Virtualized Experiments for Scientific Parallel Applications (VESPA) - Installation Manual

Giacomo Victor Mc Evoy Valenzano[1]

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

giacomo@lncc.br

1. Introduction

This document describes the installation process for the software system called Vespa (Virtualized Experiments for Scientific Parallel Applications). Vespa is designed to support the definition and production of controlled application executions running on virtual machines (VMs), as well as gathering performance metrics related to these executions.

The main goal of Vespa is to manage the systematic experimentation of applications deployed on different virtual clusters, while supporting rich definitions for the cluster topology and mappings to underlying physical resources. The results will later translate into a knowledge repository with real (non-simulated) data for studying the effects of virtual cluster features on different scientific applications.

Vespa is not meant to be a tool for deploying applications in the Cloud or in virtualized environments. It is specifically aimed at improving the understanding of how virtualization affects application performance. While Vespa performs the deployment of the virtual cluster and the execution of the application, it does **not** currently configure the VMs (assumes the VM images are ready), **nor** does it takes care of the application deployment (assumes the application has an executable properly installed and ready to be called).

Reading the manual:

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. About Vespa

2.1. Definitions

Virtual Cluster: Aggregation of Virtual Machines inter-connected with a virtual network, supporting the execution of a parallel application. It is described by a *Virtual Cluster Characterization*, see Figure 1.

Core Placement: Representation of the spatio-temporal allocation of a virtual core, in terms of its mapping onto physical cores.

Virtual Machine Placement: Traditionally, it is defined as the selection of the hosting physical node for each given VM. Here, it is defined as an aggregation of *Core Placement* instances for all the virtual cores of a VM.

Cluster Placement: Defined as aggregation of the *Virtual Machine Placement* instances for all the VMs of a *Virtual Cluster*. Unambiguously describes how each virtual core in the cluster is mapped onto physical cores. Can be separated in *Cluster Topology* and *Physical Mappings*.

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.

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.

Application: A scientific benchmark, application or workflow element with execution in finite time.

Execution: A single instance of an *Application*, has start time and finish time.

Experiment: Instantiation of a *Virtual Cluster* followed by one or more sequential application *Executions* on said *Virtual Cluster*, includes the gathering of *Performance Metrics*.

Scenario: Orchestration of one or more concurrent *Experiments*, includes the life-cycle management of the *Virtual Clusters*. Currently, Vespa supports the definition of concurrent *Experiments*, but not the actual execution.

Performance Metrics: Relevant generated data about *Experiments*. Classified as scalar metrics (single value for each *Execution*) and temporal metrics (time series for each *Execution*). Also classified as system metrics (CPU, memory, network utilization) and application metrics (runtime, application-specific throughput).

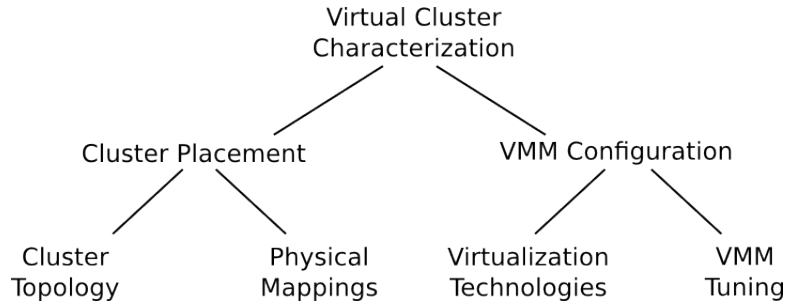

Figure 1. Hierarchical view of proposed characteristics of a virtual cluster.

2.2. Assumptions and Limitations

1. Applications consist of a variable set of processes, which are deployed in one or more VMs on execution. Each VM hosts processes corresponding to a single application.
2. Processes of a given application are either equal (SPMD approach) or reflect the Master/Worker paradigm. A process may not migrated to other VMs once it starts execution.

3. Vespa currently assumes a homogeneous physical environment, but this assumption may be relaxed in the future.
4. Vespa currently supports virtual clusters composed of homogeneous VMs, but this limitation may be relaxed in the future.
5. Vespa currently uses a simplified characterization of the *Virtual Cluster Placement* based on four primary variables and two secondary 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 item 4.
    - **Distribution 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

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

      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 $df = 0$.
    - **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, but it is possible to specify a subset of physical cores that are eligible. Five pinning strategies are currently available, called `NONE`, `GREEDY`, `BAL-ONE`, `BAL-SET` and `SPLIT`.
6. **Two secondary variables:** `deployedNodes`, which specifies the identities of the PMs that are hosting VMs for the virtual cluster; `deployedSockets`, which specifies the identities of the eligible processor sockets, which in turn affect the identities of resulting physical cores $\langle p \rangle_i$ for each virtual core v_i. On non-concurrent experiments, these variables can assume default values (first physical nodes and all physical sockets).
7. The current representation of generated data supports only scenarios with single *Executions* (without concurrency).

2.3. Features of Vespa

- **Defining Experiments:** The Scenarios that are supported by Vespa are described in an Experiment Definition File, it is the main input for running experiments. Vespa provides a module for creating the file according to different parameters. The module's arguments are used to specify ranges for sweeping parameters such as virtual cluster size, distribution, pinning strategy, etc.
- **Running Experiments:** The main function of Vespa is to execute parallel applications on virtual clusters. These executions are described in the Experiment

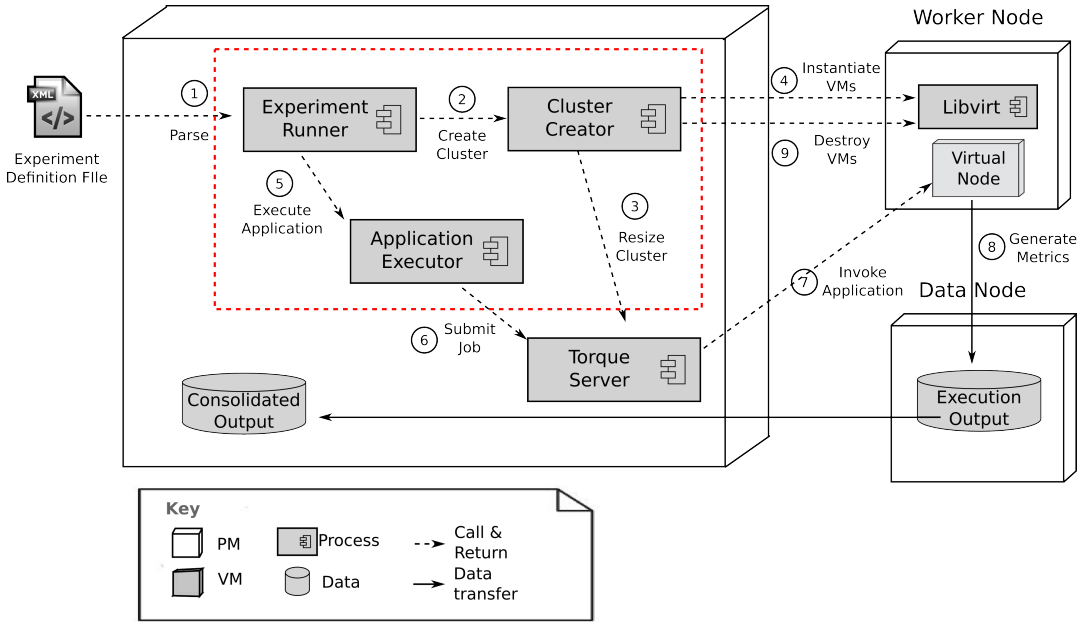

Figure 2. Deployment and execution view for Vespa architecture.

Definition File, and are orchestrated in sequence and independently. For each scenario, the necessary *Virtual Clusters* are created, the *Applications* are executed a number of times (runs), the performance metrics are gathered, and the Virtual Clusters are destroyed. This sequential process is automated as to not require human intervention.

Vespa leverages the *libvirt* virtualization library to manage the life-cycle of VMs remotely, and the *PBS*-based, *Torque* cluster manager to submit executions to the virtual cluster. It is written in both shell script and the Python programming language. Before running any experiment, a master XML must be created as a VM definition template for *libvirt*, as well as a master *PBS* file for job submission, based both on customizable parameters. Furthermore, up to *ppm* VM images are cloned in each PM, since at most *ppm* VM instances will be hosted per PM.

Figure 2 depicts a deployment diagram that shows the sequence to execute a Scenario using Vespa. The sequence is explained as follows:

1. The Experiment Definition File is parsed. Besides stating the application and the number of executions, an experiment describes the characteristics of a virtual cluster as a composition of *Cluster Placement* and *VMM Configuration*.
2. The experiment input is used to calculate and generate the VM profiles required for the virtual cluster.
3. Since the `cpv` value can change between experiments, the *Cluster Topology* needs to be redefined at the virtual head. Thus, the cluster description is updated in *Torque* and the server daemon is restarted.
4. The generated VM profiles are used to instantiate the required working

nodes in the virtual cluster. Vespa then waits for the VMs to be ready using `qnodes` command.

5. A component called Application Executor will generate the *PBS* file used to submit the application on the virtual cluster.
6. The *PBS* job is submitted to the *Torque* server. Since Vespa uses the virtual cluster exclusively and topology matches request by design, execution can begin immediately.
7. The parallel application runs on the virtual nodes. During each execution, resource utilization of VMs and PMs can be monitored using the `sysstat` tool.
8. Each time the application completes an execution, performance data is generated by collecting *sysstat* output and application-specific output.
9. Vespa queries Torque for job termination using the `qstat` command. When all executions finish, the working nodes of the virtual cluster are destroyed. The system returns to the initial state, and a new experiment can begin.

- **Gathering of Performance Metrics:** Vespa currently uses the `sysstat` software to monitor both the virtual and physical environment during each *Execution*. With `sysstat`, the utilization rate of CPU, memory and network is captured as time series that span the duration of the *Execution*. The number of data elements retrieved depends on the *Virtual Cluster Placement*, both its topology (number of VMs) and physical mappings (number of hosts). Vespa also registers the output of the `time` command, that informs the `userTime, systemTime` and `ellapsedTime`. Finally, an application-specific plugin may be configured for Vespa so that it captures an application metric such as throughput, e.g. a *Gflops* metric for each *Execution*.
- **Consolidation of Performance Metrics:** At any time, the user may be interested in producing a consolidated performance report for an application. A Vespa module is used to produce a folder and two CSV files that include all the aforementioned metrics.

3. Vespa Setup

This section describes the procedure needed for setting up Vespa 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.

3.1. Getting Vespa

The code for Vespa is currently available at: `https://github.com/ginomcevoy/vespa`

3.2. 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 3.2. 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. In this example, each node has the 172.16.X.254 address. Default Vespa configuration assumes that the first addresses of the 172.16.X.0 subnet will be used for the VMs.

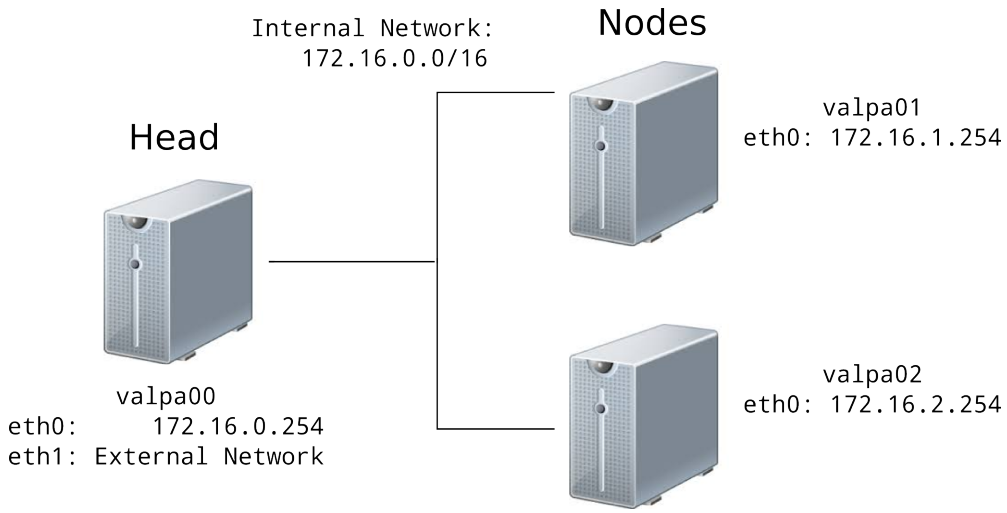

Figure 3. Topology of the physical cluster (head and nodes).

The Vespa software should be placed in the *head* filesystem (ex. `/home/vespauser/vespa`). This directory will be known as `VESPA_HOME`. The scripts will automatically execute using relative paths. The Operating System (OS) used to show the procedure is Ubuntu 14.04 64 bits. The dependencies are then met as follows:

Listing 3. Vespa package dependencies (vespauser@head)

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

Listing 4. Vespa package dependencies (vespauser@nodes)

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

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

Listing 5. Libvirt test (vespauser@head)

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

Vespa relies on the `Ansible` configuration management tool. While it is available through apt-get `quik` python package, Vespa requires version 0.9+:

Listing 6. Installing Ansible (vespauser@head)

```
sudo pip install ansible
```

For Vespa 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 (vespauser@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
```

Vespa expects the file `config/vespa.nodes` to be populated with the node names. In the following example, 12 physical nodes are available to Vespa:

Listing 9. Example for config/vespa.nodes

```
# This file is the initial inventory file in INI format.
# It is read by Vespa to find define the computing nodes.
# Write the hostnames of each computing node here, one per line.
valpa01
valpa02
valpa03
```

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

Listing 10. Example for config/hardware.params

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

This file is read automatically by Vespa when executing either a shell or python script. With this configuration, it is possible to use the `VESPA_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 11. Usage of VESPA_HOME/mgmt-nodes/cluster-ssh.sh (vespauser@head)

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

Listing 12. Example of VESPA_HOME/mgmt-nodes/cluster-ssh.sh (vespauser@head)

```
$ ./cluster-ssh.sh hostname
valpa01 | success | rc=0 >>
valpa01

valpa02 | success | rc=0 >>
valpa02

valpa03 | success | rc=0 >>
valpa03
```

3.3. Setting up the Virtual Network

The file `config/vespa.params` handles most of the configuration parameters for VESPA. Here is the relevant section for configuring the virtual network (only showing B-class parameters):

Listing 13. config/vespa.params (Networking)

```
##############################################################################
# Main networking option: one of {libvirt-bridge | external-bridge | sriov}
##############################################################################
# Vespa will use this value when creating the VM definitions.
# ALL options imply a virtual network created in libvirt for each physical node
# The setup/define-virtual-networks.sh script can be used to generate the XML
# definitions for these networks
#
# libvirt-bridge: instructs Vespa to use a bridge managed by libvirt
#        (DHCP by libvirt)
# external-bridge: instructs Vespa to use a previously defined bridge
#        (bridge and DHCP external to libvirt)
# sriov:        instructs Vespa to use the Virtual Functions of a
#        SR-IOV network
network_source=external-bridge
```

The default `network_source=external-bridge` relies on external configuration of the virtual bridges. Choosing `network_source=libvirt-bridge` will instruct Vespa 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 `network_source=external-bridge` option is preferred. What follows is then the manual configuration of the virtual bridges and the DHCP server.

3.3.1. 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 `vespa01` (172.16.1.254).

Listing 14. interfaces file (root@valpa01)

```
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 can be restarted:

Listing 15. reboot (root@nodes)

```
reboot
```

3.3.2. 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 16. Install DHCP package (vespauser@head)

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

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

Listing 17. Create the dhcpd.conf file (vespauser@head)

```
VESPA_HOME/networking$ ./dhcpd-conf-builder.sh
```

Finish configuration of the DHCP server:

Listing 18. Update DHCP configuration (vespauser@head)

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

3.3.3. Updating /etc/hosts to resolve hostnames

At this point, the physical nodes will have their final IP addresses. The following command will generate the entries for both the physical and virtual IP addresses at the specified output. These entries should be used to update the `/etc/hosts` file of the physical nodes.

Listing 19. Creating /etc/host entries (vespauser@head)

```
VESPA_HOME/vespa$ python -m network.etchosts <output>
```

3.3.4. Replicating Vespa files to physical nodes

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

Listing 20. Sending Vespa modules to nodes (vespauser@head)

```
VESPA_HOME/setup$ ./send-vespa-nodes.sh
```

3.3.5. Using vhost module (optional)

By default, Vespa 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 21. 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 22. /etc/modprobe.d/vhost-net.conf

```
options vhost_net experimental_zcopytx=1
```

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

3.3.6. Final setting

Figure 3.3.6 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.

3.4. Creating a VM

Vespa 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

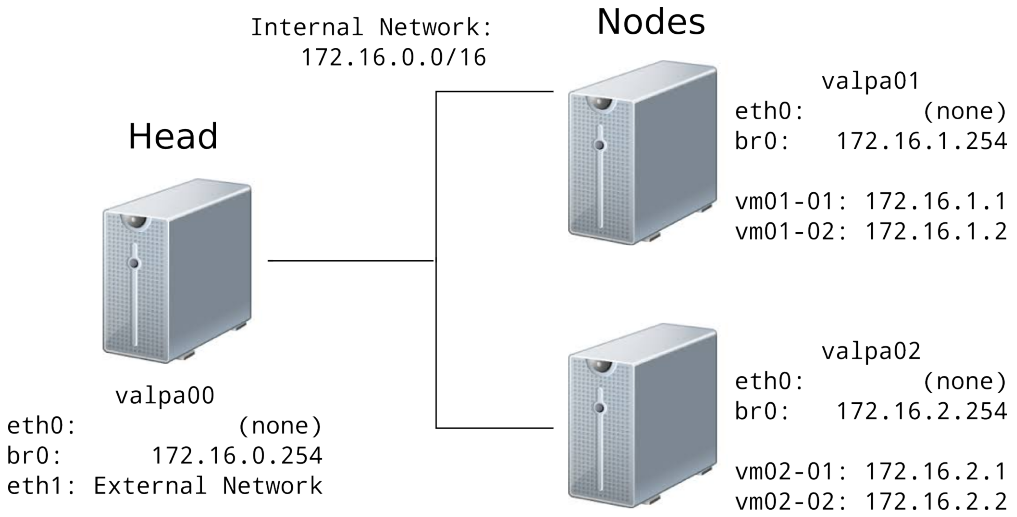

Figure 4. Topology of the physical cluster after creating the virtual bridges.

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. `vespauser`), and the `openssh-server` package should be added to gain SSH access to the node. The command is used as follows:

Listing 23. Creating the first VM image (vespauser@head)

```
UBUNTU_ISO=<path to ISO installer>
MASTER_IMAGE_PATH=<path to master VM image>
virt-install --connect qemu:///system -r 512 -n vespa-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 `vespa.params` as well for later use:

Listing 24. 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 (vespauser and root) with a procedure similar to Listing 8. Also, the `/etc/hosts` file needs to be **updated** using the entries from the `VESPA_HOME/data-output/hosts` at the head.

(Remove /etc/udev/rules.d/70-persistent-net.rules) (Set

GRUB_RECORDFAIL_TIMEOUT=0 in /etc/default/grub)

3.5. 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 25. Installing NFS server (vespauser@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/vespauser/shared`. The server is configured editing the `/etc/exports` file. The network must match the Vespa network (172.16.0.0/16 in this manual)

Listing 26. Relevant section of /etc/exports (head)

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

After creating the shared directory, activate the export:

Listing 27. Activating NFS export (vespauser@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 28. Relevant section of /etc/fstab (Master VM)

```
# NFS
172.16.100.254:/home/vespauser/shared /home/vespauser/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 29. For mounting (Master VM)

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

In order for the mount to work, the folder `/home/vespauser/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 30. Ensuring UID and GID (root@vm)

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

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

3.6. Setting up Torque

Vespa 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. Vespa 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.

3.6.1. Torque server (vespauser@head)

To install the Torque server at the head node:

Listing 31. Installing Torque server (vespauser@head)

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

Declare server name, must match hostname (using vespa00 as head)

Listing 32. Torque server name (root@head)

```
echo vespa00 >> /var/spool/torque/server_name
```

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

Listing 33. 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 34. Avoiding Torque re-configuration in `/etc/init.d/torque-server` (vespauser@head)

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

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

3.6.2. Torque node (Master VM)

Install the node package:

Listing 35. Installing Torque node (root@vm)

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

Declare server name, must match hostname (using vespa00 as head)

Listing 36. Torque server name (root@vm)

```
echo vespa00 >> /var/spool/torque/server_name
```

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

Listing 37. Contents of `/var/spool/torque/mom_priv/config`

```
$pbs_server vespa00
$usecp *:/home/vespauser/shared /home/vespauser/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. Vespa 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.

3.7. 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 38. Installing OpenMPI - Prerequisites (vespauser@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 39. Installing OpenMPI - Configure (vespauser@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 40. Installing OpenMPI - Build (vespauser@vm)

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

The installation can be tested with the following example:

Listing 41. Code example to test OpenMPI (example.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 42. Example to test OpenMPI

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

3.8. 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 Vespa for VM image paths!

Listing 43. Send the VM image over the network (vespauser@head)

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

4. Appendix A: Networking

4.1. Using the managed networking mode

Choosing `network_source=libvirt-bridge` in the `config/vespa.params` configuration file will instruct Vespa 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`, Vespa will then create the virtual network 172.16.0.0/16. The `/etc/hosts` file should be managed accordingly for these changes.

4.1.1. Creating the network bridges

Use the `VESPA_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 44. Setting up the virtual bridges (vespauser@head)

```
VESPA_HOME/setup$ ./define-virtual-networks.sh libvirt-bridge
```

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 45. 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).