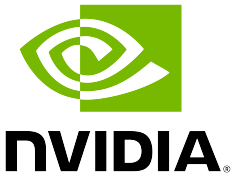

NVIDIA Base Command Manager 11

User Manual

Revision: d5b5535f1

Date: Wed Oct 29 2025

Table of Contents

Preface

Welcome to the *User Manual* for NVIDIA Base Command Manager 11.

0.1 About This Manual

This manual is intended for the end users of a cluster running NVIDIA Base Command Manager (BCM, also known as "the cluster manager"). This manual tends to see things from a user perspective, and covers the basics of using BCM's user environment to run compute jobs on the cluster. Although it does cover some aspects of general Linux usage, it is by no means comprehensive in this area. Readers are expected to have some familiarity with the basics of a Linux environment from the regular user point of view.

Regularly updated production versions of the NVIDIA Base Command Manager 11 manuals are available on updated clusters by default at `/cm/shared/docs/cm`. The latest updates are always online at `https://docs.nvidia.com/base-command-manager`.

The manuals constantly evolve to keep up with the development of the BCM environment and the addition of new hardware and/or applications. The manuals also regularly incorporate feedback from administrators and users, who can submit comments, suggestions or corrections via the website

`https://enterprise-support.nvidia.com/s/create-case`

Section 14.2 of the *Administrator Manual* has more details on submitting an issue.

0.2 Getting User-Level Support

A user is first expected to refer to this manual or other supplementary site documentation when dealing with an issue. If that is not enough to resolve the issue, then support for an end-user is typically provided by the cluster administrator, who is often a unix or Linux system administrator with some cluster experience. Commonly, the administrator has configured and tested the cluster beforehand, and therefore has a good idea of its behavior and quirks. The initial step when calling in outside help is thus often to call in the cluster administrator.

1

Introduction

This manual is intended for cluster users who need a quick introduction to the NVIDIA Base Command Manager, which manages a Beowulf cluster configuration. It explains how to use the MPI and batch environments, how to submit jobs to the queuing system, and how to check job progress. The specific combination of hardware and software installed may differ depending on the specification of the cluster, which means that parts of this manual may not be relevant to the user's particular cluster.

1.1 What Is A Beowulf Cluster?

1.1.1 Background And History

In the history of the English language, Beowulf is the earliest surviving epic poem written in English. It is a story about a hero with the strength of many men who defeated a fearsome monster called Grendel.

In computing, a Beowulf class cluster computer is a multiprocessor architecture used for parallel computations, i.e., it uses many processors together so that it has the brute force to defeat certain "fearsome" number-crunching problems.

The architecture was first popularized in the Linux community when the source code used for the original Beowulf cluster built at NASA was made widely available. The Beowulf class cluster computer design usually consists of one head node and one or more regular nodes connected together via Ethernet or some other type of network. While the original Beowulf software and hardware has long been superseded, the name given to this basic design remains "Beowulf class cluster computer", or less formally "Beowulf cluster".

1.1.2 Brief Hardware And Software Description

On the hardware side, commodity hardware is generally used in Beowulf clusters to keep costs down. These components are usually x86-compatible processors produced at the Intel and AMD chip foundries, standard Ethernet adapters, InfiniBand interconnects, and switches. From around 2019 the ARMv8 processor architecture is also gaining attention.

On the software side, free and open-source software is generally used in Beowulf clusters to keep costs down. For example: the Linux operating system, the GNU C compiler collection and open-source implementations of the Message Passing Interface (MPI) standard.

The head node controls the whole cluster and serves files and information to the nodes. It is also the cluster's console and gateway to the outside world. Large Beowulf clusters might have more than one head node, and possibly other nodes dedicated to particular tasks, for example consoles or monitoring stations. In most cases compute nodes in a Beowulf system are dumb—in general, the dumber the better—with the focus on the processing capability of the node within the cluster, rather than other abilities a computer might generally have. A node may therefore have

- one or more processing elements. The processors may be standard CPUs, as well as GPUs, FPGAs, MICs, and so on.

- enough local memory—memory contained in a single node—to deal with the processes passed on to the node
- a connection to the rest of the cluster

Nodes are configured and controlled by the head node, and do only what they are told to do. One of the main differences between Beowulf and a Cluster of Workstations (COW) is the fact that Beowulf behaves more like a single machine rather than many workstations. In most cases, the nodes do not have keyboards or monitors, and are accessed only via remote login or possibly serial terminal. Beowulf nodes can be thought of as a CPU + memory package which can be plugged into the cluster, just like a CPU or memory module can be plugged into a motherboard to form a larger and more powerful machine. A significant difference is that the nodes of a cluster have a relatively slower interconnect.

1.2 Brief Network Description

A Beowulf Cluster consists of a login, compile and job submission node, called the head, and one or more compute nodes, often referred to as worker nodes. A second (fail-over) head node may be present in order to take control of the cluster in case the main head node fails. Furthermore, a second fast network may also have been installed for high-performance low-latency communication between the (head and the) nodes (see figure 1.1).

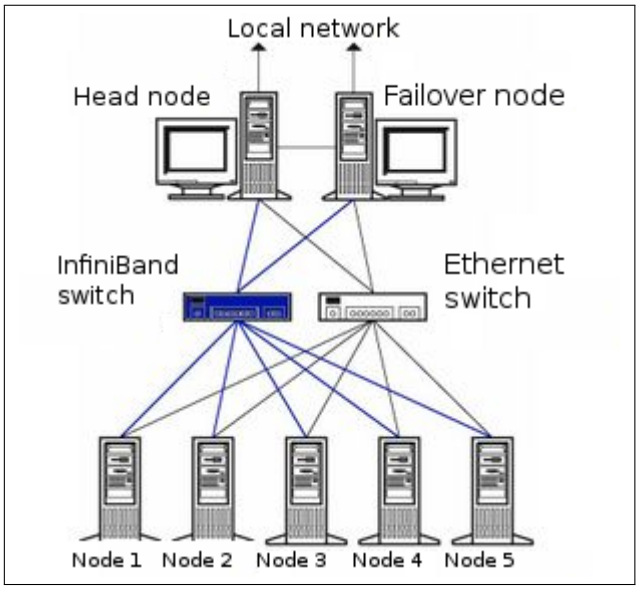

Figure 1.1: Cluster layout

The login node is used to compile software, to submit a parallel or batch program to a job queuing system and to gather/analyze results. Therefore, it should rarely be necessary for a user to log on to one of the nodes and in some cases node logins are disabled altogether. The head, login and compute nodes usually communicate with each other through a gigabit Ethernet network, capable of transmitting information at a maximum rate of 1000 Mbps. In some clusters 10 gigabit Ethernet (10GE, 10GBE, or 10GigE) is used, capable of up to 10 Gbps rates, while higher rates than that are also available.

Sometimes an additional network is used by the cluster for even faster communication between the compute nodes. This particular network is mainly used for programs dedicated to solving large scale computational problems, which may require multiple machines and could involve the exchange of vast amounts of information. One such network topology is InfiniBand, commonly capable of transmitting information at a maximum effective data rate of about 124Gbps and about 1.2μs end-to-end latency on small packets, for clusters in 2013. The commonly available maximum transmission rates will increase over the years as the technology advances.

Applications relying on message passing benefit greatly from lower latency. The fast network is usually complementary to a slower Ethernet-based network.

2

Cluster Usage

2.1 Login To The Cluster Environment

The login node is the node where the user logs in and works from. Simple clusters have a single login node, but large clusters sometimes have multiple login nodes to improve the availability of the cluster. In most clusters, the login node is also the head node from where the cluster is monitored and installed. On the login node:

- applications can be developed
- code can be compiled and debugged
- applications can be submitted to the cluster for execution
- running applications can be monitored

To carry out an ssh login to the cluster, a terminal session can be started from Unix-like operating systems:

Example

```
$ ssh myname@cluster.hostname
```

On a Windows operating system, an SSH client such as PuTTY (`http://www.putty.org`) can be downloaded. Another standard possibility is to run a Unix-like environment such as Cygwin (`http://www.cygwin.com`) within the Windows operating system, and then run the SSH client from within it.

A Mac OS X user can use the Terminal application from the Finder, or under `Application/Utilities/Terminal.app`. An X11 windowing environment must be installed for it to work. XQuartz is the recommended X11 windowing environment, and is indeed the only Apple-backed X11 version available from OS X 10.8 onward.

When using the SSH connection, the cluster's address must be added. When the connection is made, a username and password must be entered at the prompt.

If the administrator has changed the default SSH port from 22 to something else, the port can be specified with the `-p <port>` option:

```
$ ssh -X -p <port> <user>@<cluster>
```

The -X option can be dropped if no X11-forwarding is required. X11-forwarding allows a GUI application from the cluster to be displayed locally.

Optionally, after logging in, the password used can be changed using the `passwd` command:

```
$ passwd
```

2.2 Setting Up The User Environment

By default, each user uses the bash shell interpreter. In that case, each time a user login takes place, a file named `.bashrc` is executed to set up the shell environment for the user. The shell and its environment can be customized to suit user preferences. For example,

- the prompt can be changed to indicate the current username, host, and directory, for example: by setting the prompt string variable:

  ```
  PS1='[\u@\h \W]\$ '
  ```

- the size of the command history file can be increased, for example: `export HISTSIZE=100`
- aliases can be added for frequently used command sequences, for example: `alias lart='ls -alrt'`
- environment variables can be created or modified, for example: `export MYVAR=MYSTRING`
- the location of software packages and versions that are to be used by a user (the path to a package) can be set.

Because there is a huge choice of software packages and versions, it can be hard to set up the right environment variables and paths for software that is to be used. Collisions between different versions of the same package and non-matching dependencies on other packages must also be avoided. To make setting up the environment easier, BCM provides preconfigured environment modules (section 2.3).

2.3 Environment Modules

For a user to compile and run computational jobs on a cluster, a special shell environment is typically set up for the software that is used.

However, setting up the right environment for a particular software package and version can be tricky, and it can be hard to keep track of how it was set up.

For example, users want to have a clean way to bring up the right environment for compiling code according to the various MPI implementations, but can easily get confused about which libraries have been used, and can end up with multiple libraries with similar names installed in a disorganized manner.

A user might also like to conveniently test new versions of a software package before permanently installing the package.

Within a Linux distribution running without special utilities, setting up environments can be complex. However, BCM makes use of the environment modules package, which provides the `module` command. The `module` command is a special utility to make taking care of the shell environment much easier.

2.3.1 Available Commands

Practical use of the `modules` commands is given in sections 2.3.2 and 2.3.3.

For reference, the help text for the `module` command can be viewed as follows:

Example

```
[me@cluster ~]$ module --help
Modules Release 4.5.3 (2020-08-31)
Usage: module [options] [command] [args ...]

Loading / Unloading commands:
  add | load      modulefile [...]  Load modulefile(s)
  rm | unload     modulefile [...]  Remove modulefile(s)
```

```
  purge                              Unload all loaded modulefiles
  reload | refresh                   Unload then load all loaded modulefiles
  switch | swap   [mod1] mod2        Unload mod1 and load mod2

Listing / Searching commands:
  list            [-t|-l|-j]         List loaded modules
  avail   [-d|-L] [-t|-l|-j] [-S|-C] [--indepth|--no-indepth] [mod ...]
                                     List all or matching available modules
  aliases                            List all module aliases
  whatis  [-j]    [modulefile ...]   Print whatis information of modulefile(s)
  apropos | keyword | search [-j] str
                                     Search all name and whatis containing str
  is-loaded       [modulefile ...]   Test if any of the modulefile(s) are loaded
  is-avail        modulefile [...]   Is any of the modulefile(s) available
  info-loaded     modulefile         Get full name of matching loaded module(s)

Collection of modules handling commands:
  save            [collection|file] Save current module list to collection
  restore         [collection|file] Restore module list from collection or file
  saverm          [collection]      Remove saved collection
  saveshow        [collection|file] Display information about collection
  savelist        [-t|-l|-j]        List all saved collections
  is-saved        [collection ...]  Test if any of the collection(s) exists

Shell's initialization files handling commands:
  initlist                           List all modules loaded from init file
  initadd         modulefile [...]   Add modulefile to shell init file
  initrm          modulefile [...]   Remove modulefile from shell init file
  initprepend     modulefile [...]   Add to beginning of list in init file
  initswitch      mod1 mod2          Switch mod1 with mod2 from init file
  initclear                          Clear all modulefiles from init file

Environment direct handling commands:
  prepend-path [-d c] var val [...] Prepend value to environment variable
  append-path [-d c] var val [...]  Append value to environment variable
  remove-path [-d c] var val [...]  Remove value from environment variable

Other commands:
  help            [modulefile ...]  Print this or modulefile(s) help info
  display | show  modulefile [...]  Display information about modulefile(s)
  test            [modulefile ...]  Test modulefile(s)
  use     [-a|-p] dir [...]         Add dir(s) to MODULEPATH variable
  unuse           dir [...]         Remove dir(s) from MODULEPATH variable
  is-used         [dir ...]         Is any of the dir(s) enabled in MODULEPATH
  path            modulefile        Print modulefile path
  paths           modulefile        Print path of matching available modules
  clear           [-f]              Reset Modules-specific runtime information
  source          scriptfile [...]  Execute scriptfile(s)
  config [--dump-state|name [val]]  Display or set Modules configuration

Switches:
  -t | --terse    Display output in terse format
  -l | --long     Display output in long format
  -j | --json     Display output in JSON format
  -d | --default  Only show default versions available
```

```
  -L | --latest   Only show latest versions available
  -S | --starts-with
                 Search modules whose name begins with query string
  -C | --contains Search modules whose name contains query string
  -i | --icase    Case insensitive match
  -a | --append   Append directory to MODULEPATH
  -p | --prepend  Prepend directory to MODULEPATH
  --auto          Enable automated module handling mode
  --no-auto       Disable automated module handling mode
  -f | --force    By-pass dependency consistency or confirmation dialog

Options:
  -h | --help     This usage info
  -V | --version  Module version
  -D | --debug    Enable debug messages
  -v | --verbose  Enable verbose messages
  -s | --silent   Turn off error, warning and informational messages
  --paginate      Pipe mesg output into a pager if stream attached to terminal
  --no-pager      Do not pipe message output into a pager
  --color[=WHEN]  Colorize the output; WHEN can be 'always' (default if
                 omitted), 'auto' or 'never'
```

2.3.2 Managing Environment Modules As A User

There is a good chance the cluster administrator has set up the user's account, `fred` for example, so that some modules are loaded already by default. In that case, the modules loaded into the user's environment can be seen with the `module list` command:

Example

```
[fred@basecm11 ~]$ module list
Currently Loaded Modulefiles:
 1) shared   2) cmsh   3) cmd   4) cluster-tools/HEAD   5) cm-setup/HEAD   6) gcc/14.2.0
```

If there are no modules loaded by default, then the `module list` command just returns nothing.

How does a user know what modules are available? The "`module avail`" command lists all modules that are available for loading (some output elided):

Example

```
fred@basecm11:~$ module avail
---------------------------- /cm/local/modulefiles -----------------------------
boost/1.81.0              dot              module-git   python312
cluster-tools/11.0        freeipmi/1.6.14  module-info  rocm-smi/4.3.0
cm-bios-tools             gcc/14.2.0       modules      sedutil/1.16.0
cm-image/HEAD             gdb/16.2         mpc/1.3.1    shared
cm-nvfwupd/2.0.5          gmp/6.3.0        mpfr/4.2.1   slurm/slurm/24.11
cm-scale/cm-scale.module  ipmitool/1.8.19  null         use.own
cm-setup/HEAD             lua/5.4.7        openldap
cmd                       luajit           prs
cmsh                      mariadb-libs     python3

---------------------------- /cm/shared/modulefiles ----------------------------
cm-pmix3/3.1.7       hpl/2.3        mvapich2/gcc/64/2.3.7    ucx/1.18.0
cm-pmix4/4.1.3       hwloc/1.11.13  openblas/dynamic/0.3.28
default-environment  hwloc2/2.8.0   openmpi/gcc/64/4.1.5
hdf5_18/1.8.21       iperf/3.17.1   openmpi4/gcc/4.1.5
```

In the list there are two kinds of modules:

- **local modules**, which are specific to the node, or head node only
- **shared modules**, which are made available from a shared storage, and which only become available for loading after the `shared` module is loaded.

Modules can be loaded using the `add` or `load` options. A list of modules can be added by spacing them:

Example

```
[fred@basecm11 ~]$ module add shared gcc openmpi/gcc
```

Tab completion works for suggesting modules for the `add/load` commands. If the tab completion suggestion is unique, even though it is not the full path, then it is still enough to specify the module. For example, looking at the possibile available modules listed by the `avail` command previously, it turns out that specifying `gcc` is enough to specify `gcc/14.2.0` because there is no other directory path under `gcc/` besides `14.2.0` anyway.

To remove one or more modules, the `module unload` or `module rm` command is used.

To remove all modules from the user's environment, the `module purge` command is used.

The user should be aware that some loaded modules can conflict with others loaded at the same time. This can happen with MPI modules. For example, loading `openmpi/gcc` without removing an already loaded `intel/mpi/64` can result in conflicts about which compiler should be used.

The `shared` Module

The `shared` module provides access to shared libraries. By default these are under `/cm/shared`.

The `shared` module is special because often other modules, as seen under `/cm/shared/modulefiles`, depend on it. So, if it is to be loaded, then it is usually loaded first, so that the dependent modules can use it.

The `shared` module is obviously a useful local module, and is therefore often configured to be loaded for the user by default. Setting the default environment modules is discussed in section 2.3.3.

2.3.3 Changing The Default Environment Modules

.

If a user has to manually load up the same modules every time upon login it would be inefficient. That is why an initial default state for modules can be set up by the user, by using the `module init*` subcommands:

The more useful ones of these are:

- `module initadd`: add a module to the initial state
- `module initrm`: remove a module from the initial state
- `module initlist`: list all modules loaded initially
- `module initclear`: clear all modules from the list of modules loaded initially

Example

```
[fred@basecm11 ~]$ module initclear
[fred@basecm11 ~]$ module initlist
bash initialization file $HOME/.bashrc loads modules:

[fred@basecm11 ~]$ module initadd shared gcc openmpi/gcc
[fred@basecm11 ~]$ module initlist
bash initialization file $HOME/.bashrc loads modules:
   shared gcc openmpi/gcc
```

In the preceding example, the modules defined for the new initial environment for the user are loaded from the next login onward.

Example

```
[fred@basecm11 ~]$ module list
No Modulefiles Currently Loaded.
[fred@basecm11 ~]$ exit
logout
Connection to basecm11 closed
[root@basejumper ~]# ssh fred@basecm11
fred@basecm11's password:
...
[fred@basecm11 ~]$ module list
Currently Loaded Modulefiles:
 1) shared   2) gcc/9.2.0   3) openmpi/gcc/64/1.10.7
[fred@basecm11 ~]$
```

If the user is unsure about what the module does, it can be checked using "`module whatis`":

```
$ module whatis openmpi/gcc
------------------------------- /cm/shared/modulefiles --------------------------------
openmpi/gcc/64/1.10.7: adds OpenMPI to your environment variables
```

The `man` pages for `module` and `modulefile` give further details on usage.

2.4 Compiling Applications

Compiling an application is usually done on the head node or login node. Typically, there are several compilers available on the head node in BCM. These compilers provide different levels of optimization, standards conformance, and support for accelerators.

For example: The GNU compiler collection, Intel compilers, and the NVIDIA HPC SDK compilers. The following table summarizes the available compiler commands on a cluster with these compilers:

Language	GNU	NVIDIA HPC	Intel
C	`gcc`	nvc and nvcc	`icc`
C++	`g++`	nvc++ and nvcc	`icc`
Fortran77	`gfortran`	nvfortran	`ifort`
Fortran90	`gfortran`	nvfortran	`ifort`
Fortran95	`gfortran`	nvfortran	`ifort`

GNU compilers are the de facto standard on Linux and are installed by default. They are provided under the terms of the GNU General Public License. Commercial compilers by Portland and Intel are available as packages via the BCM YUM repository, and require the purchase of a license to use them. To make a compiler available to be used in a user's shell commands, the appropriate environment module (section 2.3) must be loaded first. On most clusters two versions of GCC are available:

1. The version of GCC that comes along with the Linux distribution. For example, for CentOS 7.x:

   Example

   ```
   [fred@basecm11 ~]$ which gcc; gcc --version | head -1
   /usr/bin/gcc
   gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-39)
   ```

2. The latest version suitable for general use that is packaged as a module by BCM:

Example

```
[fred@basecm11 ~]$ module load gcc
[fred@basecm11 ~]$ which gcc; gcc --version | head -1
/cm/local/apps/gcc/9.2.0/bin/gcc
gcc (GCC) 9.2.0
```

To use the latest version of GCC, the `gcc` module in a default BCM cluster must be loaded. To revert to the version of GCC that comes natively with the Linux distribution, the `gcc` module must be unloaded.

The compilers—GCC, Intel, Portland—in the preceding table are ordinarily used for applications that run on a single node. However, the applications used may fork, thread, and run across as many nodes and processors as they can access if the application is designed that way.

The standard, structured way of running applications in parallel is to use the MPI-based libraries (Chapter 3). These are the MPICH, MVAPIC, Open MPI, and Intel MPI libraries. The libraries link to the underlying compilers of the preceding table.

If the parallel library environment has been loaded, then the following MPI compiler commands automatically use the underlying compilers:

Language	**C**	**C++**	**Fortran77**	**Fortran90**
Command	`mpicc`	`mpicxx`	`mpif77`	`mpif90`

2.4.1 Open MPI And Mixing Compilers

BCM comes with multiple Open MPI packages corresponding to the different available compilers. However, sometimes mixing compilers is desirable. For example, C-compilation may be preferred using `icc` from Intel, while Fortran90-compilation may be preferred using `gfortran` from the GNU Project. In such cases it is possible to override the default compiler path environment variable, for example:

```
[fred@basecm11 ~]$ module list
Currently Loaded Modulefiles:
 1) shared   2) gcc/9.2.0   3) openmpi/gcc/64/1.10.7
[fred@basecm11 ~]$ mpicc --version --showme; mpif90 --version --showme
gcc --version
gfortran --version
[fred@basecm11 ~]$ export OMPI_CC=icc; export OMPI_FC=openf90
[fred@basecm11 ~]$ mpicc --version --showme; mpif90 --version --showme
icc --version
openf90 --version
```

Variables that may be set are OMPI_CC, OMPI_CXX, OMPI_FC, and OMPI_F77. More on overriding the Open MPI wrapper settings is documented in the man pages of `mpicc` in the environment section.

3

Using MPI

3.1 Introduction

The Message Passing Interface (MPI) is a standardized and portable message passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran or the C programming language. MPI libraries allow the compilation of code so that it can be used over a variety of multi-processor systems from SMP nodes to NUMA (non-Uniform Memory Access) systems and interconnected cluster nodes .

Depending on the cluster hardware, the interconnect available may be Ethernet, InfiniBand/OmniPath.

Typically, the cluster administrator has already custom-configured the cluster in a way to suit the workflow of users. However, users that are interested in understanding and using the MPI options available, should find this chapter helpful.

3.2 MPI Libraries

MPI libraries that are commonly integrated by the cluster administrator with NVIDIA Base Command Manager are

- MPICH (`https://www.mpich.org/`)
- MVAPICH (`http://mvapich.cse.ohio-state.edu/`)
- OpenMPI (`https://www.open-mpi.org/`)

The preceding MPI libaries can be used with compilers from GCC, or Intel.
The following MPI library implementations may also be integrated with BCM:

- Intel MPI Library (`https://software.intel.com/en-us/mpi-library`), which works with the Intel compiler suite.
- Open MPI version 4 library with CUDA-awareness. NVIDIA GPUs using CUDA can make use of this library.

3.3 MPI Packages And Module Paths

By default, the only MPI implementations installed are the ones that work using GCC.

The cluster administrator can make available for the user a variety of different MPI implementations for different compilers, by installing the appropriate packages. The actual combination of compiler and MPI implementation that a user needs for job runs can then be loaded by the user with the help of modules (section 2.3.2).

3.3.1 MPI Packages That Can Be Installed, And Their Corresponding Module Paths

The following packages are made available from the BCM repositories at the time of writing (October 2022):

Argonne National Laboratory Ethernet implementation for MPI-1, MPI-2, MPI-3	
package name	**module path**
`mpich-ge-gcc-64`	`mpich/ge/gcc`
`mpich-ge-intel-64`	`mpich/ge/intel`

MVAPICH2 InfiniBand implementation for MPI-3	
package name	**module path**
`mvapich2-gcc-64`	`mvapich2/gcc`
`mvapich2-intel-64`	`mvapich2/intel`

MVAPICH2 InfiniBand implementation with performance scaled messaging for MPI-3	
package name	**module path**
`mvapich2-psmgcc-64`	`mvapich2/psmgcc`
`mvapich2-psmintel-64`	`mvapich2/psmintel`

Open MPI version 1 implementation for Ethernet with MPI-3	
package name	**module path**
`openmpi-ge-gcc-64`	`openmpi/gcc`
`openmpi-ge-intel-64`	`openmpi/intel`

Open MPI version 1 implementation for Ethernet and InfiniBand with MPI-3	
package name	**module path**
`openmpi-geib-gcc-64`	`openmpi/gcc`
`openmpi-geib-intel-64`	`openmpi/intel`

Open MPI version 3 implementation for Ethernet and InfiniBand with MPI-3	
package name	**module path**
`openmpi3-geib-gcc-64`	`openmpi3/gcc`
`openmpi3-ge-gcc-64` (only Ethernet, only GCC)	`openmpi3/gcc`
`openmpi3-ge-intel-64` (only Ethernet, only Intel compiler)	`openmpi3/intel`

Open MPI version 4 implementation for InfiniBand with MPI-3	
package name	**module path**
`openmpi4-gcc` (GCC)	`openmpi4/gcc`
`openmpi4-intel` (Intel compiler)	`openmpi4/intel`

Open MPI version 4 implementation for OFED with MPI-3	
package name	**module path**
cm-openmpi4-cuda11.2-ofed47-gcc9	openmpi4-cuda11.2-ofed47-gcc9
cm-openmpi4-cuda11.2-ofed50-gcc9	openmpi4-cuda11.2-ofed50-gcc9
cm-openmpi4-cuda11.2-ofed51-gcc9	openmpi4-cuda11.2-ofed51-gcc9
cm-openmpi4-cuda11.7-ofed47-gcc9	openmpi4-cuda11.7-ofed47-gcc9
cm-openmpi4-cuda11.7-ofed50-gcc9	openmpi4-cuda11.7-ofed50-gcc9
cm-openmpi4-cuda11.7-ofed51-gcc9	openmpi4-cuda11.7-ofed51-gcc9

Intel MPI library implementation for the Intel compiler	
package name	**module path**
intel-mpi-2019	intel/mpi/64
intel-mpi-2020	intel/mpi/64

3.3.2 Finding The Installed MPI Packages And Their Available Module Paths

Finding The Installed MPI Packages

The packages installed on the cluster can be found with the `rpm -qa` query on a RHEL system.

Example

```
[fred@basecm11 ~]$ # search for packages starting with (open)mpi, mvapich, intel-mpi, cm-openmpi)
[fred@basecm11 ~]$ rpm -qa | egrep '(^mpi|^openmpi|^mvapich|^intel-mpi|cm-openmpi)'
openmpi-geib-gcc-64-1.10.7-656_cm9.2.x86_64
mpich-ge-gcc-64-3.4.2-197_cm9.2.x86_64
mvapich2-gcc-64-2.3.7-213_cm9.2.x86_64
openmpi4-gcc-4.1.2-100042_cm9.2_d17293b429.x86_64
intel-mpi-2020-2019.9-100008_cm9.2_aca4120975.x86_64
cm-openmpi4-cuda11.7-ofed47-gcc9-4.1.4-100012_cm9.2_a9f8de3740.x86_64
openmpi4-intel-4.1.2-100042_cm9.2_d17293b429.x86_64
```

The interconnect and compiler implementation of a package can be worked out from looking at the name of the package.
Here, for example,

```
openmpi-geib-gcc-64-1.10.7-656_cm9.2.x86_64
```

implies: Open MPI version `1.10.7`, compiled for both Gigabit Ethernet (`ge`) and InfiniBand (`ib`), with GCC (`gcc`, the GNU project cross-compiler) for a 64-bit architecture, packaged as a (BCM) cluster manager (`cm`) package, for version HEAD for the `x86_64` architecture.

Finding The Available MPI Modules

The corresponding module paths can be found by a search through the available modules:

Example

```
[fred@basecm11 ~]$ # search for modules starting with (open)mpi or mvapich
[fred@basecm11 ~]$ module -l avail | egrep '(^openmpi|^mpi|^mvapich|^intel/mpi)'
intel/mpi/64/                           default
intel/mpi/64/2020/4.304                                 2021/10/23 22:18:12
mpich/ge/gcc/64/3.4.2                                   2021/11/26 22:33:47
mvapich2/gcc/64/2.3.7                                   2022/07/14 13:02:23
```

```
openmpi/gcc/64/1.10.7                          2022/10/13 22:45:54
openmpi/gcc/64/4.1.2                           2022/09/09 22:52:03
openmpi/intel/64/4.1.2                         2022/09/09 23:35:29
openmpi4-cuda11.7-ofed47-gcc9/4.1.4            2022/10/10 22:37:29
openmpi4/gcc/4.1.2                             2022/09/09 22:52:03
openmpi4/intel/4.1.2                           2022/09/09 23:35:29
```

Tab-completion when searching for modules is another approach.

As in the case for the MPI library, for a module the interconnect and compiler implementation can likewise be worked out from looking at the name of the module. So, if a user would like to use an Open MPI implementation that works with an AMD64 node, using GCC, then loading the module `openmpi/gcc/64` should be suitable.

3.4 The Appropriate Interconnect, Compiler, And MPI Implementation For A Module

3.4.1 Interconnects

Jobs can use particular networks for inter-node communication. The hardware for these networks can be Ethernet or InfiniBand, while the software can be a particular MPI implementation.

Gigabit Ethernet

Gigabit Ethernet is an interconnect that is commonly available in consumer computing. For HPC the next generation 10 Gigabit Ethernet and beyond have also been in use for some time. For Ethernet, no additional modules or libraries are needed. The Open MPI, MPICH and MVAPICH implementations all work with any Ethernet technology.

InfiniBand

InfiniBand is a high-performance switched fabric that has come to dominate the connectivity in HPC applications. An InfiniBand interconnect has a lower latency and somewhat higher throughput than a comparably-priced Ethernet interconnect. This is because of special hardware, as well as special software that shortcuts the networking stack in the operating system layer. This typically provides significantly greater performance for most HPC applications. Open MPI and MVAPICH are suitable MPI implementations for InfiniBand.

3.4.2 Selecting A Compiler And MPI implementation

Once the appropriate compiler module has been loaded, the associated MPI implementation is selected by loading the corresponding library module. In the following simplified list, <*compiler*> indicates a choice of `gcc`, or `intel`:

- `mpich/ge/`<*compiler*>
- `mvapich2/`<*compiler*>
- `openmpi/`<*compiler*>
- `openmpi3/`<*compiler*>
- `openmpi4/`<*compiler*>

Section 3.3.1 should be refered to for a more complete list.

3.5 Compiling And Carrying Out An MPI Run

After the appropriate MPI module has been added to the user environment, the user can start compiling applications.

For a cluster using Ethernet interconnectivity, the `mpich` and `openmpi` implementations may be used. For a cluster using InfiniBand, the `mvapich`, `mvapich2` and `openmpi` implementations may be used. Open MPI's `openmpi` implementations first attempt to use InfiniBand, but revert to Ethernet if InfiniBand is not available.

In this section the loading of modules, compilation, and runs are illustrated.

3.5.1 Example MPI Run

This example covers an MPI run, which can be run inside and outside of a queuing system.

Depending on the libraries and compilers installed on the system by the cluster administrator, the availability of the packages providing these modules may differ. To see a full list of modules on the system the command `module avail` can be typed.

Examples Of Loading MPI Modules

To use `mpirun`, the relevant environment modules must be loaded. For example, to use the `mpich` over Gigabit Ethernet (`ge`) GCC implementation:

```
$ module add mpich/ge/gcc
```

or to use the `openmpi4` Open MPI version 4 MPI-3 Intel implementation:

```
$ module add openmpi4/intel
```

Similarly, to use the `mvapich2` InfiniBand GCC implementation:

```
$ module add mvapich2/gcc
```

Keeping Loading Of Modules Clean

The preceding modules can actually be loaded concurrently, and it works as expected. Paths supplied by the most recently-loaded module override the paths of any previous modules.

For example, if `mpich/ge/gcc` is loaded first, then `openmpi4/intel`, and then `mvapich2/gcc`, as suggested in the preceding excerpts, then the modules might be listed as:

```
$ module list
Currently Loaded Modulefiles:
Currently Loaded Modulefiles:
 1) shared   3) cmd                5) cm-setup/9.2            7) openmpi4/intel/4.1.2
 2) cmsh     4) cluster-tools/9.2  6) mpich/ge/gcc/64/3.4.2  8) mvapich2/gcc/64/2.3.7
```

The path of the MPI compiler `mpicc` is defined by the last module in the modulefiles stack, which is the MVAPICH GCC implementation, as can be seen with:

```
$ which mpicc
/cm/shared/apps/mvapich2/gcc/64/2.3.7/bin/mpicc
```

After removing it, then the path changes to the path supplied by the previous module in the stack, the Open MPI Intel implementation:

```
$ module remove mvapich2/gcc/64/2.3.7
$ which mpicc
/cm/shared/apps/openmpi4/intel/4.1.2/bin/mpicc
```

After removing that module too, the path again changes to the path supplied by the previous module in the stack, the MPICH GCC implementation:

```
$ module remove openmpi4/intel/4.1.2
$ which mpicc
/cm/shared/apps/mpich/ge/gcc/64/3.4.2/bin/mpicc
```

However, because loading modules on top of each other can cause confusion, a user should generally try adding modules in a simple, clean manner.

Examples Compiling And Preparing The MPI Application

The code must be compiled with MPI libraries and an underlying compiler. The correct library command can be found in the following table:

Language	C	C++	Fortran77	Fortran90
Command	mpicc	mpixx	mpif77	mpif90

An MPI application `myapp.c`, built in C, could then be compiled as:

```
$ mpicc myapp.c
```

The `mpicc` compilation requires the underlying compiler (GCC, Intel) already be available. By default, Linux systems have a version of the GCC compiler already in their paths, even if no modules have been loaded. So if the module for an MPI implementation based on GCC is loaded without explicitly loading the GCC compiler, then the `mpicc` compilation still works in this case.

The `a.out` binary that is created can then be executed using the `mpirun` command (section 3.5.1).

Creating A Machine File

A machine file contains a list of nodes which can be used by MPI programs.

The workload management system creates a machine file based on the nodes allocated for a job when the job is submitted with the workload manager job submission tool. So if the user chooses to have the workload management system allocate nodes for the job, then creating a machine file is not needed.

However, if an MPI application is being run "by hand" outside the workload manager, then the user is responsible for creating a machine file manually. Depending on the MPI implementation, the layout of this file may differ.

Machine files can generally be created in two ways:

- Listing the same node several times to indicate that more than one process should be started on each node:

```
node001
node001
node002
node002
```

- Listing nodes once, but with a suffix for the number of CPU cores to use on each node:

```
node001:2
node002:2
```

Running The Application

Creating A Simple Parallel Processing Executable

A simple "hello world" program designed for parallel processing can be built with MPI. After compiling it, it can be used to send a message about how and where it is running:

```
[fred@basecm11 ~]$ cat hello.c
#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[])
{
    int id, np, i;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int processor_name_len;
```

```
    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Get_processor_name(processor_name, &processor_name_len);

    for(i=1;i<2;i++)
    {printf(
    "Hello world from process %03d out of %03d, processor name %s\n",
    id, np, processor_name
    );}

    MPI_Finalize();
    return 0;
}
[fred@basecm11 ~]$ module add openmpi/gcc   #or as appropriate depnding on installed package
[fred@basecm11 ~]$ mpicc hello.c -o hello
```

The preceding compilation works for MPI based on `gcc` because a version of `gcc` is already available by default with the distribution. Its version can be found with, for example:

```
[fred@basecm11 ~]$ gcc --version | head -1
gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-10)
```

If the user has a need for another version of the `gcc` compiler, then that can be loaded before the compilation.

```
[fred@basecm11 ~]$ module load gcc/11.2.0
[fred@basecm11 ~]$ mpicc hello.c -o hello
```

If the user would like to use the Intel compiler with MPI instead, then the steps would instead be something like:

```
[fred@basecm11 ~]$ module load intel/compiler/64/2020/19.1.3 intel/mpi/64/2020/4.304
[fred@basecm11 ~]$ mpicc hello.c -o helloforintel
```

If in doubt about how things are compiled, then the libraries that the binary was compiled with can be checked using `ldd`. For example, the GCC-related libraries used could be checked with:

```
[fred@basecm11 ~]$ ldd hello |grep gcc |awk '{print $1,$2,$3}'
libmpi.so.12 => /cm/shared/apps/openmpi/gcc/64/1.10.7/lib64/libmpi.so.12
libopen-rte.so.12 => /cm/shared/apps/openmpi/gcc/64/1.10.7/lib64/libopen-rte.so.12
libopen-pal.so.13 => /cm/shared/apps/openmpi/gcc/64/1.10.7/lib64/libopen-pal.so.13
```

The compiled binary can be run:

```
[fred@basecm11 ~]$ ./hello
Hello world from process 000 out of 001, processor name basecm11
```

However, it still runs on a single processor, and on the node it is started on, unless it is submitted to the system in a special way.

Running An MPI Executable In Parallel Without A Workload Manager

Compute node environment provided by user's `.bashrc`: After the relevant module files are chosen (section 3.5.1) for MPI, an executable compiled with MPI libraries runs on nodes in parallel when submitted with `mpirun`. The executable running on other nodes loads environment modules on those other nodes by sourcing the `.bashrc` file of the user (section 2.3.3) and flags passed by the `mpirun` command. It is a good idea to ensure that the environment module stack used on the compute node is not confusing.

The environment of the user from the interactive shell prompt is not normally carried over automatically to the compute nodes during an `mpirun` submission. That is, compiling and running the executable normally only works on the local node without a special treatment. To have the executable run on the compute nodes, the right environment modules for the job must be made available on the compute nodes too, as part of the user login process to the compute nodes for that job. Usually the system administrator takes care of such matters in the default user configuration by setting up a default user environment (section 2.3.3), with reasonable `initrm` and `initadd` options. Users are then typically allowed to set up their personal default overrides to the default administrator settings, by placing their own `initrm` and `initadd` options to the `module` command according to their needs, or by specifying an overriding environment in the job submissions.

Running `mpirun` outside a workload manager: When using `mpirun` manually, outside a workload manager environment, the number of processes (`-np`) as well as the number of hosts (`-machinefile`) should be specified. For example, on a cluster with 2 compute-nodes and a machine file as specified on page 26:

Example

```
[fred@basecm11 ~]$ module add mvapich2/gcc gcc/11.2.0  #or as appropriate
[fred@basecm11 ~]$ mpicc hello.c -o hello
[fred@basecm11 ~]$ mpirun -np 4 -machinefile mpirun.hosts ~/hello
Hello world from process 0 of 4 on node001.cm.cluster
Hello world from process 1 of 4 on node002.cm.cluster
Hello world from process 2 of 4 on node001.cm.cluster
Hello world from process 3 of 4 on node002.cm.cluster
```

If the cluster has no InfiniBand connectors, then the preceding `mpirun` fails, because MVAPICH requires InfiniBand. That kind of failure displays an output such as:

```
rdma_open_hca(575)..............: No IB device found
```

Open MPI implementations are more forgiving. They check for InfiniBand and if that is unavailable they go ahead with Ethernet. However they have their own quirks. For example, an attempt to carry out an MPI run with Open MPI might be as follows:

Example

```
[fred@basecm11 ~]$ module initclear; module initadd openmpi/gcc
[fred@basecm11 ~]$ module add openmpi/gcc   #or as appropriate
[fred@basecm11 ~]$ mpicc hello.c -o hello
[fred@basecm11 ~]$ mpirun -np 4 -machinefile mpirun.hosts hello
bash: orted: command not found
--------------------------------------------------------------------------
ORTE was unable to reliably start one or more daemons.
This usually is caused by:
...
```

output snipped

It is generally a good idea to read through the man page for `mpirun` for the MPI implementation that the user is using. In this case, the man page for `mpirun` for Open MPI reveals that here the problem is that the environment in the interactive shell is not carried over to the compute nodes during an `mpirun` submission. So the path for `mpirun` should be specified with with either the `--prefix` option, or as an absolute path:

Example

```
[fred@basecm11 ~]$ /cm/shared/apps/openmpi/gcc/64/1.10.7/bin/mpirun -np 4 -machinefile mpirun.hosts hello
Hello world from process 002 out of 004, processor name node002.cm.cluster
Hello world from process 003 out of 004, processor name node001.cm.cluster
Hello world from process 000 out of 004, processor name node002.cm.cluster
Hello world from process 001 out of 004, processor name node001.cm.cluster
```

The output of the preceding `hello.c` program is actually printed in random order. This can be modified as follows, so that only process 0 prints to the standard output, and other processes communicate their output to process 0:

```
#include "mpi.h"
#include "string.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int  numprocs, myrank, namelen, i;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    char greeting[MPI_MAX_PROCESSOR_NAME + 80];
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    MPI_Get_processor_name( processor_name, &namelen );

    sprintf( greeting, "Hello world, from process %d of %d on %s",
             myrank, numprocs, processor_name );

    if ( myrank == 0 ) {
        printf( "%s\n", greeting );
        for ( i = 1; i < numprocs; i++ ) {
            MPI_Recv( greeting, sizeof( greeting ), MPI_CHAR,
                      i, 1, MPI_COMM_WORLD, &status );
            printf( "%s\n", greeting );
        }
    }
    else {
        MPI_Send( greeting, strlen( greeting ) + 1, MPI_CHAR,
                  0, 1, MPI_COMM_WORLD );
    }

    MPI_Finalize( );
    return 0;
}
```

Example

```
[fred@basecm11 ~]$ module clear
[fred@basecm11 ~]$  module add intel/compiler/64 intel/mpi
[fred@basecm11 ~]$  module list
Currently Loaded Modulefiles:
 1) intel/compiler/64/2020/19.1.3   2) intel/mpi/64/2020/4.304
[fred@basecm11 ~]$ mpicc hello.c -o hello
[fred@basecm11 ~]$  mpirun -np 4 -machinefile mpirun.hosts ./hello
Hello world, from process 0 of 4 on node002
```

```
Hello world, from process 1 of 4 on node002
Hello world, from process 2 of 4 on node002
Hello world, from process 3 of 4 on node003
```

Running the executable with `mpirun` outside the workload manager as shown does not take the resources of the cluster into account. To handle running jobs with cluster resources is of course what workload managers such as Slurm are designed to do. Workload managers also typically take care of what environment modules should be loaded on the compute nodes for a job, via additions that the user makes to a job script.

Running an application through a workload manager via a job script is introduced in Chapter 4.

Appendix A contains a number of simple MPI programs.

3.5.2 Hybridization

OpenMP is an implementation of multi-threading. This is a method of parallelizing whereby a parent thread—a series of instructions executed consecutively—forks a specified number of child threads, and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors and accessing the shared memory of an SMP system.

MPI can be mixed with OpenMP to achieve high performance on a cluster/supercomputer of multi-core nodes or servers. MPI creates processes that reside on the level of node, while OpenMP forks threads on the level of a core within an SMP node. Each process executes a portion of the overall computation, while inside each process, a team of threads is created through OpenMP directives to further divide the problem. This kind of execution makes sense due to:

- the ease of programming that OpenMP provides
- OpenMP not necessarily requiring copies of data structure, which allows for designs that overlap computation and communication
- overcoming the limits of parallelism within the SMP node is of course still possible by using the power of other nodes via MPI.

Example

```
#include<mpi.h>
#include <omp.h>
#include <stdio.h>
#include<stdlib.h>

int main(int argc , char** argv) {
  int size, myrank,namelength;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Get_processor_name(processor_name,&namelength);
  printf("Hello I am Processor %d on %s of %d\n",myrank,processor_name,\
size);
    int tid = 0;  int n_of_threads = 1;
    #pragma omp parallel default(shared) private(tid, n_of_threads)
    {
    #if defined (_OPENMP)
          n_of_threads= omp_get_num_threads();
          tid = omp_get_thread_num();
    #endif
    printf("Hybrid Hello World: I am thread # %d out of %d\n", tid, n_o\
```

```
f_threads);
    }
 MPI_Finalize();
 return 0;
}
```

The program can be compiled as follows:

```
fred@basecm11 ~]$ mpicc -o hybridhello omphello.c -fopenmp
```

To specify the number of OpenMP threads per MPI task the environment variable `OMP_NUM_THREADS` must be set.

Example

```
fred@basecm11 ~]$ export OMP_NUM_THREADS=3
```

The number of threads specified by the variable can then be run over the hosts specified by the `mpirun.hosts` file:

```
fred@basecm11 ~]$ mpirun -np 2 -hostfile mpirun.hosts ./hybridhello
Hello I am Processor 0 on node001 of 2
Hello I am Processor 1 on node002 of 2
Hybrid Hello World: I am thread # 0 out of 3
Hybrid Hello World: I am thread # 2 out of 3
Hybrid Hello World: I am thread # 1 out of 3
Hybrid Hello World: I am thread # 0 out of 3
Hybrid Hello World: I am thread # 2 out of 3
Hybrid Hello World: I am thread # 1 out of 3
```

Benefits And Drawbacks Of Using OpenMP

The main benefit to using OpenMP is that it can decrease memory requirements, with usually no reduction in performance. Other benefits include:

- Potential additional parallelization opportunities besides those exploited by MPI.
- Less domain decomposition, which can help with load balancing as well as allowing for larger messages and fewer tasks participating in MPI collective operations.
- OpenMP is a standard, so any modifications introduced into an application are portable and appear as comments on systems not using OpenMP.
- By adding annotations to existing code and using a compiler option, it is possible to add OpenMP to a code somewhat incrementally, almost on a loop-by-loop basis. The vector loops in a code that vectorize well are good candidates for OpenMP.

There are also some potential drawbacks:

- OpenMP can be hard to program and/or debug in some cases.
- Effective usage can be complicated on NUMA systems due to locality considerations
- If an application is network- or memory- bandwidth-bound, then threading it is not going to help. In this case it will be OK to leave some cores idle.
- In some cases a serial portion may be essential, which can inhibit performance.
- In most MPI codes, synchronization is implicit and happens when messages are sent and received. However, with OpenMP, much synchronization must be added to the code explicitly. The programmer must also explicitly determine which variables can be shared among threads and which ones cannot (parallel scoping). OpenMP codes that have errors introduced by incomplete or misplaced synchronization or improper scoping can be difficult to debug because the error can introduce race conditions which cause the error to happen only intermittently.

3.5.3 Support Thread Levels

MPI defines four "levels" of thread safety. The maximum thread support level is returned by the `MPI_Init_thread` call in the "provided" argument.

An environment variable `MPICH_MAX_THREAD_SAFETY` can be set to different values to increase the thread safety:

MPICH_MAX_THREAD_SAFETY	Supported Thread Level
not set	MPI_THREAD_SINGLE
single	MPI_THREAD_SINGLE
funneled	MPI_THREAD_FUNNELED
serialized	MPI_THREAD_SERIALIZED
multiple	MPI_THREAD_MULTIPLE

3.5.4 Further Recommendations

Users face various challenges with running and scaling large scale jobs on peta-scale production systems. For example: certain applications may not have enough memory per core, the default environment variables may need to be adjusted, or I/O may dominate run time.

Possible ways to deal with these are:

- Trying out various compilers and compiler flags, and finding out which options are best for particular applications.
- Changing the default MPI rank ordering. This is a simple, yet sometimes effective, runtime tuning option that requires no source code modification, recompilation or re-linking. The default MPI rank placement on the compute nodes is SMP style. However, other choices are round-robin, folded rank, and custom ranking.
- Using fewer cores per node is helpful when more memory per process than the default is needed. Having fewer processes to share the memory and interconnect bandwidth is also helpful in this case. For NUMA nodes, extra care must be taken.
- Hybrid MPI/OpenMP reduces the memory footprint. Overlapping communication with computation in hybrid MPI/OpenMP can be considered.
- Some applications may perform better when large memory pages are used.

4

Workload Management

4.1 What Is A Workload Manager?

A workload management system (also known as a queuing system, job scheduler or batch submission system) manages the available resources such as CPUs, GPUs, and memory for jobs submitted to the system by users.

Jobs are submitted by the users using *job scripts*. Job scripts are constructed by users and include requests for resources. How resources are allocated depends upon policies that the system administrator sets up for the workload manager.

4.2 Why Use A Workload Manager?

Workload managers are used so that users do not manually have to keep track of node usage in a cluster in order to plan efficient and fair use of cluster resources.

Users may still perhaps run jobs on the compute nodes outside of the workload manager, if that is administratively permitted. However, running jobs outside a workload manager tends to eventually lead to an abuse of the cluster resources as more people use the cluster. This leads to inefficient use of available resources. It is therefore usually forbidden as a policy by the system administrator on production clusters.

4.3 How Does A Workload Manager Function?

A workload manager uses policies to ensure that the resources of a cluster are used efficiently, and must therefore track cluster resources and jobs. A workload manager is therefore generally able to:

- Monitor:
  - the node status (up, down)
  - all available resources (available cores, memory on the nodes)
  - the state of jobs (queued, on hold, deleted, failed, completed)
- Modify:
  - the status of jobs (freeze/hold the job, resume the job, delete the job)
  - the priority and execution order for jobs
  - the run status of a job. For example, by adding checkpoints to freeze a job.
  - (optional) how related tasks in a job are handled according to their resource requirements. For example, a job with two tasks may have a greater need for disk I/O resources for the first task, and a greater need for CPU resources during the second task.

Some workload managers can adapt to external triggers such as hardware failure, and send alerts or attempt automatic recovery.

4.4 Job Submission Process

Whenever a job is submitted, the workload management system checks on the resources requested by the job script. It assigns cores, accelerators, local disk space, and memory to the job, and sends the job to the nodes for computation. If the required number of cores or memory are not yet available, it queues the job until these resources become available. If the job requests resources that are always going to exceed those that can become available, then the job accordingly remains queued indefinitely.

The workload management system keeps track of the status of the job and returns the resources to the available pool when a job has finished (that is, been deleted, has crashed or successfully completed).

4.5 What Do Job Scripts Look Like?

A job script looks very much like an ordinary shell script, and certain commands and variables can be put in there that are needed for the job. The exact composition of a job script depends on the workload manager used, but normally includes:

- commands to load relevant modules or set environment variables for the job to run
- directives for the workload manager for items associated with the job. These items can be a request for resources, output control, or setting the email addresses for messages to go to
- an execution (job submission) line

When running a job script, the workload manager is normally responsible for generating a machine file based on the requested number of processor cores (np), as well as being responsible for the allocation any other requested resources.

The executable submission line in a job script is the line where the job is submitted to the workload manager. This can take various forms.

Example

For the Slurm workload manager, the line might look like:

```
srun a.out
```

Example

For PBS Professional it may simply be:

```
mpirun ./a.out
```

4.6 Running Jobs On A Workload Manager

The details of running jobs through the following workload managers are discussed later on, for:

- Slurm (Chapter 5)
- PBS Professional (Chapter 6)

5

Slurm

Slurm is a workload management system developed originally at the Lawrence Livermore National Laboratory. Slurm used to stand for Simple Linux Utility for Resource Management. However Slurm has evolved since then, and its advanced state nowadays means that the acronym is obsolete.

Slurm has both a graphical interface and command line tools for submitting, monitoring, modifying and deleting jobs. It is normally used with *job scripts* to submit and execute jobs. Various settings can be put in the job script, such as number of processors, resource usage, and application specific variables.

The steps for running a job through Slurm are to:

- Create the script or executable that will be handled as a job
- Create a job script that sets the resources for the script/executable
- Submit the job script to the workload management system

The details of Slurm usage depends upon the MPI implementation used. The description in this chapter will cover using Slurm's Open MPI implementation, which is quite standard. Slurm documentation can be consulted (`https://slurm.schedmd.com/mpi_guide.html`) if the implementation the user is using is very different.

5.1 Loading Slurm Modules And Compiling The Executable

In section 3.5.1 an MPI "Hello, world!" executable that can run in parallel is created and run in parallel outside a workload manager.

The executable can be run in parallel using the Slurm workload manager. For this, the Slurm module should first be loaded by the user on top of the chosen MPI implementation, in this case Open MPI:

Example

```
[fred@basecm11 ~]$ module list
Currently Loaded Modulefiles:
 1) gcc/11.2.0   2) openmpi/gcc/64/4.1.1
[fred@basecm11 ~]$ module add slurm; module list
Currently Loaded Modulefiles:
1) gcc/11.2.0   2) openmpi/gcc/64/4.1.1   3) slurm/slurm/23.11.6
```

The "`hello world`" executable from section 3.5.1 can then be compiled and run for one task outside the workload manager, on the local host, as:

```
[fred@basecm11 ~]$ mpicc hello.c -o hello
[fred@basecm11 ~]$ mpirun -np 1 hello
```

Adding a full path to `mpirun`, and adding a machine file, would allow it to run on the machine file hosts, instead of just locally.

5.2 Running The Executable With `salloc`

Running it as a job managed by Slurm can be done interactively with the Slurm allocation command, `salloc`, as follows

```
[fred@basecm11 ~]$ salloc mpirun hello
```

Slurm is more typically run as a batch job (section 5.3). However execution via `salloc` uses the same options, and it is more convenient as an introduction because of its interactive behavior.

In a default BCM configuration, Slurm auto-detects the compute node cores available, and by default spreads the tasks across the cores as part of the allocation request. Specifying a machine file is therefore not required.

To change how Slurm spreads the executable across compute nodes is typically determined by the options in the following table:

Short Option	Long Option	Description
`-N`	`--nodes=`	Request this many nodes on the cluster.
		Use all cores on each node by default
`-n`	`--ntasks=`	Request this many tasks on the cluster.
		Defaults to 1 task per node.
`-c`	`--cpus-per-task=`	request this many CPUs per task.
		(not implemented by Open MPI yet)
(none)	`--ntasks-per-node=`	request this number of tasks per node.

The full options list and syntax for `salloc` can be viewed with "`man salloc`".

The requirement of specified options to `salloc` must be met before the executable is allowed to run. So, for example, if `--nodes=4` and the cluster only has 3 nodes, then the executable does not run.

5.2.1 Node Allocation Examples

The following session illustrates and explains some node allocation options and issues for Slurm using a cluster with just 1 compute node and 4 CPU cores:

Default settings: The `hello` MPI executable with default settings of Slurm runs successfully over the first (and in this case, the only) node that it finds:

```
[fred@basecm11 ~]$ salloc mpirun hello
salloc: Granted job allocation 572
Hello world from process 0 out of 4, host name node001
Hello world from process 1 out of 4, host name node001
Hello world from process 2 out of 4, host name node001
Hello world from process 3 out of 4, host name node001
salloc: Relinquishing job allocation 572
```

The preceding output also displays if `-N1` (indicating 1 node) is specified, or if `-n4` (indicating 4 tasks) is specified.

The node and task allocation is almost certainly not going to be done by relying on defaults. Instead, node specifications are supplied to Slurm along with the executable.

To understand Slurm node specifications, the following cases consider and explain where the node specification is valid and invalid.

Number of nodes requested: The value assigned to the -N|--nodes= option is the number of nodes from the cluster that is requested for allocation for the executable. In the current cluster example it can only be 1. For a cluster with, for example, 1000 compute nodes, it could be a number up to 1000.

A resource allocation request for 2 nodes with the --nodes option halts for the current cluster which only has 1 compute node:

```
[fred@basecm11 ~]$ salloc -N2 mpirun hello
salloc: Requested partition configuration not available now
salloc: Pending job allocation 573
salloc: job 573 queued and waiting for resources
```

The default behavior is to patiently wait for resources to become available. This makes sense if a cluster can increase its available resources within a reasonable time period. The user can interupt an salloc in this state with a ctrl-c.

Number of tasks requested per cluster: The value assigned to the -n|--ntasks option is the number of tasks that are requested for allocation from the cluster for the executable. In the current cluster example, which has a single compute node with 4 cores, it can be 1 to 4 tasks. The default CPU resources available on a cluster are the number of available processor cores on the compute nodes, which is 4 on this cluster with a single compute node of 4 cores.

A resource allocation request for 5 tasks for this cluster halts because it exceeds the default resources available on the 4-core cluster:

```
[fred@basecm11 ~]$ salloc -n5 mpirun hello
salloc: Requested partition configuration not available now
salloc: Pending job allocation 574
salloc: job 574 queued and waiting for resources
```

Adding and configuring just one more compute node to the current cluster would allow the resource allocation to succeed, since that would provide at least one more core to the cluster.

Number of tasks requested per node: The value assigned to the --ntasks-per-node option is the number of tasks that are requested for allocation from each compute node on the cluster. In the current cluster example, it can be 1 to 4 tasks. A resource allocation request for 5 tasks per compute node with --ntasks-per-node results in a similar output on this cluster running a single compute node with 4-cores. It gives an output like:

```
[fred@basecm11 ~]$ salloc --ntasks-per-node=5 mpirun hello
salloc: Requested partition configuration not available now
salloc: Pending job allocation 575
salloc: job 575 queued and waiting for resources
```

Adding and configuring another 4-core node to the current cluster would still not allow resource allocation to succeed, because the request is for at least 5 cores per compute node, rather than per cluster.

Restricting the number of tasks that can run per compute node: A resource allocation request for 2 tasks per compute node with the --ntasks-per-node option, and simultaneously an allocation request for 1 task to run on the cluster using the --ntasks option, runs successfully, although it uselessly leaves 2 cores unused on the compute node:

```
[fred@basecm11 ~]$ salloc --ntasks-per-node=2 --ntasks=1 mpirun hello
salloc: Granted job allocation 576
Hello world from process 000 out of 002, processor name node001
Hello world from process 001 out of 002, processor name node001
salloc: Relinquishing job allocation 576
```

The other way round, that is, a resource allocation request for 1 task per node with the `--ntasks-per-node` option, and simultaneously an allocation request for 2 tasks to run on the cluster using the `--ntasks` option, fails on this cluster running 1 compute node with 4 cores. This is because although 1 task can be allocated resources from the single node, resources for 2 tasks are being asked for on the cluster, which requires 2 nodes:

```
[fred@basecm11 ~]$ salloc --ntasks-per-node=1 --ntasks=2 mpirun hello
salloc: error: Job submit/allocate failed: Requested node configuration is not available
salloc: Job allocation 577 has been revoked.
```

5.3 Running The Executable As A Slurm Job Script

Instead of using options appended to the `salloc` command line as in section 5.2, it is usually more convenient to send jobs to Slurm with the `sbatch` command acting on a job script.

A job script is also sometimes called a batch file. In a job script, the user can add and adjust the Slurm options, which are the same as the `salloc` options of section 5.2. The various settings and variables that go with the application can also be adjusted.

5.3.1 Slurm Job Script Structure

A job script submission for the Slurm batch job script format is illustrated by the following:

```
[fred@basecm11 ~]$ cat slurmhello.sh
#!/bin/sh
#SBATCH -o my.stdout
#SBATCH --time=30     #time limit to batch job
#SBATCH -N 4
#SBATCH --ntasks=16
#SBATCH --ntasks-per-node=4
module add gcc/9.2.0 openmpi/gcc/64/1.10.7 slurm
mpirun hello
```

The structure is:

shebang line: shell definition line.

SBATCH lines: optional job script *directives* (section 5.3.2).

shell commands: optional shell commands, such as loading necessary modules.

application execution line: execution of the MPI application using `sbatch`, the Slurm submission wrapper.

In SBATCH lines, "`#SBATCH`" is used to submit options. The various meanings of lines starting with "`#`" are:

Line Starts With	Treated As
`#`	Comment in shell and Slurm
`#SBATCH`	Comment in shell, option in Slurm
`# SBATCH`	Comment in shell and Slurm

After the Slurm job script is run with the `sbatch` command (Section 5.3.4), the output goes into file `my.stdout`, as specified by the "`-o`" command.

If the output file is not specified, then the file takes a name of the form "`slurm-<jobnumber>.out`", where <*jobnumber*> is a number starting from 1.

The command "`sbatch --usage`" lists possible options that can be used on the command line or in the job script. Command line values override script-provided values.

5.3.2 Slurm Job Script Options

Options, sometimes called "directives", can be set in the job script file using this line format for each option:

```
#SBATCH {option} {parameter}
```

Directives are used to specify the resource allocation for a job so that Slurm can manage the job optimally. Available options and their descriptions can be seen with the output of `sbatch --help`. The more overviewable usage output from `sbatch --usage` may also be helpful.

Some of the more useful ones are listed in the following table:

Directive Description	Specified As
Name the job <*jobname*>	`#SBATCH -J <jobname>`
Request at least <*minnodes*> nodes	`#SBATCH -N <minnodes>`
Request <*minnodes*> to <*maxnodes*> nodes	`#SBATCH -N <minnodes>-<maxnodes>`
Request at least <*MB*> amount of temporary disk space	`#SBATCH --tmp <MB>`
Run job for a time of <*walltime*> minutes	`#SBATCH -t <walltime>`
Run job at <*time*> (format: HH:MM MM/DD/YY)	`#SBATCH --begin <time>`
Set the working directory to <*directorypath*>	`#SBATCH -D <directorypath>`
Set error log name to <*jobname.err*>*	`#SBATCH -e <jobname.err>`
Set output log name to <*jobname.log*>*	`#SBATCH -o <jobname.log>`
Mail <*user@address*> on job state change	`#SBATCH --mail-user <user@address>`
Mail on all state changes	`#SBATCH --mail-type=ALL`
Mail on job end	`#SBATCH --mail-type=END`
Run job in partition	`#SBATCH -p <destination>`
Run job using GPU with ID <*number*>, as described in section 7.5.2	`#SBATCH --gres=gpu:<number>`

*By default, both standard output and standard error go to a file:
`slurm-<%j>.out`
where <*%j*> is the job number.

5.3.3 Slurm Environment Variables

Available environment variables include:

```
SLURM_CLUSTER_NAME - name of the Slurm cluster
SLURM_CPUS_ON_NODE - CPUs on allocated node
SLURM_JOB_ID - job ID of executing job
SLURM_JOB_NODELIST - list of nodes allocated to job
SLURM_JOB_NUM_NODES - total number of nodes in job's resource allocation
SLURM_JOB_PARTITION - partition of job
SLURM_NODEID - ID of the nodes allocated
SLURM_NTASKS - total number of processes in current job (same as -n|--ntasks=)
SLURM_PROCID - MPI rank (or relative process ID) of the current process
```

```
SLURM_SUBMIT_DIR - directory from which job was launched
SLURM_TASK_PID - process ID of task started
SLURM_TASKS_PER_NODE - number of tasks to be run on each node (man page gives specification)
```

Typically, end users use SLURM_PROCID in a program so that an input of a parallel calculation depends on it. The calculation is thus spread across processors according to the assigned SLURM_PROCID, so that each processor handles the parallel part of the calculation with different values.

More information on environment variables is also to be found in the man page for sbatch.

5.3.4 Submitting The Slurm Job Script With sbatch

Submitting a Slurm job script created as in the previous section is done by executing the job script with sbatch:

```
[fred@basecm11 ~]$ sbatch slurmhello.sh
Submitted batch job 604
[fred@basecm11 ~]$ cat my.stdout
Hello world from process 001 out of 016, processor name node001
...
```

Queues in Slurm terminology are called "partitions". Slurm has a default queue called defq. The administrator may have removed this or created others.

If a particular queue is to be used, this is typically set in the job script using the -p or --partition option:

```
#SBATCH --partition=bitcoinsq
```

It can also be specified as an option to the sbatch command during submission to Slurm.

5.3.5 Checking And Changing Queued Job Status With squeue, scancel, scontrol And sview

Job Queue Listing With squeue

After a job has gone into a queue, the queue status can be checked using the squeue command. The job number can be specified with the -j option to avoid seeing other jobs.

Example

```
[fred@basecm11 ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
               673      defq slurmhel     fred PD       0:00      5 (PartitionNodeLimit)
               683      defq slurmhel     fred PD       0:00      4 (Resources)
               684      defq slurmhel     fred PD       0:00      4 (Resources)
               685      defq slurmhel     fred PD       0:00      4 (Resources)
               682      defq slurmhel     fred  R       0:02      4 node[001-004]
[fred@basecm11 ~]$ squeue -j 673
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
               673      defq slurmhel     fred PD       0:00      5 (PartitionNodeLimit)
```

The man page for squeue covers other options, and explains the meaning of the output.

A backstory for this squeue listing is that in JOBID 673 the user has tried to batch submit the slurmhello.sh job, while requesting 5 nodes, using the entry #SBATCH -N 5 in slurmhello.sh. However, since only 4 nodes were available for the cluster at the time, the submission is pending.

After modifying the entry to #SBATCH -N 4, and submitting the job several times, the jobs 674 to 681 complete successfully. The job with the ID 682 is seen in the R (running) state, and the rest are in a PD (pending) state, with the reasons being either

- Resources (that there are no resources yet available), or
- PartitionNodeLimit (the number of nodes required by this job cannot yet be met because it is outside of its partition's current limits).

Job Canceling With `scancel`

Jobs can be canceled quietly by job ID with "`scancel <job number>`". The `-v` option gives some feedback. Verbosity increases with `-vv`, `-vvv`, and `-vvvv`.

Example

```
[fred@basecm11 ~]$ scancel -v 673
scancel: Terminating job 673
```

Other cancel options, such as per state (with `-t|--state=`), per partition (with `-p|--partition=`), are also possible. The man page for `scancel` has details.

Queued Job Changes With `scontrol`

The `scontrol` command allows users to see and change the job directives while the job is still queued. For example, a user may have specified a job, using the `--begin` directive, to start at 10am the next day by mistake. To change the job to start at 10pm tonight, something like the following session may take place:

```
[fred@basecm11 ~]$ scontrol show jobid=254 | grep Time
   RunTime=00:00:04 TimeLimit=UNLIMITED TimeMin=N/A
   SubmitTime=2011-10-18T17:41:34 EligibleTime=2011-10-19T10:00:00
   StartTime=2011-10-18T17:44:15 EndTime=Unknown
   SuspendTime=None SecsPreSuspend=0
```

The parameter that should be changed is "`EligibleTime`", which can be done as follows:

```
[fred@basecm11 ~]$ scontrol update jobid=254 EligibleTime=2011-10-18T22:00:00
```

GUI Slurm Control `sview`

An approximate GUI Slurm equivalent to `scontrol` is the `sview` tool. This allows the job to be viewed under its jobs tab, and the job to be edited with a right click menu item. It can also carry out many other functions, including canceling a job.

Webbrowser-accessible job viewing is possible from the workload tab of the `User Portal` (section 10.2).

6

PBS Professional And OpenPBS

NVIDIA Base Command Manager works with PBS Professional and OpenPBS, which are the modern editions of the original Portable Batch System (PBS) software. The original PBS software was a workload management and job scheduling system to manage computing resources, and was originally developed at NASA in the 1990s.

In this manual, PBS is used as a shorter form to refer to both the modern editions.

PBS *job scripts* are used to submit and execute jobs. The user puts values into a job script for the resources being requested, such as the number of processors to be used, the memory to be used, or number of nodes required. Other values are also set for the runtime parameters and application-specific variables.

The steps for running a job through a PBS job script are:

- Creating an application to be run via the job script
- Creating the job script, adding directives, applications, runtime parameters, and application-specific variables to the script
- Submitting the script to the workload management system

This chapter covers the using the workload managers and job scripts with the PBS variants so that users can get a basic understanding of how they are used, and can get started with typical cluster usage.

In this chapter:

- section 6.1 covers the components of a job script and job script examples
- section 6.2.1 covers submitting, monitoring, and deleting a job with a job script

More on using PBS is to be found in the PBS Professional Guides, which can be accessed from: `https://community.altair.com/community?id=altair_product_documentation`.

6.1 Components Of A Job Script

To use PBS, a batch job script is created by the user. The job script is a shell script containing the set of commands that the user wants to run. It also contains the resource requirement directives and other specifications for the job. After preparation, the job script is submitted to the workload manager using the `qsub` command. The workload manager then tries to make the job run according to the job script specifications.

A job script can be resubmitted with different parameters (e.g. different sets of data or variables).

6.1.1 Sample Script Structure

A job script in PBS has a structure illustrated by the following basic example:

Example

```
#!/bin/bash
#
#PBS -l walltime=1:00:00
#PBS -l select=2:ncpus=1:mem=500mb
#PBS -j oe

cd ${HOME}/myprogs
mpirun myprog a b c
```

The first line is the standard "shebang" line used for scripts.

The lines that start with `#PBS` are PBS directive lines, described shortly in section 6.1.2.

The last two lines are an example of setting any remaining options or configuration settings up, so that the script can run. In this case, a change to the directory `myprogs` is made, and then the executable `myprog` is run, with arguments `a b c`. The line that runs the program is called the executable line (section 6.1.3).

To run the executable file in the executable line in parallel, the job launcher `mpirun` is placed immediately before the executable file. The number of nodes the parallel job is to run on is assumed to have been specified in the PBS directives.

6.1.2 Directives

Job Script Directives And `qsub` Options

A job script typically has several configurable values called job script directives, set with job script directive lines. These are lines that start with a "`#PBS`". Any directive lines beyond the first executable line are ignored.

The lines are comments as far as the shell is concerned because they start with a "`#`". However, at the same time the lines are special commands when the job script is processed by the `qsub` command. The difference is illustrated by the following:

- The following shell comment is only a comment for a job script processed by `qsub`:

  ```
  # PBS
  ```

- The following shell comment is also a job script directive when processed by `qsub`:

  ```
  #PBS
  ```

Job script directive lines with the `#PBS` part removed are the same as options applied to the `qsub` command, so a look at the `man` pages of `qsub` describes the possible directives and how they are used. If there is both a job script directive and a `qsub` command option set for the same item, then the `qsub` option takes precedence.

Since the job script file is a shell script, the shell interpreter used can be changed to another shell interpreter by modifying the first line (the "`#!`" line) to the preferred shell. Any shell specified by the first line can also be overridden by using the "`#PBS -S`" directive to set the shell path.

Walltime Directive

The workload manager typically has default *walltime* limits per queue with a value limit set by the administrator. The user can set a walltime limit for the job by setting the "`#PBS -l walltime`" directive to a specific time. The time specified is the maximum time that the user expects the job should run for, and it allows the workload manager to work out an optimum time to run the job. The job can then run sooner than it would by default.

If the walltime limit is exceeded by a job, then the job is stopped, and an error message in the following format is displayed:

```
=>> PBS: job killed: walltime <running time> exceeded limit <set time>
```

Here, <*running time*> is the time that the job actually took to run after it ran, while <*set time*> is the time that the user set as the walltime resource limit.

If there is no default walltime set by the cluster administrator, and if the user submits a job with no walltime set too, then the job only runs when all jobs with a walltime have made room. So to avoid waiting a very long time, and to experience a reasonably consistent behavior on different clusters, the user really should set a walltime.

Resource List Directives

Resource list directives specify arguments to the `-l` directive of the job script, and allow users to specify values to use instead of the system defaults.

For example, in the sample script structure earlier, a job walltime of one hour and a memory space of at least 500MB are requested (the script requires the size of the space be spelled in lower case, so "`500mb`" is used).

If a requested resource list value exceeds what is available, the job is queued until resources become available.

For example, if nodes only have 2000MB to spare and 4000MB is requested, then the job is queued indefinitely, and it is up to the user to fix the problem.

Resource list directives also allow, for example, the number of nodes (`-l select=2:`) and the number of processor cores for each node, `ncpus`, to be specified. If no value is specified, then the default is 1 node, and 1 core per node (`ncpus=1`).

As an aside: To avoid specifying `ncpus` with a directive, the default value for `ncpu` can be configured by the cluster administrator with the PBS `qmgr` configuration setting (section 7.11.1 of the *Administrator Manual*):

Example

```
qmgr -c "set server resources_default.ncpus = 4"
```

A directive for a resource such as the wall time applies to the entire job. In PBS, such a resource is called a *job-wide* resource.

A collection of resources allocated to a job, but with the resources being requested from the same physical compute node, is called a *chunk*. A directive can be applied at chunk level with the `select=`*chunk name* option. Typically, an MPI job has one chunk per MPI process (rank).

Thus, requested resources can be job-wide only (for example, `walltime`), or chunk only (for example, `ncpus`). In a job, a resource cannot be used as chunk as well as job-wide—it can only be used as chunk (one or multiple chunks), or job-wide.

So, to run a job on 8 cores, the job-wide specification could be done with:

```
#PBS -l select=8:ncpus=1
```

The preceding specification requests 8 CPU cores, and the cores can be anywhere on the cluster.

To run a job on one chunk, the chunk specification could be done with:

```
#PBS -l select=1:ncpus=8
```

The preceding specification requests 8 CPU cores, and the cores must be on the same physical node.

Further examples of node resource specification are given in a table on page 47.

Job Directives: Job Name, Logs, And IDs

If the name of the job script file is `jobname`, then by default the output and error streams are logged to `jobname.o`<*number*> and `jobname.e`<*number*> respectively, where <*number*> indicates the associated job number. The default paths for the logs can be changed by using the `-o` and `-e` directives respectively, while the base name (`jobname` here) can be changed using the `-N` directive.

Often, a user may simply merge both logs together into one of the two streams using the -j directive. Thus, in the preceding example, "`-j oe`" merges the logs to the output log path, while "`-j eo`" would merge it to error log path.

The job ID is an identifier based on the job number and the FQDN of the login node. For a login node called `basecm11.cm.cluster`, the job ID for a job number with the associated value <*number*> from earlier, would by default be <*number*>.basecm11.cm.cluster, but it can also simply be abbreviated to <*number*>.

Job Queues

Sending a job to a particular job queue is sometimes appropriate. An administrator may have set queues up so that some queues are for very long term jobs, or some queues are for users that require GPUs. Submitting a job to a particular queue <*destination*> is done by using the directive "`#PBS -q <destination>`".

Directives Summary

Some useful job directives are illustrated in the following table:

Directive Description	Specified As
Name the job <*jobname*>	`#PBS -N <jobname>`
Run the job for a maximum runtime of <*walltime*>	`#PBS -l <walltime>`
Run the job for a maximum runtime of 3 hours 10 minutes and 30 seconds	`#PBS -l walltime=03:10:30`
Run the job at <*time*>	`#PBS -a <time>`
Set error log name to <*jobname.err*>	`#PBS -e <jobname.err>`
Set output log name to <*jobname.log*>	`#PBS -o <jobname.log>`
Join standard error and standard output to standard error	`#PBS -j eo`
Join standard error and standard output to standard output	`#PBS -j oe`
Mail to <*user@address*>	`#PBS -M <user@address>`
Mail on <*event*>	`#PBS -m <event>`
where <*event*> takes the	`(a)bort`
value of the letter in	`(b)egin`
the parentheses	`(e)nd`
	`(n) do not send email`
Queue is <*destination*>	`#PBS -q <destination>`
Login shell path is <*shellpath*>	`#PBS -S <shellpath>`

Almost every `qsub` sets a job attribute, and has a corresponding PBS directive with the same syntax as the option. The man page for `qsub` and the man page for `pbs_job_attributes` explain this in detail.

Resource Request Examples

As PBS evolved, the specification for requesting nodes has changed. The form:

```
#PBS -l nodes=3
```

is deprecated, and automatically converted to:

```
#PBS -l select=3
```

The deprecated changes are described in detail the man page for `pbs_resources`, in the section on `BACKWARD COMPATIBILITY`.

Examples of requests for `select=` options are shown in the following table:

Resource Request Example Description	`#PBS -l` Specification
8 nodes, anywhere on the cluster	`select=8`
2 nodes, 1 processor per node	`select=2:ncpus=1`
3 nodes, 8 processors per node	`select=3:ncpus=8`
5 nodes, 2 processors per node, and 1 GPU per node	`select=5:ncpus=2:ngpus=1`
5 nodes, 2 processors per node, 3 virtual processors for MPI code	`select=5:ncpus=2:mpiprocs=3`
5 nodes, 2 processors per node, using any GPU on the nodes	`select=5:ncpus=2:ngpus=1`
5 nodes, 2 processors per node, using a GPU with ID 0 from nodes	`select=5:ncpus=2:gpu_id=0`

Some of the examples illustrate requests for GPU resource usage. GPUs and the CUDA utilities for NVIDIA are introduced in Chapter 7. GPU usage is treated by the workload manager like the attributes of a resource which the cluster administrator will have pre-configured according to local requirements.

For further details on resources, the man page for `pbs_resources` can be checked.

6.1.3 The Executable Line

In the job script structure (section 6.1.1), the executable line is launched with the job launcher command after the directives lines have been dealt with, and after any other shell commands have been carried out to set up the execution environment.

Using `mpirun` In The Executable Line

The `mpirun` command is used for executables compiled with MPI libraries. Executables that have not been compiled with MPI libraries, or which are launched without any specified number of nodes, run on a single free node chosen by the workload manager.

The executable line to run a program `myprog` that has been compiled with MPI libraries is run by placing the job-launcher command `mpirun` before it as follows:

```
mpirun myprog
```

6.1.4 Example Batch Submission Scripts

Node Availability

The following job script tests which out of 4 nodes requested with "`-l nodes`" are made available to the job in the workload manager:

Example

```
#!/bin/bash
#PBS -l walltime=1:00
#   #PBS -l nodes=4     <--- legacy
#PBS -l select=4
echo -n "I am on: "
hostname;

echo finding ssh-accessible nodes:
for node in $(cat ${PBS_NODEFILE}) ; do
    echo -n "running on: "
    /usr/bin/ssh $node hostname
```

```
done
```

The directive specifying `walltime` means the script runs at most for 1 minute. The `${PBS_NODEFILE}` array used by the script is created and appended with hosts by the queuing system. The script illustrates how the workload manager generates a `${PBS_NODEFILE}` array based on the requested number of nodes, and which can be used in a job script to spawn child processes. When the script is submitted, the output from the log will look like:

```
I am on: node001
finding ssh-accessible nodes:
running on: node001
running on: node002
running on: node003
running on: node004
```

This illustrates that the job starts up on a node, and that no more than the number of nodes that were asked for in the resource specification are provided.

The list of all nodes for a cluster can be found using the `pbsnodes` command (section 6.2.6).

Using InfiniBand

A sample PBS script for InfiniBand is:

```
#!/bin/bash
#!
#! Sample PBS file
#!
#! Name of job

#PBS -N MPI

#! Number of nodes (in this case 8 nodes with 4 CPUs each)
#! The total number of nodes passed to mpirun will be nodes*ppn
#! Second entry: Total amount of wall-clock time (true time).
#! 02:00:00 indicates 02 hours

#PBS -l walltime=02:00:00
#PBS -l select=8:ncpus=4

#! Mail to user when job terminates or aborts
#PBS -m ae

# If modules are needed by the script, then source modules environment:
. /etc/profile.d/modules.sh

# Add any modules you might require:
module add shared mvapich/gcc pbspro

#! Full path to application + application name
application="<application>"

#! Run options for the application
options="<options>"

#! Work directory
workdir="<work dir>"
```

```
###############################################################
### You should not have to change anything below this line ####
###############################################################

#! change the working directory (default is home directory)

cd $workdir

echo Running on host $(hostname)
echo Time is $(date)
echo Directory is $(pwd)
echo PBS job ID is $PBS_JOBID
echo This job runs on the following machines:
echo $(cat $PBS_NODEFILE | uniq)

mpirun_command="mpirun $application $options"

#! Run the parallel MPI executable (nodes*ppn)
echo Running $mpirun_command
eval $mpirun_command
```

In the preceding script, no machine file is needed, since it is automatically built by the workload manager and passed on to the `mpirun` parallel job launcher utility. The job is given a unique ID and run in parallel on the nodes based on the resource specification.

6.1.5 Links To PBS Resources

A number of useful links are:

- The PBS Professional documentation pages, accessible from `https://community.altair.com/community?id=altair_product_documentation`.
- The man pages for `qsub`, `pbs_resources`, and `pbs_job_attributes`, for setting up job scripts.
- The community support pages for PBS Professional, accessible at `https://community.altair.com/community?id=altair_community_home`. Commercial support can be arranged for PBS Professional using `https://www.altair.com/tailored-solutions`.
- The OpenPBS community pages, accessible from `http://community.openpbs.org/`.

6.2 Submitting A Job

6.2.1 Preliminaries: Loading The Modules Environment

To submit a job to the workload management system, the user must ensure that the following environment modules are loaded:

```
$ module add shared pbspro
```

Users can pre-load particular environment modules as their default using the "`module init*`" commands (section 2.3.3).

6.2.2 Using `qsub`

The command `qsub` is used to submit jobs to the workload manager system. The command returns a unique job identifier, which is used to query and control the job and to identify output. The usage format of `qsub` and some useful options are listed here:

```
USAGE: qsub [<options>] <job script>

Option  Hint   Description
------  ----   -----------
-a      at     run the job at a certain time
-l      list   request certain resource(s)
-q      queue  job is run in this queue
-N      name   name of job
-S      shell  shell to run job under
-j      join   join output and error files
```

For example, a job script called `mpirun.job` with all the relevant directives set inside the script, may be submitted as follows:

Example

```
$ qsub mpirun.job
```

A job may be submitted to a specific queue `testq` as follows:

Example

```
$ qsub -q testq mpirun.job
```

The `man` page for `qsub` describes these and other options. The options correspond to PBS directives in job scripts (section 6.1.1). If a particular item is specified by a `qsub` option as well as by a PBS directive, then the `qsub` option takes precedence.

6.2.3 Job Output

By default, the output from the job script <*scriptname*> goes into the current working directory for PBS.

By default, error output is written to <*scriptname*>`.e`<*jobid*> and the application output is written to <*scriptname*>`.o`<*jobid*>, where <*jobid*> is a unique number that the workload manager allocates. Specific output and error files can be set using the `-o` and `-e` options respectively. The error and output files can usefully be concatenated into one file with the `-j oe` or `-j eo` options. More details on this can be found in the `qsub` man page.

6.2.4 Monitoring The Status Of A Job

To use the commands in this section, the appropriate workload manager module must be loaded:

```
$ module add pbspro
```

`qstat` Basics

The main component is `qstat`, which has several options. In this example, the most frequently used options are discussed.

In PBS, the command "`qstat -an`" shows what jobs are currently submitted or running on the queuing system. An example output is:

```
[fred@basecm11 ~]$ qstat -an

basecm11:                                                     Req'd  Req'd   Elap
Job ID          Username Queue    Jobname    SessID NDS TSK Memory Time  S Time
--------------- -------- -------- ---------- ------ --- --- ------ ----- - -----
121.basecm11    noah     workq    pbjob         --    3   6    --  01:00 Q   --
    --
125.basecm11    noah     workq    pbjob       26615   3   3    --  01:00 R   --
   node001/0+node002/0+node003/0
```

The output shows the Job ID, the user who owns the job, the queue, the job name, the session ID for a running job, the number of nodes requested, the number of CPUs or tasks requested, the time requested (`-l walltime`), the job state (S) and the elapsed time. In this example, one job is seen to be running (R), and one is still queued (Q). The `-n` parameter causes nodes that are in use by a running job to display at the end of that line.

Possible job states include:

Job States	Description
E	Job is exiting after having run
F	Job is finished
H	Job is held
Q	job is queued, eligible to run or routed
R	job is running
S	job is suspended
T	job is being moved to new location
W	job is waiting for its execution time

The command "`qstat -q`" shows what queues are available. In the following example, there is one job running in the `testq` queue and 4 are queued.

```
$ qstat -q

server: master.cm.cluster

Queue            Memory CPU Time Walltime Node  Run Queue Lm  State
---------------- ------ -------- -------- ----  --- ----- --  -----
testq              --      --    23:59:59   --    1     4 --   E R
default            --      --    23:59:59   --    0     0 --   E R
                                               ---- -----
                                                  1     4
```

Viewing Job Details With `qstat`

With `qstat -f` the full output of the job is displayed. The output shows what the jobname is, where the error and output files are stored, and various other settings and variables.

```
$ qstat -f
Job Id: 137.pj-cruncher
    Job_Name = pbjob
    Job_Owner = noah@pj-cruncher.cm.cluster
    resources_used.cpupercent = 0
    resources_used.cput = 00:00:00
    resources_used.mem = 5456kb
    resources_used.ncpus = 3
    resources_used.vmem = 384164kb
    resources_used.walltime = 00:00:11
    job_state = R
    queue = workq
    server = pj-cruncher
    Checkpoint = u
    ctime = Fri Mar 20 18:07:06 2020
    Error_Path = pj-cruncher.cm.cluster:/home/noah/pbjob.e137
    exec_host = node001/0+node002/0+node003/0
    exec_vnode = (node001:ncpus=1)+(node002:ncpus=1)+(node003:ncpus=1)
    Hold_Types = n
```

```
    Join_Path = oe
    Keep_Files = n
    Mail_Points = a
    mtime = Fri Mar 20 18:08:52 2020
    Output_Path = pj-cruncher.cm.cluster:/home/noah/pbjob.o137
    Priority = 0
    qtime = Fri Mar 20 18:07:06 2020
    Rerunable = True
    Resource_List.mpiprocs = 24
    Resource_List.ncpus = 3
    Resource_List.nodect = 3
    Resource_List.place = free
    Resource_List.select = 3:ncpus=1:mpiprocs=8
    Resource_List.walltime = 01:00:00
    stime = Fri Mar 20 18:08:42 2020
    session_id = 28107
    jobdir = /home/noah
    substate = 42
    Variable_List = PBS_O_HOME=/home/noah,PBS_O_LANG=en_US.UTF-8,
        PBS_O_LOGNAME=noah,
        PBS_O_PATH=/cm/shared/apps/openpbs/20.0.1/unsupported/fw/bin:/cm/sha
        red/apps/openpbs/20.0.1/unsupported:/cm/shared/apps/openpbs/20.0.1/
        sbin:/cm/shared/apps/openpbs/20.0.1/bin:/cm/shared/apps/mpich/ge/gcc/
        64/3.3.2/bin:/cm/local/apps/gcc/9.2.0/bin:/cm/local/apps/environment-mo
        dules/4.4.0//bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbi
        n:/usr/sbin:/cm/local/apps/environment-modules/4.4.0/bin:/home/noah/.lo
        cal/bin:/home/noah/bin,PBS_O_MAIL=/var/spool/mail/noah,
        PBS_O_SHELL=/bin/bash,PBS_O_WORKDIR=/home/noah,PBS_O_SYSTEM=Linux,
        PBS_O_QUEUE=workq,PBS_O_HOST=pj-cruncher.cm.cluster
    comment = Job run at Fri Mar 20 at 18:08 on (node001:ncpus=1)+(node002:ncpu
        s=1)+(node003:ncpus=1)
    etime = Fri Mar 20 18:07:06 2020
    run_count = 1
    Submit_arguments = pbjob
    project = _pbs_project_default
```

6.2.5 Deleting A Job

An already submitted job can be deleted using the `qdel` command:

```
$ qdel <jobid>
```

Multiple space-separated job IDs can be specified.

6.2.6 Nodes According To PBS

The nodes that the workload manager knows about can be viewed using the `pbsnodes` command.

The following output is from a cluster made up of nodes with 4 physical cores, as indicated by the value of 4 for `pcpus`. If the node is available to run scripts, then its state is `free` or `time-shared`. When a node is used exclusively (section 7.5.2) by one script, the state is `job-exclusive`.

For PBS Professional the display resembles (some output elided):

```
[noah@pj-cruncher ~]$ pbsnodes -a
node001
     Mom = node001.cm.cluster
     ntype = PBS
     state = free
```

```
     pcpus = 4
     resources_available.arch = linux
     resources_available.host = node001
     resources_available.mem = 4044784kb
     ...
     queue = workq
     resv_enable = True
     sharing = default_shared
     ...
node002
     Mom = node002.cm.cluster
     ntype = PBS
     state = free
     pcpus = 4
     resources_available.arch = linux
     resources_available.host = node002
     resources_available.mem = 4044784kb
     ...
     queue = workq
     resv_enable = True
     sharing = default_shared
     last_state_change_time = Fri Mar 20 18:09:47 2020
     last_used_time = Fri Mar 20 18:09:47 2020
     ...
```

7

Using GPUs

GPUs (Graphics Processing Units) are chips that provide specialized parallel processing power. Originally, GPUs were designed to handle graphics processing as part of the video processor, but their ability to handle non-graphics tasks in a similar manner has become important for general computing. GPUs designed for general purpose computing task are commonly called General Purpose GPUs, or GPGPUs.

A GPU is suited for processing an algorithm that naturally breaks down into a process requiring many similar calculations running in parallel. GPUs cores are able to rapidly apply the instruction on multiple data points organized in a 2-D, and more recently, 3-D, image. The image is placed in a framebuffer. In the original chips, the data points held in the framebuffer were intended for output to a display, thereby accelerating image generation.

The similarity between multicore CPU chips and modern GPUs makes it at first sight attractive to use GPUs for general purpose computing. However, the instruction set on GPGPUs is used in a component called the *shader pipeline*. This has, as the name suggests, to do with a limited set of graphics operations, and so is by its nature rather limited. Using the instruction set for problems unrelated to shader pipeline manipulations requires that the problems being processed map over to a similar manipulation. This works best for algorithms that naturally break down into a process requiring an operation to be applied in the same way on many independent vertices and pixels. In practice, this means that 1-D vector operations are an order of magnitude less efficient on GPUs than operations on triangular matrices.

Modern GPGPU implementations have matured so that they can now sub-divide their resources between independent processes that work on independent data, and they provide programmer-friendlier ways of data transfer between the host and GPU memory.

Physically, one GPU is typically a built-in part of the motherboard of a node or a board in a node, and consists of several hundred processing cores. There are also dedicated standalone units, commonly called GPU Units, consisting of several GPUs in one chassis. Several of these can be assigned to particular nodes, typically via PCI-Express connections, to increase the density of parallelism even further.

NVIDIA Base Command Manager has several tools that can be used to set up and program GPUs for general purpose computations.

7.1 Packages

A number of different GPU-related packages are included in BCM. The CUDA versions supported are listed in section 9.1 of the *Installation Manual*.

The version implemented depends on how the system administrator has installed and configured CUDA.

7.2 Using CUDA

After installation of the packages, for general usage and compilation it is sufficient to load just the `CUDA<version>/toolkit` module, where <*version*> indicates the CUDA version number. At the time of

writing of this section (May 2023), for Rocky Linux version 9.1, the versions available are 11.7, 11.8, 12.0, and 12.1.

The toolkit comes with the necessary tools and the NVIDIA compiler wrapper to compile CUDA C code.

Extensive documentation on how to get started, the various tools, and how to use the CUDA suite is at `https://docs.nvidia.com/cuda`.

Also available are several other modules related to CUDA:

- `cuda12.1/blas`: Provides paths and settings for the CUBLAS library.
- `cuda12.1/fft`: Provides paths and settings for the CUFFT library.

7.3 Using OpenCL

OpenCL functionality is provided with the `cuda`*<version>*`/toolkit` environment module.

Examples of OpenCL code can be found in the `$CUDA_SDK/OpenCL` directory.

7.4 Compiling Code

Both CUDA and OpenCL involve running code on different *platforms*:

- `host`: with one or more CPUs
- `device`: with one or more CUDA enabled GPUs

Accordingly, both the host and device manage their own memory space, and it is possible to copy data between them. The CUDA and OpenCL Best Practices Guides in the `doc` directory, provided by the CUDA toolkit package, have more information on how to handle both platforms and their limitations.

The `nvcc` command by default compiles code and links the objects for both the host system and the GPU. The `nvcc` command distinguishes between the two and it can hide the details from the developer. To compile the host code, `nvcc` will use `gcc` automatically.

```
nvcc [options] <inputfile>
```

A simple example to compile CUDA code to an executable is:

```
nvcc testcode.cu -o testcode
```

The most used options are:

- `-g` or `-debug` *<level>*: This generates debuggable code for the host
- `-G` or `-device-debug` *<level>*: This generates debuggable code for the GPU
- `-o` or `-output-file` *<file>*: This creates an executable with the name *<file>*
- `-arch=sm_13`: This can be enabled if the CUDA device supports compute capability 1.3, which includes double-precision

If double-precision floating-point is not supported or the flag is not set, warnings such as the following will come up:

```
warning : Double is not supported. Demoting to float
```

The `nvcc` documentation manual, "*The CUDA Compiler Driver NVCC*" has more information on compiler options.

The CUDA SDK has more programming examples and information accessible from the file `$CUDA_SDK/C/Samples.html`.

For OpenCL, code compilation can be done by linking against the OpenCL library:

```
gcc test.c -lOpenCL
g++ test.cpp -lOpenCL
nvcc test.c -lOpenCL
```

7.5 Available Tools

7.5.1 CUDA gdb

The CUDA debugger can be started using: `cuda-gdb`. Details of how to use it are available in the "*CUDA-GDB (NVIDIA CUDA Debugger)*" manual, in the `doc` directory. It is based on GDB, the GNU Project debugger, and requires the use of the "`-g`" or "`-G`" options compiling.

Example

```
nvcc -g -G testcode.cu -o testcode
```

7.5.2 The `nvidia-smi` Utility

The NVIDIA System Management Interface command, `nvidia-smi`, can be used to allow exclusive access to the GPU. This means only one application can run on a GPU. By default, a GPU will allow multiple running applications.
Syntax:

```
nvidia-smi [OPTION1 [ARG1]] [OPTION2 [ARG2]] ...
```

The steps for making a GPU exclusive:

- List GPUs
- Select a GPU
- Lock GPU to a compute mode
- After use, release the GPU

After setting the compute rule on the GPU, the first application that executes on the GPU blocks out any of the others attempting to run. The first application does not have to be from the user that set the exclusivity lock on the GPU.

To list the GPUs, the `-L` argument can be used:

```
$ nvidia-smi -L
GPU 0: (05E710DE:068F10DE)  Tesla T10 Processor  (S/N: 706539258209)
GPU 1: (05E710DE:068F10DE)  Tesla T10 Processor  (S/N: 2486719292433)
```

To set the ruleset on the GPU, the `-c|--compute-mode` option can be used:

```
$ nvidia-smi -i 0 -c 1
```

The ruleset may be one of the following:

- 0 - Default mode (multiple applications allowed on the GPU)
- 1 - Exclusive thread mode (only one compute context is allowed to run on the GPU, usable from one thread at a time)
- 2 - Prohibited mode (no compute contexts are allowed to run on the GPU)
- 3 - Exclusive process mode (only one compute context is allowed to run on the GPU, usable from multiple threads at a time)

To check the state of the GPU, the `-q|--query` option can be used:

```
$ nvidia-smi -i 0 -q
COMPUTE mode rules for GPU 0: 1
```

In this example, GPU0 is locked, and there is a running application using GPU0. A second application attempting to run on this GPU will not be able to run on this GPU.

```
$ histogram  --device=0
main.cpp(101) : cudaSafeCall() Runtime API error :
no CUDA-capable device is available.
```

After use, the GPU can be unlocked to allow multiple users:

```
nvidia-smi -i 0 -c 0
```

7.5.3 CUDA Utility Library

CUTIL is a simple utility library designed for use in the CUDA SDK samples. There are 2 parts for CUDA and OpenCL. The locations are:

- $$CUDA_SDK/C/lib
- $$CUDA_SDK/OpenCL/common/lib

Other applications may also refer to them, and the toolkit libraries have already been pre-configured accordingly. However, they need to be compiled prior to use. Depending on the cluster, this might have already have been done.

```
[fred@demo ~] cd
[fred@demo ~] cp -r $CUDA_SDK
[fred@demo ~] cd $(basename $CUDA_SDK); cd C
[fred@demo C] make
[fred@demo C] cd $(basename $CUDA_SDK); cd OpenCL
[fred@demo OpenCL] make
```

CUTIL provides functions for:

- parsing command line arguments
- read and writing binary files and PPM format images
- comparing data arrays (typically used for comparing GPU results with CPU results)
- timers
- macros for checking error codes
- checking for shared memory bank conflicts

7.5.4 CUDA "Hello world" Example

A hello world example code using CUDA is:

Example

```
/*
  CUDA example
  "Hello World" using shift13, a rot13-like function.
  Encoded on CPU, decoded on GPU.

  rot13 cycles between 26 normal alphabet characters.

  shift13 shifts 13 steps along the normal alphabet characters
  So it translates half the alphabet into non-alphabet characters

  shift13 is used because it is simpler than rot13 in c
```

```
  so we can focus on the point

  (c) NVIDIA
  Taras Shapovalov <tshapovalov@nvidia.com>
 */
#include <cuda.h>
#include <stdio.h>

// CUDA kernel definition: undo shift13
__global__ void helloWorld(char* str) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  str[idx] -= 13;
}

void checkCudaError(cudaError_t err, const char *msg) {
  if (err != cudaSuccess) {
    fprintf(stderr, "CUDA error: %s: %s\n", msg, cudaGetErrorString(err));
    exit(EXIT_FAILURE);
  }
}

int main(int argc, char **argv) {
  char s[] = "Hello World!";
  printf("String for encode/decode: %s\n", s);

  // CPU shift13
  int len = sizeof(s);
  for (int i = 0; i < len; i++) {
    s[i] += 13;
  }
  printf("String encoded on CPU as: %s\n", s);

  // Allocate memory on the CUDA device
  char *cuda_s;
  checkCudaError(cudaMalloc((void**)&cuda_s, len), "cudaMalloc");

  // Copy the string to the CUDA device
  checkCudaError(cudaMemcpy(cuda_s, s, len, cudaMemcpyHostToDevice), "cudaMemcpy");

  // Set the grid and block sizes (dim3 is a type)
  // and "Hello World!" is 12 characters, say 3x4
  dim3 dimGrid(3);
  dim3 dimBlock(4);

  // Invoke the kernel to undo shift13 in GPU
  helloWorld<<< dimGrid, dimBlock >>>(cuda_s);
  checkCudaError(cudaGetLastError(), "kernel execution");

  // Retrieve the results from the CUDA device
  checkCudaError(cudaMemcpy(s, cuda_s, len, cudaMemcpyDeviceToHost), "cudaMemcpy");

  // Free up the allocated memory on the CUDA device
  checkCudaError(cudaFree(cuda_s), "cudaFree");

  printf("String decoded on GPU as: %s\n", s);
```

```
  return 0;
}
```

The hello world code example may be compiled and run on a node with a GPU and CUDA 12.1 with:

```
[fred@node001 ~]$ module load shared
[fred@node001 ~]$ module load cuda12.1
[fred@node001 ~]$ nvcc hello.cu -o hellocuda
[fred@node001 ~]$ ./hellocuda
String for encode/decode: Hello World!
String encoded on CPU as: Uryy|-d|yq.
String decoded on GPU as: Hello World!
[fred@node001 ~]$
```

The number of characters displayed in the encoded string are less than the number in the unencoded string. This is because there are unprintable characters generated by the encoding, because the cipher used is not exactly rot13.

To make it run from a head node via a workload manager such as Slurm, on a compute node with a GPU, the following batch file could be built and run:

```
[fred@basecm11 ~]$ cat helloslurm.sh
#!/bin/sh
#SBATCH -o my.stdout
#SBATCH --ntasks-per-node=1
#SBATCH -p defq    #assuming node in defq has a GPU
#SBATCH --gpus=1
module clear -f
./hellocuda
[fred@basecm11 ~]$ sbatch helloslurm.sh
[fred@basecm11 ~]$ cat my.stdout
String for encode/decode: Hello World!
String encoded on CPU as: Uryy|-d|yq.
String decoded on GPU as: Hello World!
```

7.5.5 OpenACC

OpenACC (`http://www.openacc-standard.org`) is a new open parallel programming standard aiming at simplifying the programmability of heterogeneous CPU/GPU computing systems. OpenACC allows parallel programmers to provide *OpenACC directives* to the compiler, identifying which areas of code to accelerate. This frees the programmer from carrying out time-consuming modifications to the original code itself. By pointing out parallelism to the compiler, directives get the compiler to carry out the details of mapping the computation onto the accelerator.

Using OpenACC directives requires a compiler that supports the OpenACC standard.

In the following example, where π is calculated, adding the `#pragma` directive is sufficient for the compiler to produce code for the loop that can run on either the GPU or CPU:

Example

```
#include <stdio.h>
 #define N 1000000

 int main(void) {
 double pi = 0.0f; long i;
 #pragma acc parallel loop reduction(+:pi)
 for (i=0; i<N; i++) {
```

```
    double t= (double)((i+0.5)/N);
    pi +=4.0/(1.0+t*t);
}
printf("pi=%16.15f\n",pi/N);
return 0;
}
```

8

Using Kubernetes

8.1 Introduction To Kubernetes Running Via NVIDIA Base Command Manager

Kubernetes is a system for managing containerized applications across multiple hosts in a cluster.

- A container is an extremely lightweight virtualized operating system that runs without the unneeded extra emulated hardware components of a regular virtualized operating system.
- A containerized application runs within a container, and it only accesses files, environment variables, and libraries within the container, unless volumes are mounted and used.
- A containerized application provides services to other software or users. Kubernetes thus manages containerized applications as a service, and is aware of the container states and resources used.

Kubernetes provides mechanisms for application deployment, scheduling, updating, maintenance, and scaling. It actively manages the containers to ensure that the state of the cluster continually matches the user's intentions. The user's desired state is communicated to the Kubernetes API server, typically in the form of a YAML file. Kubernetes stores the YAML file in Etcd, and ensures it is reflected by the current state of the containers.

This chapter describes how Kubernetes works with BCM, which currently supports Kubernetes 1.32. For details on Kubernetes that are outside the scope of its use with BCM, the official Kubernetes documentation at `https://kubernetes.io/docs/` can be consulted.

By default, in BCM the user is given access to containers only via Kubernetes. The administrator can however configure direct access if required. In this chapter, only container access via Kubernetes is described.

The `kubectl` utility is normally used to communicate with Kubernetes, although using the API directly instead of using `kubectl` is also possible. The `kubectl` utility can be used to get information about Kubernetes runtime, creation and management of *resources*, as well for other tasks. Resources are items such as pods (`https://kubernetes.io/docs/concepts/workloads/pods/`) and volumes (`https://kubernetes.io/docs/concepts/storage/volumes/`), that are consumed while containers are in use.

The official Kubernetes documentation has some introductory tutorials, including:
`https://kubernetes.io/docs/tutorials/kubernetes-basics/`

Familiarity with the concepts in those tutorials is recommended before continuing with the rest of this chapter.

8.2 Kubernetes User Privileges

The privileges (view-only, edit or admin) that a user has depends on the permissions granted to the user by the Kubernetes Administrator. Since BCM version 9.0 (known earlier as Bright Cluster manager 9.0), a custom namespace is created for each user, <*user*>, according to the following format:

<*user*>-restricted

This is intended to be a sandbox for the user.

The default setting in NVIDIA Base Command Manager version 9.0 is to have the PodSecurityPolicies (PSP) feature inactive. Users can check with their BCM administrator if that is indeed the case. If PSP is not active, then users have access to the `default` namespace.

In future versions of BCM, the `default` namespace will no longer be available for users by default.

If the Administrator enables the PSP features, then permissions to the `default` namespace are also not added by default for new users. Role bindings to the `default` namespace for existing users are also removed.

This "restricted" namespace is always created by default, and users are encouraged to use this namespace.

8.3 Kubernetes Quickstarts

This section provides quickstart recipes on some common tasks for an end user using Kubernetes.

Requirements:

- A Kubernetes cluster on BCM. The administrator should have set this up as described in Chapter 4 of the *Containerization Manual*.
- a dedicated user on the Kubernetes cluster

The simplest way for a user to use a BCM-managed Kubernetes cluster is if the cluster administrator has configured it for use via the head node. A user can then simply connect to the Kubernetes cluster via an ssh connection to the head node, and then load the environment via `module load kubernetes`. The remainder of this section can then be skipped.

An alternative way for a user to use a BCM-managed Kubernetes cluster avoids going via the head node, and uses a local PC.

8.3.1 Quickstart: Accessing The Kubernetes Dashboard

The Kubernetes Dashboard is a web browser-based way to manage Kubernetes tasks.

Requirements:

- from the local PC it should be possible to access the Kubernetes Dashboard URL at:
  `https://dashboard.`<*kubernetes cluster name*>`:30443`

  If that does not work, then the cluster administrator has probably modified the standard configuration, and should be consulted on how to access the Kubernetes Dashboard.
- If the Dashboard is accessible, then the user needs a token to authenticate with the Kubernetes Dashboard.

Obtaining And Using The Token

A user `test` is assumed to have been created according to the procedure in section 4.11 of the *Containerization Manual*. For the user `test`, a token can be obtained from the head node with the following procedure, using the namespace and username of the user:

Example

```
[test@basecm11 ~]$ module load kubernetes
[test@basecm11 ~]$ USERNAME=test
[test@basecm11 ~]$ NS=${USERNAME}-restricted
[test@basecm11 ~]$ kubectl create token -n $NS $USERNAME
eyJhbGciOiJSUzI1NiIsImtpZCI6InZMcThEdlFucXo3cGZ3Yk5zQnB1VDdxTlRYVnpMW192M2VVaDJGTU9CQlUifQ.\
eyJhdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWwiXSwiZXhwIjoxNzA2NTM\
3ODAwLCJpYXQiOjE3MDY1MzQyMDAsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGVyLm\
```

xvY2FsIiwia3ViZXJuZXRlcy5pbyI6eyJuYW1lc3BhY2UiOiJhbGljZS1yZXN0cmljdGVkIiwic2VydmljZWFjY291b\
nQiOnsibmFtZSI6ImFsaWNlIiwidWlkIjoiODQ1NTYyNmMtMGM4OC00YzgOLTliZDEtZjdlYTEyNWI2MjNlIn19LCJu\
YmYiOjE3MDY1MzQyMDAsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDphbGljZS1yZXN0cmljdGVkOmFsaWNlIn0\
.kq5qXkNuh3pApYorNf2JCPfl4lQ8LS6oHJ4z73AWMD0Zmv-_J13L5YvSP2dBBY8-dUzxDY-SyPLjK8gT20sXJhSXY-\
tTEp7PvMme77R82IpWpgPJVQxv_d52btUPBoi6cqwi-14gsv-qOX40uxJr2ITQvurHRY8V1rewmyTgz-JCdvPNoqMtU\
x6Uv61Oh1cwJWrnlqhWYpzHL6YT2OvbV9manVO_-V2I6f5ci-wytFuO-r6goczrunuRDIGn9_ZUVUqe7XH6FEqMueGM\
bbF7AArq1-R52EA9ArzjzyXMRGEj2KdbKr_NXh-09FuxZ4-mGhBuk9-OPyUbHXn8M2WFhQ

If the username and related namespace are not known to the user, then the cluster administrator should be asked to provide these.

The created token can be pasted from the terminal into the web browser, at the place where the token is asked for (figure 8.1):

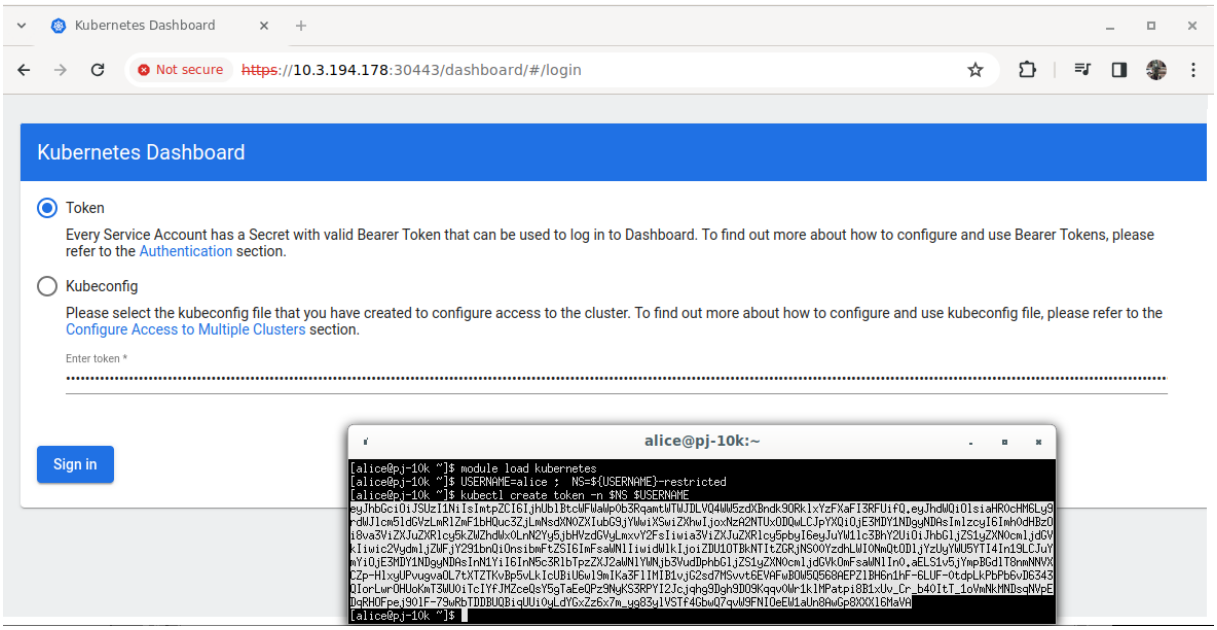

Figure 8.1: Kubernetes User Dashboard Authentication, Using Token Copy-Paste From A Terminal

After a successful authentication, the Dashboard displayed should appear as in figure 8.2:

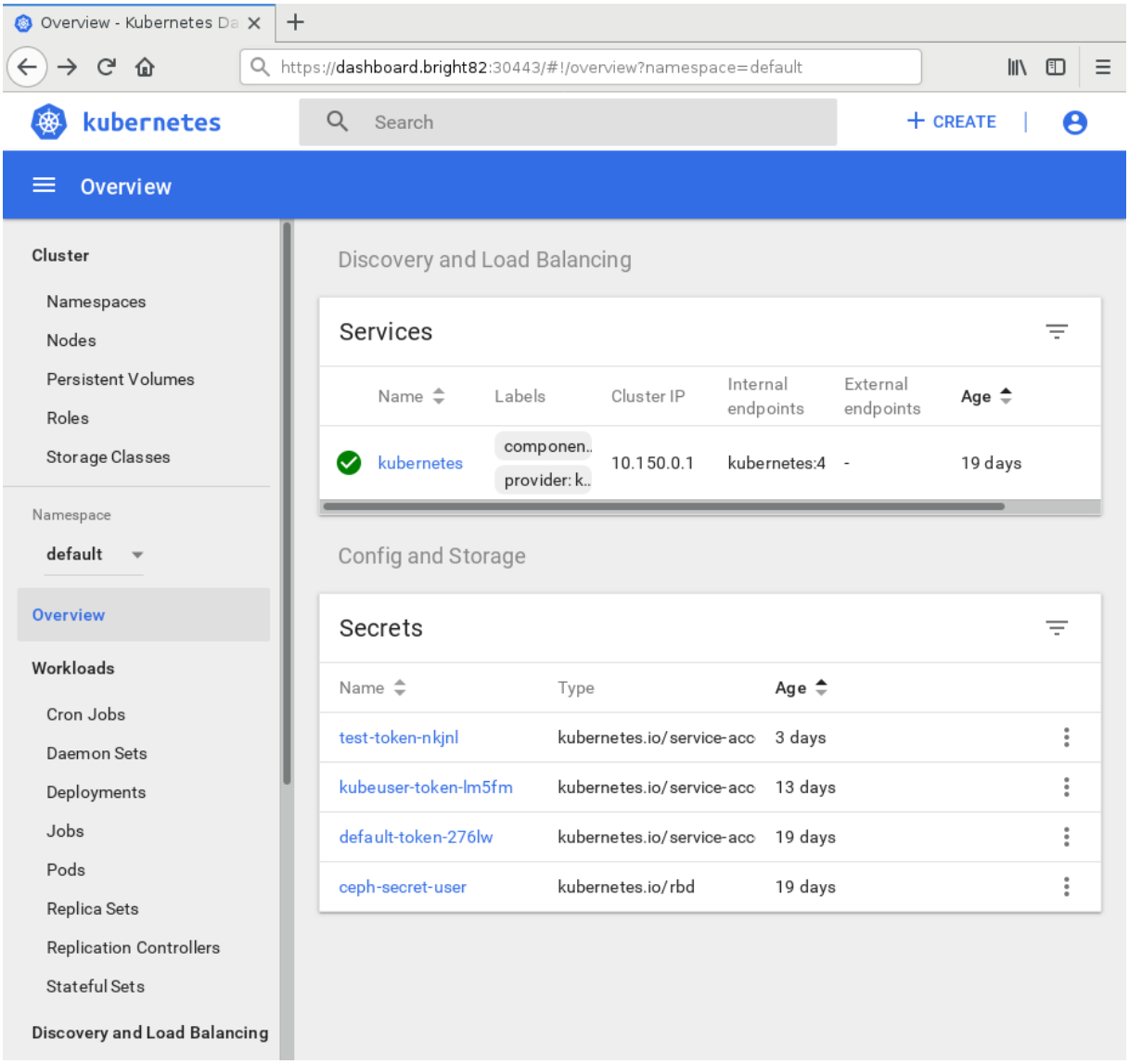

Figure 8.2: Kubernetes User Dashboard After Authentication

The user can now deploy a job using the previously-defined `default` namespace, as in the following example. The task can be submitted as a YAML file, via the Dashboard URL

`https://dashboard.`*<kubernetes cluster name>*`:30443#!/deploy?namespace=default`

The task could be the `pi-job.yml` job of section 8.3.3. The result—π to 4000 places—can be seen in the log associated with the jobs, accessible from the Dashboard URL

`https://dashboard.`*<kubernetes cluster name>*`:30443#!/job?namespace=default`

8.3.2 Quickstart: Using `kubectl` From A Local Machine

The advantage of connecting from a local PC is that there is no need to connect to the head node via SSH.

Requirements:

- the local PC should be able to access the Kubernetes API server at `https://dashboard.`<*kubernetes cluster name*>`:30443`. The URL that is actually used is set up by the cluster administrator, who should be contacted for details.
- The local PC should be Linux-based and run on an amd64 architecture.

Steps:

- On the PC, `kubectl` for Kubernetes 1.32 should be downloaded from the head node <*headnode*>. It can be downloaded to a directory in the user path, such as `/usr/bin`

Example

```
$ rsync <username>@<headnode>:/cm/local/apps/kubernetes/current/bin/kubectl \
<directory in the user path>
```

- The user can make a `.kube` directory on the PC. The Kubernetes configuration for the user <*username*> can then be picked up from <*headnode*>. This includes the keys and the certificates:

```
$ mkdir ~/.kube
$ rsync <username>@<headnode>:.kube/config-<cluster name> ~/.kube/config
```

- <*cluster name*> must be replaced with the fully qualified domain name of the Kubernetes cluster
- The user can check if `kubectl` is able to connect to the cluster by running the following commands:

Example

```
$ kubectl cluster-info
$ kubectl get nodes
$ kubectl get all
```

8.3.3 Quickstart: Submitting Batch Jobs With `kubectl`

A batch job can be created in Kubernetes. It is simply referred to as a "job". It is basically made up of non-persistent pods that run a one-off task.

A simple job to calculate π can be created by building a `pi-job.yml` file with the following content:

Example

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  completions: 8
  parallelism: 1
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(4000)"]
      restartPolicy: Never
```

This carries out a Perl-based calculation of π to 4000 places. Running it from the command line in Bash directly could be done with:

Example

```
$ perl -Mbignum=bpi -wle  "print bpi(4000)"
```

However, the idea here is to demonstrate Kubernetes batch jobs firing up and scaling pods for this task instead, which is described next:

If the administrator has allowed the user access via Kubernetes policies, and has made the user a Kubernetes user, then the job can be submitted with:

```
$ kubectl apply -f pi-job.yml
job "pi" created
```

If the job is horizontally scalable, then the number of replicas can be scaled with:

```
$ kubectl scale job/pi --replicas=4
job "pi" scaled
```

Information about the job can be obtained with (output truncated):

```
$ kubectl get job/pi
NAME      DESIRED   SUCCESSFUL   AGE
pi        8         8            6m
$ kubectl describe job/pi
Name:           pi
Namespace:      default
...
```

The jobs can be followed with (output truncated):

```
$ kubectl get pods -aw
NAME       READY     STATUS      RESTARTS   AGE
pi-74gnd   0/1       Completed   0          6m
...
```

The logs of a pod can be viewed with:

```
$ kubectl logs -f <pod name>
```

An output is shown that starts with:

```
3.14159265358979323846264338327950288419716939937510582097 4...
```

Further information on the following job topics can be found at the associated links:

- job: `https://kubernetes.io/docs/concepts/workloads/controllers/job/`
- job parallelism: `https://kubernetes.io/docs/tasks/job/parallel-processing-expansion/`

8.3.4 Quickstart: Helm, The Kubernetes Package Manager

Helm (`https://docs.helm.sh/`) is a tool for managing *charts*. Charts are packages of pre-configured Kubernetes resources.

Helm is installed and properly configured by default as a Kubernetes add-on. It is initialized for every Kubernetes user when the Kubernetes module is loaded—there is no `helm init` or similar that needs to be carried out first. For example (some text elided):

Example

```
$ module load kubernetes
$ helm version
version.BuildInfo{Version:"v3.18.3", GitCommit:"6838ebcf265a3842d1433956e8a622e3290cf324",\ GitTreeState:"clean",
GoVersion:"go1.24.4"}
```

Choices can be made from among the charts at the official repository at `https://github.com/kubernetes/charts`. For example, GitLab and WordPress can be installed with:

```
$ helm install stable/gitlab --name my-gitlab
$ helm install stable/wordpress --name my-wordpress
```

A tutorial on using Helm is available at `https://docs.helm.sh/using_helm/#using-helm`

9

Spark On Kubernetes

Apache Spark is "a lightning-fast unified analytics engine for big data and machine learning".

Since NVIDIA Base Command Manager version 9.0, the recommended way to run Spark workloads inside BCM is within Kubernetes.

Documentation for Spark is available at `https://spark.apache.org/docs/`.

9.1 Important Requirements

By default only the `root` user of the cluster can access Kubernetes. Since that is not very useful, the cluster administrator can grant access to regular users by using `cm-kubernetes-setup` with the `--add-user` flag.

The regular user should check that Kubernetes can be accessed via the user's account (sections 8.2–8.3), or Spark workloads will fail to run. Being able to load the `kubernetes` module and being able to run the `kubectl` commands suggests that Kubernetes is properly accessible to the user:

Example

```
[test@cluster ~]$ module load kubernetes/default
[test@cluster ~]$ kubectl get all
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.150.0.1   <none>        443/TCP   8h
```

9.2 Running Spark Jobs Via The Kubernetes Spark Operator

Spark jobs can be run within Kubernetes if the cluster administrator has installed the `cm-kubernetes-spark-operator` package. The installation can be done

- as part of `cm-kubernetes-setup` (page 40 of the *Containerization Manual*) or
- as part of the Base View Kubernetes Wizard using the navigation path: `Containers` > `Kubernetes` > `Kubernetes Wizard`.

The following session assumes that the cluster administrator has configured Spark to run as a Kubernetes operator (Chapter 6 of the *Containerization Manual*). In the session, a user `alice` carries out a Spark way of calculating pi. This can be regarded as a "Hello world!" type of demonstration for Spark users.

9.2.1 Example Spark Operator Run: Calculating Pi

For `alice`, a YAML file based on the specification at `https://github.com/GoogleCloudPlatform/spark-on-k8s-operator/blob/master/examples/spark-py-pi.yaml` can be used:

Example

```
[alice@basecm11 ~]$ module load kubernetes
[alice@basecm11 ~]$ cat <<EOF > pi-spark.yaml
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: pyspark-pi
spec:
  type: Python
  pythonVersion: "3"
  mode: cluster
  image: "gcr.io/spark-operator/spark-py:v3.1.1"
  imagePullPolicy: Always
  mainApplicationFile: local:///opt/spark/examples/src/main/python/pi.py
  sparkVersion: "3.1.1"
  restartPolicy:
    type: OnFailure
    onFailureRetries: 3
    onFailureRetryInterval: 10
    onSubmissionFailureRetries: 5
    onSubmissionFailureRetryInterval: 20
  driver:
    cores: 1
    coreLimit: "1200m"
    memory: "512m"
    labels:
      version: 3.1.1
    serviceAccount: spark
  executor:
    cores: 1
    instances: 1
    memory: "512m"
    labels:
      version: 3.1.1
EOF
[alice@basecm11 ~]$ kubectl apply -f pi-spark.yaml
sparkapplication.sparkoperator.k8s.io/pyspark-pi created
[alice@basecm11 ~]$ kubectl get pods
NAME                READY   STATUS              RESTARTS   AGE
pyspark-pi-driver   0/1     ContainerCreating   0          1s
[alice@basecm11 ~]$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
pyspark-pi-driver   1/1     Running   0          3s
[alice@basecm11 ~]$ kubectl get sparkapplications
NAME         AGE
pyspark-pi   7s
[alice@basecm11 ~]$ kubectl get pods
NAME                               READY   STATUS              RESTARTS   AGE
pyspark-pi-driver                  1/1     Running             0          14s
pythonpi-e768128383a881b3-exec-1   0/1     ContainerCreating   0          0s
[alice@basecm11 ~]$ kubectl get pods
NAME                               READY   STATUS        RESTARTS   AGE
pyspark-pi-driver                  0/1     Completed     0          34s
pythonpi-e768128383a881b3-exec-1   0/1     Terminating   0          20s
[alice@basecm11 ~]$ kubectl get pods
NAME                READY   STATUS      RESTARTS   AGE
```

```
pyspark-pi-driver   0/1     Completed   0          36s
```

Instead of tracking the pod with:
`kubectl get pods`
as in the preceding session, or with the more convenient:
`watch kubectl get pods`
the pod could be tracked with the `-f| --follow` option to stream the driver logs:

Example

```
[alice@basecm11 ~]$ kubectl logs pyspark-pi-driver -f
```

To get intended output of the pi run—the calculated value of pi—it is sufficient to grep the log as follows:

Example

```
[alice@basecm11 ~]$ kubectl logs pyspark-pi-driver | grep ^Pi
Pi is roughly 3.148800
```

After the pi run has completed, the resources can be removed from the namespace:

```
[alice@basecm11 ~]$ kubectl delete -f pi-spark.yaml
sparkapplication.sparkoperator.k8s.io "pyspark-pi" deleted

[alice@basecm11 ~]$ kubectl get pods
No resources found in alice-restricted namespace.
[alice@basecm11 ~]$ kubectl get sparkapplications
No resources found in alice-restricted namespace.
```

9.3 Running Spark Jobs Directly Via `spark-submit`

The Spark documentation covers a pi run via `spark-submit`. The BCM knowledge base explains how to use `spark-submit` to run a job with older BCM versions (`https://kb.brightcomputing.com/knowledge-base/how-to-deploy-spark-with-kubernetes-on-bright-9-0-9-1-9-2/`). However, using the Kubernetes operator submission method (section 9.2) is recommended instead in more recent versions.

9.4 Accessing The Spark User Interface

If a job is run via the Kubernetes Operator or `spark-submit`, then the Spark User Interface (Spark UI) can be used to monitor the job. The Spark UI is shut down once the job has finished, so it only makes sense to access the Spark UI for longer-running jobs.

The `kubectl port-forward` command can be used to allow access to the Spark UI:

```
[test@cluster ~]$ module load kubernetes
[test@cluster ~]$ kubectl get pod
NAME                              READY   STATUS      RESTARTS   AGE
spark-pi-1540840666830-driver     0/1     Completed   0          4m34s
spark-pi-1540840938874-driver     0/1     Running     0          2s
[test@cluster ~]$ kubectl port-forward spark-pi-1540840938874-driver 3000:4040
Forwarding from 127.0.0.1:3000 -> 4040
Forwarding from [::1]:3000 -> 4040
Handling connection for 3000
```

The preceding example makes the dashboard available via local port 3000 on the machine where the `port-forward` command is executed. The Spark UI runs on port 4040 inside the pod, and displays something like in figure 9.1:

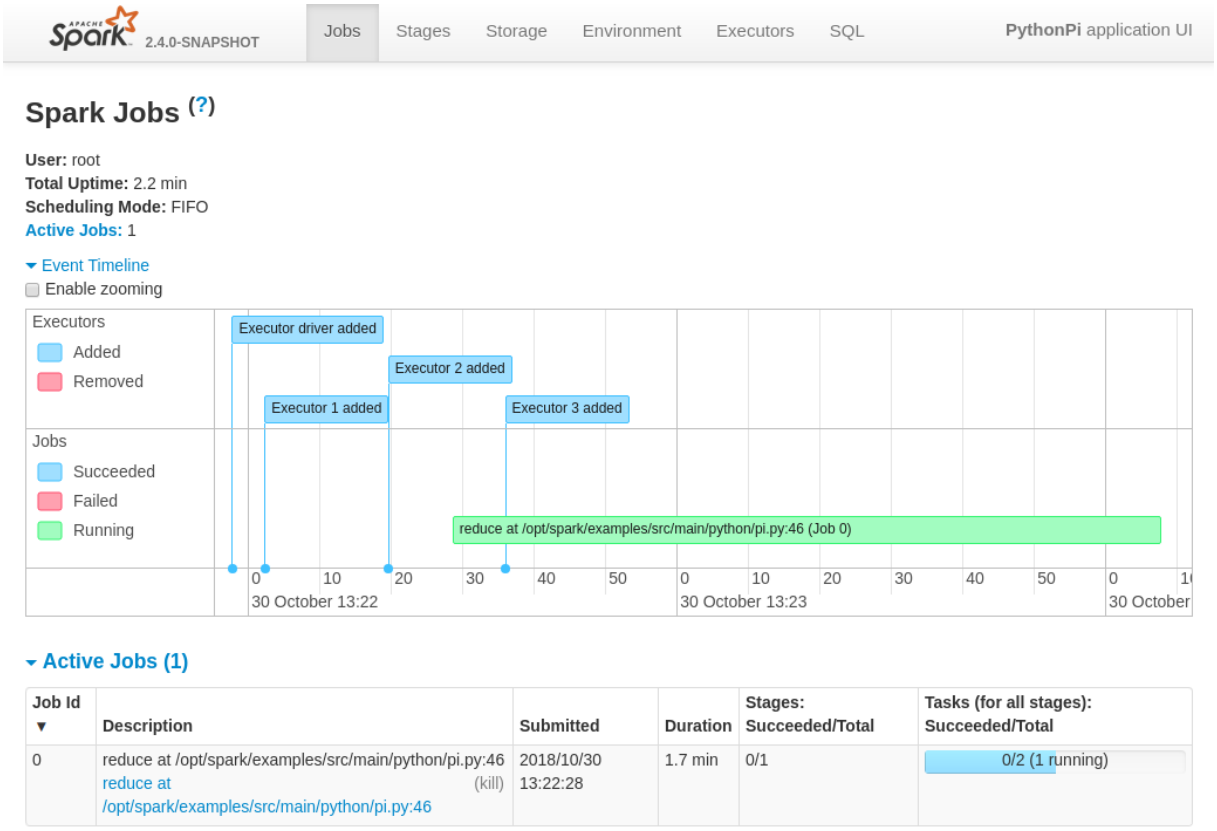

Figure 9.1: The Spark dashboard, forwarded on port 3000

9.5 Mounting Volumes Into Containers

The official documentation at `https://spark.apache.org/docs/latest/running-on-kubernetes.html#using-kubernetes-volumes` gives an outline of how volumes can be mounted into containers.

In summary, there are 3 ways a Kubernetes volume (`https://kubernetes.io/docs/concepts/storage/volumes`) can be mounted:

1. `hostPath`: mounts a file or directory from the host node's filesystem into a pod.
2. `emptyDir`: an initially empty volume, created when a pod is assigned to a node.
3. `persistentVolumeClaim`: used to mount a `PersistentVolume` into a pod.

Mounting A `hostPath`

The official documentation mentions adding the following two flags to specify how the volume is mounted inside the container:

```
--conf spark.kubernetes.driver.volumes.[VolumeType].[VolumeName].mount.path=<mount path>
--conf spark.kubernetes.driver.volumes.[VolumeType].[VolumeName].mount.readOnly=<true|false>
```

The following flag should also be added, to specify the path on the host:

```
--conf spark.kubernetes.driver.volumes.[VolumeType].[VolumeName].options.path=<mount path>
```

The volumes can be specified for each *executor* as well as for the *driver*. For an executor, the specification should use `spark.kubernetes.executor` instead of `spark.kubernetes.driver`.

For example, the NFS share `/cm/shared` of a typical BCM setup could be made available on executor pods by adding the following 3 flags to the `spark-submit` command:

Example

```
module load kubernetes/default
module load spark
spark-submit \
    --master k8s://https://localhost:10443 \
    --deploy-mode cluster \
    --name spark-pi \
    --class org.apache.spark.examples.SparkPi \
    --conf spark.kubernetes.namespace=default \
    --conf spark.executor.instances=2 \
    --conf spark.kubernetes.container.image=docker.io/brightcomputing/spark:2.4.0 \
    --conf spark.kubernetes.authenticate.driver.serviceAccountName=spark \
    --conf spark.kubernetes.executor.volumes.hostPath.cmshared.options.path=/cm/shared \
    --conf spark.kubernetes.executor.volumes.hostPath.cmshared.mount.path=/data \
    --conf spark.kubernetes.executor.volumes.hostPath.cmshared.mount.readOnly=false \
    local:///opt/spark/examples/jars/spark-examples_2.11-2.4.0.jar  10000
```

This results in `/cm/shared` being mounted in `read-write` mode on the mount path `/data`:

```
[root@cluster ~]# kubectl get pod
NAME                              READY   STATUS      RESTARTS   AGE
spark-pi-1552563523319-driver     0/1     Completed   0          23h
spark-pi-1552648768369-driver     0/1     Completed   0          5m40s
spark-pi-1552648961734-driver     0/1     Completed   0          2m26s
spark-pi-1552649048516-driver     1/1     Running     0          59s
spark-pi-1552649048516-exec-1     1/1     Running     0          53s
spark-pi-1552649048516-exec-2     1/1     Running     0          53s
[root@cluster ~]# kubectl exec -it spark-pi-1552649048516-exec-1 /bin/bash
bash-4.4# mount | grep shared
master:/cm/shared on /data type nfs (rw,relatime,vers=3,...)
bash-4.4# ls -l /data
total 12
drwxr-xr-x   39 root     root          4096 Mar 13 10:11 apps
drwxr-xr-x   13 root     root           162 Mar  3 21:22 docs
drwxr-xr-x    3 root     root            43 Mar  3 21:22 examples
-rw-r--r--    1 root     root           101 Mar 14 11:14 init.sh
drwxr-xr-x    3 root     root            16 Mar  3 21:29 licenses
drwxr-xr-x   28 root     root          4096 Mar 12 13:31 modulefiles
bash-4.4#
```

Using Persistent Volume Claims

Spark in this case assumes that the Kubernetes cluster being managed by the user has persistent volumes available. A list of persistent volumes types can be found at: `https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes`.

Claims to specific types of storage can be created for use with Spark. If `spark-submit` tries to use a persistent volume claim, then it assumes the claim already exists. It does not initiate a claim by itself.

In the following specification, for demonstration purposes, a claim `my-claim` is created:

```
apiVersion: v1
kind: PersistentVolumeClaim
```

```
metadata:
  name: my-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1000Gi
```

This claim should bind to a persistent volume that meets the criteria. If none are configured, then the user can create a local volume for `/cm/shared`, for example with:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: kube-pv-volume
  labels:
    type: local
spec:
  capacity:
    storage: 1000Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /cm/shared
```

Applying the above two YAML configurations should result in the persistent volume claim object `my-claim`, and the persistent volume `kube-pv-volume` in Kubernetes:

```
[root@cluster ~]# kubectl get pv
NAME            CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  CLAIM             STORAGECLASS
kube-pv-volume  1000Gi    RWO           Retain          Bound   default/my-claim

[root@cluster ~]# kubectl get pvc
NAME       STATUS   VOLUME           CAPACITY   ACCESS MODES   STORAGECLASS   AGE
my-claim   Bound    kube-pv-volume   1000Gi     RWO                           5m35s
```

Example

Invoking `spark-submit` as follows will then try to find the claim with name `my-claim`.

```
module load kubernetes/default
module load spark
spark-submit \
    --master k8s://https://localhost:10443 \
    --deploy-mode cluster \
    --name spark-pi \
    --class org.apache.spark.examples.SparkPi \
    --conf spark.kubernetes.namespace=default \
    --conf spark.executor.instances=2 \
    --conf spark.kubernetes.container.image=docker.io/brightcomputing/spark:2.4.0 \
    --conf spark.kubernetes.authenticate.driver.serviceAccountName=spark \
    --conf \
    spark.kubernetes.executor.volumes.persistentVolumeClaim.cmshared.options.claimName=my-claim \
    --conf spark.kubernetes.executor.volumes.persistentVolumeClaim.cmshared.mount.path=/data \
    --conf spark.kubernetes.executor.volumes.persistentVolumeClaim.cmshared.mount.readOnly=false \
    local:///opt/spark/examples/jars/spark-examples_2.11-2.4.0.jar  10000
```

```
[root@cluster ~]# kubectl get pod
NAME                              READY   STATUS      RESTARTS   AGE
spark-pi-1552563523319-driver     0/1     Completed   0          24h
spark-pi-1552648768369-driver     0/1     Completed   0          22m
spark-pi-1552648961734-driver     0/1     Completed   0          18m
spark-pi-1552649048516-driver     0/1     Completed   0          17m
spark-pi-1552649606812-driver     0/1     Completed   0          8m9s
spark-pi-1552650084516-driver     1/1     Running     0          11s
spark-pi-1552650084516-exec-1     1/1     Running     0          4s
spark-pi-1552650084516-exec-2     1/1     Running     0          3s
```

The claim that was specified can be found in the pod description:

```
[root@cluster ~]# kubectl describe pod spark-pi-1552650084516-exec-1 | grep my-claim -C 2
  cmshared:
    Type:      PersistentVolumeClaim (reference to a PersistentVolumeClaim in the same namespace)
    ClaimName: my-claim
    ReadOnly:  false
  default-token-rvrf8:
```

Inside the executor pods `/data` is then found to be available, and presents the contents of `/cm/shared` from the host OS.

10

User Portal

The user portal allows users to log in via a browser and view the state of the cluster themselves. The interface does not allow administration, but presents data about the system. The presentation of the data can be adjusted in many cases.

The user portal is accessible at a URL with the format of `https://`*<head node host name, or IP address>*`:8081/userportal`, unless the administrator has changed it.

The first time a browser is used to log in to the cluster portal, a warning about the site certificate being untrusted appears in a default NVIDIA Base Command Manager configuration. This can safely be accepted.

The user portal has several modes.

- The `Overview` mode () is opened by default, and allows a user to access the following pages via links in the left hand column:
  - Overview (section 10.1)
  - Workload (section 10.2)
  - Nodes (section 10.3)
  - Kubernetes (section 10.4)
- The `Monitoring` mode (section 10.5) () allows a user to plot device measurables
- The `Accounting and reporting` mode (section 10.6) () allows a user to plot job-based resource consumption.

10.1 Overview Page

The default Overview page allows a quick glance to convey the most important cluster-related information for users (figure 10.1):

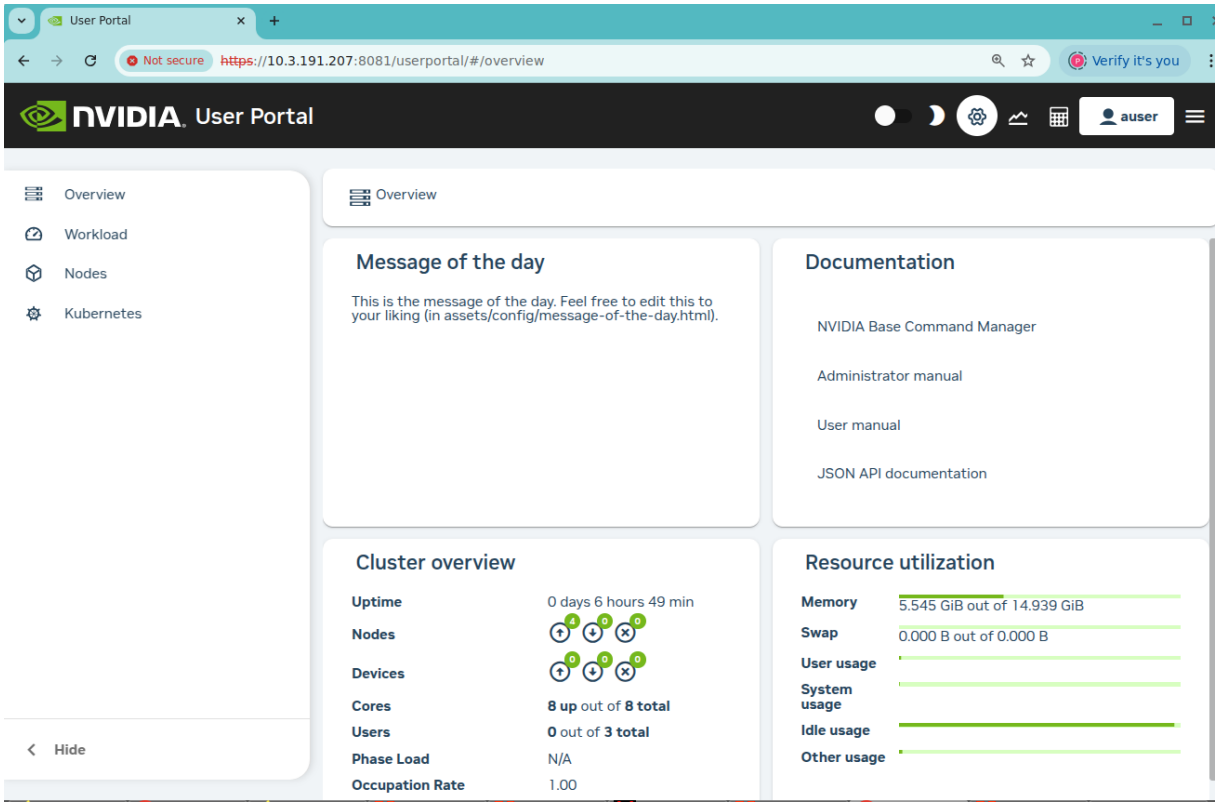

Figure 10.1: User Portal: Overview Page

The following items are displayed on a default home page:

- a Message Of The Day. The administrator may put up important messages for users here
- links to the documentation for the cluster
- contact information. This typically shows how to contact technical support
- an overview of the cluster state, displaying some cluster parameters

10.2 Workload Page

The `Workload` page allows a user to see workload-related information for the cluster (figure 10.2). The columns are sortable.

By default, only the cluster administrator can see information for all users. The administrator can adjust the profile for a user to allow that user to view information from other users.

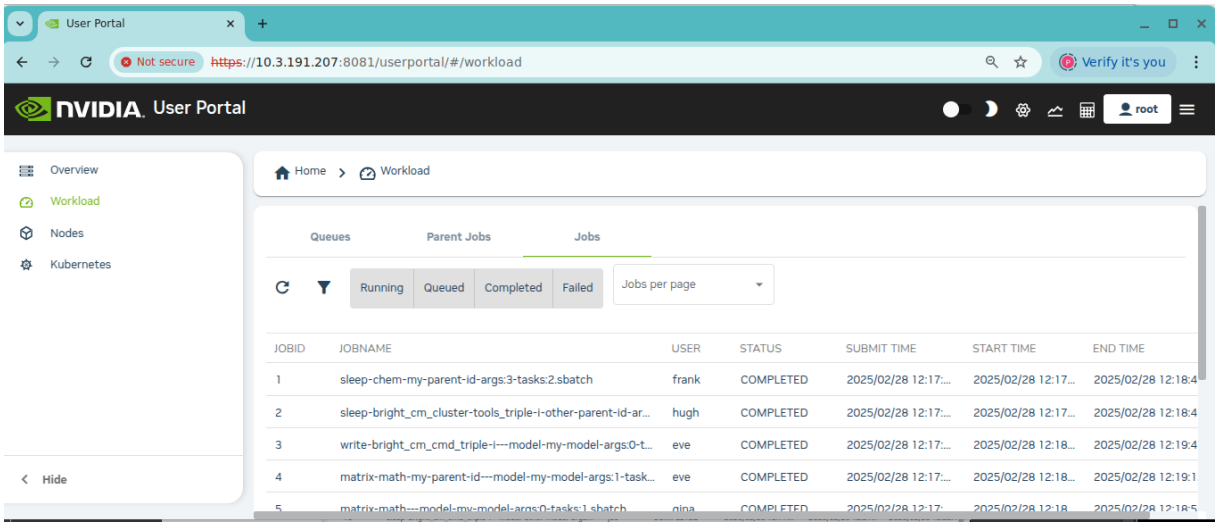

Figure 10.2: User Portal: Workload Page

The workload jobs are organized in tables according to:

- `Queues`
- `Parent Jobs`
- `Jobs`

10.3 Nodes Page

The Nodes page shows nodes on the cluster (figure 10.3), along with some of their properties. Nodes and their properties are arranged in sortable columns.

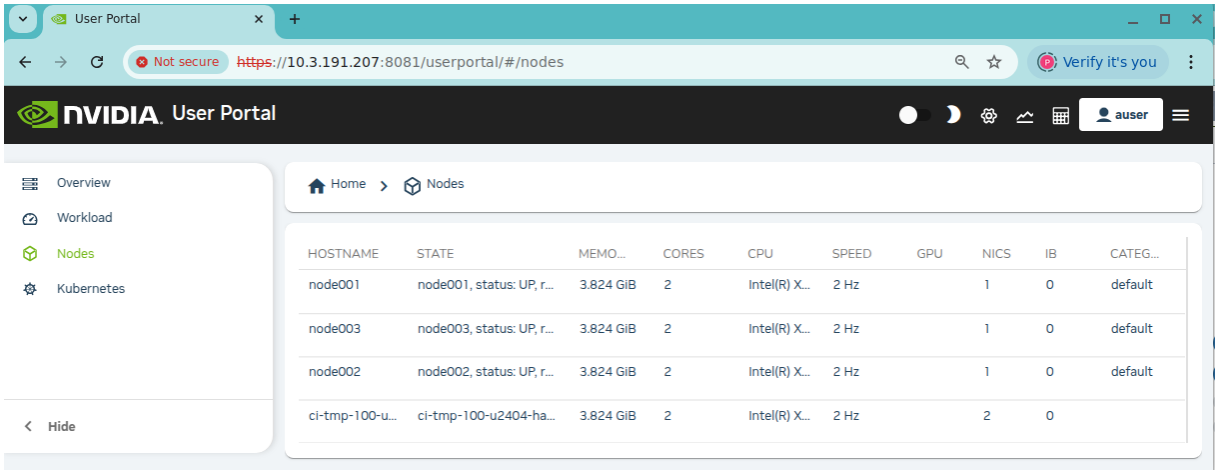

Figure 10.3: User Portal: Nodes Page

The following information about the head or regular nodes is presented:

- `HOSTNAME`: the node name
- `STATE`: For example, `UP`, `DOWN`, `INSTALLING`, along with some other information
- `MEMORY`: RAM on the node

- CORES: Number of cores on the node
- CPU: Type of CPU, for example, Dual-Core AMD Opteron™
- SPEED: Processor speed
- GPU: GPUs on the node, if any
- NICS: Number of network interface cards on the node, if any
- IB: Number of InfiniBand interconnects on the node, if any
- CATEGORY: The node category that the node has been allocated by the administrator (by default it is default)

10.4 Kubernetes Page

The Kubernetes page (figure 10.4) shows an overview of the resources available in clusters running Kubernetes.

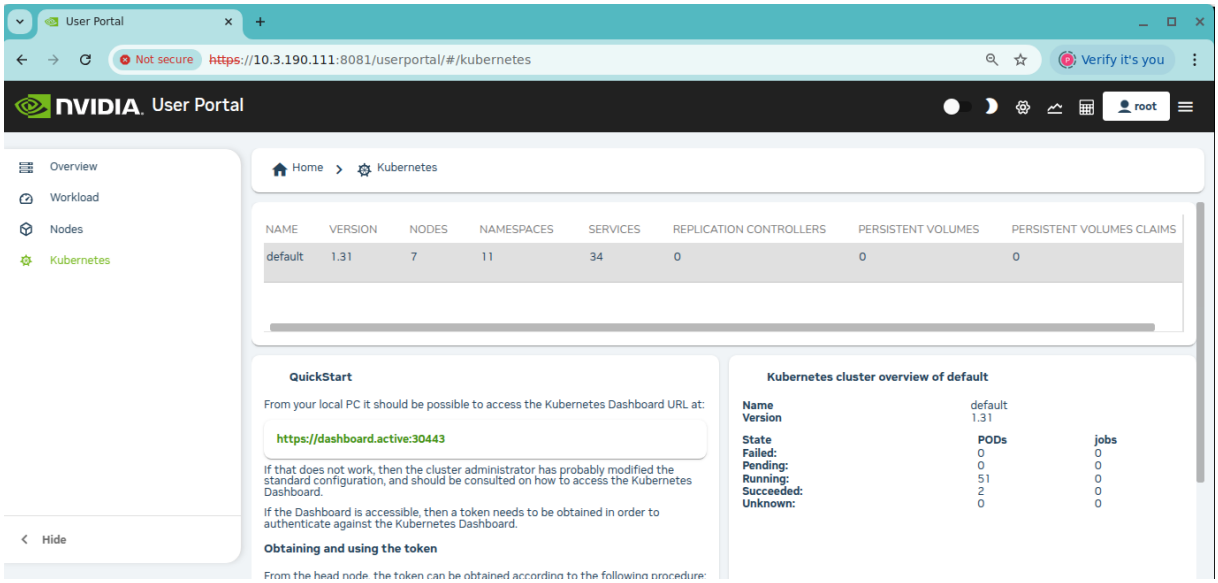

Figure 10.4: User Portal: Kubernetes Page

The Kubernetes cluster is subset of a BCM cluster, and is the part of the BCM cluster that runs and controls pods. The items shown are:

- NAME: The Kubernetes Cluster name
- VERSION: The Kubernetes version
- NODES: The number of nodes in the Kubernetes cluster
- NAMESPACES: The number of namespaces defined for the Kubernetes cluster
- SERVICES: The number of services that are served by the Kubernetes cluster
- REPLICATION CONTROLLERS: The number of replication controllers that run on the Kubernetes cluster
- PERSISTENT VOLUMES: The number of persistent volumes created for the pods of the Kubernetes cluster

- `PERSISTENT VOLUMES CLAIMS`: The number of persistent volumes claims created on the Kubernetes cluster

Some notes are also presented for the user on how to access the Kubernetes Dashboard.

10.5 Monitoring Mode

By default the `Monitoring` mode page displays two empty plot panels within the dashboard section of the page.

The panels can have measurables drag-and-dropped into them from the measurables navigation tree on the left hand side of the page (figure 10.5). The tree can be partly or fully expanded.

A filter can be used to select from visible measurables. For example, after expanding the tree, it is possible to find a measurable related to the cluster occupation rate (Appendix G of the *Administrator Manual*) by using the key word "occupation" in the filter. The measurable can then be dragged from the options that remain visible.

Extra plot panel widgets can be added to the page by clicking on the `Add new widget` option (the ⊕ button) in the panels section of the dashboard page.

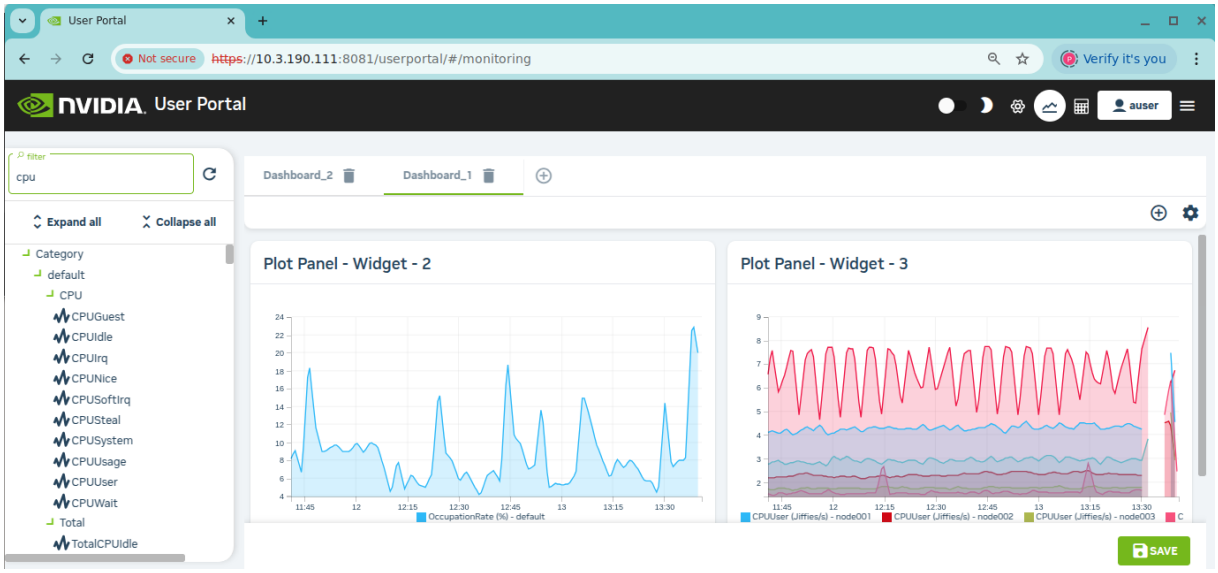

Figure 10.5: User Portal: Monitoring Page Plots

A new dashboard tab can be added to the page by clicking on the `Add new dashboard` option (the ⊕ button) in the dashboard tabs section of the page.

10.6 Accounting And Reporting Mode

The `Accounting and reporting` mode page allows resource use to be displayed and reported by running PromQL queries. Further background on this can be found in Chapter 12 of the *Administrator Manual*.

Resource reports can be created per user and per account by running selected PromQL queries (figure 10.6). The reports can be gathered in dashboard tabs.

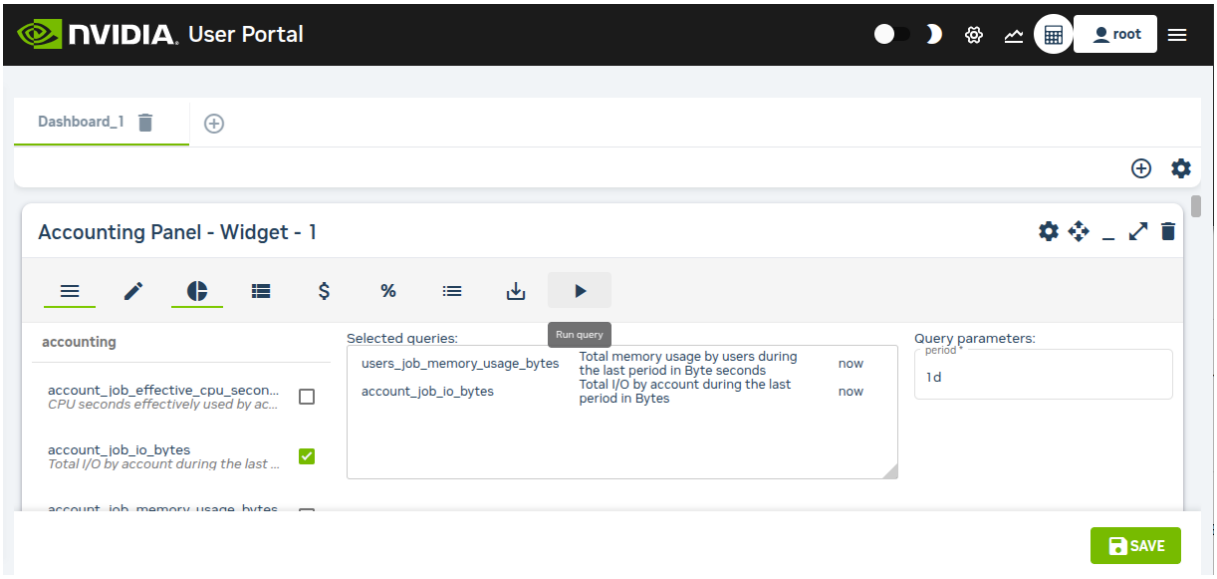

Figure 10.6: User Portal: Dashboard Tabs

The browser configuration that generates the reports can be saved, and the report data values can be displayed in the browser in a table for instant queries and range queries, as well as displayed as a pie chart for instant queries (figure 10.7). The data values can also be exported to CSV or Microsoft Excel format and downloaded.

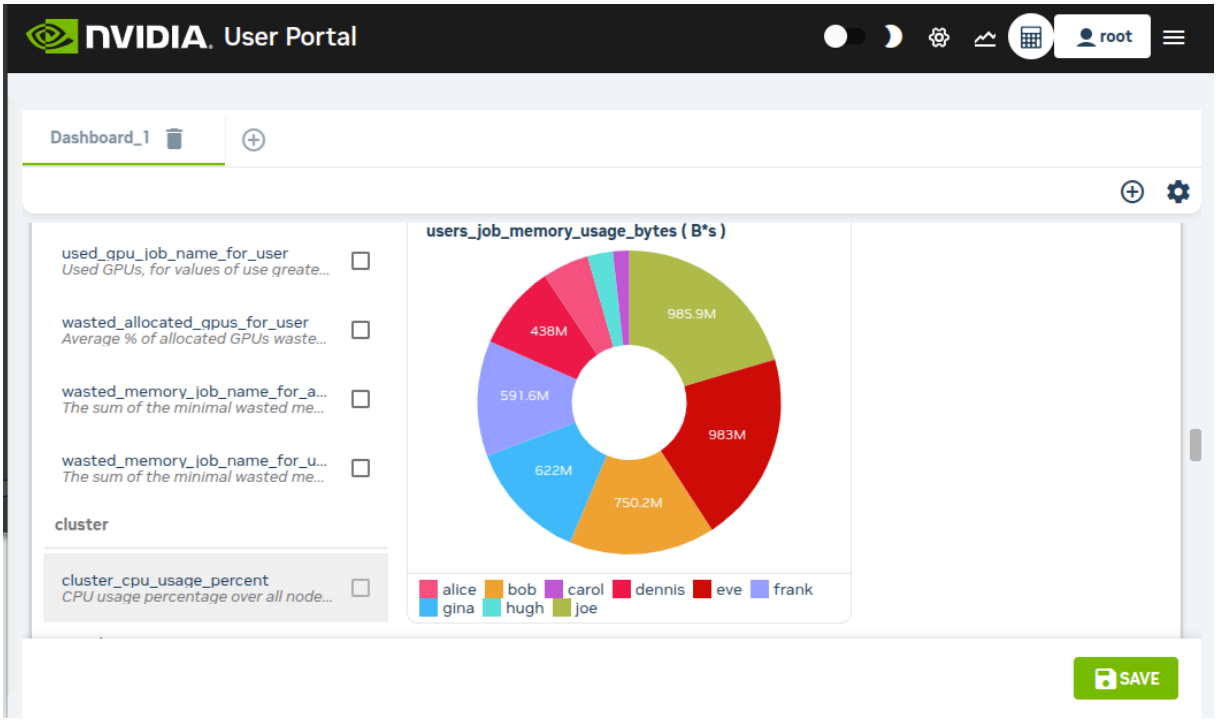

Figure 10.7: User Portal: Simple Accounts

11

Using Jupyter

11.1 Introduction

Jupyter is a general name given to a system of software components integrated with Jupyter Notebook.

Chapter 16 of the *Administrator Manual* describes Jupyter for NVIDIA Base Command Manager, but is meant for cluster administrators rather than end users.

For the orientation of end users, excerpts from that chapter that give a short overview of the Jupyter components are repeated in this introductory section here. How a user can use the components is described in greater detail in the remaining sections of this chapter.

What Is Jupyter Notebook?

Jupyter Notebook (`https://jupyter-notebook.readthedocs.io/`), or Jupyter, is a client-server open-source application that provides a convenient way for a cluster user to write and execute *notebook documents* in an interactive environment.

In Jupyter, a notebook document, or notebook, is content that can be managed by the application. Notebooks are organized in units called *cells* and can contain both executable code, as well as items that are not meant for execution.

Items not meant for execution can be, for example: explanatory text, figures, formulas, or tables. Notebooks can also store the inputs and outputs of an interactive session.

Notebooks can thus serve as a complete record of a user session, interleaving code with rich representations of resulting objects.

These documents are encoded as JSON files and saved with the `.ipynb` extension. Since JSON is a plain text format, notebooks can be version-controlled, shared with other users and exported to other formats, such as HTML, LaTeX, PDF, and slide shows.

What Is A Notebook Kernel?

A *notebook kernel* (often shortened to *kernel*) is a computational engine that handles the various types of requests in a notebook (e.g. code execution, code completions, inspection) and provides replies to the user (`https://jupyter.readthedocs.io/en/latest/projects/kernels.html`). Usually kernels only allow execution of a single language. There are kernels available for many languages, of varying quality and features.

What Is JupyterHub?

Jupyter on its own provides a single user service. *JupyterHub* (`https://jupyterhub.readthedocs.io/`) allows Jupyter to provide a multi-user service, and is therefore commonly installed with it. JupyterHub is an open-source project that supports a number of authentication protocols, and can be configured in order to provide access to a subset of users.

What Is JupyterLab?

JupyterLab (`https://jupyterlab.readthedocs.io/`) is a modern and powerful interface for Jupyter. It enables users to work with notebooks and other applications, such as terminals or file browsers. It is open-source, flexible, integrated, and extensible.

JupyterLab works out of the box with JupyterHub. It can be used to arrange the user interface to support a wide range of workflows in data science, scientific computing, and machine learning.

JupyterLab is extensible with plugins that can customize or enhance any part of the interface. Plugins exist for themes, file editors, keyboard shortcuts, as well as for other components.

What Is A Jupyter Extension?

Several components of the Jupyter environment can be customized in different ways with extensions. Some types of extensions are:

- IPython extensions (`https://ipython.readthedocs.io/en/stable/config/extensions/#ipython-extensions`)
- Jupyter Notebook server extensions (`https://jupyter-notebook.readthedocs.io/en/stable/extending/index.html`)
- JupyterLab extensions (`https://jupyterlab.readthedocs.io/en/stable/user/extensions.html`)

Extensions are usually developed, bundled, released, installed, and enabled in different ways.

Each extension provides a new functionality for a specific component. For example, JupyterLab extensions can customize or enhance any part of the JupyterLab user interface. Extensions can provide new themes, file viewers, editors and renderers for rich output in notebooks. They can also add settings, add keyboard shortcuts, or add items to the menu or command palette.

What Is Jupyter Kernel Provisionning?

By default, Jupyter runs kernels locally, which can exhaust server resources. A resource manager, such as a workload manager (Slurm, PBS, LSF) or Kubernetes, can be used to deal with this issue.

Jupyter Kernel Provisioning (`https://jupyter-client.readthedocs.io/en/latest/provisioning.html#kernel-provisioning`) provides a pluggable interface to distribute kernels across the compute cluster, and uses local underlying resource managers.

The Jupyter Kernel Provisioning framework provides scalability, an improved multi-user support, and a more granular security for Jupyter, in comparison with Jupyter Enterprise Gateway.

In BCM, all the technologies mentioned in these sections are combined to provide a powerful, customizable and user-friendly JupyterLab web interface running on a lightweight, multi-tenant, multi-language, scalable and secure environment, ready for a wide range of enterprise scenarios.

For convenience, in the following sections, *Jupyter* is generally used to collectively refer to Jupyter Notebook, JupyterHub and JupyterLab

BCM Jupyter Extensions

For a default deployment of Jupyter, BCM installs and enables the following extensions to the Jupyter environment:

- Jupyter Addons: A Jupyter Notebook server extension that performs API calls to CMDaemon and manages other server extensions;
- Jupyter Kernel Provisioning modules: A set of modules created to handle Jupyter kernels' lifecycles in different possible BCM configurations. The modules available are: Slurm, PBS, LSF, Kubernetes.
- Jupyter Kernel Creator (section 11.4): A Jupyter Notebook server extension that provides a new interactive and user-friendly way to create kernels;

- Jupyter VNC (section 11.7): A Jupyter Notebook server extension that enables remote desktops with VNC from notebooks;
- JupyterLab Tools: A JupyterLab extension that exposes BCM server extensions functionalities to the users and shows the Cluster View section;
- Jupyter WLM Magic (section 11.8): An IPython extension that simplifies scheduling of workload manager jobs from the notebook;
- Jupyter Kubernetes Operators Manager (section 16.9 of the *Administrator Manual*): An extension that integrates with Kubernetes clusters, and for which it provides basic overview and management features.

The default setup for the user is described in the next sections. It may however be the case that the cluster administrator has customized Jupyter for the needs of the organization, in which case the description that follows may differ from the real life situation.

For example, if the Jupyter login host is at an IP address of `10.2.75.147` then the URL the user uses to log in from with a browser (figure 11.1) is by default:

```
https://10.2.75.147:8000
```

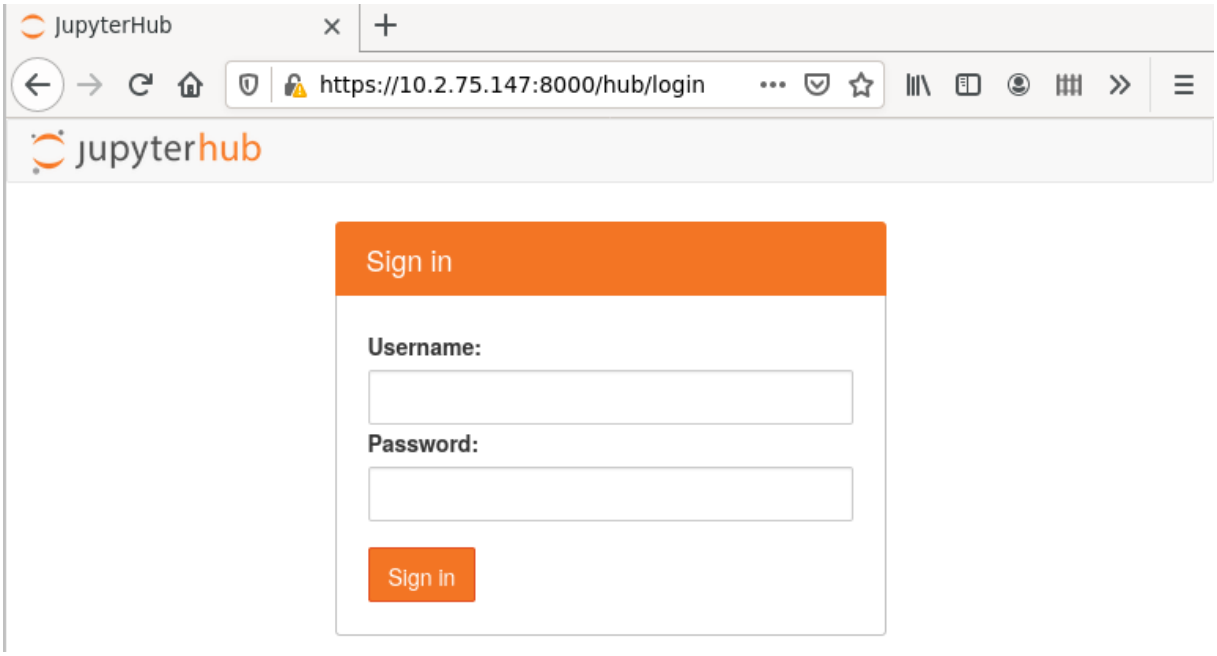

Figure 11.1: JupyterHub User Login Screen

11.2 Jupyter Notebook Examples

The default BCM implementation of Jupyter provides a number of machine learning notebook examples that can be executed with Jupyter.

The notebooks include some applications developed with TensorFlow, PyTorch, MXNet, and other frameworks. The applications can be found in the `/cm/shared/examples/jupyter/notebooks/` directory:

```
[jupyterhubuser@basecm11 ~]$ ls /cm/shared/examples/jupyter/notebooks/
Keras+TensorFlow2-addition.ipynb  psql-example.ipynb      Spark+XGBoost-mortgage.ipynb
llm-bcm-manuals-rag               Pytorch-cartpole.ipynb  TensorFlow-minigo.ipynb
llm-codellama-local               R-iris.ipynb
```

```
MXNet-superresolution.ipynb       Spark-pipeline.ipynb
```

The datasets needed to execute these notebooks can be found in the `/cm/shared/examples/jupyter/datasets/` directory:

```
[jupyterhubuser@basecm11 ~]$ ls /cm/shared/examples/jupyter/datasets/
880f8b8a6fd-mortgage-small.tar.gz  kaggle-iris.csv
```

Users can copy these examples to their home directories, create or choose appropriate kernels to execute them, and interactively run them from Jupyter. In order to edit notebooks, the write permissions must be kept during the copy.

It is the responsibility of users to make sure that the required modules are loaded by their Jupyter kernels. The list of frameworks and libraries required to run an example is usually available at the beginning of each notebook.

11.3 Jupyter Kernels

In Jupyter, kernels are defined as JSON files.

Any user that the cluster administrator has registered in the Linux-PAM system can list the available Jupyter kernels via the command line. The following example is run in the initial Jupyter environment:

Example

```
[jupyterhubuser@basecm11 ~]$ module load jupyter
[jupyterhubuser@basecm11 ~]$ jupyter kernelspec list
Available kernels:
  python3    /cm/shared/apps/jupyter/current/share/jupyter/kernels/python3
```

Each kernel directory contains a `kernel.json` file describing how Jupyter spawns that kernel:

Example

```
[jupyterhubuser@basecm11 ~]$ ls /cm/shared/apps/jupyter/current/share/jupyter/kernels/*/kernel.json
/cm/shared/apps/jupyter/current/share/jupyter/kernels/python3/kernel.json
```

In addition to specifications for shared kernels, each user can define new personal ones in the home directory. By default, the Jupyter data directory for a user is located at `$HOME/.local/share/jupyter`.

This path can be verified with Jupyter by using the `--paths` option:

Example

```
[jupyterhubuser@basecm11 ~]$ jupyter --paths
config:
    /home/jupyterhubuser/.jupyter
    /home/jupyterhubuser/.local/etc/jupyter
    /cm/local/apps/jupyter/conf
    /cm/shared/apps/jupyter/current/etc/jupyter
data:
    /home/jupyterhubuser/.local/share/jupyter
    /cm/shared/apps/jupyter/current/share/jupyter
runtime:
    /home/jupyterhubuser/.local/share/jupyter/runtime
```

The simplest definition for a Python3 kernel designed to run on the login node is:

```
{
 "argv": ["python",
    "-m",
    "ipykernel_launcher",
    "-f",
    "{connection_file}"
 ],
 "display_name": "Python 3",
 "language": "python"
}
```

In the preceding kernel definition:

- `argv`: is the command to be executed to locally spawn the kernel
- `"display_name"`: is the name to be displayed in the JupyterLab interface
- `"language"`: is the supported programming language (`"language"`)
- `"{connection_file}"` (`https://jupyter-client.readthedocs.io/en/stable/kernels.html#connection-files`) is a placeholder, and is replaced by Jupyter with the actual path to the connection file before starting the kernel.

The following kernel is Jupyter's default Python 3 kernel distributed by BCM in the initial environment:

Example

```
[jupyterhubuser@basecm11 ~]$ cat /cm/shared/apps/jupyter/current/share/jupyter/kernels/python3/kernel.json

{
 "argv": [
  "/cm/local/apps/python312/bin/python3.12",
  "-m",
  "ipykernel_launcher",
  "--InteractiveShellApp.extra_extensions=cm_jupyter_wlm_magic",
  "--TerminalIPythonApp.extra_extensions=cm_jupyter_wlm_magic",
  "-f",
  "{connection_file}"
 ],
 "display_name": "Python 3",
 "language": "python",
 "env": {
  "PYTHONPATH": "/cm/shared/apps/jupyter/current/lib64/python3.12/site-packages:/cm/shared/apps/jupyter/
current/lib/python3.12/site-packages"
 }
}
```

The two kernels are not very different. They differ from each other in the Python 3 binary path, the IPython extension (Jupyter WLM Magic), and the exported `PYTHONPATH` environment variable (`"env"`).

11.3.1 Jupyter Kernel Provisioning Kernels

Jupyter is designed to run both the kernel processes, as well as the user interface (JupyterLab or Jupyter Notebook) on the same host. The kernel `{connection_file}` is therefore stored in the `~/.local/share/jupyter/runtime` directory, or in the `/run` directory.

JupyterLab can delegate the task of spawning kernels to another component, Jupyter Kernel Provisioning, as defined in the kernel's JSON file for `kernel_provisioner`. Jupyter Kernel Provisioning allows the complete life-cycle of several Jupyter kernels to be managed at the same time—their start, status monitoring, and termination, but the particular Jupyter kernels that run are otherwise independent of Jupyter Kernel Provisioning, and can be managed by third parties.

Jupyter Kernel Provisioning requires an extended `kernel.json` definition to describe a particular process-proxy module to handle the kernel.

A simple definition for a Python3 kernel designed to be scheduled via JEG is:

```
{
  "display_name": "Python 3.12 via SLURM 250102185019",
  "language": "python",
  "metadata": {
    "kernel_provisioner": {
      "provisioner_name": "slurm-provisioner",
      "config": {
        "timeout": 60,
        "response_manager": {
            "version": 2
        },
        "submit_cmd": {
            "path": "templates/submit_cmd.sh.j2",
            "vars": {
                "modules": "shared slurm jupyter-eg-kernel-wlm-py312"
            }
        },
        "query_cmd": {
            "path": "templates/query_cmd.sh.j2",
            "vars": {
                "modules": "shared slurm jupyter-eg-kernel-wlm-py312"
            }
        },
        "info_cmd": {
            "path": "templates/info_cmd.sh.j2",
            "vars": {
                "modules": "shared slurm jupyter-eg-kernel-wlm-py312"
            }
        },
        "cancel_cmd": {
            "path": "templates/cancel_cmd.sh.j2",
            "vars": {
                "modules": "shared slurm jupyter-eg-kernel-wlm-py312"
            }
        },
        "submit_script": {
            "path": "templates/submit_script.sh.j2",
            "vars": {
                "job_prefix": "jupyter-kernel-slurm-py312",
                "partition": "",
                "ntasks": "1",
                "gres": "",
```

```
                "work_dir": "/home/alice",
                "modules": "shared slurm jupyter-eg-kernel-wlm-py312",
                "oversubscribe": false,
                "pythonuserbase_loc": "temp"
            }
        }
      }
    }
  },
  "argv": []
}
```

In this example, the "metadata" entry has been added. It includes "kernel_provisioner" and "provisioner_name", which define the exact provisioner being used to manage the life-cycle of the kernel. The provisioner is defined via Python's entry points specification.

It also contains paths to several script templates used to spawn the job within the context that the kernel process runs, and sets the corresponding environment variables and other variables for the templates. The "argv": [] is empty because it is replaced by a script defined in templates/submit_script.sh.j2

BCM is equipped with several types of provisioners to interact with a wide range of resource managers, such as Kubernetes or Slurm. This allows kernels to be scheduled across compute nodes.

BCM recommends that kernels using the Jupyter Kernel Provisioning mechanism are created and used with the Jupyter Kernel Creator (section 11.4) extension.

11.3.2 Tunables For Kernel Provisioners

BCM provides defaults for all the templates. The aim is to have the kernels that are created just work on a typical cluster. However a better fit to the running environment may be possible with some further fine-tuning.

Configuration parameters for modifying the kernel templates can be added with cmsh. These parameters are put into the Jupyter configuration file at /cm/local/apps/jupyter/conf/jupyterhub_config.py, and become accessible when JupyterLab starts. Templates or already-created kernels can be edited—which might be a preferred approach to test a parameter before simply adding it via cmsh. Parameters that are set in the kernel specification (the kernel.json file) have precedence over the ones set in jupyterhub_config.py

The following example shows a session that adds the c.KernelResponseManager.public_ip configuration parameters within cmsh:

Example

```
[root@basecm11 ~]# cmsh
[basecm11]% configurationoverlay
[basecm11->configurationoverlay]% use jupyterhub
[basecm11->configurationoverlay[jupyterhub]]% roles
[basecm11->configurationoverlay[jupyterhub]->roles]% use jupyterhub
[basecm11->configurationoverlay[jupyterhub]->roles[jupyterhub]]% configs
[basecm11->...]->roles[jupyterhub]->configs]% add c.KernelResponseManager.public_ip
[basecm11->...]->roles*[jupyterhub*]->configs*[c.KernelResponseManager.public_ip*]]% set value "'10.10.1.1'"
[basecm11->...]->roles*[jupyterhub*]->configs*[c.KernelResponseManager.public_ip*]]% commit
```

On commit, the cm-jupyter service is restarted, which means that all user sessions are dropped. To avoid this, editing the templates directly in the /cm/shared/apps/jupyter/current/share/jupyter/kerneltemplates directory can be considered.

Table 11.3.2: Jupyter Kernel Tunables

Configuration parameter	Path in kernel.json (metadata.kernel_provisioner.config)	Default	Description
c.CMKernelProvisionerBase.timeout	.timeout	5	Timeout starting kernel
c.CMKernelProvisionerBase.include_regex_env	.include_regex_env	^(.+_API_(KEY|TOKEN|HOST|TYPE|ORG_ID|ENDPOINT)|HF_.+)$	Regex for environment variable names to be inherited from JupyterLab process running on login node
c.CMKernelProvisionerBase.exclude_regex_env	.exclude_regex_env	^(JPY_API_TOKEN|JUPYTERHUB_.+|PYTHON.+|JUPYTERLAB_.+|PATH|LD_LIBRARY_PATH.*)$	Regex for environment variable names not to be passed to running kernels
c.CMJKProvisioner.poll_interval	.poll_interval	5	Polling interval and interval between retries for k8s operations
c.CMJKProvisioner.operation_timeout	.operation_timeout	5	Timeout for running commands interacting with k8s
c.KernelResponseManager.public_ip	.response_manager.public_ip	Detected automatically	The IP address on the login node that jupyter-kernel-starter will use for callbacks when the kernel is started on the compute node
c.KernelResponseManager.public_network	.response_manager.public_network	Detected automatically	The network on the login node that jupyter-kernel-starter will use for callbacks when the kernel is started on the compute node
c.KernelResponseManager.public_hostname	.response_manager.public_hostname	Detected automatically	External hostname of the login node can be specified, and the public IP address will be detected by resolving
c.KernelResponseManager bind_ip	.response_manager.bind_ip	Detected automatically	The IP address used by the the response manager to bind the socket that listens for callbacks from jupyter-kernel-starter

...continues

...continued

Configuration parameter	Path in kernel.json (metadata.kernel_provisioner.config)	Default	Description
c.KernelResponseManager.bind_network	.response_manager.bind_network	Detected automatically	The IP address used by the response manager to bind the socket that listens for callbacks from jupyter-kernel-starter
c.KernelResponseManager.bind_port_range_start	.response_manager.bind_port_range_start	1025	The start of the port range within which the response manager tries to obtain a port, for listening to callbacks from the kernel
c.KernelResponseManager.bind_port_range_end	.response_manager.bind_port_range_end	65535	The end of the port range within which the response manager tries to obtain a port, for listening to callbacks from the kernel
c.KernelResponseManager.bind_port_retries	.response_manager.bind_port_retries	16	How many times the response manager tries to find a free port
c.KernelResponseManager.max_in_requests	.response_manager.max_in_requests	5	How many incoming messages to the kernel starter can be buffered
c.KernelResponseManager.max_out_requests	.response_manager.max_out_requests	5 5	How many outgoing messages to the kernel starter can be buffered

The *response manager* in the preceding table is a part of WLM kernel provisioners. It is dedicated to getting callbacks and managing signal communications with the running kernel. It acts as a proxy for system signals and informs the kernel provisioner about the kernel being started and which ports it is listening to. The response manager works in tandem with jupyter-kernel-starter.

11.4 Jupyter Kernel Creator Extension

Creating or editing kernels can be cumbersome and error-prone for users, depending on the features of the execution context desired for their notebooks.

To provide a more user-friendly experience, BCM includes the *Jupyter Kernel Creator* extension in JupyterLab. This extension is accessed from the navigation pane in the JupyterLab interface, by clicking on the BCM icon.

Jupyter Kernel Creator allows users to create kernels using the JupyterLab interface, without the need to directly edit JSON files. With this interface users can create kernels by customizing an available

template according to their needs.

A template can be considered to be the skeleton of a kernel, with several preconfigured options, and others options that are yet to be specified. Common customizations for templates include environment modules to be loaded, workload manager queues to be used, number and type of GPUs to acquire, and so on.

Templates are usually defined by administrators according to cluster capabilities, programming languages and user requirements. Each template can provide different options for customizations.

Administrators often create different templates to take advantage of different workload managers, programming languages and hardware resources. For example, an administrator may define a template for scheduling Python kernels via Kubernetes, another one for R kernels via Slurm, and yet another one for Bash kernels via Platform LSF.

11.4.1 BCM Predefined Kernel Templates

To simplify Jupyter configuration for administrators, BCM distributes a number of pre-defined templates with Jupyter Kernel Creator. These templates can be used for default configurations of BCM workload managers, and can be customized and extended for more advanced use. Kernel templates defined by BCM can be found in the Jupyter installation directory, under the `kerneltemplates` directory:

```
[jupyterhubuser@basecm11 ~]$ ls /cm/shared/apps/jupyter/current/share/jupyter/kerneltemplates/
filter.yaml         k8s-cmjkop-py         lsf-py312       pbspro-bash    slurm-py312      slurm-pyxis-r
k8s-cmjkop-julia    k8s-cmjkop-py-spark   openpbs-bash    pbspro-py312   slurm-py-conda
k8s-cmjkop-ngc-py   lsf-bash              openpbs-py312   slurm-bash     slurm-pyxis-py
```

BCM Predefined Kernel Templates Seen By Users

Users can view the available predefined kernel templates in the Jupyter web browser interface, within the `KERNEL TEMPLATES` section of the dedicated BCM extensions panel (figure 11.2):

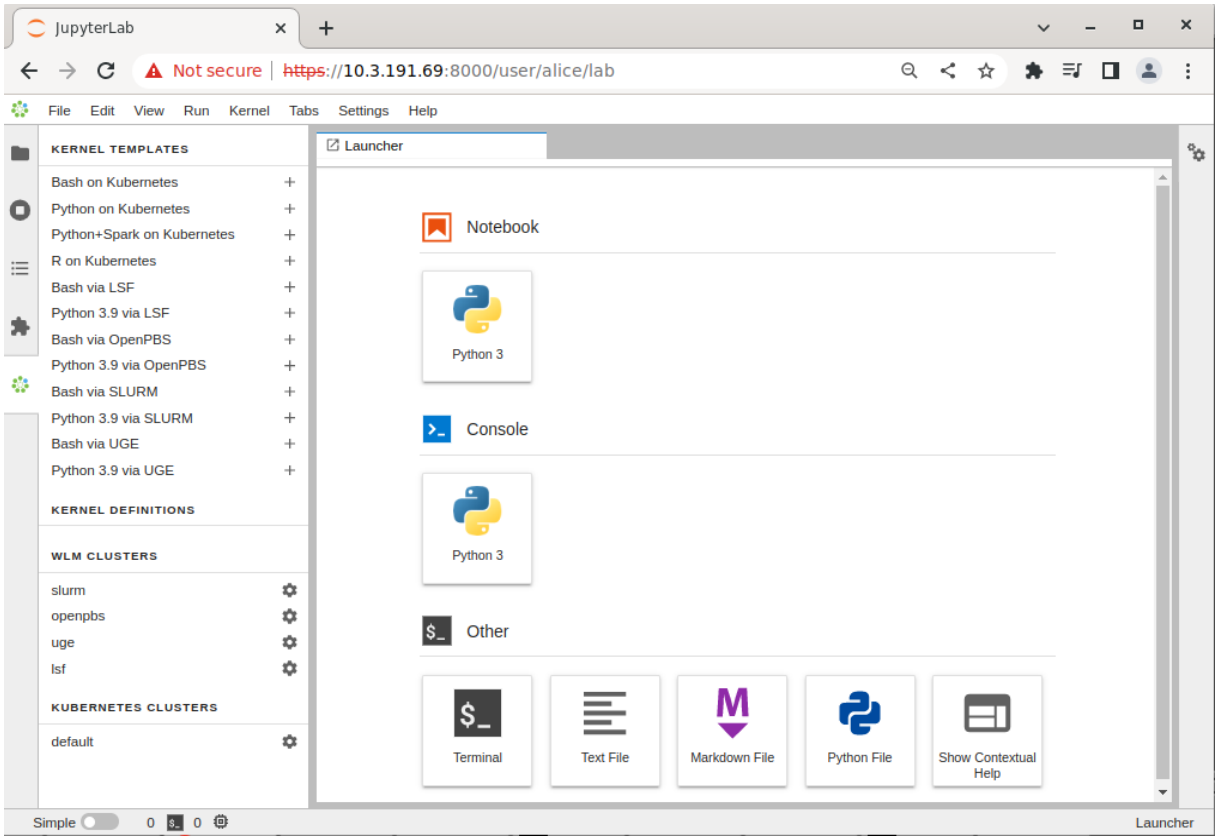

Figure 11.2: JupyterLab BCM extensions section with kernel templates

However, the templates provided by BCM are listed in the panel only if they can be used on the cluster. This means that the templates are listed only after the associated workload manager instance, or associated Kubernetes configuration (such as a Kubernetes operator), have been deployed by cluster manager utilities. For example:

- After running the `cm-wlm-setup` cluster manager utility to deploy an OpenPBS workload manager, the `openpbs-bash` and `openpbs-py312` templates become available. The templates are listed as:
  - `Bash via OpenPBS`
  - `Python 3.12 via OpenPBS`

  and accessed via the navigation path: *menu* > *dedicated BCM extension panel* > *kernel templates section*.
- After running the `cm-kubernetes-setup` cluster manager utility to deploy a Kubernetes cluster, the Kubernetes cluster instance is displayed (navigation path: *menu* > *dedicated BCM extension panel* > *Kubernetes clusters section*)

  Then, after running the `cm-jupyter-kernel-operator` cluster manager utility to deploy a Jupyter kernel operator package, and configuring a user (section 6.3 of the *Containerization Manual*), the templates `k8s-cmjkop-julia`, `k8s-cmjkop-py`, and `k8s-cmjkop-py-spark` become available.

  The templates are listed as:
  - `Julia on Kubernetes Operator`
  - `Python on Kubernete Operator`
  - `Python+Spark on Kubernetes Operator`

  and accessed via the navigation path: *menu* > *dedicated BCM extension panel* > *kernel templates section*.

Users can instantiate a kernel template to create an actual kernel from the dedicated BCM extensions section using the + button of the template. A dialog is dynamically generated for the template being instantiated, and users are asked to fill a number of customization options defined by administrators (figure 11.3):

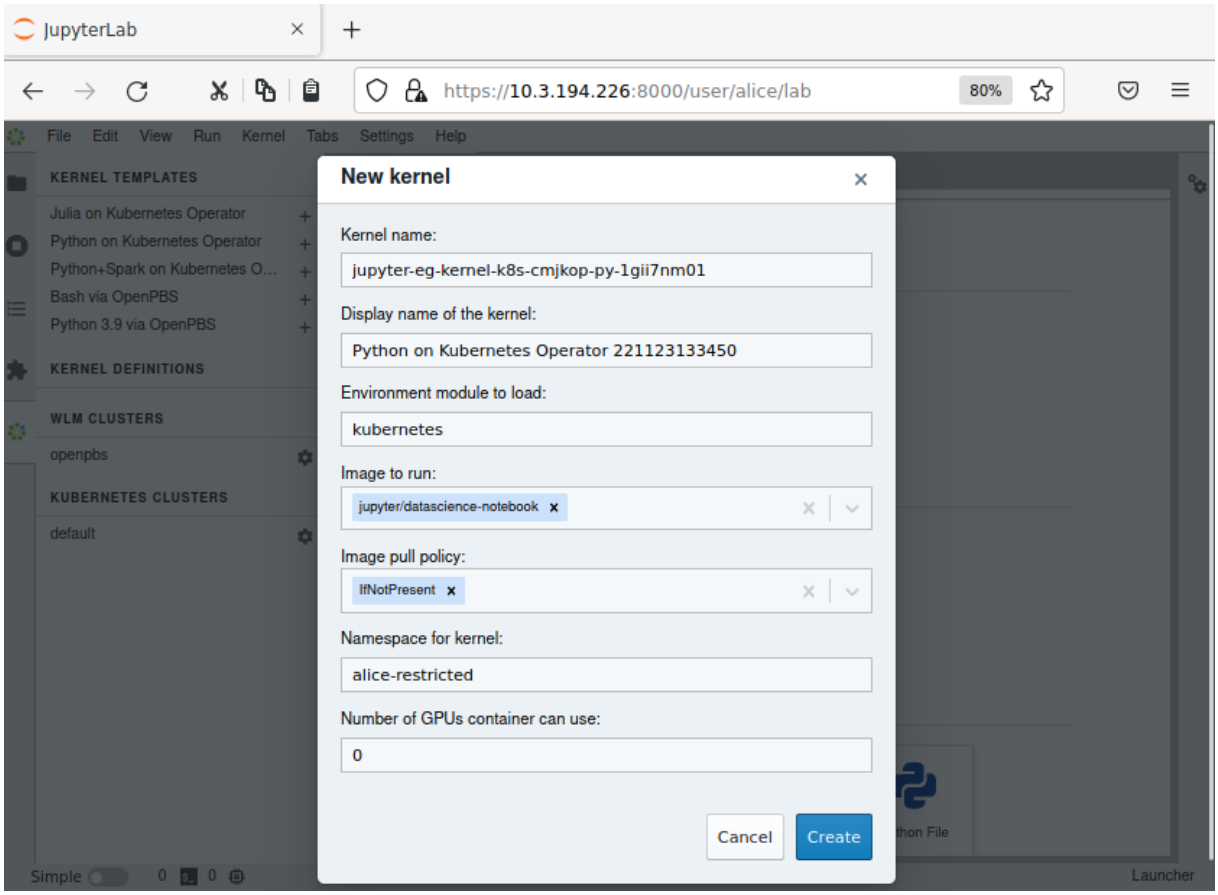

Figure 11.3: Jupyter kernel template customization screen

Once the template is completely customized, the kernel can be created. It automatically appears in the JupyterLab Launcher screen (figure 11.4) and can be used to run notebooks or a console session (figure 11.4):

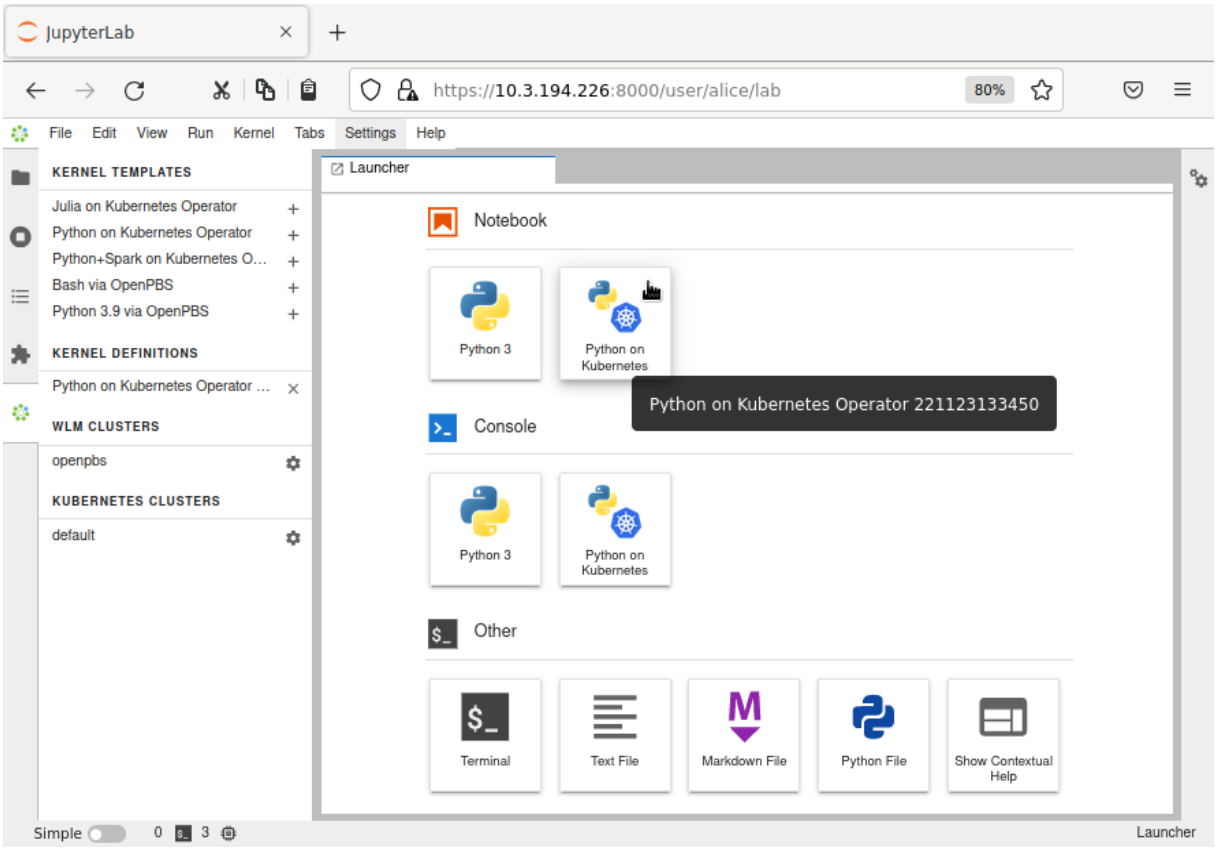

Figure 11.4: JupyterLab Launcher screen with new custom kernel

A user who lists available Jupyter kernels via the command line now sees the newly-created kernel:

```
[jupyterhubuser@basecm11 ~]$ module load jupyter
[jupyterhubuser@basecm11 ~]$ jupyter kernelspec list
Available kernels:
     k8s-cmjkop-py-1gii7nm01    /home/alice/.local/share/jupyter/kernels/k8s-cmjkop-py-1gii7nm01
     python3                    /cm/shared/apps/jupyter/current/share/jupyter/kernels/python3
```

The new kernel directory will contain the JSON definition generated by the Jupyter Kernel Creator:

```
[jupyterhubuser@basecm11 ~]$ cat .local/share/jupyter/kernels/k8s-cmjkop-py-1gii7nm01/kernel.json

{
  "language": "python",
  "display_name": "Datascience Notebook Kernel",
  "metadata": {
    "kernel_provisioner": {
      "provisioner_name": "cmjk-provisioner",
      "config": {
        "timeout": 280,
        "template": {
          "path": "templates/cmjk.yaml.j2",
          "env_module": "kubernetes",
          "vars": {
            "image": "quay.io/jupyter/datascience-notebook",
            "namespace": "alice-restricted",
            "image_pull_policy": "IfNotPresent",
            "gpu_limit": 0,
```

```
            "pythonuserbase_loc": "temp"
          }
        }
      }
    }
  },
  "argv": []
}
```

Jupyter kernel names need not be unique. Users should therefore choose meaningful and distinguishable display names for their kernels. Doing so makes the JupyterLab Launcher screen easier to use.

For convenience, a summary of the available kernel templates and their requirements is shown in table 11.1:

Table 11.1: *Available Jupyter kernel templates for BCM and their requirements*

Template name	Requirement	Description
`k8s-cmjkop-julia`	Kubernetes	Jupyter official image via Jupyter Kernel Operator (using Julia)
`k8s-cmjkop-ngc-py`	Kubernetes	NGC images via Jupyter Kernel Operator
`k8s-cmjkop-py`	Kubernetes	Jupyter official image via Jupyter Kernel Operator (Python)
`k8s-cmjkop-py-spark`	Kubernetes[1]	Python + Spark via Jupyter Kernel Operator (using Python and Spark
`lsf-bash`	Platform LSF	Bash via Platform LSF
`lsf-py312`	Platform LSF	Python 3.12 via Platform LSF
`openpbs-bash`	Open PBS	Bash via Open PBS
`openpbs-py312`	Open PBS	Python 3.12 via Open PBS
`pbspro-bash`	PBS Professional	Bash via PBS Professional
`pbspro-py312`	PBS Professional	Python 3.12 via PBS Professional
`slurm-bash`	Slurm	Bash via Slurm
`slurm-py312`	Slurm	Python 3.12 via Slurm
`slurm-py-conda`	Slurm + Conda[2]	Python 3.12 and Conda via Slurm
`slurm-pyxis-py`	Slurm + Enroot	Python running inside imported Pyxis+Enroot image in Slurm

...continues

Table 11.1: Available Jupyter kernel templates...continued

Template name	Requirement	Description
`slurm-pyxis-r`	Slurm + Enroot	R running inside imported Pyxis+Enroot image in Slurm

[1] Docker image: *docker.io/brightcomputing/jupyter-kernel-sample:k8s-spark-3.5.3-py38-cuda12.6-rapids24.08-2*
[2] Conda needs to be installed for the user, and the Conda environment needs to be configured in the user's Bash shell (section 11.4.2).
DockerHub kernels page: `https://hub.docker.com/r/brightcomputing/jupyter-kernel-sample/tags`

11.4.2 Using Conda Kernels With Jupyter

Conda is a package manager for languages. Miniconda installs a minimal Conda environment.

Installing Conda With Miniconda And Initializing Conda

If Slurm is the workload manager that is installed and available to the Jupyter user, then Conda templates can be installed and made available to the user by installing Miniconda 3. Miniconda installation is described in the official Conda documentation at:

`https://docs.conda.io/projects/miniconda/en/latest/`

At the time of writing (February 2024), the "Quick command line install" for Linux at `https://docs.anaconda.com/free/miniconda/` explains how to carry out a Linux command line installation for Miniconda 3. The following is an excerpt from that text, slightly modified for BCM use:

> These four commands quickly and quietly install the latest 64-bit version of the installer and then clean up after themselves. To install a different version or architecture of Miniconda for Linux, change the name of the `.sh` installer in the `wget` command.

```
[jupyterhubuser@basecm11 ~]$ mkdir -p ~/miniconda3
[jupyterhubuser@basecm11 ~]$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh \
  -O ~/miniconda3/miniconda.sh
[jupyterhubuser@basecm11 ~]$ bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3
[jupyterhubuser@basecm11 ~]$ rm -rf ~/miniconda3/miniconda.sh
```

If the user has modules (section 2.3) loaded that set `$PYTHONPATH`—for example by default—then it can cause Python version incompatibilities later on when using a Conda environment that uses another version of Python. This is explained further in the section "Avoiding Python Version Incompatibilities With Conda" (page 101). A warning is given during the Miniconda installation if an existing `$PYTHONPATH` is detected. However it is up to the user to ensure that modules are not left in place that interfere with regular Python usage within Conda, for example by checking the `~/.bash_profile` and similar files for what modules are normally loaded in the shell.

After installation, the newly-installed Conda must be initialized:

```
[jupyterhubuser@basecm11 ~]$ ~/miniconda3/bin/conda init bash
...
modified      /home/jupyterhubuser/.bashrc

==> For changes to take effect, close and re-open your current shell. <==
```

Users may use other shells, but initialization must be done with Bash.

On restarting the shell, the base environment is indicated with the prompt:

```
(base) [jupyterhubuser@basecm11 ~]$
```

The Python version for the Conda environment can be seen by running `python -V`. Typically it differs from the system version.

When a Miniconda installation is completed with the preceding `conda init bash` command, Conda templates become available in the kernel templates list (figure 11.5).

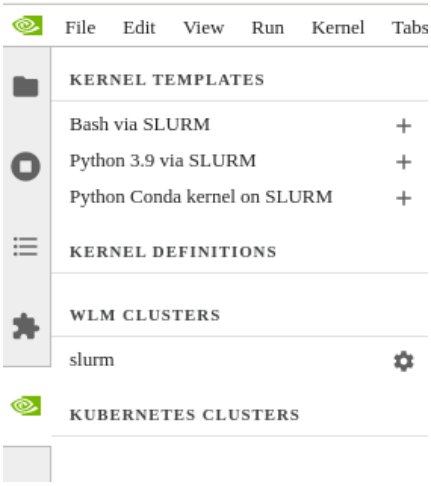

Figure 11.5: Conda template item showing up in the menu

Creating And Running A Conda Environment Compatible With Jupyter

The Conda virtual environment by default has its own version of Python. The Conda virtual environment can be created with a minimal set of packages to run a Jupyter kernel.

To make a kernel compatible with Jupyter Kernel Provisioning, the `cm-jupyter-eg-kernel-wlm` package needs to be installed in the Conda virtual environment with a particular version of Python. This can be carried out with:

Example

```
(base) [jupyterhubuser@basecm11 ~]$ conda config --append channels conda-forge
(base) [jupyterhubuser@basecm11 ~]$ conda create -n myenv python=3.11 -y
Channels:
 - defaults
 - conda-forge
...
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate myenv
#
# To deactivate an active environment, use
#
#     $ conda deactivate
(base) [jupyterhubuser@basecm11 ~]$ module remove jupyter      #do not mix Python versions
(myenv) [jupyterhubuser@basecm11 ~]$ conda activate myenv
(myenv) [jupyterhubuser@basecm11 ~]$ conda install -y -c file:///cm/shared/apps/jupyter/current/\
                                     share/conda-repo cm-jupyter-eg-kernel-wlm
Channels:
 - file:///cm/shared/apps/jupyter/current/share/conda-repo
 - defaults
 - conda-forge
...
Downloading and Extracting Packages:
```

```
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

Avoiding Python Version Incompatibilities With Conda

In the preceding example, the Conda environment `myenv` is set explicitly to use Python version 3.11.

Some BCM environment modules, such as the `jupyter` environment module. define the `$PYTHONPATH` environment variable. Before activating a Conda environment, unloading such BCM environment modules is recommended. For example, in the case of Jupyter, with `module unload jupyter`. This is because the value of `$PYTHONPATH` from Jupyter sets the version of Python that is used, which may conflict with that intended by Conda. Having Python modules that have a version incompatible with the one that of Conda can cause issues that are hard to debug.

Besides Jupyter, software that has modules that can affect Python versions in this way are also found in the Machine Learning (ML) group. The ML group is the group that is specified by the ML selection option in `https://support.brightcomputing.com/packages-dashboard/`.

If needed, additional software can be installed into the environment from the Conda channels using its `-c|--channel` option. For example, PyTorch—which outside Conda is a package in the ML group—can be provided within Conda with:

Example

```
(myenv) [jupyterhubuser@basecm11 ~]$ conda install pytorch torchvision torchaudio pytorch-cuda=11.8 \
                                     -c pytorch -c nvidia
```

The preceding PyTorch installation is based on the guidance at `https://pytorch.org/get-started/locally/`.

Available environments appear in a list dropdown:

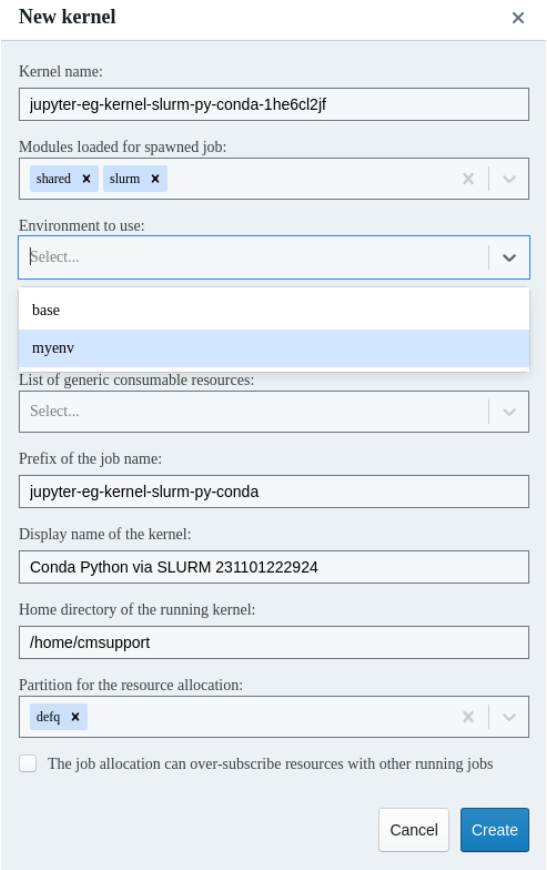

Figure 11.6: Conda template: environment list dropdown

11.4.3 Using Enroot And Pyxis With Jupyter

If Slurm is configured by the cluster administrator with Enroot and Pyxis support, then a template for running kernels inside Enroot containers appears in the list.

Enroot must be configured by the administrator with both of the following settings enabled:

- `Share raw images among users and nodes`
- `Share unpacked image files among nodes`

The first invocation of the command:

`enroot list`

must be performed by the administrator too.

Before creating a kernel from a template, the image must be imported, so that it can be prepared to have the kernel run within it. The Jupyter package has a script to do this:

Example

```
[jupyterhubuser@basecm11 ~]$ module load jupyter
[jupyterhubuser@basecm11 ~]$ jupyter-enroot-import -h
Usage: jupyter-enroot-import --container-image <image> [--container-name <name>] [--as-sqsh]
[--partition <partition>] [--install-script]
[jupyterhubuser@basecm11 ~]$ jupyter-enroot-import --container-image nvcr.io#nvidia/pytorch:23.12-py3
[jupyterhubuser@basecm11 ~]$ jupyter-enroot-import --container-image python:3.11
```

The container can also be pulled as a SquashFS file:

Example

```
[jupyterhubuser@basecm11 ~]$ module load jupyter
[jupyterhubuser@basecm11 ~]$ jupyter-enroot-import --container-image nvcr.io#nvidia/pytorch:23.12-py3 \
--as-sqsh --container-name /home/alice/images/pytorch_23.12_py3.sqsh
[jupyterhubuser@basecm11 ~]$ jupyter-enroot-import --container-image python:3.11 --as-sqsh \
--container-name /home/alice/images/python_3_11.sqsh
```

A job is submitted to pull and unpack the image. The script uses the default Slurm module that is available, and uses the default Slurm queue. If necessary, a specific module can be loaded, and a particular queue specified. The container name can also be customized. If the name is not specified, then it is chosen based on the container image name.

Example

```
[jupyterhubuser@basecm11 ~]$ module load jupyter
[jupyterhubuser@basecm11 ~]$ module load slurmslurm/slurm/23.11.3
[jupyterhubuser@basecm11 ~]$ jupyter-enroot-import --container-image \
                          quay.io#jupyter/datascience-notebook:latest --partition defq
```

When the jobs are finished then the imported images can be displayed in the `enroot list` output:

```
[jupyterhubuser@basecm11 ~]$ enroot list
pyxis_nvcr_io_nvidia_pytorch_23_12_py3
pyxis_python_3_11
```

The template can then be instantiated and the kernel created. If NGC containers are to be used, then a GPU resource must be added to the kernel.

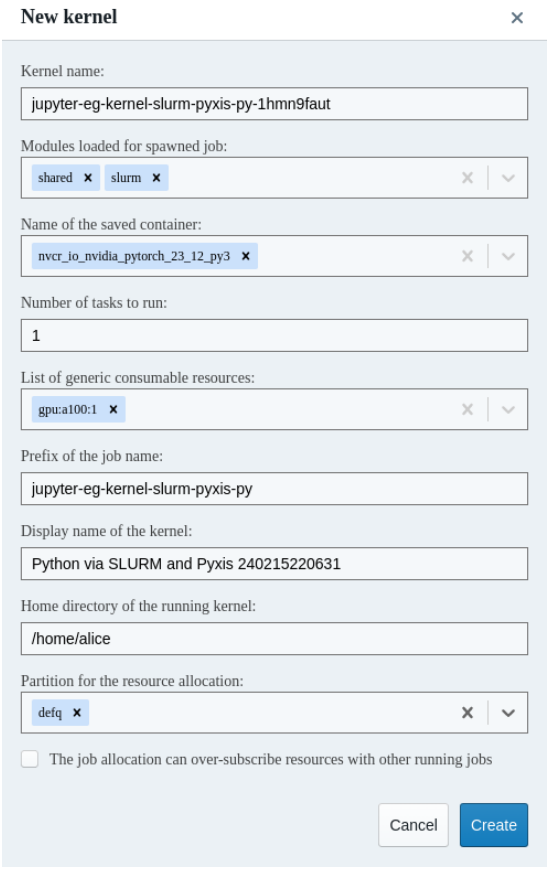

Figure 11.7: Pyxis template for Python

Alternatively, an image based on the R language can be imported and used:

Example

```
[jupyterhubuser@basecm11 ~]$ module load jupyter
[jupyterhubuser@basecm11 ~]$ jupyter-enroot-import --container-image quay.io#jupyter/r-notebook:latest
```

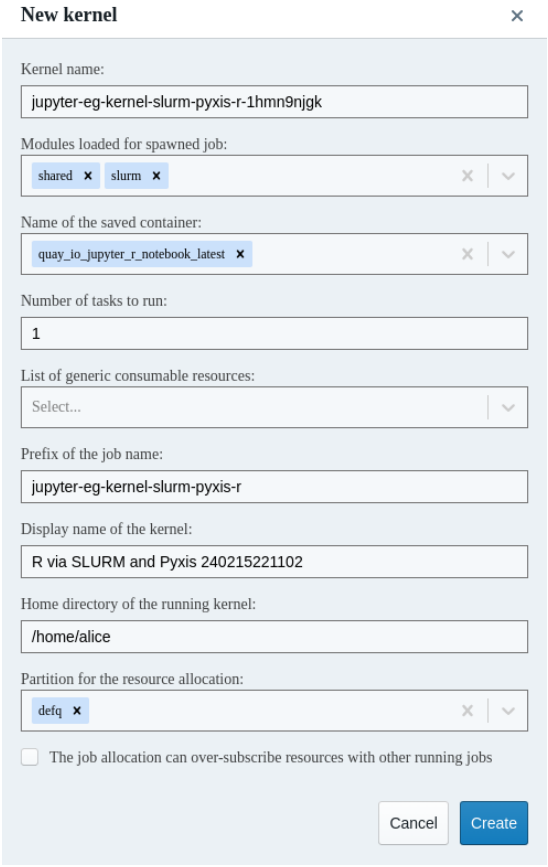

Figure 11.8: Pyxis template for R

11.5 Changing The User Base Directory In Python Kernels

BCM-provided Python kernel templates can change the user base directory (`site.USER_BASE`) during creation. It can be set to take the value of `temp`, `permanent`, or `not set` on kernel creation.

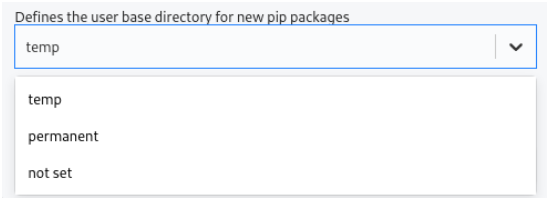

Figure 11.9: Setting PYTHONUSERBASE

The setting affects the path where pip packages are installed and searched (that is, when running `!pip install` in the Notebook) by setting `$PYTHONUSERBASE` environmental variable. Further details on `$PYTHONUSERBASE` and `site.USER_BASE` can be found in the official Python documentation at `https://docs.python.org/3/using/cmdline.html#envvar-PYTHONUSERBASE`

- `not set`: does not set PYTHONUSERBASE. So, if notebook installs additional pip packages then they are installed in `~/.local/lib/pythonX.Y[t]/site-packages`. In such a case, other kernels

with the same Python version might be affected. The directory with installed packages is not cleared after kernel removal.

- `permanent`: the pip packages are installed into the `/home/alice/.local/share/jupyter/kernels/kernel-name/` directory so that other kernels are not affected. The directory is removed if the kernel is removed. Both the `not set` and the `permanent` setting survive restarts of the kernel.
- `temp`: The pip packages are installed in an individual subdirectory. This is in `/tmp` on compute nodes, or within a running Docker container. The installed packages disappear when the kernel is restarted.

11.6 Adding Environmental Variables For JupyterLab, Processing And Accessing API keys In Notebooks

are To change the behavior of JupyterLab as spawned by JupyterHub, the `init.rc` file is processed by the `/cm/shared/apps/jupyter/current/bin/jupyterhub-singleuser-gw` script. The `init.rc` file can be used to export environmental variables such as `JUPYTER_LOG_LEVEL` to increase verbosity.

The `init.rc` file can also contain API keys exported as variables, such as `NVCF_API_KEY`, `OPENAI_API_KEY`, or `HF_TOKEN`.

These variables are passed by default in running kernels if they match rules defined in the `include_regex_env` and `exclude_regex_env` tunables. If the content of `~/.jupyter/jupyterhub-singleuser-gw/init.rc` changes, then JupyterLab needs to be restarted from the JupyterHub console, using the navigation path:

`File > Hub Control Panel > Stop My server > Start My Server`

Configuration parameters can be set by an adminstrator on a per-cluster basis using the tunables:

- `c.CMKernelProvisionerBase.include_regex_env`
- `c.CMKernelProvisionerBase.exclude_regex_env`

They can also be set in the `kernel.json.j2` template, via

- `metadata.kernel_provisioner.config.include_regex_env`
- `metadata.kernel_provisioner.config.exclude_regex_env`.

Alternatively these variables can be configured manually in the `kernel.json` files. Default values are:

- `include_regex_env`: `^(.+_API_(KEY|TOKEN|HOST|TYPE|ORG_ID|ENDPOINT)|HF_.+)$`
- `exclude_regex_env`: `^(JPY_API_TOKEN|JUPYTERHUB_.+|PYTHON.+|JUPYTERLAB_.+|PATH|LD_LIBRARY_PATH.*)$`

11.7 Jupyter VNC Extension

11.7.1 What Is Jupyter VNC Extension About?

VNC (Virtual Network Computing) is a screen sharing service that can work in a browser.

If VNC is allowed by the cluster administrator, then the Jupyter environment configured by BCM can be used to start and control remote desktops via VNC with the *Jupyter VNC* extension.

11.7.2 Enabling User Lingering

User lingering is a systemd setting that sets a user manager for a user at boot and keeps it around after logout. This allows that user to run long-running sessions despite not being logged in. Enabling user lingering may be required for Jupyter VNC extension to run for a relatively complicated desktop such as KDE or GNOME.

For each user on each machine where these environments are installed, the following command must be run:

```
loginctl enable-linger <username>
```

The command may also be carried out using prolog/epilog scripts in the chosen WLM.

11.7.3 Starting A VNC Session With The Jupyter VNC Extension

Users can start a VNC session with the button added by Jupyter VNC (figure 11.10). Additional VNC parameters can be optionally specified.

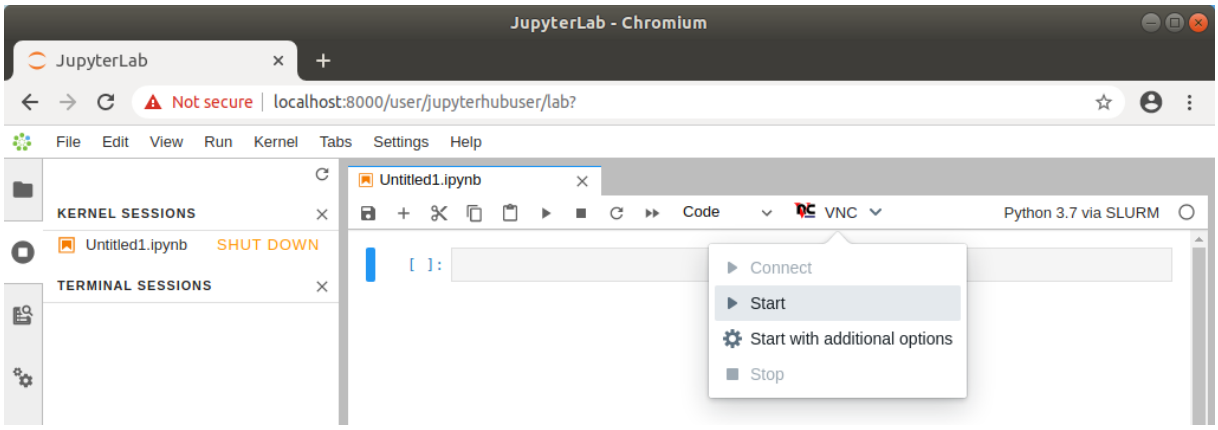

Figure 11.10: Starting Jupyter VNC session from kernel

If VNC is available and correctly configured on the node where the kernel is running, then a new tab is automatically created by Jupyter VNC containing the new session (figure 11.11). A user can now freely interact in JupyterLab both with the notebook and with the desktop environment.

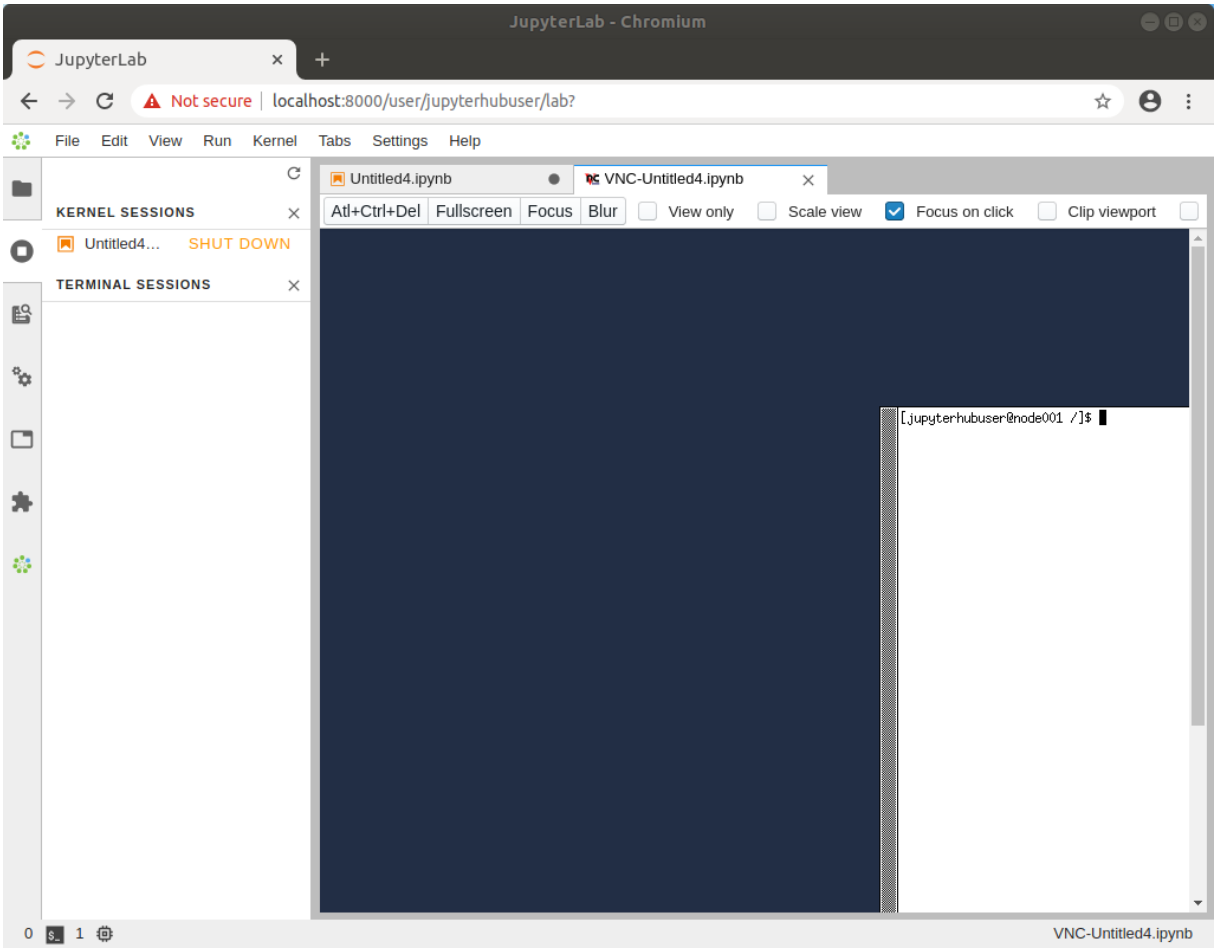

Figure 11.11: Running Jupyter VNC session from kernel

To provide a user-friendly experience, Jupyter VNC also allows the graphical viewport to be resized, so that the desktop application can run full-screen (figure 11.12).

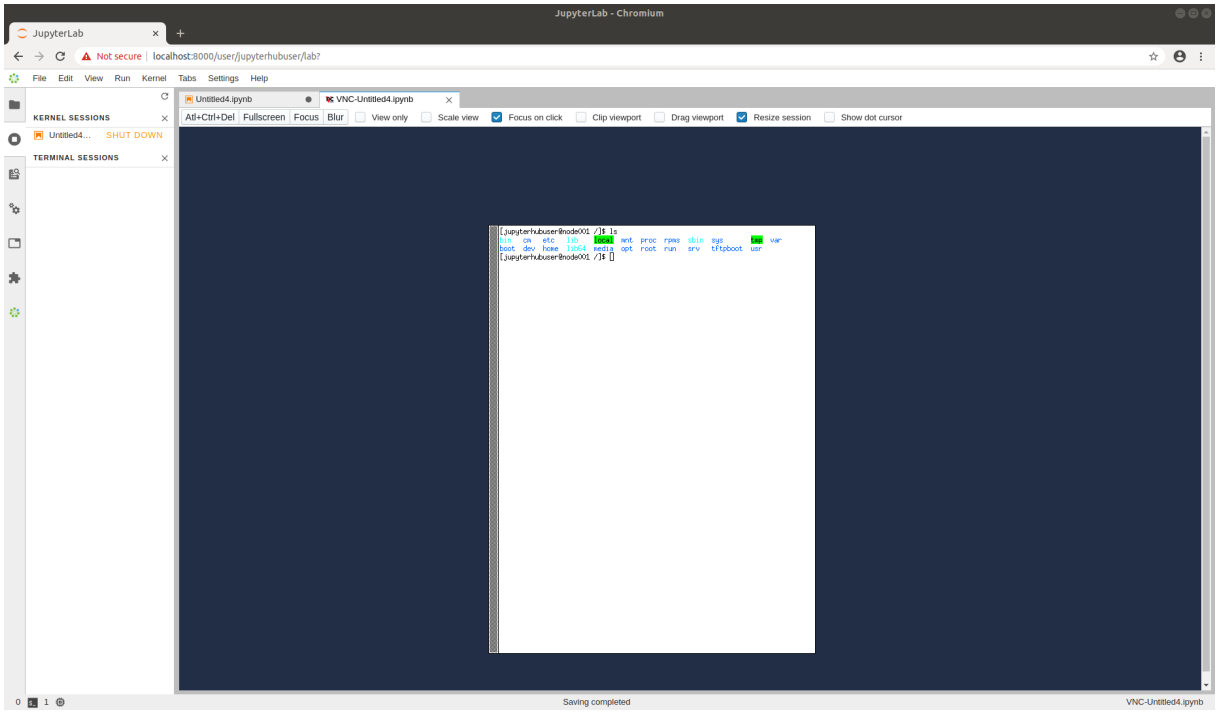

Figure 11.12: Running Jupyter VNC session from kernel (full-screen)

11.7.4 Running Examples And Applications In The VNC Session With The Jupyter VNC Extension

Once the VNC session is correctly started and the new JupyterLab tab has been created, Jupyter VNC automatically exports the `DISPLAY` environment variable to the running notebook (figure 11.13). Doing so means that any application or library running in the notebook can make use of the freshly created desktop environment. An example of such a library is OpenAI Gym, a toolkit for developing and comparing reinforcement learning algorithms, that is distributed by BCM.

Among the examples distributed by BCM (section 11.2), a notebook running PyTorch in the OpenAI Gym CartPole environment can be found. If executed after a VNC session has been started, a user can then observe the model being trained in real time in the graphical environment.

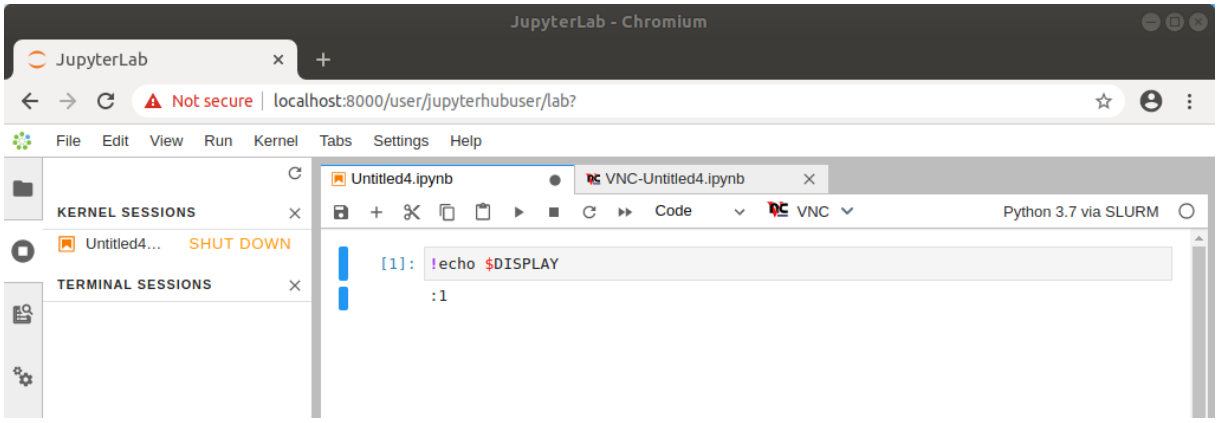

Figure 11.13: Automatic configuration of DISPLAY environment variable

11.8 Jupyter WLM Magic Extension

In the Jupyter environment configured by BCM, the *Jupyter WLM Magic* extension can be used to schedule workload manager jobs from notebooks.

The Jupyter WLM Magic extension is an IPython extension. It is designed to improve the capabilities of Jupyter's default Python 3 kernel, which runs on the login node.

The Jupyter WLM Magic extension should therefore not be used from kernels running on compute nodes, such as those typically created with BCM's Jupyter Kernel Creator extension (section 11.4), and submitted via Jupyter Kernel Provisioning. Indeed, compute nodes running these kernels are often incapable of starting workload manager jobs in many default WLM configurations.

Jupyter WLM Magic extension makes it possible for users to programmatically submit WLM jobs, and then interact with their results. This can be done while using the Python programming language and its libraries, which are available in the notebook.

Users submit jobs and check their progress from the login node. The actual computation is distributed by the underlying workload manager across compute nodes, which means that server resources are spared.

Jupyter WLM Magic commands are available in the IPython kernel as *magic functions* (`https://ipython.readthedocs.io/en/stable/interactive/tutorial.html#magic-functions`). A new line magic (`%`) and a new cell magic (`%%`) are now added in the kernel, according to the workload manager:

- Platform LSF: `%lsf_job` and `%%lsf_job`
- PBS Professional: `%pbspro_job` and `%%pbspro_job`
- Slurm: `%slurm_job` and `%%slurm_job`

A user can list the magic functions in the kernel to see if they are available, with Jupyter's builtin command `%lsmagic` (`https://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-lsmagic`):

Example

```
In  []: %lsmagic
Out []: root:
            line:
                automagic:"AutoMagics"
                autocall:"AutoMagics"
                [...]
                slurm_job:"SLURMMagic"
                pbspro_job:"PBSProMagic"
                lsf_job:"LSFMagic"
            cell:
                js:"DisplayMagics"
                javascript:"DisplayMagics"
                [...]
                slurm_job:"SLURMMagic"
                pbspro_job:"PBSProMagic"
                lsf_job:"LSFMagic"
```

The magic functions introduced by this BCM extension share a similar syntax. For convenience, Slurm is used as an example in this section. However, the same instructions are valid for the other WLMs.

Users can check which options are available for a WLM function with the line magic helper:

Example

```
In  []: %slurm_job --help
Out []: usage: %slurm_job [-h] [--module MODULE] [--module-load-cmd MODULE_LOAD_CMD]
                          [--shell SHELL] [--submit-command SUBMIT_COMMAND]
                          [--cancel-command CANCEL_COMMAND]
                          [--control-command CONTROL_COMMAND]
                          [--stdout-file STDOUT_FILE] [--stderr-file STDERR_FILE]
                          [--preamble PREAMBLE] [--timeout TIMEOUT]
                          [--check-condition-var CHECK_CONDITION_VAR]
                          [--job-id-var JOB_ID_VAR]
                          [--stdout-file-var STDOUT_FILE_VAR]
                          [--stderr-file-var STDERR_FILE_VAR] [--dont-wait]
                          [--write-updates WRITE_UPDATES]
                          [--check-status-every CHECK_STATUS_EVERY]

        optional arguments:
          -h, --help            show this help message and exit
        [...]
```

Line magic functions are typically used to set options with a global scope in the notebook. By doing so, a user will not need to specify the same option every time a job will be submitted via cell magic. For example, if two Slurm instances are deployed on the cluster and their associated environment modules are `slurm-primary` and `slurm-secondary`, a user could run the following line magic once to configure the Jupyter WLM Magic extension to always use the second deployment:

Example

```
In  []: %slurm_job --module slurm-secondary
Out []:
```

Now, jobs will always be submitted to `slurm-secondary`. This is more convenient than repeatedly defining the same module option for every cell magic upon scheduling a job:

Example

```
In  []: %%slurm_job --module slurm-secondary
        <WLM JOB DEFINITION>
Out []: <WLM JOB OUTPUT>
In  []: %%slurm_job --module slurm-secondary
        <WLM JOB DEFINITION>
Out []: <WLM JOB OUTPUT>
```

It should be noted that line magic functions cannot be used to submit WLM jobs. Cell magic functions have to be used instead.

A well-defined cell contains the WLM cell magic function provided by the extension, followed by the traditional job definition. For example, a simple MPI job running on two nodes can be submitted to Slurm by defining and running this cell:

Example

```
In  []: %%slurm_job
        #SBATCH -J mpi-job-example
        #SBATCH -N 2
        module load openmpi
        mpirun hostname
Out []: COMPLETED
        STDOUT file content: /home/demo/.jupyter/wlm_magic/slurm-1.out
```

```
        node001
        node001
        node002
        node002
```

Users can take advantage of the Jupyter WLM Magic extension to store some information into Python variables about the job being submitted. The information could be the ID or the output file name, for example. Users can then later programmatically interact with them in Python. This feature is convenient when a user wants to, for example, programmatically carry out new actions depending on the job output:

Example

```
In  []: %%slurm_job --job-id-var my_job_id --stdout-file-var my_job_out
        #SBATCH -J mpi-job-example
        #SBATCH -N 2
        module load openmpi
        mpirun hostname
Out []: COMPLETED
        STDOUT file content: /home/demo/.jupyter/wlm_magic/slurm-2.out
        node001
        node001
        node002
        node002

In  []: print(f"Job id {my_job_id} was written to {my_job_out}")
        print(f"Output lines: {open(my_job_out).readlines()}")
Out []: Job id 2 was written to /home/demo/.jupyter/wlm_magic/slurm-2.out
        Output lines: ['node001\n', 'node001\n', 'node002\n', 'node002\n']
```

Users can also exploit Python variables to define the behavior of the Jupyter WLM Magic extension. For example, they can define a Python boolean variable to submit a WLM job only if a condition is true:

Example

```
In  []: run_job = 1 == 2
Out []:

In  []: %%slurm_job --check-condition-var run_job
        #SBATCH -J mpi-job-example
        #SBATCH -N 2
        module load openmpi
        mpirun hostname
Out []: Variable run_job is 'False'. Skipping submit.
```

A

MPI Examples

A.1 "Hello world"

A quick application to test the MPI libraries and the network.

```
/*
 ``Hello World'' Type MPI Test Program
*/
#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define BUFSIZE 128
#define TAG 0

int main(int argc, char *argv[])
{
  char idstr[32];
  char buff[BUFSIZE];
  int numprocs;
  int myid;
  int i;
  MPI_Status stat;

  /* all MPI programs start with MPI_Init; all 'N' processes exist thereafter */
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs); /* find out how big the SPMD world is */
  MPI_Comm_rank(MPI_COMM_WORLD,&myid); /* and this processes' rank is */

  /* At this point, all the programs are running equivalently, the rank is used to
     distinguish the roles of the programs in the SPMD model, with rank 0 often used
     specially... */
  if(myid == 0)
  {
    printf("%d: We have %d processors\n", myid, numprocs);
    for(i=1;i<numprocs;i++)
    {
      sprintf(buff, "Hello %d! ", i);
      MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
    }
    for(i=1;i<numprocs;i++)
    {
```

```
      MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
      printf("%d: %s\n", myid, buff);
    }
  }
  else
  {
    /* receive from rank 0: */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    sprintf(idstr, "Processor %d ", myid);
    strcat(buff, idstr);
    strcat(buff, "reporting for duty\n");
    /* send to rank 0: */
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
  }

  /* MPI Programs end with MPI Finalize; this is a weak
     synchronization point */
  MPI_Finalize();
  return 0;
}
```

A.2 MPI Skeleton

The sample code below contains the complete communications skeleton for a dynamically load balanced head/compute node application. Following the code is a description of some of the functions necessary for writing typical parallel applications.

```
include <mpi.h>
#define WORKTAG     1
#define DIETAG     2
main(argc, argv)
int argc;
char *argv[];
{
        int         myrank;
        MPI_Init(&argc, &argv);   /* initialize MPI */
        MPI_Comm_rank(
        MPI_COMM_WORLD,   /* always use this */
        &myrank);      /* process rank, 0 thru N-1 */
        if (myrank == 0) {
                head();
        } else {
                computenode();
        }
        MPI_Finalize();       /* cleanup MPI */
}

head()
{
        int     ntasks, rank, work;
        double       result;
        MPI_Status     status;
        MPI_Comm_size(
        MPI_COMM_WORLD,   /* always use this */
        &ntasks);          /* #processes in application */
```

```
/*
* Seed the compute nodes.
*/
        for (rank = 1; rank < ntasks; ++rank) {
                work = /* get_next_work_request */;
                MPI_Send(&work,         /* message buffer */
                1,              /* one data item */
                MPI_INT,        /* data item is an integer */
                rank,           /* destination process rank */
                WORKTAG,        /* user chosen message tag */
                MPI_COMM_WORLD);/* always use this */
        }



/*
* Receive a result from any compute node and dispatch a new work
* request work requests have been exhausted.
*/
        work = /* get_next_work_request */;
        while (/* valid new work request */) {
                MPI_Recv(&result,       /* message buffer */
                1,              /* one data item */
                MPI_DOUBLE,     /* of type double real */
                MPI_ANY_SOURCE, /* receive from any sender */
                MPI_ANY_TAG,    /* any type of message */
                MPI_COMM_WORLD, /* always use this */
                &status);       /* received message info */
                MPI_Send(&work, 1, MPI_INT, status.MPI_SOURCE,
                WORKTAG, MPI_COMM_WORLD);
                work = /* get_next_work_request */;
        }
/*
* Receive results for outstanding work requests.
*/
        for (rank = 1; rank < ntasks; ++rank) {
                MPI_Recv(&result, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        }
/*
* Tell all the compute nodes to exit.
*/
        for (rank = 1; rank < ntasks; ++rank) {
                MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
        }
}

computenode()
{
        double              result;
        int                 work;
        MPI_Status          status;
        for (;;) {
                MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
/*
```

```
* Check the tag of the received message.
*/
                if (status.MPI_TAG == DIETAG) {
                        return;
                }
                result = /* do the work */;
                MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        }
}
```

Processes are represented by a unique rank (integer) and ranks are numbered 0, 1, 2, ..., N-1. MPI_COMM_WORLD means all the processes in the MPI application. It is called a communicator and it provides all information necessary to do message passing. Portable libraries do more with communicators to provide synchronisation protection that most other systems cannot handle.

A.3 MPI Initialization And Finalization

As with other systems, two functions are provided to initialize and clean up an MPI process:

```
MPI_Init(&argc, &argv);
MPI_Finalize( );
```

A.4 What Is The Current Process? How Many Processes Are There?

Typically, a process in a parallel application needs to know who it is (its rank) and how many other processes exist.

A process finds out its own rank by calling:

```
MPI_Comm_rank( ):
Int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

The total number of processes is returned by `MPI_Comm_size( )`:

```
int nprocs;
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

A.5 Sending Messages

A message is an array of elements of a given data type. MPI supports all the basic data types and allows a more elaborate application to construct new data types at runtime. A message is sent to a specific process and is marked by a tag (integer value) specified by the user. Tags are used to distinguish between different message types a process might send/receive. In the sample code above, the tag is used to distinguish between work and termination messages.

```
MPI_Send(buffer, count, datatype, destination, tag, MPI_COMM_WORLD);
```

A.6 Receiving Messages

A receiving process specifies the tag and the rank of the sending process. `MPI_ANY_TAG` and `MPI_ANY_SOURCE` may be used optionally to receive a message of any tag and from any sending process.

```
MPI_Recv(buffer, maxcount, datatype, source, tag, MPI_COMM_WORLD, &status);
```

Information about the received message is returned in a status variable. The received message tag is `status.MPI_TAG` and the rank of the sending process is `status.MPI_SOURCE`. Another function, not used in the sample code, returns the number of data type elements received. It is used when the number of elements received might be smaller than `maxcount`.

```
MPI_Get_count(&status, datatype, &nelements);
```

A.7 Blocking, Non-Blocking, And Persistent Messages

`MPI_Send` and `MPI_Receive` cause the running program to wait for non-local communication from a network. Most communication networks function at least an order of magnitude slower than local computations. When an MPI process has to wait for non-local communication CPU cycles are lost because the operating system has to block the process, then has to wait for communication, and then resume the process.

An optimal efficiency is usually best achieved by overlapping communication and computation. *Blocking* messaging functions only allow one communication to occur at a time. *Non-blocking* messaging functions allow the application to initiate multiple communication operations, enabling the MPI implementation to proceed simultaneously. *Persistent* non-blocking messaging functions allow a communication state to persist, so that the MPI implementation does not waste time on initializing or terminating a communication.

A.7.1 Blocking Messages

In the following example, the communication implementation executes in a sequential fashion causing each process, `MPI_Recv`, then `MPI_Send`, to block while waiting for its neighbor:

Example

```
while (looping) {
  if (i_have_a_left_neighbor)
    MPI_Recv(inbuf, count, dtype, left, tag, comm, &status);
  if (i_have_a_right_neighbor)
    MPI_Send(outbuf, count, dtype, right, tag, comm);
  do_other_work();
}
```

MPI also has the potential to allow both communications to occur simultaneously, as in the following communication implementation example:

A.7.2 Non-Blocking Messages

Example

```
while (looping) {
  count = 0;
  if (i_have_a_left_neighbor)
    MPI_Irecv(inbuf, count, dtype, left, tag, comm, &req[count++]);
  if (i_have_a_right_neighbor)
    MPI_Isend(outbuf, count, dtype, right, tag, comm, &req[count++]);
  MPI_Waitall(count, req, &statuses);
  do_other_work();
}
```

In the example, `MPI_Waitall` potentially allows both communications to occur simultaneously. However, the process as show is blocked until both communications are complete.

A.7.3 Persistent, Non-Blocking Messages

A more efficient use of the waiting time means to carry out some other work in the meantime that does not depend on that communication. If the same buffers and communication parameters are to be used in each iteration, then a further optimization is to use the MPI persistent mode. The following code instructs MPI to set up the communications once, and communicate similar messages every time:

Example

```
  int count = 0;
  if (i_have_a_left_neighbor)
    MPI_Recv_init(inbuf, count, dtype, left, tag, comm, &req[count++]);
  if (i_have_a_right_neighbor)
    MPI_Send_init(outbuf, count, dtype, right, tag, comm, &req[count++]);
  while (looping) {
   MPI_Startall(count, req);
    do_some_work();
    MPI_Waitall(count, req, &statuses);
   do_rest_of_work();
 }
```

In the example, `MPI_Send_init` and `MPI_Recv_init` perform a persistent communication initialization.

B

Compiler Flag Equivalence

The following table is an overview of some of the compiler flags that are equivalent or almost equivalent.

Cray	Intel	GCC	Explanation
default	default	`-O3 -ffast-math`	Produce high level of optimization
`-Oomp` (default)	`-openmp`	`-fopenmp`	Activate OpenMP directives and pragmas in the code
`-h byteswapio`	`-convert big_endian`	`-fconvert=swap`	Read and write Fortran unformatted data files as big-endian
`-f fixed`	`-fixed`	`-ffixed-form`	Process Fortran source using fixed form specifications.
`-f free`	`-free`	`-ffree-form`	Process Fortran source using free form specifications.
`-V`	`--version`	`--version`	Dump version.
`-h zero`	N/A	`-finit-local-zero`	Zero fill all uninitialized variables.
`-e m`			Creates `.mod` files to hold Fortran90 module information for future compiles.
`-j` <*dir_name*>			Specifies the directory <*dir_name*> to which `.mod` files are written when the `-e m` option is specified