Documentation for Virtual Allocation for Parallel Applications (VALPA)

Giacomo Victor Mc Evoy Valenzano[1]

[1]ComCiDis – Laboratório Nacional de Computação Científica (LNCC)

`giacomo@lncc.br`

1. Introduction

When reviewing related work, we did not find a satisfactory software solution that supports cluster topology definition and specification of core mappings, along with a provenance system that can relate execution output to this characterization. In this context, we propose *VALPA* (*V*irtual *AL*location for *P*arallel *A*pplications), a virtual cluster manager designed to systematically execute parallel applications on virtual clusters. VALPA is oriented initially to an **experimental environment**, generating virtual clusters with the purpose of extracting performance measurements for **researching** about HPC workloads running on virtual clusters.

Sections 2, 3, and 4 describe the context, objectives and architectural details for VALPA, respectively. Section 5 shows a procedure for installing VALPA in a physical cluster.

1.1. Terminology

Physical cluster: an aggregation of independent physical machines that are connected with a high-speed network.

Virtual cluster: a cluster of VMs, instantiated within a physical cluster. A virtual cluster is meant to host a single application, but multiple virtual clusters may co-exist within a physical cluster.

Run: a single instance of an application program.

Execution: an instance of the life-cycle of a virtual cluster. Consists of the instantiation of a virtual cluster, followed by one or more application *runs* on said virtual cluster, ending with the removal of said virtual cluster.

Experiment: A set consisting of one or more *executions*, grouped as a unit. Each experiment runs independently from other experiments. However, *executions* within the *experiment* are instantiated concurrently, possibly sharing physical resoures.

Listing 1. Example of command (auser@somehost)

```
issue this command in the console, possibly adapting variables
```

Listing 2. Example of file contents / output

```
Either the required contents of a file, or an example of console output
```

2. Context

The allocation of VMs to physical resources is an essential aspect of Cloud Provisioning. Given a request of a virtual cluster to a Cloud environment, where n VMs are to be instantiated, the VM placement process typically consists in assigning one physical machine (PM), from the set of available PMs, to each requested VM. The task of assigning physical resources is complex when considering the internal capabilities of each PM, as each VM may map its resources differently. In order to build a conceptual model that describes the process of VM allocation, the process needs first to be formally formulated; this implies restricting the definition of what is considered a proper VM allocation, as well as stating the extent of the model. The assumptions for the representation problem are then stated as follows:

1. *Applications consist of a variable set of processes, which are deployed in one or more VMs on execution.* Each VM instance hosts processes corresponding to a single application. This assumption means that the model allows the existence of concurrent, multi-threaded applications deployed in virtual clusters, with the restriction that VMs cannot be shared by applications. This restriction is consistent with the notion of dedicating the lifetime of a virtual cluster instance to a single application. However, the representation of subsequent applications executed afterwards on the same virtual cluster instance is supported, if the model treats the new executions as being deployed in another, identical virtual cluster instance.

2. *Processes of a given application are either equal (SPMD approach) or reflect the Master/Worker paradigm.* This assumption simplifies the model, and initially leaves out more elaborate deployments such as distributed workflows. It is adequate for all Bag of Tasks (BoT) applications and many scientific problems; this can be verified, for example, by the MPI implementations of the Computational Fluid Dynamics (CFD) found in the NAS Parallel Benchmarks, presented by [Bailey et al. 1994]. Additionally, an application process may not be migrated to other VMs once it executes, but the model allows the possibility of migrating a VM to a different host at runtime.

3. *Resource consumption of a VM is dependent on the spatial distribution of other collaborating VMs.* This means that the performance of a single VM in the virtual cluster can be understood as a function of the resource mappings of other VMs. This is specially true in communication-intensive applications, where throughput is sensitive to message latency, which is in turn a function of the memory and/or network topologies.

4. *Process performance can be affected by resource consumption of other VMs in the same host.* This item, together with the previous one, are statements for the support of the model in expressing the effects of VM placement on performance.

5. *Virtual Machine Manager settings used to instantiate the VMs should be registered for each request.* As shown in technical reports and studies such as those by [Hat 2011] and [Abeni et al. 2013], configuration settings on the VMM —outside VM profile definition— can also have a significant impact on the performance

of sensitive workloads such as scientific applications. Therefore, a representation model should also indicate the values of key VMM parameters that were used.

6. *Mapping of a virtual core can be manually restricted to a subset of the available physical cores.* This requires access to the VMM by either the client issuing the request for VMs or by the scheduler in a multi-tenant Cloud for HPC.

In relation to the last item, application processes are scheduled by the VM's operating system to execute on virtual cores, which in turn are mapped by the VMM onto physical cores. Thus, each CPU instruction of the application code ends up being executed on one of the physical cores. The VMM will consider a subset of the physical cores in a server, for a process due to execute on a virtual core. At any instant however, exactly one physical core is assigned to each virtual core, and the actual physical core is chosen by the VMM scheduler according to a set of criteria. For further reference, the study by [Cherkasova et al. 2007] compares three schedulers for the Xen hypervisor.

By default, all physical cores in a server are eligible for a virtual core allocation, but this can be changed —either via settings in the virtualization solution or indirectly by process affinities where applicable— to a subset of the available cores. This capability is known as *vcpu-pinning*, and was explored in our previous work, [Mc Evoy and Schulze 2011]. With *vcpu-pinning*, the scheduler that manages virtual core allocation chooses from one of the pre-specified candidates. In the current work, a virtual core is said to be mapped to these physical cores. The proposed representation of VM placement is restricted to registering these mappings, omitting the actual physical core chosen by the VMM. Likewise, the model ignores actual scheduling within a VM by its OS, i.e. how the application processes are allocated to the virtual cores throughout time.

Using *vcpu-pinning* to restrict virtual core mappings may improve performance in some conditions, specially with compute-intensive workloads, as shown in [Mc Evoy and Schulze 2011]. For instance, one-to-one mappings between virtual cores and physical cores can avoid context switching. Also, restricting the virtual cores of a VM to a single processor socket can reduce idle processing time, while maintaining a chance of shared cache hits. The current proposal assumes that the allocation of a VM is fully defined by the allocation of each of its cores. This representation may not be sufficient for other resources besides processing power, such as the linking of storage capacity from different hosts to a VM. However, this work is based on the hypothesis that the proposed representation for resource allocation is rich enough for performance models with fair accuracy given some HPC workloads. This hypothesis will be later validated by the experimental evaluation of the generated performance models.

2.1. General Characterization

This section presents a general-purpose characterization for virtual clusters, comprised of features that are classified in different aspects. In this proposal, a virtual cluster instance is characterized in part by the `Cluster Placement`. Instead of the traditional approach of stating the hosting node for each VM, the `Cluster Placement` describes the resource mappings between each of the virtual cores in the virtual cluster and the processor cores
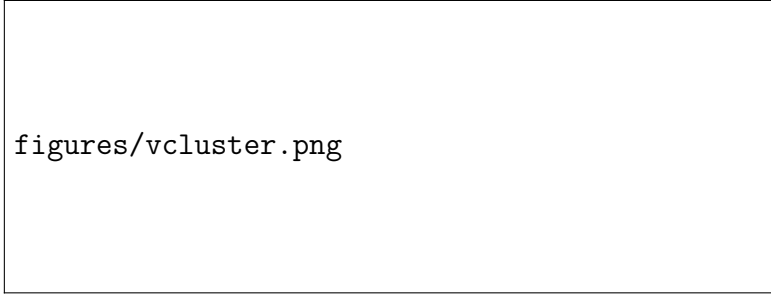
Figure 1. Hierarchical view of proposed characteristics of a virtual cluster.

of the physical cluster.

Additionally, configuration settings of the Virtual Machine Manager (VMM) solution are included, grouped in the `VMM Configuration` item as a feature of the resulting virtual cluster. Each aspect is refined in two subgroups, as shown in Figure 1, and they are described as follows:

- *Cluster Placement:* Unambiguously describes how each virtual core in the cluster is mapped onto physical cores. Given the set of virtual cores N_v, each of the $v \in N_v$ virtual cores is related to a subset of the available physical cores, given by the tuple $\langle r \rangle \in G_r$. The `map()` predicate that describes the temporal mapping between virtual and physical cores can be defined as:

$$map(v, \langle r \rangle, i) \iff (v, r, i) \in \texttt{Core Mapping};$$
$$v \in N_v, \langle r \rangle \in G_r, i \in I_t. \tag{1}$$

The predicate $map(v, \langle r \rangle, i)$ will be true if a virtual core is mapped to the group of physical cores $\langle r \rangle$ during the time interval i. When this happens, the `Core Mapping` entity will contain the corresponding tuple. This predicate is used to describe virtual core mappings resulting from *vcpu-pinning* allocations. The *map* predicate is equivalent to the mapping function mf, given by:

$$mf : N_v \times I_t \to G_r, \, mf(v, i) = \langle r \rangle. \tag{2}$$

The `Cluster Placement` can be further divided into two concepts:

- *Cluster Topology:* Describes the arrangement of cores in the same way as a physical cluster, details of this description are visible to applications deployed inside the virtual cluster. It states how many virtual cores v_{ij} belong to each VM V_i, and can be regarded as a specification of the domain N_v of the mapping function mf of the `Cluster Placement`.
- *Physical Mappings:* Correspond to details of the core placement that are not visible to applications deployed inside the virtual cluster, e.g. settings for pinning virtual cores to a subset of available physical cores. It states the size and elements of the physical cores $\langle r \rangle$ associated to each virtual core v_{ij}, and when these associations occur. Thus, it can be regarded as the specification of the range of mf and how it maps the domain $N_v \times I_t$.

- *VMM Configuration:* Consists of settings in the Virtual Machine Manager that affect how physical resources are virtualized and presented to applications in the virtual cluster, but do not affect the virtual core placement. Therefore, these settings are not directly visible to applications deployed in the virtual cluster. Comprises two concepts:
  - *Virtualization Technologies:* Lists the virtualization technologies chosen to enable devices in the VMs. For instance, states the virtualization solution for networking, as being one item in the set `{virtio, vhost, SR-IOV}`. Other options include choosing network card emulation, disk emulation, or enabling technologies such as Intel EPT and VT-x.
  - *VMM Tuning:* Lists configuration values that may improve virtualization performance for a given VM. These may be numerical values, as is the case when specifying CPU scheduling intervals and weights; as well as categorical values, e.g. when defining the CPU model that the VM will use. For a detailed list of possible tuning values, refer to the *libvirt* documentation[1].

2.2. Simplified Characterization

In order to facilitate the analysis of experiments in this work, the `Cluster Placement` description will be simplified, by reducing the number of different virtual cluster instances with additional constraints on the specification of core placements. From this point onward, given a physical cluster and the possible virtual clusters deployed in it, the following restrictions are assumed:

1. The physical cluster is composed of homogeneous physical machines (PMs), interconnected with a dedicated, balanced network. Therefore, choosing one PM over another in order to execute a dedicated workload should have no effect on performance.
2. The software layers of the physical cluster are equal for all PMs. In addition to the PMs running the same virtualization solution, all operating system (OS) settings relevant to virtualization are equal for all PMs. These include global VMM settings, networking and memory tuning. While simplifying environment specification, this restriction also improves execution stability.
3. A virtual cluster is comprised of homogeneous VM nodes. In particular, all nodes of a virtual cluster have the same number of virtual cores.
4. All PMs hosting at least one VM will host the same number of VMs. Combined with the previous constraint, PMs hosting at least one virtual core will all host the same number of virtual cores, although idle PMs are allowed.
5. The core allocations are built based on a few assignment schemes that fully identify, for each virtual core, the mapped physical cores. To achieve this, one pinning strategy (described below) is chosen, and the eligible processor sockets affected are stated (all sockets by default).
6. The description of physical mappings, virtualization technologies and VMM tuning are homogeneous throughout the virtual cluster. For instance, the pinning strategy is maintained for all VMs in a virtual cluster. These values are allowed to change for different virtual clusters, however.

These additional constraints are consistent with deployments were each node in the cluster has a similar workload, such as when using the *ppn* (processes per node) variable in PBS. An important restriction is that the virtual core mappings will not change after deployment, i.e. the pining strategy will be held constant over time for each VM. As a consequence, the mapping function mf given by eq. 2 can be simplified to $mf : N_v \rightarrow G_r, mf(v) = \langle r \rangle$.

Now, describing a `Cluster Placement` can be done with a few variables; specifically, the `Cluster Topology` of a cluster can be described with only two variables:

- **Number of cores (`nc`):** the total number of virtual cores deployed in the cluster. Assuming that each application process will be assigned to one virtual core, this matches the total number of processes *np*.

- **Number of cores per VM (`cpv`):** the number of virtual cores for each VM, held constant for a virtual cluster, due to constraint 3.

The *Physical Mappings* need more variables, even with the aforementioned constraints. The two primary variables are listed first:

- **Dispersion factor (`df`):** a value ($df \in [0, 1]$) that measures the consolidation or scattering of the VMs along physical machines. Let *vpm* be the total number of virtual cores executing on each PM, `df` is thus given by

$$df = \begin{cases} 0 & \text{if } vpm = nc \\ \frac{1}{vpm} & \text{otherwise} \end{cases} \qquad (3)$$

Let *ppm* be the total number of physical cores in every PM, a value of $df = 1/ppm$ thus indicates that the VMs are consolidated in as few physical machines as possible, unless the virtual cluster is deployed in a single physical machine, which is denoted as $df = 0$. A value of $df = 1$ indicates that the VMs have a single core each and have maximum spread over the PMs. This definition enables comparison between deployments under different physical topologies.

- **Pinning strategy (`pstrat`):** the scheme used to map virtual cores to physical cores. By default, all physical cores in a PM are eligible to execute processes assigned to a virtual core. Pinning strategies rely on virtual core pinning, which specifies a subset of physical cores that are eligible, requiring access to the hypervisor. A pinning strategy is relevant in multi-core architectures, where CPU cores are grouped in processors (sockets). While there are many ways to realize the mappings through virtual core pinning, we propose and study five strategies, defined as follows:
    - `NONE` strategy: represents the absence of pinnings and the default behavior. Each virtual core can be scheduled to any physical core at any time, managed by the virtualization solution.
    - `GREEDY` strategy: the VMs are consolidated in as few sockets as possible. Virtual cores have one-to-one mappings to physical cores. This strategy is

potentially advantageous for cache reuse and disadvantageous due to cache contention.

- BAL-ONE strategy: enforces load balancing between sockets, but each VM is constrained to one socket if possible. Virtual cores have one-to-one mappings to physical cores. This strategy theoretically reduces cache contention but offers less cache reuse.
- BAL-SET strategy: similar to BAL-ONE, but each virtual core in VM_i is mapped onto the set $\langle r \rangle_i$ of all the physical cores in a single socket that results from using BAL-ONE (does not group cores from different sockets). As a result, virtual cores from the same VM that are assigned to the same processor socket, then have the same physical core mappings. This strategy theoretically offers more cache reuse than the BAL-ONE strategy, but may produce more CPU context switching.
- SPLIT strategy: the virtual cores of each VM are spread throughout the sockets, with one-to-one mappings to physical cores. This strategy theoretically offers the fairest load balance but has low chance of cache reuse.

Aside from the default *NONE* strategy, there are no overlappings between mappings of different VMs. As a consequence of these definitions, strategies overlap on the following scenarios: a) with $cpv = 1$, BAL-ONE $\equiv$ BAL-SET $\equiv$ SPLIT; b) with one VM per machine, BAL-ONE $\equiv$ GREEDY; c) with $cpv = ppm$, BAL-ONE $\equiv$ SPLIT. Figure 2 shows an example where two VMs with two cores each are deployed on two quad-core processors, using each of the proposed strategies.
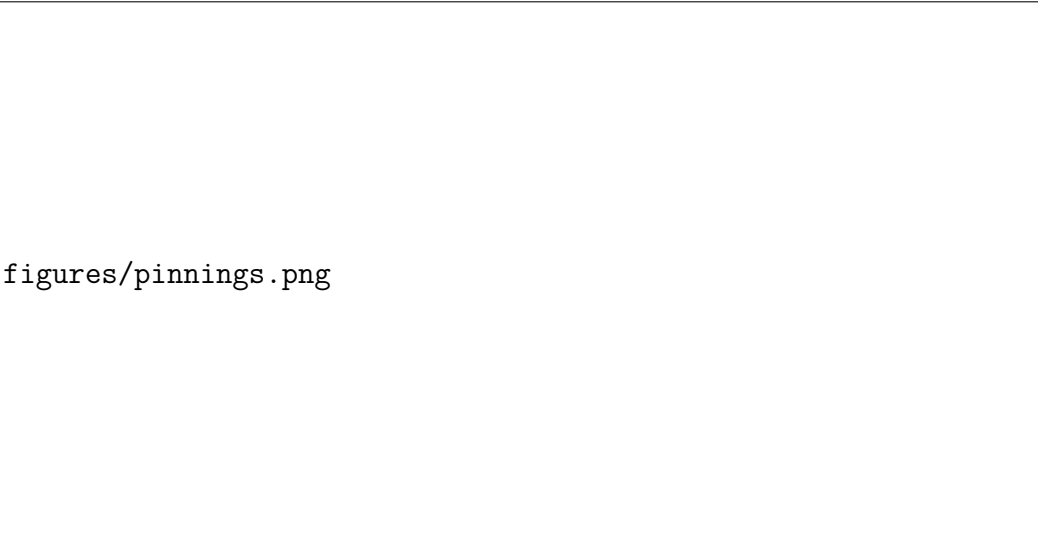

Figure 2. Diagram showing the different pinning strategies on two quad-core processors.

The following secondary variables help to complete the description of the Physical Mappings of a virtual cluster:

- the identities of the PMs that are hosting VMs for the virtual cluster;

- the identities of the eligible processor sockets, which affect the identities of resulting physical cores $\langle r \rangle_i$ for each virtual core v_i.

The secondary distribution variables play a role only when deploying concurrent virtual clusters. If the physical environment is homogeneous and only one virtual cluster is created, then these variables are not necessary to define a `Cluster Placement`, and can be set to default values, e.g. using the first available machines, and allowing all processor sockets, so that the first socket is always used.

3. Design Objectives and Decisions

VALPA was developed using an approach of Architectural Modeling of Software Systems, described in [Clements et al. 2002]. First, the main forces that drive the design were identified. Then, some key decisions were made to meet the requirements of the design. What follows is a list, in order of importance, of quality and functionality requirements for the VALPA cluster manager:
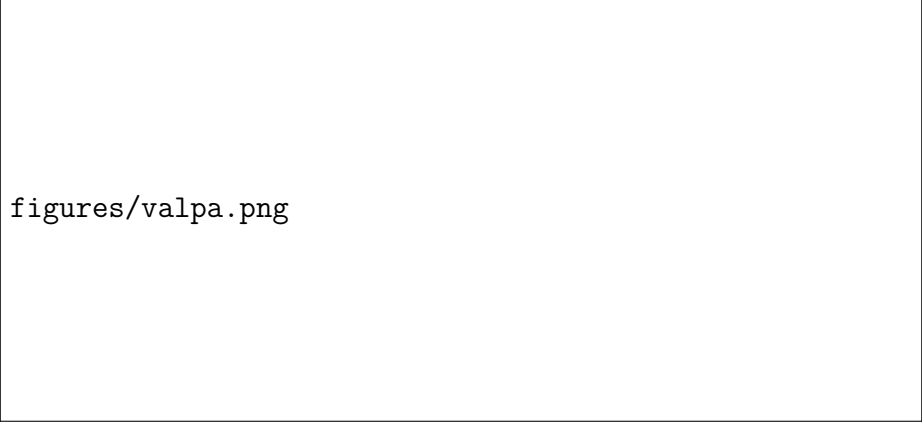
1. *Simplified Characterization:* VALPA is to be built upon the constraints presented in Section 2.2, leveraging the proposed $DIST$ variables to declare different virtual cluster topologies and configurations for each experiment. Moreover, VALPA should read the XML for experiment definition proposed in Section 7.4, with an understanding of the concept of *experiment* therein presented. The functionality should include basic support for `VMM Configuration` aspect introduced in Section 2.1, insofar as defining the network and disk technologies for the VM profiles.

2. *Automation:* given an XML with a list of experiments, the system must fully manage the virtual cluster and application life-cycles within each experiment. The virtual cluster life-cycle consists of the definition, instantiation and removal of the comprised VMs. The application life-cycle involves the deployment of a parallel application onto the instantiated virtual cluster, followed by the execution of the relevant processes —with the option of multiple sequential runs— and the extraction of generated outputs. VALPA should be able to undertake an experiment, await the execution, register the results, and conduct the next experiment, without human intervention.

3. *Provenance support:* this functionality requirement is about the ability of VALPA to store generated outputs for later examination. Any relevant performance metrics and virtual cluster features used in a particular experiment should be obtained by referring to said experiment. An experiment should be primarily identified by the application name and the values of the $DIST$ tuple (see item 1). However, any other additional setting, such as `VMM Configuration` details and application arguments, should be available, so that the generated outputs can be traced to a specific characterization of virtual cluster and execution. Finally, the performance metrics of several identical executions should be consolidated and marked with a common experiment identifier.

4. *Maintainability:* this quality requirement is related to the software development. There are several aspects of the VALPA system, such as virtual cluster manage-

ment, application registration, output parsing and storage, and consolidation of performance metrics. The corresponding components have different development and deployment cycles. Therefore, the impact of potential software changes should be minimized.

Upon the aforementioned requirements, the following architectural decisions were chosen:

1. *Leverage libvirt:* Since VALPA is not to be tied with a particular VMM, the *libvirt*[1] virtualization library was chosen in order to manage the life-cycle of VMs remotely. *Libvirt* uses an XML to fully describe a VM profile that will be used to create a VM instance. For a virtual cluster, VALPA then creates the required VM XMLs dynamically, using a VM XML template, and interacts with libvirt to manage the life-cycle of each VM. With this strategy, switching to another VMM involves just changing a single value in the VM XML template. *Libvirt* provides extensive support for several of the VMM configuration settings involved in VM deployment and tuning, fully covering the options presented in Section 2.1.

2. *Python programming language: Python* 3.1 was the language chosen to develop large portions of the VALPA modules, mainly due to its straightforward handling of the text files involved the management of the experiment XMLs and the VM XMLs. Another reasons that support this choice are that Python allows the creation of modular code using Python modules, as well as the management of unit and integration tests using PyUnit. This last item is important due to the complexity and inter-dependency of the operations involved in the virtual cluster life-cycle. The actual calls to *libvirt* are resolved by executing shell scripts by means of the Python's `os` module.

3. *Single virtual cluster:* In the current stage of development, VALPA supports the instantiation of only a single virtual cluster at any given time on the controlled environment. This is enough to test the hypothesis put forward in this work, but support for concurrent virtual clusters is scheduled for future development. As result of this simplification, the deployment of virtual clusters can be done by reusing the same 'virtual head' node, which can then be instantiated permanently.

4. *System and application configuration:* To maintain a high level of flexibility, the behavior of VALPA can be customized by several configuration files. The main preferences file defines default virtual cluster creation, such as the type of virtual network (virtual bridge) and the default technology that supports it (e.g. virtIO). VM instantiation is also configured using a VM XML template. Listing 43 shows this template, and Listing 46 shows the an example of a subsequent VM XML generated by VALPA. The handling of each application can be customized, e.g. passing parameters and parsing each output. To achieve this, the application should be previously registered with VALPA, by means of specific shell scripts and entries in configuration files.

---

[1] libvirt: The virtualization API. URL: http://libvirt.org

Figure 3. Deployment and execution view for VALPA architecture.

4. Architectural Overview of VALPA

In addition to *libvirt*, VALPA leverages the *PBS*-based, *Torque*[2] cluster manager to submit executions to the virtual cluster. Torque supports the management of cluster nodes, and is able to customize the deployment of MPI applications in order to satisfy the `Cluster Placement` specification.

Before running any experiment, a *PBS* template must be available for job submission, based on customizable parameters. Listing 44 shows the current *PBS* template. VALPA requires up to *ppm* VM image clones in each PM prior to instantiation, given that at most *ppm* VM instances will be hosted per PM. Since VALPA does not support dynamic application deployment, the VM image should contain a functioning executable for each application.

Figure 3 depicts a deployment diagram that shows the sequence to execute a series of experiments using VALPA. The sequence is explained as follows:

1. An XML document that lists each experiment is parsed. Besides stating the application and the number of executions, an experiment describes the characteristics of a virtual cluster as a composition of *Core Placement* and *VMM Configuration*. The experiment input is used to generate the required VM profiles, as well as the *PBS* file used to submit the application on the virtual cluster.

2. Since the `cpv` value may change between experiments, the cluster topology needs to be redefined at the virtual head. Thus, the description cluster is updated in *Torque* and the server daemon is restarted.

3. The generated VM profiles are used to instantiate the required working nodes in the virtual cluster. VALPA then waits for the VMs to be ready using the `qnodes` command.

4. The *PBS* job is submitted to the virtual head node. Since VALPA uses the virtual

---

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

cluster exclusively and topology matches request by design, execution can begin immediately.

5. The application runs on the virtual nodes. During each execution, resource utilization of VMs and PMs can be monitored using the `sysstat` tool.

6. Each time the application completes an execution, performance data is generated by collecting *sysstat* output and application-specific output.

7. Performance data is stored as files in a hierarchy of directories that reflects which *Core Placement* and *VMM Configuration* was used.

8. VALPA queries Torque for job termination using the `qstat` command. When all executions finish, the working nodes of the virtual cluster are removed. The system then returns to the initial state, and a new *experiment* can begin.

5. VALPA Setup

This section describes the procedure needed for setting up VALPA to a usable state. It consists of steps needed to create the virtual network, deploy the VMs and configure an execution environment for the applications.

5.1. Physical Environment

The physical cluster comprises a head node of the cluster (referred to as *head*) and one or more computing nodes for the instantiation of VMs (referred to as *nodes*).

The preferred network topology is presented in Figure 5.1. The B-class subnet (172.16.X.X) is preferred because it allows for more than 255 machines (physical and virtual) in the same network. Using the B-class, the IP addresses are organized so that the 172.16.1.X addresses (physical machine and the VMs hosted in it) correspond to the first node, the 172.16.2.X addresses to the second node, and so on. Each node should have the 172..16.X.254 address as shown.

The names of the nodes **must** be similar, with the same name prefix (`valpa` in the figure), followed by a numerical suffix that should always have the same length (e.g. `valpa01, valpa02, ... valpa99`). The head may also host VMs and thus be considered a node. In that case, it should follow the same guidelines for network configuration and naming.
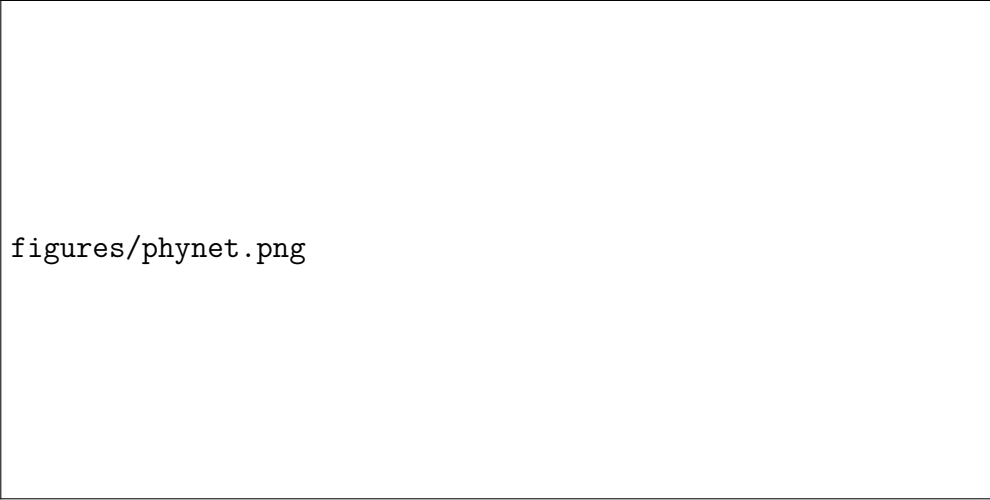
The VALPA software should be placed in the *head* filesystem (ex. `/home/valpauser/valpa`). This directory will be known as `VALPA_HOME`. The scripts will automatically execute using relative paths. The recommended Operating System (OS) is Ubuntu 14.04 64 bits. The dependencies are met as follows:

Listing 3. VALPA package dependencies (valpauser@head)

```
sudo apt-get install libvirt-bin qemu-kvm bridge-utils torque-client \
python3 python3-setuptools virt-viewer virtinst openssh-server
```

Listing 4. VALPA package dependencies (valpauser@nodes)

```
sudo apt-get install libvirt-bin qemu-kvm bridge-utils python3 openssh-server
```

Figure 4. Topology of the physical cluster (head and nodes).

In order to test the libvirt installation, connect using `virsh` command:

Listing 5. Libvirt test (valpauser@head)

```
$ virsh list
 Id Name                 State
----------------------------------
```

VALPA relies on the `quik` python package, which is found at `https://pypi.python.org/packages/source/q/quik/quik-0.2.2.tar.gz` After downloading, install with

Listing 6. Installing quik (valpauser@head)

```
sudo apt-get python3 setup.py install
```

For VALPA to function, password-less SSH is needed between the head and the nodes. The easiest way to achieve this is to generate a SSH key and copying it to all nodes. To create the SSH key:

Listing 7. SSH key (valpauser@head)

```
ssh-keygen -t rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

The key (private and public components, `id_rsa id_rsa.pub`) should be **sent via SSH** (manually) to the other nodes. Once copied, the following commands will both install the same key in a node, as well as registering the key for password-less SSH:

Listing 8. SSH key (each node)

```
mkdir ~/.ssh
cp id_rsa id_rsa.pub ~/.ssh
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

The file `VALPA_HOME/input/hardware.params` contains configuration parameters related to the physical environment, and **should be edited accordingly for correct VALPA setup**:

Listing 9. input/hardware.params

```
[Hardware]
# number of cores per node
cores=12
# number of processors (sockets) per node
sockets=2
# memory per node (GB)
mem=24

[Nodes]
# number of physical computing nodes
nodes=12
# prefix for naming nodes (node name = prefix+number)
prefix=valpa
# number of positions for indexing nodes (e.g for 2, node01..node99)
zeros=2
# first number for indexing nodes
first=2
# True: deduce node names in the form prefix+number
inferids=True
```

This file is read automatically by VALPA when executing either a shell or python script. With this configuration, it is possible to use the `VALPA_HOME/mgmt-nodes/cluster-ssh.sh` script in order to send commands to all the nodes via SSH. The same folder includes similar scripts for managing the physical cluster.

Listing 10. Usage of VALPA_HOME/mgmt-nodes/cluster-ssh.sh (valpauser@head)

```
./cluster-ssh.sh <command to be executed on nodes via SSH>
```

Listing 11. Example of VALPA_HOME/mgmt-nodes/cluster-ssh.sh (valpauser@head)

```
$ ./cluster-ssh.sh hostname
ssh -t valpa00 'hostname'
ssh -t valpa01 'hostname'
ssh -t valpa02 'hostname'
ssh -t valpa03 'hostname'
valpa00
Connection to valpa00 closed.
valpa01
Connection to valpa01 closed.
valpa02
Connection to valpa02 closed.
valpa03
Connection to valpa03 closed.
```

5.2. Setting up the Virtual Network

The file `input/valpa.params` handles most of the configuration parameters for VALPA. Here is the relevant section for configuring the virtual network (only showing B-class

parameters):

Listing 12. input/valpa.params (Networking)

```
[Networking]
# either 'network' (managed) or 'bridge' (external)
net_type=bridge

#
# If using network.type=network
#

# located in data-input
net_xml=network.xml
#desired name of virtual network in libvirt
net_name=kvm-bridge
# 'nat' recommended
net_forward=nat
# interface that will host bridge
net_dev=eth0
# either B or C class
net_class=B

# DHCP for 'network', B class
# Nodes will get 172.16.X.254
# VMs will get 172.16.X.Y, X matching node
# Class B networking for nodes and VMs
dhcp_b_prefix=172.16
# First VM of each node will have this suffix
dhcp_b_start=1

# Other networking

# desired bridge name
net_bridge=br0
# either network name (using network) or bridge name (using bridge)
net_value=kvm-bridge
# MAC address prefix for VMs
net_mac_prefix=00:16:36:ff
```

The default `net_type=bridge` relies on external configuration of the virtual bridges. Choosing `net_type=network` will instruct VALPA to manage the virtual bridges using *libvirt*, see Appendix A. In any case, the configuration file **does not need to be modified** in order to get virtual clusters in the B-class IP address range.

For the purposes of this manual, the `net_type=bridge` option is preferred. What follows is then the manual configuration of the virtual bridges and the DHCP server.

5.2.1. Setting up the virtual bridges

In each physical node, edit the `/etc/networking/interfaces` file for the virtual bridge. The example that follows corresponds to a class B network for the node `valpa01` (172.16.1.254).

Listing 13. interfaces file (root@nodes)

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet manual

auto br0
iface br0 inet static
  address 172.16.1.254
  network 172.16.0.0
  netmask 255.255.0.0
  broadcast 172.16.0.0
  gateway 172.16.0.254
  bridge_ports eth0
  bridge_stp off
  bridge_fd 0
  bridge_maxwait 0
```

In order for the bridge to become active, the node must be restarted:

Listing 14. reboot (root@nodes)

```
reboot
```

5.2.2. Configuring the DHCP server

This section assumes that the DHCP server will listen to the eth0 interface of the head server. Install the Ubuntu package:

Listing 15. Install DHCP package (valpauser@head)

```
sudo apt-get install isc-dhcp-server
```

The script at `VALPA_HOME/setup/dhcpd-conf-builder.sh` can be used to generate a suitable `dhcpd.conf` file. After executing the script, review the output file at `VALPA_HOME/data-output/dhcpd.conf` for adjustments.

Listing 16. Create the dhcpd.conf file (valpauser@head)

```
VALPA_HOME/setup$ ./dhcpd-conf-builder.sh
```

Finish configuration of the DHCP server:

Listing 17. Update DHCP configuration (valpauser@head)

```
sudo cp $VALPA_HOME/data-output/dhcpd.conf /etc/dhcp3/
sudo service isc-dhcp-server restart
```

5.2.3. Updating /etc/hosts to resolve hostnames

At this point, the physical nodes will have their final IP addresses. The VALPA_HOME/setup/generate-host-lines.sh script will generate the entries for both the physical and virtual IP addresses at VALPA_HOME/data-output/hosts. These entries should be used to update the /etc/hosts file of the physical nodes.

Listing 18. Creating /etc/host entries (valpauser@head)

```
VALPA_HOME/setup$ ./generate-hosts-lines.sh
```

5.2.4. Replicating VALPA files to physical nodes

The physical nodes require VALPA's monitor module for monitoring the consumption of physical resources. With the VALPA_HOME/setup/send-valpa-nodes.sh, this module is sent to the physical nodes, along with other support scripts.

Listing 19. Sending VALPA modules to nodes (valpauser@head)

```
VALPA_HOME/setup$ ./send-valpa-nodes.sh
```

5.2.5. Using vhost module (optional)

By default, VALPA relies on the vhost_net module in an attempt to improve inter-VM communication. This module should be loaded in the physical nodes. The following command loads the module immediately:

Listing 20. Loading vhost module (root@nodes)

```
sudo modprobe vhost_net experimental_zcopytx=1
```

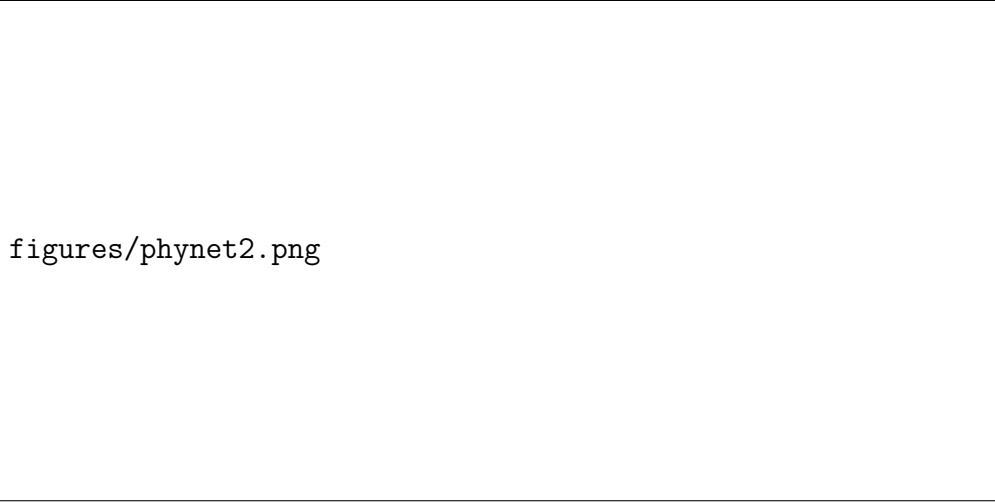To persist the change after reboot, edit the /etc/modprobe.d/vhost-net.conf file:

Listing 21. /etc/modprobe.d/vhost-net.conf

```
options vhost_net experimental_zcopytx=1
```

VALPA will automatically try to use vhost if available, else it will use the virtIO implementation from KVM. This fallback behavior is ensured by libvirt.

5.2.6. Final setting

Figure 5.2.6 shows the final setting for the physical network. The bridges will host the virtual networks. With this configuration, all nodes will be able to reach all VMs.

Figure 5. Topology of the physical cluster after creating the virtual bridges.

5.3. Creating a VM

VALPA requires the VM images used in the experiments to be created and deployed in advance. This requires having up to *ppn* VM images in each node, where *ppn* is the number of physical cores per node. The preferred way to deploy these images is to have a single master image in the head node, and create multiple copies of this image in the computing nodes.

To create a VM image, the `virt-install` command may be used to install a new OS (assuming Ubuntu 14.04). This requires access to the graphical capabilities of the head node, e.g. connecting using `ssh -X`. When installing the OS, the same username should be added (e.g. `valpauser`), and the `openssh-server` package should be added to gain SSH access to the node. The command is used as follows:

Listing 22. Creating the first VM image (valpauser@head)

```
UBUNTU_ISO=<path to ISO installer>
MASTER_IMAGE_PATH=<path to master VM image>
virt-install --connect qemu:///system -r 512 -n valpa-vm -c $UBUNTU_ISO --disk path=
    $MASTER_IMAGE_PATH/disk.img --vnc --keymap=en-us
```

The `MASTER_IMAGE_PATH` path should have a base path for all VM images and a specific directory for the master VM image. The paths should be declared in `valpa.params` as well for later use:

Listing 23. caption=params.sh (VM images)

```
VM_IMAGE_PATH=~/vms                 # Path for all VM images
VM_IMAGE_MASTER=ubuntu14.04-node    # Specific path for master image
DISK_FILENAME=disk.img
```

At this point, the **SSH keys** of the head should be copied to the VM manually for password-less SSH (valpauser and root) with a procedure similar to List-

ing 8. Also, the /etc/hosts file needs to be **updated** using the entries from the
VALPA_HOME/data-output/hosts at the head.

5.4. Setting up NFS

Most distributed applications require a shared filesystem in order to aggregate results.
In this manual, the NFS solution is used. The NFS server is installed in the head node,
and the VMs will run NFS clients. The nodes may also run NFS clients to facilitate
deployment.

Listing 24. Installing NFS server (valpauser@head)

```
sudo apt-get install nfs-kernel-server
```

A folder in the head node must be chosen to be exported to the clients, e.g.
/home/valpauser/shared. The server is configured editing the /etc/exports file. The
network must match the VALPA network (172.16.0.0/16 in this manual)

Listing 25. Relevant section of /etc/exports (head)

```
/home/valpauser/shared 172.16.0.0/16(rw,sync,no_root_squash,no_subtree_check)
```

After creating the shared directory, activate the export:

Listing 26. Activating NFS export (valpauser@head)

```
sudo exportfs -ra
```

The NFS clients should point to the server when mounting the shared folder.
A permanent solution is to use an entry in /etc/fstab. Assuming preferred network
configuration:

Listing 27. Relevant section of /etc/fstab (Master VM)

```
# NFS
172.16.100.254:/home/valpauser/shared /home/valpauser/shared nfs rsize=8192,wsize
    =8192,timeo=14,intr
```

The NFS client needs the appropriate package. Also, it may be necessary to start
the mount manually:

Listing 28. For mounting (Master VM)

```
sudo apt-get -y install nfs-common
sudo mount -a
```

In order for the mount to work, the folder /home/valpauser/shared must exist
at the client. To ensure read/write access, the UID and GID of the user at the VM must
match with the UID and GID of the user at the server. This can be enforced with the
following commands (see /etc/group at the server for the correct values):

Listing 29. Ensuring UID and GID (root@vm)

```
sudo apt-get install id-utils
sudo groupadd <groupname> --gid <GID>
sudo usermod --uid <UID> --gid <GID> valpauser
```

To test the permissions of NFS sharing, create a directory in `/home/valpauser/shared/` at the head node, and attempt to create a file within the directory using the VM console.

5.5. Setting up Torque

VALPA leverages the *PBS*-based, *Torque* cluster manager to submit executions to the virtual cluster. At the current stage of development, a single Torque server should be installed at the head node. This setting allows the instantiation of a single virtual cluster. VALPA will be later enhanced to support many concurrent virtual clusters with the same VM size, which can be supported with a single Torque server. Concurrent virtual clusters with different topologies require multiple Torque servers.

5.5.1. Torque server (valpauser@head)

To install the Torque server at the head node:

Listing 30. Installing Torque server (valpauser@head)

```
sudo apt-get install torque-server torque-scheduler torque-client
```

Declare server name, must match hostname (using valpa00 as head)

Listing 31. Torque server name (root@head)

```
echo valpa00 >> /var/spool/torque/server_name
```

Configure a default queue, and other configuration to ensure immediate execution

Listing 32. Torque server queue (root@head)

```
qmgr -c "create queue batch queue_type=execution"
qmgr -c "set queue batch started=true"
qmgr -c "set queue batch enabled=true"
qmgr -c "set queue batch resources_default.nodes=1"
qmgr -c "set queue batch resources_default.walltime=3600"
qmgr -c "set queue batch max_running = 8"
qmgr -c "set queue batch resources_max.ncpus = 144"
qmgr -c "set queue batch resources_min.ncpus = 1"
qmgr -c "set queue batch resources_max.nodes = 144"
qmgr -c "set queue batch resources_default.ncpus = 1"
qmgr -c "set queue batch resources_default.neednodes = 1:ppn=1"
qmgr -c "set queue batch resources_default.nodect = 1"
qmgr -c "set queue batch resources_default.nodes = 1"
qmgr -c "set queue batch resources_default.walltime=3600"
qmgr -c "set server default_queue=batch"
qmgr -c "set server scheduling=true"
```

The `resources_max.ncpus` and `resources_max.nodes` values must be **equal or greater than** the number of processor cores in the physical cluster. By default, this configuration is lost after rebooting the host of the Torque server. To avoid this, edit the `/etc/init.d/torque-server` file and remove the "-t create" parameter from the `DAEMON_SERVER_OPTS` variable, in the "start)" function:

Listing 33. Avoiding Torque re-configuration in `/etc/init.d/torque-server` (valpauser@head)

```
DAEMON_SERVER_OPTS="$DAEMON_SERVER_OPTS"
```

In case the service fails to start, issue the start commands again.

5.5.2. Torque node (Master VM)

Install the node package:

Listing 34. Installing Torque node (root@vm)

```
apt-get install torque-client torque-mom
```

Declare server name, must match hostname (using valpa00 as head)

Listing 35. Torque server name (root@vm)

```
echo valpa00 >> /var/spool/torque/server_name
```

Declare server name in `/var/spool/torque/mom_priv/config`. Also, setup output to match NFS shared directory:

Listing 36. Contents of `/var/spool/torque/mom_priv/config`

```
$pbs_server valpa00
$usecp *:/home/valpauser/shared /home/valpauser/shared
```

Torque will not function until the list of Torque nodes at `/var/spool/torque/server_priv/nodes` is updated, and the server's service is restarted. VALPA performs these steps when a virtual cluster is defined, but the Torque server may be tested beforehand using either the single, master VM instance or even the physical nodes as Torque nodes.

5.6. Compiling OpenMPI

Compiling OpenMPI from source is a good option if support for additional technologies is required, such as KNEM or Infiniband. The dependencies for compiling in the VM are met as follows:

Listing 37. Installing OpenMPI - Prerequisites (valpauser@vm)

```
sudo apt-get install build-essential gfortran python-dev \
libgsl0-dev cmake libfftw3-3 libfftw3-dev libgmp3-dev \
libmpfr4 libmpfr-dev libhdf5-serial-dev hdf5-tools python-h5py \
python-nose python-numpy python-setuptools python-docutils \
```

The source tarball can be downloaded at `http://www.open-mpi.org/software/`. After decompressing the TAR file use:

Listing 38. Installing OpenMPI - Configure (valpauser@vm)

```
./configure
```

Check the `config.log` for the required support, e.g. `tm.h` for Torque, `verbs.h` for infiniband, `knem.h` for KNEM. After that, use `make` to install:

Listing 39. Installing OpenMPI - Build (valpauser@vm)

```
make -j 6
sudo make install
sudo ldconfig
```

The installation can be tested with the following example:

Listing 40. Code example to test OpenMPI (example.c)

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv) {
  // Initialize the MPI environment
  MPI_Init(NULL, NULL);

  // Get the number of processes
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  // Get the rank of the process
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  // Get the name of the processor
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int name_len;
  MPI_Get_processor_name(processor_name, &name_len);

  // Print off a hello world message
  printf("Hello world from processor %s, rank %d"
      " out of %d processors\n",
      processor_name, world_rank, world_size);

  // End MPI environment
  MPI_Finalize();
}
```

Compile the code as `example` executable and call MPI in the same node:

Listing 41. Example to test OpenMPI

```
mpicc -o example example.c
mpirun -np 2 example
```

5.7. Copying the VM images

The master VM image must be replicated to the other physical machines. The following script sends the image over the network: TODO: fix VALPA for VM image paths!

Listing 42. Send the VM image over the network (valpauser@head)

```
VALPA_HOME/setup$ ./upload-disk.sh ubuntu14.04-node
```

6. Configuring VALPA

6.1. VALPA Configuration files

Listing 43. Template for the VM XML.

```
<?xml version="1.0"?>
<domain type='kvm'>
  <name>_NAME</name>
  <uuid>_UUIDGEN</uuid>
  <memory>_MEMORY</memory>
  <memoryBacking>
    <nosharepages/>
  </memoryBacking>
  <vcpu>_CORE</vcpu>
  <os>
    <type arch="x86_64" machine="pc">hvm</type>
    <boot dev="hd"/>
  </os>
  <features>
    <acpi/>
  </features>
  <clock offset="utc"/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <interface type="_NETWORK">
      <source _NETWORK="_NET_VALUE"/>
      <mac address="_MAC"/>
      <model type="virtio"/>
      <driver name="_NET_DRIVER"/>
    </interface>
    <emulator>/usr/bin/kvm</emulator>
    <disk type="file" device="disk">
      <driver name="qemu" type="_DISK_DRIVER_TYPE"/>
      <source file="_VALPAPATH/_CLUSTER_PATH/_VM_PATH/_DISKFILE"/>
      <target dev="_DISK_DEV" bus="_DISK_BUS"/>
    </disk>
    <console type="pty">
      <target port="0"/>
    </console>
    <input type="mouse" bus="ps2"/>
    <graphics type="vnc" port="-1" autoport="yes" listen="127.0.0.1" keymap="en-us"/>
    <video>
      <model type="cirrus" vram="9216" heads="1"/>
    </video>
  </devices>
</domain>
```

Listing 44. *PBS* template used by VALPA for job submissions.

```
#!/bin/bash
#
# Giacomo Mc Evoy - giacomo@lncc.br
# LNCC Brazil 2013
# Adapted from job.PARPARBench
# christian.simmendinger@t-systems.com
#
#PBS -l walltime=&WALLTIME
#PBS -l nodes=&VMS:ppn=&CPV
#PBS -N &APP_EXECUTABLE
#PBS -t 1-&EXEC_TIMES
#PBS -o &EXPERIMENT_PATH/pbs.out
#PBS -e &EXPERIMENT_PATH/pbs.err
```

```
# Var PBS_NODEFILE contains the node names (repeated)

# Execution parameters
rnum=$PBS_ARRAYID # Iteration of experiment (PBS)

# Deployment parameters
np=&NC

# Application parameters
name=&APP_EXEC_NAME

echo "Running PBS application: &APP_EXECUTABLE"
echo "Execution number: $rnum"

# Sort (don't copy) file with node names, to ensure uniqueness
EXEC_PROCS=/tmp/pbs-$name-procs.txt
sort $PBS_NODEFILE > $EXEC_PROCS

# Output dir for this execution set
rnum=$(printf "%03d\n" $rnum)
EXEC_OUTPUT_DIR=&EXPERIMENT_PATH/${name}-${rnum}
STDOUT=std.out
STDERR=std.err
CUSTOM=&CUSTOM

# Prepare execution
export PATH=$PATH:/usr/local/bin
mkdir -p $EXEC_OUTPUT_DIR
cd &APP_HOME

cp $PBS_NODEFILE /tmp/pbs-nodefile.txt

# Also monitor nodes?
if [ '&MONITOR_DO_NODES' == 'True' ]; then
  echo -e '&NODE_LIST' >> /tmp/pbs-nodefile.txt
fi

# Start monitoring
&MONITOR_START /tmp/pbs-nodefile.txt

# The actual invocation of the application
/usr/bin/time -f '&TIME_FORMAT' -a -o $EXEC_OUTPUT_DIR/../&TIME_OUTPUT mpirun --mca btl_tcp_if_include
    172.16.0.0/16 &BIND_TO_CORE &APP_EXECUTABLE &APP_ARGS > $EXEC_OUTPUT_DIR/$STDOUT 2> $EXEC_OUTPUT_DIR/
    $STDERR

# Stop monitoring (all VMs called at once, async)
&MONITOR_STOP /tmp/pbs-nodefile.txt $EXEC_OUTPUT_DIR

# Wait for monitoring to stop nicely
sleep 15

# Post-processing of output may be necessary
if [ '&APP_NEEDS_OUTPUT_COPY' == 'Y' ]; then
  cp &APP_OTHER_OUTPUT $EXEC_OUTPUT_DIR/$CUSTOM
fi

# Work the SAR files now, to save space
&MONITOR_SAR2TXT $EXEC_OUTPUT_DIR /tmp/pbs-nodefile.txt
```

7. Using VALPA

7.1. Adding a new Application

7.2. Running an Experiment with VALPA

7.3. Defining Experiments

7.4. XML for the Simplified Representation

The XML text presented in Listing 45 characterizes a hypothetical request for a virtual cluster using the simplified characterization proposed in this section.

Listing 45. XML for a virtual cluster deployment

```xml
<?xml version="1.0"?>
<experiments>
  <experiment name="exp1" trials="1">
    <clusters>
      <cluster>
        <placement>
          <topology nc="8" cpv="2" />
          <mapping idf="4" pstrat="BAL_ONE" />
        <placement/>
        <configuration>
          <technology network="vhost" disk="scsi" />
          <tuning>
            <hugepages>True</hugepages>
            <cpuTopology type="DEFAULT" guestNuma="False" />
            <numatune>False</numatune>
            <ballooning>False</ballooning>
          </tuning>
        </configuration>
        <app name="npb-ep" runs="10">
          <args>8 D</args>
          <tuning>
            <procpin>MPI_BIND</procpin>
          </tuning>
        </app>
      </cluster>
    </clusters>
  </experiment>
</experiments>
```

This XML file allows the definition of several *experiments*, which are executed sequentially. Within an experiment, several concurrent virtual clusters may be defined, each of them hosting a single application. The *cluster* element describes a virtual cluster according to the aspects shown in Figure 1. Its first sub-element represents the `Cluster Placement`, using the proposed simplified characterization of `Cluster Topology` and `Physical Mappings` (Section 2.2). The second sub-element represents the `VMM Configuration` aspect, where various virtualization technologies are specified. Finally, the third sub-element states the application (in this case, supported by MPI), specifying application arguments, MPI arguments and number of sequential executions in the deployed virtual cluster.

Listing 46. Example of VM XML ready for instantiation.

```xml
<?xml version="1.0"?>
<domain type="kvm">
  <name>kvm-pbs082-01</name>
```

```
<uuid>446bf85f-b4ba-459b-8e04-60394fc00d5c</uuid>
<memory>4194304</memory>
<memoryBacking>
  <nosharepages/>
</memoryBacking>
<vcpu>4</vcpu>
<cputune>
<vcpupin vcpu="0" cpuset="0,1,2,3"/>
<vcpupin vcpu="1" cpuset="0,1,2,3"/>
<vcpupin vcpu="2" cpuset="0,1,2,3"/>
<vcpupin vcpu="3" cpuset="0,1,2,3"/>
</cputune>
  <os>
    <type arch="x86_64" machine="pc">hvm</type>
    <boot dev="hd"/>
  </os>
  <features>
    <acpi/>
  </features>
  <clock offset="utc"/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <interface type="network">
      <source network="kvm-bridge"/>
      <mac address="00:16:36:ff:82:01"/>
      <model type="virtio"/>
      <driver name="vhost"/>
    </interface>
    <emulator>/usr/bin/kvm</emulator>
    <disk type="file" device="disk">
      <driver name="qemu" type="raw"/>
      <source file="/home/giacomo/Development/Systems/valpa/data-output/vms/node082/kvm-pbs082-01/disk.img"
          />
      <target dev="vda" bus="scsi"/>
    </disk>
    <console type="pty">
      <target port="0"/>
    </console>
    <input type="mouse" bus="ps2"/>
    <graphics type="vnc" port="-1" autoport="yes" listen="127.0.0.1" keymap="en-us"/>
    <video>
      <model type="cirrus" vram="9216" heads="1"/>
    </video>
  </devices>
</domain>
```

Notes

[1]Libvirt domain XML format. URL: http://libvirt.org/formatdomain.html

References

Abeni, L., Kiraly, C., Li, N., and Bianco, A. (2013). Tuning kvm to enhance virtual routing performance. pages 3803–3808.

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1994). The NAS parallel benchmarks. Technical Report 3, The International Journal of Supercomputer Applications.

Cherkasova, L., Gupta, D., and Vahdat, A. (2007). Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51.

Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., and Little, R. (2002). *Documenting Software Architectures: Views and Beyond.* Pearson Education.

Hat, M. W. R. (2011). KVM Performance Improvements and Optimizations. `http://www.linux-kvm.org/wiki/images/5/59/Kvm-forum-2011-performance-improvements-optimizations-D.pdf`.

Mc Evoy, G. and Schulze, B. (2011). Understanding scheduling implications for scientific applications in clouds. In *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '11, pages 3:1–3:6, New York, NY, USA. ACM.

8. Appendix A: Networking

8.1. Using the managed networking mode

Choosing `net_type=network` in the `input/valpa.params` configuration file will instruct VALPA to manage the virtual bridges using *libvirt*. Using the managed network will also use an internal DHCP server to set the IP addresses of the VMs based on their MAC address.

However, this alternative option will fail (`libvirt` limitation) if the desired virtual network already exists (for instance, if the eth0 interface has this network defined). This means that, initially, the physical cluster **must** have a different network, e.g. 192.168.1.0/8 at eth0. Using `libvirt`, VALPA will then create the virtual network 172.16.0.0/16. The `/etc/hosts` file should be managed accordingly for these changes.

8.1.1. Creating the network bridges

Use the `VALPA_HOME/setup/define-virtual-bridges.sh` script to create the virtual bridges. It will generate the XML network configurations for `libvirt`, as well as register them in the corresponding nodes.

Listing 47. Setting up the virtual bridges (valpauser@head)
```
VALPA_HOME/setup$ ./define-virtual-bridges.sh
```

In order to have a functioning virtual cluster, the bridge needs to be associated with the network interface. This must be done manually at each node, since the `cluster-ssh.sh` script may break due to the reconfiguration of the networking. Assuming default values, run the following command on each node:

Listing 48. Setting up the virtual bridges (each node)
```
sudo brctl addif br0 eth0
```

The command may hang after execution, close the console and verify connectivity afterwards. With the virtual bridge, each node will receive the 172.16.X.254 address. If using the preferred network configuration, the IP addresses will remain the same (else, updating `/etc/hosts` files is needed).