 %	VM 3	27:46.192
Bare 4	25:09.926	10,28 %	VM 4	27:45.100
Bare 5	25:13.084	9,66 %	VM 5	27:39.305
Bare 6	25:11.324	8,65 %	VM 6	27:22.007
Bare 7	25:09.160	9,04 %	VM 7	27:25.657
Bare 8	25:15.317	**8,04 %**	VM 8	27:17.132
Bare 9	25:10.293	10,15 %	VM 9	27:43.619
Average	**25:11.098**	**9,65 %**	**27:36.862**	

Hardware:

see section 5 HPC Test Environment

Software:

GCC 4.8.4, G++ 4.8.8, GNU FORTRAN 4.8.4, Open MPI 1.6.5, BLAS 3.5.0, Lapack 3.5.0, PETSC 3.3-p3

Input Data:

see D7.17[28]

Parameters:

```
# Restart option
```

```
restart='N'
# lower bounds of selected cube
lb1=0
lb2=0
lb3=0
# upper bounds of selected cube
ub1=7
ub2=7
ub3=1
# Physical domain size
pdsize=0.3
# Phi threshold for geometry extraction
phi_threshold=14250
# Average RVE strain
avg_rve_strain=1.E-06
# Isotropic Young's Modulus
e_modul=5600.
# isotropic Poisson's ratio
nu=0.3
# Element order on micro scale
elt_micro="HEX20"
# Element order on macro scale
elo_macro=1
# Step width for time monitoring
ii_mon=1
# FMPS Version to use
fmps_vers=401
# Number of CPUS to be used for FE-System
fecpus=2
# Amount of FE-result data to be written  (DEBUG or any other)
fe_res_out="PRODUCTION"
# Used system ( CLUSTER, CRAY, GAMES  or any other )
system="TESTBED"
# Execution Command for applications
aprun=""
# Execution Command for FE-System
ferun="mpirun -n "$fecpus
# Override sta file and toogle process parts
override_sta_file='N'
# **Calculation of global domain decomposition
calc_global_ddc='Y'
# ** Geometry setup
geom_setup='Y'
```

```
# ** Micromechanical FE-Simulations
fe_sim='Y'
# ** Calculation of averaged material data
calc_avg_mat='Y'
```

Appendix B - Templates

This appendix contains the templates used for the extension of Torque's workflow in order to run batch jobs inside virtual machines.

All these files contain some placeholders prefixed and suffixed by '__', e.g. '__RUID__' or '__HOSTNAME__'. These placeholders are replaced with the actual values before the submission to the real qsub command takes place. The only exception is the metadata that is generated inside the wrapped user prologue, since it requires the actual hostname (not available before the deployment).

B.1 Metadata Template for Debian derivatives (contextualization)

File: `metadata.debian`

```
#cloud-config

#
# set the hostname the same as the physical node, but prefix it
with 'v'
#
hostname: v__HOSTNAME__

#
# security, do not allow root and do not allow pw logins but SSH
keys only
#
disable_root: true
ssh_pwauth: false

#
# add the user and its group with exactly same UID/GID (crucial for
NFS access)
#
bootcmd:
  - groupadd -g __GROUP_ID__ __USER_NAME__
  -  useradd  -u  __USER_ID__  -g  __GROUP_ID__  -s  /bin/bash  -M
__USER_NAME__
  - groupadd -g 1002 nico
  - useradd -u 1002 -g 1002 -s /bin/bash -M nico
  - usermod -a -G sudo nico

#
# install missing packages
#

# update VM
package_upgrade: true
```

```
# install required NFS packages
packages:
 - nfs-common
 - libnfs1
 - libmetis5
 - libmetis-dev
 - openmpi-bin
 - libopenmpi-dev

#
# mount the same NFS shares as the physical node
#
mounts:
 - [ "storage01:/home", /home, "nfs", "rw,intr,noatime", "0", "0" ]
 -  [  "storage02:/storage/mikelangelo/ssd_data/opt",  /opt,  "nfs",
"rw,intr,noatime", "0", "0" ]
 - [ "storage02:/storage/mikelangelo/ssd_scratch", /scratch, "nfs",
"rw,intr,noatime", "0", "0" ]

#
# DNS
#
manage-resolv-conf: true
resolv_conf:
  nameservers:
      - 'first_nameserver'
      - 'second_nameserver'
  searchdomains:
      - first.domain.com
      - second.domain.com
  domain: domain.com
  options:
      option1: value1
      option2: value2
      option3: value3

#
# NTP
#
write_files:
  - path: /etc/ntp.conf
      content: |
      # Common pool
      server 0.pool.ntp.org
      server 1.pool.ntp.org
      server 2.pool.ntp.org
      server 3.pool.ntp.org

      # - Allow only time queries, at a limited rate.
      # - Allow all local queries (IPv4, IPv6)
```

```
      restrict default nomodify nopeer noquery limited kod
      restrict 127.0.0.1
      restrict [::1]

#
# ping the physical host, otherwise it cannot see the VM's IP with
the help of 'arp -an's
#
runcmd:
 - 'ping -c1 __HOSTNAME__'

# final_message
final_message: "The system is finally up, after $UPTIME seconds"
```

B.2 Wrapper Prologue Template (VM instantiation)

File: `vmPrologue.sh`

```
#!/bin/bash

# random unique ID
RUID=__RUID__

# timeout for VM boot / IP resolving in seconds
TIMEOUT=600

# do we run inside a batch job ?
if [ $# -lt 1 ] \
      && [ -z $PBS_JOBID ] ; then # no, manual execution (dev)
  # request missing parameters if not executed by Torque
  echo "usage: $(basename $0) <jobID>";
  exit 1;
fi

#get the job id
if [ -n "$PBS_JOBID" ]; then
  JOBID=$PBS_JOBID;
else
  JOBID=$1;
fi


#===============================================================#
#                                                               #
#                         VM Parameters                         #
#                                                               #
#===============================================================#
```

```
#
# mandatory VM parameters
#

# name of the VM (libvirt domain name) [mandatory parameter]
NAME=$JOBID;

# operating system image to boot (needs to be known)
IMG=__IMG__

# Absolute path to VM's disk
DISK=__DISK__


#
# generated VM parameters
#

# vm's UUID
UUID=""

# VM's MAC address (TODO one MAC is not sufficient if we have more
than one VM)
MAC=""


#
# optional VM parameters
#

# [optional] count of vitural cores
VCPUS=__VCPUS__

# [optional] RAM in MB
RAM=__RAM__

# [optional] VM's MetaData disk for VM contextualization
METADATA=__METADATA__

# [optional] VM's cpu architecture, default is x86_64
ARCH=__ARCH__

# [optional] VM's cpu architecture, default is kvm
HYPERVISOR=__HYPERVISOR__



#============================================================#
#                                                            #
```

```
#                          Script  Config                              #
#                                                                      #
#======================================================================#


# optional user epilogue to wrap
PROLOGUE_SCRIPT=__PROLOGUE_SCRIPT__

# our vm template with place holders
templateXML="/opt/torque/misc/templates/domain.xml";
templateForMetadata="/opt/torque/misc/templates/domain-fragment-
metadata.xml";
templateForDisk="/opt/torque/misc/templates/domain-fragment-
disk.xml";

# create VM's XML-definition
domainXML="$(echo ~/.$RUID/$JOBID.xml)"; #TODO delete in epilogue

#
LOCKFILES_DIR="$(echo ~/.$RUID/locks)";

#
RAMDISK=/ramdisk;

# the template to use for the MetaData generation
METADATA_TEMPLATE=__METADATA_TEMPLATE__

# the generated MetaData file
METADATA_DISK=$HOME/.$RUID/seed.img

# map for the vm paramers (used to replace the place holders in the
template)
declare -A parsedParams;

# debug set ?
if [ -z $DEBUG ]; then
  DEBUG=false;
fi



#======================================================================#
#                                                                      #
#                            FUNCTIONS                                 #
#                                                                      #
#======================================================================#


#--------------------------------------
```

```
#
#
usage() { #TODO -n|--name <name>, -f|--file <imageFile>, -c|--cpus
<vcpus>, -r|--ram <ram>, -d|--disksize <tmpdisksize> -t|--template
<templateName>
  echo " usage: $(basename $0) <VMName> <VMImageFile> [<vcpus>]
[<ram>] [tmpDiskSize]";
  exit 1;
}


#--------------------------------------
#
#
validateParameter() {

  #
  # check parameters and generate missing optional ones
  #

  # mandatory parameters
  if [ -z $NAME ] \
      && [ -z $IMG ] \
      && [ -z $DISK ]; then
      exit 1;
  fi

  # readable image file ?
  if [ ! -f $IMG ] \
      || [ ! -r $IMG ]; then
      echo "[PROLOGUE:ERROR] The image file to boot '$IMG' cannot
be read."
      exit 1;
  fi

  # metadata disk given ?
  if [ -z $METADATA ] || [ "$METADATA" == "__METADATA__" ]; then
      METADATA=""; # TODO use a default one ?
  fi
  $DEBUG && echo " METADATA='$METADATA'";

  # parameters to generate
  if [ -z $UUID ] || [ "$UUID" == "__UUID__" ]; then
      UUID="$(uuidgen)"; # generate
      $DEBUG && echo " Generated UUID='$UUID'.";
  fi
  if [ -z $MAC ] || [ "$MAC" == "__MAC__" ]; then
      # http://superuser.com/questions/218340/how-to-generate-a-
valid-random-mac-address-with-bash-shell
      hexchars="0123456789ABCDEF"
```

```
      end=$( for i in {1..6} ; do echo -n ${hexchars:$(( $RANDOM %
16 )):1} ; done | sed -e 's/\(..\)/:\1/g' );
      MAC="52:54:00$end"; # generate, use prefix '52:54:00'
      $DEBUG && echo " Generated MAC='$MAC'.";
  fi

  # optional parameters
  if [ -z $RAM ] || [ "$RAM" == "__RAM__" ]; then
      RAM="24576"; # default=24GB if not given
  fi
  $DEBUG && echo " RAM='$RAM'";

  if [ -z $VCPUS ] || [ "$VCPUS" == "__VCPUS__" ]; then
      VCPUS="8"; # default=8 if not given
  fi
  $DEBUG && echo " VCPUS='$VCPUS'";

  if [ -z $DISK ] || [ "$DISK" == "__DISK__" ]; then
      DISK=""; # default none
  fi
  $DEBUG && echo " DISK='$DISK'";

  if [ -z $ARCH ] || [ "$ARCH" == "__ARCH__" ]; then
      ARCH="x86_64"; # default x86_64
  fi
  $DEBUG && echo " ARCH='$ARCH'";

  if [ -z $HYPERVISOR ] || [ "$HYPERVISOR" == "__HYPERVISOR__" ];
then
      HYPERVISOR="kvm"; # default kvm
  fi
  $DEBUG && echo " HYPERVISOR='$HYPERVISOR'";


  # mandatory
  parsedParams["NAME"]=$NAME;
  parsedParams["IMG"]=$IMG;
  # generated
  parsedParams["MAC"]=$MAC;
  parsedParams["UUID"]=$UUID;
  # optional
  parsedParams["RAM"]=$RAM;
  parsedParams["VCPUS"]=$VCPUS;
  parsedParams["DISK"]=$DISK;
  parsedParams["METADATA"]=$METADATA;  #TODO: only metadata yaml
files are allowed; implement merging
  parsedParams["ARCH"]=$ARCH;
  parsedParams["HYPERVISOR"]=$HYPERVISOR;

  $DEBUG && echo -e "Parameter:\n ${parsedParams[*]}";
```

```
  return 0;
}


#--------------------------------------
#
#
#
generateMetaData() {

  if [ -n ${parsedParams["METADATA"]} ]; then
      #TODO (?): user provided meta data
      echo -e "Metadata is given by the user:
'"${parsedParams["METADATA"]}"'\n  Will be ignored for now and
replaced by a generated one..";
  fi

  # OSv has no metadata image (?)
  if [ -z $METADATA_TEMPLATE ]; then
      $DEBUG && echo "OSv image assumed since no metadata template
known.";
      # abort here
      return true;
  elif [ "$METADATA_TEMPLATE" == "__METADATA_TEMPLATE__" ]; then
      $DEBUG && echo "Something went wrong, the metadata
placeholder is still there ?!";
      # abort here
      return true;
  fi

  # use hidden tmp file in user's home
  metadataFile=$HOME/.$RUID/metadata;

  # get user's group id
  groudID=$(getent group $USER | grep -o [0-9]*);

  # copy template
  cp $METADATA_TEMPLATE $metadataFile;

  # substitute values
  sed -i "s,__HOSTNAME__,$(hostname -s),g" $metadataFile;
  sed -i "s,__GROUP_ID__,$groudID,g" $metadataFile;
  sed -i "s,__USER_ID__,$(id -u),g" $metadataFile;
  sed -i "s,__USER_NAME__,$USER,g" $metadataFile;

  $DEBUG && echo -e "generated METADATA file\n-----\n$(cat
$metadataFile)\n-----";

  # generate image file from tmpFile [cloud-localds my-seed.img my-
```

```
user-data my-meta-data]
  cloud-localds $METADATA_DISK $metadataFile;

  # remove tmp file (if not debugging)
  $DEBUG || rm -f $metadataFile;

  # store in global array
  parsedParams["METADATA"]=$METADATA_DISK;

  $DEBUG && echo -e " METADATA DISK = '${parsedParams[METADATA]}';
size = '$(du -sh ${parsedParams[METADATA]})'\n-----";
}


#--------------------------------------
#
#
prepareVMboot() {

  success=false;

  echo -e "Preparing boot.\n Copying image to RAMdisk ...";

  # copy the VM image there for faster access
  if [ ! -d $RAMDISK ]; then
      echo -e "\n[PROLOGUE:WARNING] RAMdisk not found '$RAMDISK'.";
      #FIXME: the image file should be readonly, we need to copy it
somewhere and make it writeable
  else
      if [ -n "${parsedParams[IMG]}" ]; then
      rsync --progress -L ${parsedParams[IMG]} $RAMDISK;
      chmod +w $RAMDISK/$(basename ${parsedParams[IMG]});
      parsedParams["IMG"]=$RAMDISK/$(basename
${parsedParams[IMG]});
      $DEBUG && echo " Copied image file to faster RAMdisk.";
      else
      echo -e "\n[PROLOGUE:ERROR] Mandatory parameter 'img' is
missing";
      exit 1;
      fi
      if [ -n "${parsedParams[METADATA]}" ]; then
      rsync --progress -L ${parsedParams[METADATA]} $RAMDISK;
      chmod +w $RAMDISK/$(basename ${parsedParams[METADATA]});
      parsedParams["METADATA"]=$RAMDISK/$(basename
${parsedParams[METADATA]});
      $DEBUG && echo " Copied metadata file to faster RAMdisk.";
      else
#     echo -e "\n[PROLOGUE:ERROR] Mandatory parameter 'metadata' is
missing";
      $DEBUG && echo " No metadata disk found to copy.";
```

```
#     exit 1;
      fi
      if [ -n "${parsedParams[DISK]}" ]; then
      rsync --progress -L ${parsedParams[DISK]} $RAMDISK;
      chmod +w ${parsedParams[DISK]};
      parsedParams["DISK"]=$RAMDISK/$(basename
${parsedParams[DISK]});
      $DEBUG && echo " Copied optional disk to faster RAMdisk.";
      else
      $DEBUG && echo " No persistent disk found to copy.";
      fi
  fi
  return 0;
}


#--------------------------------------
#
#
generateDomainXML() {

  # construct the template for the provided parameter combination
  cp $templateXML $domainXML;

  # is there a metadata file ?
    if [ -n "${parsedParams[METADATA]}" ]; then
      sed -i "s,__METADATA_XML__,$(cat $templateForMetadata | tr
'\r\n' ' '),g" $domainXML;
  else
      sed -i "s,__METADATA_XML__,,g" $domainXML;
  fi

  # is there a persistent disk ?
  if [ -n "${parsedParams[DISK]}" ]; then
      sed -i "s,__DISK_XML__,$(cat $templateForDisk | tr '\r\n' '
'),g" $domainXML;
  else
      sed -i "s,__DISK_XML__,,g" $domainXML;
  fi

  # replace place holders
    for key in "${!parsedParams[@]}"; do
      $DEBUG && echo "Replacing: key=$key,
value='${parsedParams[$key]}'";
      sed -i "s,__"$key"__,${parsedParams[$key]},g" $domainXML;
    done

    return 0;
}
```

```
#--------------------------------------
#
#
bootVM() { #TODO multiple VMs per node ?
  #
    echo "Booting VM from domainXML='$domainXML'."
    virsh create $domainXML;

  # check if it's running
    if [ $? -ne 0 ]; then
      echo "FAILED.";
      # abort the job
      exit 1;
    fi
    $DEBUG && echo "VM is running.";

    # wait for all local VMs to become available
    runningVMs=$(virsh list --all | grep [0-9] | awk '{print $2}');
    for vm in $runningVMs; do
      mac=$(virsh dumpxml $vm | grep 'mac address' | cut -d"'" -f
2);
     waitForVMtoBecomeAvailable $mac & continue;
  done
}


#--------------------------------------
#
#
waitForVMtoBecomeAvailable() {

  mac=$1;

  # create lock file
  mkdir -p $LOCKFILES_DIR 2>/dev/null;
  touch "$LOCKFILES_DIR/$mac";

  # wait until it the VM has requested an IP
  counter=0;
  arpOut=$(arp -an | grep "$mac");
  startDate=$(date +%s);
  while [ ! -n "$arpOut" ]; do
      echo "VM with MAC='$mac' has not requested an IP, yet.";
      # max iterations reached ?
      if [ $TIMEOUT -lt $(expr $startDate - $(date +%s)) ]; then
      echo "Aborting. Timeout of '$TIMEOUT' sec reached and VM is
still not available.";
      # abort the job
      exit 1; #FIXME: cleanup; i.e. kill all running VMs
```

```
      fi
      sleep 2;
      arpOut=$(arp -an | grep "$mac");
      counter=$(expr $counter + 1);
  done
  vmIP=$(echo $arpOut | cut -d' ' -f2 | sed 's,(,,g' | sed
's,),,g');

  # now wait until the VM becomes available via SSH
  counter=0;
  while [ $(ssh -q -n -o "BatchMode=yes" -o "ConnectTimeout=10"
$vmIP exit; echo $?) -eq 255  ]; do
      sleep 2;
      counter=$(expr $counter + 1);
      $DEBUG && echo "Waiting for VM to become available..";
      #
      if [ $counter -ge 100 ]; then
      echo "Aborting. VM still not available.";
      # abort the job
      exit 1; #FIXME: cleanup; i.e. kill all running VMs
      fi
  done

  # success
  echo "VM is now available: '$vmIP'";
  # remove lock file
  rm -f "$LOCKFILES_DIR/$mac";
}


#---------------------------------------
#
#
waitUntilReady() {
  sleep 1;
  # any locks remaining in the shared fs ?
  while [ -n "$(ls $LOCKFILES_DIR)" ]; do
      sleep 1;
      $DEBUG && echo "Waiting for locks to disappear..";
  done
  rm -Rf $LOCKFILES_DIR;

  # list running VMs ?
  if $DEBUG; then
      virsh list --all;
  fi
}


#TODO trap for signals (otherwise we can run the epilogue, cancel
```

```
it, but the the checking VM is still running


#=============================================================#
#                                                             #
#                              MAIN                           #
#                                                             #
#=============================================================#


# check the place holders and ensure valid values
validateParameter;

# generate the required meta data
generateMetaData;

# prepare the image for booting
prepareVMboot;

# generate the VM's domain.xml
generateDomainXML;

# start the VM
bootVM \

# wait for VMs
& waitUntilReady;

# prologue script given ?
if [ -x  "$PROLOGUE_SCRIPT" ]; then
  $DEBUG && echo "Running now user's prologue..";
  $PROLOGUE_SCRIPT;
  exit $?
fi
# run the job
exit 0;
```

B.3 Job Wrapper Template (preparation of vNode and Job start)

File: `jobWrapper.sh`

```
#!/bin/bash

#############################################################
#                                                           #
#               GENERATED VALUES / VARIABLES                #
#                                                           #
```

```
#####################################################################

#
RUID=__RUID__

#
JOB_SCRIPT=__JOB_SCRIPT__

#
JOB_TYPE=__JOB_TYPE__

#
PBS_VM_NODEFILE="~/.$RUID/pbs_nodefile.$PBS_JOBID.$PBS_O_HOST"

#
PBS_ENV_FILE="~/.$RUID/$PBS_JOBNAME\_ENV"

# Debugging ?
if [ -z $DEBUG ];
  DEBUG=false;
fi




#####################################################################
#                                                                   #
#                                FUNCTIONS                          #
#                                                                   #
#####################################################################

#
#
#
collectVmIPs(){
  # collect the ips of all vms and write it to a file
  $DEBUG && echo -e '\n-----------------------------------------'
  $DEBUG && echo -e   '-             Collect VM IPs             -'
  $DEBUG && echo -e   '-----------------------------------------
\n'

  for nodeName in $(cat $PBS_NODEFILE); do
      # get mac adresses
      for vm in $(ssh $nodeName "virsh list --all | grep [0-9] |
awk '{print \$2}'"); do
      mac=$(ssh $nodeName virsh dumpxml $vm | grep 'mac address' |
cut -d"'" -f 2)
      # get ips to mac adresses
      for vm_mac in $mac; do
```

```
      echo $(ssh $nodeName "arp -an | \
                                grep '$vm_mac' | \
                                cut -d' ' -f2 | \
                                sed 's,(,,g' | \
                                sed 's,),,g'") \
                                >> $PBS_VM_NODEFILE
      done
      done
  done
  # for simplifying the executen of the job script this will
  local tmpVar=(${PBS_VM_NODEFILE[@]})
  PBS_FIRST_VM=${tmpVar[0]}
  $DEBUG && echo 'PBS_VM_NODEFILE contend \n'; cat $PBS_VM_NODEFILE
}

#
#
#
generateEnvFile(){
  # this generates a file which can be sourced on the V_Enviroment
  # to have all the pbs values

  $DEBUG && echo -e '\n-----------------------------------------'
  $DEBUG && echo -e   '-        Generating enviroment           -'
  $DEBUG && echo -e   '-----------------------------------------
\n'

  echo -e "\
export PBS_NODEFILE='$PBS_VM_NODEFILE'\n\
export PBS_O_HOST='$PBS_O_HOST'\n\
export PBS_O_QUEUE='$PBS_O_QUEUE'\n\
export PBS_QUEUE='$PBS_QUEUE'\n\
export PBS_O_WORKDIR='$PBS_O_WORKDIR'\n\
export PBS_ENVIRONMENT='$PBS_ENVIRONMENT'\n\
export PBS_JOBID='$PBS_JOBID'\n\
export PBS_JOBNAME='$PBS_JOBNAME'\n\
export PBS_O_HOME='$PBS_O_HOME'\n\
export PBS_O_PATH='$PBS_O_PATH'\n\
export PBS_VERSION='$PBS_VERSION'\n\
export PBS_TASKNUM='$PBS_TASKNUM'\n\
export PBS_WALLTIME='$PBS_WALLTIME'\n\
export PBS_GPUFILE='$PBS_GPUFILE'\n\
export PBS_MOMPORT='$PBS_MOMPORT'\n\
export PBS_O_LOGNAME='$PBS_O_LOGNAME'\n\
export PBS_O_LANG='$PBS_O_LANG'\n\
export PBS_JOBCOOKIE='$PBS_JOBCOOKIE'\n\
export PBS_NODENUM='$PBS_NODENUM'\n\
export PBS_NUM_NODES='$PBS_NUM_NODES'\n\
export PBS_O_SHELL='$PBS_O_SHELL'\n\
export PBS_VNODENUM='$PBS_VNODENUM'\n\
export PBS_MICFILE='$PBS_MICFILE'\n\
```

```
export PBS_O_MAIL='$PBS_O_MAIL'\n\
export PBS_NP='$PBS_NP'\n\
export PBS_NUM_PPN='$PBS_NUM_PPN'\n\
export PBS_O_SERVER='$PBS_O_SERVER'\n" > $PBS_ENV_FILE;
}

#
#
#
testVM(){
  if ! $$DEBUG; then
     return 0;
  fi

  # print out debugging info
$DEBUG && echo -e "\
\n-------------------------------------------\n\
-               DEBUG INFO            -\n\
-------------------------------------------\n\
\$PBS_ENV_FILE='$PBS_ENV_FILE'\n---------";
  cat $PBS_ENV_FILE;
  echo -e "---------\n";

  echo -e "vNodes:\n---------";
  for vNodeName in $(cat $PBS_VM_NODEFILE | cut -d' ' -f1); do
     echo "vNode's hostname: $vNodeName'
     echo -e "vNode's environment: \n"
     ssh -q -o timeout=10 $vNodeName \
     "source /etc/profile; source $PBS_ENV_FILE; env; echo -e
'PBS_NODE_FILE:\n-----\$\(cat \$PBS_NODE_FILE\)\n-----\n'";
  done
}

#
#
#
runJob(){
  # Run Program here ...
$DEBUG && echo -e "\
\n-------------------------------------------\n\
-               RUN BATCH JOB              -\n\
-------------------------------------------\n\
JobScript: '$JOB_SCRIPT'\n";
  ssh $vNodeName "source \$PBS_ENV_FILE; $JOB_SCRIPT; echo \$?";
#TODO: ssh -t ?
  #FIXME: ensure that not a SSH error code is returned !
  return $?;
}

#
#
```

```
#
runSTDinJob(){
  # construct cmd to execute
  cmd="source /etc/profile; source \$PBS_ENV_FILE; $@; echo \$?";
  # Run STDIN Job here ...
  $DEBUG && echo -e "\
\n-----------------------------------------\n\
-               RUN STDIN JOB              -\n\
-----------------------------------------\n\
CMD: '$cmd'\n";
  echo $cmd | ssh $PBS_FIRST_VM; #TODO: ssh -t ?
  #FIXME: ensure that not a SSH error code is returned !
  return $?;
}


#####################################################################
#                                                                   #
#                                MAIN                               #
#                                                                   #
#####################################################################

#
collectVmIPs;

#
generateEnvFile;

#
$DEBUG && testVM;

# what kind of job is it ? (interactive, STDIN, batch-script)
if [ "$JOB_TYPE" == "@BATCH_JOB@" ]; then
  $DEBUG && echo "we are running a batch-script job now..";
  runJob;
  JOB_EXIT_CODE=$?
  $DEBUG && echo "done"
elif [ "$JOB_TYPE" == "@STDIN_JOB@" ]; then
  $DEBUG && echo "STDIN job";
  runSTDinJob $@;
  JOB_EXIT_CODE=$?
  $DEBUG && echo "done"
elif [ "$JOB_TYPE" == "@INTERACTIVE_JOB@" ]; then
  # in this case we run in an interactive session, keep that in
mind
  # TODO
  # maybe it works with while loop for readline; with pipe/redirect
for each line > into/outof SSH's **bash** (not sh)
  echo "Not supported yet";
  exit 1;
```

```
else
  echo "ERROR: unkown type of JOB '$JOB_TYPE'!";
  exit 1;
fi

# return job exit code
exit $JOB_EXIT_CODE;
```

B.4 Epilogue Wrapper Template (VM clean up)

File: `vmEpilogue.sh`

```
#!/bin/bash

# random unique ID
RUID=__RUID__

# optional user epilogue to wrap
EPILOGUE_SCRIPT=__EPILOGUE_SCRIPT__


#
# Main
#

# manual execution or via Torque ?
if [ $# -lt 1 ] \
      && [ -z $PBS_JOBID ] ; then #manual
  # required if not executed by Torque
  echo "usage: $(basename $0) <jobID>";
  exit 1;
fi

#get the job id
if [ -n "$PBS_JOBID" ]; then
  JOBID=$PBS_JOBID;
else
  JOBID=$1;
fi

# virsh domain name
NAME=$JOBID;

# shutdown VM
$DEBUG && echo "Stopping VM '$NAME'";
virsh shutdown $NAME;
```

```
#TODO wait for clean shutdown via while [ "$(virsh list | grep
domain | cut -d' ' -f..)" != "" ]; do sleep 1; done

# destroy VM
virsh destroy $NAME;

# user epilogue script wrapped ?
if [ -x  "$EPILOGUE_SCRIPT" ]; then
  $DEBUG && echo "Executing user's epilogue script now..";
  $EPILOGUE_SCRIPT;
  exit $?
fi

exit 0;
```